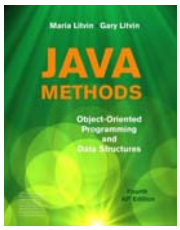


Fourth AP Edition

Java Methods

Object-Oriented Programming and Data Structures

Maria Litvin
Phillips Academy, Andover, Massachusetts

Gary Litvin
Skylight Software, Inc.

Skylight Publishing
Andover, Massachusetts

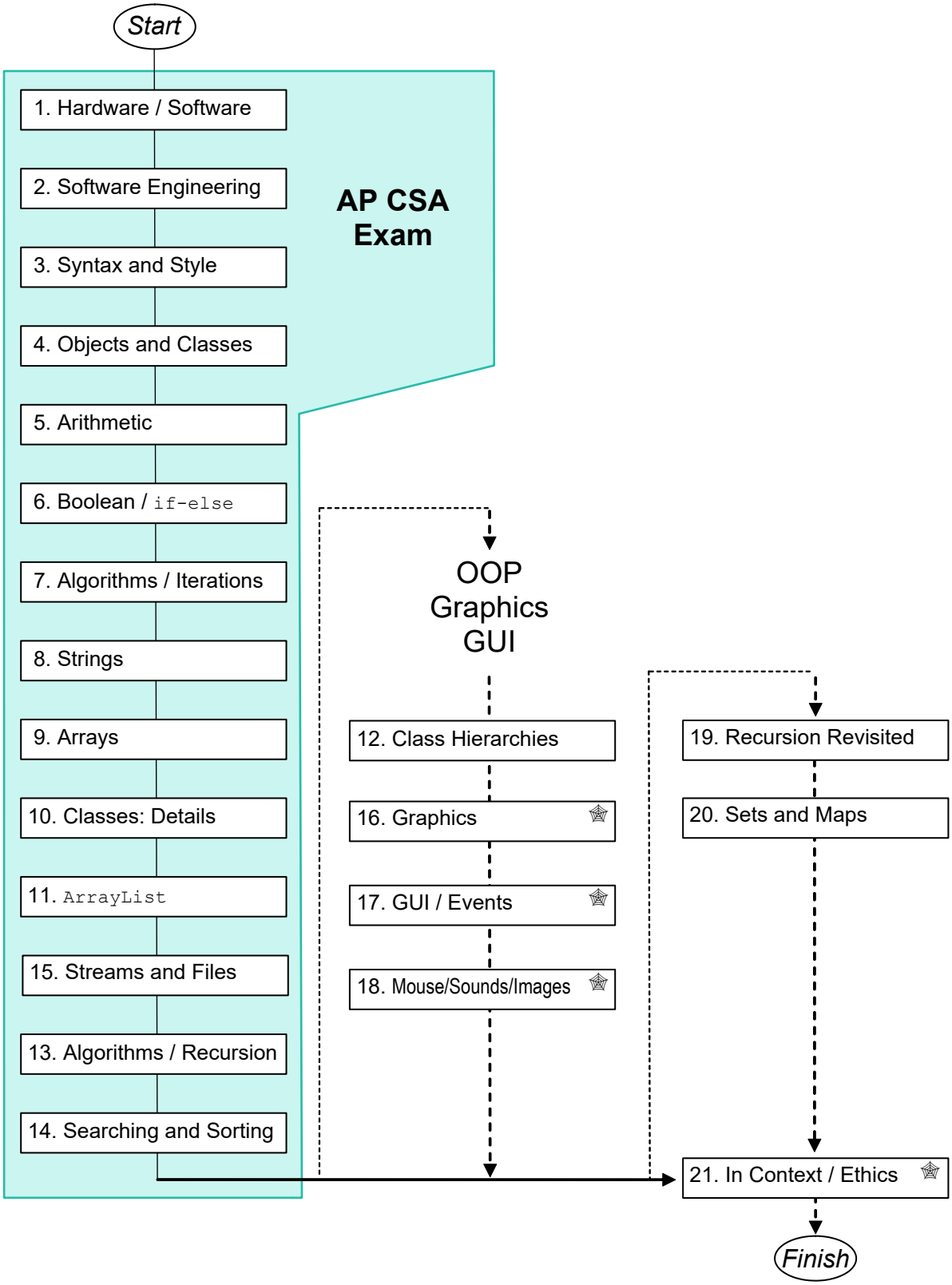
Skylight Publishing
14 Lincoln Street
Andover, MA 01810

web: http://www.skylit.com
e-mail: sales@skylit.com
support@skylit.com

Library of Congress Control Number: 2021944689
ISBN 978-0-9972528-2-8

[*] AP and Advanced Placement are registered trademarks of The College Board, which was not involved in the production of and does not endorse this book.

Brief Contents


Start
1. Hardware / Software
2. Software Engineering
3. Syntax and Style
4. Objects and Classes
5. Arithmetic
6. Boolean / if-else
7. Algorithms / Iterations
8. Strings
9. Arrays
10. Classes: Details
11. ArrayList
15. Streams and Files
13. Algorithms / Recursion
14. Searching and Sorting
AP CSA Exam
OOP
Graphics
GUI
12. Class Hierarchies
16. Graphics
17. GUI / Events
18. Mouse/Sounds/Images
19. Recursion Revisited
20. Sets and Maps
21. In Context / Ethics
Finish

Contents

Preface

This book offers a thorough introduction to the concepts and practices of object-oriented programming in Java. It also introduces some of the common data structures and related algorithms: one- and two-dimensional arrays, `ArrayList`, `Sets` and `Maps`, and their implementations in the Java Collections Framework.

Chapters 1-14 follow the syllabus of the AP Computer Science in Java course (AP CS "A"). They will prepare you well for the AP 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. Chapter 19 revisits recursion at a deeper level. Chapter 20 introduces the concepts of look-up tables, sets, and maps, and their implementation in the Java Collections Framework. 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*, second and third AP Editions (2011 and 2015). The AB-level AP CS exam was discontinued by the College Board in 2009. Teachers who continue teaching advanced data structures and students who want to learn this material on their own can find complete data structures chapters in the *Java Methods* Third AP edition e-book. In this edition we have added a lab and exercises to the `ArrayList` chapter.

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* (automatically calling the correct methods 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 code.

Our labs and case studies aim to demonstrate the most appropriate uses of the programming techniques and data structures we cover. OOP is 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 A exam. While we mostly stay within the Java AP subset defined by the College Board for AP CS exams in Java, we also want to give you a solid conceptual foundation and introduce sound software design and coding practices. If you are preparing for the AP exam, you'll need to be familiar with the College Board's Course and Exam 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 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 code 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!

❖ ❖ ❖

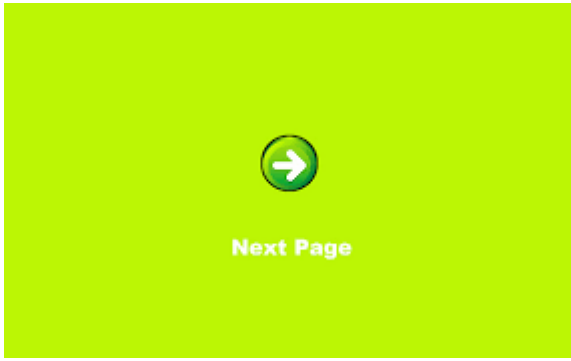
Since our first book came out in 1998, many of our colleagues, too many to name, have become good friends. We are grateful to them for their loyal support, encouragement, and the many things they have taught us over the years.

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

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

❖ ❖ ❖

The cover image of a star with lens flare and bokeh effect was generated using 3D software.


Next Page

How to Use This Book

This edition's companion web site —

`http://www.skylit.com/javamethods4`

— is an integral part of this book. It contains five 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, and 21 and the appendices on the web either because they rely on many web links or because the material they cover is 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.

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/javamethods4` 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 some coding, 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 can be skipped if you understand recursion really well (or are not at all interested in it). Chapter 21, 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.

ch 001

Hardware, Software, and the Internet

www.skylit.com/javamethods4/JM-Chapter01-www.pdf

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.

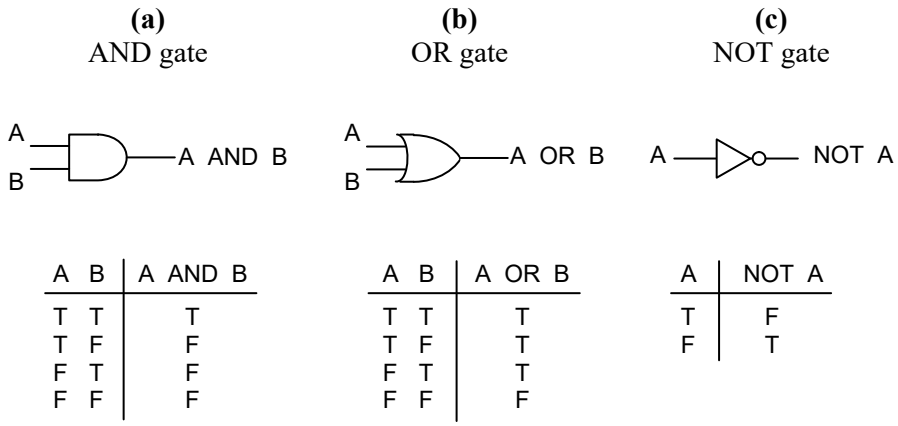


Figure 1-1. AND, OR, and NOT gates

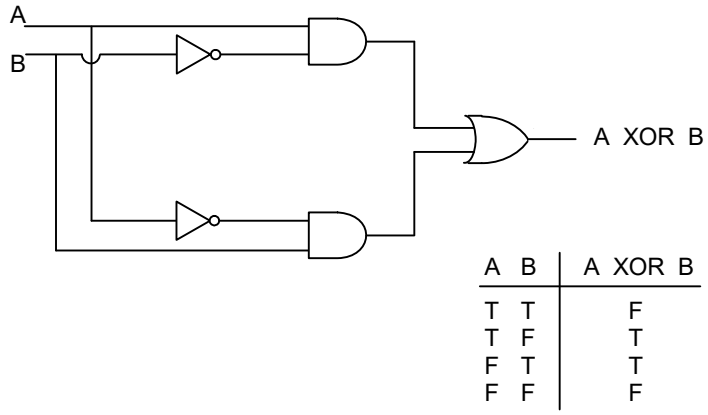


Figure 1-2. XOR circuit made of AND, OR, and NOT gates

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 2021 had 8, 12, or 16 "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 or touch screen, 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

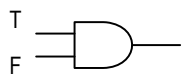
1.2 - 1.7

These sections are online at `http://www.skylit.com/javamethods4`.

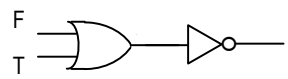
Exercises

Sections 1.1-1.4

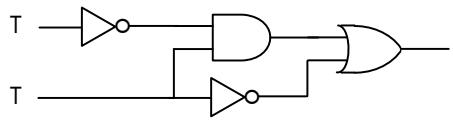
1. Mark T (true) or F (false) the output of each of the following circuits with the given inputs.



(a)



(b)



(c)■

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 apps 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:

	Binary	Decimal	Hex	
	00001000	8	08	
	00011100	28	1C	
(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:

Aγ	Bα	Cβ
Bβ	Cγ	Aα
Cα	Aβ	Bγ

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:

Decimal	Base 3
0	0
1	1
2	2
3	10
4	11
5	12
6	20
7	21
8	22
9	100
...	...

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.

Section 1.6

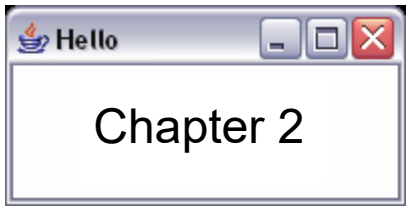
19. (MC) What does TCP stand for?

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.



An Introduction to Software Engineering

2.1 Prologue

One of the first computers, ENIAC,[eniac] 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.

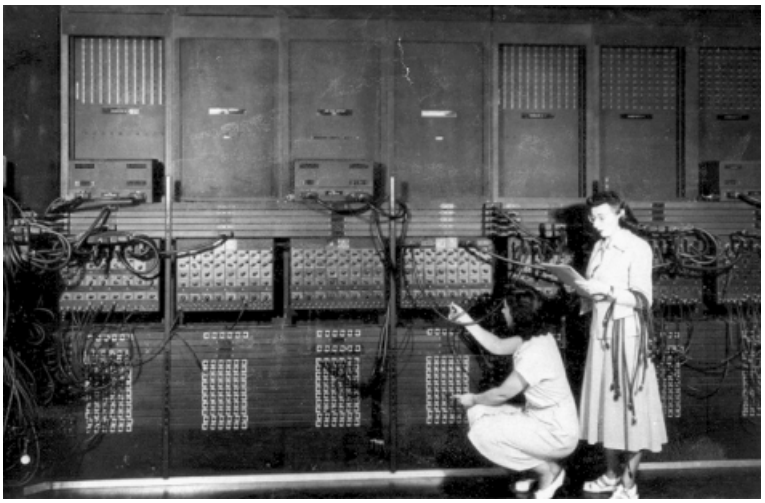


Figure 2-1. Two technicians wiring the right side of ENIAC

(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[punchcard] 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" or "app developers." 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 ("apps") that run on a personal 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:

```
index   dw      0           ; "Define word" -- reserve 2 bytes
                            ;  for an integer and call it "index".
        ...
        mov     si,index    ; Move the value of index into
                            ;   the SI register.
        ...
```

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.[languagehistory] 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).

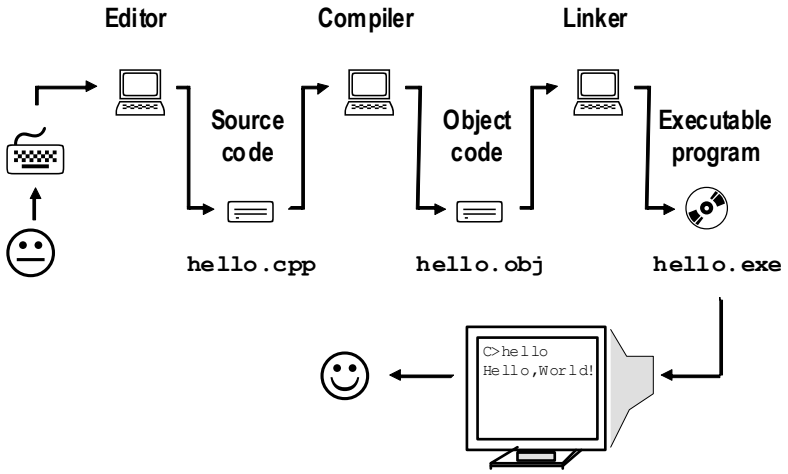


Figure 2-2. Software development cycle for a compiled program: edit-compile-link-run

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.

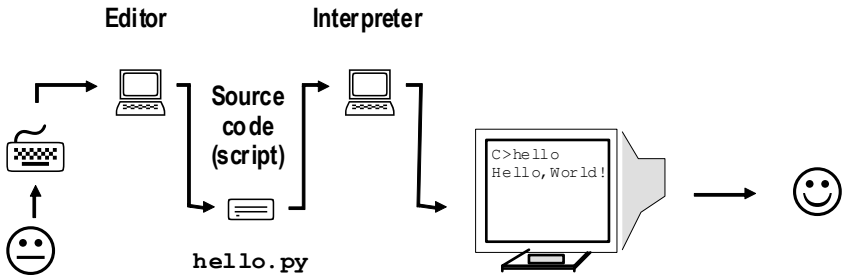


Figure 2-3. Interpreted program

❖ ❖ ❖

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.

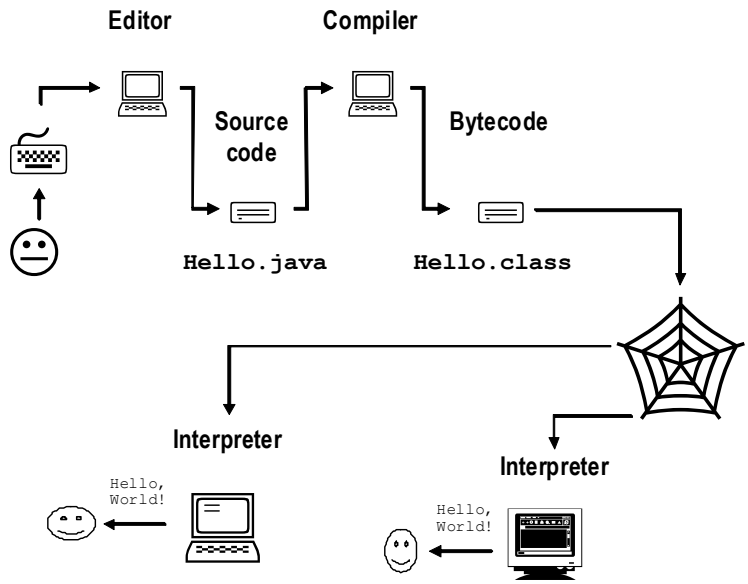


Figure 2-4. Java software development and distribution through the Internet

To speed up the loading of applets, a new software technology was developed, called *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).

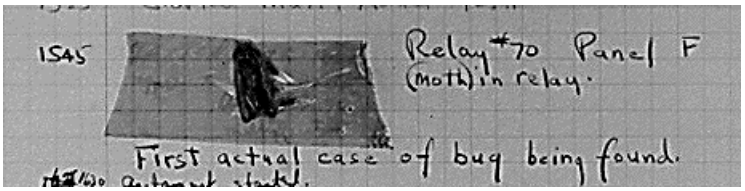


Figure 2-5. The term "bug" was popularized by Grace Hopper,[hopper] a legendary computer pioneer, who was the first to come up with the idea of a compiler and who created COBOL. One of Hopper's favorite stories was about a moth that was found trapped between the points of a relay, which caused a malfunction 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 opensource 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. The `ArrayList` class in the `java.util` package from the standard Java library is a reusable class for handling lists; it is explained in Chapter 11. The `HashSet` and `HashMap` classes from the same package are described in Chapter 20. Java coders 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*, 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/javamethods4`, has instructions for installing and using *Eclipse* and *JGrasp*.

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\jdk-16\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`:

```
/**
 * 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:

```
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\jdk-16\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.

```
/**
 * 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`.

```
/**
 * This program prompts the user to enter his or her
 * first name and last name and displays a greeting message.
 * Author: Maria Litvin
 */

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,[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[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.

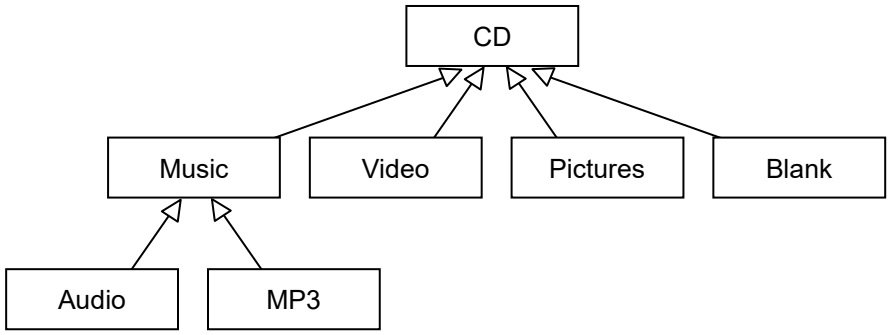


Figure 2-6. A hierarchy of classes that represent compact disks with different content

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 and Exam 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 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.

```
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/javamethods4`). 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.

```
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

```
        // Add panel to window:
        Container c = window.getContentPane();
        c.add(panel);

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

        // Set the initial position of the message:
        panel.xPos = panel.getWidth();
        panel.yPos = panel.getHeight() / 2;

        // Create a Timer object that fires every 30 milliseconds;
        // attach it to panel so that panel "listens to" and
        // processes the timer events; start the timer:
        Timer clock = new Timer(30, panel);
        clock.start();
    }
}
```

Figure 2-9. JM\Ch02\HelloGui\Banner.java

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: ✓

```
   xxxxx
  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/`) or, if you have the JDK documentation installed on your computer, open the file <*JDK base folder*>`/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:

```
for (int count = 1; 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

```
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:

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

Use

```
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:

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

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

```
msgID = -msgID;
```

Don't forget to call `repaint`.

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

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

Then use a *conditional statement* to display the appropriate message:

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

⦚

```
/**
 * Chapter 3
 */
```

Java Syntax and Style

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.

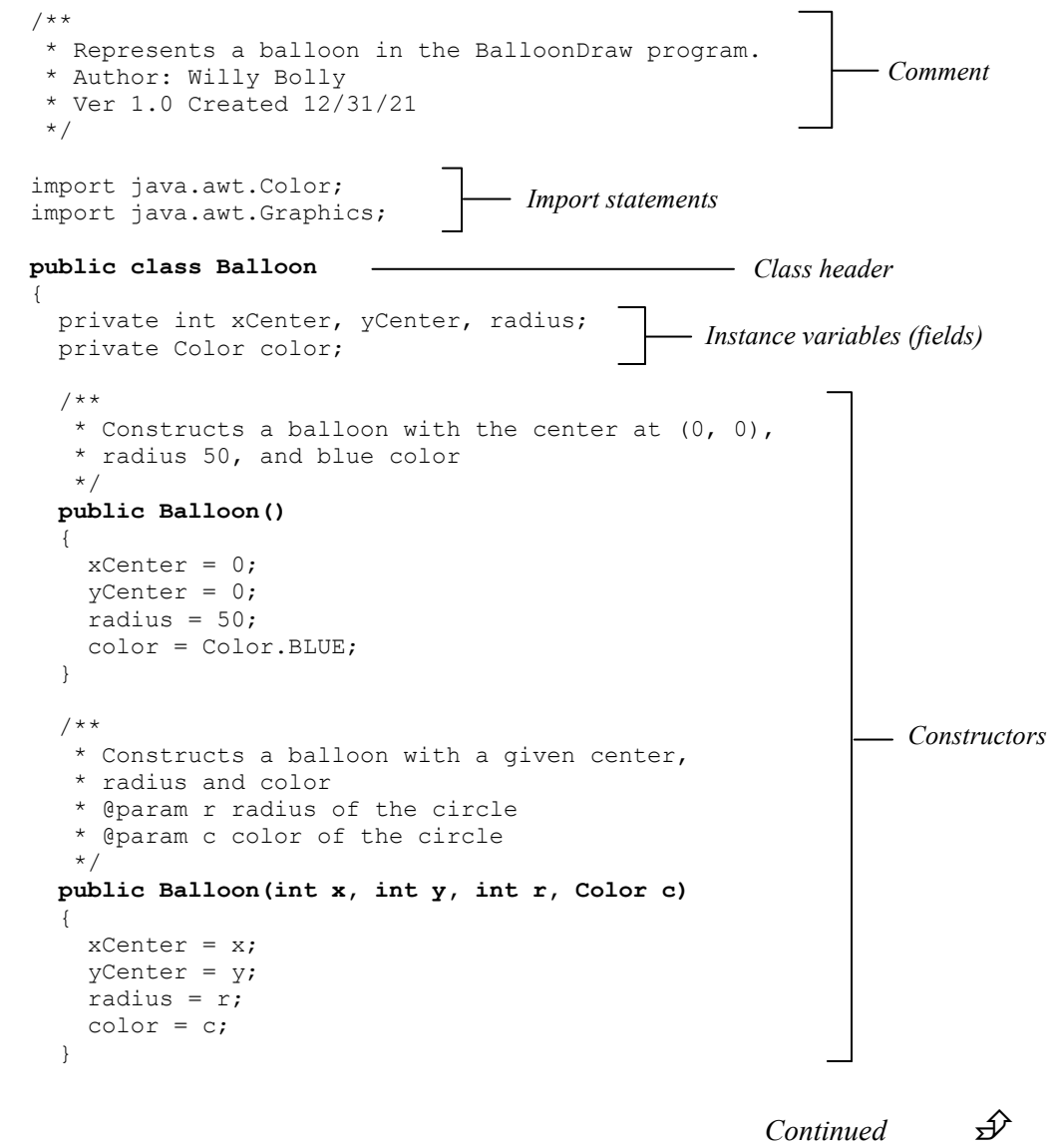
In this chapter we will take a more detailed look at Java's syntax and style. We will discuss the following topics:

- 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).



Continued

```
  /**
   * Returns the distance from a given point (x, y) to
   * the center of this balloon.
   */
  public double distance(int x, int y)
  {
    double dx = x - xCenter;
    double dy = y - yCenter;
    return Math.sqrt(dx*dx + dy*dy);
  }

  /**
   * Moves the center of this balloon to (x, y)
   */
  public void move(int x, int y)
  {
    xCenter = x;
    yCenter = y;
  }

  /**
   * Returns true if a given point (x, y) is inside this
   * balloon; otherwise returns false
   */
  public boolean isInside(int x, int y)
  {
    return distance(x, y) < radius * 0.9;
  }

  /**
   * Draws a solid circle if makeItFilled is true and
   * outline only if makeItFilled is false
   * @param g graphics context
   * @param makeItFilled draws a solid circle if true
   */
  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 >
}
```

Methods (may include a `main` *method)*

Figure 3-1. A schematic view of a Java class's source code

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:

```
/*  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:

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

or

```
// 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:

```
/**
 * <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.
 *
 * @author Al Jibris
 */
```

Or:

```
  /**
   * 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[javadoc] 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.

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

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.

```
/**
 * 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:

```
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

```
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.

```
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:

```
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 <u>something</u> 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

```
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:

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

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

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

You <u>think</u> 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 <u>before</u> 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.

```
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:

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

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

```
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 >

public class MovingDisk extands 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

```
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:

```
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?

Sections 3.6-3.8

9. Restore line spacing and proper indentation in the following code: ✓

```
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

```
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

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

with

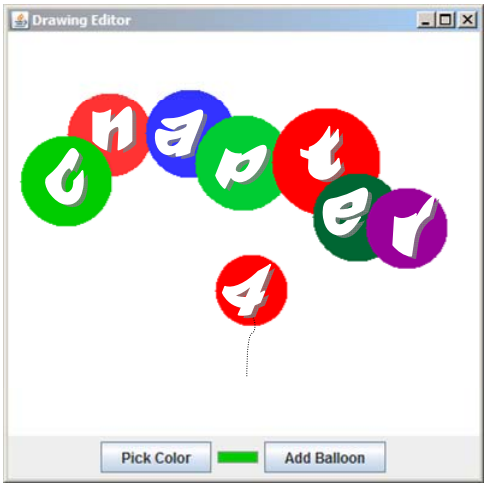
```
public void HelloGui()   // Constructor
```

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.



Objects and Classes

4.1 Prologue

Non-technical people sometimes envision a computer programmer's job as sitting at a computer and writing lines of code in a cryptic programming language. Perhaps this is how it might appear to a casual observer. This is not so. The work of a programmer involves not only lines and pages of computer code, but also an orderly structure that matches the task. Even in the earliest days of the computer era, when programs were written directly in machine code, a programmer first developed a more or less abstract view of the task at hand. The overall task was split into meaningful subtasks; then a set of procedures was designed that accomplished specific subtasks; each procedure, in turn, was divided into meaningful smaller segments.

A software engineer has to be able to see the big picture or to zoom in on more intricate details as necessary. Over the years, different software development methodologies have evolved to facilitate this process and to help programmers better communicate with each other. The currently popular methodology is *Object-Oriented Programming* (*OOP*). OOP is considered more suitable than previous methodologies for:

- Team work
- Reuse of software components
- GUI development
- Program maintenance

In OOP, a programmer envisions a software application as a virtual world of interacting objects.

This world is highly structured. To think of objects in a program simply as fish in an ocean would be naive. If we take the ocean as a metaphor, consider that its objects include islands, boats, the sails on the boats, the ropes that control the sails, the people on board, the fish in the water, and even the horizon, an object that does not physically exist! There are objects within objects within objects, and the whole ocean is an object, too.

The following questions immediately come to mind:

- Who describes all the different types of objects in a program? When and how?
- How does an object represent and store information?
- When and how are objects created?
- How can an object communicate with other objects?
- How can objects accomplish useful tasks?

We'll start answering these questions in this chapter and continue through the rest of the book. Our objective in this chapter is to learn the following terms and concepts: object, class, instance variable or field, constructor, method, public vs. private, encapsulation and information hiding, inheritance, IS-A and HAS-A relationships.

4.2 *Case Study:* A Drawing Program

The window in Figure 4-1 comes from the *BalloonDraw* program. This is a very simple drawing program, in which the user can arrange several "balloons" (colored disks) on "canvas." The user can create balloons of different colors and move and stretch them, using the mouse or the cursor keys on the keyboard.

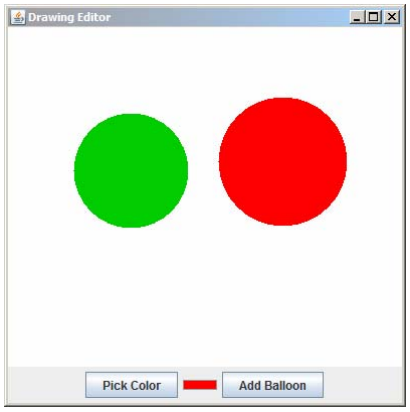


Figure 4-1. A window from the *BalloonDraw* program

Play with this program by clicking on the `ballondraw.jar` file in the `JM\Ch04\BalloonDraw` folder. "`jar`" stands for "java archive," which is similar to `.zip` format. It allows you to run a Java program with a simple click, without creating any projects in an IDE.

If you "grab" a balloon (click and hold down the mouse button on it), you can move the balloon, and if you "grab" a balloon near the border, you can stretch it. Also cursor keys move the currently selected balloon, while cursor keys with the `Ctrl` key pressed down stretch the balloon.

❖ ❖ ❖

How does one write a program like this in OOP style? A good place to start is to decide what types of objects are needed.

An *object* in a running program is an entity that represents an object or a concept from the real world.

Some of the objects in a program may model real-world objects, such as a balloon. There are also GUI objects that are visible or audible: buttons, sliders, menus, images, audio clips, and so on. Other objects may represent abstract concepts, such as a coordinate system.

Each object in a running program has a set of attributes and behaviors. The object's attributes hold specific values; some of these values can change while the program is running.

For example, a balloon object has such attributes as location, size, and color, and such behaviors as moving and stretching. A balloon also "knows" how to draw itself.

A program often employs several objects of the same type. Such objects are said to belong to the same *class*. We also say that an object is an *instance* of its class.

Objects of the same class have the same set of attributes and behaviors; they differ only in the values of some of their attributes.

In Figure 4-1, for example, you can see two balloons, that is, two objects of the `Balloon` class. They have different locations, sizes and colors.

The rest of the visible objects in Figure 4-1 are GUI components. There is a title bar, and a simple control panel at the bottom with two buttons and a color label that shows the currently selected color.

Finally, we see an object that represents the whole window in which the program is running.

It is important to distinguish objects from their visual representations in a GUI program. In a well-designed program, visual representations of objects are separate from abstract models of their behavior. For example, a balloon is displayed as a colored disk. But a `Balloon` object will remain a `Balloon` even if a program does not display it at all and just prints out some information about its location radius, and color. Similarly, GUI components (buttons, menus, etc.) are rather abstract entities. Their appearance can be modified to match the native *look and feel* of the operating system. The labels on menus and buttons, text messages, and program help text might be stored separately, too, so that they can be easily translated into different languages.

❖ ❖ ❖

Sometimes it is hard to decide whether two objects have serious structural differences and different behaviors and should belong to different classes or if they differ only in the values of some of their attributes. A color is just an attribute, so balloons of different colors are objects of the same class. But if we wanted to have balloons of different shapes, they might belong to different classes. In OOP languages, it is possible to compromise: an object can belong to a *subclass* of a given class and can "inherit" some of the code (data attributes and behaviors) from its parent class. For example, `RoundBalloon` and `SquareBalloon` may be defined as different kinds of `Balloon`. `RoundBalloon` and `SquareBalloon` will be *subclasses* of the class `Balloon` and `Balloon` will be their *superclass*. More on this later (Section 4.5).

❖ ❖ ❖

In OOP, a lot of the emphasis shifts from software development to software design. The design methodology, object-oriented design (OOD), parallels the programming methodology (OOP). A good design makes implementation easier. But object-oriented design itself is not easy, and a bad design may derail a project.

The design phase starts with a preliminary discussion of the rough design. One of the informal techniques the designers might use is *CRC cards*. CRC stands for "Class, Responsibilities, Collaborators." A CRC card is simply an index card that describes a class of objects, including the class's name, the main "responsibilities" of this type of object in the program, and its "collaborators," that is, other classes that it depends on (Figure 4-2).

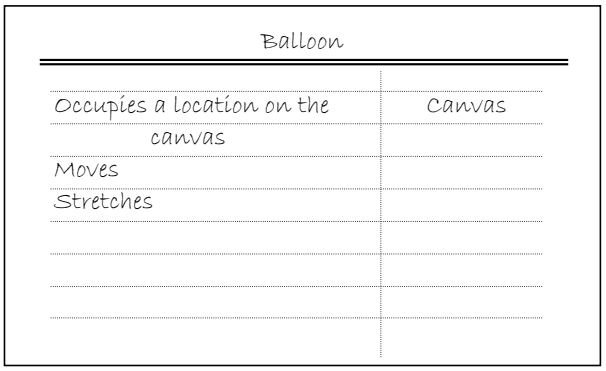


Figure 4-2. A CRC card for the class `Balloon`

At this initial stage, software designers do not have to nail down all the details. The responsibilities of each type of object are described in general terms. The need for additional types of objects may become apparent later in the process.

4.3 Classes

In Java, a programmer must describe in the program's source code the different types of objects used in the program.

> **A *class* is a piece of the program's source code that describes a particular type of objects. A formal description of a class is called a *class definition* or a *class declaration*. Java programmers write *class definitions*.**

Informally, we say that programmers write classes and that the source code of a Java program consists of one or several classes.

Figure 4-3 summarizes the concepts of a *class* and an *object* and the differences between them.

Class:	Object:
A piece of the program's source code	An entity in a running program
Written by a programmer	Created when the program is running
Specifies the structure (the number and types of attributes) for the objects of this class, the same for all of its objects	Holds specific values of attributes; some of these values can change while the program is running
Specifies the possible behaviors of its objects — the same for all of its objects	Behaves appropriately when called upon
A Java program's source code consists of several classes	A running program can create any number of objects (instances) of a class
Like a blueprint for building cars of a particular model	Like a car of a particular model that you can drive

Figure 4-3. ***Class* vs. *object***

A class is sometimes compared to a cookie cutter: all objects of the class have the same configuration but might have different flavors. A more apt comparison for a class, perhaps, would be a blueprint for making a specific model of a car. Like objects of the same class, cars of the same model have identically configured parts, but they may be of different colors, and when a car is running, the number of people in it or the amount of gas in the tank may be different from another car of the same model.

❖ ❖ ❖

The source code for a class is usually stored in a separate file.

For example, the `Balloon` class is stored in the file `Balloon.java`.

All in all, the *BalloonDraw* program includes four classes: `Balloon`, `ControlPanel`, `DrawingPanel`, and `BalloonDraw`. The latter has a `main` method that creates a `DrawingPanel` and a `ControlPanel` objects, adds them to the window, then opens the window on the screen.

In Java, the name of the source file must be the same as the name of the class, with the extension `.java`.

For example, a class `Flower` must be stored in a file named `Flower.java`.

In Java, all names, including the names of classes, are case-sensitive. By convention, the names of classes (and the names of their source files) always start with a capital letter.

A class describes three aspects that every instance (object) of this class has: (1) the data elements (attributes) of an object of this class, (2) the ways in which an object of this class can be created, and (3) what this type of object can do.

An object's data elements are called *instance variables* or *fields*. Procedures for creating an object are called *constructors*. Behaviors of an object of a class are called *methods*.

The class describes all these features in a very formal and precise manner. We have already seen the fields, constructors and some of the methods of the `Balloon` class in Figure 3-1 on pages 45-46.

`Balloon` also uses the Java library class `Color`, so the code for the `Balloon` class has an `import` statement for it:

```
import java.awt.Color;
```

This statement tells the compiler that it can find the `Color` class in the `java.awt` package[awt] of the Java's standard library.

Without `import` statements, you would have to use the *fully-qualified* names of library classes everywhere in your code, as in

```
private java.awt.Color color;
```

instead of

```
private Color color;
```

This would clutter your code.

Not every class has fields, constructors, and methods explicitly defined: some of these features might be implicit or absent.

Also some classes don't have objects of their type ever created in the program. For example, the `HelloWorld` program (Chapter 2, page 24) is described by the class `HelloWorld`, which has only one method, `main`. This program does not create any objects of this class. The `main` method is declared `static`, which means it belongs to the `HelloWorld` class as a whole, not to any particular object of that class. The `main` method is called automatically when the program is started.

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.

The class header is followed by the class definition body within braces.

❖ ❖ ❖

When a Java program starts, control is passed to the `main` method.

In our *BalloonDraw* program, the `main` method resides in the `BalloonDraw` class. The `main` method creates one object of this class, which represents the window in which the program is running (Figure 4-4). A Java program can have only one public `main` method.

```
/**
 * This drawing program helps create pictures with several
 * balloons (colored disks or other shapes)
 * Authors: Maria Litvin and Gary Litvin
 */

import java.awt.Container;
import java.awt.BorderLayout;
import javax.swing.JFrame;

public class BalloonDraw extends JFrame
{
  public BalloonDraw()  // constructor
  {
    super("Drawing Editor");

    DrawingPanel canvas = new DrawingPanel();
    ControlPanel controls = new ControlPanel(canvas);
    Container c = getContentPane();
    c.add(canvas, BorderLayout.CENTER);
    c.add(controls, BorderLayout.SOUTH);
  }

  public static void main(String[] args)
  {
    BalloonDraw window = new BalloonDraw();
    window.setBounds(100, 100, 400, 400);
    window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    window.setVisible(true);
  }
}
```

Figure 4-4. JM\Ch04\BalloonDraw\BalloonDraw.java

We could place `main` in a separate class. The name of the class, `BalloonDraw`, is chosen by its author; it could be called `DrawTest` or `BalloonTest` or `BalloonRunner` or `FirstProject` instead. `BalloonDraw`'s `main` method displays the window on the screen by calling `setVisible(true)` and waits for commands and actions from the user.

4.4 Fields, Constructors, and Methods

As we said earlier, the definition of a class describes all the *instance variables* of objects of this class. Instance variables are also called *data fields* or simply *fields*, from the analogy with fields in a form that can be filled in with different values.

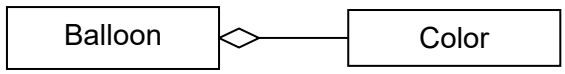
Each field has a name, given by a programmer, and a type. A `Balloon` object, for example, has four fields:

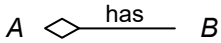
```
private int xCenter, yCenter, radius;
private Color color;
```

The type of the fields `xCenter`, `yCenter`, `radius` is `int` (integer); the type of the field `color` is `Color`. Notice that a class name (`Balloon`, `Color`) starts with a capital letter, while a name of a field (`color`), starts with a low case letter.

Think of an object's instance variables as its private "memory." (This is only a metaphor, of course: in reality, when a program is running, its objects are stored in chunks of RAM.) An object's "memory" may include other objects, and also numbers and text characters. (In Java, numbers and text characters are usually not represented by objects; for them Java provides special data types, `int`, `double`, `char`, etc., called *primitive data types*.)

The `xCenter`, `yCenter`, and `radius` fields define the current location and size of the balloon. `color` is another field; its type is `Color`, that is, color is an object of the class `Color`. An OOP designer would say that a `Balloon` *HAS-A* (has a) `Color`. We can show these relationships between classes in a UML (Unified Modeling Language) diagram:





❖ ❖ ❖

An object is created with the `new` operator, which invokes (calls) one of the *constructors* defined in the object's class. For example:

```
activeBalloon = new Balloon(w/2, h/2, 100, Color.RED);
```

A constructor is a procedure, usually quite short, that is used primarily to initialize the values of the instance variables for the object being created.

For example:

```
/**
 * 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;
}
```

A constructor must always have the same name as its class.

A constructor can accept one or more parameters or no parameters at all. The latter is called a "no-args" constructor (parameters are often called "arguments," as in math functions such as $f(x)$, or "args" for short). The above example is a no-args constructor.

A class may have several constructors that differ in the number and/or types of parameters that they accept.

For example, in addition to the above no-args constructor, the `Balloon` class defines a constructor that takes four parameters:

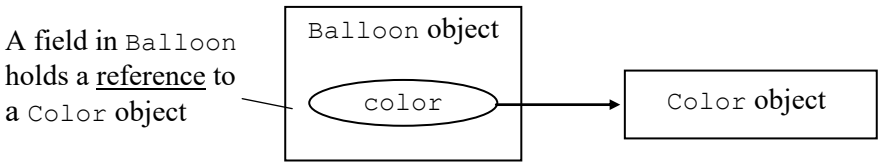
```
/**
 * Constructs a balloon with a given center, radius and color
 * @param x x-coordinate of the center
 * @param y y-coordinate of the center
 * @param r radius of the balloon
 * @param c color of the balloon
 */
public Balloon(int x, int y, int r, Color c)
{
  xCenter = x;
  yCenter = y;
  radius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

The number, types, and order of parameters passed to the `new` operator when an object is created must match the number, types, and order of parameters accepted by one of the class's constructors.

If we wanted to create a green balloon of radius 30 at location (50, 100) we would write:

```
Balloon b = new Balloon(50, 100, 30, Color.GREEN);
```

When an object is created, a chunk of RAM is allocated to hold it, and `new` returns a *reference* to that location, which is basically the object's address in RAM. The reference may be stored in a variable:



Several variables may hold references to the same object. If you compare an object to a web page, a reference is like the page's URL (web address). Many users may have that URL saved somewhere in their "favorites" or in their own web pages. A Java interpreter is equipped with a mechanism that keeps track of all the references to a particular object that are currently in existence in a running program. Gradually the references to an object may all cease to exist. Then the object is unreachable, because no other object in the program can find it or knows it exists. The Java interpreter finds and destroys such useless objects and frees the memory they occupied. This mechanism is called *garbage collection*.

Each Java class has at least one constructor. If you don't define any, the compiler supplies a default *no-args constructor* that initializes all the instance variables to default values (zeroes for numbers, `null` for objects, `false` for `boolean` fields).

❖ ❖ ❖

Metaphorically speaking, an object can "send messages" to other objects. In Java, to "send a message" means to call another object's *method*. A method is a function or a procedure that performs a certain task or computation. All of an object's methods are described in its class definition, and all the objects of a given class have exactly the same set of methods. The methods define what an object can do, what kind of "messages" it "understands" and can respond to.

Does a method belong to an object or to a class? The terminology here is not very precise. When we focus on a running program, we say that an object has a method, meaning that we can call that method for that particular object. When we focus on the program's source code, we say that a class has a method, meaning that a programmer included code for the method in the class definition.

Each method has a name, given by the programmer. Like a constructor, a method may accept one or more parameters. For example, once we have created an object `bigBlue` of the `Balloon` class, we can call its `move` method:

```
bigBlue.move(40, 60);
```

This statement moves `bigBlue`'s center to the point (40, 60) on the canvas. The compiler understands this statement because we have defined the `move` method in the `Balloon` class.

Parameters passed to a method must match the number, types, and order of parameters that the method expects.

Empty parentheses in a method's header indicate that this method does not take any parameters. Such a method is called with empty parentheses. For example, we can call `bigBlue`'s `getRadius` method:

```
int r = bigBlue.getRadius();
```

A method may return a value to the caller. The method's *header* specifies whether the method returns a value or not, and if it does, of what type. For example, `Balloon`'s `getRadius` method returns this balloon's radius, an `int` (integer) value:

```
/**
 * Returns the radius of this balloon
 */
public int getRadius()
{
  return radius;
}
```

The keyword `void` in a method's header indicates that the method does not return any value.

For example, `Balloon`'s `move` and `draw` methods are declared `void`. The `main` method is `void`, too.

A method can call other methods of the same object or of a different object. For example, `Balloon`'s `isOnBorder` method calls the same balloon's `distance` method:

```
public boolean isOnBorder(int x, int y)
{
  double d = distance(x, y);
  return d >= 0.9 * radius && d <= 1.1 * radius;
}
```

`Balloon`'s `draw` method calls `Graphics`'s methods `fillOval` or `drawOval`:

```
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);
}
```

The `Balloon` class provides a well-defined functionality through its constructors and public methods. The user of the `Balloon` class (possibly a different programmer) does not need to know all the details of how the class `Balloon` works, only how to construct its objects and what they can do. In fact, all the instance variables in `Balloon` are declared `private` so that programmers writing classes that use `Balloon` cannot refer to them directly. Some of class's methods can be declared `private`, too. This technique is called *encapsulation* and *information hiding*.

There are two advantages to such an arrangement:

- `Balloon`'s programmer can change the structure of the fields in the `Balloon` class, and the rest of the project won't be affected, as long as `Balloon`'s constructors and public methods have the same specifications and work as before;
- `Balloon`'s programmer can document the `Balloon` class for the other team members (and other programmers who want to use this class) by describing all its constructors and public methods; there is no need to document the implementation details.

It is easier to maintain, document, and reuse an encapsulated class.

After a class is written, it is a good idea to test it in isolation from other classes. We can create a small program that tests some of `Balloon`'s features. For example:

```
import java.awt.Color;

public class TestBalloon
{
  public static void main(String[] args)
  {
    // Create a Balloon called greenie, centered at x = 50, y = 100
    // with radius 30 and color Color.GREEN:

          < statement not shown >            ;

    System.out.println("x = " + greenie.getX());
    System.out.println("y = " + greenie.getY());
    System.out.println("radius = " + greenie.getRadius());

    // Call greenie's move method to move its center to (60, 110):

          < statement not shown >            ;

    System.out.println("x = " + greenie.getX());
    System.out.println("y = " + greenie.getY());

    // Call greenie's isOnBorder method to see if (81, 131)
    // is on its border:

    boolean onBorder =        < expression not shown >        ;

    System.out.println("(81, 131) is on the border: " + onBorder);
  }
}
```

Type in the above code, filling in the blanks, and save it in the `TestBalloon.java` file. Create a project that includes the `TestBalloon.java` and `Balloon.java` files and test your program. Explain the output. Now try to call `move` with the parameters 50, 100. Explain the output.

4.5 Inheritance

Suppose we want to be able to "inflate" a balloon by a certain percentage. It would be easy to add a method `inflate` to the `Balloon` class. But there are several reasons why adding methods to an existing class may be not feasible or desirable.

First, you may not have access to the source code of the class. It may be a library class or a class that came to you from someone else without its source code. For example, if you only had `balloondraw.jar` but no source code for the classes, you could still run the *BalloonDraw* program, but you couldn't change `Balloon.java`. Second, your boss may not allow you to change a working and tested class. You may have access to its source, but it may be "read-only." A large organization cannot allow every programmer to change every class at will. Once a class is written and tested, it may be off-limits until the next release. Third, your class may already be in use in other projects. If you change it now, you will have different versions floating around and it may become confusing. Fourth, not all projects need additional methods. If you keep adding methods for every contingency, your class will eventually become too large and inconvenient to use.

The proper solution to this dilemma is to *derive* a new class from an existing class.

In OOP, a programmer can create a new class by extending an existing class. The new class can add new methods or redefine some of the existing ones. New fields can be added, too. This concept is called *inheritance*.

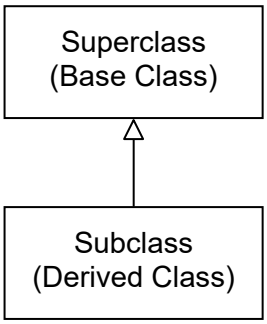
Inheritance is one of the fundamental OOP concepts, and all OOP languages support it.

Java uses the keyword `extends` to indicate that a class extends another class.

For example,

```
public class InflatableBalloon extends Balloon
{
   ...
}
```

If class *D* extends class *B*, then *B* is called a *superclass* (or a *base class*) and *D* is called a *subclass* (or a *derived class*). The relationship of inheritance is usually indicated in UML diagrams by an arrow with a triangular head from a subclass to its superclass:

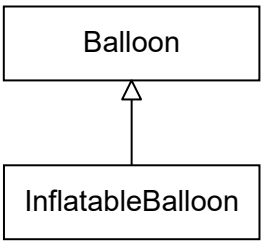


In Java you can extend a class without having its source code.

A subclass *inherits* all the methods and fields of its superclass. Constructors are not inherited; a subclass can provide its own if it needs a constructor.

In Java every class extends the library class `Object` by default. `Object` supplies a few common methods, including `toString` and `getClass`.

In our example, we create a new class `InflatableBalloon`, which extends `Balloon`:



`InflatableBalloon` can be a very short class:

```
import java.awt.Color;

/**
 * Represents an inflatable balloon
 */
public class InflatableBalloon extends Balloon
{
       < Constructors may be needed, not shown >

  public void inflate(int percentage)
  {
       < code not shown >
  }
}
```

We can provide constructors to this class or we can just let the default no-args constructor do the job. It will call automatically `Balloon`'s no-args constructor. `InflatableBalloon` adds one method, `inflate`.

Notice the following paradox. Our `InflatableBalloon` class inherits all the fields from `Balloon`. So an `InflatableBalloon` has a `radius` field. However, this and other fields are declared `private` in `Balloon`. This means that the programmer who wrote the `InflatableBalloon` class did not have direct access to them (even if he is the same programmer who wrote `Balloon`!). Recall that the `Balloon` class is fully encapsulated and all its fields are private. So the statement

```
radius = (int)Math.round(radius * (1 + 0.01*percentage));
```

won't work in `InflatableBalloon`. Try it and see for yourself. What do we do? `Balloon`'s subclasses and, in fact, any other classes that use `Balloon` might need access to the values stored in its private fields.

To resolve the issue, the `Balloon` class provides public methods that simply return the values of its private fields: `getX`, `getY`, `getRadius`, `getColor`.

Such methods are called *accessor* methods (or simply *accessors*) or *getters* because they give outsiders access to the values of private fields of an object.

It is a very common practice to provide accessor methods for those private fields of a class that may be of interest to other classes. (You will have to figure out how this works for our `inflate` method in Chapter 5 exercises, Question 17.) Methods like `setRadius` are called *setters* or *modifiers*.

❖ ❖ ❖

Inheritance represents the *IS-A relationship* between objects. If `RoundBalloon` is a subclass of `Balloon`, then a `RoundBalloon` IS-A (is a) `Balloon`. In addition to the fields and methods, an object of a subclass inherits a less tangible but also very valuable asset from its superclass: its type. It is like inheriting the family name or title. The superclass's type becomes a secondary, more generic type of an object of the subclass. Whenever a statement or a method call expects a `Balloon`-type object, you can plug in a `RoundBalloon`-type object instead.

For example, the class `DrawingPanel` has a statement:

```
balloons.add(activeBalloon);
```

It expects that `activeBalloon` refers to a `Balloon` object, but it can refer to a `RoundBalloon` object as well. The following statement will compile with no problems:

```
Balloon round = new RoundBalloon();
```

Since every class extends the class `Object` by default, every object IS-A(n) `Object`.

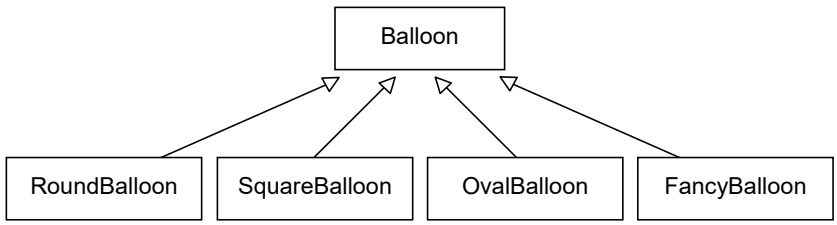
Inheritance allows us to add objects of various subclasses to a project. For example, in the *BalloonDraw* project we can define subclasses of `Balloon`: `RoundBalloon`, `OvalBalloon`, `SquareBalloon`, or `FancyBalloon` (you will do just that in the lab in the next section). These subclasses will take advantage of some of the methods of `Balloon`, but some other methods, such as `draw`, will have to be redefined appropriately for each subclass.

❖ ❖ ❖

We have made our first steps in OOP, but a lot remains to be learned. We will discuss more advanced concepts in Chapters 10 and 12.

4.6 *Case Study and Lab:* Balloons of All Kinds

The purpose of this lab is to add support for balloons of different shapes to the *BalloonDraw* program. Different balloons will be implemented as subclasses of the `Balloon` class:



We have updated the `ControlPanel` class, replacing the "Add Balloon" button with a pull-down list, which allows the user to pick a balloon of a particular shape. We have also added a parameter to the `addBalloon` method in the `DrawingPanel` class. It indicates what kind of balloon to add to the list of balloons. Your task is to write classes for different kinds of balloons.

1. Make a copy of `JM\Ch04\BalloonDraw\Balloon.java` and name it `RoundBalloon.java`. Replace the class header in `RoundBalloon` to read

   ```
   public class RoundBalloon extends Balloon
   ```

   Remove all the fields. Rename the constructors appropriately and replace the code in them with calls to `super` as follows:

   ```
   public RoundBalloon()
   {
     super();  // this is optional: default
   }

   public RoundBalloon(int x, int y, int r, Color c)
   {
     super(x, y, r, c);
   }
   ```

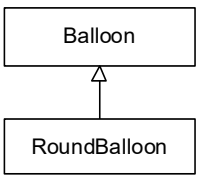
   `super` calls the corresponding constructor of the superclass.

Remove all the methods from `RoundBalloon.java` except `draw`. Go back to the `Balloon` class and remove the `draw` method, leaving only empty braces:

```
public void draw(Graphics g, boolean makeItFilled)
{
}
```

There is a better way to "disable" a method: we can declare it "abstract" and provide no code for it at all, not even empty braces. We will discuss abstract classes and methods in Chapter 12.

We now have a class `Balloon` and its subclass `RoundBalloon`.



`Balloon`'s private fields are not directly accessible in `RoundBalloon`, so you need to replace the references to them in the `draw` method with calls to "getter" methods, as explained on Page 83. It may be convenient to introduce temporary "local" variables to hold the values obtained with getters:

```
...
  int x = getX();
  int y = getY();
  int r = getRadius();
  ...
    g.fillOval(x - r, y - r, 2*r, 2*r);
```

Notice how `Balloon` and its subclass share responsibilities: `Balloon` supplies more general methods, while `RoundBalloon` supplies a `draw` method specific to the round shape.

At this point it would be nice to test what we have got, but there is a problem: the program also expects `OvalBalloon`, `SquareBalloon`, and `FancyBalloon` classes and we don't have them. The solution is to use temporary "stub" classes. For example:

```
import java.awt.Color;

public class OvalBalloon extends Balloon
{
  public OvalBalloon() {  }

  public OvalBalloon(int x, int y, int r, Color c)
  { super(x, y, r, c); }
}
```

Type up the above stub class for `OvalBalloon` and create similar stub classes for `SquareBalloon` and `FancyBalloon`.

Set up a project with the following eight classes in it: `Balloon.java`, `RoundBalloon.java`, your three stub classes, and the three classes provided for you in the `JM\Ch04\Balloons-All-Kinds` folder — `BalloonDraw.java`, `ControlPanel.java`, and `DrawingPanel.java`.

Test your program; make sure it runs the same way as the original *BalloonDraw* when you choose "Round" from the pull-down list. What happens when you choose "Oval" or another balloon that has not been fully implemented yet? Explain why.

2. Replace the `OvalBalloon` stub class with a real class. The easiest way of doing this is to adapt the code from the `RoundBalloon` class. Copy `RoundBalloon.java` into `OvalBalloon.java`, rename the class and the constructors, and change the `draw` method. An oval balloon is twice as tall as it is wide; its height is `2*radius`, same as before, but its width is `radius`. When you draw it, make sure its center remains at `xCenter`, `yCenter`.

   You also need to redefine the `distance` method. Copy it from the `Balloon` class into `OvalBalloon` and make a simple change, so that the "distance" from any point on the border is equal to `radius`. Hint: if you stretch the oval horizontally by a factor of two, it becomes a circle, so when you define the "distance," you need to multiply `dx` by 2.

   Retest your program thoroughly. Make sure you can "grab" an oval balloon at any point inside it to move it, and "grab" the balloon at any point on the border to stretch it.

3. ■ Repeat Step 2 for the `SquareBalloon` class. Explore the documentation for the Java library class `Graphics` and find the methods that draw and fill a rectangle. The parameters for drawing a square will be the same as the parameters for drawing a circle (because the library methods that draw an oval use the oval's bounding rectangle).

   You need to redefine the `distance` method again. It turns out there is a simple formula that will make the "distance" from any point on the border of the square equal to radius. Hint: `Math.abs(dx)` returns the absolute value of `dx`; `Math.max(dx, dy)` returns the larger of `dx` and `dy`. Retest your program thoroughly again.

Continued 

4.◆ Create a `FancyBalloon` class that has a shape of your own design. One example may be a vertically stretched rectangle with rounded corners (see the `Graphics` class). Another example is a diamond shape. For that you will need to use the `drawPolygon` method that takes a list of vertices. (There is a simple "distance" formula for a diamond.) Another way to create an interesting shape is to take two overlapping shapes with the same center, for example, two ovals, one stretched vertically, the other horizontally, or a square and a diamond. If you know the "distance" formula for each shape, you can combine them to make the "distance" for their combination. Hint: `Math.min(d1, d2)` returns the smaller of `d1` and `d2`.

You can also play with colors, mixing different colors and adding decorations to your balloon. Refer to the Java documentation for the `Color` class. If `c` is a `Color`, `c.darker()` returns a darker color of the same tint; `c.brighter()` returns a lighter color.

4.7 Summary

An OOP program is best visualized as a virtual world of interacting objects. A program's source code describes different types of objects used in the program. Objects of the same type are said to belong to the same *class*. An object is called an *instance* of its class. The source code of a Java program consists of *definitions of classes*.

The source code for each class is stored in a separate file with the same name as the class and the extension `.java`. By convention, a class name always starts with a capital letter. It is customary to place all your classes for a small project into the same folder. Several compiled Java classes may be collected in one `.jar` file.

A *CRC card* gives a preliminary, informal description of a class, listing its name, the key "responsibilities" of its objects, and the other classes this class depends on ("collaborators").

The `import` statements at the top of a class's source code tell the compiler where it can find the library classes and packages used in that class.

A class's source code begins with an optional brief comment that describes the purpose of the class, followed by `import` statements, if necessary, then the class's header, and the class's body within braces. A class defines the data elements of an object of that class, called *instance variables* or *fields*. Each instance variable has a name, given by a programmer, and a type. The set of fields serves as the "personal memory" of an object. Their values may be different for different objects of the

class, and these values can change while the program is running. A class also defines *constructors*, which are short procedures for creating objects of that class, and *methods*, which describe what an object can do.

A constructor always has the same name as its class. A constructor is used primarily to set the initial values of the object's fields. It can accept one or several parameters that are used to initialize the fields. A constructor that does not take any parameters is called a *no-args constructor*. A class may have several constructors that differ in the number or types of parameters that they accept. If no constructors are defined, the compiler automatically supplies one default no-args constructor that sets all the instance variables to default values (zeroes for numbers, `null` for objects, `false` for `boolean` variables).

You create a new object in the program using the `new` operator. The parameters passed to `new` must match the number, types, and order of parameters of one of the constructors in the object's class, and `new` invokes that constructor. `new` allocates memory to store the newly constructed object.

The functionality of a class — what its objects can do — is defined by its *methods*. A method accomplishes a certain task. It can be called from constructors and other methods of the same class and, if it is declared `public`, from constructors and methods of other classes. A method can take parameters as its "inputs." Parameters passed to a method must match the number, types, and order of parameters that the method expects. A method can return a value of a specified type to the caller. A method declared `void` does not return any value.

In OOP, all the instance variables of a class are usually declared `private`, so only objects of the same class have direct access to them. Some of the methods may be private, too. Users of a class do not need to know how the class is implemented and what its private fields and methods are. This practice is called *information hiding*. A class interacts with other classes only through a well-defined set of constructors and public methods. This concept is called *encapsulation*. Encapsulation facilitates program maintenance, code reuse, and documentation. A class often provides public methods that return the values of an object's private fields, so that an object of a different class can access those values. Such methods are called *accessor methods*, *accessors*, or "getters." Methods that set the values of private fields are called *modifiers* or "setters."

A class definition does not have to start from scratch: it can *extend* the definition of another class, adding fields and/or methods and/or overriding (redefining) some of the methods. This concept is called *inheritance*. It is said that a *subclass* (or *derived class*) extends a *superclass* (or *base class*). Constructors are not inherited.

An object of a subclass also inherits the type of its superclass as a secondary, more generic type. This formalizes the *IS-A relationship* between objects: an object of a subclass IS-A(n) object of its superclass.

Exercises

Sections 4.1-4.4

1. Mark true or false and explain:

(a) The name of a class in Java must be the same as the name of its source file (excluding the extension `.java`). _____
(b) The names of classes are case-sensitive. _____
(c) The `import` statement tells the compiler which other classes use this class. _____ ✓

2. Mark true or false and explain:

(a) The *BalloonDraw* program consists of one class. _____ ✓
(b) A Java program can have as many classes as necessary. _____
(c) A Java program is allowed to create only one object of each class. _____
(d) Every class has a method called `main`. _____ ✓

3. Navigate your browser to Oracle's Java API (Application Programming Interface) documentation web site (for example, `http://download.oracle.com/javase/`, or simply google "Java API"). If you have the Java documentation installed on your computer, open the file `<JDK folder>/docs/api/index.html` (for example, `C:/Program Files/Java/jdk-16/docs/api/index.html`)

(a) Approximately how many different packages are listed in the API spec?

(b) Find `JFrame` in the list of classes in the left column and click on it. Scroll down the main window to the "Method Summary" section. Approximately how many methods does the `JFrame` class have, including methods inherited from other classes? 3? 12? 25? 300-400? ✓

4. Mark true or false and explain:

(a) Fields of a class are usually declared `private`. _____
(b) An object has to be created before it can be used. _____ ✓
(c) A class may have more than one constructor. _____
(d) The programmer gives names to objects in his program. _____
(e) When an object is created, the program always calls its `init` method. _____ ✓

5. What are the benefits of encapsulation? Name three.

6. Make a copy of `balloondraw.jar` (`JM\Ch04\BalloonDraw\balloondraw.jar`) and rename it into `balloondraw.zip`. Examine its contents. As you can see, a `.jar` file is nothing more than a compressed folder. How many compiled Java classes does it hold?

7.◆ Create a class `Book` with two private `int` fields, `numPages` and `currentPage`. Supply a constructor that takes one parameter and sets `numPages` to that value and `currentPage` to 1. Provide accessor methods for both fields. Also provide a method `nextPage` that increments `currentPage` by 1, but only if `currentPage` is less than `numPages`.

Hint:

```
if (currentPage < numPages)
    currentPage++;
```

Create a `BookTest` class with a `main` method. Let `main` create a `Book` object with 3 pages, then call its `nextPage` method three times, printing out the value of `currentPage` after each call.

8.■ (a) Write a simple class `Circle` with one field, `radius`. Supply one constructor that takes the radius of the circle as a parameter:

```
public Circle(int r)
{
  ...
}
```

Supply one method, `getArea`, that computes and returns (not prints!) the area of the circle. Provide the accessor method `getRadius`. Create a small class `CircleTest` with a `main` method that prompts the user to enter an integer value for the radius, creates one `Circle` object with that radius, calls its `getArea` method, and prints out the returned value.

Hints:

1. `getArea` should return a value of the type `double`;
2. The class `Math` has a constant `Math.PI`;
3. Figure out a way to square a number: search online or look in the `Math` class, or just do it.

(b) Create a class `Cylinder` with two private fields: `Circle base` and `int height`. Is it fair to say that a `Cylinder` HAS-A `Circle`? Provide a constructor that takes two `int` parameters, `r` and `h`, initializes `base` to a new `Circle` with radius `r`, and initializes `height` to `h`. Provide a method `getVolume` that returns the volume of the cylinder (which is equal to the base area multiplied by height). Create a simple test program `CylinderTest`, that prompts the user to enter the radius and height of a cylinder, creates a new cylinder with these dimensions, and displays its volume.

9.◆ `JM\Ch04\Exercises\CoinTest.java` is part of a program that shows a picture of a coin in the middle of a window and "flips" the coin every two seconds. Your task is to supply the second class for this program, `Coin`.

The `Coin` class should have one constructor that takes two parameters of the type `Image`: the heads and tails pictures of the coin. The constructor saves these images in the coin's private fields (of the type `Image`). `Coin` should also have a field that indicates which side of the coin is displayed. The `Coin` class should have two methods:

```
/**
 * Flips this coin
 */
public void flip()
{
  ...
}
```

and

```
/**
 * Draws the appropriate side of the coin
 * centered at (x, y)
 */
public void draw(Graphics g, int x, int y)
{
  ...
}
```

≤ Hints:

1. `import java.awt.Image;`
   `import java.awt.Graphics;`
2. The class `Graphics` has a method that draws an image at a given location. Call it like this:

   ```
   g.drawImage(pic, xUL, yUL, null);
   ```

   where `pic` is the image and `xUL` and `yUL` are the coordinates of its upper left corner. You need to calculate `xUL` and `yUL` from the `x` and `y` passed to `Coin`'s `draw`. Explore the documentation for the library class `Image` to find methods that return the width and height of an image.
3. Find copyright-free image files for the two sides of a coin on the Internet or scan or take a picture of a coin and create your own image files.

≥

Sections 4.5-4.7

10. Mark true or false:

(a) A subclass inherits all the fields and public methods of its superclass. _____ ✓

(b) A subclass inherits all those constructors of its superclass that are not defined explicitly in the subclass. _____ ✓

11. Question 8-b above asks you to write a class `Cylinder` with two fields: `Circle base` and `double height`. Instead of making `base` a field we could simply derive `Cylinder` from `Circle`, adding only the `height` field. Discuss the merits of this design option. ✓

12. The classes in this question refer to the Lab 4.6. Which of the following assignment statements will compile without errors? Explain your answers.

(a) `RoundBalloon b1 = new RoundBalloon();` _____

(b)
```
Balloon b2 = new RoundBalloon(100, 100, 10,
                              Color.RED);
```
_____

(c) `Balloon b3  = new SquareBalloon();` _____

(d)■ `RoundBalloon b5 = new Balloon();` _____

13.■ Using the `Balloon` class in `JM\Ch04\BalloonDraw\Balloon.java`, write a simple console application that creates a `Balloon` and "prints it out":

```
Balloon b = new Balloon(100, 100, 20, Color.RED);
System.out.println(b);
```

What is displayed? It appears the program doesn't know how to properly "print" a `Balloon` object. Find online how to fix that by overriding `Object`'s `toString` method in the `Balloon` class. ≤ Hint: see, for example, `www.javatpoint.com/understanding-toString()-method`. ≥ ✓

14.◆ Using the `Balloon` class and its subclasses defined in the lab in Section 4.6, add a couple of balloons of different shapes and colors to the *Banner* program in Chapter 2 (`JM\Ch02\HelloGui\Banner.java`). Make them fly up, then start at the bottom again. ≤ Hints: add fields that refer to the balloons; create the balloons in the `main` method; draw the balloons in the `paintComponent` method; a `Balloon` object "knows" its own position; increase the height of the window. ≥

```
int chapter = 5;
```

Data Types, Variables, and Arithmetic

5.1 Prologue

Java and other high-level languages let programmers refer to a memory location by name. These named "containers" for values are called *variables*. The programmer gives variables meaningful names that reflect their role in the program. The compiler/interpreter takes care of all the details — allocating memory space for the variables and representing data in the computer memory.

The term "variable" is borrowed from algebra because, as in algebra, variables can assume different values and can be used in *expressions*. The analogy ends there, however. In a computer program, variables are actively manipulated by the program. A variable is like a slate on which the program can write a new value when necessary and from which it can read the current value. For example, the statement

```
a = b + c;
```

does not mean that *a* is equal to *b* + *c*, but rather a set of instructions:

1. Get the current value of `b`;
2. Get the current value of `c`;
3. Add the two values;
4. Assign the result to `a` (write the result into `a`).

The same is true for

```
a = 4 - a;
```

It is not an equation, but a set of instructions for changing the value of `a`:

1. Get the current value of the variable `a`;
2. Subtract it from `4`;
3. Assign the result back to `a` (write the new value into `a`).

In Java, a statement

```
someName = expression;
```

represents an *assignment* operation that evaluates (finds the value of) the expression on the right side of the = sign and assigns that value to (writes it into) the variable `someName` on the left side of =. The = sign is read "gets the value of": "`someName`

gets the value of *expression*." (If you want to compare two values, use another operator, ==, to mean "is equal to.")

In Java, every variable has a *data type*. This can be either a *primitive data type* (such as `int`, `double`, `boolean`, and `char`) or a class type (such as `String`, `Color`, `Balloon`). The programmer specifies the variable's data type based on the kind of data it will contain.

A variable's data type determines the range of values that can be stored in the variable, the amount of space allocated for it in memory, and the kind of operations that can be performed with it.

A variable of type `int`, for example, contains an integer value, which can be in the range from -2^{31} to $2^{31}-1$; a variable of type `double` represents a real number. A variable of type `String` refers to an object of the `String` class.

In Java, a variable that represents an object holds a *reference* to that object. A reference is basically the address of the object in RAM when the program is running. When an object is created with the `new` operator, the interpreter allocates memory space for the object and returns a reference to it. That reference can be saved in a variable and used to access the object. We'll explain references in more detail in Section 10.4.

In this chapter we explain the following concepts and elements of Java syntax:

- The syntax and placement of declarations for variables and constants
- Primitive data types
- Strings
- Literal and symbolic constants
- Conversion of values from one type to another (casts)
- The scope of variables and symbolic constants
- Java's arithmetic operators

5.2 Declaring Fields and Local Variables

Variables in Java fall into three main categories: *fields*, *local variables*, and *parameters* in constructors or methods. The fields are declared within the body of a class, outside of any constructor or method. They can be used in any constructor or method of the class. Local variables are declared and used inside a particular constructor or method. Parameters are described in the header for a constructor or method and act pretty much like local variables within that constructor or method.

The fragment of code in Figure 5-1 shows several declarations of fields in the `Balloon` class and a constructor that sets their values.

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

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

Declarations of fields (a.k.a. instance variables)

Figure 5-1. A fragment from the `Balloon` class

Fields are declared outside of any constructor or method, usually at the top of the class's body.

But regardless of where they are placed, they are "visible" in all the constructors and all the methods of the class.

Usually all fields are declared `private`, which means they can be used only by the constructors and methods of their class and are not directly accessible to methods of other classes.

However, some constants may be declared `public`.

The general format for a field declaration is

```
private sometype someName;
```

or

```
private sometype someName = expression;
```

If a field is not explicitly initialized, Java provides a default value, which is 0 for fields of numeric types, `false` for `boolean`, and `null` for objects.

For example:

```
private int xCenter;  // xCenter is set to 0

private Color color;  // color is set to null
```

Assigning a variable a `null` value indicates that it does not currently refer to any valid object. `null` is a Java reserved word.

Local variables are temporary variables that are declared <u>inside</u> a constructor or method. Once a method is finished, its local variables are discarded.

For example, the `distance` method in the `Balloon` class uses local variables `dx` and `dy`:

```
public double distance(int x, int y)
{
  double dx = Math.abs(x - getX());
  double dy = Math.abs(y - getY());
  return Math.sqrt(dx*dx + dy*dy);
}
```

The general format for a local variable declaration is similar to a field declaration:

```
sometype someName;
```

or

```
sometype someName = expression;
```

where *sometype* declares the type of the variable and `someName` is the name given by the programmer to this particular variable.

`public` or `private` cannot be used in declarations of local variables.

A local variable must be declared and assigned a value before it can be used — local variables do not get default values.

Parameters in a constructor or method are sometimes called *formal parameters*. They get their initial values from the actual parameters that the caller passes to that constructor or method. They act like local variables within the constructor or method. In the above example of the distance method, `x` and `y` act like local variables.

Note the following features in the declarations of fields and local variables:

1. A declaration, like other Java statements, always ends with a semicolon.

2. A declaration must include the type of the variable and its name. For example:

```
int x;
Color c;
String message;
```

3. Several variables of the same type may be listed in one declaration, separated by commas. For example:

```
int k, n;
double x, y, z;
```

4. A declaration of a variable may include an initialization that sets its initial value. For example:

```
int sum = 0;
Color c = getColor();
```

5. A variable may be declared `final`, which means that its value, once assigned, cannot change. For example:

```
private static final Color DEFAULT_COLOR = Color.PINK;
public static final double PI = 3.14159265358979323846;
```

So an initialized `final` "variable" is not actually a variable, but a constant. A constant's initial value is also its "final" value.

A variable can be declared only once within its *scope* ("scope" refers to the space in the program where the variable is "visible" — see Section 5.6).

For example, to have both

```
int sum;
...
int sum = m + n;
```

within the same method is a syntax error. Use either

```
int sum;
...
sum = m + n;
```

or

```
int sum = m + n;
```

Java allows you to use the same name for a field and a local variable. This may cause bugs that are sometimes hard to find.

For example, if we inadvertently write

```
Color color = Color.BLUE;
```

instead of

```
color = Color.BLUE;
```

in the `Balloon`'s constructor, the compiler will treat this statement as a declaration of a local variable `color` and leave the field `color` uninitialized (`null`). When later another method attempts to call one of `color`'s methods (for example, `brighter()`, the program will "throw" a `NullPointerException`.

You might wonder: Why do we need local variables? Why can't we make them all fields? Technically it would be possible to use only fields, but a class is much better organized if variables that are used locally are kept local. In a good design, a field is a variable that describes a truly important, "global" attribute of the objects of a class. You don't want to use global variables for your temporary needs — just like you don't carry your own plate and silverware to every fast-food restaurant you visit.

5.3 Primitive Data Types

In Java, values of primitive data types are not objects. This is a concession to earlier programming languages, like C, from which Java borrows much of its syntax. Objects have data types, too: the data type of an object is its class. But `int`, `double`, and other primitive data types are not classes.

Java has the following eight *primitive data types*, designated by reserved words:

```
boolean     byte
char        short
int         long
double      float
```

Note that like other Java reserved words, the names of the primitive data types are spelled in lowercase letters. They are called *primitive* because variables of these types do not have the properties of objects (in particular, they do not have any methods). A `char` variable holds one character (in Unicode); an `int`, `byte`, `short`, or `long` variable holds an integer; a `float` or `double` variable holds a floating-point real number.

Because variables of different types occupy different numbers of bytes in memory, we say they have different *sizes*.

In Java each data type has a fixed size, regardless of the particular computer model or brand of Java interpreter. Table 5-1 summarizes the primitive types, their sizes, and the ranges of their values.

Although `float` and `double` can represent a huge range of numbers, their precision is limited to only about seven significant digits for the `float` and about twice as many for the `double`.

Type	Size (bytes)	Range
`boolean`	1	`true` or `false`
`char`	2	Unicode character set (with ASCII subset)
`byte`	1	from $-2^7 = -128$ to $2^7-1 = 127$
`short`	2	from $-2^{15} = -32{,}768$ to $2^{15}-1 = 32{,}767$
`int`	4	from -2^{31} to $2^{31}-1$
`long`	8	from -2^{63} to $2^{63}-1$
`float`	4	approx. from -3.4×10^{38} to 3.4×10^{38}
`double`	8	approx. from -1.8×10^{308} to 1.8×10^{308}

Table 5-1. The primitive data types

In this book we will work primarily with the `int`, `double`, `char`, and `boolean` primitive data types.

The `boolean` type is discussed in the next chapter.

It is a programmer's responsibility to make sure the values of variables and all the intermediate and final results in arithmetic expressions fit within the range of the chosen data types, and that these types satisfy the precision requirements for computations.

5.4 Strings

In Java, character strings (short fragments of text) are represented by `String` objects. `String` is a library class from the `java.lang` package, which is built into Java. A `String` is an object, so `String` is not a primitive data type.

The `String` class mostly behaves like any other class: it has constructors and public methods, described in the Java API documentation. (The `String` class also has private methods and fields; we are not interested in them, because they are private.) We will discuss the most commonly used `String` methods in Chapter 8 and will also explain there why `String` constructors are rarely used.

