Second AP* Edition
— with GridWorld

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
9 Bartlet Street, Suite 70
Andover, MA 01810

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

Library of Congress Control Number: 2010915303

ISBN   978-0-9824775-7-1

ch 001

An Introduction to Hardware, Software, and the Internet

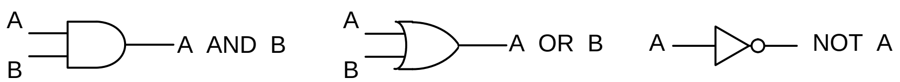
1.1 Prologue

The most important piece of a typical computer is the *Central Processing Unit* or *CPU* [1]. 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," "high" or "low," "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 [1] 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, [1] 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.

(a) AND gate | **(b)** OR gate | **(c)** NOT gate

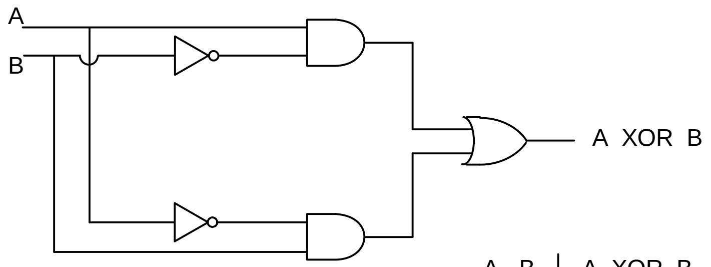


A	B	A AND B
T	T	T
T	F	F
F	T	F
F	F	F

A	B	A OR B
T	T	T
T	F	T
F	T	T
F	F	F

A	NOT A
T	F
F	T

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



A	B	A XOR B
T	T	F
T	F	T
F	T	T
F	F	F

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* [1, 2]. Also on the motherboard are memory chips. The computer memory is a uniform pool of storage units called *bits*. A bit stores the smallest possible unit of information: "on" or "off," "1" or "0." For practical reasons, bits are grouped into groups of eight, called *bytes*.

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 2010 had 2 to 4 "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 carries out 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* [1] that provide input and output and secondary mass storage. In a laptop, the "peripheral" devices are no longer quite so peripheral: a keyboard, a display, a hard drive, a DVD drive, a wireless network adapter, a web cam (camera), a touch pad, a microphone, and speakers are all built into one portable unit.

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

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

In the rest of this chapter we will briefly discuss:

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

1.2 Hardware Overview

1.2.1 The CPU

What a CPU can do is defined by its instruction set and internal *registers*. The registers are built-in memory cells that hold operands, memory addresses, and intermediate results. Some of the registers are accessible to the programmer. The instruction set includes instructions for loading CPU registers from memory and storing their values into memory, for logical and arithmetic operations, and for altering the sequence of operations.

Every computer has an internal "clock" that generates electrical pulses at a fixed frequency. All CPU operations and their component steps are synchronized with the clock's pulses; their duration is measured in *clock cycles*. The CPU's speed depends on the frequency of the clock. The Intel 8088 microprocessor in the original IBM Personal Computer (1981), for example, ran at 4.77 MHz (megahertz, or million pulses per second). Thirty years and dozens of computer generations later, CPU chips are pushing into the 3 GHz range (1 GHz, gigahertz, is equal to 1,000 MHz) [1].

A microprocessor CPU connects to memory and to other devices through a set of parallel lines (wires printed on the motherboard) controlled by digital electronics, called a *bus* [1]. A CPU uses a separate *address bus* for specifying memory addresses and a *data bus* for reading and writing memory values. Besides the internal clock speed, the computer's overall performance depends on the speed of the bus transfers and the width of the bus. The Mac 512k computer [1], introduced in September of 1984, had the Motorola MC68000 8-MHz CPU and a 16-bit bus running at 8 MHz, so it could carry 16 bits of data concurrently from memory to the CPU roughly at the same speed as the CPU could handle the data. The MacBook released by Apple Computer Inc. in 2010 has a 64-bit bus running at around 1 GHz.