There are two Java features, however, that make the `String` class special. First, the compiler recognizes strings of characters in double quotes as constant `String` objects — *literal strings*. You can write, for example,

```
String str = "Hello, World!";
```

The compiler automatically creates a `String` object with the value `"Hello, World"` and assigns a reference to that object to the variable `str`. There is no need to use the `new` operator to create a literal string.

Second, Java allows you to apply the + operator to strings. In general, you cannot apply arithmetic operators to objects. Strings are an exception: when + is applied to two strings, it concatenates them. For example,

```
String str = "Chapter" + "5";
```

is the same as

```
String str = "Chapter5";
```

5.5 Constants

A *constant* represents a "variable" whose value does not change while the program is running. Your source code may include *literal constants* and *symbolic constants*.

Examples of literal constants are decimal representations of integers and real numbers, characters in single quotes, and text in double quotes:

```
'y', 'H'                          (chars)
 7, -3                            (ints)
 1.19, .05, 12.0, 3., 0.4         (doubles)
"Hello, World!", "1776", "y"      (Strings)
```

It is possible for a literal character string to consist of only one character (for example, `"y"`). Java also allows an empty string — it is designated by two double quote characters next to each other, with nothing in between: `""`.

Character constants include a special set of non-printable characters designated by *escape sequences* (the term derives from printer control commands).

In Java, an escape sequence is a pair of characters: a designated printable character preceded by the "escape character," a backslash. An escape pair is placed within single quotes to designate a one-character constant.

Escape sequences include:

`\n`	newline
`\r`	carriage return
`\t`	tab
`\f`	form feed
`\'`	single quote
`\"`	double quote
`\\`	backslash

The most commonly used one is `\n` — "newline."

Escape pairs can be used in literal string constants. For example:

```
System.out.println("\nLet it go,\n\tlet it go");
```

will be displayed like this:

```
< blank line >
Let it go,
    let it go
```

`"\n"`, for example, represents a string that consists of one newline character.

Symbolic constants are usually represented by initialized `final` fields. For example:

```
private static final double LBS_IN_KG = 2.20462;
```

(Sometimes, programmers write names of symbolic constants using all capital letters for better visibility.) `final` is a Java reserved word.

The general form of a symbolic constant's declaration is

```
[optional modifiers] final sometype someName = expression;
```

where *sometype* is a data type followed by a name of the constant and its value. A constant may also be initialized to the value of some expression.

A constant doesn't have to be initialized right away in its declaration — its "final" value can be set in a constructor.

For example:

```
public class Die
{
    private final int numFaces;

    public Die(int n)
    {
        numFaces = n;
    }
```

It may seem, at first, that symbolic constants are redundant and we can simply use their literal values throughout the program. Consider, for instance, the following statement:

```
double totalAmt = amt * (1.0 + TAX_RATE);
```

Surely we could write the same statement with a specific number plugged in:

```
double totalAmt = amt * 1.05;
```

At a first glance it might seem simpler. However, there are several important reasons for using symbolic constants.

The most important reason for using symbolic constants is that it simplifies program maintenance. If the program is modified in the future and the value of a constant needs to be changed, a change to the constant's declaration will change the constant's value throughout the program (after the class that contains the constant is recompiled).

In the above example, if we needed to adjust the tax rate, we would have to change the numbers. A programmer making the change would have to figure out what all the numbers mean and replace them. The task is even harder if the numbers related to one constant are scattered throughout the program.

Another advantage of symbolic constants is that they make the code more readable and self-explanatory if their names are chosen well. The name can explain the role a constant plays in the program, making additional comments unnecessary.

It is also easier to change a symbolic constant into a variable if, down the road, a future version of the program requires it.

Symbolic constants, like variables, are declared with a particular data type and are visible only within their scope (explained in the next section). This introduces more order into the code and gives the compiler additional opportunities for error checking — one more reason for using symbolic constants.

On the other hand, there is no need to clutter the code with symbolic names assigned to universal constants such as 0 or 1, or

```
public static final int FULL_CIRCLE = 360;
final int HOURS_IN_DAY = 24;
```

5.6 Scope of Variables

As we have discussed above, each constructor and method in a class can have its own local variables, while the fields of a class can be used in all its methods. The question of where a variable is visible and can be used relates to the subject of *scope*.

In Java a variable is defined only within a certain space in the program called the *scope* of the variable.

Scope discipline helps the compiler perform important error checking. If you try to use a variable or constant outside its scope, the compiler detects the error and reports

an undefined name. The compiler also reports an error if you declare the same name twice within the same scope.

The scope of a field extends throughout the class, including all its constructors and methods.

The scope of a local variable extends from its declaration to the end of the block in which it is declared.

A local variable exists only temporarily while the program is executing the block where that variable is declared. When a program passes control to a method, a special chunk of memory (a *frame* on the system *stack*) is allocated to hold that method's local variables. When the method is exited, that space is released and all local variables are destroyed.

Local variables in Java do not have to be declared at the top of a method but may be declared anywhere in the method's code. But declarations inside nested blocks can lead to elusive bugs. We recommend that at first you avoid declaring local variables inside nested blocks unless you know exactly what you are doing.

As we have mentioned earlier, Java allows you to use the same name for a field and a local variable, with the local variable taking precedence over the global one. This may lead to hard-to-catch errors if you inadvertently declare an identically named local variable that overrides the global one.

This possible overlap of names is a good reason to give a class's fields conspicuous names.

`x`, `y`, or `r` are bad choices for field names — they should be saved for throwaway local variables. `radius` and `sideLength` are better names for fields.

It is perfectly acceptable to use the same name for local variables in different methods.

In fact, this is a good practice when the variables represent similar quantities and are used in similar ways. But never try to economize on declarations of temporary local variables within methods by making them fields. Everything should be declared where it belongs.

5.7 Arithmetic Expressions

Arithmetic expressions are written the same way as in algebra and may include literal and symbolic constants, variables, the arithmetic operators +, –, *, and /, and parentheses.

The order of operations is determined by parentheses and by the ranks of operators: multiplication and division are performed first (left to right), followed by addition and subtraction. Multiplication requires the `*` sign — it cannot be omitted. You can also use the minus symbol for negation. For example:

```
x = -(y + 2*z) / 5;
a = -a;                  // Negate a
```

Java also has the `%` operator for integers:

```
a % b
```

which is read "a modulo b," and means the remainder when `a` is divided by `b`. For example, 31 % 7 is equal to 3; 365 % 7 is 1; 5 % 17 is 5. This operator is handy for computing values that change in a cyclic manner. For example:

```
int minsAfterHour = totalMins % 60;
int dayOfWeek = (dayOfWeekOnFirst - 1 + day) % 7;
int lastDigitBase10 = x % 10;
```

It is also used to check whether a number is evenly divisible by another number. For example:

```
if (k % 2 == 0) ... // if k is even ...
```

The `%` operator has the same rank as `*` and `/`.

Java allows programmers to mix variables of different data types in the same expression. Each operation in the expression is performed according to the types of its operands, and its result receives a certain type.

The type of the result depends only on the types of the operands, not their values. If the two operands have the same type, the result of the operation automatically gets the same type as the operands.

This principle goes back to the C language and has serious consequences, especially for division of integers. If you write

```
int a = 7, b = 2;
System.out.println(a / b);
```

you will see `3` displayed, and not `3.5` as you might expect. The reason is that both `a` and `b` are integers and therefore the result of `a / b` must be also an integer. If it isn't, it is truncated to an integer (in the direction toward 0). So `7/2` is evaluated as `3` and `-7/2` as `-3`.

If you want to get a true ratio of two `int` variables, you have to convert your `int`s into `double`s. This can be done by using the *cast* operator, designated by the target type in parentheses.

For example:

```
double ratio = (double)a / (double)b;
```

The above statement is basically equivalent to introducing two temporary variables:

```
double tempA, tempB;
tempA = a;
tempB = b;
double ratio = tempA / tempB;
```

But casts do it for you.

The general syntax for the cast operator is

```
(sometype)variable
```

or

```
(sometype)(expression)
```

Note that the "same type" rule applies to all intermediate results of all operations in an expression.

If you write

```
int a = 7, b = 2;
double ratio = a / b;  // Too late!
```

`ratio` still gets the value of 3.0, because the result of `a / b` is truncated to an `int` before it is assigned to `ratio`. Similarly, if you write

```
int degreesCelsius = 5 / 9 * (degreesFahrenheit - 32);  // Error!
```

`degreesCelsius` will be always set to 0 (because `5 / 9` is evaluated first and its result is 0).

If the two operands have different types, the operand of the "smaller" type is *promoted* (that is, converted) to the "larger" type. (`long` is "larger" than `int`, `float` is "larger" than `long`, and `double` is the "largest"). Therefore, in the above example it would have sufficed to use only one cast:

```
double ratio = (double)a / b;
```

or

```
double ratio = a / (double)b;
```

The other operand would be promoted to a `double`. But trying to cast the resulting ratio would cause the same problem as before:

```
double ratio = (double)(a / b);
                        // Error: the result of a / b is already
                        //   truncated!  The cast is too late!
```

Your code will be better documented if you indicate explicit type conversions using the cast operator, where necessary, rather than relying on implicit type conversions.

You don't need to use casts with literal constants — just choose a constant of the right type.

For example:

```
double volume = 4.0 / 3.0 * Math.PI * Math.pow(r, 3.0);
```

computes the volume of a sphere with the radius *r* (which is equal to $\frac{4}{3}\pi r^3$).

Sometimes you may need to use a cast in the opposite direction: to convert a "larger" type into a "smaller" one, such as a `double` into an `int`. For example:

```
int pointsOnDie = 1 + (int)(6*Math.random());
  // 0.0 <= Math.random() < 1.0
```

The `(int)` cast truncates the number in the direction of 0, so `(int)1.99` is `1` and `(int)(-1.99)` is `-1`.

If you want to round a `double` value to the nearest integer, add 0.5 to a positive number or subtract 0.5 from a negative number first, and then cast it into an `int`. For example:

```
int percent = (int)((double)count / totalCount * 100 + 0.5);
```

or

```
int percent = (int)(100.0 * count / totalCount + 0.5);
```

The cast operator applies only to "compatible" data types. You cannot cast a number into a string or vice-versa.

There are several ways to convert numbers and objects into strings. One of them is discussed in Section 5.10.

5.8 Compound Assignment and Increment Operators

Java has convenient shortcuts for combining arithmetic operations with assignment. The following table summarizes the *compound assignment* operators:

Compound assignment:	Is the same as:
`a += b;`	`a = a + b;`
`a -= b;`	`a = a - b;`
`a *= b;`	`a = a * b;`
`a /= b;`	`a = a / b;`
`a %= b;`	`a = a % b;`

For example, the following statement:

```
sum += k * k;
```

is the same as:

```
sum = sum + k * k;
```

The `+=` or `*=` forms may seem cryptic at the beginning, but, once you get used to them, they become attractive — not only because they are more concise, but also because they emphasize the fact that the same variable is being modified. Not using compound assignments where they can be used immediately gives away an amateur.

❖ ❖ ❖

As we have seen, the `+` operator, when applied to two strings, concatenates them. `str1 += str2` also works for strings, creating a new string by concatenating `str1` and `str2` and assigning the result to `str1`. For example:

```
String fileName = "flower";
fileName += ".gif";     // fileName now refers to "flower.gif"
```

❖ ❖ ❖

Another syntactic shortcut is the set of special *increment/decrement* operators. These operators are used for incrementing or decrementing an integer variable by one:

Increment/ decrement:	Is the same as:
`a++`	`a = a + 1`
`a--`	`a = a - 1`
`++a`	`a = a + 1`
`--a`	`a = a - 1`

Increment and decrement operators may be used in expressions. That is where the difference between the `a++` and `++a` forms and between the `a--` and `--a` forms becomes very important. When `a++` is used, the value of the variable `a` is incremented after it has been used in the expression; for the `++a` form, the value of the variable `a` is incremented before it has been used in the expression. This can get quite confusing and hard to read. For example:

```
a = b + c++;  // Too much!
```

Consider using `a++` and `a--` only as a separate statement. Avoid `++a` and `--a` altogether, and avoid using `++` and `--` in arithmetic expressions.

5.9 Avoiding Division by Zero Errors

Division by zero errors are never caught by the compiler, even if they are obvious:

```
int a = 10/0;
```

Instead, such errors show up at run time. Different programming languages treat division by zero errors differently. In Python, for example, the program is always aborted with a `ZeroDivisionError` error message. In Java, the behavior depends on the type of the operands. If the result is supposed to be an integer, the program throws an `ArithmeticException` and displays

```
ArithmeticException: / by zero
```

But if the result is supposed to be a `double`, it gets the special value "Infinity," no message is displayed, and the program keeps running. "Infinity" is treated as the largest possible `double` number and can appear in arithmetic operations and comparisons. Thus a division by zero error won't be discovered until you print the value of the result and see the word "Infinity" displayed.

We can distinguish two kinds of situations when a division by zero error occurs. In the first situation, the denominator is never supposed to be zero, but gets the zero value anyway, due to a bug elsewhere in the program. It is not desirable to check for and prevent a division by zero error in this kind of situation because the error signals that something is wrong with the code. The programmer has to look elsewhere and find and fix the bug.

In the second situation, a division by zero error may occur because the programmer did not take into account a legitimate case when the denominator might be zero, did not consider some user input, boundary case. Perhaps the programmer forgot to consider this case and test for it. Something has to be done about it: the code should check for the 0 denominator and take a reasonable action, for example, ask the user to re-enter the number. In any case, all allowed boundary conditions must be thoroughly tested.

5.10 Converting Numbers and Objects into Strings

Java treats `String` objects in a special way: the `+` operator, when applied to two strings, concatenates them. This is simply a syntax shortcut that eliminates the need to call a method. (In fact, the `String` class has a method `concat` for concatenating a string to another, so `str1 + str2` is equivalent to `str1.concat(str2)`.)

Java also allows you to use the `+` operator for concatenation when one operand is a string and the other is a primitive data type or an object. In that case, the operand that is not a string is automatically converted into a string.

For example,

```
double v = 79.5;
System.out.print("Volume = " + v);
```

displays

```
Volume = 79.5
```

Here the + concatenates a string `"Volume = "` and a `double` value `79.5` to form a new string `"Volume = 79.5"`.

When the + operator is used for concatenation, at least one of its two operands must be a value or an expression of the `String` type.

The non-string operand is then converted into a string according to a specific rule that depends on its type. A `char` value is converted into a string that consists of that one character; an `int` value is converted into a string of its digits, with a minus sign for a negative value; a `double` value is converted into a string that may include a minus sign, then one or more digits before the decimal point, and at least one digit after the decimal point; a `boolean` value is converted into `"true"` or `"false"`. For example,

```
          (+ converts)
'A'         ===>      "A"
123         ===>      "123"
-1          ===>      "-1"
3.14        ===>      "3.14"
.1          ===>      "0.1"
Math.PI     ===>      "3.141592653589793"
false       ===>      "false"
```

The same conversion rules are used in

```
System.out.print(x);
```

for displaying the value of `x` for different types of `x`.

Note that if you have several + operators in a row without parentheses —

```
x + y + z
```

— they are evaluated from left to right. To concatenate `x`, `y`, and `z`, each + must have at least one `String` operand. For example,

```
System.out.print("***" + 2 + 2);
```

displays

```
***22
```

while

```
System.out.print(2 + 2 + "***");
```

displays

```
4***
```

Conversion also works for the += operator when the left-hand operand is a string. For example:

```
String str = "score: ";
int points = 90;
str += points;  // str now refers to "score: 90"
```

Sometimes you need to convert a value into a `String` without concatenating it with anything (for example, in order to pass it to a method that expects a `String` parameter).

The easiest way to convert a value into a `String` is to concatenate that value with an empty string.

For example:

```
JTextField scoreDisplay = new JTextField();
int score = 6;
...
scoreDisplay.setText("" + score);
```

Any object can be converted into a `String` by calling its `toString` method.

In Java, every object has a default `toString` method, which returns a `String`. The default `toString` method, however, is not very useful: it returns a string that is the object's class name followed by the object's address in memory in hexadecimal notation. Something like this: `"RoundBalloon@11a698a"`. Programmers often override the default and provide a more useful `toString` method for their class. For example:

```
public class Fraction
{
  private int num, denom;

  public Fraction(int n, int d)
  {
    num = n;
    denom = d;
  }

  ...

  public String toString()
  {
    return num + "/" + denom;
  }
}
```

With this `toString` method defined in `Fraction`, the statements

```
Fraction f = new Fraction(1, 2);
System.out.println(f);
```

display

```
1/2
```

As you can see, the `toString` method is used by `System.out`'s `print` and `println` methods for displaying an object. In other words, if `obj` is an object, then

```
System.out.print(obj);
```

is equivalent to

```
System.out.print(obj.toString());
```

5.11 *Lab:* Pie Chart

Figure 5-2 shows a snapshot from the program *Poll* that helps to run a poll for the election of a school president. The results are shown as numbers for each of the three candidates and as slices on a pie chart.

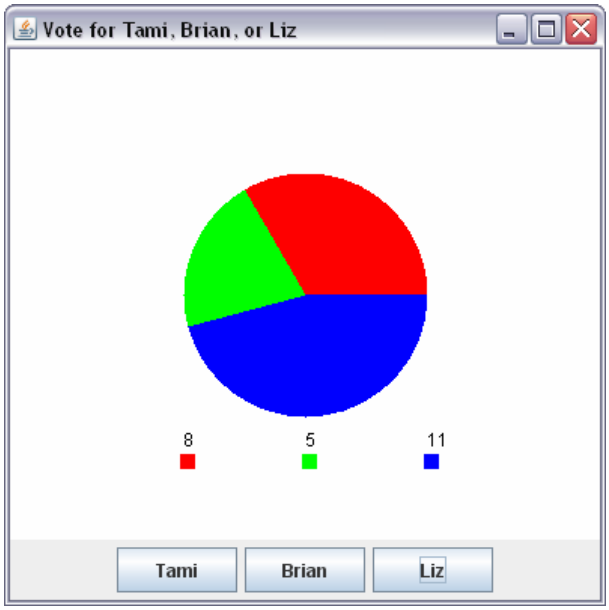


Figure 5-2. The *Poll* program

The source code for this program consists of three classes: `Poll`, `PollControlPanel`, and `PollDisplayPanel`. `Poll` is the main class: it creates a program window and adds a control panel and a display panel to it. A `PollControlPanel` object represents a control panel with the three buttons. It also handles the buttons' click events. A `PollDisplayPanel` object keeps track of the poll counts and displays them as numbers and as a pie chart.

Your task is to fill in the blanks in the `PollDisplayPanel` class. Collect the three files, `Poll.java`, `PollControlPanel.java`, and `PollDisplayPanel.java`,

from JM\Ch05\Poll into one project. Then fill in the blanks in the PollDisplayPanel, following these steps:

1. Add a declaration for three int fields, count1, count2, count3, which hold the current poll counts.

2. Implement the vote1, vote2, and vote3 methods, which increment the respective count.

3. Implement a toString method that returns a String containing the names of the candidates and their current vote counts. Something like this:

   ```
   Tami: 5  Brian: 7  Liz: 2
   ```

   Comment out the remaining blanks. Compile the PollDisplayPanel class and fix the syntax errors, if any.

4. Write a simple test class with a main method:

   ```
   public static void main(String[] args)
   {
     PollDisplayPanel votingMachine =
                   new PollDisplayPanel("Tami", "Brian", "Liz");
     votingMachine.vote1();
     votingMachine.vote2();
     votingMachine.vote2();
     System.out.println(votingMachine);
   }
   ```

   Compile and run it to test your progress so far.

   It should display

   ```
   Tami: 1  Brian: 2  Liz: 0
   ```

5. Implement the countToDegrees method that converts the ratio of its two integer parameters, count and total, into the angle measure, in degrees, of a corresponding pie chart slice, rounded to the nearest integer.

6. Fill in the blanks in the drawPieChart and drawLegend methods.

7. Compile and test the *Poll* program.

5.12 The Math Class

High-level programming languages usually include a library of functions for common mathematical calculations, such as returning the absolute value of a number, calculating the square root, raising a number to a given power, as well as trigonometric functions, logs and exponents, and so on. In Java, mathematical functions are implemented as static methods of the library class `Math`.

`Math.abs(x)` returns the absolute value of `x`. Actually, there are several versions of this method that work with different types of the parameter `x`. For example, if `x` is an `int`, `Math.abs(x)` returns an `int`; if `x` is a `double`, `Math.abs(x)` returns a `double`.

`Math.round(x)` returns `x` rounded to the nearest integer. If `x` is a `float`, `Math.round(x)` returns an `int`.

If `x` is a `double`, `Math.round(x)` returns a `long` type value.

`Math.pow(x, e)` calculates and returns the value of x^e. This method expects `x` and `e` of the `double` type, but it will automatically promote `int`s and `float`s to `double`s. `Math.pow` returns a `double`.

There is no separate method in the Java `Math` class (or, as far as we know, in any math library of any high-level or low-level language) that calculates the square of a number, because this operation can be easily programmed using one multiplication:

```
double xSquared = x*x;
```

You can use `Math.pow` for that, too, of course:

```
double xSquared = Math.pow(x, 2);
```

`Math.pow` can be used to calculate the square root of a number, too. For example:

```
double squareRootX = Math.pow(x, 0.5);
```

However, it is more common to use the dedicated method `sqrt` for that:

```
double squareRootX = Math.sqrt(x);
```

`sqrt` expects a `double` parameter and returns a `double`.

`Math.max(x, y)` and `Math.min(x, y)` methods return the largest and the smallest of `x` and `y`, respectively. If `x` and `y` are `int`s, these methods return an `int`; if `x` and `y` are `double`s (or at least one of them is a `double`) these methods return a `double`.

The `Math` class also provides a method `random` that returns a random number (a `double`) in the range from 0 (inclusive) to 1 (exclusive). We will use this method in the case study in Chapter 6.

5.13 Calling a Method from `main`

It is a common practice to separate computations from the user interface in a program. For example, suppose you need to test a method that computes something. You will write a one-class program, in which `main` will prompt the user to enter one or several numbers, call the method, passing the entered numbers to the method as parameters, then display the result returned from the method. Figure 5-3 gives an example of such a class.

Notice that in this scenario, the method does not print the result; it completes the calculation and returns the result to `main`. It is `main` that displays the result in a convenient format.

The data type of the result returned from the method must match the method's return type declared in its header. For example, in the *Gas Mileage* program, the `gasMileage` method is defined as `double`, and the result of the computation returned from the method is a `double` value.

Notice, also, that both `main` and the `gasMileage` method have the keyword `static` in their headers. We will explain the meaning of the `static` keyword later, in Section 10.11.

```
// This program prompts the user to enter miles traveled and
// gas spent and calculates the car's gas mileage

import java.util.Scanner;

public class GasMileage
{
  public static double gasMileage(int miles, double gallons)
  {
    return miles/gallons;
  }

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

    System.out.print("Miles traveled: ");
    int miles = input.nextInt();
    System.out.print("Gallons of gas spent: ");
    double gallons = input.nextDouble();

    input.close();

    double mpg = gasMileage(miles, gallons);
    System.out.printf(
         "Your car's gas mileage was %.1f miles per gallon\n", mpg);
  }
}
```

Figure 5-3. JM\Ch05\Exercises\GasMileage.java

5.14 Summary

Variables are memory locations, named by the programmer, that can hold values. *Fields* are variables declared outside of all constructors and methods of the class. Fields are "visible" in all the class's constructors and methods. *Local variables* are temporary variables declared inside a constructor or method and available only within the scope of that constructor or method.

Fields and local variables must be declared before they can be used. The declaration of a variable includes the data type of the variable, its name, and an optional initial value. Several variables of the same type may be declared in the same declaration:

```
[optional modifiers] sometype name1, name2, ...;
[optional modifiers] sometype name1 = expr1, name2 = expr2, ...;
```

Fields that never change their values are often declared with the keyword `final`, usually with initial values; they are actually not variables but constants. Declarations of such *symbolic constants* have the form

```
[optional modifiers] final sometype someName = expression;
```

where the *optional modifiers* can be the reserved words `public` or `private`, and, if appropriate, `static` (Chapter 10).

Java has `byte`, `int`, `short`, and `long` *primitive data types* for representing integers of various sizes. We will almost always use `int`, which represents integers in the range from -2^{31} to $2^{31}-1$. Real numbers are represented by the `float` and `double` types. We will always use `double`. `char` represents single characters. A `String` object represents a fragment of text. The `boolean` type is discussed in Chapter 6.

Arithmetic expressions are written the same way as in algebra and may include literal constants, variables, the *arithmetic operators* `+`, `-`, `*`, `/`, and `%`, and parentheses. A multiplication sign `*` cannot be omitted.

The result of an arithmetic operation has the same type as the operands. If the operands have different types, the operand of the "smaller" type is automatically promoted to the "larger" type (for example, an `int` may be converted to a `double`). Java provides a cast operator that explicitly converts a variable or constant from one data type into another, compatible type.

It is a programmer's responsibility to make sure the values of variables and all the intermediate and final results in arithmetic expressions fit within the range of the chosen data types, and that these types satisfy the precision requirements for computations.

The `+` operator, when applied to two strings, concatenates them. It can also be used to concatenate a string with a value of another type. To convert a number or an object into a string, concatenate it with an empty string. A `toString` method in a class definition should specify a reasonable way for representing some information about the object of that class as a string. When an object is converted into a string, its `toString` method is called.

The built-in Java library class `Math` provides common methods for arithmetic calculations: `abs`, `sqrt`, `pow`, `max` and `min`, `round`, and so on.

Exercises

Sections 5.1-5.6

1. Which of the following lines are syntactically valid declarations of fields? Of local variables?

(a) `int hours, double pay;` _____
(b) `private double dollarsAndCents;` _____ ✓
(c) `char mi; int age;` _____
(d) `private final int year = 365, leapYear = year + 1;` _____
(e) `char tab = '\t', newline = '\n', a = 'a';` _____
(f) `public final double pi = 3.14159;` _____

2. Mark true or false and explain:

(a) Each variable must be declared on a separate line. _____
(b) The scope of a variable is the largest range of its values. _____
(c) `k` is always a stylistically bad name for a variable because it is too short. _____
(d) Local variables in different methods of the same class are allowed to have the same name. _____ ✓
(e) If a local variable in a method and a field have the same name, the compiler reports a syntax error. _____ ✓

3. Name three good reasons for using symbolic constants as opposed to literal constants.

4. (MC) Which one of the following statements prints a backslash?

A. `System.out.print("\b");`
B. `System.out.print("\\");`
C. `System.out.print(\bs);`
D. `System.out.print(\);`
E. `System.out.print(\\);`

5. Choose the right word: the scope of a variable is determined when the program is _____ (*compiled* / *executed*). ✓

6. (MC) Which variables in a class have global scope access?

(A) Fields declared with the keyword `global`
(B) Public static fields only
(C) Private and public static fields only
(D) All fields

Sections 5.7-5.14

7. What is the output from the following statements?

(a) `System.out.print(5 / 10);` ✓
(b) `System.out.print(1 / 2 * 10);`
(c) `System.out.print(1.0 / 2 * 10);` ✓
(d) `System.out.print(1 / 2.0 * 10);`
(e) `System.out.print(13 % 5);`

8. (a) Assuming:

```
double rate = 1.058;
int balance0 = 100, balance = (int)(balance0 * rate);
```

what is the value of `balance`? ✓

(b) Assuming:

```
int miles = 98, gallons = 5;
double gasMileage = miles / gallons;
```

what is the value of `gasMileage`?

9. Remove as many parentheses as possible from the following statement without changing the result:

```
count += (((total/pages) - 5) * words - 1);
```

10. Find and fix a bug in the following statements:

```
final double g = 16.0;
double t = 35.5;
System.out.print ("The travel distance is ");
System.out.println(1 / 2 * (g * t * t));
```

11. If `double x` has a negative value, write an expression that rounds `x` to the nearest integer. (There is no unique rule on how to round numbers that are in the middle between two integers, such as −0.5, −1.5, −2.5, and so on. In this exercise round these numbers down to −1, −2, −3, and so on.)

12. Fill in the blanks in a short class `FeetToInches` with one static method `toInches` and `main` (`JM\Ch05\Exercises\FeetToInches.java`). `main` prompts the user to enter her height in feet and inches, calls `toInches`, and displays the returned height in inches. ✓

13. Given

```
int a, b, c;
```

write Java expressions that calculate the roots of the equation $ax^2 + bx + c = 0$ (assuming that the two real roots exist) and assign them to two `double` variables `x1` and `x2`. Use a temporary variable to hold $\sqrt{b^2 - 4ac}$ in order not to compute it twice. ⩽ Hint: `Math.sqrt(d)` returns a square root of `d` as a `double`. ⩾ ✓

14.■ Find a syntax error in the following code fragment:

```
double a, b;
int temp;

System.out.print("Enter two real numbers: ");
...
// Swap the numbers:
temp = a;
a = b;
b = temp;
...
```

✓

15.■ Write an expression that, given a positive integer n, computes a new integer in which the units and tens digits have swapped places. For example, if $n = 123$, the result should be 132; if $n = 3$, the tens digit is zero and the result should be 30.

16.■ An integer constant `dayOfWeek1` has a value from 0 to 6 and represents the day of the week for January 1st (0=Sunday, 1=Monday, etc.). A variable `day` has a value from 1 to 31 and represents a day in January. Write an expression that calculates the day of the week for any given value of `day`. For example, if `dayOfWeek1` = 0 (January 1st is a Sunday) and `day` = 13 (January 13th), then `dayOfWeek`, the day of the week for January 13th, should get a value of 5 (Friday).

17.■ Write and test the class `InflatableBalloon` as a subclass of `Balloon`, as described in Section 4.5 on pages 82-83. Supply a method

```
public void inflate(int percentage)
```

that adjusts the radius of the balloon appropriately when its volume changes by the given percentage. ≤ Hint: the volume of a sphere is proportional to its radius cubed. ≥ ✓

18. Rewrite the *Gas Mileage* (`JM\Ch05\Exercises\GasMileage.java`) to prompt the user to enter the gas mileage and the number of miles and display the amount of gas spent in that trip.

19. Write a short class `LunchTime` with no fields, no constructors, and just one static method `minutesUntilLunch`. The method should take two `int` parameters, current hours and minutes (before 1 p.m.), and return (not print!) the number of minutes left until lunch, scheduled for 1 p.m. Add a `main` method that prompts the user for the current time (in the format hh:mm), calls `minutesUntilLunch`, and prints the value returned by that call. Make sure your program displays the correct result when the user enters 9:00, 10:10, 12:00, 12:50.

≤ Hint: The following statements can be used to extract hours and minutes from an "hh:mm" string:

```
int i = s.indexOf(":");
int hours = Integer.parseInt(s.substring(0, i));
int mins = Integer.parseInt(s.substring(i+1));
```

≥

20.■ `curHour` and `curMin` represent the current time, and `depHour`, `depMin` represent the departure time of a bus. Suppose all these variables are initialized with some values; both times are between 1 p.m. and 11 p.m. of the same day. Fill in the blanks in the following statements that display the remaining waiting time in hours and minutes:

```
int _____________ =

     ____________________________________________ ;


System.out.println( ___________________________ +
  " hours and " +  ____________________________ +
  " minutes.");
```

21. The *BMI* program computes a person's body mass index (BMI). BMI is defined as the weight, expressed in kilograms, divided by the square of the height expressed in meters. (One inch is 0.0254 meters; one pound is 0.454 kilograms.) The code of the `Bmi` class , with some omissions, is in `JM\Ch05\Exercises\Bmi.java`. Supply the missing code for the `calculateBmi` method, which takes a weight in pounds and height in inches as parameters and returns the body mass index.

22.■ A jar of jam weighs 1 lb. 5 oz. (One pound is 16 ounces). An empty shipping carton weighs 1 lb. 9 oz. and can hold up to 12 jars. The shipping costs include $1.44 for each full or partial carton plus $0.96 per pound or fraction of a pound plus a $3.00 service charge.

Fill in the blanks in the following method that calculates the shipping cost for a given number of jars:

```
public double computeShippingCost(int nJars)
{
  int nCartons = (nJars + 11) / 12;

  int totalOunces = ____________________________ ;

  int lbs = ___________________________________ ;

  return ______________________________________ ;
}
```

23.■ Write a method

```
public int convertToHumanAge(int dogYears)
```

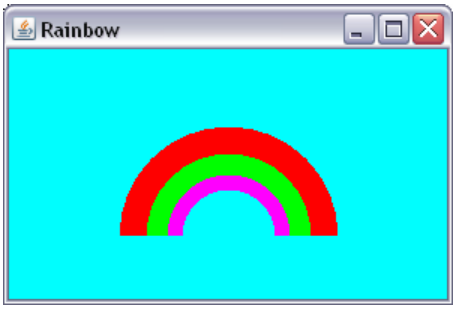
that converts a dog's age to the corresponding human age. Assume that a dog's first year corresponds to a human age of 13, so `convertToHumanAge(1)` should return 13. After that, every three years in a dog's life correspond to sixteen years in human life. The method returns the corresponding human age, rounded to the nearest integer. Write a console Java application to test your method (or, if you prefer, recycle the GUI from the *BMI* program in Question 21 into a dog-to-human-age converter). ✓

24. Using `Math`'s `max` method, write a one-line expression that assigns to the `double` variable `maxOf3` the largest of the `double` values `x`, `y`, `z`.

25. Write a method that returns the area of a triangle with the given lengths of its sides. Use Math's `sqrt` method. ≤ Hint: Search the Internet for Heron's formula. ≥

26. Research online the difference between the `float` and `double` data types in Java and the benefits of using one or the other. Which one is faster and under which circumstances? Which takes more memory space? What type of values are returned by such `Math` library methods as `sin`, `cos`, `sqrt`?

27.◆ The figure below shows a window from the *Rainbow* program.



The "rainbow" is made of four overlapping semicircles. The outer ring is red (`Color.RED`), the middle one is green (`Color.GREEN`), and the inner ring has the magenta color (`Color.MAGENTA`). The innermost semicircle has the same color as the background.

Follow the instructions below and fill in the blanks in `Rainbow.java` in `JM\Ch05\Exercises`.

1. Copy `Rainbow.java` to your work folder.

2. Add a comment with your name at the top of the file.

3. Add to the `Rainbow` class a declaration of a private final field `skyColor` of the type `Color`, initialized to `Color.CYAN` (the color of the sky). The `Rainbow`'s constructor sets the window's background to `skyColor`.

4. Make sure `Rainbow.java` compiles with no errors.

5. In the `paintComponent` method, declare local integer variables `xCenter` and `yCenter` that represent the coordinates of the center of the rings. Initialize them to 1/2 `width` and 3/4 `height` (down) of the panel, respectively. (Recall that the origin of graphics coordinates in Java is at the upper-left corner of the content pane with the *y*-axis pointing down.) Do not plug in fixed numbers from the window's dimensions.

6. Declare a local variable `largeRadius` that represents the radius of the largest (red) semicircle and initialize it to 1/4 of `width`.

7. A method call `g.fillArc(x, y, size, size, from, degrees)` (with all integer arguments) draws a sector of a circle. `x` and `y` are the coordinates of the upper-left corner of the rectangle (in this case a square) into which the oval is (logically) inscribed; `size` is the side of the square (and the diameter of the circle); `from` is the starting point of the arc in degrees (with 0 at the easternmost point of the horizontal diameter), and `degrees` (a positive number) is the measure of the arc, going counterclockwise. Add a statement to the `paintComponent` method to draw the largest (red) semicircle. Test your program.

Continued 

8. Add statements to display the medium (green) and small (magenta) semicircles. The radius of the magenta semicircle should be 1/4 of `height`. The radius of the green one should be the geometric mean (the square root of the product) of the radius of the red semicircle and the radius of the magenta semicircle, rounded to the nearest integer. (A call to `Math.sqrt(x)` returns the value of square root of `x`, a `double`.) Retest your program.

9. Add statements to display the innermost semicircle of the background ("sky") color to complete the rainbow. Use the `skyColor` constant for this semicircle's color. Choose the radius of the sky-color semicircle in such a way that the width of the middle (green) ring is the arithmetic mean of the widths of the red and magenta rings. ≤ Hint: you will need to do a little math to figure out a formula for the radius of the smallest semicircle in terms of the radii of the other three. ≥

10. Test your program.

```
if (chapter == 6)
```

Boolean Expressions and `if-else` Statements

6.1 Prologue

Normally control flows sequentially from one statement to the next during program execution. This sequence is altered by several types of control mechanisms:

1. Calls to methods
2. Iterative statements (loops)
3. Conditional (`if-else`) statements
4. `switch` statements
5. Exceptions

In this chapter we will study the `if-else` statement, which tells the program to choose and execute one fragment of code or another depending on some condition. We will also take a look at the `switch` statement, which chooses a particular fragment of code out of several based on the value of a variable or expression.

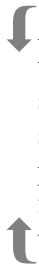
The `if-else` control structure allows *conditional branching*. Suppose, for instance, that we want to find the absolute value of an integer. The method that returns an absolute value may look as follows:

```
public static int abs(int x)
{
  int ax;

  if (x >= 0)     // If x is greater or equal to 0
    ax = x;       //   do this;
  else            // otherwise
    ax = -x;      //   do this.
  return ax;
}
```

Or, more concisely:

```
public static int abs(int x)
{
  if (x >= 0)
    return x;
  else
    return -x;
}
```

There are special CPU instructions called *conditional jumps* that support conditional branching. The CPU always fetches the address of the next instruction from a special register, which in some systems is called the Instruction Pointer (IP). Normally, this register is incremented automatically after each instruction is executed so that it points to the next instruction. This makes the program execute consecutive instructions in order.



A conditional jump instruction tests a certain condition and tells the CPU to "jump" to a specified instruction depending on the result of the test. If the tested condition is satisfied, a new value is placed into the IP, which causes the program to skip to the specified instruction. For example, an instruction may test whether the result of the previous operation is greater than zero, and, if it is, tell the CPU to jump backward or forward to a specified address. If the condition is false, program execution continues with the next consecutive instruction.

In high-level languages, conditions for branching are written using *relational operators* such as "less than," "greater than," "equal to," and so on, and the *logical operators* "and," "or," and "not." Expressions combining these operators are called *Boolean* expressions. The value of a Boolean expression may be either true or false.

In the following sections we will discuss the syntax for coding `if-else` and `switch` statements, declaring `boolean` variables, and writing Boolean expressions with relational and logical operators. We will also briefly discuss two properties of formal logic, known as *De Morgan's Laws*, that are useful in programming. We will talk about *short-circuit evaluation* in handling multiple conditions connected with "and" and "or" operators. We also discuss enumerated data types.

In Sections 6.9 and 6.12 we use a case study to practice object-oriented design and implementation methodology: how to define the classes and objects needed in an application, how to divide work among team members, and how to test parts of a project independently from other parts. You will have to contribute code with a lot of `if-else` statements in it for one of the classes in this case study.

6.2 if-else Statements

The general form of the `if-else` statement in Java is:

```
if (condition)
{
  statementA1;
  statementA2;
  ...
}
else
{
  statementB1;
  statementB2;
  ...
}
```

where *condition* is a logical expression. The parentheses around *condition* are required. When an `if-else` statement is executed, the program evaluates the condition and then executes *statementA1*, etc. if the condition is true, and *statementB1*, etc. if the condition is false. If the compound block within braces consists of only one statement, then the braces can be dropped:

```
if (condition)
  statementA;
else
  statementB;
```

The `else` clause is optional, so the `if` statement can be used by itself:

```
if (condition)
{
  statement1;
  statement2;
  ...
}
```

When `if` is coded without `else`, the program evaluates the condition and executes *statement1*, etc. if the condition is true. If the condition is false, the program simply skips the block of statements under `if`.

6.3 boolean Data Type

Java has a primitive data type called `boolean`. `boolean` variables can hold only one of two values: `true` or `false`. `boolean`, `true`, and `false` are Java reserved words. You declare `boolean` variables like this:

```
boolean aVar;
```

There is not much sense in declaring `boolean` constants because you can just use `true` or `false`.

Boolean expressions are made up of `boolean` variables, relational operators, such as >=, and logical operators. You can assign the value of any Boolean expression to a `boolean` variable. For example:

```
boolean over21 = age > 21;  // or boolean over21 = (age > 21);
```

Here `over21` gets a value of `true` if `age` is greater than 21, `false` otherwise. This is essentially a more concise version of

```
int age = < some value >;
...

boolean over21;

if (age > 21)
  over21 = true;
else
  over21 = false;
```

6.4 Relational Operators

Java recognizes six relational operators:

Operator	Meaning
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
==	is equal to
!=	is not equal to

The result of a relational operator has the `boolean` type. It has a value equal to `true` if the comparison is true, `false` otherwise.

Relational operators are frequently used in conditions. For example:

```
if (x > y)
  max = x;
else
  max = y;
```

Note that in Java the "is equal to" condition is expressed by the == operator, while a single = sign means assignment. Be careful not to confuse the two.

Relational operators are applied mostly to variables of primitive numeric data types. The == and != operators can also be applied to characters. For example:

```
if (gender == 'F')
{
  System.out.print("Dear Ms. ");
}
else
{
  System.out.print("Dear Mr. ");
}
```

Avoid using == and != for `double` or `float` variables and expressions because floating-point arithmetic is imprecise. For example, in

```
double x = 15.0/11.0;
System.out.println(11.0*x == 15.0);
```

the output will be `false`.

If you apply the == and != operators to objects, then instead of comparing the values of two objects you will be comparing two references to them (that is, their addresses). This is a potential source of bugs.

For example, the == operator in

```
String fileName;
...
if (fileName == "words.txt")
  ...
```

compares the addresses of the `String` object `fileName` and the `String` object that represents a literal string `"words.txt"`. Their addresses are most likely different, even though their current values may be the same. You have to use `String`'s `equals` method to compare string values, as in

```
if (fileName.equals("words.txt")) ...
```

or

```
if ("words.txt".equals(fileName)) ...
```

More on this in Chapter 8. However, occasionally it is useful to compare references to objects, for example if you want to know which particular object (for example, a button) caused a particular event.

6.5 Logical Operators

Java has two binary logical operators, "and" and "or," and a unary logical operator, "not." They are represented by the following symbols:

Operator	Meaning
`&&`	and
`||`	or
`!`	not

The expression

condition1* && *condition2

is true if and only if both *condition1* and *condition2* are true.

The expression

condition1* || *condition2

is true if *condition1* or *condition2* (or both) are true.

The expression

!*condition1*

is true if and only if *condition1* is false.

The following code:

```
boolean match = ...;
if (!match)
{
  ...
}
```

works the same way as:

```
boolean match = ...;
if (match == false)
{
  ...
}
```

The results of the logical operators `&&`, `||`, and `!` have the `boolean` data type, just like the results of relational operators.

The "and," "or," and "not" operations are related to each other in the following way:

Boolean expression	Has the same Boolean value as:
not (*p* and *q*)	not *p* or not *q*
not (*p* or *q*)	not *p* and not *q*

For example, "not (fun and games)" is the same as "not fun or not games."

These two properties of logic are called *De Morgan's Laws*. They come from formal logic, but they are useful in practical programming as well. In Java notation, De Morgan's Laws take the following form:

`!(p && q)` is the same as `!p || !q`
`!(p || q)` is the same as `!p && !q`

A programmer may choose either of the equivalent forms; the choice depends on which form is more readable. Usually it is better to distribute the `!` ("not"). For example:

```
if (x >= 0 && x < 5)
```

is much easier to read than:

```
if (!(x < 0 || x >= 5))
```

6.6 Order of Operators

In general, all unary operators have higher precedence than binary operators, so unary operators, including ! ("not"), are applied first. You have to use parentheses if you want to apply ! to the entire expression. For example:

```
if (!cond1 && cond2)
```

means

```
if ((!cond1) && cond2)
```

rather than

```
if (!(cond1 && cond2))
```

All binary arithmetic operators (+, *, etc.) have higher rank than all relational operators (>, < , ==, etc.), so arithmetic operators are applied first. For example, you can write simply:

```
if (a + b >= 2 * n)              // OK!
```

when you mean:

```
if ((a + b) >= (2 * n))          // The inside parentheses are
                                 //   optional
```

Arithmetic and relational operators have higher rank than the binary logical operators && and ||, so arithmetic and relational operators are applied first. For example, you can write simply:

```
if (x + y > 0 && b != 0)         // OK!
```

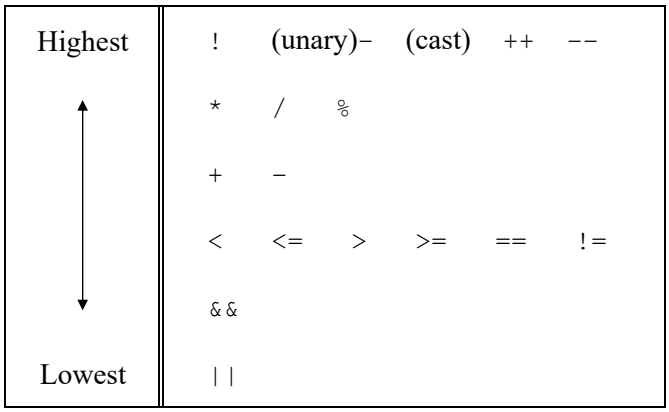
instead of:

```
if ((x + y > 0) && (b != 0))     // The inside parentheses are
                                 //   optional
```

When && and || operators are combined in one logical expression, && has higher rank than || (that is, && is performed before ||), but with these it is a good idea to always use parentheses to avoid confusion and make the code more readable. For example:

```
// Inside parentheses not required, but recommended for clarity:
if ((x > 2 && y > 5) || (x < -2 && y < -5))
{
  ...
}
```

The rules of precedence for the operators we have encountered so far are summarized in the table below:



In the absence of parentheses, binary operators of the same rank are performed left to right, and unary operators right to left.

For example, `(double)(int)x` is the same as `(double)((int)x)`. If in doubt — use parentheses!

6.7 Short-Circuit Evaluation

In `&&` and `||` operations, the left operand is always evaluated first. There may be situations when its value predetermines the result of the operation. For example, if *`condition1`* is false, then the value of the expression *`condition1 && condition2`* is false, no matter what the value of *`condition2`* is. If *`condition1`* is true, then *`condition1 || condition2`* is true.

If the value of the first (left) operand in a binary logical operation is sufficient to determine the result of the operation, the second operand is <u>not</u> evaluated. This rule is called *short-circuit evaluation*.

If the expression combines several && operations at the same level, such as

```
condition1 && condition2 && condition3 ...
```

the evaluation of conditions proceeds from left to right. If a false condition is encountered, then the remaining conditions are not evaluated, because the value of the entire expression is false. Similarly, if the expression combines several || operations at the same level,

```
condition1 || condition2 || condition3 ...
```

the evaluation proceeds from left to right only until a true condition is encountered, because then the value of the entire expression is true.

The short-circuit evaluation rule not only saves program execution time but is also convenient in some situations. For example, it is safe to write:

```
if (y != 0 && x/y > 3)
  ...
```

because x/y is not calculated when y is equal to 0.

❖ ❖ ❖

Java also provides bit-wise "and" and "or" operators that normally work on integers and operate on individual bits. These operators are denoted as & and | (as opposed to && and ||). Unfortunately these operators also work on booleans, and they do not follow the short-circuit evaluation rule. This is really confusing and may lead to a nasty bug, if you inadvertently write & instead of && or | instead of ||. Make sure you use && and || unless you are indeed working with individual bits. Bit-wise operators are explained in Chapter 18.

6.8 if-else-if and Nested if-else

Sometimes a program needs to branch three or more ways. Consider the *sign*(*x*) function:

$$sign(x) = \begin{cases} -1, \text{ if } x < 0 \\ 0, \text{ if } x = 0 \\ 1, \text{ if } x > 0 \end{cases}$$

The sign(x) method can be implemented in Java as follows:

```
public static int sign(int x)    // Correct but clumsy code...
{
  int s;

  if (x < 0)
    s = -1;
  else
  {
    if (x == 0)
      s = 0;
    else
      s = 1;
  }
  return s;
}
```

This code is correct, but it looks cumbersome. The x < 0 case seems arbitrarily singled out and placed at a higher level than the x == 0 and x > 0 cases. Actually, the braces in the outer else can be removed, because the inner if-else is one complete statement. Without braces, the compiler always associates an else with the nearest if above it. The simplified code without braces looks as follows:

```
public static int sign(int x)   // Correct but still clumsy...
{
  int s;

  if (x < 0)
    s = -1;
  else
    if (x == 0)
      s = 0;
    else
      s = 1;
  return s;
}
```

It is customary in such situations to arrange the statements differently: the second if is placed next to the first else and one level of indentation is removed, as follows:

```
public static int sign(int x)   // The way it should be...
{
  int s;

  if (x < 0)
    s = -1;
  else if (x == 0)   // This arrangement of if-else is a matter
    s = 0;           //   of style: structurally, the second
  else               //   if-else is still nested within the
    s = 1;           //   first else
  return s;
}
```

This format emphasizes the three-way branching that conceptually occurs at the same level in the program, even though technically the second `if-else` is *nested* in the first `else`.

A chain of `if-else-if` statements may be as long as necessary:

```
if (condition1)
{
  ...                  // 1st case
}
else if (condition2)
{
  ...                  // 2d case
}
else if (condition3)
{
  ...                  // 3d case
}

...
...

else if (conditionN)
{
  ...                  // N-th case
}
else  // the last "else" clause may be omitted
{
  ...                  // otherwise
}
```

This is a rather common structure in Java programs and usually quite readable. For example:

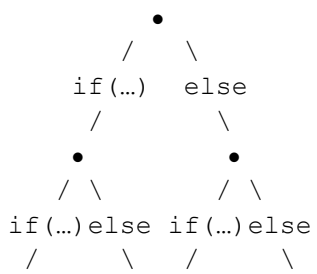
```
if (avg >= 90)
  grade = 'A';
else if (avg >= 80)
  grade = 'B';
else if (avg >= 70)
  grade = 'C';
else if (avg >= 60)
  grade = 'D';
else
  grade = 'F';
```

Or:

```
if (x < lowerLimit)
{
  x = lowerLimit;
}
else if (x > upperLimit)
{
  x = upperLimit;
}
```

❖ ❖ ❖

A different situation occurs when a program requires true hierarchical branching with nested `if-else` statements, as in a decision tree:



Consider, for example, the following code:

```
// Surcharge calculation:
if (age <= 25)
{
  if (accidents)
    surcharge = 1.4;  // Premium surcharge 40%
  else
    surcharge = 1.2;  // Surcharge 20%
}
else  // age > 25
{
  if (accidents)
    surcharge = 1.1;  // Surcharge 10%
  else
    surcharge = 0.9;   // Discount 10%
}
```

Here the use of nested `if-else` statements is justified by the logic of the task.

When `if-else` statements are nested in your code to three or four levels, the code becomes too complicated. This indicates that you probably need to restructure your code, perhaps using separate methods to handle individual cases.

Nested `if`s can often be substituted with the `&&` operation:

```
if (condition1)
  if (condition2)
    statement;
```

is exactly the same (due to short-circuit evaluation) as

```
if (condition1 && condition2)
  statement;
```

but the latter form is usually clearer.

Beware of the “dangling else” bug in nested `if-else` statements:

```
if (condition1)     // Compiled as: if (condition1)
  if (condition2)   //              {
    statement1;     //                if (condition2)
else                //                  statement1;
  statement2;       //                else
                    //                  statement2;
                    //              }
```

6.9 *Case Study and Lab:* Rolling Dice

In this section we will implement the *Craps* program. Craps is a game played with dice. In Craps, each die is a cube with numbers from 1 to 6 on its faces. The numbers are represented by dots (Figure 6-1).

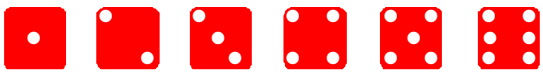


Figure 6-1. Dots configurations on a die

A player rolls two dice and adds the numbers of dots shown on them. If the total is 7 or 11, the player wins; if the total is 2, 3 or 12, the player loses. If the total is anything else, the player has to roll again. The total, called the "point," is remembered, and the objective now is to roll the same total as the "point." The player keeps rolling until he gets either "point" or 7. If he rolls "point" first, he wins, but if he rolls a 7 first, he loses. You can see why this game was chosen as a lab for `if-else` statements!

Our team has been asked to design and code a *Craps* program for our company's "Casino Night" charitable event. Three people will be working on this project. I am the project leader, responsible for the overall design and dividing the work between us. I will also help team members with detailed design and work on my own piece of code. The second person, Aisha, is a consultant; she specializes in GUI design and implementation.

The third person is you!

Run the executable *Craps* program by clicking on the `craps.jar` file in `JM\Ch06\Craps`. When you click on the "Roll" button, red dice start rolling on a green "table." When they stop, the score is updated or the "point" is shown on the display panel (Figure 6-2). The program allows you to play as many games as you want.

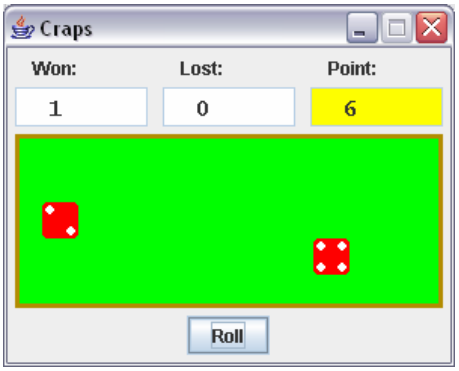


Figure 6-2. The *Craps* program

We begin the design phase by discussing which objects are needed for this application. One approach may be to try making objects in the program represent objects from the real world. Unfortunately, it is not always clear what exactly is a "real world" object. Some objects may simulate tangible machines or mechanisms, others may exist only in "cyberspace," and still others may be quite abstract and exist only in the designer's imagination.

Here we need one object that represents the program's window. Let us call this object `window` and its class `Craps`. As usual, we will derive this class from the `JFrame` class in Java's *Swing* package. The window (Figure 6-2) is divided into three "panels." The top panel displays the score and the current state of the game. Let's call it `display` and its class `DisplayPanel`. The middle panel represents the Craps table where the dice roll. Let's call it `table` and its class `CrapsTable`. The bottom panel holds the "Roll" button. Let's call it `controls` and its class `ControlPanel`. The control panel can also handle the "Roll" button's click events.

It makes sense that each of the `DisplayPanel`, the `CrapsTable`, and the `ControlPanel` classes extend the Java library class `JPanel`. For example:

```
public class DisplayPanel extends JPanel
{
  ...
}
```

The `table` object shows two "rolling dice," so we need a class that will represent a rolling die. Let's call it `RollingDie`.

These five classes, `Craps`, `DisplayPanel`, `CrapsTable`, `ControlPanel`, and `RollingDie`, form the GUI part of our *Craps* program (Figure 6-3).

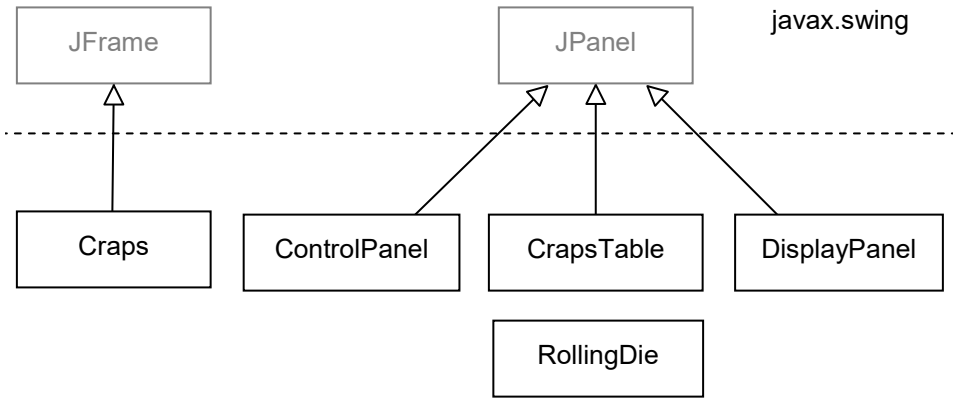
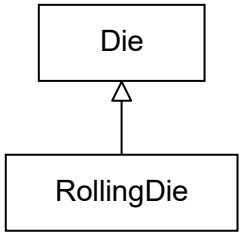


Figure 6-3. GUI classes in the *Craps* program

It makes sense to me to split the code for the visible and "numeric" aspects of a rolling die into two classes. The base class `Die` will represent a die as an abstract device that generates a random integer in the range from 1 to 6. The class `RollingDie` will extend `Die`, adding methods for moving and drawing the die:



My rationale for this design decision is that we might reuse the `Die` class in another program, but the dice there might have a different appearance (or may remain invisible).

Last but not least, we need an "object" that will represent the logic and rules of Craps. This is a "conceptual" object, not something that can be touched. Of course that won't prevent us from implementing it in Java. Let's call this object `game` and its class `CrapsGame`. The `CrapsGame` class won't be derived from anything (except the default, `Object`), won't use any Java packages, and won't process any events.

There are many good reasons for separating the rules of the game from the GUI part. First, we might need to change the GUI (if our boss doesn't like its "look and feel") while leaving the game alone. Second, we can reuse the `CrapsGame` class in other applications. For example, we might use it in a statistical simulation of Craps that runs through the game many times quickly and doesn't need a fancy GUI at all. Third, we might have a future need for a program that implements a similar-looking dice game but with different rules. Fourth, Aisha and I know only the general concept of the game and are not really interested in learning the details. And finally, it is a natural division of labor. We have a beginner on our team (you) and we have to give you a manageable piece of work.

Now we need to decide how the objects interact with each other. Figure 6-4 shows the overall design for the *Craps* program that I have come up with.

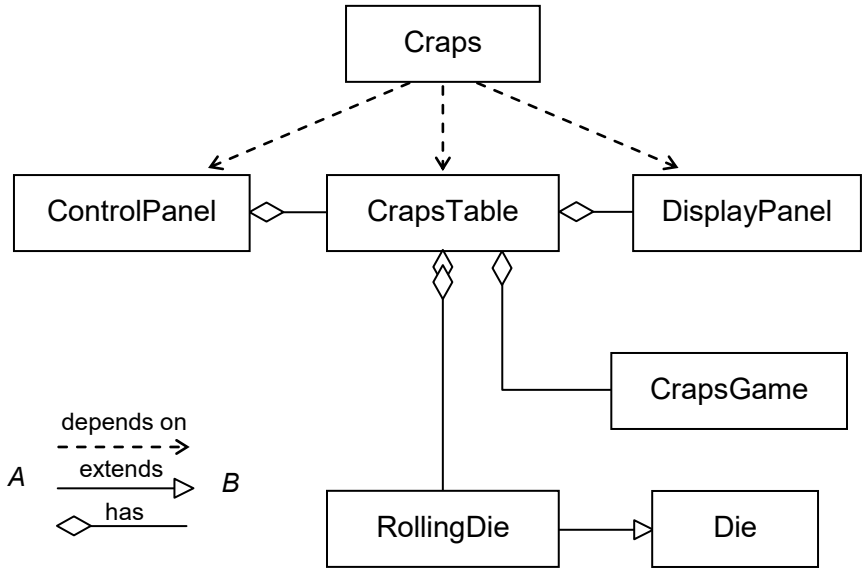


Figure 6-4. *Craps* classes and their relationships

There is no unique way, of course, of designing an application — a designer has a lot of freedom. But it is very helpful to follow some established *design patterns* and tested practices. Here we want to emphasize two principles of sound design.

First, each class must represent a single concept, and all its constructors and public methods should be related to that concept. This principle is called *cohesion*.

In a good design, classes are cohesive.

Second, dependencies between classes should be minimized. The reason we can draw a class diagram in Figure 6-4 without lines crisscrossing each other in all directions is that not all the classes depend on each other. OO designers use the term *coupling* to describe the degree of dependency between classes.

In a good design, coupling should be minimized.

It is good when a class interacts with only few other classes and knows as little about them as possible. Low coupling makes it easier to split the work between programmers and to make changes to the code.

In our *Craps* program, for example, the `ControlPanel` class and the `DisplayPanel` class do not need to know about each other's existence at all. `ControlPanel` knows something about `CrapsTable` — after all, it needs to know what it controls. But `ControlPanel` knows about only a couple of simple methods from `CrapsTable`.

A reference to a `CrapsTable` object is passed to `ControlPanel`'s constructor, which saves it in its field `table`. The `ControlPanel` object calls `table`'s methods when the "roll" button is clicked:

```
// Called when the roll button is clicked
public void actionPerformed(ActionEvent e)
{
  if (!table.diceAreRolling())  // if dice are not rolling,
    table.rollDice();           //   start a new roll
}
```

Likewise, `table` has a reference to `display`, but it knows about only one of its methods, `update`. When the dice stop rolling, `table` consults `game` (the only class that knows the rules of the game) about the result of the roll and passes that result (and the resulting value of "point") to `DisplayPanel`'s `update` method:

```
display.update(result, point);
```

The `Craps` object creates a `ControlPanel`, a `DisplayPanel`, and a `CrapsTable` in its constructor —

```
public class Craps extends JFrame
{
  // Constructor
  public Craps()
  {
    ...
    DisplayPanel display = new DisplayPanel();
    CrapsTable table = new CrapsTable(display);
    ControlPanel controls = new ControlPanel(table);
    ...
  }
  ...
}
```

— so it knows how to invoke their respective constructors. But it does not know about any of the methods of other classes, nor does it know anything about the rules of the Craps game or dice in general. In fact, if we change `Craps` to, say, `Soccer` and replace `CrapsTable` with `SoccerField`, the same code can be used for a different game.

We are now ready to divide the work among the three of us. Aisha will do the `Craps`, `ControlPanel`, and `DisplayPanel` classes. I like animations, so I will work on the `CrapsTable` and `RollingDie` classes myself. You get the `Die` and `CrapsGame` classes. Figure 6-5 shows how we split the work.

Aisha and I have already agreed on how the GUI classes will interact with each other. But we still need to nail down the details for your `Die` class and the `CrapsGame` class.

From your `Die` class I need two methods:

```
public void roll() { ... }
```

and

```
public int getNumDots() { ... }
```

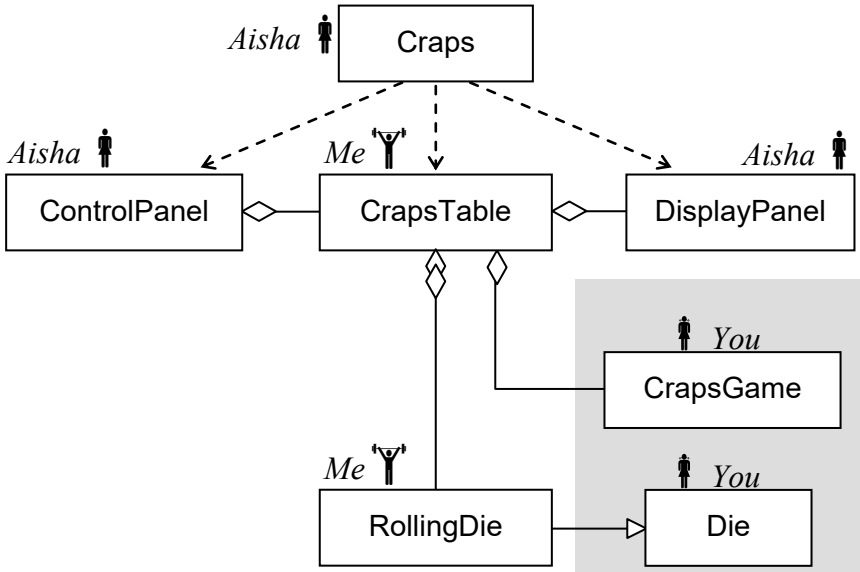


Figure 6-5. Task assignments in the *Craps* project

❖ ❖ ❖

I will call these methods from `RollingDie`. The `roll` method simulates one roll of a die. It obtains a random integer in the range from 1 to 6 and saves it in a field. The `getNumDots` method returns the saved value from that field. Do not define any constructors in `Die`: the default no-args constructor will do. To get a random number, use a call to `Math.random()`. This method returns a "random" `double` x, such that $0 \le x < 1$. Scale that number appropriately, then truncate it to an integer.

Now the `CrapsGame` class. My `CrapsTable` object creates a `CrapsGame` object called `game`:

```
private CrapsGame game;
...

// Constructor
public CrapsTable(DisplayPanel displ)
{
  ...
  game = new CrapsGame();
  ...
}
```

Again, no need to define a constructor in `CrapsGame`: we will rely on the default no-args constructor.

My `CrapsTable` object calls `game`'s methods:

```
int result = game.processRoll(total);
int point = game.getPoint();
```

The `processRoll` method takes one `int` parameter — the sum of the dots on the two dice. `processRoll` should process that information and return the result of the roll: 1 if the player wins, −1 if he loses, and 0 if the game continues. In the latter case, the value of "point" is set equal to `total`. Define a private `int` field `point` to hold that value. If the current game is over, `point` should be set to 0. `getPoint` is an accessor method in your `CrapsGame` class. It lets me get the value of `point`, so that I can pass it on to `display`.

We are ready to start the work. The only problem is the time frame. Aisha's completion date is unpredictable: she is very busy, but once she gets to work she works very fast. My task can be rather time-consuming. I will try to arrange a field trip to Las Vegas to film some video footage of rolling dice. But most likely our boss won't approve that, and I'll have to settle for observing rolling dice on the carpet in my office. Meanwhile you are anxious to start your part.

Fortunately, Aisha has found an old test program to which you can feed integers as input. She added a few lines to make it call `processRoll` and `getPoint` and display their return values (Figure 6-6). She called her temporary class `CrapsTest1`. Now you don't have to wait for us: you can implement and test your `CrapsGame` class independently. You won't see any dice rolling for now, but you will be able to test your class thoroughly in a predictable setting.

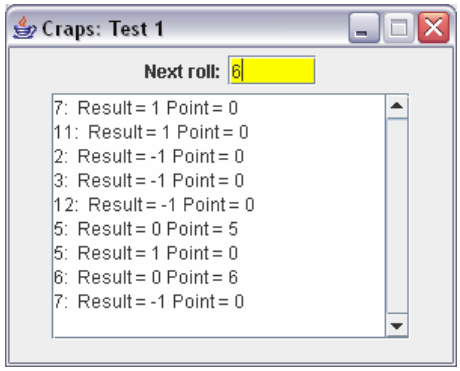


Figure 6-6. A preliminary program for testing `CrapsGame`

1. Copy `CrapsTest1.java` and `CrapsGame.java` from `JM\Ch06\Craps` to your work folder. Fill in the blanks in the `CrapsGame` class, combine it with the `CrapsTest1` class into one project, compile the classes, and test the program thoroughly.

2. Write the `Die` class and a small console application to test it by printing out the results of several "rolls." For example:

```
public static void main(String[] args)
{
  Die die = new Die();
  die.roll();
  System.out.println(die.getNumDots());
  die.roll();
  ...
}
```

3. After you get the `CrapsGame` and `Die` classes to work, test them with the *CrapsStats* application, which quickly runs the game multiple times and counts the number of wins. You will find `CrapsStats.java` in `JM\Ch06\Craps`. Note how we reuse for this task the `CrapsGame` and the `Die` classes that you have written for a different program.

   Compare your simulation result with the theoretical probability of winning in Craps, which is 244/495, or about 0.493. If you run 10,000 trial games, the number of wins should be somewhere between 4830 and 5030; 4930 on average.

6.10 The `switch` Statement

There are situations when a program must take one of several actions depending on the value of some variable or expression. If the program has to handle just two or three possible actions, you can easily use `if-else-if` statements:

```
int x = expression;

if (x == valueA)
{
  // Take action A
  statementA1;
  statementA2;
  ...
}
else if (x == valueB)
{
  // Take action B
  statementB1;
  ...
}

...
...

else if (x == valueZ)
{
  // Take action Z
  statementZ1;
  ...
}
else
{
  // Take some default action
  ...
}
```

(*valueA*, *valueB*, ..., and *valueZ* are integer constants.)

When the number of possible actions is large, the use of `if-else-if` becomes cumbersome and inefficient. Java provides a special mechanism, the `switch` statement, for handling such situations. Its general form is:

```
switch (expression)
{
  case valueA:         // Take action A
    statementA1;
    statementA2;
    ...
    break;

  case valueB:         // Take action B
    statementB1;
    ...
    break;

  ...
  ...

  case valueZ:         // Take action Z
    statementZ1;
    ...
    break;

  default:             // Take some default action
    ...
    break;
}
```

valueA, *valueB*, ... , *valueZ* in a switch are integer or character literal or symbolic constants (or `enum` type constants — see Section 6.11). Starting with Java 7, the case labels in a `switch` statement can also be literal strings (text in double quotes). For example:

```
  case "Sunday":
    ...
    break;

  case "Monday":
    ...
    break;
```

The `break` statement at the end of a case tells the program to jump out of the switch and continue with the first statement after the switch. `switch`, `case`, `default`, and `break` are Java reserved words.

Note the following properties of the `switch` statement:

1. The expression evaluated in a `switch` must have an integral or string type (`int` or `char` or `String`). In most programs it is really not an expression but simply one variable, as in `switch(k)`.
2. Each `case` must be labeled by a literal or symbolic constant. A case cannot be labeled by a variable or an expression that is not constant.
3. The same action may be activated by more than one case label. For example:

```
case '/':        // both '/' and ':' signify division
case ':':
  < ... statements >
  break;
```

4. There may be a `break` in the middle of a case, but then it must be inside an `if` or `else`, otherwise some code in that case would be unreachable. Such a `break` tells the program to jump out of the switch immediately. For example:

```
case '/':
  ...
  if (y == 0)
  {
    System.out.println("*** Division by zero ***\n");
    break;    // Jump out of the switch
  }

  < ... other statements >

  break;        // Jump out of the switch
```

5. The `default` clause is optional. If not specified, the default action is "do nothing."

It is a common mistake to omit `break` at the end of a case. The `switch` syntax does not require that each case end with a `break`. Without a `break`, though, the program falls through and continues with the next case.

This feature may lead to annoying bugs, and programmers usually take special care to put a `break` at the end of each case. Unusual situations, where a programmer intentionally allows the program to "fall through" from one case to the next, call for a comment in the code.

6.11 Enumerated Data Types

An enumerated data type defines a list of symbolic values. For example:

```
private enum State {READY, SET, GO};
```

`enum` is a Java reserved word. In the above example, `State` is the name of the enum data type, given by the programmer, and `READY`, `SET`, and `GO` are the symbolic values in the enum list, also defined by the programmer. The name of the `enum` type usually starts with a capital letter; the enum values are often spelled in all caps to make them stand out more.

An enum definition is placed inside a class outside of any constructor or method. The keyword `private` in the enum definition indicates that this enum type is visible only within its class.

Once an enum type is defined, we can declare variables of that type and assign values to them. For example, we can declare a variable of the type `State` and assign to it one of the symbolic values from the enum list:

```
State currentState = State.READY;
```

Variables and constants of the type `State` can only hold one of the three values: `State.READY`, `State.SET`, or `State.GO`. We use the "type-dot" notation to refer to the symbolic values from the enum list.

The values in an enum list are defined only by their symbols. They are not literal strings and they have no numeric values.

We are not really interested in how the enum values are represented internally in Java. (If you really want to know, they are sort of like objects; they even have a `toString` method.) As far as we are concerned, the value of `State.READY` is just `State.READY`.

Since enum values cannot be used in arithmetic and do not represent characters or strings, what do we need them for? There are often situations in programs when an object's attribute or state can have only one of a small number of values. It makes sense to define an enum type for variables that describe such an attribute or state. For example:

```
private enum Speed {LOW, MEDIUM, HIGH};
private enum BoardColor {BLACK, WHITE};
private enum DayOfWeek {sunday, monday, tuesday,
                wednesday, thursday, friday, saturday};
```

Of course we could instead represent such values as strings or integers or symbolic constants of some type. But an enum type is more compact and convenient than several declarations of symbolic constants. It is also safer: the compiler won't let us accidentally refer to a value that is not in the enum list.

In `boolean` expressions, variables of an enum type are compared to each other or to values from the enum list using the == and `!=` operators. For example:

```
if (currentState == State.GO)
  ...
```

An enum variable and symbolic values can be also used in a `switch` statement. For example:

```
switch(currentState)
{
  case READY:
    ...
    break;

  case SET:
    ...
    break;

  case GO:
    ...
    break;
}
```

(The type-dot prefix is not used in case labels within a switch.)

You can also define a public enum type. If a public enum type `SomeEnum` is defined within a class `MyClass`, then outside of `MyClass` you refer to that enum type as `MyClass.SomeEnum`, and refer to a `SomeEnum`'s symbolic value as `MyClass.SomeEnum.`*`symbolicValue`*. For instance:

```
public class Exam
{
  public enum LetterGrade {A, B, C, D, F};
  ...
}

public class Student
{
  private Exam.LetterGrade myGrade;
  ...
    if (myScore > 90)
      myGrade = Exam.LetterGrade.A;
    ...
}
```

6.12 *Case Study and Lab:* Rolling Dice Concluded

By this time you have finished your `CrapsGame` and `Die` classes and Aisha has found the time to put together her GUI classes. I myself have gotten bogged down with my `CrapsTable` and `RollingDie` classes, trying to perfect the animation effects. Meanwhile, not to stall Aisha's testing, I have written a *stub class* `CrapsTable` (Figure 6-7) to provide a temporary substitute for the actual class I am working on. A stub class has very simple versions of methods needed for testing other classes. This is a common technique when a programmer needs to test a part of the project while other parts are not yet ready.

My stub class includes a temporary version of the `rollDice` method that simply calls `game`'s `processRoll` method with a random sum of points and a version of `diceAreRolling` that always returns `false`.

```
// A temporary stub class for testing Craps.

public class CrapsTable
{
  private DisplayPanel display;
  private CrapsGame game;
  private Die die1, die2;

  public CrapsTable(DisplayPanel displ)   // constructor
  {
    display = displ;
    game = new CrapsGame();
    die1 = new RollingDie();
    die2 = new RollingDie();
  }

  public void rollDice()
  {
    die1.roll();
    die2.roll();
    int totalPoints = die1.getNumDots() + die2.getNumDots();
    int result = game.processRoll(totalPoints);
    int point = game.getPoint();
    display.update(result, point);
  }

  public boolean diceAreRolling()
  {
    return false;
  }
```

Figure 6-7. Temporary "stub" class `CrapsTable.java`

You're certainly welcome to take a look at Aisha's GUI implementation (in `JM\Ch06\Craps\src.zip`), but no one has time right now to explain to you how it works.

❖ ❖ ❖

Since you are done with your part, I thought you could help me out with my `RollingDie` class. I've made a lot of progress on it, but a couple of details remain unfinished.

I have coded the constructor, the `roll` method that starts the die rolling, and the `avoidCollision` method that keeps one die from overlapping with another. I have

also provided the `boolean` method `isRolling`, which tells whether my die is moving or not. But I am still working on drawing a rolling and a stopped die. I took what is called *top-down* approach with *step-wise refinement*, moving from more general to more specific tasks. First I coded the `draw` method in general terms:

```
// Draws this die, rolling or stopped;
// also moves this die, when rolling
public void draw(Graphics g)
{
  if (xCenter < 0 || yCenter < 0)
    return;
  else if (isRolling())
  {
    move();
    drawRolling(g);
    xSpeed *= slowdown;
    ySpeed *= slowdown;
  }
  else
  {
    drawStopped(g);
  }
}
```

Note how I used the `if-else-if` structure to process three situations: my die is off the table, it is still moving, or it is stopped.

My `draw` method calls the more specialized methods `drawRolling` and `drawStopped`. I am still working on these, but I know that each of them will call an even lower-level method `drawDots` that will draw white dots on my die:

```
// Draws this die when rolling with a random number of dots
private void drawRolling(Graphics g)
{
  ...
  Die die = new Die();
  die.roll();
  drawDots(g, x, y, die.getNumDots());
}

// Draws this die when stopped
private void drawStopped(Graphics g)
{
  ...
  drawDots(g, x, y, getNumDots());
}
```

I have started drawDots (Figure 6-8) and am counting on you to finish it. (Naturally, it involves a switch statement.) Meanwhile I will finish CrapsTable, and we should be able to put it all together.

```
// Draws a given number of dots on this die
private void drawDots(Graphics g, int x, int y, int numDots)
{
  g.setColor(Color.WHITE);

  int dotSize = dieSize / 4;
  int step = dieSize / 8;
  int x1 = x + step - 1;
  int x2 = x + 3*step;
  int x3 = x + 5*step + 1;
  int y1 = y + step - 1;
  int y2 = y + 3*step;
  int y3 = y + 5*step + 1;

  switch (numDots)
  {
    case 1:
      g.fillOval(x2, y2, dotSize, dotSize);
      break;

    < missing code >

  }
}
```

Figure 6-8. A fragment from JM\Ch06\Craps\RollingDie.java

Copy RollingDie.java from JM\Ch06\Craps into your work folder and fill in the blanks in its drawDots method. (Figure 6-1 shows the desired configurations of dots on a die.) Collect all the files for the *Craps* program together: craps.jar (from JM\Ch06\Craps); CrapsGame.java and Die.java (your solutions from the lab in Section 6.9); and RollingDie.java. Compile them, and run the program.

6.13 Summary

The general form of a *conditional statement* in Java is:

```
if (condition)
{
  statementA1;
  statementA2;
  ...
}
else
{
  statementB1;
  statementB2;
  ...
}
```

`condition` may be any Boolean expression.

Conditions are often written with the *relational operators*

`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to
`==`	equal to
`!=`	not equal to

and the *logical operators*

`&&`	and
`||`	or
`!`	not

It is useful for programmers to know two properties from formal logic called *De Morgan's Laws:*

`!(p && q)` is the same as `!p || !q`
`!(p || q)` is the same as `!p && !q`

Use the

```
if...
else if...
else if...
...
else ...
```

structure for branching in multiple ways, and use nested `if-else` for hierarchical branching.

The general form of a `switch` statement is

```
switch (expression)
{
  case valueA:         // Take action A
    statementA1;
    statementA2;
    ...
    break;

  case valueB:         // Take action B
    statementB1;
    ...
    break;

  ...
  ...

  default:             // Take the default action
    ...
    break;
}
```

where *valueA*, *valueB*, etc., are integer or character literal or symbolic constants or literal strings. The switch evaluates *expression* and jumps to the case labeled by the corresponding constant value, or to the default case if no match has been found. A switch can be used to replace a long `if-else-if` sequence.

Exercises

Sections 6.1-6.7

1. Find all relational and all logical operators and `if` statements in the `Balloon` class in `JM\Ch04\BalloonDraw\Balloon.java`.

2. Write and test a method that returns the value of the larger of the integers `x` and `y` (or either one, if they are equal), but do not use any `Math` methods. ✓

```
public static int max(int x, int y)
{
  ...
}
```

3. Write and test a method that returns the value of the largest of three integers `x`, y and `z`.

```
public static int max(int x, int y, int z)
{
  ...
}
```

Implement it in two ways: in the first approach write a one-liner using the `Math.max` method; in the second approach do not use any `Math` methods.

4. Write a boolean method that checks whether a given positive integer `n` is a perfect square. Use `Math`'s `sqrt` and `round` methods to find the square root of `n`, round it, then square the result and compare with `n`. Do not use any iterations or recursion.

5.■ Fill in the missing code in the `totalWages` method, which calculates the total earnings for a week based on the number of hours worked and the hourly rate. The pay for overtime (hours worked over 40) is 1.5 times the regular rate. For example, `totalWages(45, 12.50)` should return 593.75.

```
public double totalWages(double hours, double rate)
{
  double wages;

  < ... missing code >

  return wages;
}
```

Add your code to `Wages.java`, available in `JM\Ch06\Exercises`, and test it.

6.■ Invent three ways to express the XOR ("exclusive OR") operation in Java (that is, write a Boolean expression that involves two `boolean` variables which is true if and only if exactly one of the two variables has the value `true`). ≤ Hint: one of the possible solutions involves only one (relational) operator. ≥ ✓

7. (MC) Which of the following expressions is equivalent to `!(a || !b)` (that is, has the same value for all possible values of the `boolean` variables `a` and `b`)?

A. `a || !b` B. `!a || b` C. `!a && b`
D. `!a && !b` E. `a && !b`

8. Simplify the following expressions (remove as many parentheses as possible) using De Morgan's Laws:

(a) `!((!x || !y) && (a || b))` ✓
(b) `if (!(x == 7) && !(x > 7)) ...`

9. Remove as many parentheses as possible without changing the meaning of the condition:

(a) `if ((((x + 2) > a) || ((x - 2) < b)) && (y >= 0))` ✓
(b) `if (((a >= b) && (a >= c)) && ((a % 2) == 0))`

10. Rewrite the following condition to avoid a possible arithmetic exception:

```
if (Math.sqrt(x) < 3 && x > 7) ...
```

11. Write a Boolean expression that evaluates to `true` if and only if the values of three integer variables *a*, *b*, and *c* form a geometric sequence (that is, $a, b, c \neq 0$ and $a/b = b/c$). ⦇ Hint: recall that comparing `double` values for exact match may not work — use integer cross products instead. ⦈

12. Simplify the following statements:

(a)
```
boolean inside = !((x < left) || (x > right) ||
   (y < top) || (y > bottom));  ✓
```
(b)
```
boolean no = (ch[0] == 'N' && ch[1] == 'O') ||
             (ch[0] == 'n' && ch[1] == 'o') ||
             (ch[0] == 'N' && ch[1] == 'o') ||
             (ch[0] == 'n' && ch[1] == 'O');
```

13.■ Write a `boolean` method `isLeapYear(int year)` that returns `true` if `year` is a leap year and `false` otherwise. A leap year is a year that is evenly divisible by 4 and either is not divisible by 100 or is divisible by 400. For example, 2000 and 2004 are leap years, but 2003 and 2100 are not.

14.■ Write a method

```
public static boolean isLater(int month1, int day1, int year1,
                              int month2, int day2, int year2)
```

that returns `true` if the first date is later than the second and `false` otherwise. Test your method using the provided console application `Dates` (`JM\Ch06\Exercises\Dates.java`), which prompts the user to enter two dates, reads the input, and displays the result of the comparison. ✓

Section 6.8

15. Rewrite the following code using < and no other relational operators.

```
if (avg >= 90)
  grade = 'A';
else if (avg >= 80)
  grade = 'B';
else if (avg >= 70)
  grade = 'C';
else if (avg >= 60)
  grade = 'D';
else
  grade = 'F';
```

16.■ (a) Restore appropriate indentation and optional braces in the following code fragment using `if-else-if` sequences and nested `if-else` statements as appropriate (this code is available in `JM\Ch06\Exercises\WarmWeather.java`):

```
boolean warm;
if(location.isNorthPole() || location.isSouthPole())
{warm = false;} else if(location.isTropics()) {warm
= true;} else if (time.getMonth()==4 || time.getMonth()==10)
{if (weather.isSunny()) {warm = true;}else{warm = false;}}
else if (location.isNorthernHemisphere()) {if(time.getMonth()
>=5 && time.getMonth() <= 9) {warm=true;} else{warm=false;}}
else if(location.isSouthernHemisphere()){if(time.getMonth()
>= 11 ||time.getMonth()<= 3) {warm = true;} else{warm=
false;}} else{warm = false;}
```

(b) Simplify the statement from Part (a) by starting with `warm = false;` then setting `warm` to `true` under the appropriate conditions.

(c) Rewrite the statement from Part (b) in the form

```
warm = < logical expression >;
```

17.■ A "Be Prepared" test prep book costs $20.95; "Next Best" costs $21.95. A site called apzone.com offers a special deal: both for $39.95. If you buy three or more copies (in any mix of these two titles), they are $16.95 each. If you buy 12 or more copies, you pay only $16.00 for each.

(a) Write a method

```
public static double getOrderTotal(int bp, int nb)
{
  ...
}
```

that calculates the total for an order of `bp` copies of "Be Prepared" and `nb` copies of "Next Best," taking the above specials into account.

(b) Test your method in a class with a `main` method that prompts the user for two integers representing the quantities of "Be Prepared" and "Next Best" books desired, and displays the total cost.

18.■ Write a method

```
public Color bestMatch(int r, int g, int b)
```

The method's arguments represent the red, green, and blue components of a color. If one of the components is greater than the other two, `bestMatch` returns that component's color (`Color.RED`, `Color.GREEN`, or `Color.BLUE`). If two components are equal and greater than the third, then `bestMatch` returns their "composite" color, `Color.YELLOW`, `Color.MAGENTA`, or `Color.CYAN` (for red-green, red-blue, and green-blue, respectively). If all three components are equal, `bestMatch` returns `Color.GRAY`.

19.■ `size1` and `size2` are the sizes of two files, and `space` is the amount of available space on a memory card. Write a method that takes these integer numbers as parameters and figures out the largest combination of files that fits on the card. The method should return 3 if both files fit together, the file number (1 or 2) corresponding to the largest file that fits by itself (`1` if the files are the same size), or 0 if neither file fits on the card. Your method must have only one `return` statement.

```
public int findBestFit(int size1, int size2, int space)
{
  < ... missing code >
}
```

Sections 6.9-6.13

20. Generalize the `Die` class from Section 6.9 so that a die may have *n* faces with numbers from 1 to *n* on them. Provide a no-args constructor that initializes the number of faces to 6 and another constructor with one parameter, *n*.

21.■ In the Take-1-3 game two players take turns taking stones from a pile. On each move a player must take one, two, or three stones. The player who takes the last stone wins. A program for this game is available in `JM\Ch06\Exercises\Take1_3.java`.

(a) Determine the winning strategy in this game.

(b) Fill in the blanks in the `computerMove` method:

```
/**
 * Returns the correct number of stones to take
 * (according to the winning strategy) when nStones
 * remain in the pile; if such a move is not possible,
 * returns a random number of stones to take.
 * Precondition: nStones > 0
 */
public int computerMove(int nStones)
{

  _____________________________
  ...
}
```

Continued

(c) Fill in the blanks in the `humanMove` method:

```
/**
 * Prompts the user to take a number of stones;
 * If the move is valid, returns the number of stones
 * taken; otherwise, displays an error message --
 * "You are allowed to only take 1, 2, or 3 stones" or
 * "Can't take that many: only <nStones> left in the pile"
 * -- and returns -1:
 * Precondition: nStones > 0
 */
public int humanMove(int nStones)
{
  System.out.print("How many stones do you take? ");
  int n = kboard.nextInt();

  _____________________________________
  ...
}
```

(d) Test the game thoroughly.

22. Finish the five-line poem:

One, two, buckle your shoe;
Three, four, shut the door;

and write a console Java application that displays the appropriate line of your poem:

```
Enter a number 1-10 (or 0 to quit): 1
Buckle your shoe

Enter a number 1-10 (or 0 to quit): 2
Buckle your shoe

Enter a number 1-10 (or 0 to quit): 6
Pick up sticks

Enter a number 1-10 (or 0 to quit): 0
Bye
```

Use a `switch` statement.

23.■ Finish the program in `JM\Ch06\Exercises\Rps.java` that plays the "Rock, Paper, Scissors" game. You need to supply code for the `nextPlay` method. Use nested `switch` statements.

24.◆ In your `CrapsGame` class, you probably have an `if-else` statement to process the roll correctly depending on the current state of the game:

```
if (point == 0)  // first roll
  ...
else   // subsequent rolls
  ...
```

Define instead a private enum type that would describe the two possible states of the game, `NEW_ROLL` and `KEEP_ROLLING`. Declare a field of that enum type that would represent the current state of the game and replace the `if-else` statement with a `switch` on the value of that field.

```
while (chapter < 7)
  chapter++;
```

Algorithms and Iterations

7.1 Prologue

Historically, software development was largely viewed as an activity focused on designing and implementing algorithms. A formal definition of an algorithm is elusive, which is a sure sign that the concept is fundamentally important.

Informally, an *algorithm* is a more or less compact, general, and abstract step-by-step recipe that describes how to perform a certain task or solve a certain problem.

Algorithms are often associated with computer programs, but algorithms existed long before computers. One of the most famous, Euclid's Algorithm for finding the greatest common factor of two integers is over 2300 years old. You may also recall the algorithm for long division of integers. The question of whether computers have evolved the way they are to support the implementation of algorithms, or whether algorithms (as they are understood now) gained prominence due to the advent of computers, is of the chicken-and-egg variety.

A method of performing a task or solving a problem can be described at different levels of abstraction. Algorithms represent a rather abstract level of solving problems. A computer programmer can learn an algorithm from a book or from an expert. The "algorithmist" knows the capabilities of computers and the general principles of computer programming, but he or she does not have to know any specific programming language. In fact, an algorithm can be used without any computer by a person equipped with a pencil and paper. The programmer can then implement the algorithm in Python, Java, C++, or any other programming language of choice.

The purpose of this chapter is to give you some feel for what algorithms are all about, to consider a few examples and to review one of the main algorithmic devices: iterations. At this stage, we are not going to deal with any complicated algorithms. We will discuss searching and sorting algorithms in Chapter 14.

Loops or *iterative statements* tell the program to repeat a fragment of code several times for as long as a certain condition holds. Java provides three convenient iterative statements: `while`, `for`, and `do-while`. Strictly speaking, any iterative code can be implemented using only the `while` loop. But the other two add flexibility and make the code more concise and idiomatic.

Iterations are often used in conjunction with lists or files. We can use iterations to process all the elements of a list (useful for finding a particular value in a list or calculating the sum of all the elements) or read and process lines of text from a file. We will discuss how loops are used with lists later, in Chapters 9 and 11. Java also has a convenient "for each" loop for traversing a collection of values.

In this chapter you will learn the Java syntax for `while`, `for`, and `do-while` loops and how to use `break` and `return` in loops.

7.2 Properties of Algorithms

Suppose we want to use a computer to calculate $1^2 + 2^2 + ... + 100^2$. Potentially, we could add up the squares of numbers from 1 to 100 by brute force, making our "algorithm" quite long (Figure 7-1).

```
sum ← 0
sq = 1 * 1; sum ← sum + sq
sq = 2 * 2; sum ← sum + sq
sq = 3 * 3; sum ← sum + sq
sq = 4 * 4; sum ← sum + sq
...
 (and so on, spelling out every
single line)
...
sq = 99 * 99; sum ← sum + sq
sq = 100 * 100; sum ← sum + sq
```

Figure 7-1. Brute-force "non-algorithm" for calculating $1^2 + 2^2 + ... + 100^2$

But what good would it do to have a computer that could execute billions of instructions per second if we had to write out every single one of these instructions separately? You probably feel intuitively that there is a more concise way to describe this task: start with the sum 0 and the number 1; add the square of that number to the sum; take the next number, and so on: repeat for all the numbers from 1 to 100. It turns out your intuition is not far from a formal algorithm. Such an algorithm can be built around two variables: one that holds the accumulating sum of squares and another that holds the current number.

In descriptions of algorithms, as in computer programs, we give names to variables for convenience. In this algorithm, let us call our two variables *sum* and *k*. At the beginning, we want to set *sum* to 0 and *k* to 1. Let's record these steps using a more formal notation, called *pseudocode*:

```
sum ← 0
k ← 1
```

Now we want to compute k^2 and add it to *sum*. For extra clarity, let's introduce another variable that will hold k^2. Let's call it *sq*. So our next steps are

```
sq ← k * k
sum ← sum + sq
```

The last statement indicates that we are updating the value of *sum*. That values of variables can be updated is the key for creating algorithms.

Now we want to "take the next number." How can we formalize that? Easily: to take the next number, we simply increment the value of the variable *k* by 1:

```
k ← k + 1
```

The above three steps constitute the core of our algorithm. We need to repeat these steps as long as *k* remains less than or equal to 100. Once *k* becomes greater than 100, the process stops and *sum* contains the result. Thus, for each value of *k*, we need to make a decision whether to continue or quit. The ability to make such decisions is one of the devices available for implementing algorithms.

OK, suppose we have an algorithm for computing $1^2 + 2^2 + ... + 100^2$. But what if we need to compute $1^2 + 2^2 + ... + 500^2$? Do we need another algorithm? Of course not: it turns out that our algorithm, with a small change, is general enough to compute $1^2 + 2^2 + ... + n^2$ for any positive integer *n*. We only need to introduce a new input variable *n* and replace $k \le 100$ with $k \le n$ in the decision test.

Figure 7-2-a shows pseudocode for the final algorithm. Pseudocode can be somewhat informal as long as people who need to implement this algorithm in a particular programming language understand what it means.

Figure 7-2-b represents the same algorithm graphically. This kind of representation is called a *flowchart*. Parallelograms represent input and output; rectangles represent processing steps; diamonds — conditions checked.

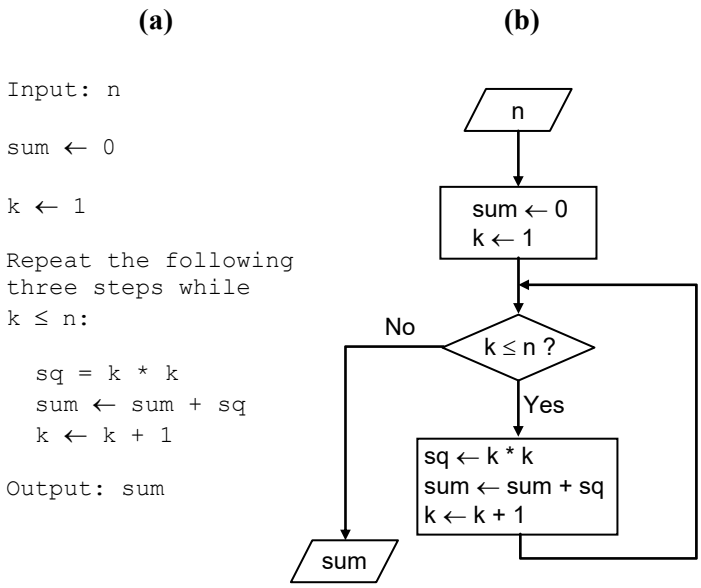


Figure 7-2. Pseudocode and flowchart for an algorithm that calculates $1^2 + 2^2 + ... + n^2$

We can deduce several general properties from this simple example. First, an algorithm is rather compact. A reasonable algorithm folds the computation into one or several fragments that can be repeated multiple times. These repetitions, called *iterations*, execute exactly the same instructions but work with different values of variables. In the example in Figure 7-2 the algorithm iterates n times through three steps, updating *sum* and incrementing k by 1 in each iteration.

The second property of a good algorithm is that it is rather general. The brute-force "algorithm" in Figure 7-1 works for $n = 100$, but you have to change your program if you want it to work for, say, $n = 500$. The algorithm in Figure 7-2, a truly usable algorithm, works for any n without changing anything. The value of n is a parameter, an input value for the procedure. The running time of a program based on this algorithm will be different for different n, but the length of the program text itself remains the same regardless of the value of n.

The third property of an algorithm, as we have mentioned above, is that an algorithm is abstract: it does not depend on a particular programming language or computer system. Figure 7-3 shows how the same algorithm might be coded in Python, C or C++, and Java.

In Python

```
def addSquares(n):
    s = 0
    k = 1
    while k <= n:
      s += k*k
      k += 1
    return s
```

In Java

```
public class MyMath
{
  public static int
            addSquares(int n)
  {
    int sum = 0;

    for (int k = 1; k <= n; k++)
      sum += k*k;
    return sum;
  }
}
```

In C or C++

```
int addSquares(int n)
{
    int k, sum = 0;

    for (k = 1; k <= n; k++)
        sum += k*k;
    return sum;
}
```

Figure 7-3. Python, Java, and C/C++ implementations of the sum-of-squares algorithm

At the same time, an algorithm does depend on the general computing model. For example, an algorithm for solving a problem with an abacus will be different from solving the same problem on a conventional computer, which in turn may be different from solving the same problem on a supercomputer with parallel processing capabilities.

❖ ❖ ❖

Different algorithms can solve the same problem. Because algorithms are abstract, we can study their properties and compare them for efficiency without ever implementing them on a computer. For example, we can see that in the sum-of-squares algorithm in Figure 7-2, a multiplication operation will be executed *n* times.

Multiplication is in general a more time-consuming operation than addition. A slightly more efficient version of this algorithm can calculate the same result without any multiplications. This version is based on a simple observation that $(k+1)^2 = k^2 + m$, where $m = 2k+1$. To go from one value of *sq* to the next we need to add *m* to it. To go from one value of *m* to the next we need to increment *m* by 2. This way we can update the value of *sq* without multiplications (Figure 7-4).

```
Input: n

sum ← 0
sq ← 1
k ← 1
m ← 3

Repeat the following four
steps while k ≤ n:
  sum ← sum + sq
  sq ← sq + m
  m ← m + 2
  k ← k + 1

Output: sum
```

Figure 7-4. A version of the sum-of-squares algorithm without multiplications

We can verify that this algorithm indeed works by tracing its steps while keeping track of the values of the variables at the end of each iteration (Figure 7-5).

Iteration #	*k*	*m*	*sq*	*sum*
0	1	3	1	0
1	2	5	4	1
2	3	7	9	5
3	4	9	16	14
...	...	...	...	...

Figure 7-5. Tracing the values of variables in the sum-of-squares algorithm

The gain in efficiency from this no-multiplications algorithm is insignificant, of course, when this task is performed on a modern computer, but it would save the day if we had to use an adding machine or the first ENIAC computer. Even now a slight improvement like this can make a difference in a computationally-intensive problem.

7.3 The `while` and `for` Loops

The general form of the `while` statement in Java is:

```
while (condition)
{
  statement1;
  statement2;
  ...
  statementN;
}
```

condition can be any logical expression; it is evaluated exactly as in an `if` statement.

Informally the `while` statement is often called a *while loop*. The statements within braces are called the *body* of the loop. If the body consists of only one statement, the braces surrounding the body can be dropped:

```
while (condition)
  statement1;
```

It is important not to put a semicolon after `while(condition)`. With a semicolon, the loop would have no body, only an empty statement; *statement1* would be left completely out of the loop.

❖ ❖ ❖

The following method returns the sum of all integers from 1 to *n*:

```
/**
 * Returns the sum of all integers from 1 to n, if n >= 1,
 * and 0 otherwise.
 */
public static int sumUpTo(int n)
{
  int sum = 0;
  int k = 1;

  while (k <= n)
  {
    sum += k;
    k++;
  }

  return sum;
}
```

Three elements must be present with any `while` loop: an initialization, a test of the condition, and a change.

1. Initialization. The variables tested in *condition* must be initialized to some values before the loop. In the above example, `k` is initially set to `1` in the declaration `int k = 1`.

2. Testing. The condition is tested before each pass through the loop. If it is false, the body is not executed, the iterations end, and the program continues with the first statement after the loop. If the condition is false at the very beginning, the body of the `while` loop is not executed at all. In the `sumUpTo` example, the condition is `k <= n`. If `n` is zero or negative, the condition will be false on the very first test (since `k` is initially set to `1`). Then the body of the loop will be skipped and the method will return `0`.

3. Change. At least one of the variables tested in the condition must change within the body of the loop. Otherwise, the loop will be repeated over and over and never stop, and your program will "hang." The change of a variable is often implemented with increment or decrement operators, but it can come from any assignment or input statement. In any case, the tested variables must at some point get values that will make the condition false. Then the program jumps to the first statement after the body of the loop.

In the `sumUpTo` method, the change is achieved by incrementing `k`:

```
...
k++;         // add 1 to k (increment k)
```

These three elements — initialization, testing, and change — must be present, explicitly or implicitly, in every `while` loop.

The `for` loop is a shorthand for the `while` loop that combines the initialization, condition, and change in one statement. Its general form is:

```
for (initialization; condition; change)
{
  statement1;
  statement2;
  ...
}
```

where *initialization* is a statement that is always executed once before the first pass through the loop, *condition* is tested before each pass through the loop, and *change* is a statement executed at the end of each pass through the loop.

A typical `for` loop for repeating the same block of statements `n` times is:

```
for (int count = 1; count <= n; count++)
{
  < ... statements >
}
```

For instance, the following `for` loop prints *n* spaces:

```
for (int count = 1; count <= n; count++)
{
  System.out.print(" ");
}
```

The braces can be dropped if the body of the loop has only one statement, but many people like to have braces even around one statement because that makes it easier to add statements to the body of the loop later. We don't feel strongly about either style, so we will use either, depending on the situation or our mood.

Note that a variable that controls the loop can be declared inside the `for` statement. For example:

```
for (int k = 1; k <= 100; k++)
{
  sum += k * k;
}
```

This is common style, but you have to be aware that the scope of a variable declared this way does not extend beyond the body of the loop. In the above example, if you add

```
System.out.println(k);
```

after the closing brace, you will get a syntax error: "cannot find symbol: variable k."

The `sumUpTo` method can be rewritten with a `for` loop as follows (see `JM\Ch07\MyMath\MyMath.java`):

```
public static int sumUpTo(int n)
{
  int sum = 0;

  for (int k = 1; k <= n; k++)
    sum += k;

  return sum;
}
```

The following method (also in the `MyMath` class) calculates *n*! (*n factorial*), which is defined as the product of all integers from 1 to *n*:

```
/**
 * Returns 1 * 2 * ... * n, if n >= 1; otherwise returns 1
 */
public static long factorial(int n)
{
  long f = 1;

  for (int k = 2; k <= n; k++)  // if n < 2, this loop is skipped
    f *= k;

  return f;
}
```

7.4 The `do-while` Loop

The `do-while` loop differs from the `while` loop in that the condition is tested <u>after</u> the body of the loop. This ensures that the program goes through the loop at least once.

The `do-while` statement's general form is:

```
do
{
  ...
} while (condition);
```

The program repeats the body of the loop as long as *condition* remains true. It is better always to keep the braces, even if the body of the loop is just one statement, because the code is hard to read without them.

`do-while` loops are used less frequently than `while` and `for` loops. They are convenient when the variables tested in the condition are calculated or entered within the body of the loop. The following example comes from `main` in the `MyMath` class (`JM\Ch07\MyMath\MyMath.java`):

```
public static void main(String[] args)
{
  Scanner kb = new Scanner(System.in);
  int n;

  do
  {
    System.out.print("Enter an integer from 4 to 20: ");
    n = kb.nextInt();
  } while (n < 4 || n > 20);

  kb.close();

  System.out.println();
  System.out.println("1 + ... + " + n + " = " + sumUpTo(n));
  System.out.println(n + "! = " + factorial(n))
}
```

In this code, the `do-while` loop calls `Scanner`'s `nextInt` method to get the value of `n` from the user's input. The iterations continue until the user enters a number within the requested range.

If for some reason you do not like do-while loops, you can easily avoid them by using a while loop and initializing the variables in a way that makes the condition true before the first pass through the loop. The do-while loop in the above code, for example, can be rewritten as follows:

```
int n = -1;

while (n < 4 || n > 20)
{
  System.out.print("Enter an integer from 4 to 20: ");
  n = kb.nextInt();
}
```

7.5 return and break in Loops

We saw in Section 6.10 that break is used inside a switch statement to end a case and break out of the switch. break can be also used in the body of a loop. It instructs the program to break out of the loop immediately and go to the first statement after the body of the loop. break must always appear inside a conditional (if or else) statement — otherwise you will just break out of the loop on the very first iteration.

The following method checks whether a positive integer *n* is a prime. (A prime is an integer that is greater than 1 and has no factors besides 1 and itself.)

```
/**
 * Returns true if n is a prime, false otherwise.
 */
public static boolean isPrime(int n)
{
  boolean noFactors = true;

  if (n <= 1)
    return false;

  for (int m = 2; noFactors; m++)
  {
    if (m * m > n)
      break;

    if (n % m == 0)
      noFactors = false;
  }
  return noFactors;
}
```

Our algorithm has to check all potential factors *m*, but only as long as $m^2 \le n$ (because if *m* is a factor, then so is *n*/*m*, and one of the two must be less than or equal to the square root of *n*). The above `isPrime` method employs `break` to reduce the number of iterations.

Another way to break out of the loop (and out of the method) is to put a `return` statement inside the loop. A shorter version of `isPrime` (which can be found in (JM\Ch07\MyMath\MyMath.java) uses `return`:

```
public static boolean isPrime(int n)
{
  if (n <= 1)
    return false;

  int m = 2;

  while (m * m <= n)
  {
    if (n % m == 0)
      return false;
    m++;
  }

  return true;
}
```

Either version is acceptable, but the latter may be clearer. You will find some programmers, though, who like to have only one `return` in each method and who find a `break` or `return` inside a loop objectionable.

`break` (or `return`) can be used to break out of an "infinite" loop. For example:

```
// Find the first prime over 10000:

int n = 10001;

while (true)
{
  if (isPrime(n))
    break;
  n++;
}

System.out.println(n);
```

7.6 Nested Loops

A nested loop is a loop within a loop. For example,

```
for (int row = 1; row <= n; row++)
{
  for (int count = 1; count <= row; count++)
    System.out.print("*");
  System.out.println();
}
```

prints a "triangle" made up of `n` rows of stars:

```
*
**
***
****
...
*******
```

Nested loops are convenient for traversing two-dimensional grids and arrays. For example:

```
for (int r = 0; r < grid.getNumRows(); r++)
{
  for (int c = 0; c < grid.getNumCols(); c++)
  {
    System.out.print(grid.get(r, c));
  }
  System.out.println();
}
```

Nested loops are also used for finding duplicates in a list. For example:

```
for (int i = 0; i < list.size(); i++)
{
  for (int j = i + 1; j < list.size(); j++)
  {
    if (list.get(i).equals(list.get(j)))
      System.out.println("Duplicates at " + i + ", " + j);
  }
}
```

We will talk about lists and arrays in Chapters 9 and 11.

The following method (JM\Ch07\MyMath\MyMath.java) tests the so-called Goldbach conjecture that any even integer greater than or equal to 4 can be represented as a sum of two primes:*

```
/**
 * Tests Goldbach conjecture for even numbers
 * up to bigNum
 */
public static boolean testGoldbach(int bigNum)
{
  for (int n = 6; n <= bigNum; n += 2)  // obviously true for n = 4
  {
    boolean found2primes = false;

    for (int p = 3; p <= n/2; p += 2)
    {
      if (isPrime(p) && isPrime(n - p))
        found2primes = true;
    }

    if (!found2primes)
    {
      System.out.println(n + " is not a sum of two primes!");
      return false;
    }
  }

  return true;
}
```

This method can be made a little more efficient if we break out of the inner loop once a pair of primes is found:

```
    for (int p = 3; p <= n/2; p += 2)
    {
      if (isPrime(p) && isPrime(n - p))
      {
        found2primes = true;
        break;
      }
    }
```

* In 1742, Christian Goldbach, an amateur mathematician, in a letter to Euler stated a hypothesis that any even number greater than or equal to 4 can be represented as a sum of two primes. For example, 18 = 5 + 13; 20 = 7 + 13; 22 = 11 + 11. The Goldbach conjecture remains to this day neither proved nor disproved.

Note that `break` takes you out of the inner loop, but not the outer loop. But

```
if (isPrime(p) && isPrime(n - p))
{
  return true;
}
```

would be a mistake, because it would quit the method right away, before we had a chance to test the conjecture for all `n <= bigNum`.

You can have a loop within a loop within a loop — loops can be nested to any level. But once you go to more than two or three levels, your code may become intractable. Then you might consider moving the inner loop or two into a separate method. To be honest, our `testGoldbach` method would be simpler if we moved the inner loop into a separate method:

```
private static boolean found2Primes(int n)
{
  for (int p = 3; p <= n/2; p += 2)
  {
    if (isPrime(p) && isPrime(n - p))
      return true;
  }
  return false;
}
```

Then `testGoldbach` would become simply

```
public static boolean testGoldbach(int bigNum)
{
  for (int n = 6; n <= bigNum; n += 2)
  {
    if (!found2Primes(n))
    {
      System.out.println(n + " is not a sum of two primes!");
      return false;
    }
  }
  return true;
}
```

7.7 *Case Study:* Euclid's GCF Algorithm

Figure 7-6 shows a flowchart for Euclid's Algorithm, mentioned earlier, for finding the greatest common factor of two positive integers. This algorithm is based on the observation that if we subtract the smaller number from the larger number, and replace the larger number with the result, the GCF of the two numbers remains the same. In other words, GCF(a, b) = GCF($b - a$, a), assuming $b > a$.

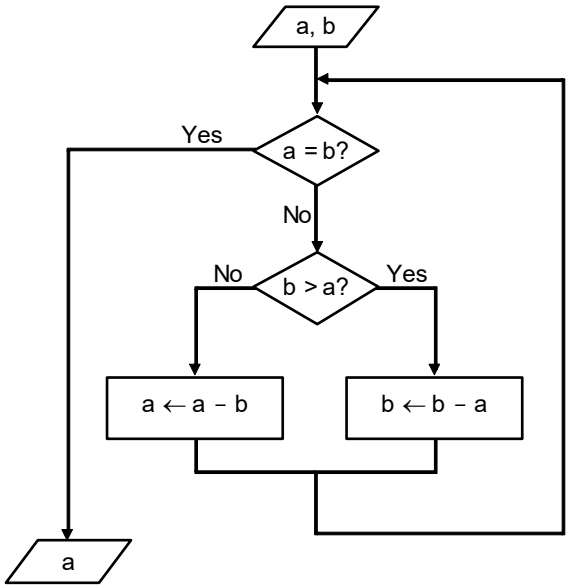


Figure 7-6. Euclid's Algorithm for finding the greatest common factor of two positive integers

Let us take an example, $a = 30$, $b = 42$, and see how this algorithm works:

Iteration #	**0**	**1**	**2**	**3**	**4**
a	30	30	18	6	6
b	42	12	12	12	6

An iterative implementation of this algorithm is shown in Figure 7-7. After each iteration through the `while` loop, both *a* and *b* remain positive but the larger of them gets reduced, so eventually they must become equal and the iterations stop.

```
// Returns GCF of a and b
// Precondition: a > 0, b > 0
public static int gcf(int a, int b)
{
  while (a != b)
  {
    if (a > b)
      a -= b;
    else
      b -= a;
  }
  return a;
}
```

Figure 7-7. An iterative implementation of Euclid's Algorithm in Java

As you know, the modulo division operator `%` computes the remainder when `a` is divided by `b`. The statement

```
a %= b;   // same as a = a % b;
```

replaces `a` with the result of `a % b`, which is equivalent to subtracting `b` from `a` as many times as possible while the result remains greater than or equal to zero. We could use the modulo division operator to make the code for the iterative `gcf` method more efficient. This is left to you as an exercise (Question 26).

7.8 *Lab:* Perfect Numbers

A whole number is called *perfect* if it is equal to the sum of all of its divisors, including 1 (but excluding the number itself). For example, 28 = 1 + 2 + 4 + 7 + 14. Perfect numbers were known in ancient Greece. In Book VII of *Elements*, Euclid (300 BC) defined a perfect number as one "which is equal to its own parts."

Nicomachus, a Greek mathematician of the first century, wrote in his *Introduction to Arithmetic* (around A.D. 100):

> *In the case of the too much, is produced excess, superfluity, exaggerations and abuse; in the case of too little, is produced wanting, defaults, privations and insufficiencies. And in the case of those that are found between the too much and the too little, that is, in equality, is produced virtue, just measure, propriety, beauty and things of that sort — of which the most exemplary form is that type of number which is called perfect.*perfectnumbers

Unfortunately, Nicomachus had many mistakes in his book. For example, he stated erroneously that the *n*-th perfect number has *n* digits and that perfect numbers end alternately in 6 and 8. He knew of only four perfect numbers and jumped to conclusions.

Add code to `MyMath.java` (`JM\Ch07\MyMath\MyMath.java`) to find the first four perfect numbers.

❖ ❖ ❖

You might be tempted to use your program to find the fifth perfect number. Then you'd better be patient: on a relatively fast computer, it could take almost an hour. There is a better strategy. Euclid proved that if you find a number of the form $2^n - 1$ that is a prime, then $2^{n-1}(2^n - 1)$ is a perfect number! For example, $(2^3 - 1) = 7$ is a prime, so $28 = 2^2(2^3 - 1)$ is a perfect number. Many centuries later, Euler proved that any even perfect number must have this form. Therefore the search for even perfect numbers can be reduced to the search for primes that have the form $2^n - 1$. Such primes are called *Mersenne primes*, after the French math enthusiast Marin Mersenne (1588-1648) who made them popular.

In 1996, George Woltman, a software engineer, started The Great Internet Mersenne Prime Search project (GIMPS).[mersenne] In this project, volunteers contribute idle CPU time on their personal computers for the search. At the time of this writing, 51 Mersenne primes have been found; the largest, $2^{82,589,933}-1$, was found on December 7, 2018. (It has 24,862,048 digits.)

Add code to `MyMath.java` (`JM\Ch07\MyMath\MyMath.java`) to find and print out the first six Mersenne primes, and the corresponding first six perfect numbers. Note that while the sixth Mersenne is still well within the Java `int` range, the sixth perfect number, 8,589,869,056, is not. Use a `long` variable to hold it.

It is unknown to this day whether any odd perfect numbers exist. It has been shown that such a number must have at least 300 digits!

7.9 Summary

An *algorithm* is an abstract and formal step-by-step recipe that tells how to perform a certain task or solve a certain problem on a computer. The main properties of algorithms are compactness, generality, and abstraction. Compactness means that a fairly large computational task is folded into a fairly short sequence of instructions that are repeated multiple times through iterations or recursion. Generality means that the same algorithm can work with different sets of input data values. Abstraction means that the algorithm does not depend on a particular programming language.

Flowcharts and *pseudocode* are two popular ways to describe algorithms.

Decisions and iterations and are some of the major devices used in algorithms. A decision allows the algorithm to take different paths depending on some condition. Iterations help fold the task into one or several mini-tasks that are repeated multiple times.

The computer may execute the same instructions on each iteration, but the values of at least some of the variables must change. The iterations continue while the values of the variables meet a specified condition.

Java offers three iterative statements:

```
while (condition)
{
  ...
}

for (initialization; condition; change)
{
  ...
}

do
{
  ...
} while (condition);
```

In a `while` loop, the variables tested in *condition* must be initialized before the loop, and at least one of them has to change inside the body of the loop. The program tests *condition* before each pass through the loop. If *condition* is false on the very first test, the `while` loop is skipped, and the program jumps to the first statement after the body of the loop. Otherwise the program keeps iterating for as long as *condition* holds true.

The `for` loop combines *initialization*, *condition*, and *change* in one statement. The *initialization* statement is executed once, before the loop. *condition* is tested before each pass through the loop, and if it is false, the loop is skipped and the program jumps to the next statement after the body of the loop. The *change* statement is executed at the end of each pass through the loop.

The `do-while` loop is different from the `while` loop in that *condition* is tested after the body of the loop. Thus the body of a `do-while` loop is always executed at least once.

A `break` statement inside the body of a loop tells the program to jump immediately out of the loop to the first statement after the body of the loop. `break` should appear only inside an `if` or `else` statement; otherwise it will interrupt the loop on the very first iteration. A `break` statement inside a nested loop will only break out of the inner loop. However, a `return` statement inside a loop immediately quits the loop and the whole method, too.

Exercises

Sections 7.1-7.5

1. Draw a flowchart and write pseudocode for an iterative algorithm that calculates $1+\frac{1}{2^2}+\frac{1}{3^2}+...+\frac{1}{n^2}$ for any given *n*. An interesting mathematical fact: these sums converge to $\frac{\pi^2}{6}$ as *n* increases. Write a program to see how close is the sum for *n* = 10000 to $\frac{\pi^2}{6}$. ✓

2. Draw a flowchart and write pseudocode for an iterative algorithm that calculates $1-\frac{1}{2}+\frac{1}{3}-...+(or-)\frac{1}{n}$ for any given *n*. $\frac{1}{k}$ is added to the sum if *k* is odd and subtracted from the sum if *k* is even. ≤ Hint: multiply $\frac{1}{k}$ by a factor that is equal to 1 or −1 before adding it to the sum. Flip the sign of the factor on each iteration. ≥ An interesting mathematical fact: these sums converge to $\ln 2$, the natural logarithm of 2. Write a program that compares the sum for *n* = 10000 with the value returned by `Math.log(2)`.

3. Let's pretend for a moment that Java does not support multiplication. Write and test an iterative version of the following method:

```
// Returns the product of a and b
// Precondition: a >= 0, b >= 0
public int product(int a, int b)
{
  ...
}
```

4. Design an iterative algorithm that, given two positive integers *m* and *n*, calculates the integer quotient and the remainder when *m* is divided by *n*. Your algorithm can use only +, −, and comparison operations for integers. Implement your algorithm in a Java program. ✓

5. Add missing code to the following program:

```
/**
 * This program prompts the user to enter
 * a positive integer n and a line of text
 * and displays that line n times
 */

import java.util.Scanner;

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

    System.out.print("Enter a positive integer: ");
    int n = kb.nextInt();
    kb.nextLine();  // consume the rest of the line

    System.out.print("Enter a line of text: ");
    String text = kb.nextLine();
    kb.close();

    < missing code >
  }
}
```

6. The population of Mexico at the end of 2021 was around 130.3 million. Write and test a program that will prompt the user for a number (a `double`) that represents a target population number (in millions) and print out the year in which the population of Mexico will reach or exceed that number, assuming a constant growth rate of 1.1 percent per year. Declare the starting year (2021), the starting population number (130.3), and the growth rate (1.1 percent) as symbolic constants. ✓

7. Each time Kevin re-reads his Java book (which happens every month), he learns 10% of whatever material he didn't know before. He needs to score at least 95% on the comprehensive exam to become a certified Java developer. When Kevin started, he knew nothing about Java. Write a method that simulates Kevin's learning progress and returns the number of months it will take him to get ready for the exam. Write a `main` method that displays the result (in years and months).

8. Write a method `int addOdds(int n)` that calculates and returns the sum of all odd integers from 1 to *n*. Your method should use exactly one `for` loop and no other iterative or `if-else` statements. (Do not use the formula for the sum of odd numbers.) ✓

9. Write a program that produces the following output (where the user may enter any positive integer under 10): ✓

```
Enter a positive integer under 10: 6
1 + 2 + 3 + 4 + 5 + 6 = 21
```

10.■ (a) Modify one of the versions of the `isPrime` method on pages 189-190 so that if the argument is not 2 it tests only odd numbers as potential factors of *n*. ✓

(b) Make `isPrime` even more efficient by testing only potential factors that are relatively prime with 6 (that is, factors that are not evenly divisible by either 2 or 3). ✓

11. Recall that $1+3+\ldots+(2p-1)=p^2$ for any integer $p \geq 1$. Write a "simple" method

```
public static boolean isPerfectSquare(int n)
```

that tests whether a given number is a perfect square. A "simple" method cannot use arrays, nested loops, `Math` functions, or arithmetic operations except addition. ✓

12.■ Write a method `sumDigits` that calculates and returns the sum of all the digits of a given non-negative integer.

13. Finish the program `AverageScore.java` (in `JM\Ch07\Exercises`), which reads integer scores from a file and prints out their average. ⩽ Hint: if `input` is a `Scanner` object associated with a file, `input.hasNextInt()` returns `true` if there is an integer value left unread in the file; otherwise it returns `false`; `input.nextInt()` returns the next integer read from the file. ⩾

14.■ A pair of prime numbers is called *twin primes* if their difference is equal to 2. For example, 3 and 5 is a pair of twin primes, and so is the pair 11 and 13. Write a program that prints out the first 20 pairs of twin primes. Your program must examine for being a prime only odd numbers and check whether a particular number is a prime only once. Use the `isPrime` method you wrote in Question 10.

15.■ Given a positive number a, the sequence of values

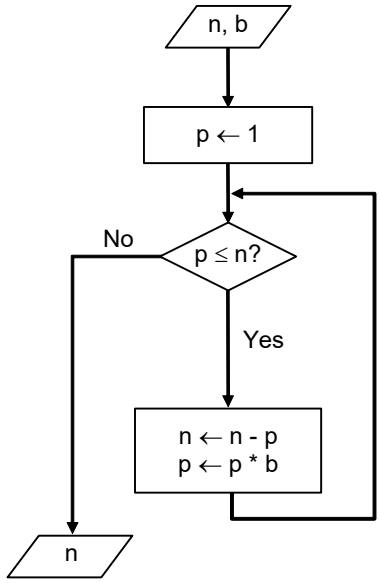
$$x_0 = \frac{a}{2}$$

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n} \right) \quad (n \geq 0)$$

converges to $\sqrt{a}$. Fill in the blanks in the following method, which uses iterations to estimate the square root of a number:

```
// Returns an estimate x of the square root of a,
// such that |x^2 - a| < 0.01
// Precondition: a is greater than 5.
public static double sqrtEst(double a)
{
  double x = a/2;
  double diff;

  do
  {
    ...

  } while ( ... );

  return x;
}
```

16. What is the output from the algorithm shown in the flowchart below when the input is $n = 37$, $b = 2$? Write a program and check your answer.



✓

17. Consider a sequence $x_1 = 1$, $x_2 = 1 + \frac{1}{1}$, $x_3 = 1 + \frac{1}{1 + \frac{1}{1}}$, ... In this sequence $x_{n+1} = 1 + \frac{1}{x_n}$ (for $n \geq 1$). Write a method in Java that computes x_n. As n increases, the terms of this sequence get closer and closer to the golden ratio $\frac{1+\sqrt{5}}{2} \approx 1.61803398875$. How close is x_{10} to the golden ratio?

18.■ In the sequence in Question 17, $x_1 = 1$, $x_2 = \frac{2}{1}$, $x_3 = \frac{3}{2}$, $x_4 = \frac{5}{3}$, $x_5 = \frac{8}{5}$, ... Write a program that computes and prints out x_n as a simple fraction, giving its numerator and denominator. Which fraction with the denominator that does not exceed 50 best approximates the golden ratio?

Sections 7.6-7.9

19. Write and test a program that prints out a nicely formatted multiplication table up to 12.

20. Write a method

```
public static void printStarTriangle(int n)
```

that displays *n* rows of stars, as follows:

```
     *
    ***
   *****
  ...
*************
```

The last row of stars should start at the first position on the line. ✓

21. Write and test a method `printCheckerboard(int n)` that displays a checkerboard pattern with `n` rows and `n` columns. For example, `printCheckerboard(7)` should display

```
#o#o#o#
o#o#o#o
#o#o#o#
o#o#o#o
#o#o#o#
o#o#o#o
#o#o#o#
```

22.■ How many positive integers under 1,000,000 have at least one 7 and at least one 9 among its digits? The mathematical result is $\sum_{n=1}^{5}\left(9\cdot 10^n - 16\cdot 9^n + 7\cdot 8^n\right) = 199{,}262$. Write a program that counts all such numbers using a "brute-force" method (examining the digits in each number from 1 to 999,999) and confirms the mathematical result.

23.◆ Consider all fractions that have positive denominators between 1 and 100. Write a program that finds the two such fractions that are closest to 17 / 76: one from above and one from below.

24.■ Write a program that supports the following dialog with the user:

```
Enter quantity: 75
You have ordered 75 ripples -- $19.50

Next customer (y/n): y

Enter quantity: 97
Ripples can be ordered only in packs of 25.

Next customer (y/n): t
Next customer (y/n): n

Thank you for using Ripple Systems.
```

If, in response to the "Next customer" prompt, the user presses <Enter> or enters anything other than a 'y' or an 'n', the program should repeat the prompt.

Define the unit price of a ripple as a constant equal to 26 cents.

Hints:

Use the following statement to display the quantity ordered and the total dollar amount of the order:

```
System.out.printf("You have ordered %d Ripples -- $%.2f\n\n",
                        quantity, price * quantity);
```

Use the following statements to read the quantity ordered and the user response to the "Next customer?" prompt:

```
Scanner keyboard = new Scanner(System.in);
char answer;
...

  int quantity = keyboard.nextInt();
  keyboard.nextLine();  // skip the rest of the line
  ...

  String str = keyboard.nextLine().trim();
  if (str.length() == 1)
    answer = str.charAt(0);
  else
    answer = ' ';
```

25.■ Write and test a method that takes an amount in cents and prints out all possible representations of that amount as a sum of several quarters, dimes, nickels, and pennies. For example:

```
30 cents = 0 quarters + 2 dimes + 1 nickels + 5 pennies
```

(There are 18 different representations for 30 cents and 242 for $1.00.)

26.■ Fill in the blanks in the `gcf` method (see Section 7.7) using the modulo division operator:

```
// Returns GCF of a and b
// Precondition: a > 0, b > 0
public static int gcf(int a, int b)
{
  while ( _________________ )
  {
    int temp = ;

    a = _____________________________ ;

    b = _____________________________ ;
  }
  return b;
}
```

Test your method.

27.◆ How many <u>different</u> straight lines pass through the origin (0, 0) and a point with coordinates (x, y) such that $1 \le y \le x \le 100$? Write a program to find out.

"Chapter 8"

Strings

8.1 Prologue

In Java, a string of characters is represented by an object of the `String` type. `String` objects are treated pretty much like any other type of objects: they have methods and they can be passed to other methods (always as references) or returned from methods. But, they are different in two respects: the Java compiler knows how to deal with *literal strings* (represented by text in double quotes), and the + and += operators can be used to concatenate a string with another string, a number, or an object.

The `String` class belongs to the `java.lang` package, which is automatically imported into all programs. In this chapter we will cover some properties and methods of the `String` class that help us use strings in our programs. In particular, we will discuss:

- `String` constructors
- The immutability property
- Commonly used `String` methods
- How to format a number into a string and extract a number from a string
- A few methods of the `Character` class that identify digits and letters

8.2 Literal Strings

Literal strings are written as text in double quotes. For example, `"cupcake"` or `"Twitter"` are literal strings. The text may contain escape characters (see Section 5.5). Recall that the backslash character '`\`' is used as the "escape" character: inside a literal string `\n` stands for "newline," `\t` represents a tab character, `\"` represents a double quote, and `\\` represents a backslash. For example:

```
String pathname = "C:\\Ch08\\funny.txt";
                                  // meaning C:\Ch08\funny.txt
```

A literal string can be empty, too, if there is nothing between the quotes.

```
String s = "";  // empty string
```

Literal strings act as `String` objects, but they do not have to be created — they are "just there" when you need them. The compiler basically treats a literal string as a reference to a `String` object with the specified value that is stored somewhere in the computer memory. If you want, you can actually call that object's methods (for example, `"Internet".length()` returns 8). A declaration

```
String city = "Boston";
```

sets the reference `city` to a `String` object `"Boston"`. Note that `Boston` here is not the name of the variable (its name is `city`) but its value.

8.3 `String` Constructors and Immutability

The `String` class has over a dozen constructors, but it is less common to use constructors for strings than for other types of objects. Instead, we can initialize `String` variables either to literal strings or to strings returned from `String`'s methods.

One of the constructors, `String()`, takes no parameters and builds an empty string; rather than invoking this constructor with the `new` operator, we can simply write:

```
String str = "";  // str is initialized to an empty string
```

Another constructor is a copy constructor `String(String s)`, which builds a copy of a string `s`. But in most cases we do not need to make copies of strings because, as we'll explain shortly, strings, once created, never change; so instead of copying a string we can just copy a reference. For example:

```
String str = "Foo Fighters";
```

This is not exactly the same as

```
String str = new String("Foo Fighters");
```

but, as far as your program is concerned, these two declarations of `str` act identically.

Other constructors create strings from character and byte arrays. They are potentially useful, but not before we learn about arrays (Chapter 9).

There is a big difference between an empty string and an uninitialized `String` reference.

Empty strings are initialized to `""` or created with the no-args `String` constructor, as in

```
String s1 = "";           // s1 is set to an empty string
String s2 = new String(); // s2 is set to an empty string
```

A field of the `String` type is set to `null` by default:

```
private String pathname; // instance variable pathname is set to null
```

You can call methods for an empty string, and they will return the appropriate values. For example, `s1.length()` returns 0, and `s2.equals("")` returns `true`. But if a method is called for a reference that is equal to `null`, the program throws a `NullPointerException` and quits.

Once a string object is constructed, it cannot be changed! If you look carefully at `String`'s methods, summarized in Figure 8-3, you will notice that none of these methods changes the content of a string.

A string is an *immutable* object: none of its methods can change the content of a `String` object.

For example, you can get the value of a character at a given position in the string using the `charAt` method. But there is no method to set or replace one character in a string. If you want to change, say, the first character of a string from upper to lower case, you have to build a whole new string with a different first character. For example:

```
String bandName = "Foo Fighters";
char c = bandName.charAt(0);
bandName = Character.toLowerCase(c) + bandName.substring(1);
    // bandName now refers to a new string
    //   with the value "foo Fighters"
```

This code changes the reference — `bandName` now refers to your new string with the value `"foo Fighters"` (Figure 8-1).

```
String bandName = "Foo Fighters";
char c = bandName.charAt(0);
bandName = Character.toLower(c) + bandName.substring(1);
```

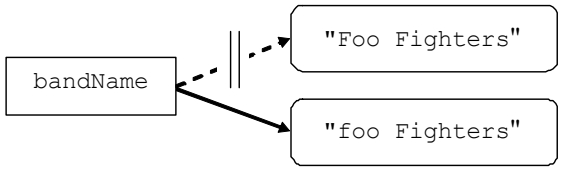


Figure 8-1 A new value assigned to a `String` variable

The old string is thrown away (unless some other variable refers to it). Java's automatic garbage collector releases the memory from the old string and returns it to the free memory pool. This is a little wasteful — like pouring your coffee into a new mug and throwing away the old mug each time you add a spoonful of sugar or take a sip.

However, the immutability of strings makes it easier to avoid bugs. It allows us to have two `String` variables refer to the same string (Figure 8-2-a) without the danger of changing the string contents through one variable without the knowledge of the other. In some cases it also helps avoid copying strings unnecessarily. Instead of creating several copies of the same string —

```
String s2 = new String(s1); // s2 refers to a copy of s1
```

as in Figure 8-2-b — you can use

```
String s2 = s1;  // s2 refers to the same string as s1
```

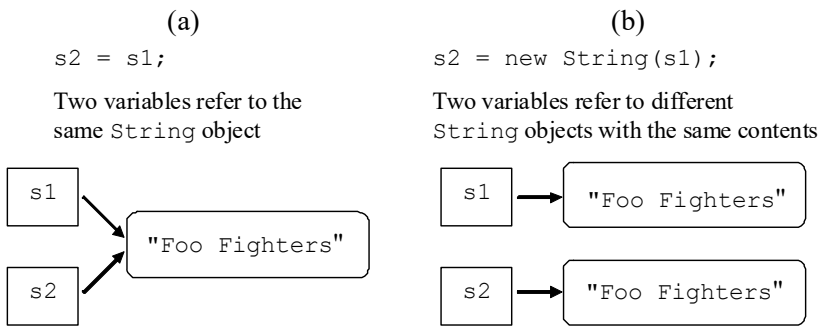


Figure 8-2. Assigning references vs. copying strings

On the other hand, if you build a new string for every little change, a program that frequently changes long strings, represented by `String` objects, may become slow.

Fortunately Java provides another class for representing character strings, called `StringBuffer`. `StringBuffer` objects are not immutable: they have the `setCharAt` method and other methods that change their contents. `String`s may be easier to understand, and they are considered safer in student projects. However, with the `StringBuffer` class we can easily change one letter in a string without moving the other characters around. For example:

```
StringBuffer bandName = new StringBuffer("Foo Fighters");
char c = bandName.charAt(0);
bandName.setCharAt(0, Character.toLowerCase(c));
  // bandName still refers to the same object, but its first
  //   character is now 'f';
```

If some text applications run rather slowly on your fast computer, it may be because the programmer was too lazy to use (or simply never learned about) the `StringBuffer` class. We have summarized `StringBuffer` constructors and methods in Section 8.9. Make sure you've read it before writing commercial applications!

8.4 `String` Methods

The more frequently used `String` methods are summarized in Figure 8-3. There are methods for returning the string's length, for getting the character at a specified position, for building substrings, for finding a specified character or substring in a string, for comparing strings alphabetically, and for converting strings to upper and lower case.

`length` and `charAt`

The `length` method returns the number of characters in the string. For example:

```
String s = "Internet";
int len = s.length();  // len gets the value 8
```

The `charAt` method returns the character at the specified position.

Character positions in strings are counted starting from 0.

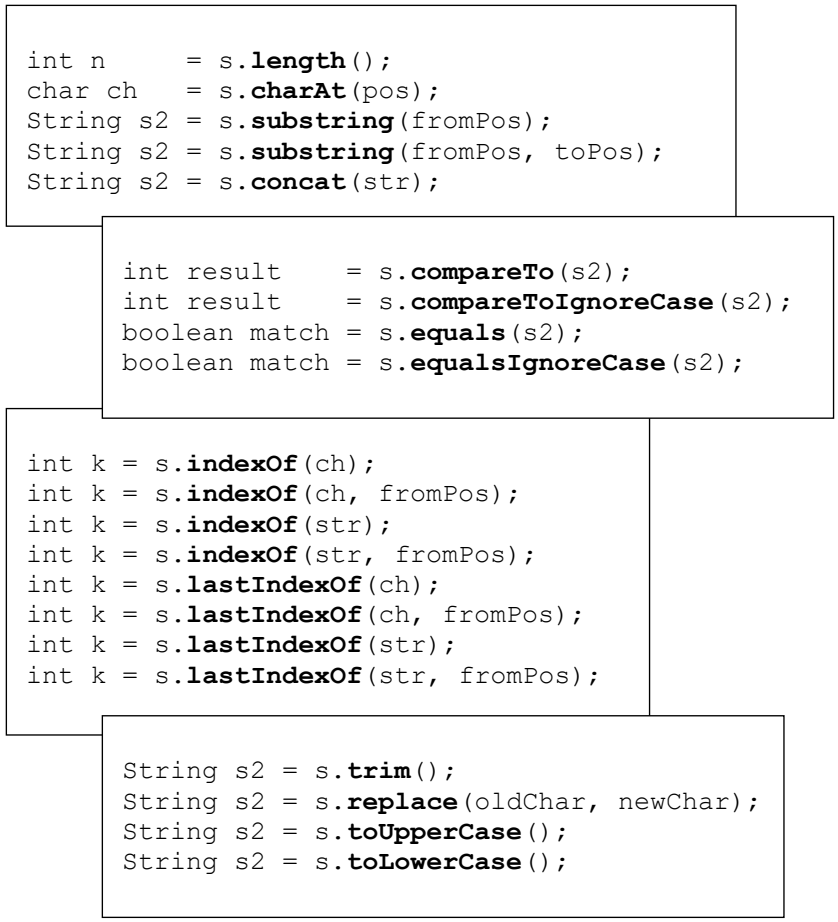


Figure 8-3. Commonly used `String` methods

This convention goes back to the C programming language, where elements of arrays are counted from 0. So the first character of a string is at position (or index) 0, and the last one is at position `s.length()-1`. For example:

```
String s = "Internet";
char c1 = s.charAt(0);     // c1 gets the value 'I'
char c2 = s.charAt(7);     // c2 gets the value 't'
```

If you call `charAt(pos)` with `pos` less than 0 or `pos` greater than or equal to the string length, the method will throw a `StringIndexOutOfBoundsException`.

Always make sure that when you refer to the position of a character in a string, the position is an integer in the range from 0 to string length – 1.

Substrings

The `String` class has two `substring` methods with the same name (methods with the same name but different parameters are called *overloaded*). The first one, `substring(fromPos)`, returns the tail of the string starting from `fromPos`. For example:

```
String s = "Internet";
String s2 = s.substring(5);  // s2 gets the value "net"
```

The second one, `substring(fromPos, toPos)` returns the segment of the string from `fromPos` to `toPos-1`. For example:

```
String s = "Internet";
String s2 = s.substring(0, 5);  // s2 gets the value "Inter"
String s3 = s.substring(2, 6);  // s3 gets the value "tern"
```

Notice that the second parameter is the position of the character following the substring, and that character is not included into the returned substring. The length of the returned substring is always `toPos - fromPos`.

Concatenation

The `concat` method concatenates strings; it works exactly the same way as the string version of the + operator. For example:

```
String s1 = "Sun";
String s2 = "shine";
String s3 = s1.concat(s2);  // s3 gets the value "Sunshine"
String s4 = s1 + s2;        // s4 gets the value "Sunshine"
```

The += operator concatenates the operand on the right to the string on the left. For example:

```
String s = "2*2 ";
s += "= 4";  // s gets the value "2*2 = 4"
```

It may appear at first that the += operator violates the immutability of strings. This is not so. The += first forms a new string by concatenating the right-hand operand to

the original s. Then it changes the reference s to point to the new string. The original string is left alone if some other variable refers to it, or thrown away. So s += s2 accomplishes exactly the same thing as s = s + s2.

As we said in Section 5.10, you can also concatenate characters and numbers to strings using the + and += operators, as long as the compiler can figure out that you are working with strings, not numbers. For example:

```
String s = "Year: ";
s += 1776;   // s gets the value "Year: 1776";
```

But if you write

```
String s = "Year:";
s += ' ' + 1776;  // space in single quotes
```

it won't work as expected because neither ' ' nor 1776 is a String. Instead of concatenating them it will first add 1776 to the Unicode code for a space (32) and then append the sum to s. So s would get the value "Year:1808". On the other hand,

```
String s = "Year:";
s += " " + 1776;       // space in double quotes
```

does work, because the result of the operation " " + 1776 is a String.

Finding characters and substrings

The indexOf(char c) method returns the position of the first occurrence of the character c in the string. Recall that indices are counted from 0. If c is not found in the string, indexOf returns −1. For example:

```
String s = "Internet";
int pos1 = s.indexOf('e');  // pos1 gets the value 3
int pos2 = s.indexOf('x');  // pos2 gets the value -1
```

You can also start searching from a position other than the beginning of the string by using another (overloaded) version of indexOf. It has a second parameter, the position from which to start searching. For example:

```
String s = "Internet";
int pos = s.indexOf('e', 4);  // pos gets the value 6
```

You can search backward starting from the end of the string or from any other specified position using one of the two `lastIndexOf` methods for characters. For example:

```
String s = "Internet";
int pos1 = s.lastIndexOf('e');     // pos1 gets the value 6
int pos2 = s.lastIndexOf('e', 4);  // pos2 gets the value 3
int pos3 = s.lastIndexOf('e', 2);  // pos3 gets the value -1
```

`String` has four similar methods that search for a specified substring rather than a single character. For example:

```
String s = "Internet", s2 = "net";
int pos1 = s.indexOf("e");           // pos1 gets the value 3
int pos2 = s.indexOf("net");         // pos2 gets the value 5
int pos3 = s.indexOf(s2, 6);         // pos3 gets the value -1
int pos4 = s.lastIndexOf(s2);        // pos4 gets the value 5
int pos5 = s.lastIndexOf("net", 6);  // pos5 gets the value 5
```

Comparisons

You cannot use relational operators (==, !=, <, >, <=, >=) to compare strings.

Recall that relational operators == and != when applied to objects compare the objects' references (that is, their addresses), not their values. Strings are no exception. The `String` class provides the `equals`, `equalsIgnoreCase`, and `compareTo` methods for comparing strings. `equals` and `equalsIgnoreCase` are `boolean` methods; they return `true` if the strings have the same length and the same characters (case-sensitive or case-blind, respectively), `false` otherwise. For example:

```
String s = "OK!";
boolean same1 = s.equals("Ok!");           // same1 is set to false
boolean same2 = s.equals("OK");            // same2 is set to false
boolean same3 = s.equalsIgnoreCase("Ok!"); // same3 is set to true
```

Occasionally the string in the comparison may not have been created yet. If you call its `equals` method (or any other method) you will get a `NullPointerException`. For example:

```
private String name;            // name is an instance variable
...
  boolean same = name.equals("Sunshine");
    // NullPointerException if name has not been initialized
```

To avoid errors of this kind you can write

```
boolean same = (name != null && name.equals("Sunshine"));
```

The above statement always works due to short-circuit evaluation (see Section 6.7). However, real Java pros may write

```
boolean same = "Sunshine".equals(name);
```

This always works, whether `name` is initialized or `null`, because you are not calling methods of an uninitialized object. The same applies to the `equalsIgnoreCase` method.

The `compareTo` method returns an integer that describes the result of a comparison. `s1.compareTo(s2)` returns a negative integer if `s1` lexicographically precedes `s2`, `0` if they are equal, and a positive integer if `s1` comes later than `s2`. (To remember the meaning of `compareTo`, you can mentally replace "compareTo" with a minus sign.) The comparison starts at the first character and proceeds until different characters are encountered in corresponding positions or until one of the strings ends. In the former case, `compareTo` returns the difference of the Unicode codes of the characters, so the string with the first "smaller" character (that is, the one with the smaller Unicode code) is deemed smaller; in the latter case `compareTo` returns the difference in lengths, so the shorter string is deemed smaller. This is called "lexicographic ordering," but it is not exactly the same as used in a dictionary because `compareTo` is case-sensitive, and uppercase letters in Unicode come before lowercase letters. For example:

```
String s = "ABC";
int result1 = s.compareTo("abc");
       // result1 is set to a negative number:
       //   "ABC" is "smaller" than "abc"

int result2 = s.compareTo("ABCD");
       // result2 is set to a negative number:
       //   "ABC" is "smaller" than "ABCD"
```

Naturally, there is also a `compareToIgnoreCase` method.

Conversions

Other useful `String` method calls include:

```
String s2 = s1.toUpperCase();
       // s2 is set to a string made up of the characters
       // in s1 with all letters converted to the upper case

String s2 = s1.toLowerCase();
       // Same for lower case

String s2 = s1.replace(c1, c2);
       // s2 is set to a string that has the same
       // characters as s1, except all occurrences of
       // c1 are replaced with c2.

String s2 = s1.trim();
       // s2 is set to the same string as s1, but with
       // the "whitespace" characters (spaces, tabs,
       // and newline characters) trimmed from the
       // beginning and end of the string.
```

For example:

```
String s1 = " <u>String Methods</u>  ";

String s2 = s1.trim();  // s2 becomes "<u>String Methods</u>"
                        // s1 remains " <u>String Methods</u>  "

String s3 = s2.toUpperCase();
                        // s3 becomes "<U>STRING METHODS</U>"
                        // s2 remains "<u>String Methods</u>"

String s4 = s3.replace('U', 'B')
                        // s4 becomes "<B>STRING METHODS</B>"
                        // s3 remains "<U>STRING METHODS</U>"
```

None of these methods (nor any other `String` methods) change the `String` object for which they are called. Instead, they build and return a new string.

This is a potential source of tricky bugs. The names of these methods might imply that they change the string, and it is easy to call them but forget to put the result anywhere. For example:

```
String s1 = " <code>  ";
s1.trim();   // A useless call: s1 remains unchanged!
             // You probably meant s1 = s1.trim();
```

8.5 Formatting Numbers into Strings

As we discussed in Section 5.10, the easiest way to convert a number into a string is to concatenate that number with a string. For example:

```
int n = -123;
String s = "" + n; // s gets the value "-123"
s = "n = " + n;    // s gets the value "n = -123"
double x = -1.23;
s = "" + x;        // s gets the value "-1.23"
```

Java offers two other ways to convert an `int` into a string.

The first way is to use the static method `toString(int n)` of the `Integer` class:

```
int n = -123;
String s = Integer.toString(n);  // s gets the value "-123";
```

The `Integer` class belongs to the `java.lang` package, which is automatically imported into all programs. It is called a *wrapper class* because it "wraps" around a primitive data type `int`: you can take an `int` value and construct an `Integer` object from it. "Wrapping" allows you to convert a value of a primitive type into an object. For example, you might want to hold integer values in a list represented by the Java library class `ArrayList` (see Chapter 11), and `ArrayList` only works with objects. `java.lang` also has the `Double` wrapper class for `double`s and the `Character` wrapper class for `char`s. `Integer` has a method `intValue`, which returns the "wrapped" `int` value, and a method `toString` that returns a string representation of this `Integer` object. For example,

```
Integer obj1 = new Integer(3);
Integer obj2 = new Integer(5);
Integer sum = new Integer(obj1.intValue() + obj2.intValue());
System.out.println(sum);
```

displays 8. Similarly, the `Double` class has a method `doubleValue` and the `Character` class has a method `charValue`.

For now, it is important to know that the `Integer`, `Double`, and `Character` classes offer several "public service" static methods. An overloaded version of `toString`, which takes one parameter, is one of them.

The second way is to use the static method `valueOf` of the `String` class. For example:

```
int n = -123;
String s = String.valueOf(n);    // s gets the value "-123";
```

Similar methods work for `double` values (using the `Double` wrapper class). For example:

```
double x = 1.5;
String s1 = Double.toString(x);  // s1 gets the value "1.5";
String s2 = String.valueOf(x);   // s2 gets the value "1.5";
```

For `double`s, though, the number of digits in the resulting string may vary depending on the value, and a `double` may even be displayed in scientific notation.

It is often necessary to convert a `double` into a string according to a specified format. This can be accomplished by using an object of the `DecimalFormat` library class and its `format` method. First you need to create a new `DecimalFormat` object that describes the format. For example, passing the `"000.0000"` parameter to the `DecimalFormat` constructor indicates that you want a format with at least three digits before the decimal point (possibly with leading zeroes) and four digits after the decimal point. You use that format object to convert numbers into strings. We won't go too deeply into this here, but your programs can imitate the following examples:

```
import java.text.DecimalFormat;
...
    // Create a DecimalFormat object specifying at least one digit
    //   before the decimal point and 2 digits after the decimal point:
    DecimalFormat money1 = new DecimalFormat("0.00");

    // Create a DecimalFormat object specifying $ sign
    //   before the leading digit and comma separators:
    DecimalFormat money2 = new DecimalFormat("$#,##0");

    // Convert totalSales into a string using these formats:
    double totalSales = 12345678.9;
    String s1 = money1.format(totalSales);
        //  s1 gets the value "12345678.90"
    String s2 = money2.format(totalSales);
        //  s2 gets the value "$12,345,679" due to rounding

    // Create a DecimalFormat object specifying 2 digits
    //   (with a leading zero, if necessary):
    DecimalFormat twoDigits = new DecimalFormat("00");
```

```
// Convert minutes into a string using twoDigits:
int minutes = 7;
String s3 = twoDigits.format(minutes);
    //  s3 gets the value "07"
```

If, for example, `totalSales` is 123.5 and you need to print something like

```
Total sales: 123.50
```

you could write

```
System.out.print("Total sales: " +
                         money1.format(totalSales));
```

If `hours` = 3 and `minutes` = 7 and you want the time to look like `3:07`, you could write

```
System.out.print(hours + ":" + twoDigits.format(minutes));
```

Starting with the Java 5.0 release, `PrintStream` and `PrintWriter` objects (including `System.out` and text files open for writing) have a convenient method `printf` for writing formatted output to the console screen and to files. `printf` is an unusual method: it can take a variable number of parameters. The first parameter is always a format string, usually a literal string. The format string may contain fixed text and one or more embedded *format specifiers*. The rest of the parameters correspond to the format specifiers in the string. For example:

```
int month = 5, day = 19, year = 2007;
double amount = 123.5;
System.out.printf("Date: %02d/%02d/%d  Amount: %7.2f\n",
                          month, day, year, amount);
```

displays

```
Date: 05/19/2007  Amount:  123.50
```

Here `%02d` indicates that the corresponding output parameter (`month`, then `day`) must be formatted with two digits including a leading zero if necessary; `%d` indicates that the next parameter (`year`) should be an integer in default representation (with whatever sign and number of digits it might have); `%7.2f` indicates that the next parameter (`amount`) should appear as a floating-point number, right-justified in a field of width 7, with two digits after the decimal point, rounded if necessary. `\n` at the end tells `printf` to advance to the next line. The details of `printf` formatting are rather involved — refer to the Java API documentation.

The Java 5.0 release has also added an equivalent of `printf` for "writing" into a string. The static method `format` of the `String` class arranges several inputs into a formatted string and returns that string. For example:

```
int month = 5, day = 19, year = 2007;
double amount = 123.5;
String msg = String.format("Date: %02d/%02d/%d  Amount: %7.2f",
                           month, day, year, amount);
```

The above statements set `msg` to `"Date: 05/19/2007  Amount:  123.50"`.

8.6 Extracting Numbers from Strings

The reverse operation — converting a string of digits (with a sign, if present) into an `int` value — can be accomplished by calling the static `parseInt` method of the `Integer` class. For example:

```
String s = "-123";
int n = Integer.parseInt(s);   // n gets the value -123
```

What happens if the string parameter passed to `parseInt` does not represent a valid integer? This question takes us briefly into the subject of Java *exception handling*.

If `parseInt` receives a bad parameter, it throws a `NumberFormatException`. You have already seen several occasions when a program "throws" a certain "exception" if it encounters some bug or unexpected situation. This exception, however, is different in nature from the other exceptions that we have experienced up to now, such as `NullPointerException`, `IllegalArgumentException`, or `StringIndexOutOfBoundsException`. Those exceptions are the programmer's fault: they are caused by mistakes in the program. When one of them is thrown, there is nothing to do but to terminate the program and report where the error occurred.

But a `NumberFormatException` may be caused simply by incorrect input from the user. The user will be very surprised if the program quits just because he types an 'o' instead of a '0'. The program should handle such situations gracefully, and Java provides a special tool for that: the `try-catch-finally` statement. You can call `parseInt` "tentatively," within a `try` block, and "catch" this particular type of exception within the `catch` block that follows. The `catch` block is executed only when an exception is thrown. It may be followed by the `finally` block that is always executed and therefore can perform the necessary clean-up. `try`, `catch`, and `finally` are Java reserved words. Figure 8-4 shows how all this may be coded.

```
Scanner input = new Scanner(System.in);
int n = 0;

while (n <= 0)
{
  System.out.print("Enter a positive integer: ");
  String str = input.next();   // read a token
  input.nextLine();  // skip the rest of the line
  try   // try to extract an int from str
  {
    n = Integer.parseInt(str);
  }
  catch (NumberFormatException ex)  // skip this if successful
  {
    System.out.println("*** Invalid input ***");
  }
  finally // either way execute this
  {
    if (n <= 0)
      System.out.println("Your input must be a positive integer");
  }
}

// Process n:
...
```

Figure 8-4. Converting input into an integer with exception handling

A similar method, `parseDouble` of the `Double` class, can be used to extract a `double` value from a string. For example:

```
String s = "1.5";
double x = Double.parseDouble(s);   // x gets the value 1.5
```

8.7 `Character` Methods

When you work with characters and strings, you often need to find out whether a particular character is a digit, a letter, or something else. The `Character` wrapper class has several "public service" static `boolean` methods that test whether a character belongs to a particular category. All of these take one parameter, a `char`, and return `true` or `false`. For example:

```
boolean result = Character.isDigit(c);
                  // result is set to true if c is a digit;
                  //   otherwise result is set to false
```

Other character "category" methods include `isLetter`, `isLetterOrDigit`, `isUpperCase`, `isLowerCase`, and `isWhitespace` (space, tab, newline, etc.).

There are also two methods that return the uppercase and lowercase versions of a character, if these are available. These are called `toUpperCase` and `toLowerCase`. For example:

```
char c1 = Character.toUpperCase('a');  // c1 is set to 'A'
char c2 = Character.toUpperCase('*');  // c2 is set to '*'

Scanner input = new Scanner(System.in);
String firstName = input.next();

// Change the first letter in firstName to upper case:
char c = firstName.charAt(0);
firstName = Character.toUpperCase(c) + firstName.substring(1);
```

8.8 *Lab:* Lipograms

According to *Wikipedia*,

> a *lipogram* (from Greek lipagrammatos, "missing letter") is a kind of writing with constraints or word game consisting of writing paragraphs or longer works in which a particular letter or group of letters is missing, usually a common vowel, the most common in English being *e*.
>
> "*Gadsby* is a notorious book by Californian author E. V. Wright, circa 1939. It was Wright's fourth book. It is famous for consisting only of words not containing any e's. Gadsby is thus a lipogram, or a display of constraint in writing. It is 50,100 words long. Wright informs us in Gadsby's introduction of having had to impair his own typing contraption to avoid slipups."