1.2.2 Memory

Each byte of memory has a unique address that can be used to fetch the value stored in the byte or write a new value into it. A CPU does not have to read or write memory bytes sequentially: bytes can be accessed in any arbitrary order. This is why computer memory is called *random-access memory* or *RAM*. This is similar to a CD where you can choose any track to play, as opposed to a tape that has to be played in sequence.

Table 1-1 shows units used to describe memory size. (Powers of 2 have a special significance in computer technology for a reason that will become clear shortly.)

Name	Abbrev.	Bytes	Equals
Kilobyte	KB	$2^{10} = 1{,}024$	
Megabyte	MB	$2^{20} = 1{,}048{,}576$	1024 KB
Gigabyte	GB	$2^{30} = 1{,}073{,}741{,}824 \approx 10^9$	1024 MB
Terabyte	TB	$2^{40} \approx 10^{12}$	1024 GB
Petabyte	PB	$2^{50} \approx 10^{15}$	1024 TB

Table 1-1. Memory units

In the early days, designers of personal computers thought 64KB of RAM would suffice for the foreseeable future. An additional hardware mechanism, the *segment registers*, had to be added to the later versions of Intel's microprocessors to access a larger memory space, up to one megabyte, while maintaining compatibility with the old programs. But the one megabyte limit very quickly proved inadequate too. A 32-bit memory address bus allows programs to directly address four *gigabytes* (GB) of memory. This is becoming a bottleneck as computers beginning to appear with 8GB of RAM.

A small part of the computer memory is permanent non-erasable memory, known as *read-only memory* or *ROM*. ROM contains, among other things, the initialization code that *boots up* the *operating system* (that is, loads into memory the *boot record* or initialization code from the disk and passes control to it).

An executable program has to be loaded from a hard disk or another peripheral device (or from the Internet) into RAM before it can run.

ROM solves the "first-program" dilemma — some program must already be running to load any other program into memory. The *operating system* is a program that has the job of loading and executing other programs. In a personal computer, ROM also contains the computer configuration program and hardware diagnostic programs that check various computer components. The ROM BIOS (Basic Input Output System) contains programs for controlling the keyboard, display, disk drives, and other devices. A special small memory — EPROM (Erasable Programmable ROM) — preserves system configuration data when the power is off.

1.2.3 Secondary Storage Devices

A computer's RAM has only limited space, and its contents are wiped out when the power is turned off. All the programs and data in a computer system have to be stored in secondary mass storage. The auxiliary storage devices include hard disks, CD-ROM drives, USB flash drives, and other devices. A hard disk can hold hundreds of gigabytes and even terabytes. A CD-ROM can hold more than 700 MB (or up to 2 or 3 GB with data compression). Access to data on these devices is much slower than access to RAM.

The operating system software organizes the data in secondary storage into *files*. A file may contain a set of related data, a program, a document, an image, and so on; it has a unique name. The operating system maintains a *directory* of file names, locations, sizes, dates and times of the last updates, and other attributes.

Thus a "file" is not a hardware but rather a software concept.

When a program is running, it can read and write data directly to and from files stored on secondary storage devices.

1.2.4 Input and Output Devices

A personal computer receives user input through the keyboard and displays the output on the computer display (also called the *monitor*). In many programs the input is echoed on the screen as you type, creating the illusion that the keyboard is directly connected to the monitor. In fact these are two entirely different devices connected only through the CPU and the currently running program. The keyboard sends to the program digital codes that represent the pressed keys. The program captures these codes and takes appropriate actions, which may include displaying corresponding characters on the screen.

The screen is controlled by a *video adapter* and displays the contents of special video memory in the adapter, called *VRAM*. VRAM is addressable by the CPU and may contain codes, colors and attributes of characters when running in text modes, or colors or intensities of individual *pixels* ("picture elements") in graphics modes. The original IBM PC ran mostly in the text mode. You don't see the text mode much any more — everything is in graphics now, except diagnostic and setup programs.
A *mainframe* computer [1, 2] (a very large multi-user computer) may have hundreds of terminals attached to it. The terminals send keystrokes and receive commands and character codes from the computer through digital transmission lines.