The *Lipogrammer* program, shown in Figure 8-5, helps to create and verify lipograms. It shows the original text, below it the same text with all letters *e* replaced with #, and to the right, the list of all 'offending' words (with an *e* in them). The user can load a lipogram text from a file or type it in or cut and paste it from another program. There is also a menu command to save the text. In this lab, you will write the `LipogramAnalyzer` class for this program.

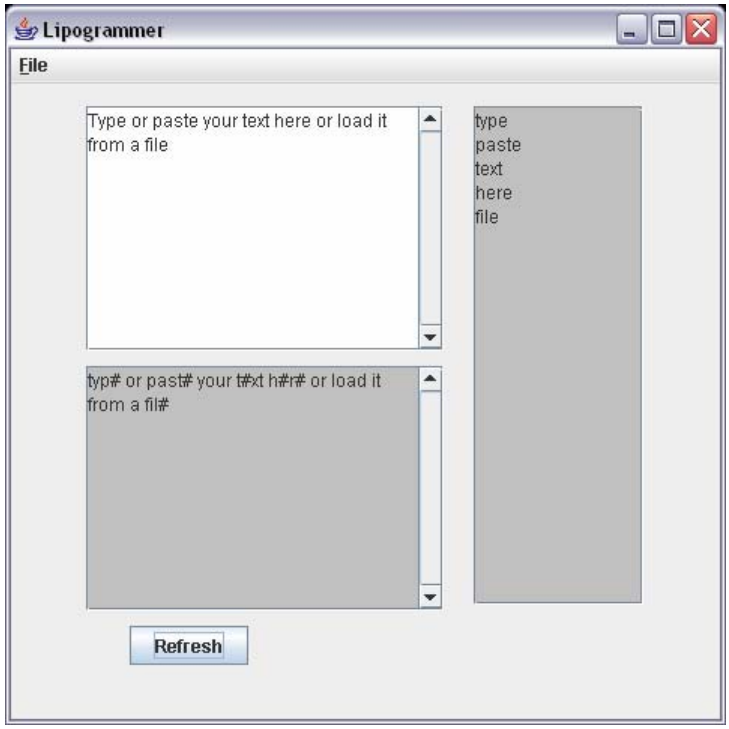


Figure 8-5. The *Lipogrammer* program

The `LipogramAnalyzer` should have the following constructor and two public methods:

Constructor Summary	
`public LipogramAnalyzer(String text)` Saves the `text` string.	
Method Summary	
`String`	`mark(char letter)` Returns the saved `text` string with all characters equal to `letter` replaced with `'#'`.
`String`	`allWordsWith(char letter)` Returns a string that concatenates all "offending" words from `text` that contain `letter`; the words are separated by `'\n'` characters; the returned string does not contain duplicate words: each word occurs only once; there are no punctuation or whitespace characters in the returned string other than `'\n'` characters.

Hint: write a private method to extract and return the word that contains the character at a specified position in `text`. Find the boundaries of the word by scanning the text to the left and to the right of the given position.

Combine in one project your `LipogramAnalyzer` class with the `Lipogrammer` and `LipogrammerMenu` GUI classes from the `JM\Ch08\Lipogrammer` folder. Test the program.

8.9 The `StringBuffer` Class

`StringBuffer` objects represent character strings that can be modified. Recall that `String` objects are immutable: you cannot change the contents of a string once it is created, so for every change you need to build a new string. To change one or several characters in a string or append characters to a string, it is usually more efficient to use `StringBuffer` objects.

This is especially true if you know in advance the maximum length of a string that a given `StringBuffer` object will hold. `StringBuffer` objects distinguish between the current capacity of the buffer (that is, the maximum length of a string that this buffer can hold without being resized) and the current length of the string held in the buffer. For instance, a buffer may have the capacity to hold 100 characters and be empty (that is, currently hold an empty string). As long as the length does not exceed the capacity, all the action takes place within the same buffer and there is no need to reallocate it. When the length exceeds the capacity, a larger buffer is allocated automatically and the contents of the current buffer are copied into the new buffer. This takes some time, so if you want your code to run efficiently, you have to arrange things in such a way that reallocation and copying do not happen often.

The `StringBuffer` class has several constructors. Among them:

```
StringBuffer()          // Constructs an empty string buffer with the
                        //   default capacity (16 characters)
StringBuffer(int n)     // Constructs an empty string buffer with the
                        //   capacity n characters
StringBuffer(String s) // Constructs a string buffer that holds
                        //   a copy of s
```

Figure 8-6 shows some of `StringBuffer`'s more commonly used methods at work. As in the `String` class, the `length` method returns the length of the string currently held in the buffer. The `capacity` method returns the current capacity of the buffer.

In addition to the `charAt(int pos)` method that returns the character at a given position, `StringBuffer` has the `setCharAt(int pos, char ch)` method that sets the character at a given position to a given value.

`StringBuffer` has several overloaded `append(`*`sometype`* `x)` methods. Each of them takes one parameter of a particular type: `String`, `char`, `boolean`, `int`, and other primitive types, `Object`, or a character array (Chapter 9). `x` is converted into a string using the default conversion method, as in `String.valueOf(…)`. Then the string is appended at the end of the buffer. A larger buffer is automatically allocated if necessary. The overloaded `insert(int pos,` *`sometype`* `x)` methods insert characters at a given position.

The `substring(fromPos)` and `substring(fromPos, toPos)` methods work the same way as in the `String` class: the former returns a `String` equal to the substring starting at position `fromPos`, the latter returns a `String` made of the characters between `fromPos` and `toPos-1`, inclusive. `delete(fromPos, toPos)` removes a substring from the buffer and `replace(fromPos, toPos, str)` replaces the substring between `fromPos` and `toPos-1` with `str`. Finally, the `toString` method returns a `String` object equal to the string of characters in the buffer.

```
StringBuffer sb = new StringBuffer(10); // sb is empty

int len = sb.length();                  // len is set to 0
int cap = sb.capacity();                // cap is set to 10

sb.append("at");                        // sb holds "at"
sb.insert(0, 'b');                      // sb holds "bat"

char ch = sb.charAt(1);                 // ch is set to 'a'
sb.setCharAt(0, 'w');                   // sb holds "wat"

sb.append("er");                        // sb holds "water"
sb.replace(1, 3, "int");                // sb holds "winter"

String s1 = sb.substring(1);            // s1 is set to "inter"
String s2 = sb.substring(1, 3);         // s2 is set to "in"
sb.delete(4, 6);                        // sb holds "wint"
sb.deleteCharAt(3);                     // sb holds "win"

sb.append(2020);                        // sb holds "win2020"
String str = sb.toString();             // str is set to "win2020"
```

Figure 8-6. Examples of common `StringBuffer` methods

8.10 Summary

Text in double quotes represents a *literal string*. Literal strings are treated as string objects, and you can assign them to `String` references without explicitly creating a string with the `new` operator. Literal strings may include *escape characters*, such as `\n`, `\t`, `\\`, and `\"`.

Once a string is constructed, there is no way to change it because objects of the `String` class are *immutable*: no `String` method can change its object. However, you can reassign a `String` reference to another string. If no other variable refers to the old string, it will be released by Java's garbage collection mechanism. This may be quite inefficient; professional programmers often use the `StringBuffer` class instead of `String`.

Java supports the + and += operators for concatenating strings. If several + operators are combined in one expression for concatenation, you have to make sure that at least one of the operands in each intermediate operation is a string.

Figure 8-3 summarizes the most commonly used `String` methods. The positions of characters in strings are counted from 0, so the `charAt(0)` method returns the first character of a string.

The == or `!=` operators are usually not useful for comparing strings because these operators compare the references to (addresses of) strings, not their contents. Use the `equals`, `equalsIgnoreCase`, and `compareTo` methods instead.

The `Integer` class is what is called a *wrapper class* for the primitive data type `int`. It provides a way to represent an integer value as an object. The static methods `Integer.toString(int n)` or `String.valueOf(int n)` return a string: a representation of `n` as a string of digits, possibly with a sign. The same can also be accomplished with `"" + n`. With `double`s it is better to use a `DecimalFormat` object or the static method `format` in `String` for conversion into strings. For example:

```
DecimalFormat moneyFormat = new DecimalFormat("0.00");
...
String s1 = moneyFormat.format(totalSales);
String s2 = String.format("%.2f", totalSales);
```

To convert a string of decimal digits into an `int` value, call the static `parseInt` method of the `Integer` class. For example:

```
String s = "-123";
int n = Integer.parseInt(s);   // n gets the value -123
```

This method throws a `NumberFormatException` if `s` does not represent an integer. Your program should be able to catch the exception, alert the user, and keep running (see Figure 8-4).

The `Character` class (a wrapper class for `char`) has useful `boolean` static methods `isLetter`, `isDigit`, `isWhitespace`, and a few others that take a `char` as a parameter and return `true` or `false`. `Character`'s other two static methods, `toUpperCase(ch)` and `toLowerCase(ch)`, return a `char` value equal to `ch` converted to the appropriate case.

`StringBuffer` objects represent character strings that can be modified. In addition to `length`, `charAt`, `substring`, `indexOf`, and some other `String` methods, the `StringBuffer` class has `setCharAt`, `append`, `insert`, `deleteCharAt` and `delete` methods.

Exercises

Sections 8.1-8.4

1. Find a bug in the following declaration: ✓

```
String fileName = "C:\dictionaries\words.txt";
```

2. (a) Write a method that returns `true` if a given string is not empty and ends with a star (`'*'`), `false` otherwise. ✓

(b) Write a method that returns `true` if a given string has at least two characters and ends with two stars, `false` otherwise.

3. Write a method that eliminates two dashes from a social security number in the format "ddd-dd-dddd" and returns a 9-character string of digits. For example, `removeDashes("987-65-4321")` returns a string equal to `"987654321"`.

4. (a) A string `dateStr` represents a date in the format "mm/dd/yyyy" (for example, `"05/31/2019"`). Write a statement or a fragment of code that changes `dateStr` to the format "dd-mm-yyyy" (for example, `"31-05-2019"`). ✓

(b)■ Make the code in Part (a) more general, so that it can handle dates written with or without leading zeroes (for example, it should convert `"5/3/2022"` into `"03-05-2022"`).

(c) Use the program in `JM\Ch08\Exercises\StringTest.java` to test the code in Part (a) and Part (b) and for other exercises.

5. A credit card number is represented as a `String ccNumber` that contains four groups of four digits. The groups are separated by one space. For example:

```
String ccNumber = "4111 1111 1111 1111";
```

(a) Write a statement that declares a string `last4` and sets it to the last four digits in `ccNumber`. ✓

(b) Write a statement that sets `String last5` to a string that holds the last five digits in `ccNumber`.

6. Write a `scroll` method that takes a string as a parameter, moves the first character to the end of the string, and returns the new string.

7. Suppose a string holds a person's last name and first name, separated by a comma. Write a method `convertName` that takes such a string and returns a string where the first name is placed first followed by one space and then the last name. For example:

```
String firstLast = convertName("von Neumann, John");
   // firstLast is set to "John von Neumann"
```

≤ Hint: `trim` helps get rid of extra white space. ≥

8.■ A string contains only `'0'` and `'1'` characters and spaces. Write a method that takes such a string and makes and returns a "negative" string in which all the 0s are replaced with 1s and all the 1s with 0s. Your method must rely only on `String`'s methods and not use any explicit iterations or recursion.

9.◆ Write a method that determines whether all the characters in a string are the same, using only library `String` methods, but no loops or recursion.
≤ Hint: there are several approaches. For example, see Question 6 above. ≥

10. Write a method that tries to find the first opening comment mark (`"/*"`) and the last comment closing mark (`"*/"`) in a string. If both are found and the closing mark is after the opening mark, the method removes the first opening mark, the last closing mark, and all the characters between them from the string and returns the new string. If it fails to find both marks in the correct order, the method returns the original string unchanged. Your method must rely only on `String`'s methods and not use any iterations explicitly.

11. Write a method `cutOut` that removes the first occurrence of a given substring (if found) from a given string. For example:

```
String str = "Hi-ho, hi-ho";
String result = cutOut(str, "-ho");
  // result is set to "Hi, hi-ho"
```

✓

12.■ Write your own implementation of `indexOf(ch, fromPos)`.

13. The `String` class has `boolean` methods `startsWith(String prefix)` and `endsWith(String suffix)`. `startsWith` tests whether this string starts with a given substring; `endsWith` tests whether this string ends with a given substring. Pretending that these methods do not exist, write them using other string methods (but no iterations or recursion).

14.■ Web developers use HTML tags in angle brackets to format the text on web pages. Write a method `removeTag` that checks whether a given string starts with an apparent HTML tag (a character or word in angle brackets) and ends with a matching closing HTML tag (the same character or word preceded by the '/' character, all in angle brackets). If yes, the method removes both tags and returns the result; otherwise the method returns the original string unchanged. For example,
`removeTag("<b>Strings are immutable</b>")` should return a string equal to `"Strings are immutable"`.

Sections 8.5-8.10

15. Write and test a method that tests whether a given string contains only digits. ✓

16. If two strings, `s1` and `s2`, represent positive integers `n1` and `n2` in the usual way, as sequences of decimal digits, is it true that the sign of `s1.compareTo(s2)` is always the same as the sign of `(n1 - n2)`? Write a simple console application that prompts the user to enter two strings and tests this "hypothesis."

17.■ In *MS-DOS*, a file name consists of up to eight characters (excluding `'.'`, `':'`, backslash, `'?'`, and `'*'`), followed by an optional dot (`'.'` character) and extension. The extension may contain zero to three characters. For example: `1STFILE.TXT` is a valid file name. File names are case-blind. Write and test a method

```
private String validFileName(String fileName)
```

that validates the input, appends the default extension `".TXT"` if no extension is given (that is, no `'.'` appears in `fileName`), converts the name to the upper case, and returns the resulting string to the caller. If `fileName` ends with a dot, remove that dot and do not append the default extension. If the name is invalid, `validFileName` should return `null`.

18.■ (a) Write a method

```
public boolean isPalindrome(String word)
```

that tests whether `word` is a palindrome (the same when read forward or backward, as in "madam"). Test `isPalindrome` using the appropriately modified *String Test* program (`JM\Ch08\Exercises\StringTest.java`).

(b) Upgrade `isPalindrome` so that it can handle any phrase (as in "Madam, I'm Adam"). In testing for a palindrome, disregard all spaces, punctuation marks, apostrophes, and other non-alphanumeric characters and consider lower- and uppercase letters the same. Do not count an empty string as a palindrome. ≤ Hint: recall that the `Character` class has static methods `boolean isLetterOrDigit(ch)` and `char toUpperCase(ch).` ≥

19.■ The program *Cooney* (JM\Ch08\Exercises\cooney.jar plays a game in which the player tries to guess which words Cooney "likes" and which ones Cooney "doesn't like." After five correct guesses in a row Cooney congratulates the player and the game stops. Play the game and guess the rule; then write the *Cooney* program. ≤ Hint: write a console application or use JM\Ch08\Exercises\StringTest.java as a basis for your program. ≥

20.■ An ISBN (International Standard Book Number) has thirteen digits. The first twelve digits identify the country in which the book was printed, the publisher, and the individual book. The thirteenth digit is the "check digit." It is chosen in such a way that

$$(d_1 + 3d_2 + d_3 + 3d_4 + ... + d_{11} + 3d_{12} + d_{13}) \mod 10 = 0$$

where $d_1, d_2, ..., d_{13}$ are the digits of the ISBN. "mod" stands for modulo division (same as % in Java).

Note that if we simply took the sum of all the digits, the check digit would remain valid for any permutation of the digits. Different coefficients make the number invalid when any two consecutive digits are swapped, catching a common typo.

Write a method

```
public static boolean isValidISBN(String isbn)
```

that returns `true` if `isbn` represents a valid ISBN, `false` otherwise. Test your method thoroughly: use ISBNs from several books as well as several entries that are not valid ISBNs to test your method. ≤ Hint: The `Character` class has the `static int` method `digit(char ch, int base)` that returns the numeric value of the digit in the specified base. For example, `Character.digit('7', 10)` returns `7`. ≥

21.■ Write a class `HangmanGame` that can help someone implement the *Hangman* game. (Do a web search for "Hangman" to find the rules and lots of versions of the program on the Internet.)

Provide three fields: a `String` to hold the answer word; a `StringBuffer` (of the same length as the word) to hold the partially filled string, with dashes for the letters that have not been guessed yet; and a `StringBuffer` to hold all the letters tried (with no duplicates), initially empty.

Provide a constructor that initializes the answer to a given string and all the other fields appropriately. Provide accessor methods: `String getWord()`, `String getGuessed()`, and `String getTried()` (note that accessors for the `StringBuffer` fields return `String`s). Finally, provide a method `int tryLetter(char letter)` that processes the player's next attempt. `tryLetter` should make the necessary adjustments to the current state of the game and return 0 if the letter has been tried before, −1 if it is not in the word, and 1 if the guess was successful.

Combine your `HangmanGame` class with a simple main class `Hangman`, provided in `JM\Ch08\Exercises\Hangman.java`, and test the program.

22.◆ Write and test a method

```
public String shuffle(String abc)
```

that returns a new string with all the characters from `abc` rearranged in random order. Your method must first create a temporary `StringBuffer` object from `abc`, then shuffle the characters in that string buffer, then convert the string buffer back into a string and return the result. Read the Java API documentation for the `StringBuffer` class.

To shuffle the characters use the following algorithm:

Set *n* to the total number of characters in the string buffer
While *n* is greater than 1, repeat the following steps:
    Pick a random character among the first *n*
    Swap that character with the *n*-th character
    Decrement *n* by 1

```
Section[] chapter9 =
                new Section[9];
```

Arrays

9.1 Prologue

Java programmers can declare several consecutive memory locations of the same data type under one name. Such a memory block is called an *array*, and the individual memory locations are called the *elements* of the array. The number of elements is called the *size* or *length* of the array. Your program can refer to an individual element of an array using the array name followed by the element's number (called its *index* or *subscript*) in brackets. An index can be any integer constant, variable, or expression.

There are many good reasons for using arrays. Suppose your program requires you to enter a number of integers, such as test scores, and calculate their average. Of course you could try to hold the entered values in separate variables, `score1`, `score2`, ... But this would not work very well. First, since you might not know in advance how many scores would be entered, you would have to declare as many variables as the maximum possible number of inputs. Then you would have to read each score individually:

```
score1 = input.nextInt();
score2 = input.nextInt();
...
```

This could get tedious. And then adding the scores up would require a separate statement for each addition:

```
int sum = 0;
sum += score1;
sum += score2;
...
```

Now suppose you wanted to see the *k*-th score. Imagine programming it like this:

```
if (k == 1)
  System.out.print(score1);
else if (k == 2)
  System.out.print(score2);
else if < ... etc. >
```

Fortunately, arrays make the coding of such tasks much easier. You can write

```
int sum = 0;
for (int k = 0; k < scores.length; k++)
  sum += scores[k];
```

and

```
System.out.print(scores[k]);
```

Here `scores[0]` refers to the first score, `scores[1]` to the second, and so on.

In this chapter we will discuss how to

- Declare and create arrays
- Access elements of an array using indices
- Access an array's length
- Pass arrays to methods
- Declare and create two-dimensional arrays
- Traverse an array using a "for each" loop

9.2 One-Dimensional Arrays

Java treats arrays as objects of the type "array of `int`s," "array of `double`s," "array of `String`s," and so on. You can have an array of elements of any type. Use empty brackets after the type name to indicate that a variable refers to an array, as follows:

```
sometype[] someName;
```

For example:

```
int[] scores;
```

Arrays in Java are similar to objects in some ways.

In particular, you need to first declare an array-type variable, then create the array using the `new` operator. You can declare and create an array in the same statement. For example, the following statement declares an array of integers, called `scores`, and creates that array with 10 elements:

```
int[] scores = new int[10];
```

An array of 5000 strings and an array of 16 "colors" can be declared and created as follows:

```
String[] words = new String[5000];
Color[] colors = new Color[16];
```

Note that brackets, not parentheses, are used here with the `new` operator. The number in brackets, as in `new int[10]` or `new String[5000]`, indicates the size (length) of the array.

When an array is created, its elements are initialized to default values. Numeric elements are initialized to 0, `boolean` to `false`.

If array elements are of a class type, then the array contains references to objects of that type; these are initialized to `null`.

If the array elements are references, you have to initialize each element by setting it to a valid reference before that element is used. For example:

```
colors[0] = new Color(207, 189, 250);
colors[1] = Color.BLUE;
```

and so on. You will get a `NullPointerException` if you call a method of a non-existing object (the reference is `null`).

Another way to declare and create an array is to list explicitly, between braces, the values of all its elements. The `new` operator is not used in this form.

For example:

```
int[] scores = {95, 97, 79, 99, 100};
String[] names = {"Vikas", "Larisa", "Jun"};

Color[] rainbowColors =
{
  Color.RED, Color.ORANGE, Color.YELLOW, Color.GREEN,
  Color.CYAN, Color.BLUE, Color.MAGENTA
};
```

The first statement creates an array of five integers; the second creates an array of three strings; the third creates an array of seven colors. The initial values within braces can be any constants, initialized variables, or expressions of the same type as given in the array declaration.

In Java, once an array is declared and initialized, either with the `new` operator or with a list of values, it is not possible to change its size.

To increase the size, you have to create a larger array, copy the values from the original array to the new one, reassign the name of the original array to the new array, and abandon the original array (to be picked up by the garbage collector).

❖ ❖ ❖

A program can access individual elements of an array using *indices* (also called *subscripts*). An index is an integer value placed in square brackets after the array name to identify the element. The elements of an array are numbered starting from 0.

The following statements declare an array of 100 integer elements:

```
final int MAXCOUNT = 100;
int[] a = new int[MAXCOUNT];
```

The elements of this array can be referred to as `a[0]`, `a[1]`, ... , `a[99]`.

The power of arrays lies in the fact that an index can be any integer variable or expression. A program can refer, for example, to `a[i]`, where `i` is an integer variable. When the program is running, it interprets `a[i]` as the element of the array whose index is equal to whatever value `i` currently has. For example, if the variable `i` gets the value `3` and the program accesses `a[i]` at that point, `a[i]` will refer to `a[3]`, which is the fourth element of the array (`a[0]` being the first element). The index can be any expression with an integer value. For example:

```
double[] coordinates = new double[12];
for (int i = 0; i < 6; i++)
{
  coordinates[2 * i] = i;
  coordinates[2 * i + 1] = Math.sqrt(i);
}
```

❖ ❖ ❖

In Java, every array "knows" its own size (length). Java syntax allows you to access the length of an array by using the expression *`arrayName`*`.length`. In terms of syntax, `length` acts as a public field that holds the size of the array.

In arrays, `length` is not a method (as in the `String` class). It is accessed like a field, without parentheses.

For example:

```
double[] samples = new double[10];
...
if (i >= 0 && i < samples.length)
  ...
```

All indices must fall in the range from `0` to *length* – 1, where *length* is the number of elements in the array. If an index happens to be out of this range, your program will throw an `ArrayIndexOutOfBoundsException`.

You can set any element in an array with a simple assignment statement:

```
a[i] = < constant, variable, or expression >;
```

Arrays are always passed to methods as references.

If an array is passed to a method, the method gets the address of the original array and works with the original array, not a copy. Therefore, a method can change an array passed to it. For example:

```
// Swaps a[i] and a[j]
public void swap(double[] a, int i, int j)
{
  double temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}
```

A method can also create a new array and return it (as a reference). For example:

```
public double[] readScores()  // return type is an array of doubles
{
  double[] scores = new double[10];

  for (int i = 0; i < 10; i++)
    scores[i] = readOneScore();

  return scores;
}
```

9.3 *Lab:* Fortune Teller

The window in Figure 9-1 is from the *Fortune Teller* program. When the user presses the "Next" button, the program displays a message randomly chosen from an array of messages. The program is implemented in one class, `FortuneTeller`.

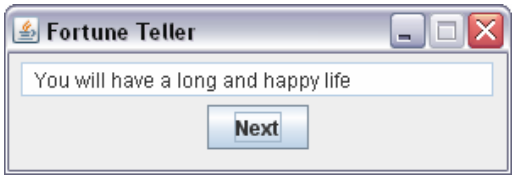


Figure 9-1. The *Fortune Teller* program

Set up a project with the `FortuneTeller.java` and `ding.wav` files from `JM\Ch09\Fortunes`. Add `EasyClasses.jar` in `JM\EasyClasses\` as a required library. Fill in the blanks in `FortuneTeller.java`, adding an array of a few "fortunes" (strings) and the code to randomly choose and display one of them. Recall that the static `Math.random` method returns a random `double` value x such that $0 \le x < 1$. We have used it in earlier programs (for example, `JM\Ch06\Craps\Die.java`). Scale the value returned by `Math.random` appropriately to obtain a random value for an index within the range of your array. Use `display`'s `setText` method to show the chosen message.

9.4 Two-Dimensional Arrays

Programmers use two-dimensional arrays to represent rectangular tables of elements of the same data type. For instance, a 2D array may hold positions in a board game, elements of a matrix, or pixel values in an image.

The following example shows two ways to declare and initialize a 2D array of `doubles`:

```
int rows = 2;
int cols = 3;

double[][] a = new double[rows][cols]; // Declares an array of doubles
                                       //   with 2 rows and 3 columns
                                       //   and sets them to 0

double[][] b =                         // Declares a 2 by 3 array of
{                                      //   doubles initialized to
  {0.0, 0.1, 0.2},                     //   specified values
  {1.0, 1.1, 1.2}
};
```

We access the elements of a 2D array with a pair of indices, each placed in square brackets. We can think of the first index as a "row" and the second as a "column." Both indices start from 0.

In the above example,

```
b[0][0] = 0.0;    b[0][1] = 0.1;    b[0][2] = 0.2;
b[1][0] = 1.0;    b[1][1] = 1.1;    b[1][2] = 1.2;
```

In Java, a 2D array is represented as a 1D array of 1D arrays, its rows. Each row is an array.

In the example above, `b[0]` is the first row, which contains the values 0.0, 0.1, and 0.2. `b[1]` is the second row of values: 1.0, 1.1, and 1.2. Strictly speaking, it is possible for different rows in a 2D array to have different numbers of "columns." In this book, we will deal only with "rectangular" 2D arrays that have the same number of elements in all rows.

If `m` is a 2D array, then `m.length` is the number of rows in the array, `m[0]` is the first row (a 1D array), and `m[0].length` is the number of columns (in the first row). `m[r][c]` is the element in row `r` and column `c`.

It is convenient to use nested `for` loops to traverse a 2D array. For example:

```
int rows = 12, cols = 7;
char[][] grid = new char[rows][cols];
    ...
// Set all elements in grid to '*':
for (int r = 0; r < rows; r++)
{
  for (int c = 0; c < cols; c++)
  {
    grid[r][c] = '*';
  }
}
```

(We find that `r` and `c` or `row` and `col` are often better choices for the names of the indices than, say, `i` and `j`.)

Braces are optional here since the body of each loop consists of only one statement. You could just as well write:

```
for (int r = 0; r < rows; r++)
  for (int c = 0; c < cols; c++)
    grid[r][c] = '*';
```

The following code fragment prints out the values of a 2D array of `int`s `m`:

```
for (int r = 0; r < m.length; r++)
{
  for (int c = 0; c < m[r].length; c++)
  {
    System.out.printf("%5d ", m[r][c]);
  }
  System.out.println();
}
```

Be careful when using `break` in nested loops: a `break` statement inside a nested loop will only break out of the inner loop.

For example, suppose you have a 2D array of characters `grid` and you want to find the first occurrence of the letter 'A' in it (scanning the first row left to right, then the next row, etc.). You might be tempted to use the following code:

```
int rows = grid.length, cols = grid[0].length;
int firstArow = -1, firstAcol = -1;

for (int r = 0; r < rows; r++)
{
  for (int c = 0; c < cols; c++)
  {
    if (grid[r][c] == 'A')
    {
      firstArow = r;
      firstAcol = c;
      break;
    }
  }
}
```

Unfortunately, it will find the first occurrence of 'A' in the last row where one exists.

You can declare three-dimensional and multi-dimensional arrays in a manner similar to two-dimensional arrays. Arrays in three or more dimensions are not used very often. In this book we never go beyond two dimensions.

9.5 *Case Study and Lab:* Chomp

The game of Chomp can be played on a rectangular board of any size. The board is divided into squares (let's say the board represents a chocolate bar). The rules are quite simple: the two players alternate taking rectangular "bites" from the board. On each move, the player must take any one of the remaining squares as well as all the squares that lie below and to the right of it (Figure 9-2). The square in the upper-left corner of the board is "poison": whoever takes it loses the game. Click on the `chomp.jar` file in the `JM\Ch09\Chomp` to run the *Chomp* program (`chomp.wav` must be in the same folder.)

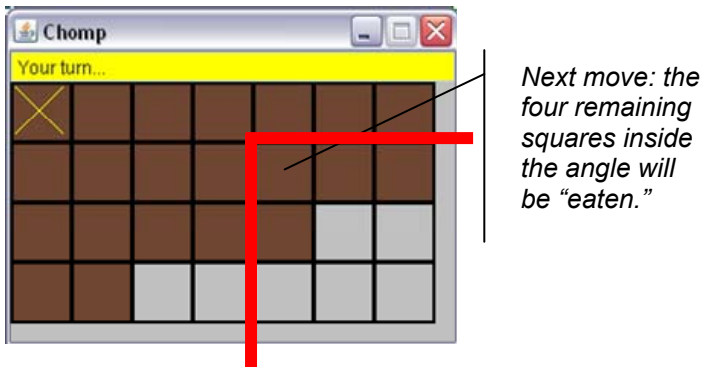


Figure 9-2. The Chomp game program

The number of all possible positions in Chomp is finite, and the players make steady progress from the initial position to the end, as the total number of remaining "edible" squares on the board decreases with each move. Games of this type always have a winning strategy either for the first or for the second player. But, despite its simple rules, Chomp turns out to be a tricky game: you can prove mathematically that the first player has a winning strategy, but the proof does not tell you what that strategy is.* You know you can win if you go first, but you don't know how! Frustrating...

As far as we know, at the time of this writing no one has been able to come up with a formula for the winning Chomp positions (except for two special cases, the 2 by *n* and *n* by *n* boards). There are computer programs that can backtrack from the final position (where only the "poison" square is left) and generate a list of all the winning positions. Our *Chomp* program uses such a list, so the computer has an unfair advantage. You could try to "steal" the winning moves from the computer, but the program's author has foreseen such a possibility and programmed the computer to intentionally make a few random moves before it settles into its winning strategy.

Luckily our goal here is not to beat the computer at Chomp, but to practice object-oriented software design and Java programming.

---

* The proof goes like this. The first player can try to take the lower right corner on the first move. If this is the correct move in a winning strategy, the first player is all set. If it is not, the second player must have a winning move in response. But the first player could "steal" that winning response move and make it his own first move! In the theory of finite games, this argument is called "strategy stealing." Unfortunately, this proof gives no clue as to what the winning strategy might be.

❖ ❖ ❖

Let us begin by looking at the overall structure of this program (Figure 9-3).

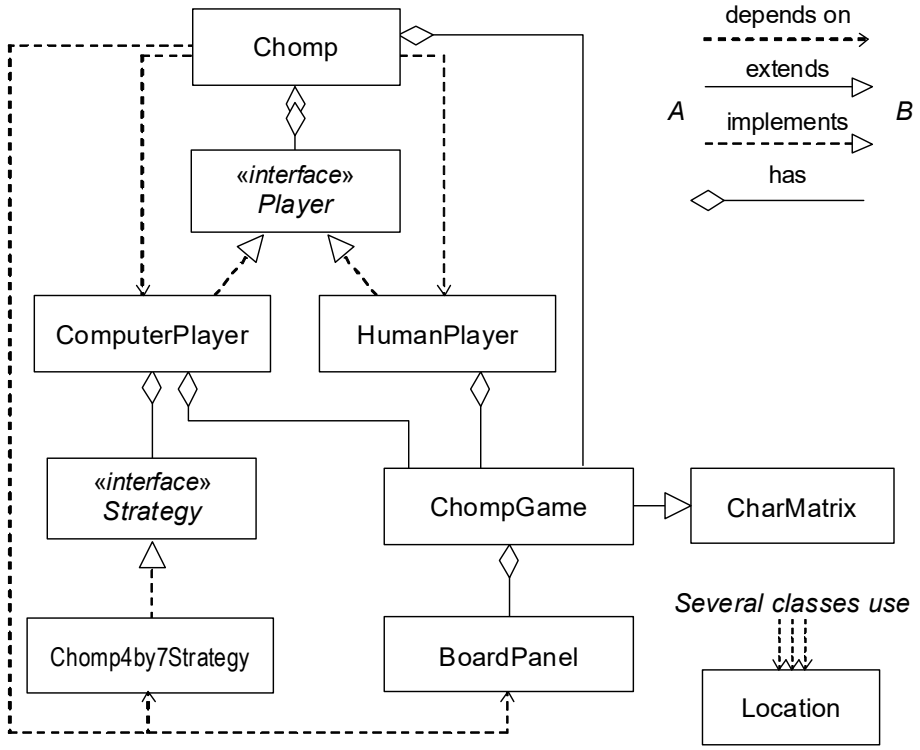


Figure 9-3. The classes in the *Chomp* program

The program consists of eight classes and two *interfaces*, `Player` and `Strategy`. (In Java, an interface specifies which methods must be defined in a class that *implements* that interface. We will explain interfaces in Chapter 12). In designing this program we tried, as usual, to reduce *coupling* (dependencies between classes) and to separate the logic/calculations part from the GUI. `Chomp`, `HumanPlayer`, and `ComputerPlayer`, for instance, know very little about `BoardPanel`. `ComputerPlayer` is the only class that is aware of the `Strategy` methods (but not of any particular Chomp strategy).

The top class, `Chomp`, derived from `JFrame`, represents the main program window. Its constructor creates a `ChompGame`, a `BoardPanel` (the display panel for the board), and the human and computer "players." It also attaches a particular strategy to the computer player:

```
BoardPanel board = new BoardPanel();
...

game = new ChompGame(board);

HumanPlayer human = new HumanPlayer(this, game, board);
ComputerPlayer computer = new ComputerPlayer(this, game, board);
computer.setStrategy(new Chomp4by7Strategy());
```

A `ChompGame` object models Chomp in an abstract way. It knows the rules of the game, and, with the help of its superclass `CharMatrix`, keeps track of the current board position and implements the players' moves. But `ChompGame` does not display the board — that function is left to a separate "view" class, `BoardPanel`. `ChompGame` only represents the "model" of the game. This class and the `Chomp4by7Strategy` class are really the only classes that "know" and use the rules of Chomp.

The `ChompGame` class extends `CharMatrix`, a general-purpose class that represents a 2D array of characters. A matrix of characters helps `ChompGame` to represent the current configuration of the board. We could potentially put all the code from `CharMatrix` directly into `ChompGame`. That would make the `ChompGame` class quite large. More importantly, general methods dealing with a character matrix would be mixed together in the same class with more specific methods that deal with the rules of Chomp. Such a class would hardly be reusable. By separating the more general functions from the more specific ones, we have created a reusable class `CharMatrix` without any extra effort.

`BoardPanel` is derived from the library class `JPanel`, and it knows how to display the current board position. `Chomp` adds the display to the main window's content pane and also attaches it to `ChompGame`. When the board situation changes, `ChompGame` calls `BoardPanel`'s `update` method to update the screen. `BoardPanel` is the longest class in this project. Its code has to deal with rendering different squares in different colors and to support the "flashing" feedback for computer moves.

The `HumanPlayer` and `ComputerPlayer` objects obviously represent the two players. The `HumanPlayer` captures and interprets mouse clicks on the board and tells `ChompGame` to make the corresponding move. The `ComputerPlayer` asks the `Strategy` given to it to supply the next move and tells `ChompGame` to make that move.

Finally, the `Location` class represents a (*row*, *col*) location on the board. `Location` objects are simple "data carriers": `Location` has one constructor, `Location(row, col)`, and two accessors, `getRow()` and `getCol()`. In the *Chomp* program, `Location` objects are passed from the board to the human player and from a `Strategy` object to the computer player.

❖ ❖ ❖

You might have noticed that our designs of *Craps* and *Chomp* have some similarities. This is not a coincidence: we have followed established *design patterns*. Design patterns represent an attempt by experienced designers to formalize their experience and share it with novices. Design patterns help solve common design problems and avoid common mistakes.

In the projects mentioned above, we used the *Model-View-Controller* (*MVC*) design pattern.MVC The main idea of such a design is to clearly separate the "model" (a more abstract object that describes the situation) from the "controller" (the object that changes the state of the model) and from the "view" (the object that displays the model). The "view" is attached to the model and changes automatically (well, almost automatically) when the model changes. This way we isolate GUI from number crunching. Also, we can easily use several different controllers and attach several different views to the same model if we need to.

In the *Chomp* program, the `ChompGame` class is the "model," the `BoardPanel` object is the "view," and the "players" work as "controllers."

Chomp also follows the *Strategy* design pattern. This design pattern applies when a "player" needs to follow different "strategies" at different times. Rather than making the player itself create all the strategies and choose among them, we provide a "setStrategy" method in the player object. A "strategy" and a "player" do not have to be interpreted literally, of course.

Your job in the *Chomp* project is to supply the missing code in the `CharMatrix` class, as described in the comments in `CharMatrix.java`. Then set up a project with `Chomp.java`, `CharMatrix.java`, `chomp.jar`, and `chomp.wav`, provided in `JM\Ch09\Chomp`, and test the program. Add the `EasySound` class (or the library `EasyClasses.jar`) from `JM\EasyClasses` to the project. The complete source code is available in `JM\Ch09\Chomp\src.zip`.

9.6 Iterations and the "For Each" Loop

Iterations are indispensable for handling arrays and lists for two reasons. First, if a list is large and we want to access every element (for example, to find the sum of all the elements), it is not practical to repeat the same statement over and over again in the source code:

```
int sum = 0;
sum += a[0];
sum += a[1];
...
...
sum += a[999];
```

A simple `for` loop saves 998 lines of code:

```
int sum = 0;
for (int i = 0; i < 1000; i++)
  sum += a[i];
```

Second, a programmer may not know the exact size of a list in advance. The actual number of elements may become known only when the program is running. For example, a list may be filled with data read from a file, and we may not know in advance how many items are stored in the file. The only way to deal with such a "variable-length" list is through iterations.

Traversal is a procedure in which every element of a list is "visited" and processed once. An array or list is usually traversed in order of the indices (or in reverse order of the indices). Usually elements are not added or removed during traversal.

An array or list traversal can be accomplished easily with a simple `for` loop. For example:

```
String[] names = new String[numGuests];
...
for (int i = 0; i < names.length; i++)
{
  String str = names[i];
  ...  // process str
}
```

Starting with version 5.0, Java offers a convenient "for each" loop for traversing arrays (as well as `java.util.ArrayList`s, discussed in Chapter 11, and other Java "collections"). The above code fragments can be rewritten with a "for each" loop as follows:

```
for (String str : names)
{
  ...  // process str
}
```

Another example:

```
double sum = 0;

for (double num : scores)
{
  sum += num;
}

double ave = sum / scores.length;
```

The "for each" syntax requires that the variable that refers to an element be declared inside the `for` statement, as in `String str` or `double num`.

Note that if you use a "for each" loop to traverse an array that holds elements of a primitive data type, you <u>cannot</u> change their values because the variable that refers to an element holds a copy of the element.

The "for each" loop does not give you access to the index of the "visited" element. For example, it will work fine in a `boolean` method like `contains` —

```
public boolean contains (String[] names, String target)
{
  for (String name : names)
    if (name.equals(target))
      return true;
  return false;
}
```

— but not in a method that returns the index of `target` in `names`. Use a regular `for` loop if you need access to the indices.

It is possible to use nested "for each" loops with two-dimensional arrays. Notice that in the outer loop the element is described as a 1D array. For example, if `m` is a two-dimensional array of `double`s, you can write:

```
for (double[] oneRow : m)
{
  for (double x : oneRow)
  {
    System.out.printf("%3.2f ", x);
  }
  System.out.println();
}
```

A frequent reason for traversing a list is to find the list's largest or smallest element. To find the maximum value we can initialize a variable representing the maximum to, say, the value of the first element of the array, then compare it with all the other elements and update its value each time we encounter a larger element. For example:

```
// Returns the value of the largest element in the array a
public static double findMax(double[] a)
{
  double aMax = a[0];

  for (int i = 1; i < a.length; i++)
  {
    if (a[i] > aMax)
      aMax = a[i];
  }

  return aMax;
}
```

Alternatively we can keep track of the position of the maximum:

```
// Returns the position of the largest element in the array a
public static int findMaxPos(double[] a)
{
  int iMax = 0;

  for (int i = 1; i < a.length; i++)
  {
    if (a[i] > a[iMax])
      iMax = i;
  }

  return iMax;
}
```

To find the minimum we can proceed in a similar way but update the current minimum value (or its position) each time we encounter a smaller element.

9.7 Inserting and Removing Elements

To insert an element into an array, you have to first make sure that the array has room to hold one more element. When you create your array, you have to allocate sufficient space for the maximum possible number of elements. In this case, the array's `length` will refer to the maximum capacity of the array. You need a separate variable to keep count of the elements actually stored in it. For example:

```
final int maxCount = 5000;      // Maximum number of words
String[] dictionary = new String[maxCount];
int count = 0;                  // Start with an empty dictionary
< ... etc. >
```

To add an element at the end of an array you need to check that there is still room and, if so, store the element in the first vacant slot and increment the count. For example:

```
String word;

<... other statements >

if (count < maxCount)
{
  dictionary[count] = word;
  count++;
}
```

If you want to keep an array sorted in ascending or descending order, it may be necessary to insert a new element in the middle of the array. To do that, you first need to shift a few elements toward the end of the array to create a vacant slot in the desired position. You have to start shifting from the last element — otherwise you may overwrite an element before you get it out of the way. Figure 9-4 illustrates the process.

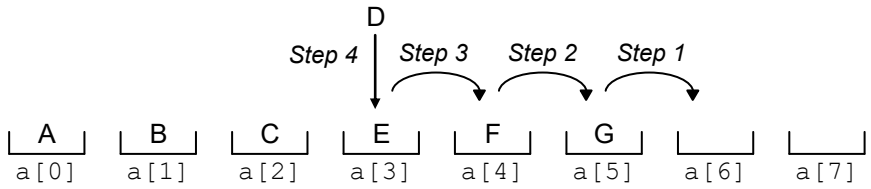


Figure 9-4. Inserting an element into the middle of an array

The following code serves as an example:

```
String word;
int insertPos;

<... other statements >

if (count < maxCount)  // count is the number of words already in
{                      //   dictionary
  for (int i = count; i > insertPos; i--)
  {
    dictionary[i] = dictionary[i-1];
  }
  dictionary[insertPos] = word;
  count++;
}
```

To remove an element you have to shift all the elements that follow it by one toward the beginning of the array. This time you have to start shifting at the position that follows the element that you are removing and proceed toward the end of the array (Figure 9-5).

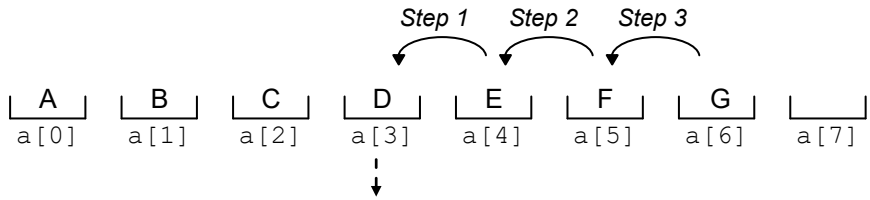


Figure 9-5. Removing an element from the middle of an array

For example:

```
/**
 * The array a holds n values. Removes the value at index i,
 * shifting the subsequent values to the left.
 * Returns the number of elements stored in the new array.
 * Precondition: 0 <= i < n
 */
public static int remove(int[] a, int n, int i)
{
  for (int k = i+1; k < n; k++)
    a[k-1] = a[k];
  return n-1;
}
```

If we are removing the last element (that is, `i == n-1`), then, of course, no shifting takes place.

If we need to repeatedly remove values from a large array, shifting the values repeatedly may become inefficient. It might be better to just mark all the elements that have been removed by setting their values to a special marker, for example, `null`, then, eventually, "pack" the array, removing the `null`s and moving the remaining values into their appropriate new locations in one sweep. For example:

```
/**
 * Removes all null values from words
 * Returns the number of the remaining words
 */
public static removeNulls(String[] words)
{
  int j = 0;

  for (int i = 0; i < words.length; i++)
  {
    String s = words[i];
    if (s != null)
      words[j++] = s;
  }

  return j;
}
```

9.8 *Case Study and Lab:* the Sieve of Eratosthenes

In Chapter 7, we presented a simple method that checks whether a given positive integer *n* is prime. This method works well for small *n*. But suppose we want to find the first 100,000 primes. We will need to examine about 1,500,000 numbers; the "brute-force" algorithm of Chapter 7 will be too slow for that.

In this case study we will analyze a more efficient method for finding primes, called the Sieve of Eratosthenes. Then you will modify the given code to further optimize its performance and space requirements.

Eratosthenes of Cyrene[eratosthenes] (pronounced air-a-TAWS-the-nEEss) was a Greek philosopher, geographer, music theorist, poet, and mathematician. He lived in the 3rd century BC. We discuss his Sieve algorithm here because of its sheer elegance and beauty and because it provides excellent practice for working with arrays and nested loops.

The idea of the Sieve is simple: instead of examining the primality of each number, let's make a list of all numbers and cross out all the composite numbers, and we will be left with primes. A composite number is a multiple of some smaller prime.

With a little patience, the Sieve can be performed with pencil and paper for a fairly large number of primes.

- Make a list or a paper tape with the numbers 1, 2, 3, 4, 5, ..., 120;
- Cross out 1;
- Find the next number *p* that has not been crossed out and circle it; then cross out its multiples 2*p*, 3*p*, 4*p*, ...
- Repeat the previous step until you reach the end of the paper tape.

Practice with this algorithm on paper first, to make sure you understand how it works. How many primes are there among the first 120 positive integers?

In the computer implementation, we will use a `boolean` array `isPrime` to represent numbers; each number is represented as an index into this array. Initially, all elements of the array starting from `isPrime[2]` are set to `true`, because they are all candidates for being prime (0 and 1 are not primes). Crossing out a number on the paper tape is represented in the computer algorithm by changing the value of the corresponding element of `isPrime` from `true` to `false`. At the end, `isPrime[p]` will be set to `true` if and only if `p` is prime. Here is the Java code for that (from `JM\Ch09\Sieve\Sieve.java`):

```
int n = ... // a large number, for example 1,000,000

boolean[] isPrime = new boolean[n]; // all set to false by default

isPrime[0] = isPrime[1] = false; // optional

for (int i = 2; i < n; i++) // {false, false, true, true,..., true}
  isPrime[i] = true;

for (int p = 2; p < n; p++)
{
  if (isPrime[p])  // if isPrime[p] is true
  {
    for (int i = 2*p; i < n; i += p)
      isPrime[i] = false;
  }
}
```

That's all.

The above code can be modified to make it faster and save space. Your task is to do that.

1. The first `for` loop that sets the values of `isPrime` to `true` wastes time. Change it to set `isPrime[p]` to `true` only for `p` = 2 and for all odd `p` > 1. This takes care of even numbers: we know that the only even prime is 2.
2. Since we have taken care of even numbers, start the second `for` loop at `p` = 3, and change the step from 1 to 2.
3. Given the changes made in the previous two items, can the step in the third (inner) `for` loop be increased? How?
4. Note that when we get to *p*, the numbers 2*p*, 3*p*, ..., (*p*−1)*p* will already have been crossed out on previous iterations. Make a change to the inner `for` loop to utilize that fact to make the code a little faster.
5. ◆ We know that no even number except 2 is prime. To save space used to store the numbers, change the Sieve code so that only odd numbers are represented in the `isPrime` array: `isPrime[k]` is set to `true` if and only if 2`k`+1 is prime.

9.9 Summary

Java allows programmers to declare *arrays* — blocks of consecutive memory locations under one name. An array represents a collection of related values of the same data type.

You can refer to any specific element of an array by placing the element's *index* (*subscript*) in brackets after the array name. An index can be any integer constant, variable, or expression. In Java the index of the first element of an array is 0 and the index of the last element is *length* – 1, where *length* is the array's length. An index must always be in the range from 0 to *length* – 1; otherwise the program throws an `ArrayIndexOutOfBoundsException`.

In Java, arrays are objects. If `a` is an array, `a.length` acts as a public field of the array `a` that holds the length of `a`. Arrays are passed to methods as references, so a method can change the contents of an array passed to it.

Programmers can also declare and use two-dimensional and multi-dimensional arrays. We can refer to an element in a 2D array by placing two indices, each in brackets, after the array's name. Think of the first index as a "row" and the second as a "column." Both indices start from 0.

In Java, a 2D array is a 1D array of 1D arrays, its rows. If `m` is a 2D array, `m.length` is the number of rows and `m[0].length` is the number of columns (assuming that all rows have the same number of columns).

Traversal is a procedure in which every element of a collection is "visited" and processed once. An array or a list is usually traversed in sequential order of its elements (or in reverse order). Elements are not added or removed during traversal.

The "for each" loop —

```
for (E elmt : list)
{
  ...  // process elmt
}
```

— provides a convenient way to traverse an array or a list from beginning to end.

Use nested conventional `for` loops to traverse a 2D array:

```
for (int r = 0; r < m.length; r++)
{
  for (int c = 0; c < m[0].length; c++)
  {
    ...
  }
}
```

Or use nested "for each" loops:

```
for (E[] row : m)
{
  for (E x : row)
  {
    ...
  }
}
```

Exercises

Sections 9.1-9.3

1. (a) Write a statement that declares an array of three integers initialized to 1, 2, and 4. ✓

(b) Write an expression that represents the sum of the three elements of the above array (regardless of their current values).

2. Mark true or false and explain:

(a) The following array has 101 elements:

```
int[] x = new int[100]; _____  ✓
```

(b) Java syntax allows programmers to use any expression of the `int` data type as an array subscript. _____

(c) While a program is running, it verifies that all array subscripts fall into the valid range. _____ ✓

(d) Any one-dimensional array object has a `length` method that returns the size of the array. _____ ✓

3. Write a method that takes an array of integers as a parameter and swaps the first element with the last one. ✓

4. An array of integers `scores` has at least two elements, and its elements are arranged in ascending order (that is, `scores[i] ≤ scores[i+1]`). Write a condition that tests whether all the elements in `scores` have the same values. ≤ Hint: you do not need iterations. ≥

5. Write a method `getRandomRps` that returns a character `'r'`, `'p'`, or `'s'`, chosen randomly with odds of 3 : 5 : 6, respectively. ≤ Hint: declare an array of `char`s and initialize it with values `'r'`, `'p'`, and `'s'`, with each value occurring a number of times proportional to its desired odds. Return a randomly chosen element of the array. ≥ ✓

6. What does the `mysteryCount` method count?

```
private int mysteryCount(int[] v)
{
  int count = 0;

  for (int i = 0; i < v.length; i++)
  {
    if (v[i] != 0) break;
    count++;
  }
  return count;
}
```

7. Write and test a method that returns the roots of a quadratic equation $ax^2 + bx + c = 0$, where $a \neq 0$. Recall that the roots of such an equation are given by the formulas $x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$ and $x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$. The value $d = b^2 - 4ac$ is called the *discriminant* of the equation. If $d > 0$, the equation has two distinct real roots. If $d = 0$, the roots "merge" into one. If $d < 0$, the equation has no real roots (it has two distinct complex roots). If $a = 0$, throw an `IllegalArgumentException`. ☜ Hint:

```
if (a == 0)
  throw new IllegalArgumentException("a = 0");
```

☞

Make your method return an array of two elements if the equation has two (not necessarily distinct) real roots or `null` if the equation has no real roots.

8. Generalize the method from the previous question so that it can handle the case when `a` is equal to zero. In this case, the method should return an array with one element, holding $-\frac{c}{b}$, when $b \neq 0$ and `null` when $b = 0$ and $c \neq 0$. If $a = b = c = 0$, your method should throw an `IllegalArgumentException`.

9. If you take any two positive integers *m* and *n* (*m* > *n*), then the numbers *a*, *b*, and *c*, where

$$a = m^2 - n^2; \quad b = 2mn; \quad c = m^2 + n^2$$

form a Pythagorean triple:

$$a^2 + b^2 = c^2$$

You can use algebra to prove that this is always true.

Write a method `makePythagoreanTriple` that takes two integer parameters, `m` and `n`, swaps them if necessary to make `m > n`, calculates the Pythagorean triple using the above expressions, swaps *a* and *b*, if necessary, to make *a* < *b*, places the resulting values *a*, *b*, and *c* into a new array of three elements, and returns that array. Test your method in a simple program.

10. Complete the following method:

```
// Returns an array filled with values
// 1, 2, ..., n-1, n, n-1, ..., 2, 1
// Precondition: n >= 1
public static int[] createWedge(int n)
{
  ...
}
```

11. Write a method that returns an array filled with the first *n* Fibonacci numbers. The first element should be $F_0 = 0$, the second element should be $F_1 = 1$; each subsequent element should be equal to the sum of the two previous ones. For example, `fibonacci(6)` should return an array with seven elements: 0, 1, 1, 2, 3, 5, 8.

12. In *SCRABBLE*,® different letters are assigned different numbers of points:

A – 1	E – 1	I – 1	M – 3	Q – 10	U – 1	X – 8
B – 3	F – 4	J – 8	N – 1	R – 1	V – 4	Y – 4
C – 3	G – 2	K – 5	O – 1	S – 1	W – 4	Z – 10
D – 2	H – 4	L – 1	P – 3	T – 1		

Write a method `computeScore(String word)` that returns the score for a word without using any `if` or `switch` statements. ⦇ Hint: find the position of a given letter in the alphabet string by calling `indexOf`; get the score for that letter from the array of point values, and add to the total. ⦈

Sections 9.4-9.6

13. A two-dimensional array `matrix` represents a square matrix with the number of rows and the number of columns both equal to `n`. Write a condition to test that an element `matrix[i][j]` lies on one of the diagonals of the matrix. ✓

14. Write a method that returns the value of the largest positive element in a 2D array, or 0 if all its elements are negative: ✓

```
// Returns the value of the largest positive element in
// the 2D array m, or 0, if all its elements are negative
private static double positiveMax(double[][] m)
```

15. Write a method

```
public void fillCheckerboard(Color[][] board)
```

that fills `board` with alternating black and white colors in a checkerboard pattern. For example:

Test your method by printing out the array, showing `'x'` for black and `'.'` for white.

16. Let us say that a matrix (a 2D array of numbers) `m1` "covers" a matrix `m2` (with the same dimensions) if `m1[i][j] > m2[i][j]` for at least half of all the elements in `m1`. Write the following method: ✓

```
// Returns true if m1 "covers" m2, false otherwise.
// Precondition: m1 and m2 have the same dimensions.
private static boolean covers(double[][] m1, double[][] m2)
{
  ...
}
```

17. Write a `boolean` method `isMagicSquare(int m[][])` that returns `true` if `m` is a magic square; otherwise it should return `false`. To be a magic square, `m` must be an *n*-by-*n* 2D array that holds all integers from 1 to n^2, and the sum of the numbers in each row, each column, and each of the two main diagonals must be the same. For example, this is a 3-by-3 magic square:

8	1	6
3	5	7
4	9	2

✓

18. (a) The `Location` class represents a (*row*, *col*) location in a grid (2D array). It has one constructor, `Location(row, col)`, and two accessors, `getRow()` and `getCol()`. Using the `Location` class from the *Chomp* lab (Section 9.5), write a `boolean` method `areAdjacent(Location loc1, Location loc2)` that returns `true` if `loc1` and `loc2` are neigbors, that is, are next to each other in the same row or in the same column.

(b) Write a `boolean` method `isSnake(int[][] m)`. `m` is a "snake" if it holds all numbers from 1 to *n* for some positive integer *n*, each of these numbers occurs in `m` only once, and consecutive numbers are in adjacent locations. The rest of the elements in `m` must be zeroes. For example:

```
0  0  0  0  0
0  1  2  0  0
0  0  3  0 11
0  5  4  9 10
0  6  7  8  0
```

19. Write and test a method that computes the average of the values stored in an array:

```
// Returns the average of the values stored in scores
public double average(int[] scores)
```

Use a "for-each" loop. ✓

20. (a) Write a method that returns the sum of all the elements in a one-dimensional array of integers. Use a "for-each" loop.

(b) Using the method you wrote in Part (a), write and test a method that takes a 2D array of integers `t` and returns a 1D array `sums`, such that `sums[k]` holds the sum of all the elements in the *k*-th row of `t`. Do not use any arithmetic operators.

✓

21. (a) Write and test a method that takes a `String` and returns an array of all its substrings (including the whole string but excluding the empty string). For example, `allSubstrings("cat")` should return an array with values `"c"`, `"a"`, `"t"`, `"ca"`, `"at"`, `"cat"` (in any order). ≤ Hint: If *n* is the length of the string, the number of substrings is $\frac{n(n+1)}{2}$. ≥

(b)■ Using the method from Part (a), write and test a method that takes an array of `String`s `words` and returns a two-dimensional array in which the *k*-th row is the array of all substrings of `words[k]`. ≤ Hint:

```
String[][] s = new String[n][];
```

declares a 2D array of `String`s with `n` "rows," where each row is undefined (set to `null`). ≥

22.■ Pascal's Triangle, named after the French mathematician Blaise Pascal (1623-1662), holds binomial coefficients. All the numbers on the left and right sides are equal to 1, and each of the other numbers is equal to the sum of the two numbers above it. It looks like this:

```
              1
            1   1
          1   2   1
        1   3   3   1
      1   4   6   4   1
    ......................
```

Write a method

```
public int[][] pascalTriangle(int n)
```

that returns a 2D array that holds the first n+1 rows of Pascal's Triangle. The method should return a "jagged" 2D array with n+1 rows. The k-th row ($0 \le k \le n$) should be a 1D array of length k+1 that holds the values from the k-th row of Pascal's Triangle (the top row is called row 0).

To test your method, generate and print out Pascal's Triangles for a few values of n (including $n = 0$). Try to make the output symmetrical, as above.

23. What values are stored in `list` after the following code is executed?

```
String[] list = {"One", "Two", "Three"};
for (String s : list)
  s += "*";
```

Sections 9.7-9.9

24. Find and fix the bug in the following code:

```
char[] hello = {' ', 'h', 'e', 'l', 'l', 'o'};

// Shift to the left and append '!':
int i = 0;

while (i < 6)
{
  hello[i-1] = hello[i];
  i++;
}

hello[5] = '!';
```

25. Write and test a method that determines whether a given number is a median for values stored in an array:

```
// Returns true if m is a median for values in the array
// sample, false otherwise. (Here we call m a median if
// the number of elements that are greater than m is the
// same as the number of elements that are less than m)
public boolean isMedian(double[] sample, double m)
```

Use one "for-each" loop to traverse `sample`. Traverse it only once.

26.■ (a) Write methods `void rotateLeft(int[] a)` and `void rotateRight(int[] a)` that rotate the array `a` by one position in the respective direction.

(b) Write a method

```
public static void rotate(int[] a, int d)
```

that rotates an array by a given number of positions *d*. A positive *d* rotates the array to the right; a negative *d*, to the left. For example, if `a` holds elements 1, 4, 9, 16, 25, 36, after `rotate(a, -2)` the values in `a` are 9, 16, 25, 36, 1, 4.

27.■ A non-negative "large integer" is represented as an array of *N* digits. The value of each digit is an integer from 0 to 9. The most significant digits are at the beginning of the array; zero values at the beginning indicate leading zeroes. Write the following method that calculates and returns the sum of two "large integers" *a* and *b*: ✓

```
private final int N = 30;

/**
 * Calculates the sum of two "large integers" a and b,
 * represented as arrays of digits, with the units digit
 * at the end of the array
 * @return array of N digits that represents the sum of a and b
 * Precondition: a.length == b.length == N;
 *               the sum fits into N digits
 */
private static int[] add(int[] a, int[] b)
{
  ...
}
```

28. (a)■ Write and test a class `Polynomial` that represents a polynomial $P(x) = a_0 + a_1x + \ldots + a_nx^n$. Use an array of `double`s of size $n + 1$ to hold the coefficients. Provide a constructor

```
public Polynomial(double[] a)
```

that sets the coefficients by copying them from a given array. Provide a method `degree` that returns the degree of the polynomial, n; provide a method `getValue(double x)` that calculates and returns this polynomial's value for a given x. Also include a reasonable `toString` method.

(b) The same polynomial can be rewritten as:

$$P(x) = a_0 + x(a_1 + x(a_2 + (\ldots + x(a_n))\ldots)$$

The latter representation is more efficient because it takes the same number of additions but fewer multiplications to calculate $P(x)$. Modify the `getValue` method from Part (a) using this formula.

(c)◆ Write a method

```
public Polynomial multiply(Polynomial other)
```

that multiplies this polynomial by `other` and returns their product.

29.■ Fill in the blanks in the following method that returns the average of the two largest elements of an array: ✓

```
// Finds the two largest elements in scores
// and returns their average.
// Precondition: scores.length >= 2
public static double averageTopTwo(int[] scores)
{
  int i, n = scores.length;
  int iMax1 = 0;        // index of the largest element
  int iMax2 = 1;        // index of the second largest element

  // If scores[iMax2] is bigger than scores[iMax1] --
  //   swap iMax1 and iMax2
  if (scores[iMax2] > scores[iMax1])
  {
    i = iMax1;

    _____________________________________

    _____________________________________
  }

  for (i = 2; i < n; i++)
  {
    if (scores[i] > scores[iMax1])
    {
      _____________________________________

      _____________________________________
    }
    else if ( ___________________________________ )
    {
      ____________________________
    }
  }
  return _____________________________;
}
```

30.◆ Write an OCR (Optical Character Recognition) program that uses template matching to distinguish between images of the letters "A" and "B". A grayscale image is represented as a two-dimensional array that holds light intensity values for individual *pixels* (picture elements) of the image. The intensity is an integer from 0 to 255. Values close to 255 represent the white background colors, and values close to 0 represent the "black ink" colors.

(a) The first task is to locate a bounding rectangle for the letter in an image. The dimensions of the rectangle correspond to the expected dimensions of the letter. The rectangle should have the most "ink" in it (the smallest total intensity in all the pixels in the rectangle).

Write a method

```
private int findVertPosition(int[][] image, int h)
```

that finds the horizontal band of height `h` in the image (`h` consecutive rows of pixels) that has more "ink" than any other band. The method should return the index of the top row in the band.

Then write a method

```
private int findHorzPosition(int[][] image, int row0,
                                            int h, int w)
```

that finds the rectangle of height `h` and width `w` within the band `row0 <= row < row0 + h` with most "ink" in it. The method should return the index of the leftmost column of the rectangle.

(b) A template for a letter is a 2D array with `h` rows and `w` columns that holds the values 0, 1, or –1. 1 indicates that an ideal picture of the letter should have "ink" at that location, and –1 indicates that there should be no ink. 0 can be either — not considered. The letters in the test images provided for you are approximately 12 pixels wide by 16 pixels high. Create two templates of 12 by 16 pixels, one for "A" and one for "B". The dimensions of a template also define the dimensions of the bounding rectangle used to locate the letter in an image.

Continued 

(c) Come up with a reasonable measure of fit between an image and a template superimposed on it with the upper left corner at (`row`, `col`) (or research online the mathematical definition of correlation between two sets of values). Write a method

```
private int match(int[][] image, int[][] template,
                                        int row, int col)
```

that calculates your measure of fit.

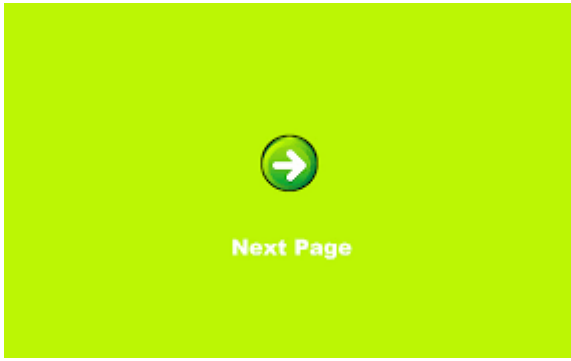
(d) Write a method

```
public char ocr(int[][] image)
```

The method returns the letter that best matches the letter in `image`. Use the methods from Part (a) to locate the letter, the templates from Part (b), and the measure of fit from Part (c).

(e) Test your OCR algorithm on several images of "A" and "B" available in `JM\Ch09\Exercises`. These images are 20 pixels wide by 24 pixels high, and the letters in the images are approximately 12 by 16 pixels.

You can examine an image file in a text editor and also display it in an image editing program such as *Photoshop* or *Paint Shop Pro*. Each image file has a header that consists of three lines, so you need to skip the first three lines when you read the file. The header is followed by 480 integers that represent the light intensities of all the pixels. The pixels data are stored in row-major order (first row, second row, and so on), but the line breaks in the file do not correspond to the rows in the image. Use `java.util.Scanner` or our `EasyReader` class to read the image files.


Next Page

```
Chapter chapter10 =
      new Chapter(10);
```

Implementing and Using Classes

10.1 Prologue

In the previous chapters we took a rather general view of classes and objects and did not explain in detail how to write classes and how to invoke constructors and call methods. The time has come to fill in the gaps. Our example will be a simple class called `Fraction`, which has the common features that we are interested in. A `Fraction` object represents something very familiar: a fraction with an integer numerator and denominator.

Occasionally, a class has its own `main` method and works as a stand-alone program or as a "main" class in a program. More often, though, a class exists to be used by other classes. For example, our class `Fraction` provides a set of tools for dealing with fractions in various applications related to arithmetic. Our `Fraction` class provides several constructors (for creating fractions) and methods (for adding and multiplying fractions and for converting a `Fraction` into a `double`). It also includes a `toString` method that returns a description of a `Fraction` as a string.

A class that uses a given class *X* is called a *client* of *X*.

The developer of a class looks at the class from a different perspective than its user (the author of client classes). The developer needs to know precisely how different features of the class are defined and implemented. A user is interested to know which features of the class are available and how to use them. In real life, the developer and the user can be the same programmer, just wearing different hats and switching between these points of view.

The developer's and the user's points of view must meet somewhere, of course: any usable feature of a class and the way to use it are defined by the developer, and the user must use it in just the right way. That is why we have invited Alex, the developer of `Fraction`, and Blair, who wrote a small client class `TestFractions`, to help us co-teach this chapter (see `Fraction.java` and `TestFractions.java` in the `JM\Ch10\Fraction` folder).

Blair: Hi!

Alex: That `Fraction` class — it wasn't a big deal. ⇦

Blair and Alex will help us present the following topics:

- Public and private features of a class
- The details of syntax for defining constructors and methods
- Syntax for invoking constructors and calling methods and the rules for passing parameters to them
- Returning values from methods using the `return` statement

Later in this chapter we will discuss two slightly more technical topics:

- Overloaded methods (giving different methods in a class the same name)
- Static (class) and non-static (instance) fields and methods.

By the end of this chapter, you should understand every feature of the `Fraction` class in Figure 10-1 and be able to use these features in client classes. You will then write your own class for the *Snack Bar* lab described in Section 10.9.

Blair: Wow, I didn't realize this `Fraction` class had so much stuff in it! ⇦

```
/**
 * Represents a fraction with an int numerator and int denominator
 * and provides methods for adding and multiplying fractions.
 *
 * Author: Alex
 */

public class Fraction
{

  // ******************  Instance variables  ******************

  private int num;
  private int denom;

  // *********************  Constructors  *********************

  public Fraction()       // no-args constructor
  {
    num = 0;
    denom = 1;
  }

  public Fraction(int n)
  {
    num = n;
    denom = 1;
  }
```

Figure 10-1 `Fraction.java` *continued* ⇩

```
public Fraction(int n, int d)
{
  if (d != 0)
  {
    num = n;
    denom = d;
    reduce();
  }
  else
  {
    throw new IllegalArgumentException(
         "Fraction construction error: denominator is 0");
  }
}

public Fraction(Fraction other)  // copy constructor
{
  num = other.num;
  denom = other.denom;
}

// ********************  Public methods  ********************

// Returns the sum of this fraction and other
public Fraction add(Fraction other)
{
  int newNum = num * other.denom + denom * other.num;
  int newDenom = denom * other.denom;
  return new Fraction(newNum, newDenom);
}

// Returns the sum of this fraction and m
public Fraction add(int m)
{
  return new Fraction(num + m * denom, denom);
}

// Returns the product of this fraction and other
public Fraction multiply(Fraction other)
{
  int newNum = num * other.num;
  int newDenom = denom * other.denom;
  return new Fraction(newNum, newDenom);
}

// Returns the product of this fraction and m
public Fraction multiply(int m)
{
  return new Fraction(num * m, denom);
}
```

Figure 10-1 `Fraction.java` *continued*

```
  // Returns the value of this fraction as a double
  public double getValue()
  {
    return (double)num / (double)denom;
  }

  // Returns a string representation of this fraction
  public String toString()
  {
    return num + "/" + denom;
  }

  // ******************  Private methods  ********************

  // Reduces this fraction by the gcf and makes denom > 0
  private void reduce()
  {
    if (num == 0)
    {
      denom = 1;
      return;
    }

    if (denom < 0)
    {
      num = -num;
      denom = -denom;
    }

    int q = gcf(Math.abs(num), denom);
    num /= q;
    denom /= q;
  }

  // Returns the greatest common factor of two positive integers
  private int gcf(int n, int d)
  {
    if (n <= 0 || d <= 0)
    {
      throw new IllegalArgumentException(
                  "gcf precondition failed: " + n + ", " + d);
    }

    while (d != 0)
    {
      int temp = d;
      d = n % d;
      n = temp;
    }
    return n;
  }
}
```

Figure 10-1. **JM\Ch10\Fraction\Fraction.java**

10.2 Public and Private Features of a Class

As you can see, every field, constructor, and method in the `Fraction` class is declared as either `public` or `private`. These keywords tell the compiler whether or not the code in other classes is allowed to access a field or call a constructor or method directly. These access rules are not complicated.

Private features of a class can be directly accessed only within the class's own code. Public features can be accessed in client classes (with an appropriate name-dot prefix).

Alex: As you see, I have declared the `num` and `denom` fields `private`:

```
private int num;
private int denom;
```

I made them private because I don't want anyone to change them from the outside, and I want to be able to change their names or data types without affecting client classes. I also made the `gcf` and `reduce` methods private.

Blair: Yes, I ran into a problem with that just the other day when I wrote

```
Fraction f = new Fraction(12, 20);
System.out.println("num = " + f.num + " denom = " + f.denom +
      " gcf = " + f.gcf(12, 20));
```

The compiler said:

```
num has private access in Fraction
denom has private access in Fraction
gcf(int, int) has private access in Fraction
```

You should have made at least the `gcf` method public. What if someone needs to find the GCF of two numbers?

Alex: I learned in school to provide as little information to clients as possible. This is called "information hiding." ⇦

The reason for making some of a class's features private is to control access to its fields by the class's *clients* and to "hide" its implementation details. That way the inner mechanics of the class (its private fields and private methods) can change without any changes to the rest of the program. This makes program maintenance easier.

In OOP, instance variables (fields) are almost always private. If necessary, the developer provides a special *accessor* method that returns the value of a particular private field. The developer may also define methods that update private fields. These are called *modifiers* or *mutators*.

But some constants are declared public. For example, `Color.RED`, `Math.PI`, `Integer.MAX_VALUE`.

Some "helper methods" of a class may be useful only to the objects of this class, but not to the class's clients. It makes sense to declare such methods private, too. Making all of a class's fields private and making the helper methods private ensures that the class can be completely described to outsiders by its constructors and public methods. These constructors and public methods describe everything the class and its objects can do for clients. This concept is known in OOP as *encapsulation*.

Alex: I didn't provide any modifier methods for the private fields `num` and `denom` because I didn't want anyone to fiddle with their values outside the class. No need for accessor methods either. As to the `gcf` method, you're right: I should have made it `public static`. I'll change that later.

Blair: Good idea, whatever that "static" is.

Alex: But my `reduce` method will remain private. ⇦

The concepts of public and private apply to the class as a whole, not to individual objects of the class.

For example, the compiler has no problem when `Fraction`'s `add` method refers directly to instance variables of `other Fraction`:

```
int newNum = num * other.denom + denom * other.num;
```

We mentioned earlier that a constructor can be declared private, too. You might be wondering: Why would one make a constructor private? This is indeed unusual. You might define a private constructor for one of two reasons. First, other constructors of the same class can call it. We will see the syntax for that in Section 10.3. Second, if a class has only one constructor and it is private, Java won't allow you to create objects of that class. This is used in such classes as `Math` or `System`, which are never instantiated.

10.3 Constructors

Constructors are procedures for creating objects of a class.

A constructor always has the same name as the class. Unlike methods, constructors do not return any values. They have no return type, not even `void`.

All constructors are defined inside the class definition. Their main task is to initialize all or some of the new object's fields. Fields that are not explicitly initialized are set to default values: zero for numbers, `false` for `boolean`s, `null` for objects.

A constructor may take parameters of specified types and use them for initializing the object that is being created. If a class has more than one constructor, then they must differ in the number or types of their parameters.

Alex: I have provided four constructors for my `Fraction` class:

```
public Fraction()        // no-args constructor
{
  num = 0;
  denom = 1;
}

public Fraction(int n)
{
  num = n;
  denom = 1;
}
```

```
public Fraction(int n, int d)
{
  if (d != 0)
  {
    num = n;
    denom = d;
    reduce();
  }
  else
  {
    throw new IllegalArgumentException(
         "Fraction construction error: denominator is 0");
  }
}

public Fraction(Fraction other)  // copy constructor
{
  num = other.num;
  denom = other.denom;
}
```

The first one, the *no-args* constructor, takes no parameters (arguments) and just creates a fraction 0/1. The second one takes one `int` parameter *n* and creates a fraction *n*/1. The third one takes two `int` parameters, the numerator and the denominator. The last one is a *copy constructor*: it takes another `Fraction` object as a parameter and creates a fraction equal to it.

Blair: I have tested all the constructors and they seem to work:

```
Fraction f1 = new Fraction();
Fraction f2 = new Fraction(7);
Fraction f3 = new Fraction(12, -20);
Fraction f4 = new Fraction(f3);

System.out.println("f1 = " + f1);
System.out.println("f2 = " + f2);
System.out.println("f3 = " + f3);
System.out.println("f4 = " + f4);
```

I got

```
f1 = 0/1
f2 = 7/1
f3 = -3/5
f4 = -3/5
```

⇦

The number, types, and order of parameters passed to the `new` operator must match the number, types, and order of parameters expected by one of the constructors. That constructor will be invoked.

You don't have to define any constructors for a class. If a class doesn't have any constructors, the compiler supplies a default *no-args constructor*. It allocates memory for the object and initializes its fields to default values. But if you define at least one constructor for your class, then the default no-args constructor is not supplied.

Normally constructors should prevent programs from creating invalid objects.

Alex: My constructor —

```
public Fraction(int n, int d)
{
  if (d != 0)
  {
    ...
  }
  else
  {
    throw new IllegalArgumentException(
        "Fraction construction error: denominator is 0");
  }
```

— throws an `IllegalArgumentException` if the parameter it receives for the denominator is 0. "Argument," as in math, and "parameter" is roughly the same thing.

Blair: What do I get if this "exception" happens?

Alex: Your program is aborted and the Java interpreter displays an error message, which shows the sequence of method and constructor calls, with their line numbers, that have led to the error. I am sure you've seen quite a few of those... ⇦

A constructor can call the object's other methods, even while the object is still "under construction."

Alex: One of my constructors, the one with two parameters, calls my `reduce` method. ⇦

A constructor can call another constructor of the same class by using the keyword `this`.

Alex: Sure, I could've written:

```
public Fraction()
{
  this(0, 1);
}

public Fraction(int n)
{
  this(n, 1);
}
```

And then those constructors would have called my two-parameter constructor. But I thought it'd be a little too fancy for these simple constructors. ⇦

If a constructor calls `this(...)`, the call must be the first statement in the constructor's body.

Unfortunately, Java allows you to use the class's name for a method name. This is a potential source of bugs that are hard to catch.

Alex: Yep. I got burnt on this one many times. I'd accidentally write something like

```
public MyWindow extends JFrame
{
  public void MyWindow()
  {
    ...
  }
  ...
}
```

instead of

```
public MyWindow()
```

No calls to `this` or `super` in my "constructor," right? So the compiler thinks I've defined a `void` method, `MyWindow`, rather than a constructor. It supplies a default no-args constructor for my class. I compile, run, and all I see is a blank window! ⇦

10.4 References to Objects

Objects are created by using the `new` operator. When you declare a variable of a class type — as in

```
private JButton go;
private CrapsTable table;
private RollingDie die;
```

— such a variable holds a *reference* to an object of the corresponding type. You can think of a reference simply as the object's address in memory. In the above declarations the three references are not explicitly initialized: they do not yet refer to valid objects. If the value of a variable is `null`, it indicates that currently the variable does not refer to a valid object.

It is crucial to initialize a reference before using it. The `new` operator is one way of doing this.

Another way to initialize a variable is to set it to a reference returned from a method. With the exception of literal strings and initialized arrays, objects are always created with `new`, either in one of your methods or in one of the library methods that you call.

If you try to call a method or access a public field through a `null` reference, your program "throws" a `NullPointerException`.

Sometimes you can create an anonymous temporary object "on the fly," without ever naming it. For example:

```
System.out.println(new Fraction(12, 20));
```

This is basically the same as:

```
Fraction temp = new Fraction(12, 20);
System.out.println(temp);
```

Alex: Like in my `add` method:

```
return new Fraction(newNum, newDenom);
```

⇦

In Java, several variables can hold references to the same object. The assignment operator, when applied to references, copies only the value of the reference (that is, the object's address), not the object itself. For example, after the statements

```
Fraction f1 = new Fraction(3, 7);
Fraction f2 = f1;
```

`f2` refers to exactly the same `Fraction` object as `f1` (Figure 10-2-a). This is not the same as

```
Fraction f1 = new Fraction(3, 7);
Fraction f2 = new Fraction(f1);
```

The latter creates a new `Fraction` object, a copy of `f1` (Figure 10-2-b).

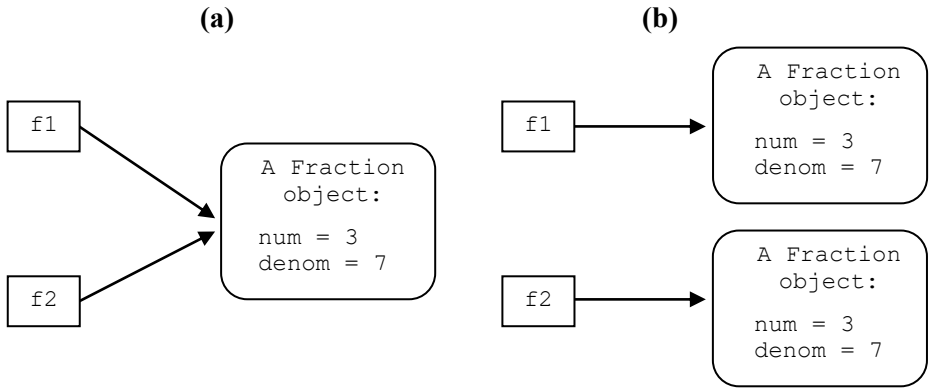


Figure 10-2. Copying references vs. copying objects

10.5 Defining Methods

A method is a segment of code that implements a certain task or calculation. Like a constructor, a method can take some parameters. For example, if the method's task is to calculate the GCF of two integers, it needs to know their values. A method either returns a value (for example, the GCF of two numbers) or completes a task (for example, reduces a fraction), or both.

From a pragmatic point of view, methods are self-contained fragments of code that can be called as often as needed from different places in your program. When a method is called, the caller places the parameters for the call in an agreed-upon place where the method can fetch them (for example, the system stack). The return address is also saved in a place accessible to the method (for example, the same system stack). When the method has finished, it returns control to the place in the program from which it was called. If the method returns a value, it places that value into an agreed-upon location where the caller can retrieve it (for example, a particular CPU register).

A method is always defined inside a class. When you define a method you need to give it a name, specify the types of its parameters, and assign them names so that you can refer to them in the method's code. The parameters can be of any primitive type or any type of objects or arrays. Some methods have no parameters at all, only empty parentheses.

You also specify the type of its return value:

```
public [or private] [static]
  returntype methodName(type1 paramName1, ..., typeN paramNameN)
```

returntype can be any primitive data type (such as `int`, `double`, `boolean`, `char`, etc.), an array, such as `int[]`, or an object type defined in Java or in your program (such as `String`, `Color`, `RollingDie`, `Fraction`, etc.). *returntype* can also be `void` (a reserved word) which means that this method performs some task but does not return any value.

Common Java style for naming methods is to choose a name that sounds like a verb and to write it starting with a lowercase letter. If the name consists of several words, subsequent words are capitalized. If a method returns the value of a field, its name usually starts with `get...`, and if a method sets the value of a field, its name usually starts with `set...`, as in `getWidth()` or `setText(...)`. Following these conventions helps other programmers understand your code.

Even if a method's parameters have the same type, you still need to explicitly state the type for each parameter. For example, the following won't work:

```
private int countToDegrees(int count, total)
                                      ^ Syntax error!
```

You need:

```
private int countToDegrees(int count, int total)
```

Alex: I have defined the `add` and `multiply` methods in my `Fraction` class and also a `toString` method.

Blair: As far as I am concerned, your methods are black boxes to me. I don't really want to know how they do what they do.

Alex: You need to know how to call them. For example, one of my `add` methods takes one parameter of the type `Fraction` and returns a new `Fraction`, the sum of `this` and `other`:

```
// Returns the sum of this fraction and other
public Fraction add(Fraction other)
{
  ...
}
```

Blair: So I can write:

```
Fraction f1 = new Fraction(1, 2);
Fraction f2 = new Fraction(1, 3);
Fraction sum = f1.add(f2);
System.out.println(sum);
```

and see 5/6 displayed? By the way, how does `println` know to print 5/6, as opposed to [5, 6] or 0.833333333?

Alex: That's because I have defined the `toString` method in my `Fraction` class:

```
// Returns a string representation of this fraction
public String toString()
{
  return num + "/" + denom;
}
```

See the output string that I defined for a fraction: numerator, slash, denominator? My `toString` method overrides `Object`'s `toString` method. Without it, `System.out.println(new Fraction(1, 3))` instead of `1/3` would print some garbage, something like `Fraction@1db9742`.

When you call

```
System.out.println(obj);
```

it calls

```
System.out.println(obj.toString());
```

This version of overloaded `println` takes a parameter of the type `Object`, but due to polymorphism, the correct `toString` method is called automatically for any type of object passed to `println`.

Blair: Whoa, hold it! Overloaded? Polymorphism? I think I'm the one who is getting overloaded here! ⇦

Overloaded methods will be explained shortly, in Section 10.10. The `Object` class, class hierarchies, and *polymorphism* are discussed in Chapter 12.

❖ ❖ ❖

After you specify the method's name, its return type, and its parameters with their types, you need to supply the code for your method, called its *body*. The body is placed within braces following the method's header. The code uses Java operators and control statements and can call other methods.

The names of parameters (such as `other` above) matter only inside the method's body. There they act pretty much like local variables. You cannot use these same names for local variables that you declare in that method.

10.6 Calling Methods and Accessing Fields

You have probably noticed from earlier code examples that the syntax for calling methods in Java is different in different situations. Sometimes a method name has a prefix that consists of an object name with a dot:

```
die1.roll();
result = game.processRoll(pts);
display.setText(str);
Fraction sum = f1.add(f2);
```

Sometimes the prefix is a class name with a dot:

```
y = Math.sqrt(x);
System.exit(1);
```

Sometimes there is no prefix at all:

```
degrees = countToDegrees(count1, total);
drawDots(g, x, y, getNumDots());
reduce();
```

To understand the difference we need to distinguish between *static* and *instance* methods. A static method is designated by the keyword `static` in its header. `static` is a Java reserved word. We will discuss static fields and methods in detail in Section 10.11. For now it is sufficient to understand how to call static methods.

A static method belongs to the class as a whole and doesn't deal with any instance variables. Such a method can be called by simply using the class's name and a dot as a prefix to the method's name, as in

```
x = Math.random();
ms = System.currentTimeMillis();
```

When we call a non-static (instance) method, we call it for a particular object:

```
obj.itsMethod(< parameters >);
```

In effect, the object for which the method is called becomes an implicit parameter passed to the call.

This parameter is not in the list of regular parameters — it is specified by the object's name and a dot as a prefix to the method's name, as in

```
int total = die1.getNumDots() + die2.getNumDots();
int result = game.processRoll(total);
Fraction sum = f1.add(f2);
```

This syntax signifies the special status of the object whose method is called. While the method is running, the reference to this object becomes available to the method and all other methods of the same class under the special name `this`. `this` is a Java reserved word. `this` acts like an instance variable of sorts, and its value is set automatically when an instance method is called or a constructor is invoked.

When a constructor or an instance method calls another instance method for the same object, strictly speaking the latter should be called with the prefix `this`. Java made this prefix optional, and it is usually omitted.

Alex: I could've written

```
this.reduce();
```

instead of

```
reduce();
```

and some people do that all the time, but to me it's just a waste of keystrokes. ⇦

❖ ❖ ❖

Similar syntax rules apply to accessing fields. A non-static field of another object (of the same class) is accessed by adding the object's name and a dot as a prefix.

Alex: For example, I wrote:

```
public Fraction multiply(Fraction other)
{
  int newNum = num * other.num;
  int newDenom = denom * other.denom;
  ...
}
```

⇦

An object's methods and constructors can refer to its own fields simply by their name. The prefix `this` is optional.

Alex: I could've written:

```
public Fraction multiply(Fraction other)
{
  int newNum = this.num * other.num;
  int newDenom = this.denom * other.denom;
  ...
}
```

More symmetrical, but, again, who wants to type extra stuff?

Sometimes I use the prefix `this` to distinguish the names of a class's fields from the names of the parameters in methods or constructors. For example, in my code for the constructor —

```
public Fraction(int n, int d)
{
  ...
  num = n;
  denom = d;
}
```

— I called the parameters `n` and `d` to avoid clashing with the field names `num` and `denom`. I could have written instead

```
public Fraction(int num, int denom)
{
  ...
  this.num = num;
  this.denom = denom;
}
```

Sometimes I find it easier to use `this`-dot than to think of good names for the parameters that are different from the names of the corresponding fields. Beware, though: if you misspell the parameter name in the method's header, the compiler won't complain. You can spend hours looking for a bug like this! ⇦

`this` is also used for passing "this" object (whose method is running) to another object's method or constructor as a parameter. For example,

```
clock = new Timer(delay, this);
```

invokes `Timer`'s constructor with two parameters. The first parameter is the time interval between consecutive firings of the timer (in milliseconds); the second is an `ActionListener` object. `this` means that the same object that is creating a new `Timer` will serve as its "action listener": it will capture `clock`'s events in its `actionPerformed` method.

10.7 Passing Parameters to Constructors and Methods

When a method is called or a constructor is invoked with a parameter of a primitive data type (`int`, `double`, etc.), the parameter can be any expression of the appropriate type: a literal or symbolic constant, a variable, or any expression that uses arithmetic operators, casts, and/or calls to other methods. For example,

```
game.processRoll(die1.getNumDots() + die2.getNumDots());
```

is equivalent to the sequence:

```
int dots1 = die1.getNumDots();
int dots2 = die2.getNumDots();
int total = dots1 + dots2;
game.processRoll(total);
```

The former form is preferable because it is shorter and just as readable. But when an expression gets too complicated, it is better to compute it in smaller steps.

❖ ❖ ❖

Parameters of primitive data types are always passed "by value."

If a variable of a primitive type is passed to a constructor or method as a parameter, its value is copied into some location (for example, the system stack) that is accessible to the method.

Suppose you write a method to increment its parameter:

```
public static void increment(int n)
{
  n++;
}
```

You test it somewhere in your program —

```
int n = 0;
MyMath.increment(n);
System.out.println("n = " + n);
```

— but nothing happens: the value of `n` remains `0`.

Blair: I know. I tried that when I first started learning Java. ⇦

The reason is that `n` in the method is not the same variable as `n` passed to `increment` as a parameter: it is a copy that has the same value (and happens to have the same name). You increment the copy, but the original remains unchanged.

Alex: In C or C++, you can specify whether you want a parameter passed "by value" or "by reference." ⇦

These languages have special syntax that allows programmers to pass variables to functions "by reference." When you pass "by reference," a reference to (address of) the original variable is passed to the function and so there is a way to write a method

similar to the one above that would work. But not in Java: primitive types are always passed to methods by value.

The situation is different when a parameter is an object of some class type (such as `RollingDie` or `Balloon`). When you pass an object to a method, a copy of the reference to the original object is passed. The method can reach and manipulate the data and methods of the original object through that reference. For example:

```
public void avoidCollision(RollingDie other)
{
  ...
  other.move();
}
```

Here `other` is a reference to another moving die that was passed to the method as a parameter. `other` holds the address of the original. Therefore `other.move()` will indeed move the other die.

Alex: In C++, you can pass a class-type variable by value: the whole object is copied and passed to a function. This is handy when the function needs to work with a temporary copy of the object and leave the original unchanged. ⇦

Not so in Java.

In Java, objects are always passed to methods and constructors as copies of references.

❖ ❖ ❖

Blair: So, I take it, at any given time several variables can refer to the same object. Suppose I pass a reference to my object to someone's constructor or method, and it changes my object behind my back. This is unsafe!

Alex: That's why I made sure my `Fraction` objects are *immutable*. I provided no public methods that could change the instance variables of a `Fraction`. Once created, a `Fraction` object cannot change. You can pass your fraction safely to anyone and be sure they won't change it. Even my `add` method does not change either `this` fraction or `other`. Instead it creates a new one, equal to their sum, and returns a reference to it. ⇦

10.8 `return` Statement

A method that is not `void` returns something to the caller. The returned value is specified in the `return` statement in the method's body:

```
return <expression>;
```

What type of value is returned depends on the return type of the method. *expression* must be of the method's specified return type or something that can be readily converted into it.

If the return type is a primitive type (`int`, `double`, `boolean`, etc.), then the method returns a value of that type. The return expression in a `boolean` method can include a constant (`true` or `false`), a `boolean` variable, or a `boolean` expression. For example, you can write:

```
public static boolean inRange(int x, int a, int b)
{
  return a <= x && x <= b;
}
```

This is more concise than

```
public static boolean inRange(int x, int a, int b)
{
  if (a <= x && x <= b)
    return true;
  else
    return false;
}
```

Alex: My `getValue` method returns the value of this fraction as a `double`:

```
// Returns the value of this fraction as a double
public double getValue()
{
  return (double)num / (double)denom;
}
```

Blair: I can use the returned value in assignments and expressions. For example:

```
Fraction f = new Fraction(2, 3);
double x = f.getValue();   // x gets the value 0.6666...7
```

or

```
System.out.println(f.getValue());
```

A method can also have a return type of some class. That means the method returns a reference to an object of that class.

Often such a method constructs and initializes a new object and sets its fields in a particular way and then returns a reference to the new object.

Alex: My `add` and `multiply` do that:

```
// Returns the sum of this fraction and other
public Fraction add(Fraction other)
{
  int newNum = num * other.denom + denom * other.num;
  int newDenom = denom * other.denom;
  return new Fraction(newNum, newDenom);
}
```

Come to think of it, I could have made it shorter:

```
public Fraction add(Fraction other)
{
  return new Fraction(num * other.denom + denom * other.num,
                                                denom * other.denom);
}
```

Blair: I can assign the returned value to a variable or pass it to a method:

```
Fraction sum = f1.add(f2);
```

or

```
System.out.println(f1.add(f2));
```

Alex: You can even call its method right away:

```
double x = f1.add(f2).getValue();
        // first calls add, then calls getValue for the sum
```

We have encountered objects returned from methods when we dealt with `String` objects. For example, the `String` class has a method `toUpperCase` that returns a new string with all the letters converted to upper case. It also has a `trim` method that returns a new string with all whitespace characters removed from both ends. You can write, for example:

```
JTextField input = new JTextField();
...
String s = input.getText().trim().toUpperCase();
```

A `return` statement tells the method what value to return to the caller and immediately quits the method's code and passes control back to the caller.

A method can have several `return` statements, but all of them except one must be placed inside a conditional statement (or inside a case in a switch):

```
... returntype myMethod(...)
{
  ...
  if (...)
    return <expression1>;
  ...
  return <expression2>;
}
```

Otherwise a method will have some unreachable code that is never executed because it is positioned after an unconditional `return`.

A `void` method can also have a `return`, but without any value, just as a command to quit the method.

For example:

```
... void myMethod(...)
{
  ...
  if (...)
    return;
  ...
}
```

10.9 *Case Study and Lab:* Snack Bar

When Java was first conceived by James Gosling[gosling] at Sun Microsystems in the early 1990s, it was supposed to be a language for programming embedded microprocessors — chips that control coffee makers and washers and microwave ovens. As it turned out, that was not Java's destiny. The language might have been completely forgotten, but the advent of the Internet and the World Wide Web gave it another life.

In a tribute to Java's early history, let's implement a program that simulates a set of vending machines in an automatic snack bar. As you will see in this case study, Java is a very convenient tool for such a project. Our vending machines will be quite simple: each machine can sell only one product. The user "deposits" quarters, dimes, or nickels into a machine, then presses the "Buy" button and "receives" a soda or a snack and change. When one of the machines is empty, you can call "service." After a concession operator enters a password ("jinx"), all the machines are refilled with "merchandise" and emptied of "cash." Figure 10-3 shows a snapshot of this program. You can play with it by clicking on the `snackbar.jar` file in `JM\Ch10\SnackBar`.

We begin, as usual, by identifying the classes involved in this project. One class, called `SnackBar`, is derived from `JFrame` and represents the program window. It constructs and displays a number of vending machines (in this case three) and handles service calls with a password login. The full source code for this class is in `JM\Ch10\SnackBar`.

The second class, called `VendingMachine`, represents a vending machine. `SnackBar`'s constructor creates three instances of this class — three machines. `VendingMachine` declares and creates the necessary display fields and buttons for a machine, handles events generated by these buttons, and displays the results of these events. This is where event-driven OOP is at its best: we can create three almost identical machines and let each of them process its own events automatically without any confusion. Several "customers" can "deposit coins" in random order into each of

the three machines, and the Java run-time environment will sort out all the events correctly.

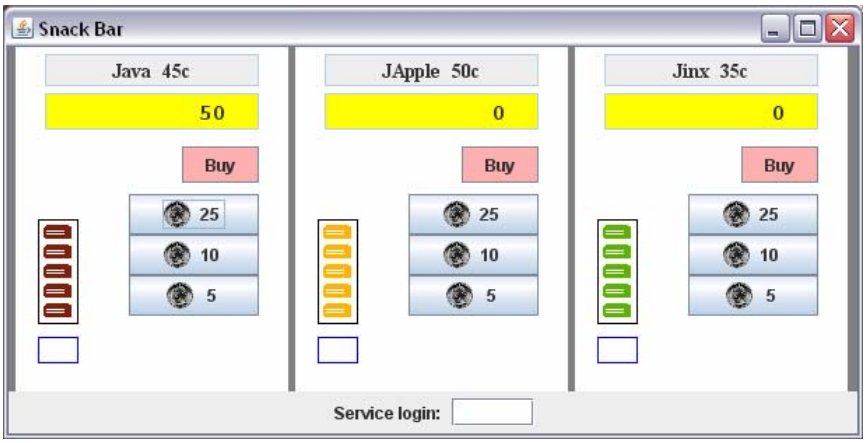


Figure 10-3. The *Snack Bar* program with three vending machines

Note that the `SnackBar` class actually knows very little about its vending machines: only how to construct them and that they need service or reloading once in a while (namely that `VendingMachine` has a constructor with three specific parameters and a `void` method `reload`). This is good OOP design: each class knows only what it really needs to know about other classes.

The complete `VendingMachine` class code is included in `JM\Ch10\SnackBar`. If you examine `VendingMachine`'s code, you will notice that a vending machine is really not the entire machine, but only its front panel, the GUI. One of the fields in this class is a `Vendor` object:

```
private Vendor vendor;
```

This object is created in `VendingMachine`'s constructor:

```
vendor = new Vendor(price, FULL_STOCK);
```

It is the machine's `vendor` that actually handles sales and keeps track of cash and stock (the remaining number of "snacks"). For example, when the user "deposits" 25 cents into a machine, the machine calls its `vendor`'s `addMoney` method:

```
vendor.addMoney(25);
```

If the user presses the "Buy" button, the machine calls its `vendor`'s `makeSale` and `getChange` methods:

```
trayFull = vendor.makeSale();
int change = vendor.getChange();
```

And when a machine needs to know whether there are any "snacks" left in the machine, it calls its vendor's `getStock` method:

```
if (vendor.getStock() > 0)
  ...
```

As in our *Craps* program in Chapter 6 and for the same reasons, we have once again separated the GUI part from the number crunching. We try to define the `Vendor` class in rather general terms so we can reuse it in different situations: it can represent not only a mechanism of a vending machine, but a cash register, an online store, any kind of vendor. In fact, a `Vendor` is so general it doesn't even know what it sells, only the price of the items!

Figure 10-4 shows the objects involved in the *Snack Bar* program: a `SnackBar`, three `VendingMachine`s, and three `Vendor`s. The `SnackBar` object represents the main program window. It "knows" about the features of a vending machine and creates three of them, but it is not aware of "vendors" behind them. Each vending machine creates and utilizes its own `vendor` object. Even though all vendors have the same name, there is no confusion because each vending machine knows its own `vendor`.

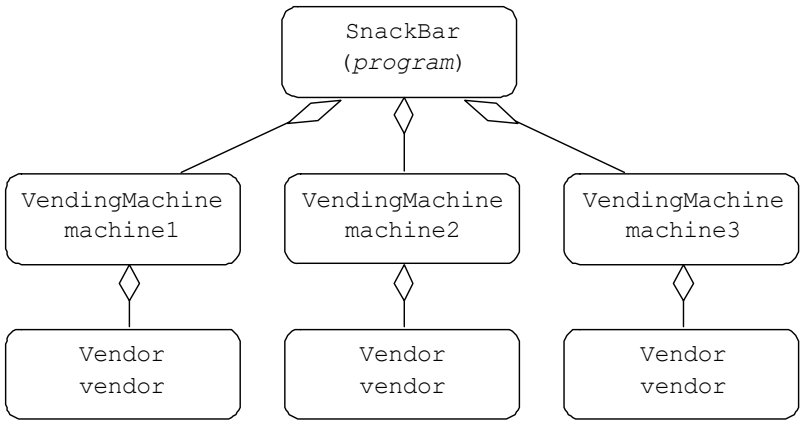


Figure 10-4. *Snack Bar* program's objects

As you have probably guessed, your job is to implement the `Vendor` class from the written specifications in `JM\Ch10\SnackBar\Vendor.java` (Figure 10-5). Each `Vendor` object can sell only one kind of item at one specified price. The vendor operates by collecting payment from a buyer in several steps. `Vendor` has four fields, representing the available stock (the number of remaining items for sale), the price, the currently deposited amount, and the change due to the customer from the last sale. `Vendor`'s constructor sets the price and the initial stock and zeroes out the deposit and change fields.

```
/**
 * This class implements a vendor that sells one kind
 * of items. A vendor carries out sales transactions.
 */
public class Vendor
{
  // Fields:
  ...

  /**
   * Constructs a Vendor
   * @param price the price of a single item in cents (int)
   * @param stock number of items to place in stock (int)
   */
  ... Vendor ...
  {
    ...
  }

  /**
   * Sets the quantity of items in stock.
   * @param qty number of items to place in stock (int)
   */
  ... setStock ...
  {
    ...
  }

  /**
   * Returns the number of items currently in stock.
   * @return number of items currently in stock (int)
   */
  ... getStock ...
  {
    ...
  }
```

Figure 10-5 `Vendor.java` *continued*

```
  /**
   * Adds a specified amount (in cents) to the
   * deposited amount.
   * @param number of cents to add to the deposit (int)
   */
  ... addMoney ...
  {
    ...
  }

  /**
   * Returns the currently deposited amount (in cents).
   * @return number of cents in the current deposit (int)
   */
  ... getDeposit ...
  {
    ...
  }

  /**
   * Implements a sale.  If there are items in stock and
   * the deposited amount is greater than or equal to
   * the single item price, then adjusts the stock and
   * calculates and sets change, sets deposit to 0 and
   * returns true; otherwise refunds the whole deposit
   * (moves it into change) and returns false.
   * @return true for a successful sale, false otherwise (boolean)
   */
  ... makeSale ...
  {
    ...
  }

  /**
   * Returns and zeroes out the amount of change (from
   * the last sale or refund).
   * @return number of cents in the current change (int)
   */
  ... getChange ...
  {
    ...
  }
}
```

Figure 10-5. JM\Ch10\SnackBar\Vendor.java

`Vendor`'s methods work as follows:

- The `addMoney` method adds a specified number of cents to the already deposited amount.
- The `makeSale` method is called when the buyer tries to complete the transaction. If the vendor is not out of stock and if the buyer has deposited enough money, then a sale takes place: the stock is decreased, the change is calculated, the deposit is reset to zero, and `makeSale` returns true. Otherwise the sale fails: the stock remains unchanged, the whole deposit is returned to the buyer (by transferring it to the change amount), and `makeSale` returns false.
- The `getDeposit` accessor simply returns the value of the current deposit (not the money itself!) to the caller. A vending machine calls this method when it needs to display the deposited amount on its display panel.
- The `getChange` method completes the transaction: it returns the change due to the buyer after a sale and at the same time (well, almost at the same time) resets the change field to 0. You have to be a little careful here: save the return value in a temporary local variable before setting change to 0.
- The `getStock` accessor returns the current stock.
- The `setStock` method sets the new stock quantity.

Set up a project with the `SnackBar.java`, `VendingMachine.java`, and `Vendor.java` files from `JM\Ch10\SnackBar`. Complete the `Vendor` class. Also add `coin.gif` to the folder where the data files go. Test the program thoroughly.

10.10 Overloaded Methods

It is not surprising that methods in different classes may have exactly the same name. Since a method is always called for a particular object (or a particular class), there is no confusion. For example:

```
int point = game.getPoint(); // calls getPoint of CrapsGame
Point p = e.getPoint();      // calls getPoint of MouseEvent

Fraction sum = f1.add(f2);
panel.add(button);
```

A more interesting fact is that several methods of the same class may also have the same name, as long as the number or the types of their parameters are different.

Methods within the same class that have the same name but different numbers or types of parameters are called *overloaded* methods.

A method can have any number of overloaded versions as long as their parameter lists are different. For example, in

```
public class SomeClass
{
  ...
  public int fun(int a)
  {
    ...
  }

  public int fun(double b)
  {
    ...
  }

  public double fun(int a, double b)
  {
    ...
  }

  ...
}
```

`SomeClass` has three different methods, all called `fun`. The code may be completely different in each of these methods. Overloading allows you to use the same method name for tasks that are similar. For example, the same name `print` is used for the methods of the `System.out` object that display a `char`, an `int`, a `double`, a `String`, and an `Object`.

The compiler knows which one of the overloaded methods to call based on the types of the parameters passed to it. In the above example, if you call `fun(1)`, the first overloaded method will be called, because `1` is an integer, but if you call `fun(1.5)`, the second overloaded method will be called because `1.5` is a `double`. If you call `fun(1, .99)`, the third version, `fun(int, double)`, will be called.

If there is no exact match between the parameter types in a call and the available overloaded versions of a method, the compiler will make a reasonable effort to convert one or more parameters into something acceptable to one of the versions. For example, if you call `fun(1, 2)` the compiler will call `fun(1, 2.0)`. If an appropriate method is not found and a reasonable conversion is not possible, the compiler reports an error. For example, if `s` is a `String`, and you call `fun(s)`, the compiler reports something like "method `fun(String)` not defined in class

`SomeClass`." (The compiler also reports an error if there is no exact match for parameter types and more than one overloaded method can handle them.)

Note that the names of the formal parameters in the method definition do not distinguish overloaded methods. Only the types of the parameters matter. For example,

```
public int fun(int a, double b)
{
  ...
}
```

and

```
public int fun(int x, double y)
{
  ...
}
```

cannot be defined in the same class.

The return type alone cannot distinguish two methods either. For example,

```
public int fun(int a, double b)
{
  ...
}
```

and

```
public double fun(int a, double b)
{
  ...
}
```

cannot be defined in the same class.

When you are designing a class, be careful not to have too many overloaded versions of a method, because they may get confusing and cause bugs.

Alex: I've defined two overloaded versions of the `add` method in my `Fraction` class: one takes another `Fraction` as a parameter, and the other takes an `int` as a parameter:

```
// Returns the sum of this fraction and other
public Fraction add(Fraction other)
{
  return new Fraction(num * other.denom + denom * other.num,
                                                denom * other.denom);
}


// Returns the sum of this fraction and m
public Fraction add(int m)
{
  return new Fraction(num + m * denom, denom);
}
```

Same for multiplication.

Blair: You mean I can write

```
Fraction f = new Fraction(1, 2);
Fraction f2 = new Fraction(1, 3);
Fraction sum1 = f.add(f2);
Fraction sum2 = f.add(3);
```

and the compiler will figure out which of your two `add` methods to call? That's very convenient. ⇦

All constructors in a class have the same name, so they are overloaded by definition.

10.11 Static Fields and Methods

In the previous chapters we said that an object's fields can be thought of as the "private memory" of the object. This is not the whole truth. A Java class may define two kinds of fields: non-static fields and *static* fields. Non-static fields are also called *instance variables*. They may have different values in different objects (instances of a class). Static fields are also called *class variables*. They are shared by all objects of the class. Static fields are declared with the keyword `static`.

When an object is created, a chunk of RAM is allocated to hold its instance variables. This is called *dynamic memory allocation*. But there is only one chunk of memory for the whole class that holds the values of static fields. Static fields are stored separately from all objects.

The word "static" may seem to imply that the values of static fields are constant. In fact it has nothing to do with constants. Static fields are called "static" because their memory is not dynamically allocated: memory for static fields is reserved even before any objects of the class have been created.

Why do we need static fields? Several reasons.

1. We might want to define a "universal" public constant. It makes sense to attribute it to the class as a whole, and not to waste memory space duplicating it in all objects. When we refer to such constants, we use the class name with a dot as a prefix, as opposed to a specific object's name. We have already seen many of these: `Color.BLUE`, `Math.PI`, and so on.
2. We might want to have all objects of the class share the same constants or settings. For example, in the `RollingDie` class, we have static fields that define the motion constants and the dimensions of the craps "table:"

```
private static final double slowdown = 0.97,
                            speedFactor = 0.04,
                            speedLimit = 2.0;

private static int tableLeft, tableRight, tableTop, tableBottom;
```

3. We may need to collect statistics or accumulate totals for all objects of the class that are in existence. Suppose, for example, we wanted to keep track of the total sales from all of the vending machines in the *Snack Bar* program. We need a common variable to which every vendor has access.

A class definition may also include *static methods*. Such methods do not access or manipulate any instance variables — they only work with static fields, or do not access any fields at all. Therefore, they are attributed to the class as a whole, not to individual instances (objects) of the class. Static methods are also called *class methods*. They are declared with the keyword `static`.

There are two primary reasons for defining static methods:

1. A static method may provide a "public service" that has nothing to do with any particular object. For example, `Math.max(int a, int b)` returns the largest of the integers `a` and `b`. Or `System`'s `exit` method that forces the program to quit.
2. A static method may work with static fields of the class. For example, it may be an accessor or a modifier for a static field.

Alex: While you were talking, I changed my `gcf` method in `Fraction` to make it `public` and `static`.

```
//  Returns the greatest common factor of two positive integers
public static int gcf(int n, int d)
```

Blair: That was my idea! But why is it static?

Alex: It does not need to deal with any fields at all — just calculates the GCF for two given numbers. If I didn't make it static, you'd have to create a `Fraction` object to call this method, even though the `gcf` method has nothing to do with any particular fraction (except that I call it from my `reduce` method). In fact, I could have placed it into another class or in a separate class. Too bad the folks at Oracle didn't put a `gcf` method into their `Math` class.*

Blair: You mean I can now write simply

```
int r = Fraction.gcf(a, b);
```

Alex: Yep. ⇦

`this` is undefined in static methods. A static method is not allowed to access or modify instance fields or to call instance methods (of the same class) without an object-dot prefix because such a call implies `this`-dot.

* In fact, the `BigInteger` class in the `java.math` library package (not to be confused with the `java.lang.Math` class) has a method `gcd` (greatest common divisor, the same as `gcf`).

Blair: I had this problem recently when I wrote

```
public class Test
{
  public void testConstructors()  { ... }
  public void testArithmetic()  { ... }

  public static void main(String[] args)
  {
    testConstructors();
    testArithmetic();
  }
}
```

I got

```
non-static method testConstructors() cannot be referenced from
a static context
non-static method testArithmetic() cannot be referenced from
a static context
```

I had to add `static` to the `testConstructors`'s and `testArithmetic`'s headers to make it work:

```
public static void testConstructors()  { ... }
public static void testArithmetic()  { ... }
```

Alex: That's because `main` is always static, so it can't call a non-static method without a reference to a particular object. You could've instead created one object of your `Test` class in `main` and then called that object's methods:

```
public class Test
{
  public void testConstructors()  { ... }
  public void testArithmetic()  { ... }

  public static void main(String[] args)
  {
    Test obj = new Test();
    obj.testConstructors();
    obj.testArithmetic();
  }
}
```

Instance variables are initialized in constructors and used in instance (non-static) methods. Theoretically, a constructor is allowed to set static fields, too, but it doesn't make much sense to do that because normally you don't want to affect all class objects while constructing one of them.

Instance methods can access and modify both static and non-static fields and call both static and non-static methods.

Alex: Sure. My instance method `reduce`, for example, calls my static method `gcf`.

```
int q = gcf(Math.abs(num), denom);
```

I still call `gcf` without any dot prefix because it is in the same class. I could have written

```
int q = Fraction.gcf(Math.abs(num), denom);
```

but I am not a pedant.

Blair: I see that the `Math` class has another static method, `abs`. That must be for getting an absolute value of an integer. Are all `Math` methods static?

Alex: That's right. ⇦

Some classes have only static fields and methods and no public constructors. For example, the `Math` class doesn't have any non-static methods or fields, and `Math` objects are never created because all such objects would be identical! All `Math` methods — `sqrt`, `pow`, `random`, `abs`, `max`, `min`, `round`, `sin`, `cos`, etc. — are static. The `Math` class also defines the public static constants `PI` (for π) and `E` (for e, the base of the natural logarithm). But the `Math` class exists in name only: it is not really "a class of objects."

Blair: You mean I can't write

```
Math xyz = new Math();
```

?

Alex: Ha-ha. Try it. ⇦

Another example of such a class is `System`. It has a few static fields (such as `System.out`) and static methods, such as `exit`, which quits the application, and `currentTimeMillis`, which returns the current time in milliseconds, but you cannot create an object of the `System` class.

The `Math` and `System` examples are a little extreme — more typical classes have some instance variables and may also have some static fields. If a class has a mix of instance variables and *class variables* (static fields), it is also likely to have a mix of non-static methods and some static methods.

Now let us see how a static variable can be used to accumulate the total amount of sales for all `VendingMachine` objects in our *Snack Bar* program.

10.12 *Case Study:* Snack Bar Concluded

Suppose we want to modify our *Snack Bar* program so that it reports the total "day sales" from all of the machines. Suppose a "day" is the time from the start of the program to the first "service call" or between two service calls. How can we implement a mechanism to keep track of the total sales? Clearly each machine (or, more precisely, its vendor object) needs access to this mechanism.

One approach could be based on the following scenario:

1 We define a new class and create a special object, "bookkeeper," that will keep track of the total sales.

2. When we create a vendor, we pass a reference to the bookkeeper object to the vendor's constructor, and the vendor saves it in its instance variable `bookkeeper`.

3. The vendor reports each sale to `bookkeeper`, and `bookkeeper` adds it to the total.

4. At the time of a "service call," the `SnackBar` object gets the total from `bookkeeper`, and `bookkeeper` resets the total to zero.

This is doable, and it is elegant in a way. This is also a flexible solution: it allows different sets of vendors to have different bookkeepers, if necessary. However, it is a little too much work for a simple additional feature. (Besides, we are here to practice static fields and methods). So let us take a more practical approach.

1. Add a private static field `double totalSales` to the `Vendor` class. This class variable will hold the total amount of all sales (in dollars) from all vendors (three here). Because `totalSales` is static, all `Vendor` objects share this field. Initially `totalSales` is set to zero by default. If you wish, for extra clarity add an explicit initialization to zero in the declaration of `totalSales`.

2. Add a public static method `getTotalSales` that returns the current value of `totalSales` and at the same time (well, almost at the same time) resets `totalSales` to zero.

3. Modify the `makeSale` method: if a sale is successful, add the amount of the sale to `totalSales`. (Don't forget to convert cents into dollars!)

This way, each vendor updates the same `totalSales` field, so `totalSales` accumulates the total amount of sales from all vendors.

Meanwhile, we have modified the `SnackBar` class: we have added a call to `Vendor`'s `getTotalSales` at the time of service:

```
double amt = Vendor.getTotalSales();
for (VendingMachine machine : machines)
  machine.reload();
... etc.
```

Since the `getTotalSales` method is static in `Vendor`, we call it for the `Vendor` class as a whole and do not need access to any particular `Vendor` object. Still, we have inadvertently made the `SnackBar` class dependent on `Vendor` because `SnackBar`'s code now mentions `Vendor`. Perhaps a cleaner solution would be to add a static method `getTotalSales` to the `VendingMachine` class and call that method from `SnackBar`. `VendingMachine`'s `getTotalSales` in turn would call `Vendor`'s `getTotalSales`. That way we would keep `SnackBar` completely isolated from `Vendor`, reducing *coupling*.

10.13 Summary

Public fields, constructors, and methods can be referred to directly in the code of any other class. *Private* fields and methods are directly accessible only in the code of the same class. If necessary, the programmer provides public accessor "get" methods that return the values of private fields and/or modifier "set" methods that update private fields. In OOP, all instance variables are usually declared private. The only exception is public static constants.

Constructors are short procedures for creating objects of a class. A constructor always has the same name as the class. Constructors may take parameters of specified types. If a class has several constructors, they are by definition overloaded and must take different numbers and/or types of parameters. Constructors do not return a value of any type and should not have a return type, not even `void`. A constructor initializes instance variables and may perform additional validation to make sure that the fields are set to reasonable values.

Constructors are invoked by using the `new` operator:

```
SomeClass someVar = new SomeClass(<parameters>);
```

The parameters passed to `new` must match the number, types, and order of parameters expected by one of the constructors. The `new` operator returns a reference to the newly created object.

Methods are always defined within a class. A method can take a number of parameters of specific types (or no parameters at all) and return a value of a specified type. The syntax for defining a method is:

```
public [or private] [static]
  returntype methodName(type1 paramName1, ..., typeN paramNameN)
  {
    < method body (code) >
  }
```

The parameters and the return value can be of any primitive type (`int`, `double`, `boolean`, etc.), an array type, or any type of objects defined in the program. The return type can be `void`, which indicates that a method does not return any value.

Programmers often provide a `public String toString()` method for their classes. `toString` returns a reasonable representation of an object as a `String`. `toString` is called automatically when the object is passed to `System.out.print` or `System.out.println` or concatenated with a string.

Methods and fields of an object are accessed using "dot" notation. Object `obj`'s `doSomething` method can be called as `obj.doSomething(`*`<parameters>`*`)`. However, an object can call its own methods without any prefix, using just `doSomethingElse(`*`<parameters>`*`)`. The same applies to fields.

All parameters of primitive data types are always passed to methods and constructors *by value*, which means a method or a constructor works with copies of the variables passed to it and cannot change the originals. Objects of class types, on the other hand, are always passed as copies of references. A method can change the original object through the supplied reference. *Immutable objects* have no modifier methods and, once created, can never change.

A method specifies its return value using the `return` statement. `return` tells the method what value to return to the caller. When a `return` statement is executed, the program immediately quits the method's code and returns control to the caller. A method can have several `return`s, but all of them must return a value or expression of the specified type, and all but one must be inside a conditional statement (or in a switch). All primitive types are returned by value, while objects (class types) are returned as references. A `void` method can also have a `return`, but without any value, just as a command to quit the method.

Several methods of the same class can have the same name as long as they differ in the numbers and/or types of their parameters. Such methods are called *overloaded* methods. Parameters passed to a method must match the number, types, and order of parameters in the method definition (or in one of the overloaded methods).

In addition to instance variables (non-static fields), a class definition may include *static fields* that are shared by all objects of the class. Likewise, a class may have static (class) methods that work for the class as a whole and do not touch any instance variables. Static fields are useful for sharing global settings among the objects of the class or for collecting statistics from all active objects of the class. Static methods cannot access non-static fields or call non-static methods. Static methods are called using the class's name as opposed to the individual object:

```
SomeClass.doSomething(<parameters>);
```

Exercises

Sections 10.1-10.8

1. Write a header line for a public method `replace` that takes two parameters, a `String` and a `char`, and returns another `String`. ✓

2. If a class `Complex` has two constructors, `Complex(double a)` and `Complex(double a, double b)`, which of the following statements are valid ways to construct a `Complex` object?

(a) `Complex z = new Complex();` _____ ✓
(b) `Complex z = new Complex(0);` _____ ✓
(c) `Complex z = new Complex(1, 2);` _____
(d) `Complex z = new Complex(0.0);` _____
(e) `Complex z = new Complex(1.0, 2);` _____
(f) `Complex z = new Complex(1.0, 2.0);` _____

3. (MC) Which of the following constructors of a class `Date` are in conflict?

I. `Date(int month, int day, int year)`
II. `Date(int julianDay)`
III. `Date(int day, String month, int year)`
IV. `Date(int day, int month, int year)`

A. I and II
B. II, III, and IV
C. I and IV
D. I, III, and IV
E. There is no conflict — all four can coexist

4. Find out by looking it up in the Java API specifications whether the `String` and `Color` classes have copy constructors. ✓

5. Java's class `Color` has a constructor that takes three integers as parameters: the red, green, and blue components of the color. A class `Balloon` has two fields: `double radius` and `Color color`. Write a constructor for the `Balloon` class that takes no parameters and sets the balloon's radius to 10 and its color to "sky blue" (with RGB values 135, 206, and 250).

6. Add `subtract` and `divide` methods to the `Fraction` class and test them. If the parameter for the `divide` method is a zero fraction, `divide` should throw an `IllegalArgumentException`.

7.■ The program *Temperature* (`JM\Ch10\Exercises\Temperature.java`) converts degrees Celsius to Fahrenheit and vice-versa using the `FCConverter` class. Examine how this class is used in the `actionPerformed` method in the `Temperature` class. Now write and test the `FCConverter` class. ≤ Hint: Recall that 0°C is 32°F; one degree Fahrenheit is 5/9 degree Celsius. For example, 68°F is $5/9 \cdot (68 - 32) = 20$°C ≥

8.■ Add an integer parameter `size` to `RollingDie`'s constructor (see Section 6.9) and set `dieSize` to `size` in the constructor. Change the `avoidCollision` method to compare the horizontal and vertical distances between the centers of `this` and `other` dice to the arithmetic mean of their sizes instead of `dieSize`. Change the `CrapsTable` class to roll two dice of different sizes in the program.

9. (a) A class `Point` has private fields `double x` and `double y`. Write a copy constructor for this class.

(b) A class `Disk` has private fields `Point center` and `double radius`. Write a copy constructor for this class. ✓

10. (a) Write a class `Rectangle` that represents a rectangle with integer width and height. Include a constructor that builds a rectangle with a given width and height and another constructor (with one parameter) that builds a rectangle that is actually a square of a given size. Make sure these constructors check that the width and height are positive. Add a constructor that takes no parameters and builds a square of size 1.

(b) Add a `boolean` method `isSquare` that returns `true` if and only if the rectangle is a square. Add a `void` method `quadratize` that converts this rectangle into a square with approximately the same area — the closest possible for a square with an integer side.

(c) Test all your constructors and methods in a simple console application. Define several rectangles, check which ones among them are squares, and print appropriate messages. "Quadratize" one of the rectangles, verify that it becomes a square, and print an appropriate message.

11. Examine the Java documentation and tell which of the following library classes define immutable objects:

`java.lang.Integer` _____
`java.awt.Color` _____
`java.awt.Point` _____
`java.awt.Rectangle` _____

12.■ A Java class can be declared `final`, which means that you cannot derive classes from it. For example, `Integer` and `String` are `final` classes. Why? ✓

13. The class `Time` represents the time of day in hours and minutes using the military time format (for example, 7:30 p.m. is 19:30, midnight is 00:00):

```
public class Time
{
  private int hours;
  private int mins;
  < ... etc. >
}
```

(a) Write a constructor `Time(int h, int m)` that checks that its parameters are valid and sets `hours` and `mins` appropriately. If the parameters are invalid, the constructor should throw an exception.

(b) Write a private method `toMins` that returns the time in minutes since the beginning of the day for this `Time` object.

(c) Write a public `boolean` method `lessThan(Time t)` that returns `true` if this time is earlier than `t` and `false` otherwise.

(d)■ Write a method `elapsedSince(Time t)` that returns the number of minutes elapsed from `t` to this time. Assume that `t` ≤ *this time* < `t+24h`. For example, if `t` is 22:45 and this time is 8:30, the method assumes that `t` is on the previous day and returns 585 (minutes). ≤ Hint: use `toMins`. ≥

(e) Supply a reasonable `toString` method.

(f) Test your `Time` class using the provided `TestTime` console application class (`JM\Ch10\Exercises\TestTime.java`).

Sections 10.9-10.13

14. Will the class below compile? If not, suggest a way to fix it. ✓

```
public class Pair
{
  private double first, second;

  public Pair(double a, double b)
  {
    first = a;
    second = b;
  }

  public void swap()
  {
    double temp = first; first = second; second = temp;
  }

  public Pair swap()
  {
    return new Pair(second, first);
  }
}
```

15. Add the fourth vending machine to the *SnackBar* program.

16. Write a class `Coins` with one constructor that takes a number of cents as a parameter. Supply four public methods, `getQuarters`, `getNickels`, `getDimes`, and `getPennies`, that return the number of corresponding coins that add up to the amount (in the optimal representation with the smallest possible number of coins). Make sure `Coins` objects are immutable (that is, none of the class's methods changes any fields). ≤ Hint: It is easier to do all the work in the constructor and save the four coin counts in fields, making the four methods simple accessor methods. ≥

(a) Test your class in a small console application that prompts the user for the change amount in cents and displays the number of quarters, dimes, nickels, and pennies.

(b)■ Integrate your class into the *Snack Bar* program, so that the program reports the amount of change received by the customer in specific coin denominations (for example, `"Change 65c = 2q + 1d + 1n"` or `"Change 20c = 2d"`). ≤ Hint: modify the statements in the `actionPerformed` method of the `VendingMachine` class after `vendor.getChange` is called. ≥

17. Figure out how to use the `Vendor` class from the *Snack Bar* program (without making any changes to it) for adding several integers. Write a console application that prompts the user to enter integers and adds the entered positive numbers separately and negative numbers separately. When the user enters a zero, the program displays both sums and exits. The + and – operators are not allowed in the program (except + for concatenating strings). Use `Scanner`'s `nextInt` method for entering numbers.

18.■ (a) Implement a class `Complex` (which represents a complex number $a + b \cdot i$) with two fields of the type `double` and two constructors described in Question 2. `Complex(a)` should make the same complex number as `Complex(a, 0.0)`.

(b) Add a method `abs` to your class that returns $\sqrt{a^2 + b^2}$ for a complex number constructed as `Complex(a, b)`.

(c) Recall that if $a + b \cdot i$ and $c + d \cdot i$ are two complex numbers, their sum is defined as $(a + c) + (b + d) \cdot i$. Write the `Complex` class's `add` method, which builds and returns the sum of this number and `other`:

```
public Complex add(Complex other)
{
  < ... missing statements >
}
```

(d) Find the rule for multiplying two complex numbers. ⦇ Hint: you can derive this rule yourself if you know that $i \cdot i = -1$. ⦈ Implement and test a method `multiply` for multiplying this complex number by another complex number. As with the `add` method, the `multiply` method should not change this object; it should build and return a new complex number, the product.

(e) Can you pass as a parameter a `double` rather than a `Complex` to the `add` and `multiply` methods? Add overloaded versions of `add` and `multiply` that would allow you to do that.

(f) Add a `toString` method to your `Complex` class that returns a string representation of the number in the form $a + bi$.

(g) Test your `abs`, `add`, `multiply`, and `toString` methods in a console application.

19. Find and fix a syntax error in the following program:

```
public class Puzzle
{
  private String message = "Hello, World";

  public void hello()
  {
    System.out.println(message);
  }

  public static void main(String[] args)
  {
    hello();
  }
}
```

✓

20. Rewrite the `FCConverter` class from Question 7, eliminating the fields and all constructors and providing two static methods:

```
public static double cToF(double degrees)
public static double fToC(double degrees)
```

Adjust the `Temperature` class accordingly and retest the program.

21. Suppose a class `Location` represents a location on a two-dimensional grid (as in *Chomp* in Chapter 9). It has the methods `getRow` and `getCol` that return the location's integer row and column, respectively. We can define a "distance" between two locations, `loc1` and `loc2`, as the length of the shortest path from `loc1` to `loc2`, in which consecutive locations are neighbors. (A location has eight neighbors, so diagonal steps are allowed. The "distance" from a location to itself is 0 and to any adjacent location is 1.) Devise a formula that computes this "distance." Write and test a class with one static `int` method `distance(Location loc1, Location loc2)` that returns the distance between `loc1` and `loc2`.

22.■ Add a static method `valueOf(double x)` to the `Fraction` class. This method should return a `Fraction` whose value is approximately equal to `x`. Define a public static symbolic constant `DFLT_DENOM` (for example, set to 10000) and use it as the denominator of the new fraction. Calculate its numerator as `x * DFLT_DENOM`, rounded to the nearest integer. Call `Math.round` to round the numerator. ≤ Hint: `Math.round` returns a `long`; you need to cast it into an `int`. ≥

23.◆ (a) Write a class `SoccerTeam` with fields that hold the number of wins, losses, and ties for this team in the current tournament.

Write a method

```
public void played(SoccerTeam other, int myScore,
                   int otherScore)
```

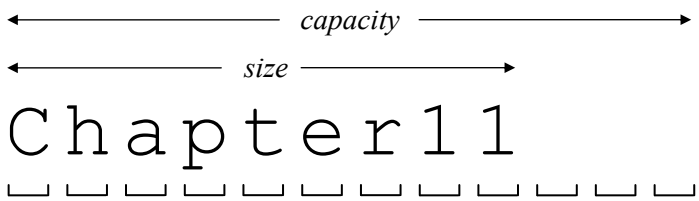
that compares the number of goals scored in a game by `this` team and `other` team and increments the appropriate fields (wins, losses, ties) for both teams.

(b) Write a method that returns this team's current number of points (each win is three points, each tie is one point). Write a `reset` method that zeroes out this team's wins, losses, and ties.

(c) Add fields to keep track of the total number of games played and the total number of goals scored by all teams in a tournament, combined. Modify the `played` method from Part (a) to update these fields. Add static accessor methods for these two fields and a static `startTournament` method to zero them out.

(d) Write a program that defines four teams, makes them "play" a few games with each other, and then reports each team's points as well as the total number of games played and the total number of goals scored by all teams in the tournament. The program should then repeat this for another tournament.

24.◆ Get rid of the static fields and methods in Part (c) of the previous question; instead, use an object of a separate class `TournamentOfficial` to keep track of the total number of games and the total number of goals scored in a tournament. Pass a reference to `official` to the `SoccerTeam` constructor and save it in an instance variable.



Chapter 11

java.util.ArrayList

11.1 Prologue

Java arrays have one limitation: once an array is created, its size cannot change. If an array is full and you want to add an element, you have to create a new array of a larger size, copy all the values from the old array into the new array, then reassign the old name to the new array. For example:

```
Object[] arr = new Object[someSize];
...

Object[] temp = new Object[2 * arr.length]; // double the size

for (int i = 0; i < arr.length; i++)
  temp[i] = arr[i];

arr = temp;
```

The old array is eventually recycled by the Java garbage collector.

If you know in advance the maximum number of values that will be stored in an array, you can declare an array of that size from the start. But then you have to keep track of the actual number of values stored in the array and make sure you do not refer to an element that is beyond the last value currently stored in the array. The library class `ArrayList` from the `java.util` package provides that functionality.

If your class uses `ArrayList`, add

```
import java.util.ArrayList;
```

at the top of your source file.

11.2 `ArrayList`'s Structure

As the name implies, an `ArrayList` allows you to keep a list of values in an array. The `ArrayList` keeps track of its *capacity* and *size* (Figure 11-1). The capacity is the length of the currently allocated array that can hold values. The size is the number of values currently stored in the list. When the size reaches `ArrayList`'s capacity and you need to add an element, the `ArrayList` automatically increases its capacity by executing code similar to the fragment shown above.

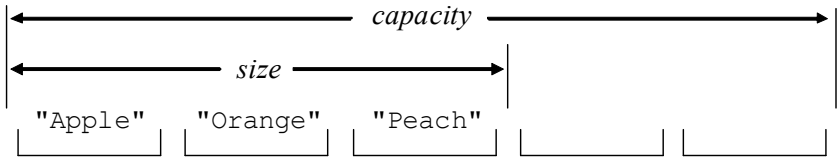
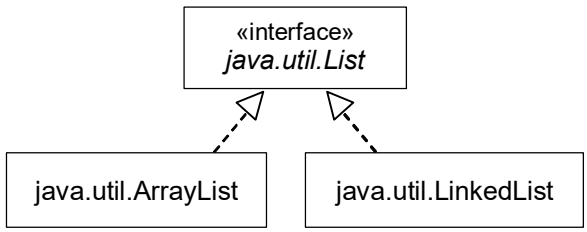


Figure 11-1. The capacity and size of an `ArrayList`

The `ArrayList` class provides the `get` and `set` methods, which access and set the value of the *i*-th element, respectively. These methods check that $0 \le i < size$ and will throw an `IndexOutOfBoundsException` if *i* is not in that range. In addition, an `ArrayList` has convenient methods to add and remove an element.

From a more abstract point of view, an `ArrayList` represents a "list" of objects. A "list" holds several values arranged in a sequence and numbered.

The `ArrayList` class is part of the *Java Collections Framework* and it implements the `java.util.List` *interface*. In Java, an interface defines the methods that a class that *implements* it must have (interfaces are discussed in Chapter 12). Another library class, `java.util.LinkedList`, also implements `java.util.List`:



`LinkedList` has the same methods as `ArrayList` (and a few additional methods), but it uses a different data structure to store the list values.

❖ ❖ ❖

An `ArrayList` holds objects of a specified type.

Starting with Java 5.0, when you declare an `ArrayList`, you should specify the type of its elements. The type is placed in angle brackets after `ArrayList`. For example:

```
private ArrayList<String> names;
```

or

```
private ArrayList<Color> colors;
```

Syntactically, the whole combination `ArrayList<sometype>` acts as one long class name, which is read "an ArrayList of sometypes." You need to use the whole thing whenever you use the `ArrayList` type: in declaring variables, method parameters, and return values, and when you create an `ArrayList`.*

For example:

```
// Returns an ArrayList with the elements from words
// that have 5 or more letters
public ArrayList<String> fiveOrLonger(ArrayList<String> words)
{
  ArrayList<String> words5 = new ArrayList<String>();

  for (String word : words)
    if (word.length() >= 5)
      words5.add(word);

  return words5;
}
```

Note that a "for each" loop works for `ArrayList`s and is a convenient way to traverse an `ArrayList`.

* Starting with Java 7, you can leave angle brackets empty on the right side of the declaration when the type is unambiguous. For example:

```
ArrayList<String> words5 = new ArrayList<>();
```

<> is known as the *diamond operator*.

An `ArrayList`, like other kinds of Java collections, can only hold objects. If you try to add a value of a primitive data type to a list, the compiler will convert that value into an object of a corresponding wrapper class (`Integer`, `Double`, etc.). For it to work, that wrapper type must match the declared type of the list elements.

For example,

```
ArrayList<Double> samples = new ArrayList<Double>();
samples.add(5.0);
```

works. This is called *autoboxing*. In fact

```
samples.add(new Double(5.0));
```

is now *deprecated* (no longer recommended and may be eventually removed).

But

```
samples.add(5);  // can't add an int to an ArrayList<Double>
```

won't work.

An `ArrayList` can also hold `null`s after calls to `add(null)` or `add(i, null)`. (We'll examine `add` and other `ArrayList` methods in the next section.)

11.3 `ArrayList`'s Constructors and Methods

`ArrayList`'s no-args constructor creates an empty list (`size() == 0`) of the default capacity (ten). Another constructor, `ArrayList(int capacity)`, creates an empty list with a given initial capacity. If you know in advance the maximum number of values that will be stored in your list, it is better to create an `ArrayList` of that capacity from the outset to avoid later reallocation and copying of the list. For example:

```
ArrayList<String> names = new ArrayList<String>(10000);
```

The `ArrayList` class implements over two dozen methods, but we will discuss only a subset of more commonly used methods.

```
int size()                        // Returns the number of elements
                                  //   currently stored in the list
boolean isEmpty()                 // Returns true if the list is empty;
                                  //   otherwise returns false
boolean add(E elmt)               // Appends elmt at the end of the list;
                                  //   returns true
void add(int i, E elmt)           // Inserts elmt into the i-th position;
                                  //   shifts the element currently at
                                  //   that position and the subsequent
                                  //   elements to the right (increments
                                  //   their indices by one)
E get(int i)                      // Returns the value of the i-th
                                  //   element
E set(int i, E elmt)              // Replaces the i-th element with elmt;
                                  //   returns the old value
E remove(int i)                   // Removes the i-th element from the
                                  //   list and returns its value;
                                  //   shifts the subsequent elements
                                  //   (if any) to the left (decrements
                                  //   their indices by 1)
boolean contains(Object obj)      // Returns true if this list contains
                                  //   an element equal to obj (the equals
                                  //   method is used for comparison)
int indexOf(Object obj)           // Returns the index of the first
                                  //   occurrence of obj in this list,
                                  //   or -1 if obj is not found (the
                                  //   equals method is used for comparison)
String toString()                 // Returns a string representation of this
                                  //   list as [elmt0, elmt1, ..., elmtN_1]
```

Figure 11-2. Commonly used `ArrayList<E>` methods

As we've noted, the elements in an `ArrayList` are numbered by their indices from `0` to *`list`*`.size() - 1`. In the `get(i)` and `set(i, elmt)` methods, the parameter `i` must be in the range from `0` to *`list`*`.size() - 1`; otherwise these methods throw an `IndexOutOfBoundsException`.

The `add(`*`E`*` elmt)` method appends `elmt` at the end of the list and increments the size of the list by one. The `add` method is overloaded: the version with two parameters, `add(int i, `*`E`*` elmt)`, inserts `elmt` into the list, so that `elmt` becomes the *i*-th element. This method shifts the old *i*-th element and all the subsequent elements to the right and increments their indices by one. It also increments the size of the list by one. This method checks that $0 \le i \le$ *`list`*`.size()` and throws `IndexOutOfBoundsException` if it isn't. If called with *i* = `list.size()`, then `add(int i, E elmt)` works the same as `add(E elmt)`.

The `remove(i)` method removes the *i*-th element from the list, shifts all the subsequent elements (if any) to the left by one, and decrements their indices. It also decrements the size of the list by one. `remove` returns the value of the removed element.

`ArrayList` implements the `get` and `set` methods very efficiently, because the elements are stored in an array, which gives direct access to the element with a given index. Inserting or removing an element at the beginning or somewhere in the middle of an `ArrayList` is less efficient, because it requires shifting the subsequent elements. Adding an element may occasionally require reallocation and copying of the array, which may be time-consuming for a long list.

The convenient `toString` method allows you to get a string representation of a list and to print the whole list in one statement. For example:

```
System.out.println(list);
```

The list will be displayed within square brackets with consecutive elements separated by a comma and a space.

The `indexOf(Object obj)` and `contains(Object obj)` methods are interesting. They need some way to compare `obj` to the elements of the list. These methods call `obj`'s `equals` method, something like this:

```
if (list.get(i).equals(obj))
  ...
```

Therefore, you should make sure a reasonable `equals` method is defined for your objects if you plan to place them into an `ArrayList`. Appropriate `equals` methods are defined, of course, for `Integer`, `Double`, `String`. We will explain how to define `equals` in your own class in Chapter 14.

11.4 *Lab:* Exploding Dots

The "Exploding Dots" math activity was popularized by James Tanton as part of the Global Math Project. The GMP was co-founded in 2015 by James and a small group of like-minded mathematicians; their idea was to design a math activity and hold a world-wide math event similar to *The Hour of Code* initiative of Code.org. This history is described in a short article, "*Global Joy: Uplifting Mathematics in Classrooms Across the Planet*," written by James in collaboration with the GMP team*. The activity itself, including student handouts and Teaching Guide, is described at globalmathproject.org/exploding-dots/.

Exploding Dots teaches the concept of place value in number systems. Start with an array of boxes. The first box initially contains *n* dots; the rest of the boxes are empty. Choose a positive integer $b \geq 2$, the base. On each step, take *b* dots from the current box, "explode" (remove) them, and put one dot into the next box. Repeat while the number of dots in the box remains greater than or equal to *b*, then move on to the next box.

It is no surprise, of course, that for $b = 10$ the number of dots in each box at the end matches the corresponding decimal digit of *n* (with the units digit in the first box). If we choose $b = 2$, the dots left in boxes will be binary digits of *n*. Figure 11-3 shows examples.

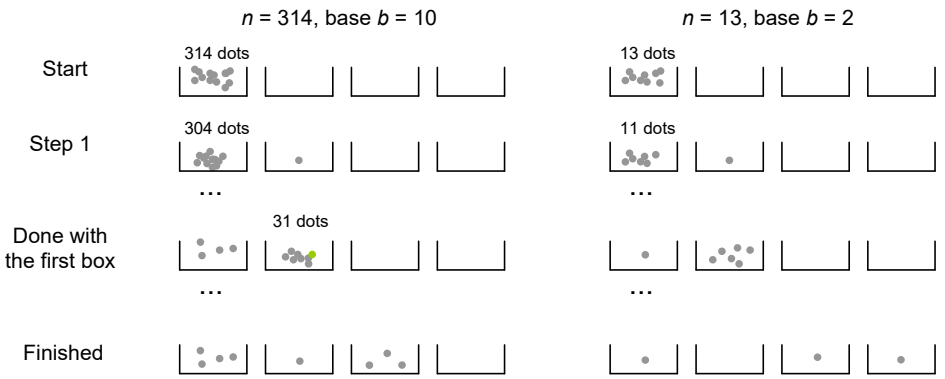


Figure 11-3. Exploding dots examples

* medium.com/@jamestanton/global-joy-uplifting-mathematics-in-classrooms-across-the-planet-ebcb6aa9fae7.pdf

In James Tanton's original Exploding Dots "machine," the dots start out in the rightmost box and the machine works from right to left. This is done so that the number of dots left in each boxmatches the digits of *n* in the usual order, with the units digit on the right. However, here we proceed from left to right, because we want to keep the numbers of dots in an `ArrayList`, and it is convenient to place the dots initially into the first element of the list. We usually envision an `ArrayList` as going from left to right, with the first element being the leftmost element. So in Figure 11-3 the number of dots left in each box matches the digits of *n* in reverse, with the units digit on the left.

In this lab you will emulate the Exploding Dots activity in a Java program using an `ArrayList<Integer>`.

Write and test a static method `explodeDots` that takes integers *n* and *b* as parameters and returns an `ArrayList<Integer>` that holds the numbers of dots left in each box after exploding *n* dots in base *b*. For example, `explodeDots(314, 10)` should return an `ArrayList` of three elements [4, 1, 3]; `explodeDots(13, 2)` should return an `ArrayList` [1, 0, 1, 1]. "Explode" the dots using subtraction, not the modulo division operator.

Write a complementary method `collectDots` that takes two parameters, the `ArrayList` returned by `explodeDots` and *b*, and returns *n*. Do not use any temporary lists or arrays, and make sure the given `ArrayList` remains unchanged. Instead "collect" the dots into one integer variable. Devise economical code for this method that uses *L* multiplications and *L* additions (with no `Math.pow` calls), where *L* is the size of the given list.

❖ ❖ ❖

Now let's explode dots in a fractional base, *p*/*q* (p > q). For every *p* dots in a box we remove them and add *q* dots in the next box. The resulting list will represent *n* in the base $\frac{p}{q}$ number system! For example, exploding 31 dots with $p = 3$ and $q = 2$ gives [1, 2, 0, 2, 1, 2], which corresponds to $1+2\cdot b+0\cdot b^2+2\cdot b^3+1\cdot b^4+2\cdot b^5=31$ for $b=\frac{3}{2}$.

Write and test an overloaded version of `explodeDots` with integer parameters *n*, *p*, and *q*, and a complementary method `collectDots(dotsList, p, q)` that takes the `ArrayList` returned by `explodeDots(n, p, q)` and returns *n*.

`collectDots` will work the same way as `collectDots(dotsList, b)`, only set *b* equal to a `double` equal to *p*/*q*. *n* will be a `double`, too, rounded and cast to an integer at the end. (Interestingly, for any integer *n* and any fractional base, the result of "collecting the dots," that is, converting the list of digits of *n* in base $\frac{p}{q}$ into the corresponding number, always results in an integer!)

In general, if coded correctly, `collectDots(list, x)` offers an economical way to compute the value of a polynomial $f(x) = a_0 + a_1 x + ... + a_n x^n$, represented by the list of its coefficients $[a_0, a_1, ... a_n]$, using only *n*+1 multiplications and *n*+1 additions, without computing powers of *x*. See Question 28 (b) in the exercises for Chapter 9.

11.5 *Lab:* Shuffler

Shuffler is a program that allows the user to enter a few lines of text or load a text file, shuffle the lines, and then restore the order by dragging and dropping the lines into the desired place. If not all the lines fit vertically inside the window, the text pane will scroll up or down when "pushed" at the upper or lower edge. There is also the "Delete" button; the user can click it to delete the currently selected line. Experiment with the program by running `shuffler.jar` in `JM\Ch11\Shuffler`. The program can be used to restore the order of lines in a Java source file, and for other puzzles and games.

Your task is to write the class `LineList` for the *Shuffler* program. The `LineList` class has one field, an `ArrayList` of `String`s. `LineList` has only one constructor; it is a no-args constructor that sets the list of lines to an empty list. `LineList` has the following public methods:

`int size()` — returns the number of lines in the list.
`String get(int k)` — returns the line with the index `k` from the list.
`void add(String line)` — appends `line` at the end of the list.
`String remove(int k)` — removes and returns the *k*-th line from the list.

`void move(int index, int newIndex)` — moves the line at `index` to a new position, at `newIndex`, shifting the rest of the lines as necessary.
`void shuffle()` — shuffles the list of lines, so that any line can end up in any position with equal probabilities.

The first four methods simply call the respective methods of the `ArrayList`. Test your `move` method very thoroughly. In particular, make sure your method works properly when `newIndex < index` and when `newIndex >= index`. No need to check that the indices fall into the allowed range in any of these methods: this is a *precondition*.

The `java.util.Collections` library class has a static method `shuffle` that shuffles a list. Do not use this method — write your own. Use the following shuffling algorithm:

1. Set n to the size of the list;
2. Randomly select an element among the first n and swap it with the n-th element;
3. Decrement n by one;
4. Repeat Steps 2-3 while $n \geq 2$.

In Step 2, call `Math.random()` and scale appropriately the returned value to generate a random index in the range from 0 to $n-1$.

Write a small test program to test your `LineList` class in isolation, and test thoroughly. Once it works as expected, set up a project combining your class with `Shuffler.java` and `shuffler.jar` from `JM\Ch11\Shuffler` and test the *Shuffler* program. The source code for other classes is included in `JM\Ch11\Shuffler\src.zip`.

11.6 `ArrayList`'s Pitfalls

You might have noticed in the lab in the previous section that working with an `ArrayList` may be treacherous.

> **You have to be careful with the `add` and `remove` methods: keep in mind that they change the size of the list and the indices of the subsequent elements.**

The following innocent-looking code, for example, intends to remove all occurrences of the word `"like"` from an `ArrayList<String>`:

```
ArrayList<String> words = new ArrayList<String>();
...

int n = words.size();

for (int i = 0; i < n; i++)
{
  if ("like".equals(words.get(i)))
    words.remove(i);
}
```

However, after the first `"like"` is found and removed, the size of the list `words` is decremented and becomes smaller than `n`. Once `i` goes past the new list size, the program will be aborted with an `IndexOutOfBoundsException`.

And that is not all. Even if we fix this bug by getting rid of `n` —

```
ArrayList<String> words = new ArrayList<String>();
...

for (int i = 0; i < words.size(); i++)
{
  if ("like".equals(words.get(i)))
    words.remove(i);
}
```

— another bug still remains. When an occurrence of `"like"` is removed, the subsequent words are shifted to the left. Then `i` is incremented in the `for` loop. As a result, the next word is skipped. If `"like"` occurs twice in a row, the second one will not be removed. The correct code should increment `i` only if the word is not removed. For example:

```
int i = 0;

while (i < words.size())
{
  if ("like".equals(words.get(i)))
    words.remove(i);
  else
    i++;
}
```


An `ArrayList` holds references to objects. It can hold duplicate values — not only equal objects (that is, `obj1.equals(obj2)`), but also several references to the same object (that is, `obj1 == obj2`).

It is important to understand that an object can change after it has been added to a list (unless that object is immutable) and that the same object can belong to several ArrayLists.

Consider, for example, the following two versions of the method makeGuestList that builds a list of people (objects of the class Person) from a given array of names. Let's assume that the class Person has the constructors used in the code and a setName method, which sets the person's name.

Version 1:

```
public ArrayList<Person> makeGuestList(String[] names)
{
  ArrayList<Person> list = new ArrayList<Person>();
  for (int i = 0; i < names.length; i++)
    list.add(new Person(names[i]));
  return list;
}
```

Version 2:

```
public ArrayList<Person> makeGuestList(String[] names)
{
  ArrayList<Person> list = new ArrayList<Person>();
  Person p = new Person();
  for (int i = 0; i < names.length; i++)
  {
    p.setName(names[i]);
    list.add(p);
  }
  return list;
}
```

After the statements

```
String[] names = {"Alice", "Bob", "Claire"};
List<Person> guests = makeGuestList(names);
System.out.println(guests);
```

are executed, Version 1 displays

```
[Alice, Bob, Claire]
```

as expected (Figure 11-4-a).

Version 2, however, displays

```
[Claire, Claire, Claire]
```

because the list contains three references to the same object (Figure 11-4-b). Adding this object to the list does not shield it from being modified by the subsequent `setName` calls.

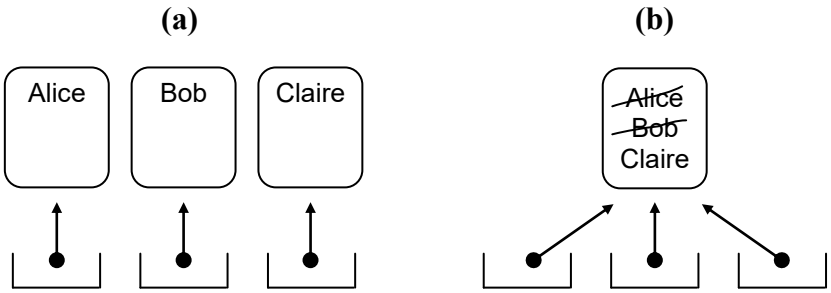


Figure 11-4. (a) A list with references to different objects; (b) A list with references to the same object

11.7 *Lab:* Creating an Index for a Document

In this lab you will write a program that reads a text file and generates an index for it. All the words that occur in the text should be listed in the index in upper case in alphabetical order. Each word should be followed by a list of all the line numbers for lines that contain that word. Figure 11-5 shows an example.

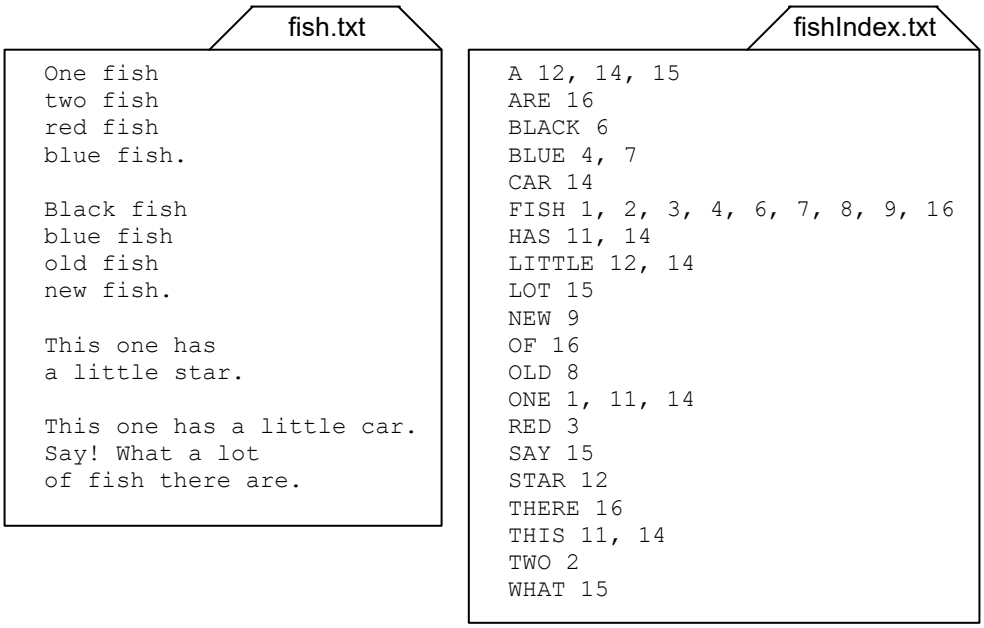


Figure 11-5. A sample text file and its index

The *Index Maker* program consists of three classes (Figure 11-6). It also uses `ArrayList` in two ways: `IndexEntry` has an `ArrayList<Integer>` field that holds the line numbers, and `DocumentIndex` extends `ArrayList<IndexEntry>`.

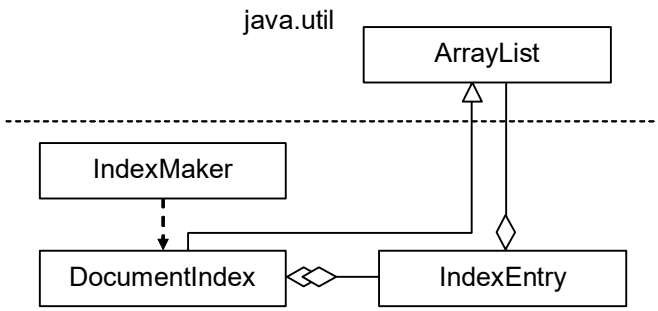


Figure 11-6. *Index Maker* classes

The `IndexMaker` class is the main class. We have provided this class for you in `JM\Ch11\IndexMaker`. Its `main` method prompts the user for the names of the input and output files (or obtains them from command-line arguments, if supplied), opens the input file, creates an output file, reads and processes all the lines from the input file, then saves the resulting document index in the output file.

Writing the `DocumentIndex` and `IndexEntry` classes is left to you (possibly in a team with another programmer). You don't have to deal with reading or writing files in this lab.

The `IndexEntry` class

An `IndexEntry` object represents one index entry. It has two fields:

```
private String word;
private ArrayList<Integer> numsList;
```

The numbers in `numsList` represent the line numbers where `word` occurs in the input file. (Note that the `IndexEntry` class is quite general and reusable: the numbers can represent line numbers, page numbers, etc., depending on the application.)