Printers, plotters, digitizing tablets, scanners, and other devices receive commands and data from the computer in digital form and may send data or control codes back to the computer according to a specific communications protocol.

Network adapters and modems are essential for getting your computer connected to other computers. Network adapters and cables are used to connect several computers into a LAN (Local Area Network), which in turn may be connected to the Internet.

Special data acquisition devices equipped with *A/D* (*analog-to-digital*) converters [1] allow computers to convert an electrical signal into digital form by frequently sampling the amplitude of the signal and storing the digitized values in memory. *D/A* (*digital-to-analog*) converters perform the reverse transformation: they generate electrical currents from the digitized amplitudes stored in the computer. These devices allow the computer to receive data from all kinds of instruments and to serve as a universal control device in industrial applications and scientific experiments.

Input and output devices are connected to the computer via hardware *interface* modules that implement specific data transfer protocols. In a personal computer, the interfaces may be built into the motherboard or take the form of special adapter cards that plug into special sockets on the motherboard, called extension slots. Devices connected to the computer are usually controlled by special programs called *drivers* that handle all the details and peculiarities of the device and the data transfer protocol. Today's personal computers are equipped with several USB (Universal Serial Bus) ports, used for connecting virtually all peripheral devices: mice, printers, digital cameras, external disk drives, and so on. A high-speed USB interface can handle a data transmission rate of up to 480 megabits per second. (The emerging Superspeed USB 3.0 will increase the transmission rate tenfold.)

1.3 Software Overview

The term *software* refers to computer programs; it is also used as an adjective to refer to tasks or functions implemented through programs, as in "software interface," "software fonts," and so on. The line between hardware and software is not always clear. In the modern world, microprocessors are embedded in many objects, from microwave ovens and DVD players to cars and satellites. Their programs are developed using simulation tools on normal computers; when a program is finalized, it is permanently "burned" into ROMs. Such programs are referred to as *firmware*.

A modern computer not only runs individual programs but also maintains a "software environment." This environment involves several functional layers (Figure 1-3). The bottom layer in this environment comprises BIOS, device drivers, interrupt handlers, etc. — programs that directly support hardware devices and functions. The next layer is the *operating system*, a software program that provides computer access services to users and standard support functions to other programs. The top layer is software applications (word processors, Internet browsers, business, industrial, or scientific applications, games, etc.).

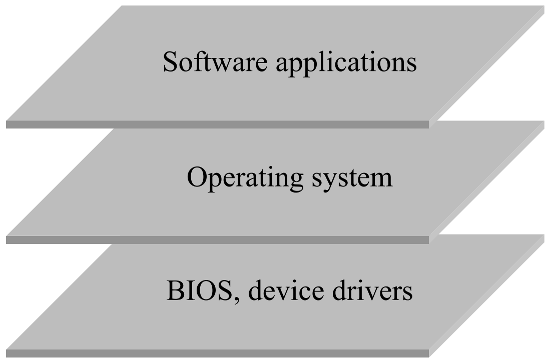


Figure 1-3. Software functional layers

The operating system loads programs into RAM from secondary storage and runs them. On a mainframe, the operating system allows multiple users to work on the computer at once through *time sharing*. In such a multi-user system, one user may be slowly editing a file or entering data on a terminal using only a small fraction of the available CPU time. At the same time another program may be doing "number crunching." A multi-user operating system allocates "time slices" to each program and automatically switches between them. The operating system prioritizes the jobs and swaps segments of programs in and out of memory as needed. A personal computer assumes one user, but contemporary users enjoy *multi-tasking* operating systems that let them keep several programs active concurrently (for example, a word processor, an e-mail program, and a music player).

The operating system also establishes and maintains a file system in secondary storage. Files are organized into a branching structure of directories and subdirectories (also called *folders*). The operating system provides commands and utility programs for navigating through the directory tree. Part of the operating system is a set of *routines* (sets of instructions, callable from other programs) that provide standard service functions to programs. These include functions for creating, reading, and writing files. The operating system *shell* provides a set of user commands, including commands for displaying, copying, deleting, and printing files, executing programs, and so on. Modern operating systems use *GUI* (*Graphical User Interface*, pronounced "gooey"), where commands can be entered by selecting items in menus or by clicking a mouse on an icon that represents a command or an object graphically.

The top layer of software consists of *application* programs that make computers useful to people. In the old days, application programs mostly performed calculations or processed files. If interaction with the user was required, it was accomplished via an unsophisticated text dialog: a computer would display questions (or *prompts*) in plain text and let the user enter responses, then print out the resulting numbers, tables, or reports. Such applications are called *console applications*. In a modern computing environment, users expect programs to have GUI. Menus, buttons, and icons select different functions in the program; dialog boxes and text-edit fields are used to set options and enter data or text. The shift from console applications to GUI applications has brought changes in the way programs are designed and written and brought about new software development tools and methodologies, such as *OOP* (Object-Oriented Programming).

1.4 What do Software Engineers Do?

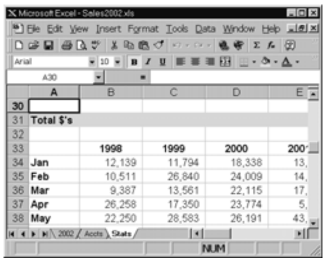
Don't they write computer software? In brief, yes, of course. But this simple answer no longer explains what the computer programming (or, to sound more lofty, "software engineering") profession has become. Software professionals create intricate and immensely complex invisible and intangible worlds. These worlds have their laws, their architectures, their aesthetics. These worlds also have "windows" into the "real" world. So many different skills are involved that computer programming has become one of the most versatile and demanding professions. We are not talking about specific technical skills, listed by cryptic acronyms in a typical employment ad for a software engineer. They are important, of course, but they are changing so rapidly that many programmers just learn them on the job. What we are talking about are the core skills that define the profession.

The most important skill is, probably, the ability to absorb new technical information all the time and to put it to work. In no other technical field are things changing so fast. Some of the changes are driven by new hardware and the new capabilities it provides; others by user demands, and some simply by fashion.

A programmer also needs to be an architect. Computers manipulate data and follow specific instructions written by a programmer. To make a computer perform useful tasks, a programmer must represent both the data and the instructions in coherent structures and procedures. Consider, for example, a word processor application. Before you proceed to write code for it, you need to have an abstract model of a "document." You consider: a document may have text, drawings, images, links, and so on. The user has to be able to edit a document, cut and paste pieces, and so on. Each of these elements and procedures must be reflected in the structure of your program. Sound software design makes it easier to implement and test the program, modify it when necessary, share the work among a team of programmers, and reuse pieces of the software in future projects.

Programmers are not building cities, bridges, or cathedrals, of course, and you won't find pieces of computer software exhibited in a museum. Very few people in the world will be able to examine the software structures that a programmer creates. But if you could enter and walk around software systems, you would find all kinds of things, from rather elegant structures to huge monstrosities, often barely standing, with pieces held together by the software equivalent of string and duct tape. A small change in one place upsets the whole structure; more strings and tape and props become necessary to prevent the contraption from collapsing. Perhaps these specimens do belong in a museum after all! For better or worse, software remains mostly invisible, hidden behind thousands of lines of code. It is mysterious, sometimes even full of surprises (Figure 1-4).

Another important skill: a programmer needs to be able to understand *algorithms* and devise his own. An algorithm is a more or less abstract, formal, and general step-by-step recipe that tells how to perform a certain task or solve a certain problem on a computer. We will discuss algorithms in Chapter 4.