Provide a constructor for this class that takes a given word (a `String`), converts it into the upper case (by calling `toUpperCase`), and saves it in `word`. The constructor should also initialize `numsList` to an empty `ArrayList<Integer>`.

This class should have the following three methods:

1. `void add(int num)` — appends `num` to `numsList`, but only if it is not already in that list. (The `ArrayList`'s `contains` method expects an object as a parameter, and the `add` method expects an `Integer`, but `num` can be converted into an `Integer` automatically, due to autoboxing.)
2. `String getWord()` — this is an accessor method; it returns `word`.
3. `String toString()` — returns a string representation of this `IndexEntry` in the format used in each line of the output file (Figure 11-5).

The `DocumentIndex` class

A `DocumentIndex` object represents the entire index for a document: the list of all its index entries. The index entries should always be arranged in alphabetical order, as shown in Figure 11-5.

Make the `DocumentIndex` class extend `ArrayList<IndexEntry>`. Provide two constructors: one that creates a list with the default capacity, the other that creates a list with a given capacity. (These constructors simply call the respective constructors of the superclass, `ArrayList`.)

`DocumentIndex` should have the following two public methods:

1. `void addWord(String word, int num)` — adds `num` to the `IndexEntry` for `word` by calling that `IndexEntry`'s `add(num)` method. If `word` is not yet in this `DocumentIndex`, the method first creates a new `IndexEntry` for `word` and inserts it into this list in alphabetical order (ignoring the upper and lower case).
2. `void addAllWords(String str, int num)` — extracts all the words from `str` (skipping punctuation and whitespace) and for each word calls `addWord(word, num)`.

   You could code the word extractor yourself, of course, but it is much better to use the `String` class's `split` method. Look it up in the Java API. Use the one that takes one parameter, `regex`, that is, a *regular expression*[regex]. Regular expressions are not specific to Java: they are used in many languages and text parsers. `regex` describes the match pattern for all possible word separators. Use `"\\W+"` here. `\W` (with an uppercase 'W') stands for any "non-word" character, that is, any character that is not a digit or a letter. + means "occurs at least once." (Regular expressions use backslash as the escape character; hence the double backslash in the literal string.)

   `split` returns an array of `String`s. Use a "for each" loop to call `addWord` for each word in that array. Note, however, that `split` may put an empty string into the resulting array — when `str` starts with a separator or when `str` is empty. This is an unfortunate decision (or a bug). Make sure you skip empty strings and do not call `addWord` for them.

We recommend that you also define a private helper method

```
private int foundOrInserted(String word)
```

and call it from addWord. This method should traverse this DocumentIndex and compare word (case-blind) to the words in the IndexEntry objects in this list, looking for the position where word fits in alphabetically. If an IndexEntry with word is not already in that position, the method creates and inserts a new IndexEntry for word at that position. The method returns the position (we'd like to say "the index" but we have too many indices going already!) of the either found or inserted IndexEntry.

Test your program thoroughly on different text data files, including an empty file, a file with blank lines, a file with lines that have leading spaces or punctuation, a file with multiple occurrences of a word on the same line, and a file with the same word on different lines.

11.8 Summary

The java.util.ArrayList class helps implement "dynamic arrays" — arrays that can grow as needed. An ArrayList keeps track of the list's capacity (the length of the allocated array) and its size (the number of elements currently in the list). The no-args constructor creates an empty list of some small default capacity; another constructor creates an empty list of a specified capacity. When an ArrayList is full and we add a value, the ArrayList increases its capacity automatically by allocating a larger array and copying all the elements into it.

Starting with Java 5.0, ArrayList holds elements of a specified type. ArrayList doesn't work with values of a primitive data type — they are converted to objects of the corresponding wrapper class.

The most commonly used ArrayList methods are shown in Figure 11-2. The add(obj) method appends an element at the end of the list. The get(i), set(i, obj), add(i, obj), and remove(i) methods check that i is from 0 to size() - 1 (from 0 to size() in case of add) and throw an IndexOutOfBoundsException if an index is out of the valid range. The add and remove methods adjust the indices of the subsequent elements and the size of the list. The contains method checks whether a given value is in the list, and the indexOf method returns the index of a given value (or -1 if not found).

ArrayList's get(i) and set(i, obj) methods are efficient because an array provides random access to its elements. The add(i, obj) and remove(i) methods are on average less efficient, because they require shifting of the elements that follow the *i*-th element. The add methods need to allocate more memory and copy the whole list when the list's capacity is exceeded.

An `ArrayList` holds references to objects. An object can be changed after it is added to the list (unless the object is immutable). A list is allowed to hold `null` references and multiple references to the same object.

The "for each" loop —

```
for (E elmt : list)
{
  ...  // process elmt
}
```

— works for an `ArrayList` as well as for a one-dimensional array.

Exercises

1. Mark true or false and explain:

(a) An `ArrayList` can contain multiple references to the same object. _____ ✓
(b) The same object may belong to two different `ArrayList`s. _____
(c) `ArrayList`'s `remove` method destroys the object after it has been removed from the list. _____ ✓
(d) `ArrayList`'s `add` method makes a copy of the object and adds it to the list. _____
(e) Two variables can refer to the same `ArrayList`. _____ ✓

2. The `ArrayList` class has a method `trimToSize()`, which trims the capacity of the list to its current size. True or false? Calling this method

(a) will save space in the program.
(b) will make `add(x)` calls more efficient
(c) will make `add(0, x)` calls more efficient
(d) will make `get(i)` and `set(i, x)` calls more efficient

3. Can you give a couple of reasons why you might use a simple Java array as opposed to an `ArrayList`?

4. What is the output from the following code?

```
ArrayList<Integer> lst = new ArrayList<Integer>();
lst.add(0);
lst.add(1);
lst.add(2);
lst.add(0, 0);
lst.add(1, 1);
lst.add(2, 2);
System.out.println(lst);  ✓
```

5. Write and test a method that takes an `ArrayList<String>` and returns a new `ArrayList<String>` in which the elements are stored in reverse order. The original list should remain unchanged. (Use `ArrayList`'s methods but no other library methods.) ✓

6. Write and test a method that removes the smallest value from an `ArrayList<Integer>`. ≤ Hint: `Integer` has a method `compareTo(Integer other)` that returns a positive integer if `this` is greater than `other`, a negative integer if `this` is less than `other`, and 0 if `this` is equal to `other`. ≥

7.■ Write and test a method

```
public void filter(ArrayList<Object> list1,
                                ArrayList<Object> list2)
```

that removes from `list1` all objects that are also in `list2`. Your method should compare the objects using the == operator, not `equals`. ≤ Hint: the `contains` and `indexOf` methods cannot be used. ≥ ✓

8. Can an `ArrayList<Object>` be its own element? Test this hypothesis.

9.■ Derive a subclass from `ArrayList<Object>`. Pretend that `ArrayList`'s `toString` method does not exist and write your own for your class (overriding the `ArrayList`'s `toString`). Create a list of objects of different types (for example, `String`, `Integer`, `Fraction`) and test your `toString` method.

10. Fill in the blanks in the method `removeConsecutiveDuplicates`, which removes consecutive duplicate values from an `ArrayList` of strings. For example, if `letters` contains `["A", "A", "A", "B", "C", "C", "A", "A"]`, after a call to `removeConsecutiveDuplicates(letters)` `letters` should contain `["A", "B", "C", "A"]`.

```
public void removeConsecutiveDuplicates(ArrayList<String> lst)
{
  for (_____________________________________________________)

    if (_____________________________________________________)

      lst.remove(_________________);
}
```

≤ Hint: it is easier to traverse the list backwards. ≥

11. The Java library `java.math` package (not to be confused with the `Math` class) includes the class `BigInteger`. A `BigInteger` represents an integer with as many digits as you want. A `BigInteger` has a constructor that takes a `String` of digits as a parameter. It also has a method `add(BigInteger other)`, which adds `other` to `this` and returns a new `BigInteger`, their sum. Write and test a method that generates a list (an `ArrayList<BigInteger>`) of the first *n* Fibonacci numbers. The Fibonacci sequence is defined as follows:
$F_0 = 0;\ F_1 = 1;\ F_k = F_{k-1} + F_{k-2}$ for $k \geq 2$. How many digits does F_{100} have?
≤ Hint: `BigInteger` has a `toString` method, too. ≥

12. Rewrite the `LineList` class from the lab in Section 11.5 so that instead of using an `ArrayList<String>` field it extends `ArrayList<String>`. Remove all of the redundant methods.

13.■ Write and test a method that takes an `ArrayList<String> words`, which contains strings of letters, throws them into 26 "buckets," according to the first letter, and returns the `ArrayList` of buckets. Each bucket should be represented by an `ArrayList<String>`. The first bucket should contain all the strings from `words` that start with an 'a', in the same order as they appear in `words`; the second bucket should contain all the strings that start with a 'b'; and so on. Your method should traverse the list `words` only once and leave it unchanged.

14.■ A generalized Fibonacci sequence can start with any two numbers, F_0 and F_1, not necessarily 0 and 1 as shown in Question 11 above. We can also choose a positive integer *p* and reduce each Fibonacci number modulo *p*. If $p = 10$, we only keep the units digit of F_k.

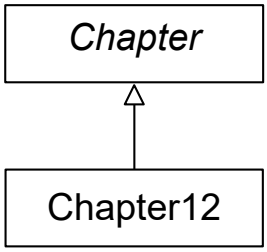
For example, if the sequence starts with $F_0 = 1;\ F_1 = 8$ and $p = 10$, the Fibonacci sequence modulo *p* becomes {1, 8, 9, 7, 6, 3, 9, 2, 1, 3, 4, 7, 1, 8, ...}. As we can see, the 13th and 14th terms are the same as the first and the second terms, so from that point on the sequence repeats itself. If you "glue" the 13th term to the first and the 14th term to the second, you make a "Fibonacci bracelet." The bracelet in this example has 12 elements.

(a)■ Write a static method `fiboBracelet(int a, int b, int p)` that returns an `ArrayList<Integer>` holding the elements of the Fibonacci bracelet modulo *p*, starting with $F_0 = a;\ F_1 = b$. Assume that $0 \le a < p$ and $0 \le b < p$ and at least one of *a* and *b* is greater than 0. Do not include the two terms that repeat the first and the second terms. For example, `fiboBracelet(1, 8, 10)` should return an `ArrayList` of size 12 that holds the values [1, 8, 9, 7, 6, 3, 9, 2, 1, 3, 4, 7].

(b)■ Write a short program that finds the lengths of the shortest and the longest possible bracelets modulo *p* and prints the lengths of these bracelets for all *p* in the range from 5 to 15. (A valid bracelet can start with any $0 \le a < p$ and $0 \le b < p$, except $a = b = 0$.)

(c)◆ Give a mathematical proof of the fact that any integers $0 \le a < p$ and $0 \le b < p$ generate a Fibonacci bracelet modulo *p* of a finite length.

≷ Hint: the length of the bracelet never exceeds p^2. ≷ ✓



Class Hierarchies

12.1 Prologue

In the previous chapters we discussed the basics of OOP philosophy and design principles. We now continue with more advanced OOP concepts: class hierarchies, abstract classes, polymorphism, and interfaces. But first, let us quickly review the basics.

As you know, a class can extend another class. This feature of OOP programming languages is called *inheritance*. The base class is called a *superclass*, and the derived class is called a *subclass* (Figure 12-1). The statement

```
public class D extends B
```

declares that *D* is a subclass of *B*.

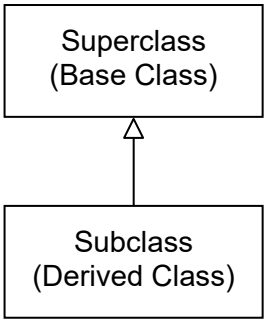


Figure 12-1. Inheritance terminology and notation in class diagrams: a subclass `extends` superclass

A subclass inherits all the fields and methods of its superclass (but not its constructors). An object of a subclass also inherits the type of the superclass as its own secondary, more generic type.

Inheritance represents the IS-A relationship between types of objects. A superclass defines more general features of the objects of its subclass; a subclass defines additional, more specific features (fields and methods) and may override (redefine) some of the methods of the superclass. In Figure 12-2, for example, `RoundBalloon`

`extends Balloon` (that is, `RoundBalloon` is a *subclass* of `Balloon`). The class `Balloon` is a generic class that represents any kind of "balloon." `RoundBalloon` represents a balloon of a particular shape. We say that a `RoundBalloon` IS-A(n) `Balloon`.

As you can see in Figure 12-2, a class can have several subclasses. A subclass can have its own subclasses. And so on. Once the concept of inheritance is introduced into programming, it becomes possible to create a *hierarchy of classes*, with more general classes higher up in the hierarchy and more specific classes lower down.

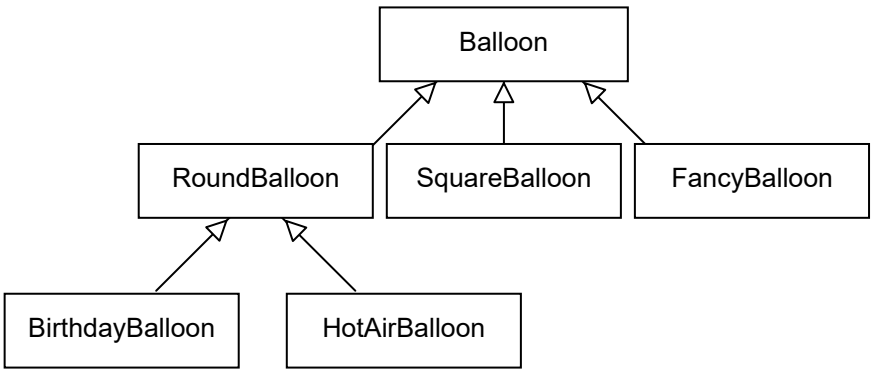


Figure 12-2. A hierarchy of classes

The concept of a hierarchy of types of objects in computer science can be traced to Artificial Intelligence (AI) research: in particular, to studies of models for representing knowledge using hierarchical *taxonomies*. Taxonomy is a system of classification in which an object can be defined as a special case of another object. In Linnaeus' zoological taxonomy, for example, a human being is a kind of primate, which is a kind of mammal, which is a kind of vertebrate, which is a kind of animal. Taxonomies have been one of the main ways of thinking in natural science for centuries, and they undoubtedly reflect an inclination of the human mind toward descriptive hierarchies.

In the rest of this chapter we will examine the use of class hierarchies as Java tools for managing different levels of abstraction.

12.2 Class Hierarchies

Why do we need to make `RoundBalloon`, `SquareBalloon`, and `FancyBalloon` subclasses of `Balloon`? Why can't they be independent classes? The reason is that they have a lot of common code.

Duplicating code in several classes is not only wasteful, but also bad for software maintenance.

This is because, if you ever need to change something in the code, you will spend a lot of time making the same changes in several different classes.

If you define a number of classes for a project and realize that these classes share a lot of code, it may be a good idea to *factor out* the common fields and methods into one common superclass.

All "balloons" share some common features. These common features are not duplicated in the individual classes — they are collected in one common superclass `Balloon`.

Besides avoiding duplication of code, such an arrangement has another advantage: objects of `Balloon`'s subclasses get a secondary, more generic type (`Balloon`), which comes in handy in client classes. For example, the class `DrawingPanel` in the *BalloonDraw* project (Section 4.6) maintains a list of balloons:

```
private ArrayList<Balloon> balloons;
```

We call its `add` method to add a balloon to the list:

```
balloons.add(activeBalloon);
```

When we call this method, `activeBalloon` can be any kind of `Balloon` (a `RoundBalloon`, a `SquareBalloon`, and so on). The compiler allows that because objects of subclasses of `Balloon` have a secondary, more generic type — `Balloon`. If `RoundBalloon` and `SquareBalloon` were not subclasses of `Balloon`, we couldn't put them in the same list of balloons.

Each kind of balloon has a `distance` method that represents the "distance" from a point to the center, based on the balloon's shape. These methods are different. Java

automatically calls the appropriate `distance` method for each type of balloon — the OOP feature known as *polymorphism*. (Polymorphism is discussed later, in Section 12.6.)

❖ ❖ ❖

To summarize, arranging classes in an inheritance hierarchy helps us avoid duplicate code by "factoring out" common code from subclasses into their common superclass. A class hierarchy also helps us avoid duplicate code in client classes by letting us write more general methods and take advantage of polymorphism.

12.3 Abstract Classes

As we have seen in our *BalloonDraw* project (Section 4.6), we sometimes need to leave some of the methods in a class undefined. In the `Balloon` class, we left the method `draw` empty, because balloons of different shapes should be drawn differently. Leaving a method empty is not a very good approach, because if we fail to override it in a subclass, the method will do nothing. A better way is to formally state that the method is undefined. Java allows us to do that by using the keyword `abstract`:

```
public abstract void draw(Graphics g, boolean makeItFilled);
```

Notice that an abstract method does not have a body, not even empty braces, just a semicolon at the end of the header.

A class that has one or more abstract methods must be declared *abstract*. For example:

```
public abstract class Balloon
{
  ...

  public abstract void draw(Graphics g, boolean makeItFilled);
}
```

❖ ❖ ❖

Java doesn't allow us to create objects of an abstract class.

For example, if you use the "abstract" version of `Balloon` class and write

```
Balloon b = new Balloon();
```

— the compiler will give you an error message, something like this:

```
Balloon is abstract; cannot be instantiated.
```

This means you cannot create an *instance* (that is, an object) of an abstract class.

A class with no abstract methods is called a *concrete class*.

A concrete class supplies its own constructors and defines all the abstract methods inherited from its superclass and other ancestors higher up the inheritance line.

Abstract and concrete classes can be intermixed in the same hierarchy at different levels as long as all the classes at the bottom are concrete.

OO designers sometimes define rather elaborate hierarchies. Figure 12-3 shows a fragment of the hierarchy of Java library classes. You can clearly see how the `javax.swing` GUI classes, starting with `JComponent`, were added under `Container`, while all the old `java.awt` classes (`Button`, etc.) were left untouched. The hierarchy has the class `Object` at the very top.

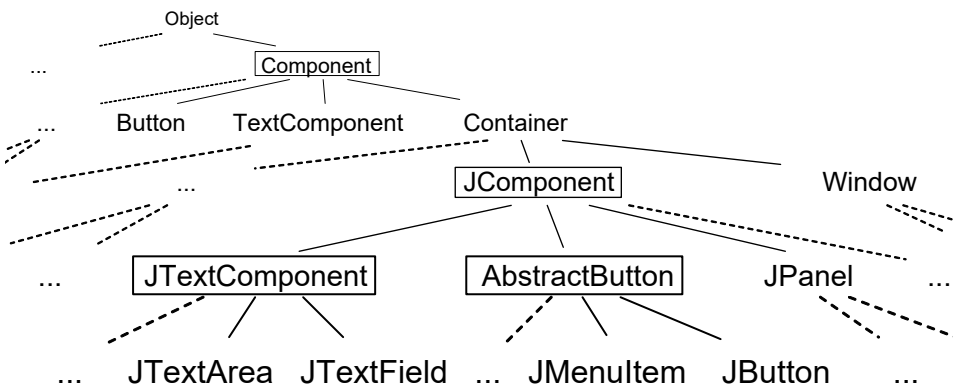


Figure 12-3. A fragment of Java library GUI class hierarchy (abstract classes are boxed)

In Java, if you do not explicitly specify a superclass for your class, your class automatically extends the class `Object`. So all Java classes fit into one giant hierarchical tree with `Object` at the root.

In other words, every object IS-A(n) `Object`. `Object` is a concrete (not abstract) class. It provides a few generic methods, such as `toString` and `equals`, but these methods are usually overridden in classes down the hierarchy.

12.4 Invoking Superclass's Constructors

In Section 4.5 we mentioned a paradox: a subclass inherits all the fields of its superclass but cannot access them directly if they are declared private. The solution is to provide public *accessor* methods for these fields. The `Balloon` class, for example, has the method `getColor`, which returns `Balloon`'s color. However, one question still remains: How do the private fields of the superclass get initialized?

One approach is to call public methods of the superclass to set its fields. For example, the `Balloon` class has a *modifier* method `setRadius`. `RoundBalloon`'s constructors can call that method. But a more common solution is to invoke a particular constructor of the superclass and pass the desired parameters to it.

Whenever an object of a subclass is created, the first thing Java does is it calls a constructor of its superclass. A programmer can specify which of the superclass's constructors to call and what parameters to pass to it using the keyword `super`.

For example:

```
public RoundBalloon(int x, int y, int r, Color c)
{
  super(x, y, r, c);
}
```

If your constructor has a `super(…)` statement, then it must be the first statement in the constructor.

What happens if the programmer does not include an explicit call to `super`?

If a subclass constructor does not make an explicit call to a superclass's constructor (does not have a `super` statement), then superclass's no-args constructor is called by default. In that case, the superclass must have a no-args constructor.

For example,

```
public RoundBalloon()
{
  // super(); -- optional, called by default
}
```

This `super` call is optional because `Balloon`'s no-args constructor is called by default. This works because the `Balloon` class does have a no-args constructor. If it didn't, the compiler would display an error message.

The class `Object` obviously has a no-args constructor, and so classes that "do not extend anything" (that is, classes that are derived directly from `Object`) rely on that no-args constructor and do not need a `super()` statement in their own constructors.

❖ ❖ ❖

In a hierarchy of classes, the superclass's constructor in turn calls its superclass's constructor, and so on, up the inheritance line, all the way up to `Object`. If `Car` extends `Vehicle`, which extends `Object`, then `Car`'s constructor calls `Vehicle`'s constructor, which calls `Object`'s constructor (Figure 12-4).

After all the calls to constructors of all the superclasses have been completed, the subclass's constructor can initialize subclass's own additional fields, if any, and perform other initialization tasks specific to the subclass.

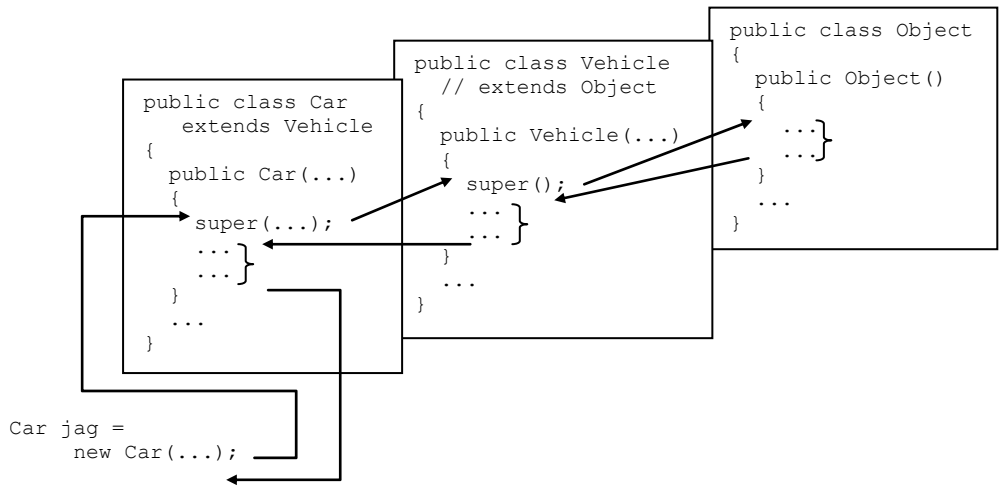


Figure 12-4. A constructor invokes its superclass's constructor, all the way up to `Object`

To summarize:

- Every class has at least one constructor. If no constructors are explicitly defined, the compiler provides a default no-args constructor. If at least one constructor is explicitly defined by the programmer, the compiler does not supply a default no-args constructor.
- If a constructor has a call to `super`, it must be the first statement in the constructor. The number and types of parameters passed to `super` must match the number and types of parameters expected by one of the constructors of the superclass.
- If `super(…)` does not appear in the constructor of a subclass, then the no-args constructor of the superclass is called by default. In that case, the superclass must have a no-args constructor.
- Placing

  ```
  super();
  ```

  in your constructor is redundant: it calls the superclass's no-args constructor, which is called by default anyway.

12.5 Calling Superclass's Methods

Sometimes a method or constructor in a subclass needs to call a superclass's method that it has overridden. For example, suppose the class `BirthdayBalloon` extends `RoundBalloon`. A `BirthdayBalloon` has the same shape as a `RoundBalloon`, but in addition to a solid disk we want to paint a number on it. It would be nice not to duplicate `RoundBalloon`'s code for drawing a circle, just to add the number on top. It turns out we can do that.

Java has special syntax for explicitly calling a method of the superclass. This syntax uses the `super`-dot prefix in the method call.

For example:

```
public void draw(Graphics g, boolean makeItFilled)
{
  super.draw(g, makeItFilled);  // call superclass's draw
  ... // other code
}
```

Without the super-dot prefix in the above code, `draw` would call itself repeatedly, and the program would eventually crash.

A call to a superclass's method with a super-dot prefix can be placed anywhere in a constructor or method of the subclass: it doesn't have to be the first statement, and, in fact, it doesn't have to be in the method that overrides the superclass's method.

The complete `BirthdayBalloon` class is shown in Figure 12-5.

```
public class BirthdayBalloon extends RoundBalloon
{
  private final Font ageFont = new Font(Font.SANS_SERIF, Font.PLAIN, 20);
  private int age;

  public BirthdayBalloon()
  {
    age = 16;
  }

  public BirthdayBalloon(int x, int y, int r, Color c)
  {
    super(x, y, r, c);
    age = 16;
  }

  public BirthdayBalloon(int x, int y, int r, Color c, int age)
  {
    super(x, y, r, c);
    this.age = age;
  }

  public void draw(Graphics g, boolean makeItFilled)
  {
    g.setFont(ageFont);
    super.draw(g, makeItFilled);
    g.setColor(Color.BLACK);
    g.drawString("" + age, getX() - 10, getY());
  }
}
```

Figure 12-5. JM\Ch12\HappyBirthday\BirthdayBalloon.java

❖ ❖ ❖

Calls to a superclass's method are frequently employed when implementing a subclass's `toString` method. Suppose the `Balloon` class has a `toString` method that returns the balloon's coordinates of the center, radius, and color, arranged in a string. Than `BirthdayBalloon`'s `toString` method may look like this:

```
  public String toString()
  {
    return super.toString() + " age=" + age;
  }
```

12.6 Polymorphism

Any object has a `toString` method. When you concatenate strings and objects of different types, the appropriate `toString` method is called for each object. For example:

```
Fraction f = new Fraction(1, 2);
Double d = new Double(0.5);
Color c = Color.RED;

String s = f + " " + d + " " + c;
// Compiled as:
// String s = f.toString() + " " + d.toString() + " " + c.toString();

System.out.println(s);

// Displays:
// 1/2 0.5 java.awt.Color[r=255,g=0,b=0]
```

The feature of Java and other OOP languages that makes this possible is called *polymorphism* (from Greek: *poly* = many; *morph* = form).

Polymorphism ensures that the object's method is called for an object of a specific type, even when that object is disguised as a reference to a more generic type (that is, the type of some ancestor higher up the inheritance line).

For another example of polymorphism, consider the `paintComponent` method in the `HappyBirthday` class (`JM\Ch12\HappyBirthday\HappyBirthday.java`) that draws all the balloons in the `balloons` list:

```
public void paintComponent(Graphics g)
{
  super.paintComponent(g);  // Call JPanel's paintComponent method
                            //  to paint the background
  for (Balloon b : balloons)
    b.draw(g, true);
}
```

Different types of balloons can be mixed in the `balloons` list. Each has a `draw` method, and these methods may have different code, but the appropriate `draw` method is called for each balloon.

Polymorphism is implemented in Java using the mechanism called *late* or *dynamic method binding*. When `DrawingPanel` is compiled, the compiler does not know which `draw` method to call for each balloon. This decision is left until run time. The compiler creates a table of entry points for the methods of each class. This table includes all the methods defined in the class itself plus all the methods inherited from classes higher up the inheritance line that are not overridden in this class. When `draw` is called for an object of the `BirthdayBalloon` class, that class's table of entry points for methods is used.

12.7 Interfaces

As we have seen, Java lets you create a neat hierarchy of classes and conveniently supports polymorphism. Together with some fields and methods, a subclass inherits a secondary data type, the data type of its superclass. However, class hierarchies in Java have a limitation: Java assumes that each class neatly falls into one hierarchy. In real life, the situation is more complex. The same object may play different roles in different situations. For example, a person named Natalie can be a student, an hourly employee, an online customer, a granddaughter, and a ballroom dancer. We would need five different class hierarchies to represent these aspects of Natalie: Natalie IS-A `Student`; Natalie IS-A(n) `Employee`; Natalie IS-A `Customer`; Natalie IS-A `FamilyMember`; Natalie IS-A `Dancer`.

It would be convenient for a class to be able to inherit certain features from one class and other features from another class. This is not possible in Java — it does not support such *multiple inheritance*. Some programming languages, such as C++ and Python, do support multiple inheritance, but not Java.

Still, Java provides a mechanism for a class to inherit if not the code, at least the data type from several sources. This mechanism is based on the concept of an *interface*.

A Java interface is similar to an abstract class: it has one or several abstract methods declared but left undefined. The difference is that all of an interface's methods are abstract: they have headers but no method bodies. Interfaces have no constructors and no other code (except, perhaps, a few static constants).*

* Starting with Java 8, an interface can have static and *default* methods defined — see `http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html`.

The keywords `public abstract` are omitted in the method headers in an interface because every method is public and abstract.

Once an interface is defined, we can "officially" state that a class *implements* that interface. If the class is concrete, it must supply all the methods listed in the interface. If the class is abstract, it can leave some of the interface's methods abstract.

`interface` and `implements` are Java reserved words.

The Java Collections Framework defines several interfaces in the `java.util` package to set the standards for what implementations of various data structures should be able to do. The `Collection` interface, with methods `isEmpty`, `size`, `add`, `remove`, `contains`, should be implemented by any collection. The `List` interface adds methods `get`, `set`, `indexOf`, and a few other methods. `java.util.ArrayList` and `java.util.LinkedList` implement this interface. The `Set` and `Map` interfaces and their implementations are described in Chapter 20.

You might ask: Why do we need interfaces? Why not just turn the interface into an abstract class? This has to do with the main difference between superclasses and interfaces: it is not possible for a class to have more than one superclass, but a class can implement any number of interfaces. We come across such situations frequently in OOP, and interfaces help. If a class implements several interfaces, that class supplies the code for all the methods of all the interfaces it implements. If it happens that the same method is declared in more than one interface, there is no conflict, because the methods of interfaces have no code — only headers.

If a concrete class implements several interfaces, it must supply all the methods specified in each of them.

An interface provides a secondary data type to objects of classes that implement that interface. Polymorphism fully applies to interface data types.

❖ ❖ ❖

We used two interfaces in the *Chomp* case study in Section 9.5: `Player` and `Strategy` (see Figure 9-3 on page 246). Each of the two players implements the interface `Player`. The reason we want both players to implement `Player` is that we want to hold them in the same array:

```
private Player[] players;
...

  players = new Player[2];
  players[0] = human;
  players[1] = computer;
```

This is convenient and adds flexibility: we can easily add other types of players in the future if we want (see Questions 16 and 17 in the exercises).

A secondary common data type is useful for mixing different but related types of objects in the same array, list, or other collection.

We could have made `Player` an abstract class rather than an interface, but the `HumanPlayer` and the `ComputerPlayer` do not share any code, so why waste the valuable "extends" slot in them? Eventually we might need it for something else.

The purpose for the `Strategy` interface is different: it isolates the subsystem that deals with different Chomp strategies from the rest of the program. `Chomp4by7Strategy` class implements the `Strategy` interface. A more advanced version of the *Chomp* program will need to support different strategies for different levels of play and board sizes. So `Strategy` serves, well, as an interface between different strategies and the rest of the program. If all the strategy classes implement the same `Strategy` interface, we can pass any particular strategy as a parameter to the `ComputerPlayer`'s `setStrategy` method.

The `Strategy` interface is also useful for splitting the work among team members: more math-savvy developers can work on the strategy classes. The `Chomp4by7Strategy` class, for example, is rather cryptic. Luckily we don't really need to know much about it beyond the fact that it implements the `Strategy` interface and has a no-args constructor. Whoever wrote and tested this class is responsible for it! This is team development at its best.

12.8 Summary

When classes *X* and *Y* share a lot of code, this is a clear indication that *X* and *Y* represent related types of objects. It then makes sense to "factor out" their shared code into a common superclass *B*. This helps avoid duplication of code in *X* and *Y* and, at the same time, provides a common secondary, more generic type *B* to objects of *X* and *Y*. This common type *B* allows you to write more general methods in clients of *X* and *Y*: instead of having separate overloaded methods

```
... someMethod(X x) { ... }
```

and

```
... someMethod(Y y) { ... }
```

you can have one method

```
... someMethod(B b) { ... }
```

The latter will work when you pass either an *X* or *Y* type of object *b* to `someMethod` as a parameter. The parameter *b* appears as type *B* in the method, but *b* itself knows what specific type of object it is (*X* or *Y*), and the appropriate method (*X*'s or *Y*'s) will be called automatically. This feature is called *polymorphism*. Polymorphism ensures that the correct method is called for an object of a specific type, even when that object is disguised as a reference to a more generic type.

A subclass of a class can have its own subclasses, and so on, so a programmer can design an *inheritance hierarchy* of classes. Classes lower in the hierarchy inherit fields and methods from their ancestors (the classes higher up along the inheritance line) and can add or redefine some of the methods along the way. Objects of a class also accumulate the IS-A relationships and the secondary types from all of the ancestors of that class.

Java allows you to leave some of the methods in a class declared but undefined. Such methods are called *abstract*. You "officially" declare a method abstract by using the keyword `abstract` in the method's header and supplying no code for the method, not even empty braces:

```
public abstract sometype someMethod(< parameters if any >);
```

If a class has at least one abstract method, it must be declared `abstract`. Abstract classes can have fields, constructors, and regular methods, but it is impossible to create objects of an abstract class (in other words, abstract classes cannot be *instantiated*). The purpose of an abstract class is to serve as a common superclass to two or several classes.

A class with no abstract methods is called a *concrete* class. A concrete class must define all of the abstract methods of its superclass. Abstract and concrete classes can be intermixed at different levels in the same inheritance hierarchy, as long as all the classes at the bottom are concrete.

In Java, if you do not explicitly specify a superclass for your class, your class will automatically extend the class `Object`. So all Java classes fit into one giant hierarchy tree with `Object` at the root. `Object` is a concrete (not abstract) class. It provides a few generic methods, such as `toString` and `equals`, but these are usually overridden in classes lower in the hierarchy.

Whenever an object of a subclass is created, the first thing Java does is to invoke a constructor of its superclass. The programmer specifies which one of the superclass's constructors to call and what parameters to pass to it using the keyword `super`. The call `super()` is optional. If there is no explicit `super(…)` statement, the no-args constructor of the superclass is invoked, and, in that case, the superclass is required to have one. If your constructor has a `super(…)` statement, then it must be the first statement in the constructor.

You can explicitly call a method defined in the superclass from a method defined in its subclass:

```
super.someMethod(...);
```

Such statement would typically be used in the code of the subclass's `someMethod`, the one that overrides the superclass's `someMethod`.

An *interface* is similar to an abstract class: it has one or several abstract methods declared but left undefined. The difference is that an interface does not have any code (except, perhaps, a few static constants): no constructors, no method bodies. In an interface, the keywords `public abstract` are not used in method headers because all methods declared in an interface are public and abstract.

Once an interface is defined, you can "officially" state that a class `implements` that interface. In that case, to be a concrete class, it must supply all the methods that are listed in the interface. An abstract class that implements an interface can leave some of the interface's methods abstract. The same class can implement several interfaces. In that case it should define all the methods specified in all of them.

An interface also provides a secondary type to objects of classes that implement that interface, and that secondary type can be used polymorphically the same way as the secondary type provided by a common superclass.

Exercises

1. True or false?

(a) You can't create objects of an abstract class. _____ ✓
(b) You can derive concrete classes from an abstract class. _____
(c) You can derive abstract classes from an abstract class. _____
(d) An abstract class can be useful as a common type for parameters in methods. _____
(e) `Object` is an abstract class. _____ ✓

2. True or false?

(a) In Java, the constructors of a subclass are inherited from its superclass. _____
(b) A subclass's constructor can explicitly call a superclass's constructor and pass parameters to it. _____
(c) A subclass's method can explicitly call a superclass's method. _____
(d) `super` is a Java reserved word. _____

3. Suppose we have

```
public abstract class Toy { ... }
public class Doll extends Toy { ... }
public class BarbieDoll extends Doll { ... }
```

`Doll` and `BarbieDoll` each have a no-args constructor. If an object `child` has a method `play(Doll doll)`, which of the following statements will compile with no errors?

(a) `child.play(new Object());` _____
(b) `child.play(new Toy());` _____
(c) `child.play(new Doll());` _____
(d) `child.play(new BarbieDoll());` _____

✓

4. Define a class `Diploma` and its subclass `DiplomaWithHonors`, so that the statements

```
Diploma diploma1 = new Diploma("Adam Smith", "Gardening");
System.out.println(diploma1);
System.out.println();

Diploma diploma2 = new DiplomaWithHonors("Betsy Smith",
                                         "Robotics");
System.out.println(diploma2);
System.out.println();
```

display

```
This certifies that Adam Smith
has completed a workshop in Gardening

This certifies that Betsy Smith
has completed a workshop in Robotics
*** with honors ***
```

Make your class definitions consistent with the information-hiding principle and avoid duplication of code. ✓

5. Define an abstract class `Poem` and two concrete subclasses of `Poem`, `Haiku` and `Limerick`. `Poem` has no constructors and has the following methods:

`public abstract int numLines()` — returns the number of lines in the poem.

`public abstract int getSyllables(int k)` — returns the number of syllables in the *k*-th line.

`public void printRhythm()` — shows the rhythm of the poem. For example, a haiku has 3 lines with 5, 7, and 5 syllables in them, so its `printRhythm` should print

```
ta-ta-ta-ta-ta
ta-ta-ta-ta-ta-ta-ta
ta-ta-ta-ta-ta
```

A limerick has 5 lines of 9, 9, 6, 6, and 9 syllables. Make sure the `printRhythm` method works for a `Haiku` and a `Limerick` without duplicating code.

6.■ The class `Triangle` (`JM\Ch12\Exercises\Triangle.java`) has methods for calculating the area, the perimeter, and their ratio. The class works for equilateral triangles and for right isosceles triangles; the type of the triangle is passed in a string to the constructor. The class also has a `main` method.

(a) Restructure this program in the OOP style. Make the `Triangle` class abstract. Keep the `side` field, but eliminate the `type` field. Make the `getArea` and `getPerimeter` methods abstract. Derive the concrete classes `EquilateralTriangle` and `RightTriangle` from `Triangle`. Provide an appropriate constructor for each of the two derived classes and make them call the superclass's constructor. Redefine the abstract methods appropriately in the derived classes. Put `main` in a separate test class and modify it appropriately.

(b) The area of a triangle is equal to one half of its perimeter times the radius of the inscribed circle. If the length of a side of an equilateral triangle is the same as the length of the legs in a right isosceles triangle, which of these triangles can hold a bigger circle inside? ✓

7.■ Explain why the method `equals` in the `String` class is defined as `equals(Object obj)` as opposed to `equals(String obj)`.

8.■ `MyArrayList` is a subclass of `ArrayList<Integer>`. When you print an object of `MyArrayList`, it is displayed with no opening and closing brackets and no comma separators. Fill in the blanks in the `MyArrayList`'s `toString` method below. ✓

```
public String toString()
{
  return ____________________________________

                ____________________________________________;
}
```

9.◆ Java developers made it impossible to extend the library class `java.util.Arrays`. Why did they make this decision? What device did they use to make extending `Arrays` impossible? ⦉ Hint: Try to extend this class and examine the error message. ⦊ Answer the same questions for the `Math` and `String` classes.

10. Derive a class `ArrayListWithSum` that extends `ArrayList<Integer>`, adding the method `sum` that returns the sum of the values in the list.

11.■ Explain why the `toString` method is never listed in any interface.

12. Java operator `instanceof` is used to ascertain the data type of an object. Will `x instanceof I` compile if `x` is an object and `I` is an interface? If so, when does it evaluate to `true`? Take a guess, then experiment and find out.

13. Consider the following interface:

```
public interface Place
{
  int distance(Place other);
}
```

Write a program that tests the following method:

```
// Returns true if p1 is equidistant from p2 and p3
public boolean sameDistance(Place p1, Place p2, Place p3)
{
  return p1.distance(p2) == p1.distance(p3);
}
```

✓

14.■ Design and implement a hierarchy of classes, such that the code fragment in a client class

```
ArrayList<Money> piggyBank = new ArrayList<Money>();
piggyBank.add(new Quarter());
piggyBank.add(new Bill(1));
piggyBank.add(new Nickel());
piggyBank.add(new Dime());
piggyBank.add(new Quarter());
piggyBank.add(new Bill(5));
System.out.println(piggyBank);
double amount = 0;
for (Money item : piggyBank)
  amount += item.getAmount();
System.out.printf("The piggy bank holds $%.2f\n", amount);
```

displays

```
[quarter, $1, nickel, dime, quarter, $5]
The piggy bank holds $6.65
```

`Money` should be an interface with one method. Each of the classes `Quarter`, `Nickel`, and `Dime` should extend the same superclass `Coin` and only have a no-args constructor and a `toString` method — no fields or other methods.

15.■ In the *Chomp* program from Chapter 9, make the board scalable. Eliminate the `CELLSIZE` constant in the `BoardPanel` class and obtain the cell's width and height from the current dimensions of the panel when you need them.

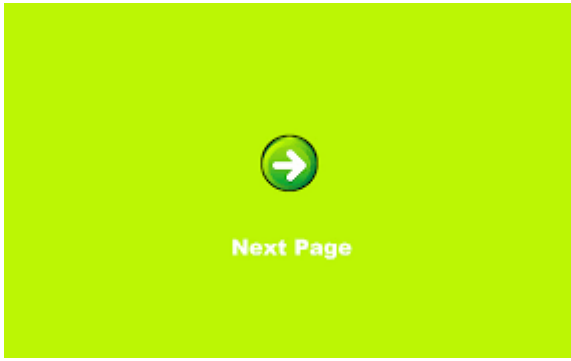
Add a constructor to the `BoardPanel` class that sets the row and column dimensions of the board. Make the program play on a 3 by 6 board. Which properties of the code make this change easy?

16.■ (a) Modify the *Chomp* program so that it can be played by two human players.

(b) Change the program further so that it displays different prompts for the two players (for example, "Your turn, Player 1" and "Your turn, Player 2"). Accept the name of the player as a parameter in `HumanPlayer`'s constructor and make `getPrompt` return a standard message concatenated with the player's name.

(c) Similar functionality can be achieved by deriving `HumanPlayer1` and `HumanPlayer2` from `HumanPlayer` and redefining the `getPrompt` method in them. Is this implementation more appropriate or less appropriate in an OOP program than the one suggested in Part (b)? Why? ✓

17.■ Turn the *Chomp* program into a game for three players: two human players and one computer player.


Next Page

```
Chapter(13)=
      Chapter(12)+1
```

Algorithms and Recursion

13.1 Prologue

In Chapter 7 we defined an *algorithm* as a more or less compact, general, and abstract step-by-step recipe that describes how to perform a certain task or solve a certain problem. We said that a reasonable algorithm folds the computation into one or several fragments that can be repeated multiple times. These repetitions, called *iterations*, execute exactly the same instructions but work with different values of variables.

Recursion is another tool for describing algorithms and implementing them in computer programs.

According to "Common Notion" Number 5 in Book I of Euclid's *Elements*,[elements] "The whole is greater than the part." This may be true for the lengths of segments and the volumes of solids in geometry, but in the intangible world of computer software the whole is sometimes the same as the part, at least in terms of its structural description and use.

Consider the picture in Figure 13-1. As you can see, it consists of graphics elements. Some of these elements are "primitives": lines, circles, rectangles, and so on. But others are smaller pictures in their own right. In fact, you can have pictures within pictures within pictures...

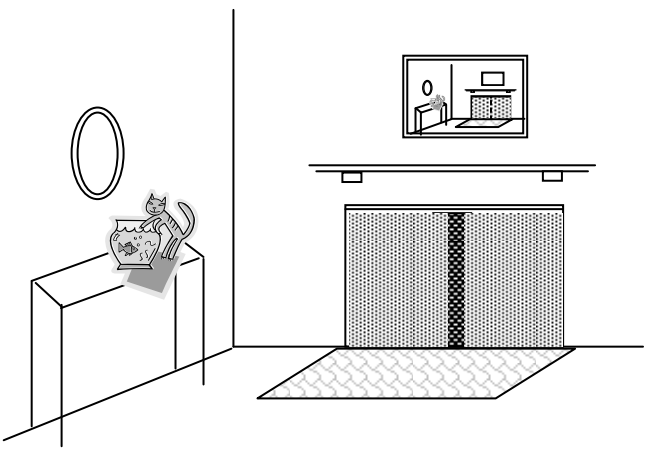


Figure 13-1. Some of the graphics elements are "primitives"; others are "pictures"

The overall complexity of the "whole" picture is greater than the complexity of each picture inside it, but the pictures inside may have the same basic structure as the big one. This is an example of a *recursive* structure whose substructures have the same form as the whole. Such structures are best handled by *recursive* procedures, which operate the same way on a substructure as on the whole structure.

In this chapter we only take a first look at recursion. Chapter 19 offers more in-depth coverage of this topic.

13.2 Recursive Methods

A recursive solution describes a procedure for a particular task in terms of applying the same procedure to a similar but smaller task.

A recursive solution also isolates simple situations, called *base cases* (or *stopping cases*) when a computation is obvious and recursion is not required.

For a recursive algorithm to work, it must have a base case (or cases) that do not need recursion, and the recursive calls must eventually reach the base case (or one of the base cases).

For example, in the task of calculating $1^2 + 2^2 + ... + n^2$, recursive thinking would go like this: if $n = 1$, the result is simply 1 (base case); if $n > 1$, we can calculate the sum $1^2 + 2^2 + ... + (n-1)^2$, then add n^2 to it. The underlined phrase represents a recursive call to the same procedure for a smaller number, $n-1$.

To most people, though, this algorithm will seem totally unsatisfactory. They may ask, "Right, but how do I calculate $1^2 + 2^2 + ... + (n-1)^2$? The algorithm doesn't tell me anything about that!" In fact it does, because the same procedure applies to any input, and, in particular, to $n-1$. So in order to calculate $1^2 + 2^2 + ... + (n-1)^2$ you will calculate $1^2 + 2^2 + ... + (n-2)^2$, then add $(n-1)^2$. And so on, until you get down to 1, and you know the answer for $n = 1$. This is how it might look in Java:

```
// Precondition: n >= 1
public static int addSquares(int n)
{
  if (n == 1)  // base case
    return 1;
  else
    return addSquares(n-1) + n*n;
}
```

Recursion is based on the mathematical concept that a definition of a function can use itself. This idea may seem absurd at first: if we define something using its own definition, we get a circular definition, right? Well, not necessarily. Consider, for example, the function $f(n) = n!$ (n-factorial). It is defined as the product of all integers from 1 to n: $n! = 1 \cdot 2 \cdot 3 \cdot ... \cdot n$. Is there a way to get rid of the "dot-dot-dot" in the description of this function? It turns out there is: we can define $n!$ recursively, as follows:

$$f(n) = \begin{cases} 1, & \text{if } n = 1 \\ f(n-1) \cdot n, & \text{if } n > 1 \end{cases}$$

The above definition tells us how to compute $f(n)$ if we know $f(n-1)$. But the same definition tells us how to compute $f(n-1)$ if we know $f(n-2)$. And $f(n-2)$ if we know $f(n-3)$. And so on. Eventually we will get to $f(1)$, and we know that $f(1) = 1$. Here is how a method that computes n-factorial might look in Java:

```
// Precondition: n >= 1
public static int factorial(int n)
{
  if (n == 1) // base case
    return 1;
  else
    return factorial(n-1) * n;
}
```

❖ ❖ ❖

Recursion may seem very tricky to an untrained eye, and some people have a hard time coming to grips with it. But it is easy for computers. In high-level languages, such as Python, Java, or C++, recursion is implemented by means of functions (methods) calling themselves.

A computer program handles recursion as a form of iterations, which however are hidden from the programmer.

Computers have a hardware mechanism, called the *stack*, that facilitates recursive function calls Figure 13-2.

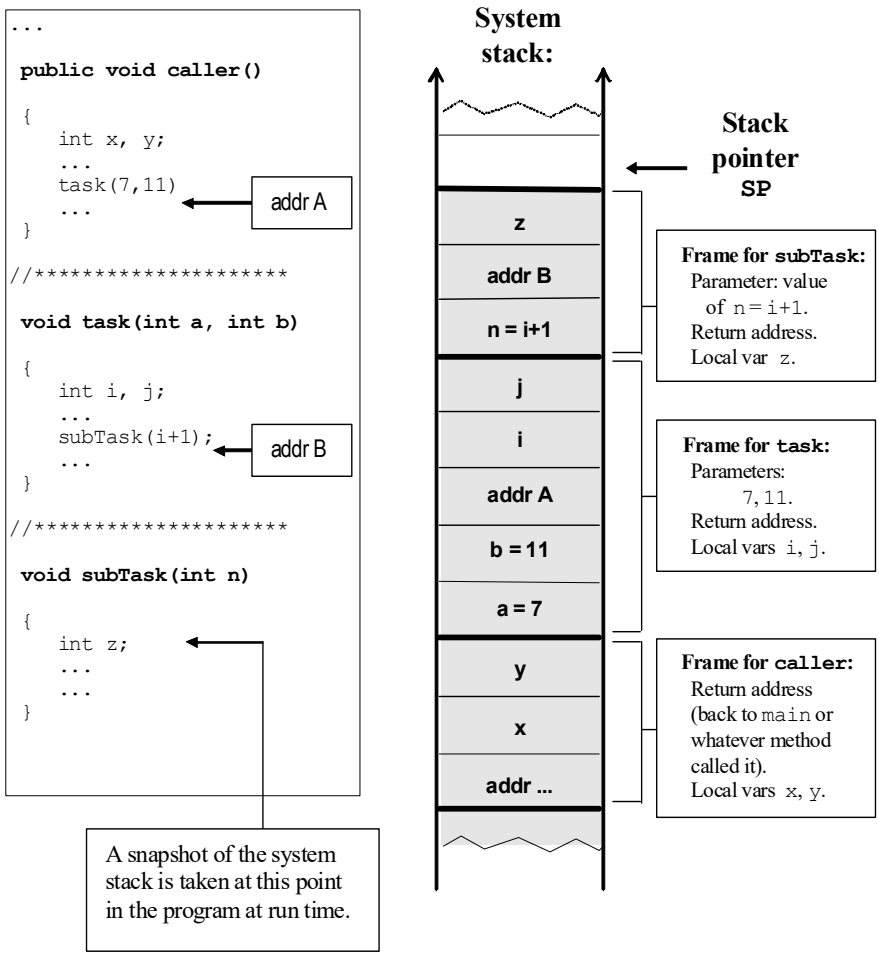


Figure 13-2. Frames on the system stack after a few method calls

We do not recommend recursive solutions for such computations as sum-of-squares or factorial because they can be easily and more efficiently handled with regular iterations. For example:

```
// Precondition: n >= 1
public static int factorial(int n)
{
  int f = 1;

  for (int k = 2; k <= n; k++)
    f *= k;

  return f;
}
```

Some people, once they understand it, find recursion so elegant that they are tempted to use it everywhere. Indeed recursive solutions may be short and expressive. But they may be also more costly than iterative solutions in terms of running time and memory space, and in some cases this cost becomes prohibitive.

In general, there is no need to use recursion when a simple iterative solution exists. Some programming languages, such as LISP or Scheme, almost always suggest the use of recursion instead of explicit iterations. Java programmers do not have to program recursive methods themselves very often. Still, understanding recursion is essential for understanding certain common algorithms such as Mergesort (Section 14.7) or dealing with branching or nested structures (such as the structure of pictures within pictures in the above example).

13.3 Tracing Recursive Methods

We can trace a non-recursive method by keeping track of the values of the variables as they change from one iteration to the next (see, for example, Figure 7-5 on page 183). The situation is more complicated for recursive methods. The difficulty is that a recursive method is defined in the top-down manner — it shows how to go from a bigger task to a smaller task — but to untangle the method we often need to proceed in the reverse direction, from a base case to a more general case.

Suppose for example that you are given the following recursive method:

```
public String someFun(String s)
{
  if (s.length() >= 2)
    s = someFun(s.substring(1)) + s.charAt(0);
  return s;
}
```

What does this method do? If you cannot guess right away, consider the base cases first. In this example the base cases, `s.length() == 0` and `s.length() == 1`, are implicit: nothing to do. In these cases the method returns `s` unchanged. Now take a string of length 2, for example, `s = "AB"`. `"AB".substring(1)` is `"B"` (length 1), so `someFun("AB".substring(1))` returns `"B"`. `"AB".charAt(0)` is `"A"`. Thus `someFun("AB")` returns `"B" + "A" = "BA"`.

Does `someFun` reverse the string or only move the first character to the end? If you are still not sure, take a string of length 3, say `"ABC"`. `"ABC".substring(1)` is `"BC"` and, as we have seen, `someFun("BC")` returns `"CB"`. `"ABC".charAt(0)` is `"A"`. Thus `someFun("ABC")` returns `"CB"+"A" = "CBA"`.

It looks like `someFun` returns a string with the characters from `s` in reverse order. Now that we have a working hypothesis, we can apply it to a string of any length. Our hypothesis is true when `s.length()` is 0 or 1 (the reversed string is the same as the original string). In the recursive case, `someFun` appends the first character of `s` at the end of the reversed substring of `s`. So `someFun` returns reversed `s`. A more rigorous proof would rely on the method of *mathematical induction*, which is explained in Section 19.4.

13.4 *Case Study:* File Manager

Recursion is especially suitable for handling nested structures. Consider for example the file system on your computer (Figure 13-3). A "file folder" is an inherently recursive definition: it is something that contains files and/or file folders! If you right-click on a folder in *Windows* and choose "Properties," you will see a screen that shows the total number of files in that folder and all its subfolders and the total number of bytes that the folder and all its files and all its subfolders take. How does your operating system compute that?

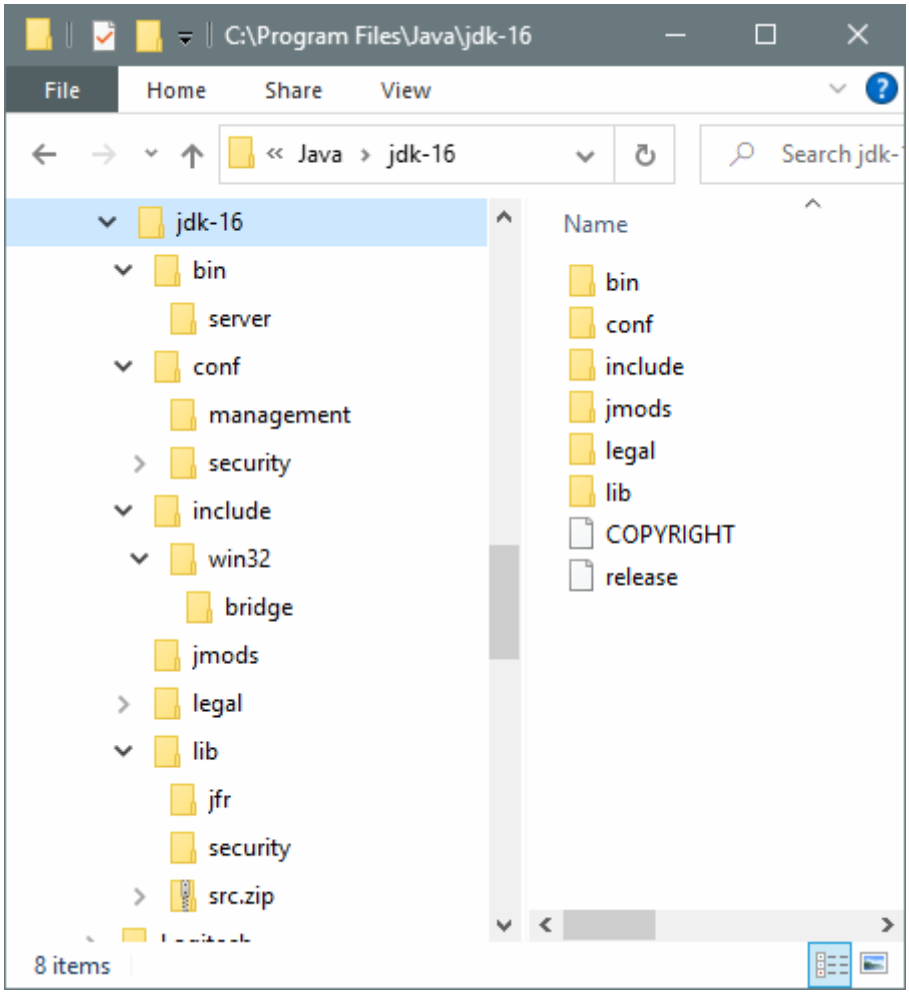


Figure 13-3. A snapshot of the *Windows Explorer* screen

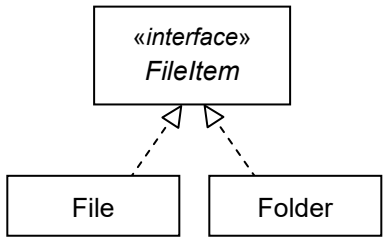
It is not easy to come up with an iterative algorithm to scan through all the files in every subfolder. When you jump into a deeper level, you need to remember where you are at the current level to be able to return there. You jump one level deeper again, and you need to save your current location again. To untangle this tree-like hierarchy of folders using iterations, you need a storage mechanism called a *stack*. But a recursive solution is straightforward and elegant. Here is a sketch of it in Java:

```
public int fileCount(List<FileItem> items)
{
  int count = 1;

  for (FileItem item : items)
  {
    if (item instanceof File)
      count++;
    else // if this item is a folder
      count += fileCount((List<FileItem>)item);
  }

  return count;
}
```

The base case here is a folder that contains only files and no other folders. Since the file structure is finite, recursion always stops. (Luckily we can't have a folder that holds itself as a subfolder — the operating system won't allow that.)

❖ ❖ ❖

Java/OOP is very appropriate for handling this type of structures. In a more elegant implementation, we can define an interface `FileItem` and two classes, `File` and `Folder`, that implement this interface:



As we say in OOP, a `File` IS-A (is a) `FileItem` and a `Folder` IS-A `FileItem`. A `Folder` also HAS-A list of `FileItem`s:

```
public class Folder implements FileItem
{
  private List<FileItem> items;
  ...
}
```

Suppose `FileItem` specifies two methods:

```
public interface FileItem
{
  int fileCount();
  int totalSize();
}
```

Let us define a `fileCount` method both for a `File` and a `Folder`. For a `File`, `fileCount` simply returns 1:

```
public class File implements FileItem
{
  ...

  public int fileCount()
  {
    return 1;
  }

  ...

}
```

For a `Folder`, the `fileCount` method is recursive: it returns the sum of all counts for all items in its `items` list:

```
public class Folder implements FileItem
{
  private List<FileItem> items;

  ...

  public int fileCount()
  {
    int count = 1;

    for (FileItem item : items)
      count += item.fileCount();

    return count;
  }

  ...

}
```

Due to polymorphism, Java will automatically call the correct `fileCount` method for each item in the `items` list, be it a `File` or a `Folder`.

Question 16 in the exercises asks you to write the `getSize` method for the `Folder` class.

13.5 Summary

Recursion is especially useful for handling nested structures, such as folders within folders or pictures within pictures. A recursive description of a data structure uses references to "smaller" structures of the same type. A recursive procedure that performs a certain task or computation relies on applying the same procedure to a similar but smaller task. A recursive method calls itself, but the parameters passed to this recursive call must be somehow "reduced" with each call. A recursive method also has one or several simple cases that do not need recursion. These are called *base* (or *stopping*) *cases*. All recursive calls must eventually terminate with a base case.

To understand a recursive method, start with the base case and examine a few "bigger" cases. Then formulate a hypothesis about what the method does and apply it to the general case to verify that it works.

Exercises

1. Let's pretend for a moment that Java does not support multiplication. Write a recursive version of the following method:

```
// Returns the product of a and b
// Precondition: a >= 0, b >= 0
public int product(int a, int b)
{
  ...
}
```

2. Consider the following recursive method:

```
public int mysterySum(int n)
{
  if (n == 0)
    return 0;
  else
    return 3 + mysterySum(n - 1);
}
```

What value is returned when `mysterySum(5)` is called? ✓

3. Consider the following method:

```
public String process(String str)
{
  int n = str.length();
  if (n >= 2)
  {
    int n2 = n/2;
    str = process(str.substring(n2)) +
          process(str.substring(0, n2));
  }
  return str;
}
```

What is the output from

```
System.out.println(process("HAVE") + " " + process("FUN"));
```

4. Write a recursive version of the following method.

```
// Returns the smallest among the first n elements of list
// Precondition: 1 <= n <= list.length
public int findMin(int[] list, int n)
```

Do not use any loops. ✓

5. Consider the following method:

```
public boolean isGood(String s)
{
  int n = s.length();
  return n < 2 || (s.charAt(0) == s.charAt(n-1) &&
                   isGood(s.substring(1, n-1)));
}
```

For which of the following strings will `isGood` return `true`?

- ❑ (a) `""` (empty string)
- ❑ (b) `"X"`
- ❑ (c) `"XOX"`
- ❑ (d) `"OXOX"`
- ❑ (e) `"XOOX"`
- ❑ (f) `"XXOX"`
- ❑ (g) `"XXXX"`
- ❑ (h) `"XOXOX"`

6. Suppose you have a method `printStars`, such that `printStars(n)` prints *n* stars on one line. For example, a statement

```
printStars(5);
```

displays the line

```
*****
```

(a) Using the `printStars` method, write a recursive method `printTriangle` so that `printTriangle(n)` prints a triangle with one star in the first row, two stars in the second row, and so on, up to `n` stars in the last row. For example, a statement

```
printTriangle(5);
```

should display

```
*
**
***
****
*****
```

Do not use any loops in your method.

(b) Modify your method `printTriangle` from Part (a) so that `printTriangle(5)` displays

```
*****
****
***
**
*
```

7. Consider

```
public void enigma(int n)
{
  for (int i = 0; i < n; i++)
    enigma(i);
  System.out.print(n);
}
```

Does the call `enigma(3)` terminate? If so, what is the output?

8. (a) Write a recursive method `sumDigits` that calculates and returns the sum of all the digits of a given non-negative integer. ✓

(b) Write a `boolean` method that tests whether a given number is evenly divisible by 3. A number is divisible by 3 if and only if the sum of its digits is divisible by 3. Use the `sumDigits` method from Part (a). Do not use any arithmetic operators in this method.

9. What is the output from the following method when called with $n = 3$? ✓

```
public void printX(int n)
{
  if (n <= 0)
    System.out.print(0);
  else
  {
    printX(n - 1);
    System.out.print(n);
    printX(n - 2);
  }
}
```

10. Rewrite the `gcf` method from Section 7.7 using recursion and not a single loop.

11.■ The following recursive method calculates 3^n:

```
// Precondition: n >= 0
public int power3(int n)
{
  if (n == 0)
    return 1;
  else
  {
    int p = power3(n/2);
    p *= p;

    if (n % 2 == 1)
      p *= 3;

    return p;
  }
}
```

How many multiplications will be performed when the program calls `power3(15)`?

12.■ Consider the following recursive method:

```
public long someFun(int n)
{
  if (n <= 0)
    return 2;
  else
    return someFun(n-1) * someFun(n-1);
}
```

(a) When the program calls `someFun(5)`, how many times will `someFun(3)` be called?

(b)■ (MC) What does this method calculate when the input parameter n is a non-negative integer? ✓

A. n^2 B. 2^n C. 2^{n+1} D. 2^{2n+1} E. $2^{(2^n)}$

13. What is the output from the following method when called with the argument $x = 2035$? ✓

```
public void display(int x)
{
  if (x >= 10)
  {
    display(x/10);
    System.out.print(x % 10);
  }
}
```

14.■ The following recursive method returns the n-th Fibonacci number:

```
// Returns the n-th Fibonacci number.
// Precondition: n >= 1
public static long fibonacci(int n)
{
  if (n == 1 || n == 2)
    return 1;
  else
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Rewrite it without recursion, using <u>one</u> loop.

15.■ The numbers $\binom{n}{0}, \binom{n}{1}, \binom{n}{2}, ..., \binom{n}{n}$ in the expansion

$$(x+y)^n = \binom{n}{0}x^n + \binom{n}{1}x^{n-1}y + \binom{n}{2}x^{n-2}y^2 + ... + \binom{n}{n-1}xy^{n-1} + \binom{n}{n}y^n$$

are called *binomial coefficients*. For example,

$(x+y)^2 = x^2 + 2xy + y^2$, so $\binom{2}{0}=1, \binom{2}{1}=2, \binom{2}{2}=1$.

$(x+y)^3 = x^3 + 3x^2y + 3xy^2 + y^3$, so $\binom{3}{0}=1, \binom{3}{1}=3, \binom{3}{2}=3, \binom{3}{3}=1$.

$(x+y)^4 = x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4$, so

$\binom{4}{0}=1, \binom{4}{1}=4, \binom{4}{2}=6, \binom{4}{3}=4, \binom{4}{4}=1$.

$\binom{n}{k}$ is pronounced "n-choose-k" and sometimes written as $C(n, k)$.

Binomial coefficients have the following properties: for any $n \geq 0$,

$\binom{n}{0}=\binom{n}{n}=1$, and for any integers n and k, such that $0 < k < n$,

$\binom{n}{k}=\binom{n-1}{k-1}+\binom{n-1}{k}$.

Complete the recursive method `binomialCoeff` below, which computes a specified binomial coefficient.

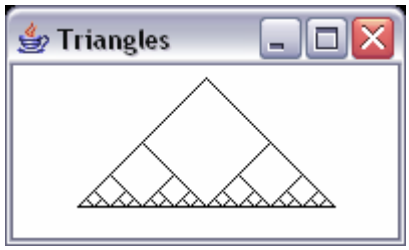
```
// Returns the value of the binomial coefficient C(n, k)
// Precondition: 0 <= k <= n
public int binomialCoeff(int n, int k)
{
   ...
}
```

16. Suppose the `File` class from Section 13.4 has a method `getSize()`, which returns the size of the file in bytes. Let's say that the size of a folder includes 512 bytes for the description of the folder itself, plus 128 bytes for the directory entry for each item (file or subfolder) in the folder, plus the sizes in bytes of all the items in the folder. Write the `getSize` method for a `Folder` that returns the size in bytes of the folder and all its subfolders and all the files in all the subfolders. ✓

17.■ Suppose we have added the `String getName()` method to the `FileItem` interface from Section 13.4 and implemented this method in the `File` and `Folder` classes. Write `boolean` methods `contains(String name)` for the `File` and `Folder` classes. For a `File`, `contains` should always return `false`. For a `Folder`, the method should return `true` if this folder or any of its subfolders contain a file or folder with the given name; otherwise it should return `false`.

18. Consider a sequence $x_1 = 1$, $x_2 = 1 + \frac{1}{1}$, $x_3 = 1 + \frac{1}{1 + \frac{1}{1}}$, ... In this sequence $x_{n+1} = 1 + \frac{1}{x_n}$ (for $n \geq 1$). In Chapter 7 Question 17 you wrote an iterative version of a method that computes x_n. Now write a recursive version of this method.

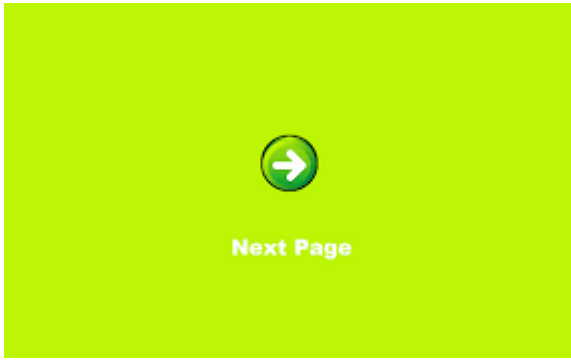
19.■ The program *Fractal* (`JM\Ch13\Exercises\Fractal.java`) displays a picture made of nested right isosceles triangles, as shown below.



The half-length of the base of the smallest triangle is 4. In the picture, the half-length of the base of the largest triangle is 64, and the coordinates of the midpoint of the base are (100, 100). Fill in the blanks in the recursive method `drawTriangles`. ☇ Hint: a statement

```
g.drawLine(x1, y1, x2, y2);
```

draws a line from point `(x1, y1)` to point `(x2, y2)` in the graphics context `g`. ☇


Next Page

14ACEHPRT

Searching and Sorting

14.1 Prologue

Searching and *sorting* are vast and important subjects. At the practical level they are important because they are what many large computer systems do much of the time. At the theoretical level they help distill the general properties and interesting theoretical questions about algorithms and data structures and offer rich material on which to study and compare them. We will consider these topics in the context of working with arrays, along with other common algorithms that work with arrays.

Searching tasks in computer applications range from finding a particular character in a string of a dozen characters to finding a record in a database of 100 million records. In the abstract, searching is a task involving a set of data elements represented in some way in computer memory. Each element includes a *key* that can be tested against a target value for an exact match. A successful search finds the element with a matching key and returns its location or some information associated with it: a value, a record, or the address of a record.

Searching refers to tasks where matching the keys against a specified target is straightforward and unambiguous. If, by comparison, we had to deal with a database of fingerprints and needed to find the best match for a given specimen, that application would fall into the category of *pattern recognition* rather than searching. It would also be likely to require the intervention of some human experts.

To *sort* means to arrange a list of data elements in ascending or descending order. The data elements may be numeric values or some records ordered by keys. In addition to preparing a data set for easier access (for example, as required for Binary Search), sorting has many other applications. One example is matching two data sets. Suppose we want to merge two large mailing lists and eliminate the duplicates. This task is straightforward when the lists are alphabetically sorted by name and address but may be unmanageable otherwise. Another use may be simply presenting information to a user in an ordered manner. A list of the user's files on a personal computer, for example, may be sorted by name, date, or type. A word processor sorts information when it automatically creates an index or a bibliography for a book. In large business systems, millions of transactions (for example, bank checks or credit card charges) are sorted daily before they are posted to customer accounts or forwarded to other payers.

In this chapter we first consider different ways of comparing objects in Java. We then look at two searching algorithms, Sequential Search and Binary Search, and

several common sorting algorithms: Selection Sort, Insertion Sort, and two faster ones: Mergesort and Quicksort.

14.2 equals, compareTo, and compare

Java offers three ways for comparing objects:

```
public boolean equals(Object other)
public int compareTo(T other)
public int compare(T obj1, T obj2)
```

where *T* is a type of objects. The `boolean` method `equals` compares `this` object to `other` for equality. The `int` method `compareTo` compares `this` object to another object of the same type and returns an integer that indicates whether `this` is greater than, equal to, or less than `other`. The `int` method `compare` compares two objects of the same type and returns an integer that indicates which of the two objects is greater than the other. Let us take a closer look at each of these methods: where they come from and how we can benefit from them.

1. `equals`

The `equals` method —

```
public boolean equals(Object other)
```

— is a method of the class `Object`. It compares the addresses of `this` and `other` and returns `true` if they are the same and `false` otherwise. Since every class has `Object` as an ancestor, every class inherits this method from `Object`. However, we are more often interested in comparing the contents of objects rather than their addresses (and we have the == operator to compare the addresses). So programmers often override `Object`'s `equals` method in their classes.

We have already seen `equals` in the `String` class: strings are compared character by character for an exact match. Consider another example, the class `Country` in Figure 14-1. The `equals` method in `Country` compares `this` country to `other` based on their names: two countries with the same name are considered equal. It is common for an `equals` method in a class to employ calls to `equals` for one or several of its fields.

```
public class Country implements Comparable<Country>
{
  private String name;
  private int population;

  public Country(String nm) { name = nm; population = 0; }
  public Country(String nm, int pop) { name = nm; population = pop; }
  public String getName() { return name; }
  public int getPopulation() { return population; }

  public boolean equals(Object other)
  {
    if (other != null)
      return name.equals(((Country)other).getName());
    else
      return false;
  }

  public int compareTo(Country other)
  {
    return name.compareTo(other.getName());
  }

  public String toString()
  {
    return name + ": " + population;
  }
}
```

Figure 14-1. JM\Ch14\Compare\Country.java

In order to override `Object`'s `equals` method in your class, the header of your `equals` method must be exactly the same as the header of `equals` in `Object`. In particular, the declared type of the parameter `other` must be `Object`.

That's why we had to cast `other` into `Country`. If you write

```
public boolean equals(Country other)  // Error
{
  if (other != null)
    return name.equals(other.getName());
  else
    return false;
}
```

you will define a different `equals` method and not override the one from `Object`. You might say: "So what? I only intend to compare countries to each other, not to objects of other types." The problem is that certain library methods, such as `contains` and `indexOf` in `ArrayList`, call your `equals` method polymorphically when they need to compare objects for equality. So if you plan to store your objects in an `ArrayList` (or another Java collection), you have to override the `Object` class's `equals` properly in your class.

"Then," you might wonder, "What happens if I accidentally pass an incompatible type of parameter to `equals`?" For example,

```
Country country = new Country("USA");
...
if (country.equals("USA"))    // error!
  ...
```

Since `"USA"` is an object, this code compiles with no errors, but at run time the `equals` method throws a `ClassCastException`, because it cannot cast a `String` into a `Country`. It would be better to catch such errors at compile time, but better late than never. The correct comparison would be

```
if (country.getName().equals("USA"))
  ...
```

or

```
Country usa = new Country("USA");
if (country.equals(usa))
  ...
```

Some programmers make `equals` simply return `false` if the parameter is of an incompatible type. This may be necessary if a programmer plans to mix different types of objects in the same array or list or another collection. For example:

```
public boolean equals(Object other)
{
  if (other instanceof Country)
    return name.equals(((Country)other).getName());
  else
    return false;
}
```

Java's `boolean` operator

```
x instanceof T
```

returns `true` if and only if *x* IS-A *T*. More precisely, if *x*'s class is *X*, *x* `instanceof` *T* returns `true` when *X* is exactly *T*, or *X* is a subclass of *T* or has *T* as an ancestor, or *T* is an interface and *X* implements *T*.

If you define `equals` this way, then you have to be extra careful when you use it because the program won't tell you when you pass a wrong type of parameter to `equals` — the call will just return `false`.

Note that overriding the `Object`'s `equals` method does not change the meaning of the == operator for the objects of your class: it still compares addresses of objects.

2. `compareTo`

The `compareTo` method is an abstract method defined in the `java.util.Comparable<T>` (pronounced *com-'parable*) interface, so there is no default implementation for it. To implement `Comparable<T>` in your class, you need to supply the following method:

```
public int compareTo(T other)
```

where *T* is the name of your class. For example:

```
public class Country implements Comparable<Country>
{
  ...
  public int compareTo(Country other)
  {
    return name.compareTo(other.getName());
  }
  ...
}
```

`compareTo` returns an `int`: a positive value indicates that `this` is "greater than" `other`; a zero indicates that they are "equal," and a negative value indicates that `this` is "less than" `other`. So `x.compareTo(y)` is sort of like `x` minus `y`.

It is the programmer who decides what is "greater" and what is "less." `compareTo` is said to define a *natural ordering* among the objects of your class. In the example in Figure 14-1, the "natural ordering" among countries is alphabetical, by name.

Note that `compareTo` takes a parameter of your class type, not `Object`.

Why do we need the `Comparable` interface and why would we want our classes to implement it? The reason is the same as for the `equals` method: certain library methods expect objects passed to them to be `Comparable`. For example, the `java.util.Arrays` class has a `sort(Object[] arr)` method, which compares elements of `arr` by calling their `compareTo` method. So if there is a reasonable ordering among the objects of your class, and you plan to use library methods or classes that deal with `Comparable` objects, it makes sense to make the objects of your class `Comparable`.

`String`, `Integer`, `Double`, and several other library classes implement `Comparable`.

If you do define a `compareTo` method in your class, don't forget to state in the header of your class

```
... implements Comparable<YourClass>
```

If your class implements `Comparable`, then it is a good idea to define the `equals` method, too, and to make `compareTo` and `equals` agree with each other, so that `x.equals(y)` returns `true` if and only if `x.compareTo(y)` returns 0. Otherwise, some of the library methods (and you yourself) might get confused.

Very well. You've made `Country` objects comparable. You can sort an array of `Country` objects by name using `Arrays.sort`. But suppose that sometimes you need to sort them by population. What can you do? Then you need a *comparator*.

3. `compare`

A *comparator* is an object that specifies a way to compare two objects of your class. Suppose the name of your class is *T*. A comparator is an object of a class that implements the `java.util.Comparator<T>` interface and has a method

```
public int compare(T obj1, T obj2)
```

If `compare` returns a positive integer, `obj1` is considered greater than `obj2`; if the returned value is 0, they are considered equal; if the returned value is negative, `obj1` is considered less than `obj2`. So `compare(obj1, obj2)` is sort of like `obj1` minus `obj2`.

The purpose of comparators is to be passed as parameters to constructors and methods of certain library classes (or your own classes). By creating different types of comparators, you can specify different ways of comparing objects of your class. You can create different comparators for ordering objects by different fields in ascending or descending order.

For example, the `PopulationComparator` class in Figure 14-2 defines comparators that compare countries by population. You can create an "ascending" comparator and a "descending" comparator by passing a `boolean` parameter to `PopulationComparator`'s constructor.