Microsoft Excel - Sales2002.xls
File Edit View Insert Format Tools Data Window Help
Arial
10
A30
2002
Accts
Stats
NUM

	A	B	C	D	E
30					
31	Total $'s				
32					
33		1998	1999	2000	200
34	Jan	12,139	11,794	18,338	13,
35	Feb	10,511	26,840	24,009	14,
36	Mar	9,387	13,561	22,115	17,
37	Apr	26,258	17,350	23,774	5,
38	May	22,250	28,583	26,191	43,

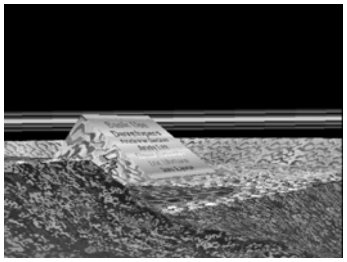


Figure 1-4. In *Excel 97*, if you pressed `Ctrl-G`, typed `L97:X97`, pressed `Enter`, pressed `Tab`, and then clicked on the "Chart Wizard" icon while holding the `Shift` and `Ctrl` keys down, a toy *Flight Simulator* popped up. Expert flying eventually revealed a monolith with the names of the product development team scrolling on it. Such a hidden "feature" is called an "Easter Egg" [1].

The next programmer's skill is using programming languages. People often refer to programmers as "a C++ programmer," "a Java programmer," and so on. They think that knowing a particular programming language is what the programming profession is about. This is not so. First, a programmer is likely to encounter several programming languages over his or her career. Second, as we have seen, a programming job has many different aspects, and mastering a particular language is just one of them.

Still, ultimately a programmer needs to be able to express his or her brilliant structural designs and clever algorithms in a written program. A program written in a high-level language must obey the very precise *syntax rules* of that language and must also follow less precise but as important stylistic conventions established among professionals. We discuss Java syntax and style in Chapter 5.

But even when a program *compiles* with no errors, there is no guarantee that it works as desired. It may show unexpected behavior, produce incorrect output, or just crash. The programmer then has to figure out what is wrong and fix the problem. This takes yet another important skill: solving mysteries. A programmer must analyze the behavior of the program, hypothesize what may be wrong, either in the algorithm or in the code, then verify his hypothesis and make corrections. Of course, there are software tools, called *debuggers*, that help programmers locate *bugs* (errors) in their programs. But it would be too slow and tedious to use a debugger for every little error. Like a good doctor, a good programmer must develop an intuition for diagnosing and locating bugs quickly.

Believe it or not, there was time when programmers wrote their programs on paper and sent them to keypunch operators who would put them on *punch cards* [1]. Now programmers use sophisticated *software development tools*. Being able to use these tools proficiently is another skill in the programmer's skill set.

In addition to general tools, a development environment for a particular programming language includes libraries of pre-programmed and pre-tested reusable software modules. There is no way to remember their technical details, of course. For example, in the 6.0 release, the standard Java library documentation consisted of 12,485 files in 844 folders and took 289 MB of disk space (which is roughly equivalent to 150,000 pages). Luckily the documentation is well organized and available online, so programmers can quickly find the information they need. But like a librarian, a programmer must know where and how to look.

Sometimes programmers forget that there are other people in the world who are not programmers, and that programs are written mostly for those other people, "users." A programmer must make sure that his or her software is "user-friendly," that is, easy to use and appears logical and intuitive to a non-technical user. Designing a friendly and effective *user interface* is another very important programming skill. A good user interface designer knows how to lay out screens, use colors, sound, animation, menus, keyboard commands, printouts and reports, and so on. The user interface is the only part of your software that is visible to the outside world, and this is often how your program is judged by non-technical users.

Last but not least, programmers must be familiar with the ethical codes for their profession and strive to uphold the highest professional ethical standards (see Appendix F).

To summarize, these are some of the things that a programmer must be able to do:

- Absorb and use emerging technical information
- Create sound software system architectures
- Understand and devise effective algorithms
- Be proficient with the syntax and style of programming languages
- Diagnose and correct programming errors
- Use software development tools and documentation
- Find and utilize reusable software components
- Design and implement friendly user interfaces
- Uphold the highest standards of professional ethics

Of course, when we said "A programmer must be able to do this; a programmer should know that," we meant an abstract "programmer" who does not exist in the real world. In reality, a number of people are involved in every software development project, and different people may possess different skills. One person may specialize in designing system architectures, another person may be good at devising complex algorithms, a third person can quickly write and debug code, while someone else may not know much about programming but be an excellent designer of user interfaces. In the software development enterprise, there is a need for different types of work and room for people with different interests and skills.

The discipline that focuses on all aspects of software development is called *computer science*. A computer scientist takes a more theoretical view of one or another aspect of software development, looking for general ways to improve the efficiency of algorithms or for better software development methodologies and new programming languages. Unlike the computer scientist, a software engineer (or programmer) works on specific software development projects.

1.5 Representation of Information in Computer Memory

Computer memory is a uniform array of bytes that does not privilege any particular type of information. The memory can contain CPU instructions, numbers and text characters, and any other information that can be represented in digital form. Since a suitable analog/digital (A/D) converter can fairly accurately convert any electrical signal to digital form, any information that can be carried over a wire can be represented in computer memory. This includes sounds, images, motion, and so on (but, so far, excludes taste and smell).

CPU instructions are represented in computer memory in a manner specific to each particular brand of CPU. The first byte or two represent the operation code that identifies the instruction and the total number of bytes in that instruction; the following bytes may represent the values or memory addresses of the operands. How memory addresses are represented depends on the CPU architecture, but they are basically numbers that indicate the absolute sequential number of the byte in memory. A CPU may have special *index registers* that help calculate the actual address in memory for a specified instruction or operand.

The format for numbers is mostly dictated by the CPU, too, because the CPU has instructions for arithmetic operations that expect numbers to be represented in a certain way. Characters (letters, digits, etc.) are represented using one of the several character code systems that have become standard not only for representing text inside computers but also in computer terminals, printers, and other devices. The code assigns each character a number.

Fortunately, high-level programming languages, such as Java, shield computer programmers from the intricacies of how to represent CPU instructions, memory addresses, numbers, and characters.

Representing other types of information is often a matter of a specific application's design. A black-and-white image, for example, may be represented as a sequence of bytes where each bit represents a pixel of the image: 0 for white and 1 for black. The sequence of pixels typically goes from left to right along each horizontal line of the image and then from top to bottom by row. Other memory locations may hold numbers that represent the image dimensions.

1.5.1 Numbers

Integers from 0 to 255 can be represented in one byte using the binary (base-2) system as follows:

Decimal	Binary
0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
6	00000110
7	00000111
8	00001000
9	00001001
10	00001010
...	...
252	11111100
253	11111101
254	11111110
255	11111111

If we use 2 bytes (16 bits), we can represent integers from 0 to $2^{16}-1 = 65535$:

Decimal	Binary
0	00000000 00000000
1	00000000 00000001
2	00000000 00000010
...	...
65534	11111111 11111110
65535	11111111 11111111

In general, k bits can produce 2^k different combinations of 0s and 1s. Therefore, k bits used as binary digits can represent non-negative integers in the range from 0 to 2^k-1. A 32-bit memory address can identify 2^{32} = 4,294,967,296 different memory locations. So if we want to be able to address each individual byte, 32-bit addresses cover 4 GB of memory space.

❖ ❖ ❖

CPUs perform all their arithmetic operations on binary numbers. A CPU may have instructions that perform 8-bit, 16-bit, or 32-bit arithmetic, for instance. Since it is difficult for a human brain to grasp long sequences of 0s and 1s, programmers who have to deal with binary data often use the *hexadecimal* (or simply "*hex*") representation in their documentation and programs. The hex system is the base-16 system, which uses 16 digits. The first ten digits are the usual 0 through 9, with the eleventh through sixteenth digits represented by the letters A through F. A byte can be split into two four-bit *quads*; each quad represents one hex digit, as follows:

Decimal	Binary	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Experienced programmers remember the bit patterns for the sixteen hex digits and can easily convert a binary number into hex and back. It is often convenient to use hex representation as an intermediate step for converting numbers from binary to decimal and back. For example:

$$700_{10} = 2\cdot16^2 + 11\cdot16 + 12 = 2BC_{16} = \underbrace{0010}_{2}\underbrace{1011}_{B}\underbrace{1100}_{C}{}_2$$

$$\underbrace{110}_{6}\underbrace{1101}_{D}{}_2 = 6D_{16} = 6\cdot16 + 13 = 109_{10}$$

The following examples show a few numbers represented in the decimal, hex, and 16-bit binary systems:

Decimal	Hex	Binary
0	0000	00000000 00000000
1	0001	00000000 00000001
12	000C	00000000 00001100
32	0020	00000000 00100000
128	0080	00000000 10000000
255	00FF	00000000 11111111
256	0100	00000001 00000000
32767	7FFF	01111111 11111111
32768	8000	10000000 00000000
65535	FFFF	11111111 11111111

For a software developer, knowing the hex system is a matter of cultural literacy. In practice, programmers who use a high-level programming language, like Java, don't have to use it very often and there are calculators and programs that can do conversions.

❖ ❖ ❖

What about negative numbers? The same bit pattern may represent an unsigned (positive) integer and a negative integer, depending on how a particular instruction interprets it. Suppose we use 32-bit binary numbers, but now we decide that they represent signed integers. Positive integers from 0 to $2^{31}-1$ can be represented as before. These use only the 31 least significant bits. As to negative integers, their representation may be *machine-dependent*, varying from CPU to CPU. Many CPUs, including the Intel family, use a method called *two's-complement arithmetic*. In this method, a negative integer x in the range from -1 to -2^{31} is represented the same way as the unsigned binary number $2^{32}-|x|$ where $|x|$ is the absolute value of x. For example, 21 and -21 will be represented as:

```
00000000 00000000 00000000 00010101 =  21₁₀
11111111 11111111 11111111 11101011 = -21₁₀
```

❖ ❖ ❖

Real numbers are represented using one of the standard formats expected by the CPU (or a separate floating-point arithmetic unit). Like scientific notation, this representation consists of a fractional part (mantissa) and an exponent part, but here both parts are represented as binary numbers. The IEEE (Institute of Electrical and Electronics Engineers) standard for a 4-byte (32-bit) representation uses 1 bit for the sign, 8 bits for the exponent and 23 bits for the mantissa. 127 is added to the exponent to ensure that negative exponents are still represented by non-negative numbers. This format lets programmers represent numbers in the range from approximately -3.4×10^{38} to 3.4×10^{38} with at least seven digits of precision. Figure 1-5 gives a few examples.

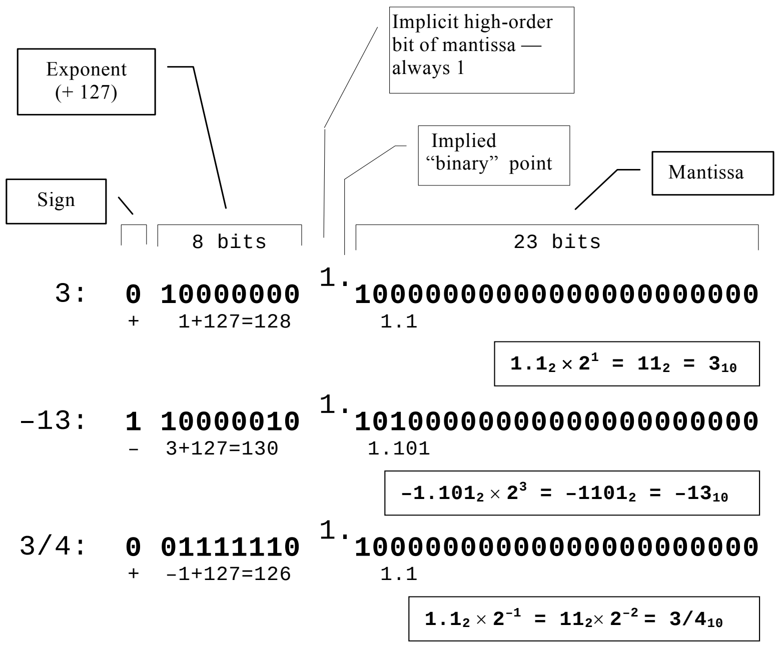


Figure 1-5. IEEE standard representation of 32-bit floating-point numbers

There are dozens of applets on the Internet that illustrate binary representation of numbers and conversions from decimal to binary.