```
// Comparator for Country objects based on population

import java.util.Comparator;

public class PopulationComparator implements Comparator<Country>
{
  private boolean ascending;

  // Constructors
  public PopulationComparator() { ascending = true; }
  public PopulationComparator(boolean ascend) { ascending = ascend; }

  public int compare(Country country1, Country country2)
  {
    int diff = country1.getPopulation() - country2.getPopulation();
    if (ascending)
      return diff;
    else
      return -diff;
  }
}
```

Figure 14-2. JM\Ch14\Compare\PopulationComparator.java

The `Arrays` class has an overloaded version of the `sort` method that takes a comparator as a parameter. Now we can either rely on the natural ordering or create different comparators and pass them to `Arrays.sort` (Figure 14-3). The output from the `main` method in Figure 14-3 is

```
[Brazil: 193, China: 1338, India: 1188, Indonesia: 236, USA: 310]
[Brazil: 193, Indonesia: 236, USA: 310, India: 1188, China: 1338]
[China: 1338, India: 1188, USA: 310, Indonesia: 236, Brazil: 193]
```

```
import java.util.Arrays;

public class ComparatorTest
{
  public static void main(String[] args)
  {
    Country[] countries =
      { // population in millions as of 1/1/2015
        new Country("China", 1394),
        new Country("India", 1267),
        new Country("USA", 323),
        new Country("Indonesia", 253),
        new Country("Brazil", 202)
      };

    // Sort by name:
    Arrays.sort(countries);
    System.out.println(Arrays.toString(countries));

    // Sort by population ascending:
    Arrays.sort(countries, new PopulationComparator(true));
    System.out.println(Arrays.toString(countries));

    // Sort by population descending:
    Arrays.sort(countries, new PopulationComparator(false));
    System.out.println(Arrays.toString(countries));
  }
}
```

Figure 14-3. JM\Ch14\Compare\ComparatorTest.java

14.3 Sequential and Binary Search

Suppose we have an array of a certain size and we want to find the location of a given "target" value in that array (or ascertain that it is not there). If the elements of the array are in random order, we have no choice but to use *Sequential Search*, that is, to check the value of each consecutive element one by one until we find the target element (or finish scanning through the whole array). For example:

```
String[] words = { < ... some words> };
String target = < ... a word >;

for (int k = 0; k < words.length; k++)
{
  if (target.equals(words[k]))
    return k;
}
...
```

This may be time-consuming if the array is large. For an array of 1,000,000 elements, we will examine an average of 500,000 elements before finding the target (assuming that the target value is somewhere in the array). This algorithm is called an $O(n)$ ("order of n") algorithm because it takes an average number of operations roughly proportional to n, where n is the size of the array. ($O(...)$ is called the "big O" notation.)

Binary Search is a more efficient algorithm, which can be used if the elements of the array are arranged in ascending or descending order (or, as we say, the array is *sorted*). Let's say our array is sorted in ascending order and we are looking for a target value x. Take the middle element of the array and compare it with x. If they are equal, the target element is found. If x is smaller, the target element must be in the left half of the array, and if x is larger, the target must be in the right half of the array. Each time we repeat this procedure, we divide the remaining range of search into two approximately equal halves and narrow the search to one of them. This sequence stops when we find the target or get down to just one element, which happens very quickly.

A binary search in an array of 3 elements requires at most 2 comparisons to find the target value or establish that it is not in the array. An array of 7 elements requires at most 3 comparisons. An array of 15 elements requires at most 4 comparisons, and so on. In general, an array of $2^n - 1$ (or fewer) elements requires at most n comparisons. So an array of 1,000,000 elements will require at most 20 comparisons ($2^{20} - 1 = 1{,}048{,}575$), which is much better than 500,000. That is why such methods are called "divide and conquer."

Binary Search is an $O(\log n)$ algorithm because the number of operations it takes is roughly $\log_2 n$.

The `binarySearch` method in Figure 14-4 implements the Binary Search algorithm for an array of `Comparable` objects sorted in ascending order.

```
public class BinarySearch
{
  /**
   * Uses Binary Search to look for target in an array a, sorted in
   * ascending order.  If found, returns the index of the matching
   * element; otherwise returns -1.
   */
  public static <T> int find(T[] a,
                                    Comparable<? super T> target)

     // Wow!  We wanted to show you this bizarre syntax once!
     // <T> indicates that this method works for an array of
     // Comparable objects of any type T.  Comparable<? super T>
     // ensures that the method will work not only for a class T
     // that implements Comparable<T> but also for any subclass
     // of such a class.

  {
    int left = 0, right = a.length - 1;

    while (left <= right)
    {
      // Take the index of the middle element between
      //   "left" and "right":

      int middle = (left + right) / 2;

      // Compare this element to the target value
      // and adjust the search range accordingly:

      int diff = target.compareTo(a[middle]);

      if (diff < 0)   // target < a[middle]
        right = middle - 1;
      else if (diff > 0)   // target > a[middle]
        left = middle + 1;
      else   // target is equal to a[middle]
        return middle;
    }

    return -1;
  }
}
```

Figure 14-4. **JM\Ch14\BinarySearch\BinarySearch.java**

One way to understand and check code is to *trace* it manually on some representative examples. Let us take, for example:

```
Given:
    int[] a = {8, 13, 21, 34, 55, 89};
      // a[0] =  8; a[1] = 13; a[2] = 21; a[3] = 34;
      // a[4] = 55; a[5] = 89;
    target = 34

Initially:
    left = 0; right = a.length - 1 = 5

First iteration:
    middle = (0+5)/2 = 2;
    a[middle] = a[2] = 21;
    target > a[middle] (34 > 21)
      ==> Set left = middle + 1 = 3; (right remains 5)

Second iteration:
    middle = (3+5)/2 = 4;
    a[middle] = a[4] = 55;
    target < a[middle] (34 < 55)
      ==> Set right = middle - 1 = 3; (left remains 3)

Third iteration:
    middle = (3+3)/2 = 3;
    a[middle] = a[3] = 34;
    target == a[middle] (34 = 34)
      ==> return 3
```

A more comprehensive check should also include tracing special situations (such as when the target element is the first or the last element, or is not in the array) and "degenerate" cases, such as when `a.length` is equal to 0 or 1.

We also have to make sure that the method terminates — otherwise, the program may "hang." This is better accomplished by logical or mathematical reasoning than by tracing specific examples, because it is hard to foresee all the possible paths of an algorithm. Here we can reason as follows: our `binarySearch` method must terminate because on each iteration the difference `right - left` decreases by at least `1`. So eventually we either quit the loop via `return` (when the target is found), or reach a point where `right - left` becomes negative and the condition in the `while` loop becomes false.

14.4 *Lab:* Keeping Things in Order

In this lab you will write a class `SortedWordList`, which represents a list of words, sorted alphabetically (case blind). `SortedWordList` should extend `ArrayList<String>`. You have to redefine several of `ArrayList`'s methods to keep the list always alphabetically sorted and with no duplicate words.

1. Provide a no-args constructor and a constructor with one `int` parameter, the initial capacity of the list.

2. Redefine the `contains` and `indexOf` methods: make them use Binary Search.

3. Redefine the `set(i, word)` method so that it first checks whether `word` fits alphabetically between the (i−1)-th and (i+1)-th elements and is not equal to either of them. If this is not so, `set` should throw an `IllegalArgumentException`, as follows:

```
if (...)
  throw new IllegalArgumentException("word=" + word + " i=" + i);
```

4. Redefine the `add(i, word)` method so that it first checks whether `word` fits alphabetically between the (i−1)-th and i-th elements and is not equal to either of them. If this is not so, throw an `IllegalArgumentException`.

5. Redefine the `add(word)` method so that it inserts `word` into the list in alphabetical order. If `word` is already in the list, `add` should not insert it and should return `false`; otherwise, if successful, `add` should return `true`. Use Binary Search to locate the place where word should be inserted.

Combine your class with `SortedListTest.java` from `JM\Ch14\KeepInOrder` and test the program.

14.5 Selection Sort

The task of rearranging the elements of an array in ascending or descending order is called *sorting*. We are looking for a general algorithm that works for an array of any size and for any values of its elements. There exist many sorting algorithms for accomplishing this task, but the most straightforward one is probably *Selection Sort*.

Selection Sort

1. Initialize a variable *n* to the size of the array.
2. Find the largest among the first *n* elements.
3. Make it swap places with the *n*-th element.
4. Decrement *n* by 1.
5. Repeat steps 2 - 4 while $n \geq 2$.

On the first iteration we find the largest element of the array and swap it with the last element. The largest element is now in the correct place, from which it will never move again. We decrement *n*, pretending that the last element of the array does not exist anymore, and repeat the procedure until we have worked our way through the entire array. The iterations stop when there is only one element left, because it has already been compared with every other element and is guaranteed to be the smallest. The `SelectionSort` class in Figure 14-5 implements this algorithm for an array of the type `double`.

A similar procedure will sort the array in descending order; instead of finding the largest element on each iteration, we can simply find the smallest element among the first *n*.

Sorting is a common operation in computer applications and a favorite subject on which to study and compare algorithms. Selection Sort is an $O(n^2)$ algorithm because the number of comparisons in it is $n \cdot (n-1)/2$, which is roughly proportional to n^2. It is less efficient than other sorting algorithms considered in this chapter, but more predictable: it always takes the same number of comparisons, regardless of whether the array is almost sorted, randomly ordered, or sorted in reverse order.

```
public class SelectionSort
{
  // Sorts a[0], ..., a[a.length-1] in ascending order
  //   using Selection Sort.
  public static void sort(double[] a)
  {
    for (int n = a.length; n > 1; n--)
    {
      // Find the index iMax of the largest element
      //   among a[0], ..., a[n-1]:

      int iMax = 0;
      for (int i = 1; i < n; i++)
      {
        if (a[i] > a[iMax])
          iMax = i;
      }

      // Swap a[iMax] with a[n-1]:

      double aTemp = a[iMax];
      a[iMax] = a[n-1];
      a[n-1] = aTemp;

      // Decrement n (accomplished by n-- in the for loop).
    }
  }
}
```

Figure 14-5. **JM\Ch14\Benchmarks\SelectionSort.java**

14.6 Insertion Sort

The idea of the *Insertion Sort* algorithm is to keep the beginning part of the array sorted and insert each next element into the correct place in it. It involves the following steps:

Insertion Sort

1. Initialize a variable *n* to 1 (keep the first *n* elements sorted).
2. Save the next element and find the place to insert it among the first *n* so that the order is preserved.
3. Shift the elements as necessary to the right and insert the saved one in the created vacant slot.
4. Increment *n* by 1.
5. Repeat steps 2 - 4 while *n* < *array length*.

The `InsertionSort` class in Figure 14-6 implements this algorithm for an array of `doubles`.

```
public class InsertionSort
{
  // Sorts a[0], ..., a[a.length-1] in ascending order
  //   using Insertion Sort.
  public static void sort(double[] a)
  {
    for (int n = 1; n < a.length; n++)
    {
      // Save the next element to be inserted:
      double aTemp = a[n];

      // Going backward from a[n-1], shift elements to the
      //   right until you find an element a[i] <= aTemp:

      int i = n;
      while (i > 0 && aTemp < a[i-1])
      {
        a[i] = a[i-1];
        i--;
      }

      // Insert the saved element into a[i]:
      a[i] = aTemp;

      // Increment n (accomplished by n++ in the for loop).
    }
  }
}
```

Figure 14-6. **JM\Ch14\Benchmarks\InsertionSort.java**

Insertion Sort is also on average an $O(n^2)$ algorithm, but it can do better than Selection Sort when the array is already nearly sorted. In the best case, when the array is already sorted, Insertion Sort just verifies the order and becomes an $O(n)$ algorithm.

14.7 Mergesort

The tremendous difference in efficiency between Binary Search and Sequential Search hints at the possibility of faster sorting, too, if we could find a "divide and conquer" algorithm for sorting. Mergesort is one such algorithm. It works as follows:

Mergesort

1. If the array has only one element, do nothing.
2. (Optional) If the array has two elements, swap them if necessary.
3. Split the array into two approximately equal halves.
4. Sort the first half and the second half.
5. Merge both halves into one sorted array.

This recursive algorithm allows us to practice our recursive reasoning. Step 4 tells us to sort half of the array. But how will we sort it? Shall we use Selection Sort or Insertion Sort for it? Potentially we could, but then we wouldn't get the full benefit of faster sorting. For best performance we should use Mergesort again!

Thus it is very convenient to implement Mergesort in a recursive method, which calls itself. This fact may seem odd at first, but there is nothing paradoxical about it. Java and other high-level languages use a *stack* mechanism for calling methods. When a method is called, a new frame is allocated on the stack to hold the return address, the parameters, and all the local variables of a method (see Figure 13-2 on page 369). With this mechanism there is really no difference whether a method calls itself or any other method.

Recall that any recursive method must recognize two possibilities: a *base case* and a *recursive case*. In the base case, the task is so simple that there is little or nothing to do, and no recursive calls are needed. In Mergesort, the base case occurs when the array has only one or two elements. The recursive case must reduce the task to similar but smaller tasks. In Mergesort, the task of sorting an array is reduced to sorting two smaller arrays. This ensures that after several recursive calls the task will fall into the base case and recursion will stop.

Figure 14-7 shows a `Mergesort` class that can sort an array of `double`s. This straightforward implementation uses a temporary array into which the two sorted halves are merged. The `sort` method calls a recursive helper method that sorts a particular segment of the array.

```
public class Mergesort
{
  private static double[] temp;

  // Sorts a[0], ..., a[a.length-1] in ascending order
  //   using the Mergesort algorithm.
  public static void sort(double[] a)
  {
    int n = a.length;
    temp = new double[n];
    recursiveSort(a, 0, n-1);
  }

  // Recursive helper method: sorts a[from], ..., a[to]
  private static void recursiveSort(double[] a, int from, int to)
  {
    if (to - from < 2)       // Base case: 1 or 2 elements
    {
      if (to > from && a[to] < a[from])
      {
        // swap a[to] and a[from]
        double aTemp = a[to]; a[to] = a[from]; a[from] = aTemp;
      }
    }
    else                     // Recursive case
    {
      int middle = (from + to) / 2;
      recursiveSort(a, from, middle);
      recursiveSort(a, middle + 1, to);
      merge(a, from, middle, to);
    }
  }

  // Merges a[from] ... a[middle] and a[middle+1] ... a[to]
  //   into one sorted array a[from] ... a[to]
  private static void merge(double[] a, int from, int middle, int to)
  {
    int i = from, j = middle + 1, k = from;

    // While both arrays have elements left unprocessed:
    while (i <= middle && j <= to)
    {
```

Figure 14-7 `Mergesort.java` continued

```
        if (a[i] < a[j])
        {
          temp[k] = a[i];   // Or simply temp[k] = a[i++];
          i++;
        }
        else
        {
          temp[k] = a[j];
          j++;
        }
        k++;
      }

      // Copy the tail of the first half, if any, into temp:
      while (i <= middle)
      {
        temp[k] = a[i];     // Or simply temp[k++] = a[i++]
        i++;
        k++;
      }

      // Copy the tail of the second half, if any, into temp:
      while (j <= to)
      {
        temp[k] = a[j];     // Or simply temp[k++] = a[j++]
        j++;
        k++;
      }

      // Copy temp back into a
      for (k = from; k <= to; k++)
        a[k] = temp[k];
    }
  }
```

Figure 14-7. **JM\Ch14\Benchmarks\Mergesort.java**

The `merge` method is not recursive. To understand how it works, imagine two piles of cards, each sorted in ascending order and placed face up on the table. We want to merge them into the third, sorted, pile. On each step we take the smaller of the two exposed cards and place it face down on top of the destination pile. When one of the original piles is gone, we take all the remaining cards in the other one (the whole pile or one by one — it doesn't matter) and place them face down on top of the destination pile. We end up with the destination pile sorted in ascending order.

Mergesort is an $O(n \log n)$ algorithm — much better than the $O(n^2)$ performance of Selection Sort and Insertion Sort.

14.8 Quicksort

Quicksort is another $O(n \log n)$ sorting algorithm. The idea of Quicksort is to pick one element, called the *pivot*, then rearrange the elements of the array in such a way that all the elements to the left of the pivot are smaller than or equal to it, and all the elements to the right of the pivot are greater than or equal to it. The pivot element can be chosen arbitrarily among the elements of the array. This procedure is called *partitioning*. After partitioning, Quicksort is applied (recursively) to the left-of-pivot part and to the right-of-pivot part, which results in a sorted array.

Figure 14-8 illustrates the partitioning algorithm. You proceed from both ends of the array toward the meeting point comparing the elements with the pivot. If the element on the left is not greater than the pivot, you increment the index on the left side; if the element on the right is not less than the pivot, you decrement the index on the right side. When you reach a deadlock (the element on the left is greater than pivot and the element on the right is less than pivot), you swap them and advance both indices. When the left- and the right-side elements meet at a certain position, you swap the pivot with one of the elements that have met.

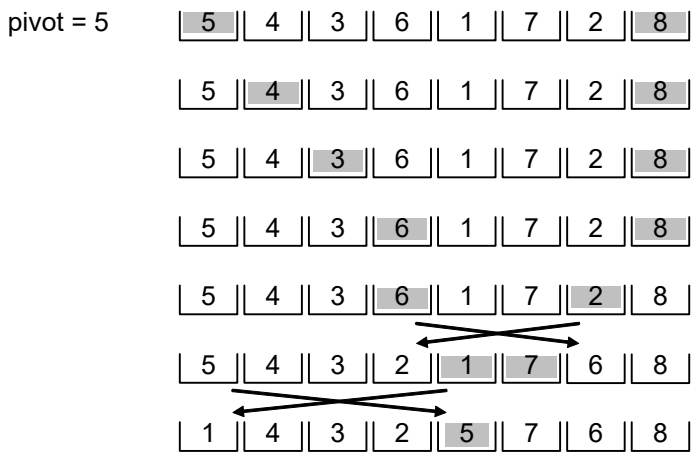


Figure 14-8. Array partitioning algorithm

The Quicksort algorithm was invented by C. A. R. Hoare in 1962. Although its performance is less predictable than Mergesort's, it averages a faster time for random arrays. Figure 14-9 shows an implementation of Quicksort in Java.

```
public class Quicksort
{
  // Sorts a[0], ..., a[a.length-1] in ascending order
  //   using the Quicksort algorithm.
  public static void sort(double[] a)
  {
    recursiveSort(a, 0, a.length - 1);
  }

  // Recursive helper method: sorts a[from], ..., a[to]
  private static void recursiveSort(double[] a, int from, int to)
  {
    if (from >= to)
      return;

    // Choose pivot a[p]:
    int p = (from + to ) / 2;
      // The choice of the pivot location may vary:
      //   you can also use p = from or p = to or use
      //   a fancier method, say, the median of the above three.

    // Partition:
    int i = from;
    int j = to;
    while (i <= j)
    {
      if (a[i] <= a[p])
        i++;
      else if (a[j] >= a[p])
        j--;
      else
      {
        swap (a, i, j);
        i++;
        j--;
      }
    }

    // Finish partitioning:
    if (p < j)    // place the pivot in its correct position
    {
      swap (a, j, p);
      p = j;
    }
    else if (p > i)
    {
      swap (a, i, p);
      p = i;
    }
```

Figure 14-9 *Quicksort.java continued*

```
      // Sort recursively:
      recursiveSort(a, from, p - 1);
      recursiveSort(a, p + 1, to);
    }

    private static void swap (double[] a, int i, int j)
    {
      double temp = a[i]; a[i] = a[j]; a[j] = temp;
    }
}
```

Figure 14-9. `JM\Ch14\Benchmarks\Quicksort.java`

14.9 *Lab:* Benchmarks

A benchmark is an empirical test of performance. The program in Figure 14-10 is designed to compare running times for several sorting methods. Enter the array size, select one of the four sorting methods in the "combo box" (pull-down list) and click the "Run" button. The program fills the array with random numbers and sorts them. This test is repeated many times for more accurate timing. (The default number of repetitions is set to 20, but you can enter that number as a command-line argument if you wish.) Then the program displays the average time it took to sort the array.

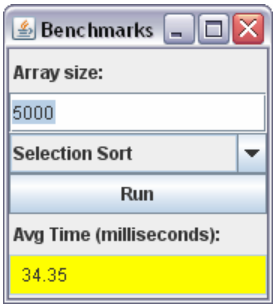


Figure 14-10. The *Benchmarks* program

Your task is to write the `runSort` method, which returns the total time spent sorting an array filled with random values the specified number of times.

First you need to learn how to generate a sequence of random numbers. To be precise, such numbers are called *pseudo-random*, because they are not entirely random: they are generated according to some algorithm. We have already used the `Math.random` method, but this time we won't use it because we want to have more control over how the sequence of random numbers is "seeded."

A "seed" is a value that is used to initialize the random number generator. If you seed the random number generator with the same seed, you will get exactly the same sequence of random numbers from it. Here we want to create a random number generator seeded each time the same way as we enter the `runSort` method, because we can call `runSort` for different sorting algorithms and want to compare them on exactly the same data.

To achieve that, use the `Random` class from the `java.util` package. `Random` has a no-args constructor that initializes the random number generator with an unpredictable seed, different each time. However, it has another constructor that takes a seed value as a parameter. This is the one you need to use. For example:

```
Random generator = new Random(seed);
```

Place this statement at the top of `runSort`.

Define `seed` as a `final` field in `Benchmarks`. The question remains, though, how to set its value. If we set it to a literal constant we will get exactly the same results each time we run *Benchmarks*. We do not want that: what if a particular seed value we picked produces a result that is not typical? We would never know. We will be more confident in our results if we use a different seed each time we run *Benchmarks* and the results are consistent. A common technique for obtaining a different seed value for different runs of a program is to set that value to the current system time in milliseconds. Thus, in `Benchmarks`'s constructor, we have

```
seed = System.currentTimeMillis();
```

Once you have created a random number generator (a `Random` object), you can generate the next random `double` by calling the generator's `nextDouble` method. For example:

```
a[k] = generator.nextDouble();   // 0.0 <= a[k] < 1.0
```

The `runSort` method should fill the array with random numbers and then sort it. This should be repeated `RUNS` times, and `runSort` should return the total time it took to sort the array in milliseconds. Call the system time before and after each sort and add the elapsed time to the total.

Set up a project with the files `SelectionSort.java`, `InsertionSort.java`, `Mergesort.java`, `Quicksort.java`, and `Benchmarks.java`, provided in `JM\Ch14\Benchmarks`. Fill in the blanks in the `Benchmarks` class. Once your program is working, collect benchmarks for each sorting algorithm for arrays of various sizes: ranging, say, from 10,000 to 50,000 elements (to 500,000 for Mergesort and Quicksort). Plot the running time vs. the array size for all four sorting methods. You can do this on your graphing calculator, manually, or by entering the results into a spreadsheet or another data analysis program. See how well your experimental results fit with parabolas for Selection Sort and Insertion Sort and with *n* log *n*-shaped curves for Mergesort and Quicksort.

14.10 `java.util.Arrays` and `java.util.Collections`

No matter what you need to do in Java, chances are it has been done before. Sure enough, `java.util` package has a class `Arrays` and a class `Collections` with methods that implement Binary Search and sorting (using a fast version of the Mergesort algorithm). You will need to import `java.util.Arrays` or `java.util.Collections` in order to use their methods (unless you are willing to type in the full names every time). All of `Arrays`'s and `Collections`'s methods are static, and you cannot create objects of these classes.

`Arrays`'s `binarySearch` method is called as follows:

```
int pos = Arrays.binarySearch(a, target);
```

`Arrays` class has overloaded versions of `binarySearch` for arrays of `char`s, `int`s, and other primitive data types. There is also a version for any `Comparable` objects. In particular, it can be used with `String`s.

`Arrays.sort` methods can sort an array of `char`s, `int`s, `double`s, `String`s, and so on, either the whole array or a segment within specified limits. For example:

```
String[] dictionary = new String[maxWords];
int wordsCount;
< ... other statements >

Arrays.sort(dictionary, 0, wordsCount - 1);
```

As we have seen, there is also a version that works with any `Comparable` objects and a version that takes a comparator as a parameter.

The `Arrays` class also offers a set of `fill` methods for different data types. A `fill` method fills an array or a portion of an array with a specified value. Another useful method is `toString`. For example,

```
int[] nums = {1, 2, 4, 8, 16};
System.out.println(Arrays.toString(nums));
```

produces the output

```
[1, 2, 4, 8, 16];
```

The `asList(T[] arr)` method returns a representation of the array `arr` as a fixed-length `List`. You can't add elements to this list, but you can call other methods and pass it as a parameter to library methods that expect a `List` or a `Collection`. For example, to shuffle the elements of the array `names` you can write

```
Collections.shuffle(Arrays.asList(names));
```

You can also print out the list:

```
System.out.println(Arrays.asList(names));
```

produces the same output as

```
System.out.println(Arrays.toString(names));
```

The `Collections` class works with Java collections, such as `ArrayList`. For example, to find a target word in `ArrayList<String> words`, you can call

```
Collections.binarySearch(words, target);
```

The `Collections` class also has methods to sort a `List` of `Comparable` objects (or to sort a `List` using a comparator), to shuffle a list, to copy a list, to fill a list with a given value, to find max or min, and other methods.

14.11 Summary

The `boolean` method `equals(Object other)` in the `Object` class compares the address of `this` object to `other`. The correct way to override the `equals` method in your class is:

```
public boolean equals(Object other)
{
  if (other != null)
    ...
  else
    return false;
}
```

The `int` method `compareTo` of the `java.util.Comparable<T>` interface has the following header:

```
int compareTo(T other)
```

`compareTo` returns an `int` value that indicates whether `this` is larger than, equal to, or less than `other`. It is like "`this` minus `other`." To implement the `Comparable` interface in `SomeClass`, state

```
public class SomeClass implements Comparable<SomeClass>
```

and provide a method

```
public int compareTo(SomeClass other) { ... }
```

that returns a positive integer if `this` is greater than `other`, 0 if they are equal, and a negative integer if `this` is less than `other`. The main reason for making objects of your class `Comparable` is that certain library methods and data structures work with `Comparable` objects. If provided, the `compareTo` method should agree with the `equals` method in your class.

A *comparator* is an object dedicated to comparing two objects of `SomeClass`. The comparator's class implements the `java.util.Comparator<SomeClass>` interface and provides a method

```
public int compare(SomeClass obj1, SomeClass obj2) { ... }
```

The `compare` method returns an integer, sort of like "`obj1 - obj2`." Comparators are passed as parameters to constructors and methods of library classes, telling them how to compare the objects of your class.

Sequential Search is used to find a target value in an array. If the array is sorted in ascending or descending order, *Binary Search* is a much more efficient searching method. It is called a "divide and conquer" method because on each step the size of the searching range is cut in half.

The four sorting algorithms discussed in this chapter work as follows:

Selection Sort

Set n to the size of the array. While n is greater than 1, repeat the following: find the largest element among the first n, swap it with the n-th element of the array, and decrement n by one.

Insertion Sort

Keep the first n elements sorted. Starting at $n = 2$, repeat the following: find the place for `a[n]` in order among the first $n - 1$ elements, shift the required number of elements to the right to make room, and insert `a[n]`. Increment n by one.

Mergesort

If the array size is less than or equal to 2, just swap the elements if necessary. Otherwise, split the array into two halves. Recursively sort the first half and the second half, then merge the two sorted halves.

Quicksort

Choose a pivot element and partition the array, so that all the elements on the left side of pivot are less than or equal to the pivot and all the elements on the right side of pivot are greater than or equal to the pivot; then recursively sort each part.

Selection Sort is the slowest and most predictable of the four: each element is always compared to every other element. Insertion Sort works quickly on arrays that are almost sorted. Mergesort and Quicksort are both "divide and conquer" algorithms. They work much faster than the other two algorithms on arrays with random values.

Java's `Arrays` and `Collections` classes from the `java.util` package have `binarySearch`, `sort`, and other useful methods. All `Arrays` and `Collections` methods are static. The `Arrays` methods work with arrays, with elements of primitive data types, as well as with `String`s and other `Comparable` objects. `Arrays` also offers the convenient methods `fill`, `toString(`*`T`*`[] arr)`, where *`T`* is a primitive or a class data type, and `asList(Object[] arr)`. The `Collections` class's methods work with `List`s, such as `ArrayList`, and other Java collections.

Exercises

Sections 14.1-14.2

1. Describe the difference between searching and pattern recognition.

2. Suppose a class `Person` implements `Comparable<Person>`. A `Person` has the `getFirstName()` and `getLastName()` methods; each of them returns a `String`. Write a `compareTo` method for this class. It should compare this person to another by comparing their last names; if they are equal, it should compare their first names. The resulting order should be alphabetical, as in a telephone directory. ✓

3.■ Make the `Fraction` objects, defined in Chapter 10, `Comparable`. The natural ordering for `Fraction`s should be the same as for the rational numbers that they represent. Also override `Object`'s `equals` method in `Fraction` to make it agree with this ordering.

4. (a) Write a class `QuadraticFunction` that represents a quadratic function $ax^2 + bx + c$, with integer coefficients *a*, *b*, and *c*. Provide a constructor with three `int` parameters for *a*, *b*, and *c*. Provide a method `double valueAt(double x)` that returns the value of this quadratic function at *x*. Also provide a `toString` method. For example,

```
System.out.println(new QuadraticFunction(1, -5, 6));
```

should display

```
x^2-5x+6
```

(b) Override `Object`'s `equals` method in the `QuadraticFunction` class. Two `QuadraticFunction`s should be considered equal if their respective coefficients are equal.

(c) Make the `QuadraticFunction` objects `Comparable`. The `compareTo` method should first compare the *a*-coefficients; if equal, then compare the *b*-coefficients; if also equal, compare the *c*-coefficients. (This ordering basically defines which function will have greater values for very large *x*.)

Continued

(d)■ Define a comparator class for comparing two `QuadraticFunction` objects. Provide two constructors: a no-args constructor and a constructor that takes one `double` parameter. When a comparator is created by the no-args constructor, it should compare two `QuadraticFunction`s based on their values at $x = 0$; when a comparator is created by the constructor with a parameter x, it should compare `QuadraticFunction`s based on their values at x.

Sections 14.3-14.4

5. Describe a situation where the performance of Sequential Search on average is better than $O(n)$. ≤ Hint: different target values in the array do not have to come with the same probability. ≥ ✓

6. Given a sorted array of 80 elements, what is the number of comparisons required by Binary Search in the worst case? Consider two scenarios:

(a) We know for sure that the target value is always in the array; ✓
(b) The target may be not in the array. ✓

7.■ A string contains several X's followed by several O's. Devise a divide-and-conquer algorithm that finds the number of X's in the string in $\log_2 n$ steps, where n is the length of the string.

8.■ Write a recursive method that tries to find `target` among the elements `a[m]`, ..., `a[n-1]` of a given array (any array, not necessarily sorted).

```
public static int search(int[] a, int m, int n, int target)
```

If found, the method should return the position of the target value; otherwise it should return −1. The base case is when the searching range is empty or consists of one element ($m \geq n$). For the recursive case, split the searching range into two approximately equal halves and try to find the target in each of them. ✓

9.■ Write a recursive implementation of Binary Search with specifications similar to the `search` method in Question 8.

10.◆ An array originally contained different numbers in ascending order but may have been subsequently rotated by a few positions. For example, the resulting array may be:

```
21 34 55 1 2 3 5 8 13
```

Is it possible to adapt the Binary Search algorithm for such data? Explain.

Sections 14.5-14.11

11. Mark true or false and explain:

(a) If the original array was already sorted, 190 comparisons would be performed in a Selection Sort of an array containing 20 elements. _____ ✓

(b) Mergesort works faster than Insertion Sort on any array. _____ ✓

12. An array of six integers — 6, 9, 74, 10, 22, 81 — is being sorted in ascending order. Show the state of the array after two iterations through the outer `for` loop in Selection Sort (as shown in Figure 14-5).

13. What are the values stored in an array `a` after five iterations in the `for` loop of the Insertion Sort (as shown in Figure 14-6) if the initial values are 3, 7, 5, 8, 2, 0, 1, 9, 4, 3? ✓

14. Write a variation of Insertion Sort in which the tail of the array is kept sorted and the next element is inserted in the proper place in the tail of the array.

15. What is the state of the array after the partitioning phase of the Quicksort algorithm, at the top level of recursion, if its initial values are

```
int[] a = {6, 9, 74, 10, 22, 81, 2, 11, 54};
```

and the middle element is chosen as a pivot? ✓

16. Add `Arrays`'s built-in sorting method to the *Benchmarks* program to see how it compares to our own sorting methods.

17. Research online *Timsort* invented by Tim Peters in 2002. What programming languages and libraries use it? How does it compare to Mergesort?

```
outputFile.printf
    ("Chapter %d", 15);
```

Streams and Files

15.1 Prologue

Any program that processes a considerable amount of data has to read the data from a file (or several files) and is likely to write data into a file. A file is a software entity supported by the operating system. The operating system provides commands for renaming, copying, moving, and deleting files, as well as low-level functions, callable from programs, for opening, reading, and writing files. A file has to be opened before it can be read. To open a file the program needs to know its *pathname* in the system (the path to the folder that holds the file and the name of the file). A new file has to be created before we can write into it. To create a new file, a program needs to know its path and what name to give it.

Data files are not a part of the program source code, and they are not compiled. The same executable program can work on different files as long as it can handle the particular format of the data in the files and knows the file pathnames (or receives them from the user at run time).

A file can contain any kind of information: text, images, sound clips, binary numbers, or any other information that can be stored digitally. The size of a file is measured in bytes. The format of the data in a file is determined by the program that created that file. There are many standardized file formats that allow different programs to understand the same files. A standard format is usually designated by the file name's extension. For example, standard formats for representing compressed images may have the `.gif` or `.jpg` extensions, music files may have the `.mp3` extension, and so on. A text file often has the `.txt` extension. A Java source file with the extension `.java` is a text file, too.

A computer program may treat a file as either a *stream* or a *random-access file*. The term "stream" has its origins in operating systems, such as *UNIX* and *MS-DOS*. It refers to the abstract model of an input or output device in which an input device produces a stream of bytes and an output device receives a stream of bytes. Some input/output devices, such as a keyboard and a printer, are rather close to this abstract model. Ignoring the technical details, we can say that a keyboard produces an input stream of characters and a printer receives an output stream.

Other devices, such as graphics adapters, hard drives, or flash drives, are actually random-access devices rather than stream devices: software can transfer a whole block of bytes to or from any sector on a disk or set the values of pixels anywhere on a screen. Still, the operating system software lets us implement input/output from/to

a disk file as a logical stream. For example, when we read from a disk file, the input is *buffered*: whole chunks of data are read into specially allocated memory space, called *buffer*, and the program takes bytes or characters from the buffer. Likewise, a console window on the screen can be thought of as a logical output stream of characters positioned line by line from left to right.

A stream can be opened for input (reading) or for output (writing), but not for both at the same time. Bytes are read sequentially from an input stream, although Java input streams have a method to skip several bytes. Bytes are written to an output stream sequentially, without gaps. If an output stream represents an existing file, a program can open it in the "create" mode and start writing from the beginning, destroying the old contents, or it can open it in the "append" mode and start adding data at the end of the file.

In a random-access file, the program can start reading or writing at any place. In fact, a random-access file can be opened for both input and output at the same time.

A program can work with several input and/or output files concurrently.

In a text file, each line is marked by a terminating end-of-line character or combination of such characters (for example, CR+LF, "carriage return" + "line feed").

Files that do not hold text are often called *binary files*, because any byte in a file holds eight bits of binary data.

Most programming languages provide special functions for reading characters or a line of text from a text file or writing to a text file or to the screen.

It is more common to treat text files as streams. A binary file with fixed-length records can be opened as a random-access file.

This is so because text lines may have different lengths, and it is hard to track where each line begins. Therefore random access is not useful for a text file. In a binary file, if all the records have the same length, we can easily calculate the position of the *n*-th record; then we can go directly to that place and read or write that record.

Programming languages usually include library functions or classes for opening, reading, creating, and writing files. These functions call low-level functions provided by the operating system. Programming languages and operating systems also support "standard input" and "standard output" streams for reading keyboard

input and writing text to the screen. These streams are implemented in a manner similar to file streams, but they are automatically open when a program is running.
Java's classes for handling input and output streams and files are rich in functionality but confusing to a novice. Java's I/O package `java.io` offers two sets of classes: one for dealing with streams of bytes, the other for dealing with streams of characters. The byte stream input and output hierarchies are rooted in the abstract classes `InputStream` and `OutputStream`, respectively. The character stream input and output class hierarchies are rooted in the abstract classes `Reader` and `Writer`, respectively.

In the past, there was little difference between bytes and characters because each byte held the code for one character. With the internationalization of software and the introduction of Unicode, the situation has changed. As you know, in Unicode a character is represented by two bytes. "Native" files in many operating systems still use one byte per character, using a subset of Unicode. In the U.S. version of *Windows*, for example, text files use ASCII encoding. Java character streams provide conversion from the native or specified encoding to Unicode.

In this chapter we consider only reading from and writing to text files using some of Java's character stream classes. We won't deal with reading or writing to a file in a random-access manner.

15.2 Pathnames and the `java.io.File` Class

In a typical operating system, such as *Unix* or *Windows*, files are arranged in a hierarchy of nested *directories* (or folders). Each file is identified by its *pathname*, which consists of the drive letter, a sequence of directories that lead to the file from the root directory, and the file name and extension. For example, the pathname for the file `words.txt` in the `Dictionary` subdirectory in the `Ch15` subdirectory in the `JM` directory (folder) on drive `C` is `C:/JM/Ch15/Dictionary/words.txt`. This is called *absolute pathname* because it traces the file's location all the way up to the root directory.

The class `File` in the `java.io` package represents an entry in the operating system's file management subsystem. (A `File` can refer to a file or the whole subdirectory.) `File` has a constructor with one `String` parameter, a pathname. The following statement, for example, creates a `File` object for the file named `words.txt`:

```
File wordsFile = new File("words.txt");
```

Note that we have not used the absolute pathname for the file in the above statement.

Programs almost never refer to a file's absolute pathname explicitly, because if the folders somewhere above the file are rearranged or renamed, the program won't be able to find the file.

Most programs refer to a file simply by its name or using a *relative pathname*, which starts at some folder but does not go all the way up to the root. The above declaration of `wordsFile`, for example, assumes that the file `words.txt` is located in the same directory (folder) as the program's compiled class files. If the class files were in `Ch15\Classes`, and `words.txt` were in `Ch15\Data`, we would write

```
File wordsFile = new File("../Data/words.txt");
```

`".."` means "go one directory level up."

Figure 15-1 shows a small subset of `File`'s methods.

```
String getName();            // Returns the name of this file.
String getAbsolutePath();    // Returns the absolute pathname
                             //   of this file.
long length();               // Returns the size of this file
                             //   in bytes.
long lastModified();         // Returns the time when this file was
                             //   created or last modified.
boolean isDirectory();       // Returns true if this file represents
                             //   a subdirectory; otherwise
                             //   returns false.
File[] listFiles();          // Returns an array of files in the
                             //   subdirectory represented by this
                             //   file.
```

Figure 15-1. A subset of `java.io.File`'s methods

Note that a `File` object does not represent a file ready for reading or writing. In fact, a directory entry associated with a `File` object may not even exist yet. You need to construct a readable or writable stream associated with a file to read or write the data.

Another way to obtain a reference to a `File` in a Java program is to use a `JFileChooser` GUI component (from the `javax.swing` package). Figure 15-2 gives an example of that.

```
// Set the initial path for the file chooser:
private String pathname = System.getProperty("user.dir") + "/";
...

  ...
  JFileChooser fileChooser = new JFileChooser(pathname);

  // Allow choosing only files, but not subfolders:
  fileChooser.setFileSelectionMode(JFileChooser.FILES_ONLY);

  // Open a dialog box for choosing a file; locate it
  // in the middle of the JFrame window (or use null to
  // locate the dialog box in the middle of the screen):
  int result = fileChooser.showOpenDialog(window);

  // Check whether the "Cancel" button was clicked:
  if (result == JFileChooser.CANCEL_OPTION)
    return;

  // Get the chosen file:
  File file = fileChooser.getSelectedFile();

  // Save pathname to be used as a starting point for
  // the next JFileChooser:
  pathname = file.getAbsolutePath();
  ...
```

Figure 15-2. A `JFileChooser` example

15.3 Reading from a Text File

Java I/O classes provide many ways to accomplish the same task. Java developers have recognized the need to simplify the `java.io` package for novices, and in the 5.0 release of Java they have introduced a new class, `Scanner`, for reading numbers, words, and lines from a text file.

The `Scanner` class has been added to the `java.util` package.

To use this class, you first need to create a `Scanner` associated with a particular file. You can do this by using a constructor that takes a `File` object as a parameter.

Be careful: `Scanner` also has a constructor that takes a parameter of the type `String`. However, that string is not a pathname, but rather a string to be used as an input stream.

If a file described by `File file` does not exist, `new Scanner(file)` throws a `FileNotFoundException`, which you have to catch. For example:

```
String pathname = "words.txt";
File file = new File(pathname);
Scanner input = null;
try
{
  input = new Scanner(file);
}
catch (FileNotFoundException ex)
{
  System.out.println("*** Cannot open " + pathname + " ***");
  System.exit(1);  // quit the program
}
```

In general, Java I/O classes report errors by throwing "checked" exceptions. A checked exception is a type of event that can either be "caught" inside a method using the `try-catch` syntax or left unprocessed and passed up to the calling method (or to the Java run-time environment). In the latter case, you have to declare up front that your method might `throw` a particular type of exception. For example:

```
public StringBuffer loadFile(String pathname)
            throws IOException
  // this method doesn't have to catch I/O exceptions
{
  ...
}
```

The `Scanner` class is easier to use than other `java.io` classes because its methods do not throw checked exceptions — only its constructor does.

Some of the `Scanner` methods are summarized in Figure 15-3.

Don't forget

```
import java.io.*;
import java.util.Scanner;
```

when working with `java.io` classes and `Scanner`.

```
boolean hasNextLine();        // Returns true if this stream has another
                              //   line in it; otherwise returns false.
String nextLine();            // Reads all the characters from the
                              //   current position in the input stream
                              //   up to and including the next newline
                              //   character; removes these characters
                              //   from the stream and returns a string
                              //   that holds the read characters
                              //   (excluding newline).
boolean hasNext();            // Returns true if this stream has another
                              //   token (a contiguous block of
                              //   non-whitespace characters) in it.
String next();                // Reads the next token from this stream,
                              //   removes it from the stream, and
                              //   returns the read token.
boolean hasNextInt();         // Returns true if the next token in
                              //   this stream represents an integer.
int nextInt();                // Reads the next token from this stream,
                              //   removes it from the stream, and
                              //   returns its int value.
boolean hasNextDouble();      // Returns true if the next token in
                              //   this stream represents a double.
double nextDouble();          // Reads the next token from this stream,
                              //   removes it from the stream, and
                              //   returns its value as a double.
void close();                 // Closes this file.
```

Figure 15-3. A subset of `java.util.Scanner`'s methods

Once a `Scanner` object has been created, you can call its methods to read `int`s, `double`s, words, and lines. For example:

```
int sum = 0;
while(input.hasNextInt())
{
  int score = input.nextInt();
  sum += score;
}
```

A `Scanner` assumes that tokens (numbers, words) are separated by whitespace. `Scanner`'s `next`, `nextInt`, and `nextDouble` methods skip the white space between tokens. `Scanner` does not have a method to read a single character.

Note that if a token read by `next`, `nextInt`, or `nextDouble` is the last one on a line, then the newline character that follows the token remains in the stream. Before

reading the next line, first get rid of the tail of the previous line (unless you will continue reading `int`s, `double`s, or individual words). For example:

```
int num = input.nextInt();
input.nextLine();               // skip the rest of the line
                                //   and newline
String str = input.nextLine();  // read the next line
```

❖ ❖ ❖

If you need to read a file character by character, use a `FileReader` with a `BufferedReader` "wrapped" around it. Figure 15-4 gives an example.

```
public static StringBuffer loadFile(String pathname)
      throws IOException
{
  File file = new File(pathname);
  StringBuffer strBuffer = new StringBuffer((int)file.length());
  BufferedReader input = new BufferedReader(new FileReader(file));

  int ch = 0;
  while ((ch = input.read()) != -1)
      strBuffer.append((char)ch);  // input.read() returns an int

  input.close();
  return strBuffer;
}
```

Figure 15-4. Reading a file character by character using a `BufferedReader`

15.4 Writing to a Text File

Use the class `PrintWriter` to write to a text file. This class has two constructors: one creates an output character stream from a pathname string, the other from a `File` object. Each throws a `FileNotFoundException` if the file cannot be created. This exception has to be caught.

Be careful: if you try to write to a file and a file with the given pathname already exists, it is truncated to zero size and the information in the existing file is lost.

The `PrintWriter` class has `println`, `print`, and `printf` methods similar to the ones we use with `System.out`. The `printf` method, added in Java 5.0, is used for writing formatted output (see Section 8.5).

It is important to close the file by calling its `close` method when you have finished writing. Otherwise some of the data that you sent to the output stream may end up not written to the file.

Data is usually written not directly to disk, but into an intermediate buffer. When the file is closed, the data remaining in the buffer is written to the file.

Figure 15-5 gives an example of creating a text file.

```
    String pathname = "output.txt";
    File file = new File(pathname);
    PrintWriter output = null;

    try
    {
      output = new PrintWriter(file);
    }
    catch (FileNotFoundException ex)
    {
      System.out.println("*** Cannot create " + pathname + " ***");
      System.exit(1);  // quit the program
    }

    output.println("Metric measures:");
    output.printf("%2d kg = %5.3f lbs\n", 1, 2.2046226);
    output.close();

/* output.txt will contain:
Metric measures:
 1 kg = 2.205 lbs    */
```

Figure 15-5. Creating a text file

If you want to append text to an existing file, use a `PrintWriter` "wrapped" around a `FileWriter` (Figure 15-6).

```
String pathname = "output.txt";
Writer writer = null;
try
{
  writer = new FileWriter(pathname, true);
     // "true" means open in the append mode
}
catch (IOException ex)  // Note: not FileNotFoundException!
{
  System.out.println("*** Cannot create/open " + pathname + " ***");
  System.exit(1);  // quit the program
}

PrintWriter output = new PrintWriter(writer);
output.printf("%2d km = %5.3f mile\n", 1, 0.6213712);
output.close();

/* output.txt will contain:
Metric measures:
 1 kg = 2.205 lbs
 1 km = 0.621 mile    */
```

Figure 15-6. Appending data to a text file

15.5 *Lab:* Choosing Words

Suppose I am working on a program for a word game. (I call my game Ramblecs.) My game uses three-, four-, and five-letter words. In this lab you will help me create a dictionary for my program. You will start with a text file, `words.txt`, which contains about 20,000 English words (`JM\Ch15\Dictionary\words.txt`). Your program (a simple console application) should choose the three-, four-, and five-letter words from `words.txt`, convert them into the upper case, and write them to an output file. Make the output file use the syntax of a Java class, as shown in Figure 15-7. Then I will be able to use this class in my program.

At first glance, this plan seems to contradict our earlier statement that data files are different from program source files. In fact there is no contradiction. In this lab, both the input file and the output file are data files. It just so happens that the output file uses Java syntax. The same thing happens when you create Java source using a program editor: for the editor, the resulting file is just some output text.

```
public class RamblecsDictionary
{
  private String[] words =
  {
    "ABACK",
    "ABASE",
    "ABASH",
    "ABATE",
    ...
    ...
    "ZIPPY",
    "ZLOTY",
    "ZONE",
    "ZOO",
    "ZOOM",
  };
}
```

Figure 15-7. The output file format in the *Dictionary Maker* program

Use a `Scanner` to read words from `words.txt` and a `PrintWriter` to create `RamblecsDictionary.java`. Review the output file and run it through the Java compiler to verify that the output of your program is correct.

15.6 Summary

Java's I/O classes are rich in functionality but are not very easy to use.

Use an object of the `java.util.Scanner` class to read a text file. `Scanner` has a constructor that takes a `File` object as a parameter and opens that file for reading. Some of `Scanner`'s methods are summarized in Figure 15-3.

Use an object of the `java.io.PrintWriter` class to write to a text file. `PrintWriter` has a constructor that takes a `File` as a parameter and another constructor that takes a `String pathname` as a parameter. It creates a new file and opens it for writing (or, if the file already exists, opens it and truncates it to zero size). `PrintWriter` has convenient `print`, `println`, and `printf` methods, which are similar to `System.out`'s methods.

Exercises

1. (MC) Which of the following is true? ✓

A. Any file can be opened for reading either as a stream or as a random-access file.
B. All files in *UNIX* are streams, while in *Windows* some may be random-access files.
C. In both *Windows* and *UNIX*, all text files are streams and all binary (non-text) files are random-access files.
D. When a file is first created, an attribute is set that designates this file as a stream or as a random-access file for all future applications.

2. Explain why random-access files usually contain fixed-length records.

3. Many methods in Java stream I/O classes throw an exception whenever an I/O error occurs. It would perhaps be more convenient for a programmer to handle certain errors by making the program check the return value or the stream status after the operation rather than letting it throw an exception. Which errors among the following would be good candidates for such no-exception treatment?

(a) Failure to open a file because it does not exist _____ ✓
(b) Failure to open a file that is already opened by another application _____ ✓
(c) Failure to create a file because a read-only file with the same name already exists _____ ✓
(d) Device is not ready for reading (for example, a CD-ROM drive is empty) _____
(e) System error reading from a device (for example, a damaged sector on disk) _____
(f) System write error (for example, disk full) _____
(g) End of file is encountered while trying to read data _____

4. Write a program that checks whether all the braces are balanced in a Java source file. ≤ Hint: read the file one character at a time. When you encounter an opening brace, increment the count, and when you encounter a closing brace, decrement the count. The count should never be negative and should be zero at the end. ≥ ✓

5. Write a program that compares two files for exactly matching data. ✓

6.■ *grep* is an old utility program from *UNIX* that scans a file or several files for a given word (or a regular expression) and prints out all lines in which that word occurs. (The name "grep" comes from the "qed/ed" editor commands g/re/p — globally search for a regular expression and print the lines.)

Write a simplified version of *grep* that looks for a word in one file. A "word" is defined as a contiguous string of non-whitespace characters. For every line that contains the word, print out the line number and the text of the line in brackets. For example:

```
Line  5: [  private static PrintWriter output;]
```

Prompt the user for the target word and the file name or take them from the command-line arguments.

The search for a matching word can be accomplished by using `String`'s method `indexOf`. This method will find the target word even if it occurs as part of another word in the line. If you want to find only matches between complete words, write your own method. ≤ Hint: search for all occurrences of the word in the line and then check that the matching substring is surrounded by not "isLetterOrDigit" characters wherever it doesn't touch the beginning or the end of the line. ≥

7.■ Write a class that represents a picture: a 2-D array of characters set to either `'x'` or `'.'`. Supply a constructor that loads the picture from a file with a specified name. The first line of the file contains two integers, the number of rows and the number of columns in the picture. The following lines, one for each row, contain strings of `x`'s and dots. The length of each string is equal to the number of columns in the picture. For example:

```
3 4
x.xx
xx.x
..x.
```

Supply a `toString` method for your class and test your class. ✓

8.■ Write and test a program that merges two sorted files into one sorted file. The files contain lines of text that are sorted lexicographically (the same ordering used by `String`'s `compareTo` method). Your program should read each file only once and should not use arrays or lists.

9.■ A word is said to be an *anagram* of another word if it is made up of the same letters arranged in a different order. (For instance, "GARDEN" and "DANGER" are anagrams.) Write a program that finds all anagrams of a given target word in a file of words. Follow these steps:

(a) Write a method that reads the words from a dictionary file into an array of strings.

(b) Write a method that creates an array of letter counts for a word. The array should have 26 elements; the *k*-th element represents the number of times the *k*-th letter of the alphabet occurs in the word.

(c) Write a method that compares two letter count arrays for an exact match.

(d) For each word in the dictionary, generate a letter count array, compare it with the letter count array for the target word, and print out the word if they match.

(e) See if you can find any anagrams for "RAMBLECS" in `JM\Ch15\Dictionary\words.txt`.

Continued ☞

(f) There is another way to ascertain that two words are anagrams of each other: if we sort the letters in each word in ascending order, we should get equal strings. Implement this anagram searching algorithm instead of the one described in Parts (b) - (d). ≤ Hints: (1) The `String` class has a convenient method `toCharArray`, which returns an array of `char`s from the string, in the same order. Use `Arrays.sort` to sort that array. (2) Convert the sorted `char` array back into a `String` by using the `String(char[])` constructor. ≥

10.◆ In the Mad Libs™ party game, the leader has the text of a short story with a few missing words in it. The missing words are tagged by their part of speech or function: <noun>, <verb>, <place>, etc. For example:

```
It was a <adjective> summer day.
Jack was sitting in a <place>.
...
```

The leader examines the text and prompts the players for the missing words:

```
Please give me an/a:
adjective
place
...
```

She then reads the text with the supplied words inserted into their respective places.

Write a program that acts as a Mad Libs leader. It should prompt the user for a file name, read the text from the file, find the tags for the missing words (anything within <...>), prompt the user for these words, and save them in an `ArrayList`. It should then reopen the file and display the completed text, inserting the saved words in place of the corresponding tags. A sample file, `madlibs.txt`, is provided in `JM\Ch15\Exercises`. ≤ Hint: you can read the text line by line and assume that tags are not split between lines, but it may be easier to read the text character by character. ≥

Graphics

16.1 Prologue

What you see on your computer screen is ultimately determined by the contents of the video memory (*VRAM*) on the graphics adapter card. The video memory represents a rectangular array of *pixels* (picture elements). Each pixel has a particular color, which can be represented as a mix of red, green, and blue components, each with its own intensity. A typical graphics adapter may use eight bits to represent each of the red, green, and blue intensities (in the range from 0 to 255), so each color is represented in 24 bits (that is, three bytes). This allows for $2^{24} = 16{,}777{,}216$ different colors. With a typical screen resolution of 1680 by 1050 pixels, your adapter needs $1680 \cdot 1050 \cdot 3$ bytes — a little over 5 MB — to hold the picture for one screen. The picture is produced by setting the color of each pixel in VRAM. The video hardware scans the whole video memory continuously and refreshes the image on the screen.

A graphics adapter is what we call a *raster* device: each individual pixel is changed separately from other pixels. (This is different from a *vector* device, such as a plotter, which can draw a line directly from point *A* to point *B*.) To draw a red line or a circle on a raster device, you need to set just the right group of pixels to the red color. That's where a graphics package can help: you certainly don't want to program all those routines for setting pixels yourself.

A typical graphics package has functions for setting drawing attributes, such as color, line style and width, fill texture or pattern for filled shapes, and font for text, and another set of functions for drawing simple shapes: lines, arcs, circles and ovals, rectangles, polygons, text, and so on. Java's graphics capabilities are based on the `Graphics` class and the more advanced `Graphics2D` class. The `Graphics` class is pretty rudimentary: it lets you set the color and font attributes and draw lines, arcs, ovals (including circles), rectangles, rectangles with rounded corners, polygons, polylines (open polygons), images, and text. There are "draw" and "fill" methods for each basic shape (for example, `drawRect` and `fillRect`).

The `Graphics2D` class is derived from `Graphics` and inherits all its methods. It works with a number of related interfaces and classes:

- The `Shape` interface and classes that implement it (`Line2D`, `Rectangle2D`, `Ellipse2D`, and so on) define different geometric shapes.
- The `Stroke` interface and one implementation of it, `BasicStroke`, represent in a very general manner the line width and style for drawing lines.

- The `Paint` interface and its implementations `Color`, `GradientPaint`, and `TexturePaint` represent a color, a color gradient (gradually changing color), and a texture for filling in shapes, respectively.

`Graphics2D` also adds methods for various coordinate transformations, including rotations.

More importantly, `Graphics2D` adds a capability for treating shapes polymorphically. In the `Graphics` class, contrary to the OOP spirit, shapes are not represented by objects, and there is a separate special method for drawing each shape. Suppose you are working on a drawing editor program that allows you to add different shapes to a picture. You keep all the shapes already added to the picture in some kind of a list. To redraw the picture you need to draw all the shapes from the list. With `Graphics` you have to store each shape's identification (such as "circle," "rectangle," etc.) together with its dimensions and position and use a `switch` or an `if-else` statement to call the appropriate drawing method for each shape.

With `Graphics2D`, you can define different shapes (objects of classes that implement the `Shape` interface) and store references to them in your list. Each shape provides a "path iterator" which generates a sequence of points on that shape's contour. These points are used by `Graphics2D`'s `draw(Shape s)` and `fill(Shape s)` methods. Thus, shapes are treated in a polymorphic manner and at the same time are drawn using the currently selected `Paint` and `Stroke`. If your own class implements `Shape` and supplies a `getPathIterator` method for it, then your "shapes" will be drawn properly, too, due to polymorphism.

Like any package with very general capabilities, `Graphics2D` and the interfaces and classes associated with it are not easy to use. We will stay mostly within the limits of the `Graphics` class, but, if you are adventurous, you can examine the `Graphics2D` API and learn to use some of its fancy features.

In the following sections we will examine Java's event-driven graphics model and review the basic drawing attributes and methods of the `Graphics` class. We will then use its capabilities to write a simple *Puzzle* program in which a player rearranges the pieces of a scrambled picture.

16.2 - 16.8

These sections are online at `http://www.skylit.com/javamethods4/`.

Exercises

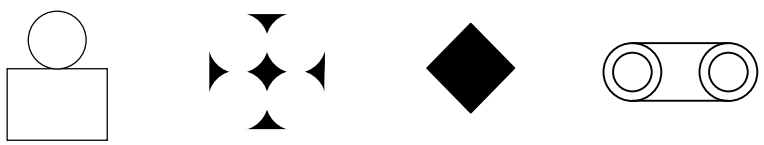
1. The program below (JM\Ch16\Exercises\Drawings.java) displays a red rectangle in the middle of the content pane:

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

public class Drawings extends JPanel
{
  public void paintComponent(Graphics g)
  {
    super.paintComponent(g);
    int w = getWidth();
    int h = getHeight();
    g.setColor(Color.RED);
    g.drawRect(w/4, h/4, w/2, h/2);
  }

  public static void main(String[] args)
  {
    JFrame window = new JFrame("Drawings");
    window.setBounds(100, 100, 300, 200);
    window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JPanel canvas = new Drawings();
    canvas.setBackground(Color.WHITE);
    window.getContentPane().add(canvas);
    window.setVisible(true);
  }
}
```

Add code to display a message inside the red rectangle. ✓

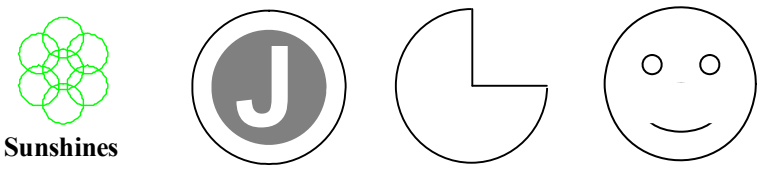
2. Modify the program in Question 1 to display the following designs:

(a) ✓ (b) (c) (d)



Continued

(e) (f) (g) ✓ (h)

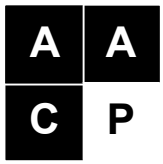


⦑ Hint: (b) is a black rounded rectangle with four white circles; (c) is a polygon. ⦒

Use any colors you want.

3. Define a class `Canvas` that extends `JPanel`. Make it show one of the designs from Question 2 (or your own design) by redefining its `paintComponent` method. Make your design centered approximately at the middle of the panel and make it scalable. Put the `main` method into a separate class.

4. (a) Adapt the program from Question 1 to show the logo of the American Association of Chomp Players:



(b)■ Now take a different approach to showing the same logo. Create a class `LetterPanel` derived from `JPanel`. Supply a constructor that takes a letter and a background color as parameters and saves them. Make `LetterPanel`'s `paintComponent` method draw the letter on the specified background. Set the 2 by 2 grid layout for the program's content pane as follows:

```
Container c = getContentPane();
c.setLayout(new GridLayout(2, 2, 1, 1));
```

Add four `LetterPanel` objects to it, one for each letter in the logo.

5.■ The program *Boxes* (`JM\Ch16\Exercises\Boxes.java`) can display different types of boxes (rectangles) in different colors. The user chooses the color of the box from a "combo box" that shows all 13 predefined colors from the `Color` class. The user sets the box type by checking option boxes for filled, rounded, and 3-D rectangles. The box type is represented as an integer in which individual bits, when set to 1, indicate the attributes: bit 0 for filled, bit 1 for rounded, and bit 2 for 3-D. For example, the value 3 (binary 011) indicates a filled rounded box with no 3-D effect.

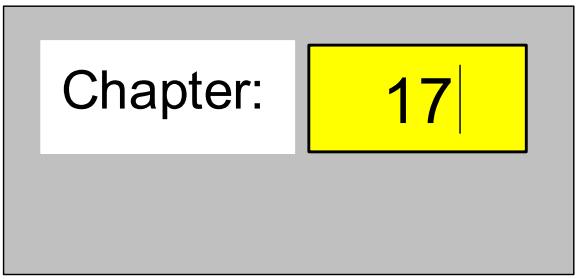
The *Boxes* program relies on a `BoxDrawer` class derived from `JPanel`. It has the additional methods `setBoxType(int boxType)`, `setBoxColor(Color color)`, and `drawBox(Graphics g)`. `BoxDrawer` also redefines the `paintComponent` method to repaint the background and call `drawBox`. Write the `BoxDrawer` class. Set the background to gray or black in `paintComponent` if the white color is chosen. Display an error message instead of a box if both rounded and 3-D attributes are selected. ≤ Hint: use a `switch` on the box type in `drawBox`. ≥

6. (a) Change the program from Question 4-b to display a checkerboard pattern with 8 rows and 8 columns. Using an 8 by 8 grid layout, add panels with alternating colors. ≤ Hint: see Question 15 in the exercises for Chapter 9. ≥

(b) Create a class `ChessSquarePanel` derived from `JPanel`. Supply a constructor that takes as parameters a background color and a `boolean` flag indicating whether the square is empty or holds a picture of a chess queen figure.

(c)■ Find on the web a description and a solution to the "Eight Queens" problem. Change the program to show the solution. ≤ Hint: create an 8 by 8 array of appropriately colored `ChessSquarePanel`s and set the `hasQueen` flags appropriately in eight of them to show the queens; set the content pane's layout to an 8 by 8 grid and add the panels to the grid. ≥

(d)◆ Provide a method to find your own solution to the "Eight Queens" problem. ≤ Hint: make it a recursive method `addQueens` with one parameter: a list of queens already placed. Start with an empty list. If the list has eight queens, return `true` (the problem is solved). If not, try to add one more queen to the list. If all such attempts fail, return `false`. ≥



GUI Components and Events

17.1 Prologue

In this chapter we discuss the basics of *graphical user interfaces* and *event-driven programming* in Java. Event-driven GUI is what made the OOP concept popular, and it remains the area where it's most relevant. While you can write console applications in Java, such programs won't take full advantage of Java or OOP; in most cases such programs could just as well be written in C or Python. The style of modern user interfaces — with many different types of control components such as menus, buttons, pull-down lists, checkboxes, radio buttons, and text edit fields — provides an arena where OOP and event-driven programming naturally excel.

Our task in this chapter and the following one is to organize more formally the bits and pieces of the *Swing* package and multimedia that you have managed to grasp from our examples so far. Overall, the Java API lists hundreds of classes and interfaces, with thousands of constructors and methods. A sizable portion of the API — more than 100 classes — deals with *Swing* GUI components, events handlers, and multimedia. The online documentation in HTML format gives you convenient access to these detailed specifications. Still, it is not very easy to find what you need unless you know exactly what to look for. In many cases it may be easier to look up an example of how a particular type of object is used than to read about it in the API spec file. In most cases you need only the most standard uses of classes and their methods, and there are thousands of examples of these in JDK demos, books, online tutorials, and other sources.

This is the approach our book has taken, too. While introducing Java's fundamental concepts and basic syntax and control structures, we have sneaked in a variety of commonly used GUI methods and "widgets." We have added some bells and whistles here and there, just so you could see, if you cared to, how such things might be done. This chapter summarizes what you have already seen and fills in some gaps. Appendix C presents a synopsis and an index of the more commonly used GUI components that appear in the code examples in this book.

Knowing all the details of the latest GUI package still does not guarantee that the GUI in your application will "work." In addition to working the way you, the programmer, intend, it must also be intuitive and convenient for the user. Designing a good user interface is a matter of experience, good sense, trial and error, paying attention to the software users, developing prototypes, and in some cases, relying on more formal "human factors" research. To become a good user interface designer, one should gain experience with a wide range of applications, observe what works

and what doesn't, and absorb the latest styles from cyberspace. Note that, strictly speaking, this skill may not be directly related to programming skills.

In this chapter we will discuss the "pluggable look and feel" and a few basic *Swing* components:

- `JLabel` — displays an icon or a line of text
- `JButton` — triggers an "action event" when pressed
- `JToggleButton` and `JCheckBox` — toggle an option
- `JComboBox` and `JRadioButton` — choose an option out of several possibilities
- `JSlider` — adjusts a setting
- `JTextField`, `JPasswordField`, and `JTextArea` — allow the user to enter and display or edit a line of text, a password, or a multi-line fragment of text, respectively
- `JMenuBar`, `JMenu`, `JMenuItem` — support pull-down menus

We will discuss some of the methods of these GUI objects and the events they generate. We will also get familiar with four layout managers that help to arrange GUI components on the application's window.

17.2 - 17.7

These sections are online at `http://www.skylit.com/javamethods4`.

Exercises

1. For each the following Swing GUI components in Column One, mark the listeners from Column Two and the "status accessors" from Column Three that are commonly used with them. ✓

(a)	`JPanel`	*< none >*	*< none >*
(b)	`JLabel`	`ActionListener`	`isSelected`
(c)	`JButton`	`ItemListener`	`getSelectedIndex`
(d)	`JCheckBox`	`ChangeListener`	`getSelectedItem`
(e)	`JRadioButton`		`getText`
(f)	`JComboBox`		`getValue`
(g)	`JTextField`		
(h)	`JSlider`		
(i)	`JMenuItem`		

2. Mark true or false and explain:

(a) An object's `actionPerformed` method can be called directly in the program by other methods. _____ ✓

(b) The same object can serve as an action listener for one component and an item listener for another component. _____ ✓

(c) A button can have several different action listeners attached to it. _____ ✓

(d) An object can serve as its own action listener. _____

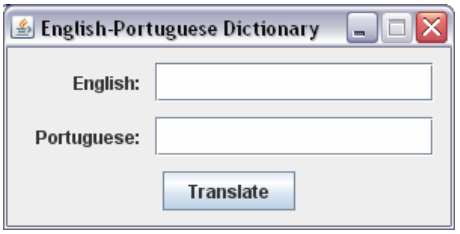
3.■ Write a program that serves as a GUI front end for the code you created for Question 7 in Chapter 9, the quadratic formula.

4.■ Write a GUI application that adds several "large integers" as described in Chapter 9, Question 25.

5. (a) In the `Boxes` program (`JM\Ch16\Exercises\Boxes.java`) from Question 5 in Chapter 16, add a menu bar with only one menu of two items: "Increase size" and "Decrease size." When clicked, these items should increase or decrease the current size of the box by 10%. Modify the `BoxDrawer` class (your solution to Question 5 in Chapter 16) to support this functionality.

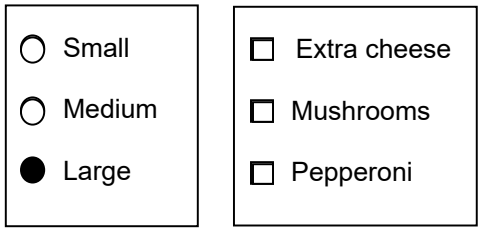
(b) Unite the `rounded` and `3-D` checkboxes in one group (a `ButtonGroup` object), so that they cannot be both selected at the same time. Usually such groups are used with radio buttons (see Appendix C), but they can work for checkboxes as well.

6.■ Create the following GUI layout:



ⵑ Hint: Put the two labels on a panel with a 2 by 1 grid layout. Use right-justified labels `JLabel(<text>, null, JLabel.RIGHT)`. (Instead of `null` you can put an icon showing the flag or an emblem of a country that speaks the language.) Put input fields on another 2 by 1 panel. Put both panels into a horizontal box with a strut between them; put this box and the "Translate" button into a vertical box with struts around them. There are other ways of doing it, of course. ⵒ

7.■ (a) Create the following GUI layout:



≤ Hint: put all the radio buttons into a panel and all the checkboxes into another panel, each with a 3 by 1 grid layout. Add a border to each panel by calling its `setBorder` method. For example:

```
import javax.swing.border.*;
< ... other statements >
    CompoundBorder border = new CompoundBorder(
        new LineBorder(Color.BLACK, 1),
        new EmptyBorder(6, 6, 6, 6));
    // outside border: black, 1 pixel thick
    // inside border: empty, 6 pixels on
    //   top, left, bottom, right

    panel.setBorder(border);
```

Put both panels into a horizontal box with struts between them and on the left and right sides. Unite the radio buttons into a group (see Appendix C). ≥ ✓

(b) Add a `JToggleButton` "To Go" on the right, initially set to "No." ✓

8.◆ Create a *Keypad* program. This program can serve as a basis for calculators of different models. First run a compiled version (`JM\Ch17\Exercises\keypad.jar`). The program consists of two classes: `Keypad` and `DigitalDisplay`. `Keypad` is the main class. Its constructor creates an object `display` of the `DigitalDisplay` type. Write a class `DigitalDisplay` by extending `JTextField`. Make `DigitalDisplay` implement `ActionListener` and make `display` serve as an action listener for all the buttons. The "C" button clears the display; any other button appends the text from the button to the current display text. For example:

```
JButton b = (JButton)e.getSource();
String str = b.getText();
...
```

Let `display`'s constructor configure the display: dark green background (for example, RGB 50, 200, 50), white or yellow foreground. Make it non-editable to prevent extraneous keyboard input. Set text alignment to the right:

```
setHorizontalAlignment(RIGHT);
```

Set the preferred size of the display in the program and add an empty border on the left and right to make it look better. For example:

```
import javax.swing.border.*;
...

    display.setPreferredSize(new Dimension(width, width/6));
    display.setBorder(new EmptyBorder(0, 10, 0, 10));
                                // top, left, bottom, right
```

Also set `display`'s font to a fairly large fixed-pitch font (in proportion to `width`). Is it a good idea to set display's size, border, and font in `DigitalDisplay`'s constructor? Explain.

In the `Keypad` class, place all the buttons on a 4 by 3 grid on a panel with some space between them. Add the `display` field above the buttons.
≤ Hint: place the display and the buttons panel into a vertical box. You can add the box directly to the content pane, or you can add it to a panel with a border, as we did. ≥

9.◆ Rewrite the *Madlibs* program described in Chapter 15 Question 10 as a GUI application. ⩽ Hint: use a `JTextArea` component to display the final text. ⩾

Chapter 18

Mouse, Keyboard, Sounds, and Images

18.1 Prologue

JVM (Java Virtual Machine) has a "virtual" mouse that rolls on an *x*-*y* plane and has up to three buttons. Mouse coordinates are actually the graphics coordinates of the mouse cursor; they are in pixels, relative to the upper-left corner of the component that registers mouse events. Mouse events can be captured by any object designated as a `MouseListener` (that is, any object of a class that implements the `MouseListener` interface).

Keyboard events can be captured by any object designated as a `KeyListener`. Handling keyboard events in an object-oriented application is complicated by the fact that a computer has only one keyboard and different objects need to listen to it at different times. There is a fairly complicated system of passing the *focus* (the primary responsibility for processing keyboard events) from one component to another and of passing keyboard events from nested components up to their "parents." Handling mouse events is easier than handling keyboard events because the concept of "focus" does not apply.

In this chapter we will discuss the technical details of handling the mouse and the keyboard in a Java GUI application. We will also learn how to load and play audio clips and how to display images and icons.

18.2 - 18.7

These sections are online at `http://www.skylit.com/javamethods4`.

Exercises

1. Write a *Four Seasons* program that changes the background color (from white to green to dark green to gold to white again, etc.) each time a mouse is clicked on it. ⦉ Hints: (1) Increment an index into an array of colors with wraparound. (2) Don't forget to repaint after each click. ⦊ ✓

2. Write a `DrawingPanel` class that extends `JPanel` and implements `MouseListener`. Redefine `paintComponent` to draw a filled circle or another shape. Make `mousePressed` change the shape's size or color when the mouse button is pressed anywhere on the panel and restore the shape when the button is released. Test your class with a simple program that creates one `DrawingPanel` object and makes it its own mouse listener. ✓

3. Adapt the `DrawingPanel` class from Question 2 to draw polygons. Each time the mouse is pressed, add the coordinates to a list of points. ⦉ Hints: (1) Use `mousePressed`, not `mouseClicked`, because a mouse "click" (in the Java definition) requires that pressed and released locations be the same, so "clicks" do not always register. (2) It is easier to hold the x- and y-coordinates in separate arrays for the sake of the `drawPolyline` method. (3) Don't forget to repaint after adding a point. ⦊

Consider a polygon finished when the number of points reaches some maximum number (for example, 10) or when it has three or more points and the last one is close to (for example, is within 10 pixels of) the first one. ⦉ Hint: `java.awt`'s `Point` class has a static method

```
static double distance(int x1, int y1, int x2, int y2)
```

that returns the distance between the points (x_1, y_1) and (x_2, y_2), and a method

```
static double distanceSq(int x1, int y1, int x2, int y2)
```

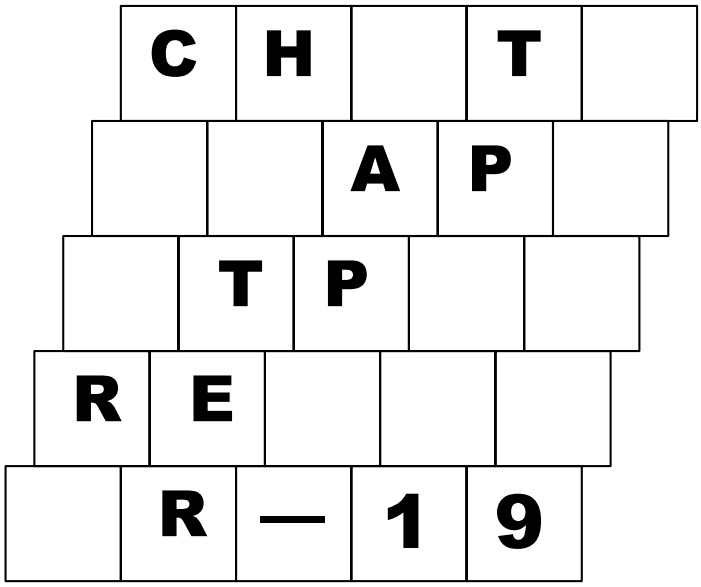
that returns the squared distance. The latter is more efficient because it does not need to calculate a square root. ⦊

Draw an unfinished polygon as a polyline and a finished one as a filled polygon. Once a polygon is finished, mark that state and zero out the point count before starting a new polygon.

4. Make the *Boxes* program from Question 5 in Chapter 17 respond to cursor keys: make the cursor keys move the box vertically or horizontally; make the cursor keys with the `Ctrl` key held down stretch or squeeze the box vertically or horizontally. Use `Boxes.java` and update the `BoxDrawer` class (from your solution to Question 5 in Chapter 17). ≤ Hints: make the `BoxDrawer` class implement `KeyListener`; and its object its own key listener; request focus in the `BoxDrawer`'s `paintComponent` method. ≥

5. Write a GUI program that displays the name of a key and its code in hex when that key is pressed. ≤ Hint: the `KeyEvent` class has a static method `getKeyText(int keyCode)` that returns a string that describes the key. ≥

6. Write a class `ImagePanel`, derived from `JPanel`, that displays a given image in the center. Provide a constructor that takes the image file name as a parameter. Write a simple GUI class that displays the image from `coin.gif` in a window.



Recursion Revisited

19.1 Prologue

In this chapter we continue the discussion of recursion started in Chapter 13. Recursion is a powerful tool for handling branching processes and nested structures. Take a Java list, for example. It can contain any type of objects as values, including other lists. In an extreme case, a list even can be its own element! Java's documentation says: "While it is permissible for lists to contain themselves as elements, extreme caution is advised..." It is no coincidence that recursive structures and processes are especially common in the computer world. It is easier to implement and use a structure or a process when its substructures and subprocesses have the same form. The same function, for example, can deal with a picture and its graphics elements that are also pictures, or a list and its elements that are also lists.

Recursion is not specific to Java: it works the same way with any language that allows functions (methods) to call themselves. In Java a method can call itself, or there can be a circular sequence of method calls. When amethod iscalled, all method parameters, the return address, and the local variables are kept in a separate frame on the *system stack*, so several methods, including several copies of the same method, can be waiting for control to be returned to them without any conflict. Multiple copies of the same method all share the same code but operate on different data.

In this chapter we will consider a few examples of recursion, discuss when the use of recursion is appropriate and when it is better to stay away from it, and learn how to debug and prove the correctness of recursive methods.

19.2 Three Examples

In Chapter 13, we discussed several recursive methods. Question 10 in exercises for that chapter, for example, asks for a recursive implementation of the `gcf` method that finds the greatest common factor of two positive integers. In that example, however, a simple non-recursive implementation exists as well. In our first example here, we take a situation where a recursive implementation is short, while a non-recursive implementation is not at all obvious.

Suppose we want to write a method that evaluates the "degree of separation" between two people in a given set of people. The degree of separation between *A* and *B* is defined as the number of links in the shortest chain of people connecting *A* and *B*, in which neighbors know each other. If *A* knows *B*, the degree of separation between *A*

and *B* is 1; if *A* knows *B* and *B* knows *C*, but *A* does not know *C*, the degree of separation between *A* and *C* is 2; and so on. Suppose a person is represented by an object of a class `Person`, which has a `boolean` method `knows`, so `Person p1` "knows" `Person p2` if and only if `p1.knows(p2)` returns `true`. We want to write the following method:

```
// Returns true if and only if the degree of separation between
// p1 and p2 in the set people is less than or equal to n
public boolean degreeOfSeparation(Set<Person> people,
                                  Person p1, Person p2, int n)
```

If $n = 1$, we only need to check whether `p1` and `p2` "know" each other. If $n > 1$ we can try to find a `Person` in the set that can serve as an intermediate link in a chain that connects `p1` and `p2`. It would be rather hard to implement this algorithm without recursion. But with recursion it is straightforward:

```
public boolean degreeOfSeparation(Set<Person> people,
                                  Person p1, Person p2, int n)
{
  if (n == 1)              // Base case
  {
    return p1.knows(p2);
  }
  else                     // Recursive case
  {
    for (Person p : people)
    {
      if (p1.knows(p) && degreeOfSeparation(people, p, p2, n-1))
        return true;
    }
    return false;
  }
}
```

As you know, a recursive method must have a base case (or several base cases), when no further recursion is necessary, and a recursive case (or several), where the same method is called recursively. For a recursive method to terminate properly, it must explicitly or implicitly handle the base case(s). When a method calls itself recursively, it must be applied to a similar but "smaller" task that eventually converges to one of the base cases; otherwise, the recursive calls will go deeper and deeper and eventually "blow the stack."

In our `degreeOfSeparation` method, the parameter *n* determines the "size" of the task. *n* equal to 1 is the base case. Since that parameter in the recursive call to `degreeOfSeparation` is reduced to `n-1`, the recursive calls eventually converge to the base case and recursion stops.

The above code is deceptively short, but it may become prohibitively time-consuming. For example, if the set is split into two groups of equal sizes such that everyone knows everyone else in each group but no one from the other group, and `p1` and `p2` belong to different groups, then the total number of calls to the `knows` method will be $\frac{3N^n - N^{n-1} - 2N}{N-1}$, where N is the number of people in each group. This is exponential growth, in terms of n. Also, each step deeper into recursion requires a separate frame on the system stack, so this method takes space on the system stack proportional to n.

There are actually more efficient algorithms for this task. Even a simple divide and conquer trick —

```
public boolean degreeOfSeparation(Set<Person> people,
                           Person p1, Person p2, int n)
{
  if (n == 1)              // Base case
  {
    return p1.knows(p2);
  }
  else if (n == 2)         // Another base case
  {
    for (Person p : people)
    {
      if (p1.knows(p) && p.knows(p2))
        return true;
    }
    return false;
  }
  else                     // Recursive case
  {
    int m = n/2;
    for (Person p : people)
    {
      if (degreeOfSeparation(people, p1, p, m) &&
                  degreeOfSeparation(people, p, p2, n-m))
        return true;
    }
    return false;
  }
}
```

— improves things. In this version `knows` will be called $\frac{n\left(3N^{k+1} - 2N^k - N\right)}{2\left(N-1\right)}$ times, where $n = 2^k \Rightarrow k = \log_2 n$, which is better than exponential growth in terms of n, and the amount of stack space needed is only proportional to $\log n$. There is a much

more efficient solution that takes time proportional to N^2, regardless of n (see Question 3 in the end-of chapter exercises).

In our second example, let us consider a method that implements "deep" traversal of a list that may contain strings and similar lists as elements. In effect, it is no longer a simple list, but a whole *tree*. For now, let's call such a list a "composite list" (Figure 19-1).

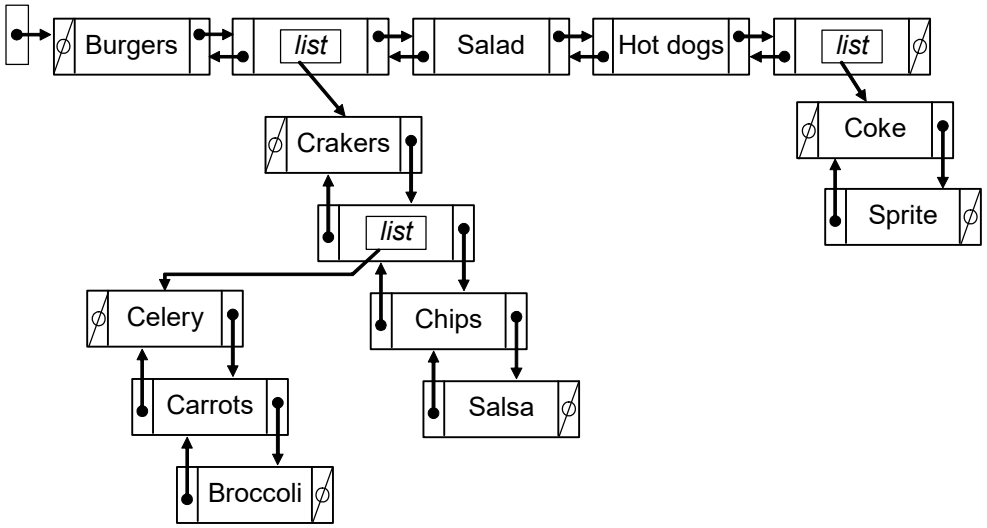


Figure 19-1. A "composite list": some elements hold strings, others hold lists

A more formal definition of a "composite list" is recursive: a "composite list" is a list where each element is either a string or a "composite list..." Suppose our task is to print all the strings in the list, including all the strings in all the lists that are its elements, and in all the lists that are elements of elements, and so on. Without recursion, this would be a little tricky. With recursion, it's just a few lines of code:

```
public void printAll(CompositeList list)
{
  String separator = "[";

  for (Object obj : list)
  {
    System.out.print(separator);

    if (obj instanceof String)
      System.out.print(obj);
    else  // if (obj instanceof CompositeList)
      printAll((CompositeList)obj);

    separator = ", ";
  }
  System.out.print("]");
}
```

Here the base case is when `list` contains only `String`s and no lists.

Given

```
CompositeList veggies = new CompositeList();
veggies.add ("Celery");
veggies.add ("Carrots");
veggies.add ("Broccoli");
CompositeList munchies = new CompositeList();
munchies.add ("Crackers");
munchies.add (veggies);
munchies.add ("Chips");
munchies.add ("Salsa");
CompositeList drinks = new CompositeList();
drinks.add ("Coke");
drinks.add ("Sprite");
CompositeList partyFood = new CompositeList();
partyFood.add("Burgers");
partyFood.add(munchies);
partyFood.add("Salad");
partyFood.add("Hot dogs");
partyFood.add(drinks);

printAll(partyFood);
```

the output is

```
[Burgers, [Crackers, [Celery, Carrots, Broccoli], Chips, Salsa], Salad, Hot
dogs, [Coke, Sprite]]
```

— a recursive method generates the output with nested lists.

Actually, it would be more conventional to define a `toString` method in the `CompositeList` class itself, so that we could write

```
System.out.println(list);
```

rather than

```
printAll(list);
```

Here is how such method might look, assuming that `CompositeList` extends `ArrayList<Object>`:

```
public String toString()
{
  String s = "", separator = "[";
  for (Object obj : this)
  {
    s += separator;
    if (obj instanceof String)
      s += obj.toString();
    else  // if (obj instanceof CompositeList)
      s += ((CompositeList)obj).toString();
    separator = ", ";
  }
  s += "]";
  return s;
}
```

Now note that all this "`instanceof`" type checking is totally redundant: polymorphism takes care of it. We can just write:

```
public String toString()
{
  String s = "", separator = "[";
  for (Object obj : this)
  {
    s += separator;
    s += obj;
    separator = ", ";
  }
  s += "]";
  return s;
}
```

And this is roughly how `toString` is defined in `ArrayList`.

Now the recursion is hidden, but it kicks in when one of the elements of a list happens to be a list. It turns out an `ArrayList<Object>` can handle a composite list without any extra code.

It is possible, of course, to write the above `toString` method with no recursion. But the code will be much longer, harder to understand, more error prone, and will not take advantage of polymorphism.

Do not avoid using recursion where it simplifies code without significant loss of efficiency.

In our third example, we will generate all possible permutations of a string of characters. Suppose we are building a computer word game that tries to make a valid word out of a given set of letters. The program will require a method that generates all permutations of the letters and matches them against a dictionary of words. Suppose a string of *n* characters is stored in a `StringBuffer`. (We use the `StringBuffer` class rather than `String` to make rearranging characters more efficient — `String`s are immutable objects.) Our strategy for generating all permutations is to place each character in turn in the last place in the string, then generate all permutations of the first (*n*–1) characters. In other words, `permutations` is a recursive method. The method takes two arguments: a `StringBuffer` object and the number *n* of characters in the leading fragment that have to be permuted.

In this example, the process is branching and recursive in nature although there are no nested structures. The base case is when *n* is equal to 1 — there is nothing to do except to report the permutation.

The `permutations` method below is quite short and readable; still, it is hard to grasp why it works! We will return to it in Section 19.4, which explains the best way of understanding and debugging recursive methods.

```
private void swap(StringBuffer str, int i, int j)
{
  char c1 = str.charAt(i); char c2 = str.charAt(j);
  str.setCharAt(i, c2); str.setCharAt(j, c1);
}
```

```
public void permutations(StringBuffer str, int n)
{
  if (n <= 1)                    // Base case --
    System.out.println(str);     //   the permutation is completed
  else                           // Recursive case
  {
    for (int i = 0; i < n; i++)
    {
      swap(str, i, n-1);
      permutations(str, n-1);
      swap(str, n-1, i);
    }
  }
}
```

19.3 When Not to Use Recursion

Any recursive method can be also implemented through iterations, using a stack if necessary. This raises a question: When is recursion appropriate, and when is it better avoided?

The most important rule is that recursion should be used only when it significantly simplifies the code without excessive performance loss. Recursion is especially useful for dealing with nested structures or branching processes. One typical example is algorithms for traversing tree structures. On the other hand, when you are dealing with linear structures and processes, normally you can use simple iterations. The following test will help you decide when to use recursion and when iterations. If the method branches in two or more directions, calling itself recursively in each branch, it is justified to use recursion. If the method calls itself only once, you can probably do the same thing just as easily with iterations.

There are also some technical considerations that may restrict the use of recursive methods:

1. If a method allocates large local arrays, the program may run out of memory in recursive calls. A programmer has to have a feel for how deeply the recursive calls go; she may choose to implement her own *stack* and save only the relevant variables there, reusing the same temporary array.
2. When a method manipulates an object's fields, a recursive call may change their values in an unpredictable way unless the manipulation is done on purpose and thoroughly understood.
3. If efficiency is important, a method implemented without recursion may work faster.

As an example, let us consider the `factorial(n)` method that calculates the product of all numbers from 1 to *n*. This method has a simple recursive form:

```
// Precondition: n >= 0
public long factorial(int n)
{
  if (n <= 1)      // Base case
    return 1;
  else             // Recursive case
    return n * factorial(n - 1);
}
```

Our test shows that `factorial`'s code has only one recursive call. We are dealing with a linear process. It should be just as easy to accomplish the same thing with iterations, thus avoiding the overhead of recursive method calls:

```
public long factorial(int n)
{
  long product = 1;
  for (int k = 2; k <= n; k++)
  {
     product *= k;
  }
  return product;
}
```

Both versions are acceptable. The recursive version takes amount of space proportional to *n*, but the factorial of large *n* is far too large, anyway.

A more pernicious example is offered by the famous Fibonacci numbers. These are defined as a sequence where the first two numbers are equal to one, with each consecutive number equal to the sum of the two preceding numbers:

1, 1, 2, 3, 5, 8, 13, ...

Mathematically this is a recursive definition:

$F_1 = 1;\ F_2 = 1;$
$F_n = F_{n-1} + F_{n-2}$, if $n > 2$.

It can be easily converted into a recursive method:

```
// Computes and returns the n-th Fibonacci number.
// Precondition: n >= 1
public long fibonacci(int n)
{
  if (n == 1 || n == 2)  // Base case
    return 1;
  else                   // Recursive case
    return fibonacci(n-1) + fibonacci(n-2);
}
```

It may seem, at first, that this method meets our test of having more than one recursive call to `fibonacci`. But in fact, there is no branching here: `fibonacci` simply recalls two previous members in the same linear sequence. Don't be misled by the innocent look of this code. The first term, `fibonacci(n-1)`, will recursively call `fibonacci(n-2)` and `fibonacci(n-3)`. The second term, `fibonacci(n-2)`, will call (again) `fibonacci(n-3)` and `fibonacci(n-4)`. The `fibonacci` calls will start multiplying like rabbits.* To calculate the *n*-th Fibonacci number, F_n, `fibonacci` will actually make more than F_n recursive calls, which, as we will see in the following section, may be quite a large number.

On the other hand, the same method implemented iteratively will need only *n* iterations:

```
public long fibonacci(int n)
{
  long f1 = 1, f2 = 1;
  while (n > 2)
  {
    long next = f1 + f2;
    f1 = f2;
    f2 = next;
    n--;
  }
  return f2;
}
```

* The numbers are named after Leonardo Pisano (Fibonacci) who invented the sequence in 1202, as part of an effort to develop a model for the growth of a population of rabbits.

For our final example of when recursion is inappropriate, let us consider Selection Sort of an array of *n* elements. As you know, the idea is to find the largest element and swap it with the last element, then apply the same method to the array of the first *n*–1 elements. This can be done recursively:

```
public void selectionSort(int[] a, int n)
{
  if (n == 1)     // Base case: array of length 1 -- nothing to do
      return;
  else
  {
    // Find the index of the largest element:
    int iMax = 0;
    for (int i = 1; i < n; i++)
      if (a[iMax] < a[i])
        iMax = i;

    // Swap it with the last element:
    int aTemp = a[n-1]; a[n-1] = a[iMax]; a[iMax] = aTemp;

    // Call selectionSort for the first n-1 elements:
    selectionSort(a, n-1);
  }
}
```

This is a case of so-called *tail recursion*, where the recursive call is the last statement in the method: only the return from the method is executed after that call. Therefore, by the time of the recursive call, the local variables (except the parameters passed to the recursive call) are no longer needed. A good optimizing compiler will detect this situation and, instead of calling `selectionSort` recursively, will update the parameter `n` and pass control back to the beginning of the method. Or, we can easily do it ourselves:

```
public void selectionSort(int a[], int n)
{
  while (n > 1)
  {
    // Find the index of the largest element:
    int iMax = 0;
    for (int i = 1; i < n; i++)
      if (a[iMax] < a[i])
        iMax = i;

    // Swap it with the last element:
    int aTemp = a[n-1]; a[n-1] = a[iMax]; a[iMax] = aTemp;

    n--;
  }
}
```

To quote Niklaus Wirth, the inventor of the Pascal programming language,

> In fact, the explanation of the concept of recursive algorithm by such inappropriate examples has been a chief cause of creating widespread apprehension and antipathy toward the use of recursion in programming, and of equating recursion with inefficiency.*

19.4 Understanding and Debugging Recursive Methods

A common way of understanding and debugging non-recursive methods is to trace, either mentally or with a debugger, the sequence of statements and method calls in the code. Programmers may also insert some debugging print statements that will report to them the method's progress and the intermediate values of variables.

These conventional methods are very hard to apply to recursive methods, because it is difficult to keep track of your current location in the hierarchy of recursive calls. Getting to the bottom of the recursive process requires a detailed examination of the system stack — a tedious and useless activity. Instead of such futile attempts, recursive methods can be more easily understood and analyzed with the help of a method known as *mathematical induction*.

In a nutshell, mathematical induction works as follows. Suppose we have a sequence of propositions (that is, statements that can be either true or false):

$$P_1, P_2, ..., P_n, ...$$

Suppose we can prove:

Base case: P_1 is true; and
Inductive step: for any $n \geq 2$, if $P_1, ..., P_{n-1}$ are true (*induction hypothesis*), then P_n is also true.

Then we can conclude that P_n is true for any $n \geq 1$.

This is so because P_1 implies P_2, P_1 and P_2 imply P_3, and so on. However, we do not have to go through the entire logical sequence for every step. Instead, we can

* Niklaus Wirth, *Algorithms + Data Structures = Programs*. Prentice Hall, 1976.

take a shortcut and just say that all propositions $P_1, P_2, ..., P_n, ...$ are true by mathematical induction.

As an exercise in mathematical induction, let us estimate the running time for the recursive `fibonacci` method discussed in the previous section:

```
public long fibonacci(int n)
{
  if (n == 1 || n == 2)  // Base case
    return 1;
  else                   // Recursive case
    return fibonacci(n-1) + fibonacci(n-2);
}
```

We will prove that `fibonacci(n)` executes not less than $(3/2)^{n-2}$ total calls to the `fibonacci` method.

Base cases: This is true for $n = 1$ and $n = 2$, because both require one call:

$$n = 1: \quad 1 \geq (3/2)^{1-2} = (3/2)^{-1} = 2/3$$
$$n = 2: \quad 1 \geq (3/2)^{2-2} = (3/2)^{0} = 1$$

Inductive step: For any n > 2, in addition to the initial call, the method calls `fibonacci(n-1)` and `fibonacci(n-2)`. From the induction hypothesis, the number of times `fibonacci` is called in `fibonacci(n-1)` is not less than $(3/2)^{n-3}$, and the number of times `fibonacci` is called in `fibonacci(n-2)` is not less than $(3/2)^{n-4}$. So the total number of `fibonacci` calls in `fibonacci(n)` is not less than:

$$1 + (3/2)^{n-3} + (3/2)^{n-4} > (3/2)^{n-3} + (3/2)^{n-4} = (3/2)^{n-4}(3/2 + 1) =$$
$$(3/2)^{n-4} \cdot (5/2) > (3/2)^{n-4} \cdot (9/4) = (3/2)^{n-4} \cdot (3/2)^2 = (3/2)^{n-2}$$

By mathematical induction, the number of calls to `fibonacci` in `fibonacci(n)` is not less than $(3/2)^{n-2}$, for any $n \geq 1$. Q.E.D.

Assuming that a reasonably fast computer can execute a hundred million calls per second, `fibonacci(100)` would run for over $(3/2)^{98}/10^8$ seconds, or more than 57 years! The iterative implementation, by contrast, would run in a small fraction of a second.

You may notice a close conceptual link between recursion and mathematical induction. The key feature of mathematical induction is that we do not have to trace the sequence of statements to the bottom. We just have to first prove the base case and then, for an arbitrary *n*, show that if the induction hypothesis is true at all previous levels, then it is also true at the *n*-th level.

Let us see how mathematical induction applies to the analysis of code in recursive methods. As an example, let's take the `permutations` method from Section 19.2, which generates all permutations of a string of characters:

```
public void permutations(StringBuffer str, int n)
{
  if (n <= 1)                   // Base case --
    System.out.println(str);    //   the permutation is completed
  else                          // Recursive case
  {
    for (int i = 0; i < n; i++)
    {
      swap(str, i, n-1);
      permutations(str, n-1);
      swap(str, n-1, i);
    }
  }
}
```

We will prove two facts about this code using mathematical induction:

1. `permutations` returns the string to its original order when it is finished.
2. `permutations(str, n)` generates all permutations of the first *n* characters.

In the base case, $n = 1$, the method just reports the string and does nothing else — so both statements are true. For the inductive step, let us assume that both statements are true for any level below *n* (induction hypothesis). Based on that assumption let us prove that both statements are also true at the level *n*.

In the recursive case, the method swaps `str[i]` and `str[n-1]`, then calls `permutations(str, n-1)`, then swaps back `str[n-1]` and `str[i]`. By the induction hypothesis, `permutations(str, n-1)` preserves the order of characters in `str`. The two swaps cancel each other. So the order of characters is not changed in `permutations(str, n)`. This proves Statement 1.

In the `for` loop we place every character of the string, in turn, at the end of the string. (This is true because the index `i` runs through all values from `0` to `n-1` and, as we have just shown, the order of characters does not change after each iteration through the loop.) With each character placed at the end of the string we call `permutations(str, n-1)`, which, by the induction hypothesis, generates all permutations of the first $n-1$ characters. Therefore, we combine each character placed at the end of the string with all permutations of the first $n-1$ characters, which generates all permutations of n characters. This proves Statement 2.

The above example demonstrates how mathematical induction helps us understand and, with almost mathematical rigor, prove the correctness of recursive methods. By comparison, conventional code tracing and debugging and attempts at unfolding recursive calls to the very bottom are seldom feasible or useful.

19.5 *Lab:* The Tower of Hanoi

The Tower of Hanoi puzzle is probably the most famous example of recursion in computer science courses. The puzzle has three pegs, with several disks on the first peg. The disks are arranged in order of decreasing diameter from the largest disk at the bottom to the smallest disk on top. The rules require that the disks be moved from peg to peg, one at a time, and that a larger disk never be placed on top of a smaller one. The objective is to move the whole tower from the first peg to the second peg.

The puzzle was invented by French mathematician François Edouard Anatole Lucas[lucas] and published in 1883. The "legend" that accompanied the game stated that in Benares, India, there was a temple with a dome that marked the center of the world. The Hindu priests in the temple moved golden disks between three diamond needles. God placed 64 gold disks on one needle at the time of creation and, according to the legend, the universe will come to an end when the priests have moved all 64 disks to another needle.

There are hundreds of apps on the Internet that move disks from peg to peg, either with animation or interactively.[hanoi] In this lab, fancy display is not required.

1. Write a program that solves the puzzle and prints out the required moves for a given number of disks. The output might look like this:

```
Enter the number of disks: 5
From 1 to 2
From 1 to 3
From 2 to 3
... (etc.)
```

2. Examine the number of moves required for 1, 2, 3, etc. disks, find the pattern, and come up with a formula for the minimum number of moves required for *n* disks. Prove the formula using the method of mathematical induction. Estimate how much time it will take your program to move a tower of 64 disks.

19.6 Case Study and Lab: the Game of Hex

The game of Hex was first invented in 1942 by Piet Hein, a Danish mathematician. (The same game was apparently reinvented independently a few years later by John Nash, then a graduate student at Princeton, who later won a Nobel prize in economics.) Martin Gardner made the game popular when he described it in his *Scientific American* article in the late 1950s and in a later book.

Hex is played on a rhombic board with hexagonal fields, like a honeycomb. A common board size is 11 by 11; Figure 19-2 shows a smaller 5 by 5 board. The game starts with an empty board. Each of the two players, *B* and *W*, is assigned a pair of opposite sides of the board. For example, *B* gets northwest and southeast, and *W* gets northeast and southwest. *B* has a pile of black stones and *W* has a pile of white stones. Players take turns placing a stone of their color on an empty field. A player who first connects his sides of the board with a contiguous chain of his stones wins.

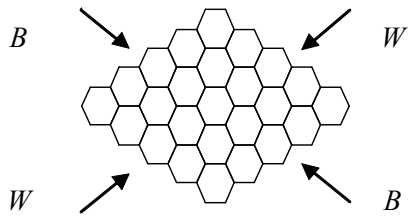


Figure 19-2. A 5 by 5 Hex board
(the hexagons at the four corners belong to both sides)

An interesting property of Hex is that a game can never end in a tie: when the board is filled, either *B* has his sides connected or *W* has his sides connected, always one or the other, but never both. Like all finite strategy games with no repeating positions, Hex has a winning strategy for one of the players. In this case it's the first player (it never hurts to have an extra stone of your color on the board). But this winning strategy is hard to find, because the number of all possible positions is large. For a smaller board, a computer can calculate it. It is not too difficult to write such a program, relying on a recursive algorithm. However (unless you have a lot of free time) in this lab our task is more modest: only to decide whether a given Hex position is a win for one of the players.

A human observer can glance at a Hex board and immediately tell whether one of the players has won. Not so in a computer program: it takes some computations to find out whether there is a chain of stones of a particular color connecting two opposite sides of the board. Our task is to develop an algorithm and write a method that does this.

But first we have to somehow represent a Hex board position in the computer. It is not very convenient to deal with rhombuses and hexagons in a program. Fortunately, an equivalent board configuration can be achieved on a regular square board, represented by a 2D array. Each inner field on a Hex board has six neighbors; border fields have four, and corner fields have two or three. We can emulate the same configuration on a square board using the appropriate designation of "logical" neighbors. Figure 19-3 shows which squares are supposed to be "neighbors": the rows of a 2D array are slightly shifted so that "neighbors" share a piece of border. Basically, a square at (*row*, *col*) has neighbors at (*row*−1, *col*−1), (*row*−1, *col*), (*row*, *col*−1), (*row*, *col+1*), (*row*+1, *col*), and (*row*+1, *col*+1), excluding those positions that fall outside the array.

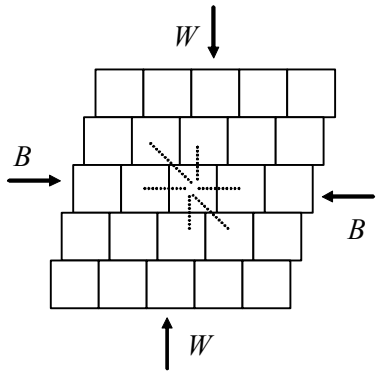


Figure 19-3. A Hex board represented as a 2D array

Write a class `HexBoard` that represents a board position. You can represent stones of different colors using `char`s, `int`s, `String`s, or `Color`s. If you use `char`s (for example, `'b'` for black, `'w'` for white, and space for an empty square), then you can conveniently derive your `HexBoard` class from the `CharMatrix` class that you wrote for the *Chomp* project in Chapter 9. To hide the implementation details, your `HexBoard` class should provide the modifiers `setBlack(row, col)` and `setWhite(row, col)` and the `boolean` accessors `isBlack(row, col)` and `isWhite(row, col)`. Also provide a `toString` method, which will allow us to print the board, each row on a separate line. It is convenient to use a private `boolean` method `isInBounds(row, col)` to determine whether (*row*, *col*) refers to a legal square on the board.

It is probably easier to write a separate method that detects a win for each of the players. You need to implement only one of them. For example:

```
/**
 * Returns true if there is a contiguous chain of black stones
 * that starts in col 0 and ends in the last column of the board;
 * otherwise returns false.
 */
public boolean blackHasWon()
```

At first, the task may appear trivial: just try starting at every black stone in the first column and follow a chain of black stones to the last column. But a closer look at the problem reveals that chains can branch in tricky ways (Figure 19-4), and there is no obvious way to trace all of them. What we really need is to find a "blob" of connected black stones that touches both the left and the right sides of the board.

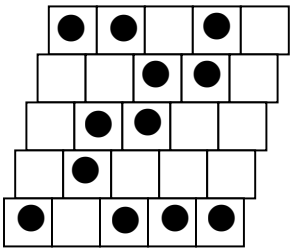


Figure 19-4. Hex: detecting a win for black stones

The task of finding a connected "blob" is known as an "area fill" task. A very similar situation occurs when you need to flood a connected area in a picture, replacing one color with another, or to fill the area inside a given contour (see Question 11 in the exercises for this chapter). The only difference is that usually a pixel or a cell has four neighbors (or eight, if you count the diagonal neighbors), while in Hex a square on the board has six neighbors.

Since the black and white colors are taken, we will be filling the area in "gray," so add private `setGray(row, col)` and `isGray(row, col)` methods to your `HexBoard` class.

1. Write a method `areaFill(int row, int col)`. This method should check whether the (*row*, *col*) square is in bounds and holds a black stone. If both conditions are true, `areaFill` should color in gray the largest contiguous black area that contains the (*row*, *col*) square. To do that, set the color at (*row*, *col*) to gray, then call `areaFill` recursively for each of its six neighbors. Make the `areaFill` method public, so that you can test it from `main`.
2. In the `blackHasWon` method:
   - call `areaFill` for each square in the first column;
   - count the gray stones in the last column to see if any of the gray blobs touch the last column;
   - Restore all gray stones on the board back to black and return `true` or `false`, appropriately.

❖ ❖ ❖

You might wonder whether there is a way to look for a win while doing `areaFill` and quit as soon as you reach the rightmost column. This is possible, of course. However, quitting a recursive method for good is not so easy: your original call may

be buried under a whole stack of recursive calls and you need to properly quit all of them. Two approaches are possible.

First approach: make `areaFill` return a `boolean` value: `true` if it `touched` the rightmost column, `false` if it didn't. Then, if one of the recursive calls to `areaFill` returns `true`, skip the rest of the recursive calls and immediately return `true`. Something like this:

```
if (areaFill(row-1, col-1) == true)
  return true;
if (areaFill(row-1, col) == true)
  return true;
...
```

Second approach: define a field in your class (a local variable won't do!) and set it to `true` as soon as `areaFill` touches the rightmost column. Something like:

```
if (col == numCols() - 1)
  won = true;
if (!won)
  areaFill(row-1, col-1);
if (!won)
  areaFill(row-1, col);
...
```

Note how treacherous recursion may be: if you write what seems to be less redundant code —

```
if (col == numCols() - 1)
  won = true;
if (!won)
{
  areaFill(row-1, col-1);
  areaFill(row-1, col);
  ...
}
...
```

— you will lose all the benefits of quitting early and your code will dramatically slow down!

"For extra credit," make `blackHasWon` work using one of the above two variations of a more efficient method that quits as soon as it detects a win. (Note that with this change, the `areaFill` method does not necessarily fill the area completely, so it might be a good idea to rename it into something more appropriate, say `findWin`.)

Test your class using a test program `Hex.java` and a sample data file `hex.dat` provided in `JM\Ch19\Hex`. You will also need the `CharMatrix` class, your solution to the lab in Section 9.5.

19.7 Summary

Recursion is a programming technique based on methods calling themselves.

Recursive method calls are supported by the system stack, which keeps the method parameters, return address, and local variables in a separate frame for each call.

Recursion is useful for dealing with nested structures or branching processes where it helps to create short, readable, and elegant code that would otherwise be impossible.

Recursion is not essential in situations that deal with linear structures or processes, which can be as easily and more efficiently implemented with iterations.

The best way to understand and analyze recursive methods is by thinking about them along the lines of *mathematical induction*; attempts at unfolding and tracing recursive code "to the bottom" usually fail, except in simple exercises.

Exercises

Sections 19.1-19.3

1. Fill in the blanks in the following recursive method:

```
// Returns the value of the largest element among
// the first n elements in vector v.
// Precondition: 1 <= n <= v.length
public double max(double[] v, int n)
{
  double m = v[n-1];

  if ( _________________________________ )
  {
    double m2 = ____________________________________ ;

    if (m2 > m)
      m = m2;
  }
  return m;
}
```

2. A positive integer is evenly divisible by 9 if and only if the sum of all its digits is divisible by 9. Suppose you have a method `sumDigits` that returns the sum of the digits of a positive integer *n* (see Chapter 13 Question 8). Fill in the blanks in the method `isDivisibleBy9(int n)` that returns `true` if `n` is divisible by 9 and `false` otherwise. Your method may use the assignment and relational operators and `sumDigits`, but no arithmetic operators (`*`, `/`, `%`, `*=`, `/=`, `%=`) are allowed. ✓

```
// Returns true if n is divisible by 9; otherwise returns false
// Precondition: n > 0
public boolean isDivisibleBy9(int n)
{
  if (_____________________________________)

    return _____________________________________ ;

  else if (_____________________________________)

    return _____________________________________ ;

  else

    return _____________________________________ ;
}
```

3.■ Rewrite the `degreeOfSeparation` method from Section 19.2 without recursion. ☹ Hint: start with a set that contains only `p1` and keep expanding it, adding on each iteration all the people who know anyone from that set. Iterate `n` times or until you find `p2` in the set of acquaintances. ☺.

4.■ (a) The method below attempts to calculate x^n economically:

```
public double pow(double x, int n)
{
  double y;
  if (n == 1)
    y = x;
  else
    y = pow(x, n/2) * pow(x, n - n/2);    // Line 7
  return y;
}
```

How many multiplications will be executed when `pow(1.234, 5)` is called? ✓

(b) How many multiplications will be executed if we replace the Line 7 above with the following block of statements?

```
{ y = pow(x, n/2); y *= y; if (n % 2 != 0) y *= x; }
```

(c) How many multiplications will Version (a) above take to calculate `pow(1.234, 9)`? Version (b)?

5.■ The method `doTheTrick` below can change the values in the first n elements in a list. If the `ArrayList list` has five elements: 10, 20, 30, 40, and 50 (in that order), what values will be stored in `list` after the call `doTheTrick(list, list.size())`?

```
public static void doTheTrick(ArrayList<Integer> list, int n)
{
  if (n < 2)
    return;
  if (n % 2 == 1)
    doTheTrick(list, n-1);
  else
  {
    doTheTrick(list, n-2);
    int temp = list.set(n-1, list.get(n-2));
    list.set(n-2, temp);
  }
}
```

Sections 19.4-19.7

6. What is the return value of `mysterySum(10)`, where

```
public int mysterySum(int n)
{
  if (n == 1)
    return 1;
  else
    return mysterySum(n - 1) + 2*n - 1;
}
```

Justify your answer by using mathematical induction. Explain why this is an inappropriate use of recursion. ✓

7.◆ In the `degreeOfSeparation` example in Section 19.2, prove the formula $\frac{3N^n - N^{n-1} - 2N}{N-1}$ for the number of calls to `knows` in the first version of the method. ⦇ Hint: use mathematical induction for n. ⦈ ✓

8. Write a recursive method

```
public String removeConsecutiveDuplicateChars(String str)
```

that takes a string and returns a new string with all duplicate consecutive occurrences of the same character removed. For example, `removeConsecutiveDuplicateChars("ABBCDDDEEFGG")` should return `"ABCDEFG"`. Do not use any iterative statements.

9.■ Fill in the blanks in the following method:

```
// Returns the number of all possible paths from the
// point(0, 0) to the point(x, y), where x and y are
// any non-negative integers.  From any point the path
// may extend only down or to the right by one unit
// (that is, one of the current coordinates x or y can be
// incremented by one).
public long countPaths(int x, int y)
{
  if (x == 0 || y == 0)
    return _________________________________;
  else
    return ____________________________________________ ;
}
```

10.◆ Write a recursive method `optimalPath` and use it in a program in which Cookie Monster finds the optimal path from the upper-left corner (`0`, `0`) to the lower-right corner (`SIZE-1`, `SIZE-1`) in a cookie grid (a 2D array). Each element of the grid contains either some number of cookies (a non-negative number) or a barrel (`-1`). On each step Cookie Monster can only go down or to the right. He is not allowed to step on barrels. The optimal path contains the largest number of cookies.

The program reads the cookie grid from a file and reports the number of cookies on the optimal path. (The path itself is not reported.) A sample data file is provided in `JM\Ch19\Exercises\cookies.txt`.

For recursive handling, it often helps to restate the question in more general terms. Here we need to refer to the optimal path from (0, 0) to any position (*row*, *col*). So our `optimalPath` method should take two parameters: `row` and `col`. Since Cookie Monster can only move down and to the right, the maximum number of cookies accumulated at a position (*row*, *col*) is related to the previous positions as follows:

```
optimalPath(row, col) = cookies[row][col] +
the larger of the two:
 {optimalPath(row-1, col),optimalPath(row, col-1)}
```

The only problem is invalid positions: either out of bounds or "barrels." How can we define `optimalPath` for an invalid position (*row*, *col*), so that the above formula still works? Identify the base case(s) and recursive case(s).

11.■ A "pool" is an irregularly shaped contiguous blob of cells with `Color.BLACK` values in a 2D array. (Two cells in a 2D array are considered neighbors if they share a side, so each cell, except the cells on the border, has four neighbors.) The pool is completely surrounded by a "white wall," a contour of cells with `Color.WHITE` values. Write a class `AreaFill` with one public static method

```
public static void fillPool(Color[][] plane, int row,
                                    int col, Color color)
```

that takes a location `(row, col)` inside a pool in `plane` and fills all the cells in that pool with `color`. Write a GUI program with mouse input to test your method (or, if you skipped Chapters 17 and 18, use the `ArearFillTest` class in the `JM\Ch19\Exercises` folder).

12.■ On Babbah Island, the alphabet consists of three letters, A, B, and H — but no word may have two A's in a row. Fill in the blanks in the following recursive method `allWords`, which prints out all Babbah Island words of the given length:

```
/**
 * Prints all Babbah Island words that have
 * n letters.  word contains the initial
 * sequence of letters in a word that is being built
 * (use an empty string buffer of capacity n
 * when calling allWords from main).
 */
public void allWords(StringBuffer word, int n)
{
  if (n == word.length())  // base case
  {
    // Display the string:

    ____________________________________;
  }
  else  // recursive case
  {
    int k = word.length();
    word.append('*'); // reserve room for one char

    // Append 'A' only if last letter is not an 'A':
    if (k == 0 || word.charAt(k-1) != 'A')
    {
      word.setCharAt(k, 'A');

      ____________________________________;
    }
    ____________________________________; // append 'B'

    ____________________________________;

    ____________________________________; // append 'H'

    ____________________________________;

    ____________________________________;
  }
}
```

≤ Hint: you might need `StringBuffer`'s `setLength` method, which truncates the string in the buffer to a specified length. ≥

13.◆ Suppose we have a list of positive integers. We want to choose several of them so that their sum is as large as possible but does not exceed a given limit. This type of problem is called a "Knapsack Problem." For example, we may want to choose several watermelons at the market so that their total weight is as large as possible but does not exceed the airline limit for one bag.

(a) Write a recursive method that solves a simplified Knapsack Problem: it only calculates the optimal sum but does not report the selected items:

```
/**
 * w contains n positive integers (n <= w.length).
 * Returns the sum of some of these integers such that
 * it has the largest possible value without exceeding
 * limit.
 */
public int knapsackSum(int[] w, int n, int limit)
```

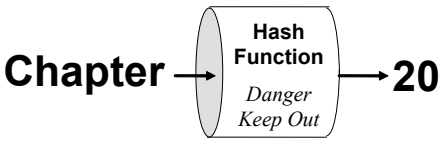
Use mathematical induction to prove that your code is correct.

(b) Write a more complete version —

```
public int knapsackSum(int[] w, int n, int limit,
                                    ArrayList<Integer> list)
```

— that in addition builds a list of the values selected for the optimal sum. `list` is initially empty when the method is called from `main` (or from another method).

(c)◆ Can you think of an alternative algorithm that uses neither recursion nor a stack?



Sets and Maps

20.1 Prologue

In mathematics, a *set* is a collection that has no duplicate elements. All positive integers, all points on the *x-y* plane, all students enrolled in a particular course, and all chapters in this book are examples of sets. The concept is so general that one might wonder why we need it. But it serves as a foundation in mathematics.

In math, a *mapping* is basically the same as a *function*: it is a kind of relation between two sets in which each element of the first set is mapped onto one and only one element of the second set. In the software context, we say that a set of *keys* is mapped onto a set of *values* (Figure 20-1).

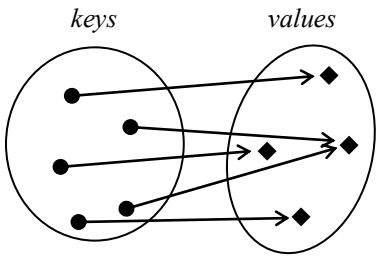


Figure 20-1. A mapping from a set of keys onto a set of values

Maps are very common: account numbers can be mapped onto customers, usernames can be mapped onto passwords, people can be mapped onto their birthdays, index entries can be mapped onto lists of page numbers, and so on.

Sets and maps can be implemented very efficiently. For example, when you log in to a web site, the software has to quickly match your user ID against a large database of valid subscribers. The IDs of subscribers can be held in a set. When you enter your password, it has to be verified against the one on record. The passwords for valid subscribers can be kept in a map. Here a user ID is the key and the associated password is the value.

The Java Collections Framework defines the `Set` and `Map` interfaces in the `java.util` package, and two implementations of each of them: `TreesSet` and `HashSet` and `TreeMap` and `HashMap`.

The `TreeSet` and `TreeMap` implementations use Binary Search Trees. We leave these to you to explore on your own. In the rest of this chapter we explain the hashing technique and the constructors and methods of `java.util.HashSet` and `java.util.HashMap`.

20.2 Lookup Tables

A lookup table implements a map. The idea is to avoid searching altogether: each key in a map tells us right away where we can find the value associated with it. A lookup table is simply an array that holds the values. The key is converted either directly or through some simple formula into an integer, which is used as an index into the lookup table array, and the associated value is stored in the element of the array with that index. One special reserved value (for example, `null`) may be used to indicate that a particular slot in the table is empty — the key is not used. The indices computed for two different keys must be different, so that we can go directly to the corresponding lookup table entry and store or fetch the data.

Suppose, for example, that an application such as entering shipping orders requires a database of U.S. zip codes that would quickly find the town or locality with a given code. Suppose we are dealing with 5-digit zip codes, so there are no more than 100,000 possible values, from 00000 to 99999. Actually, only a fraction of the 5-digit numbers represent real zip codes used by the U.S. Postal Service. But in this application it may be important to make the zip code lookup as quick as possible. This can be accomplished by using a table with 100,000 entries. The 5-digit zip will be used directly as an index into the table. Those entries in the table that correspond to a valid zip code will point to the corresponding record containing the locality name; all the other entries will remain unused (Figure 20-2).

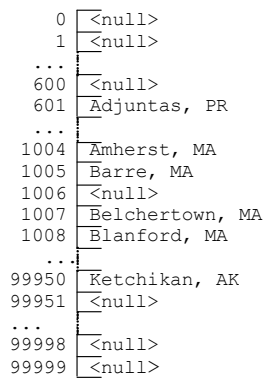


Figure 20-2. A lookup table for zip codes in the United States

Lookup tables are useful for many other tasks, such as data compression or translating one symbolic notation into another. In graphics applications and in hardware, for example, a "logical" color code (usually some number, say, from 0 to 255) can be converted into an actual screen color by fetching its red, green, and blue components from three lookup tables.

Another common use of lookup tables is for tabulating functions when we need to speed up time-critical computations. The function argument is translated into an integer index that is used to fetch the function value from its lookup table. In some cases, when the function argument may have only a small number of integer values, the lookup table may actually take less space than the code that would be needed to implement the function! If, for example, we need to compute 3^n repeatedly for $n = 0,...,9$, the most efficient way, in terms of both time and space, is to use a lookup table of 10 values:

```
private static int[] n_thPowerOf3 =
      {1, 3, 9, 27, 81, 243, 729, 2187, 6561, 19683};
...

  // Precondition: 0 <= n < 10
  public int powOf3(int n)
  {
    return n_thPowerOf3[n];
  }
```

In another example, an imaging application may need to count quickly the number of "black" pixels (picture elements) in a scan line (for instance, in order to locate lines of text). In a large black and white image, pixels may be packed eight per byte to save space. The task then needs a method that finds the number of bits in a byte that are set to 1. This method can easily do the job by testing individual bits (using the bit-wise AND operator and hex constants 0x01, 0x02, 0x04, etc. for integers that have a single bit set — see Chapter 18):

```
// Count the number of bits in byte b that are set to 1:
int count = 0;
if ((b & 0x01) != 0) count++;  // bit 0
if ((b & 0x02) != 0) count++;  // bit 1
if ((b & 0x04) != 0) count++;  // bit 2
...
if ((b & 0x80) != 0) count++;  // bit 7
```

But, if performance is important, a lookup table with 256 elements that holds the bit counts for all possible values of a byte (0–255) may be a more efficient solution:

```
private static int[] bitCounts =
{
  0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4,    // 00000000 - 00001111
  1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,    // 00010000 - 00011111
  ...                                          ...
  4,5,5,6,5,6,6,7,5,6,6,7,6,7,7,8     // 11110000 - 11111111
};

  ...
  int i = (int)b;
  if (i < 0)        // In Java, byte-type values are signed:
    i += 128;       //   -128 <= b <= 127; we need 0 <= i <= 255
  int count = bitCounts[i];
```

20.3 *Lab:* Cryptogram Solver

Almost everyone is sooner or later tempted to either send a message in code or to decode an encrypted message. In a simple cryptogram puzzle, a small fragment of text is encrypted with a substitution cipher in which each letter is substituted with another letter. Something like: "Puffm, Cmafx!" To solve a cryptogram we usually look for familiar patterns of letters, especially in short words. We also evaluate frequencies of letters, guessing that in English the most frequent letters stand for 'e' or 't', while the least frequent letters stand for 'j' or 'q'. The purpose of this lab is to solve a cryptogram and to master the use of lookup tables in the process.

Our *Cryptogram Solver* program is interactive. After opening an encrypted text file, the user sees the encrypted text on the left side of the screen and the decoded text on the right. Initially, nothing is decoded — the decoded text has only dashes for all the letters. The user then enters substitutions for one or several letters, clicks the "Refresh" button, and immediately sees an updated version of the decoded text. In addition, the program can offer decoding hints based on the frequencies of letters in the encrypted text.

Cryptogram Solver can also create cryptograms. If you enter random substitutions for all letters (or click the "Encode" menu item) and then enter your text fragment on the left side (by typing it in, loading it from a text file, or cutting and pasting it from another program), then the text shown on the right side will be an encrypted version of your text.

Your task is to write a class `Enigma`, named after the famous German encryption machine.[enigma] Enigma was invented in 1918 and was first used in the banking business, but it very quickly found its way into the military. Enigma's codes were considered "unbreakable," but they were eventually broken, first by Polish codebreakers in 1932 and later, during WWII, by the codebreakers of the Bletchley Park project in Great Britain, led by Alan Turing,[turing] one of the founders of modern computer science. The battle between Enigma codemakers and codebreakers lasted through WWII, and the dramatic successes of Allied cryptanalysts provided invaluable intelligence information.

Your `Enigma` class should maintain a lookup table for substitutions for the letters 'A' through 'Z'. For example:

```
private char[] lookupTable;
```

Initially the lookup table contains only dashes. As decryption proceeds, the table is updated and gradually filled. You can change your guess for a letter as often as you want.

The `getNumericValue(char ch)` static method of the `Character` class returns consecutive integers for letters 'A' through 'Z' (10 for 'A', 11 for 'B', and so on, 35 for 'Z'; 0 though 9 are returned for characters '0' through '9'). Therefore, if `ch` is an upper case letter, then

```
int i = Character.getNumericValue(ch) -
                              Character.getNumericValue('A');
```

sets `i` to an integer in the range from 0 to 25, which can be used as an index into our lookup table.

Your `Enigma` class should define a constructor with one `int` parameter — the number of letters in the alphabet — and three public methods:

```
void setSubstitution(int i, char ch);
String decode(String text);
String getHints(String text, String lettersByFrequency);
```

The first two methods support decoding (or encoding) of text; the last one supports computer-generated hints based on letter counts.

The `setSubstitution(int i, char ch)` method should set the *i*-th element of the lookup table to `ch`.

The `decode(String text)` method decodes all the letters in `text` according to the current lookup table. `decode` should leave all characters that are not letters unchanged and preserve the upper or lower case of letters. It should return the decoded string, which has the same length as `text`.

The `getHints(String text, String lettersByFrequency)` method returns computer-generated hints for each letter in the encrypted text. It works as follows. First it counts the number of occurrences for each of the letters 'a' through 'z' in `text` (case blind) and saves these 26 counts in an array. Write a separate private method for that:

```
private int[] countLetters(String text)
```

You should count all letters in one sweep over `text`. Start with zeroes in all counts, then increment the appropriate count for each letter.

After getting the counts for all letters, `getHints` creates and returns a `String` of "hints." The returned string `hints` should hold a permutation of letters 'A' through 'Z'; `hint.charAt(k)` will be displayed as a computer-generated hint for decoding the *k*-th letter of the alphabet. The hints should be based on comparing the order of letters by frequency in letter counts in encrypted text with the order of letters by frequency in plain (unencrypted) text. The `lettersByFrequency` parameter contains the letters 'A' through 'Z' arranged in increasing order of their frequencies in a sample text file. Suppose `lettersByFrequency`, passed to `getHints` is `"JQXZKBVWFUYMPGCLSDHROANITE"`. Then 'J' should be the hint for the least frequent letter in `text`, 'Q' for the second least frequent letter, and so on. The method's code is quite short once you figure out the algorithm. Try to figure it out yourself or read the (encrypted) hint below.

Kwy ep efhjdxqct wxtxfed qj "wjdqxph iv ajkpqxph," sywadxiys xp Aceuqyd 18. Ojd yeac yfytypq `ajkpqw[x]` xp qcy fyqqyd ajkpqw eddev oxps qcy pktiyd jo yfytypqw `ajkpqw[r]` wkac qceq `ajkpqw[r] < ajkpqw[x]` jd `ajkpqw[r] == ajkpqw[x]` eps `r < x`. Qcxw pktiyd (aeff xq depz) xw qcy depz jo qcy *x*-qc fyqqyd jo qcy efuceiyq xp qydtw jo xqw odynkypav xp qcy ypadvuqys qygq. Wj `cxpqw[x]` wcjkfs iy wyq qj `fyqqydwIvOdynkypav.acedEq(depz).`

(The above paragraph is also available in `JM\Ch20\Cryptogram\hint.txt` file. You can use *Cryptogram Solver* on it even before you get the hints part working correctly. Just use a "stub" method for `getHints` that returns an arbitrary string of 26 characters.)

We generated our `lettersByFrequency` string by counting occurrences for the 26 letters in the file `sample.txt`. Therefore, if you load `sample.txt` (a plain text file) into the program, the hint displayed for each letter should be that same letter. This is an easy way to test your `countLetters` and `getHints` methods.

Combine your `Enigma` class with the `Cryptogram` and `CryptogramMenu` classes located in `JM\Ch20\Cryptogram`. Test your program with `sample.txt`, then try to decode `secret.txt`. Both these files are in `JM\Ch20\Cryptogram`, too.

Unfortunately, as you can see, our computer-generated hints turn out to be entirely useless, except for the most frequent letter 'e'. Apparently we need a more sophisticated tool for solving cryptograms automatically — perhaps counting 2-D distributions for all pairs of adjacent letters, or even 3-D distributions for all triplets of letters, or a way to look for other patterns in the text.

20.4 Hash Tables

The technique of *hashing* builds on the lookup table concept. In a lookup table, a key is either used directly or converted through a very simple formula into an integer index. Different keys translate into different indices in the lookup table. This method is not practical, however, when the range of possible key values is large. It is also wasteful when the mapping from keys to integer indices is very sparse — many lookup table entries remain unused.

We can avoid these problems by using a tighter system of mapping from keys to integer indices in the table. We can try to map all possible key values into a narrower range of indices and to cover that range more uniformly. Such a transformation is called a *hash function*; a table used with it is a *hash table*.

The price of hashing is that we lose the one-to-one correspondence between keys and table entries: two different keys may be mapped into the same location in the hash table. Thus when we try inserting a new value into the table, its slot may already be occupied. These situations are called *collisions*. We have to devise some method of dealing with them. When we retrieve a value, we have to verify that its key indeed matches the target; therefore, the key must be explicitly stored in the table with the rest of the record.

The design of a hash table thus hinges upon successful handling of two problems: how to choose a good hash function and how to handle collisions. There is room for ingenious solutions for both.

A good hash function must have the following properties:

1. It must be easy to compute.
2. It must map all possible values of keys onto a range that is not too large.
3. It must cover that range uniformly and minimize collisions.

To devise such a function, we can try some "random" things akin to the transformations used for generating random numbers in a specified range. If the key is a string of characters, we can use some numeric codes (for instance, Unicode) for them. We can then chop the key into pieces and combine these together using bit-wise or arithmetic operations — hence the term "hashing." The result must be an integer in the range from 0 to `tableSize-1`.

Overly simplistic hash functions that simply truncate the key or take it modulo the table size —

```
public int hashCode(long key) { return (int)(key % tableSize); }
```

— may create unexpected clusters of collisions. Fortunately, we can evaluate our hash function on some simulated and real data before using it in an application.

One approach to handling collisions is to implement each entry in the hash table as a structure that can itself hold more than one value. This approach is called *chaining*, and the table entry is referred to as a *bucket*. A bucket may be implemented as a sorted array, a linked list, or even a *binary search tree* (Figure 20-3). This approach works well for densely populated hash tables.

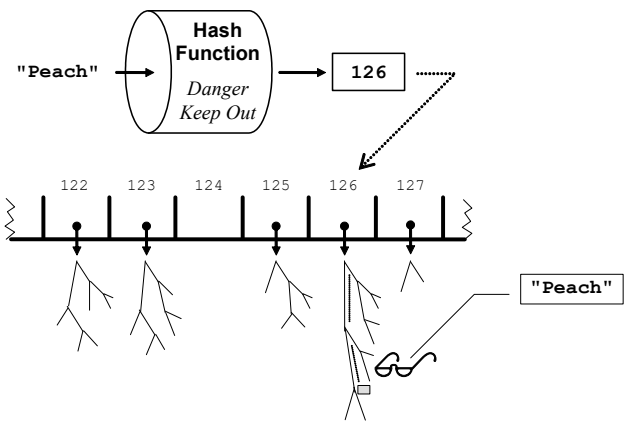


Figure 20-3. Resolving collisions in a hash table by chaining

As we can see, the performance of a search in a hash table varies greatly with the details of implementation. With many collisions, the performance may deteriorate. The ratio of the number of items stored in a hash table to the number of buckets is called the hash table's *load factor*. If a load factor is too small, a lot of space is wasted. When the load factor becomes very high, all the advantages of hashing are lost. Then it may be time to rehash all the items into a new table with a larger number of buckets. A reasonable load factor may range from 0.5 to 2.0.

20.5 `java.util`'s `HashSet` and `HashMap`

`java.util`'s `HashSet` and `HashMap` classes use hashing and resolve collisions through chaining.

`HashSet`'s no-args constructor creates an empty hash table with a default initial capacity (number of buckets) of 16 and a load factor limit of 0.75. If you know in advance the maximum number of entries to be stored in the table, it is better to use another constructor, `HashSet(int initialCapacity)`. Otherwise, when the table becomes too crowded, `HashSet`'s `add` method has to allocate a larger array and "rehash" all the elements into it. `initialCapacity` is the number of buckets to be used in the table; it should be roughly twice the number of entries expected to be stored. The `HashMap` class provides similar constructors.

When using `HashSet` or `HashMap`, you have to provide a reasonable hash function for the objects in the set or keys in the map.

When an object is added to a `HashSet` or serves as a key in a `HashMap`, that object's `hashCode` method is called. `hashCode` is a method of the `Object` class, so every object has it. Unfortunately, this method is hardly usable because it is based on the object's current address in memory and does not reflect the object's properties. Programmers usually override `Object`'s `hashCode` in their classes.

It is important to override `Object`'s `hashCode` method with a `hashCode` method appropriate for your class if you plan to store the objects of your class in a `HashSet` or use them as keys in a `HashMap`.

Java library classes such as `String`, `Double`, and `Integer` have suitable `hashCode` methods defined for their objects. For example, `String`'s `hashCode` method computes the hash code as $s_0 \cdot 31^{n-1} + s_1 \cdot 31^{n-2} + \ldots + s_{n-1}$, where s_i is Unicode for the *i*-th character in the string. (This computation is performed using integer arithmetic, ignoring overflows; `hashCode` returns 0 for an empty string.)

Normally, you do not have to invent your own hash code formula for your class from scratch but can rely on the library `hashCode` methods available for one or several fields of your class.

For example:

```
public class MsgUser
{
  private String screenName;
  ...
  public int hashCode()
  {
    return screenName.hashCode();
  }
}
```

No matter how simple it is, it is important not to forget to supply a `hashCode` if you plan to use your objects in a `HashSet` or a `HashMap`.

`HashSet` and `HashMap` also use the `equals` method when they look for a target within a bucket.

To successfully store and retrieve objects from a `HashSet`, or to use them as keys in a `HashMap`, the objects must have the `hashCode` method and the `equals` method. The `hashCode` method must agree with `equals`: if `x.equals(y)` is true, `x.hashCode() == y.hashCode()` must also be true.

You don't have to worry about the range of values returned by a `hashCode` method.

The `hashCode` method returns an integer from the whole integer range. The methods of `HashSet` and `HashMap` further map the hash code onto the range of valid table indices for a particular table (by taking it modulo table size).

❖ ❖ ❖

In the Java Collections Framework, a set is represented by the interface `Set<E>`, where `E` is the type of elements in the set. `HashSet<E>` implements `Set<E>`. A map is represented by the interface `Map<K, V>`, where *K* is the type of key objects and *V* is the type of value objects. `HashMap<K, V>` implements `Map<K, V>`. `Set`, `Map`, `HashSet`, and `HashMap` are part of the `java.util` package.

`HashSet`'s no-args constructor creates and empty set. Some of `HashSet`'s methods are shown in Figure 20-4.

```
boolean isEmpty();
int size();
boolean contains(Object obj);
boolean add(E obj);
boolean remove(Object obj);
```

Figure 20-4. Some of the methods of `java.util.Set<E>` interface and the `HashSet<E>` class

`HashMap`'s no-args constructor creates an empty map, with an empty set of keys and an empty set of values. Some of `HashMap`'s methods are shown in Figure 20-5.

```
boolean isEmpty();                 // Returns true if this map is
                                   //   empty; otherwise returns false.
int size();                        // Returns the number of key-value pairs
                                   //   in this map.
V get(Object key);                 // Returns the value associated with key.
V put(K key, V value);             // Associates value with key; returns the
                                   //   value previously associated with key
                                   //   (or null if there was none).
V remove(Object key);              // Removes the key-value pair from this
                                   //   map; returns the value previously
                                   //   associated with key.
boolean containsKey(Object key); // Returns true if key is in the set of
                                   //   keys; otherwise returns false.
Set<K> keySet();                   // Returns the set of all the keys in
                                   //   this map.
```

Figure 20-5. Some of the methods of `java.util.Map<K,V>` interface and the `HashMap<K,V>` class

The `get`, `put`, and `containsKey` methods work "instantaneously" in a `HashMap`, provided the number of collisions is small.

If truth be told, the `HashMap` implementation does not use separate sets for keys and for values. It uses a hash table of entries in which each entry holds a (*key*, *value*) pair. The `keySet` method returns a `Set` "view" of the keys, which is implemented as an inner class in `HashMap`. Any changes to the set of keys will affect the map.

To avoid duplication of code, Java developers actually implemented the `HashSet` class as a special case of `HashMap`, in which all keys are mapped onto the same object. `HashSet` has a field called `map`, and a `HashSet`'s methods are channeled through that field. For instance, `contains(obj)` is coded as `return map.containsKey(obj)`. This technical detail has no effect on your use of these classes.

20.6 *Lab:* Search Engine

In this lab we'll design and program our own miniature search engine, which we'll call "Giggle." Rather than searching the Internet for keywords or phrases, *Giggle* searches a single file for all lines of text that contain a given word. *Giggle*'s code uses lists and hash tables. More precisely, it takes advantage of `java.util`'s `List`, `Set`, and `Map` interfaces and `ArrayList`, `HashSet`, and `HashMap` classes.

Before you start searching, you need to create an index of all the words in the file you are going to search. This is analogous to the indexing process in which real search engines constantly update their indices and "crawl" the web looking for new web pages. In *Giggle*, the index is a map that associates each word in a text file with a list of all lines in the file that contain that word. With a little more work, we can upgrade *Giggle* to build an index for multiple files.

As usual, we will supply a GUI class, `Giggle`, that loads files, accepts user input, and displays the search results. You'll write the `SearchEngine` class that builds the index for a file and generates the search results. Your class must use a `HashMap` to hold the word index. In this map, a key is a word (in lowercase letters) and the associated value is a `List<String>`. The list holds all the lines in the file that contain the corresponding keyword.

Here are the specs for your class:

Fields:

```
private String myURL;
  // holds the name for the "url" (file name)

private Map<String, List<String>> myIndex;
  // holds the word index
```

Constructor:

```
public SearchEngine(String url)
```

Saves `url` in `myURL`; initializes `myIndex` to an empty `HashMap` with an initial capacity of 20,000. Note: this constructor does not load the file; the `Giggle` class reads the file and passes one line at a time to `SearchEngine`.

Public methods:

```
public String getURL()
```

Returns `myUrl`. I call this method from `Giggle` to display the name of the file in which hits were found. In the present version I already know the file name, but eventually an expanded version of `Giggle` may need to index several files.

```
public void add(String line)
```

Extracts all words from `line`, and, for each word, adds `line` to its list of lines in `myIndex`. This method obtains a set of all words in `line` by calling a private method `parseWords(line)` described below. Use an `ArrayList<String>` object to represent a list of lines associated with a word.

```
public List<String> getHits(String word)
```

Returns the list of lines associated with `word` in `myIndex`.

Private method:

```
private Set<String> parseWords(String line)
```

> Returns a set of all words in `line`. Use the same technique for extracting all the words from `line` as you did in the *Index Maker* lab in Section 11.6: call `line.split("\\W+")`. Add all the elements from the resulting array to a `HashSet`. Skip empty words and convert each word to lower case before adding it to the set. `parseWords` uses a set, as opposed to a list, because we don't want to index the same line multiple times when the same word occurs several times in it. When we add words to the set, duplicates are automatically eliminated.

Combine your class with the `Giggle` class, located in `JM\Ch20\Giggle`. Test `Giggle` thoroughly on a small text file with line breaks (for example, any java file or `JM\Ch11\IndexMaker\fish.txt`). Be sure to try searching for words that are not in the file as well as those that appear in several lines and multiple times in the same line.

20.7 Summary

In a *lookup table*, each key is converted through some simple formula into a non-negative integer, which is used as an index into the lookup table array. The associated value is stored in the element of the array with that index. Lookup tables can be used when the keys can be easily mapped onto integers in a relatively narrow range. All allowed keys correspond to valid indices in the table, and different keys correspond to different indices. Lookup tables provide instantaneous access to data, but a sparsely populated lookup table may waste a lot of space.

In the *hashing* approach, a hash function converts the key into an integer that is used as an index into a hash table. Different keys may be hashed into the same index, causing *collisions*. The *chaining* technique resolves collisions by turning each slot in the hash table into a "bucket" that can hold several values. The performance and space requirements for hash tables may vary widely depending on the implementation. Data access time in a hash table is usually very quick, but the performance may deteriorate with a lot of collisions.

`java.util`'s `HashSet` and `HashMap` classes implement the `Set` and `Map` interfaces, respectively, using hash tables. The objects kept in a `HashSet` and the keys in a `HashMap` have to have a reasonable `hashCode` method defined for them, which agrees with the `equals` method. Java library classes, such as `String`, `Double`, and `Integer`, have suitable `hashCode` methods defined. You can often write a reasonable `hashCode` method for your own class by calling library `hashCode` methods for one or several fields of your class.

A `hashCode` method may return an integer from the whole integer range. The `HashSet` and `HashMap` methods further map the hash code onto the range of valid table indices for a particular table.

Exercises

Sections 20.1-20.3

1. A class `LookupState` implements a lookup table that helps find the full names of states from their two-letter postal abbreviations (in which both letters are uppercase). The lookup table is implemented as an array of 676 entries (676 is 26 times 26), of which only 50 are used.

```
public class LookupState
{
  private static String[] stateNames = new String[676];

  public static void add(String abbr, String name)
  { ... }

  public static String find(String abbr)
  { ... }

  private static int lookupIndex(String abbr)
  { ... }
}
```

Devise a method for mapping a two-letter state abbreviation into an integer index from 0 to 675 returned by the `lookupIndex` method, write the `add` and `find` methods, and test your class. ≤ Hint: Recall from the *Cryptogram* lab in Section 20.3 that `Character.getNumericValue(ch)` returns consecutive integers for letters 'A' - 'Z'. ≥

2. Write a method

```
public List<String> sortByFirstLetter(List<String> words)
```

that takes a list of words, all of which start with capital letters, and returns a new list (an `ArrayList`) in which all the words that start with an "A" are followed by all the words that start with a "B," and so on. All the words that start with a particular letter must be in the same order as in the original list. Use the following algorithm:

1. Create an `ArrayList` of 26 queues.
2. Scan `words` once from the beginning and put each word into the appropriate queue based on its first letter.
3. Collect all the words from the queues into a new `ArrayList`.

Hint:

```
import java.util.Queue;
import java.util.LinkedList;
...
     Queue<String> q = new LinkedList<String>();
     ...
     q.add(word);
     ...
     word = q.remove();
```

3. The class `PhoneCall` represents a record of a telephone call:

```
public class PhoneCall
{
  ...
  public int getStartHour() { ... }  // European time: 0 -- 23
  public int getStartMin() { ... }
  public int getDuration() { ... }   // in seconds
  ...
}
```

Write a method

```
public int busiestHour(List<PhoneCall> dayCalls)
```

that returns the hour (a value from 0 to 23) in which the largest number of calls originated. Count only those calls that lasted at least 30 seconds. Your method must scan the list only once, using a "for each" loop. ✓

4.◆ Implement *Radix Sort* for a list of words.

Radix Sort is a sorting method that is not based on comparing keys but rather on applying the lookup or hashing idea to them. Suppose we have a large list of integers with values from 0 to 9. We can create 10 buckets, corresponding to the 10 possible values. In one pass through the list we add each value to the appropriate bucket. Then we scan through the ten buckets in ascending order and collect all the values together. The result will be the list sorted in ascending order.

Now suppose we have some data records with keys that are integers in the range from 0 to 99999. Suppose memory limitations do not allow us to use 100000 buckets. The Radix Sort technique lets us sort the keys one digit at a time: we can complete the task with only 10 buckets, but we will need five passes through the list. We have to make sure that the buckets preserve the order of inserted values; for instance, each bucket can be a list with the values inserted at the end (or a queue). We start with the least significant digit in the key (the units digit) and distribute the data values into buckets based on that digit. When we are done, we scan all the buckets in ascending order and collect the data back into one list. We then take the second digit (the tens digit) and repeat the process. We have to make as many passes through the data as there are digits in the longest key. After the last pass, the list is sorted.

The Radix Sort method works for data with any keys that permit positional representation. For integers, using hexadecimal digits or whole bytes is actually more appropriate than decimal digits. To sort words in lexicographic order we can use radix sort with a bucket for each letter or symbol. We have to pad (logically) all the words with "spaces" to the maximum length and start sorting from the rightmost character position.

Write the method

```
public LinkedList<String> sort(List<String> words)
```

The method should take a list of words and return a linked list of these words sorted alphabetically in ascending order, using Radix Sort. Assume that all the words are made up of uppercase letters 'A' through 'Z'. Recall that `Character.getNumericValue(ch)` returns consecutive integers for letters 'A' through 'Z'.

Continued ⇩

Use an `ArrayList` of `List<String>` to hold the temporary buckets (each bucket is a `LinkedList` of words). Don't forget to "pad" (logically) shorter words with spaces: the first bucket should be reserved for words that are shorter than the letter position at the current pass, so you'll need to use 27 buckets. Use a "for each" loop for each bucket to append the words from that bucket to the new list (or use `LinkedList`'s `addAll` method).

Sections 20.4-20.7

5. Define:

hashing *chaining*
hash function *bucket*
collisions

6. A class `RecordsHashTable` implements a set of `Record` objects as a hash table —

```
private ArrayList<Record> buckets = new ArrayList<Record>();
```

It resolves collisions by chaining, with buckets implemented as `ArrayList`s.

(a) Assuming that a class `Record` has a method `hashCode` that returns an integer from 0 to 999 and a method `equals(Object other)`, write a `RecordsHashTable`'s method

```
public boolean contains(Record record)
```

that returns `true` if `record` is found in the set; `false` otherwise.

(b)■ Suppose a reference takes 4 bytes and `Record` information takes 20 bytes. The average number of collisions is 5. Suppose we can convert our hash table into a lookup table by using 12 times more slots and a different `hashCode` method. Will we use more or less space? By approximately what percentage? How many times faster, on average, will the retrieval operation run, assuming that computing the old and the new `hashCode` method and comparing two records takes the same time? ✓

7.■ A hash table has sixty entries. Devise and test a hash function for English words such that all the different words from this paragraph are hashed into the table with no more than four collisions. Do not call any `hashCode` methods.

Compare the performance of your method with the more standard `Math.abs(word.hashCode()) % 60`.

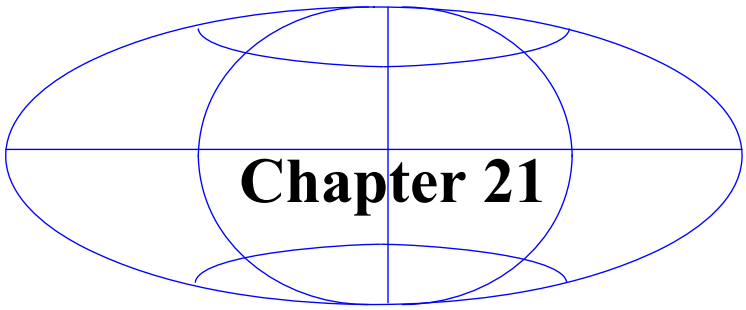
8.■ Write a `hashCode` method for the `TicTacToeBoard` class. An object of this class represents a tic-tac-toe position. It has a method

```
public char charAt(int row, int col)
```

that returns the character at the (`row`, `col`) position ($0 \le$ `row` < 3; $0 \le$ `col` < 3): space, `'o'`, or `'x'`. There are 3^9 possible configurations on a 3 by 3 board (although only a fraction of them can occur in a real tic-tac-toe game). Define your `hashCode` method in such a way that it returns different values for all 3^9 board configurations.

9. (MC) In a `HashSet`, which of the following approaches is used to determine whether two objects are equal?

A. Only == operator
B. Only `equals`
C. Both `equals` and `hashCode` (a `HashSet` works properly only when the two agree)
D. Both `compareTo` and `hashCode`



Chapter 21

Computing in Context:

Creative, Responsible, and Ethical Computer Use

21.1 Prologue

When early humans harnessed the power of fire, they found it kept them warm, helped prepare their food, and kept wild beasts at bay but could also burn down their huts or rage uncontrollably as a forest fire, destroying everything in its path. Humans have been playing with fire ever since. Technological progress has brought new benefits but also new dangers and fears. For instance, advances in nuclear physics promise unlimited sources of cheap energy. But they also threaten to pollute our planet with nuclear waste or destroy it completely in a nuclear war. Advances in chemistry, biology, and medicine are helping eradicate devastating diseases, develop new life-saving drugs and vaccines, and increase food production. But they have also brought pollution, large-scale production of brain-damaging drugs, and hideous chemical and biological weapons. Computer technology seems pretty harmless by comparison: bits and bytes flipping inside tiny silicon chips. But is it?

In the second decade of the twenty-first century we are still at the very dawn of the computer era. What will it bring us? This technology goes to the very core of humanity: the human mind, the way we acquire and process information and communicate with each other. Will computers help make us safer, better informed, or more free? Or will we become a population of networked slaves, duped by misinformation, serving an invisible master? As science-fiction authors have imagined the worst, professional ethicists and system developers have thought a great deal about how to keep it from happening. Issues of responsible computer use, computer ethics, security, and privacy have been in the foreground since the first computers were built. This field of inquiry, in fact, is much broader and more complicated than computer technology itself. In this brief overview we can only give you a glimpse of the many complex issues involved.

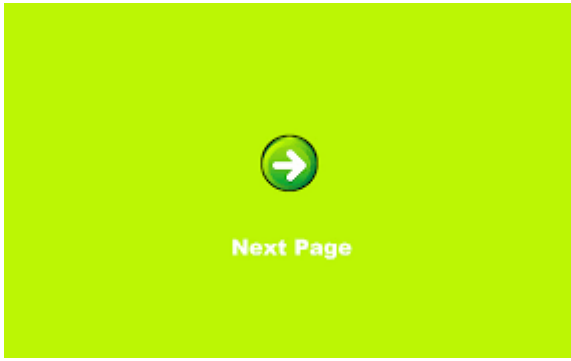
The first difficulty is the newness of the field. The use of more traditional technologies is governed by laws and ethical principles accepted in a community and transmitted from the elders to new generations. Computer technology, however, is developing and changing so fast that laws have been unable to keep up and customs and traditions have had no time to develop. Right now it is the younger generation, the teenagers, who have the most experience and savvy in using computers and the Internet. Your parents most likely cannot teach you ethical computer behavior or good Internet manners. You are on your own, and you are holding the future in your hands.

Another problem is that cyberspace is the first truly global phenomenon. It has no boundaries, no tariffs, no customs inspectors, no immigration visas. The distance between point *A* and point *B* in cyberspace is measured only by common interest, opinion, and intent.

Of course, this allows inter-regional and international collaborative projects of unprecedented speed and scale. For instance, an organization called the Global Schoolhouse maintains the Internet Project Registry, which lists hundreds of collaborative projects for all ages. However, like any society, the global society of Cyberspace has its "bad guys." A malicious computer virus released in a remote country can reach computer systems all over the world in seconds. A user in the United States can log into online gambling or pornography sites located halfway around the world. The Internet unites communities and countries with different legal systems, different customs and cultures, and different languages. How do you go about developing universal legal and ethical codes for all one billion users?

21.2 - 21.6

These sections and suggested activities are online at
`http://www.skylit.com/javamethods4`.


Next Page

Index