1.5.2 Characters

Characters are represented by numeric codes. These days most applications use platform-independent Unicode standard for encoding characters [1]. For example, the reason you can see the sentence written in Russian — Многие программы используют Юникод [1] — is that the software that displays this document understands the Unicode representation of Cyrillic characters.

Java programs use Unicode internally for representing characters and text strings.

Unicode uses two bytes per character. It can encode up to 65,000 characters, enough to encode the alphabets of most world languages and many special characters. Unicode has a provision to extend the character set even further, into millions of different codes.

In the old days, the two most common character codes were EBCDIC (Extended Binary Coded Decimal Interchange Code) [1], used in IBM mainframes, and ASCII (American Standard Code for Information Interchange, pronounced as'-kee), used in personal computers, printers and other devices. Both of these use one byte per character. In the PC world, the term *ASCII file* refers to a text file (in which characters are represented in ASCII code), as opposed to a *binary file* that may contain numbers, images, or any other digitized information. Normally you won't find EBCDIC-encoded data on a PC unless the file originated on a mainframe.

Unicode includes ASCII codes as a subset (called "C0 Controls and Basic Latin" [1]. ASCII code defines 128 characters with codes from 0 to 127 and uses only the seven least-significant bits of a byte. Codes from 33 to 127 represent "printable" characters: digits, upper- and lowercase letters, punctuation marks, and so on. 32 (hex 20) is a space.

The first 32 ASCII codes (0-31) are reserved for special control codes. For example, code 13 (hex 0D) is "carriage return" (CR), 10 (hex 0A) is "line feed" (LF), 12 (hex 0C) is "form feed" (FF) and 9 (hex 09) is "horizontal tab" (HT). How control codes are used may depend to some extent on the program or device that processes them. A standard ASCII table, including the more obscure control codes, is presented in Figure 1-6.

Hex	0_	1_	2_	3_	4_	5_	6_	7_
_0	0 NUL	16 DEL	32 (SPACE)	48 0	64 @	80 P	96 `	112 p
_1	1 SOH	17 DC1	33 !	49 1	65 A	81 Q	97 a	113 q
_2	2 STX	18 DC2	34 "	50 2	66 B	82 R	98 b	114 r
_3	3 ETX	19 DC3	35 #	51 3	67 C	83 S	99 c	115 s
_4	4 EOT	20 DC4	36 $	52 4	68 D	84 T	100 d	116 t
_5	5 ENQ	21 NAK	37 %	53 5	69 E	85 U	101 e	117 u
_6	6 ACK	22 SYN	38 &	54 6	70 F	86 V	102 f	118 v
_7	7 BEL	23 ETB	39 '	55 7	71 G	87 W	103 g	119 w
_8	8 BS	24 CAN	40 (	56 8	72 H	88 X	104 h	120 x
_9	9 HT	25 EM	41)	57 9	73 I	89 Y	105 i	121 y
_A	10 LF	26 SUB	42 *	58 :	74 J	90 Z	106 j	122 z
_B	11 VT	27 ESC	43 +	59 ;	75 K	91 [	107 k	123 {
_C	12 FF	28 FS	44 ,	60 <	76 L	92 \	108 l	124 |
_D	13 CR	29 GS	45 -	61 =	77 M	93]	109 m	125 }
_E	14 SO	30 RS	46 .	62 >	78 N	94 ^	110 n	126 ~
_F	15 SI	31 US	47 /	63 ?	79 O	95 _	111 o	127 (NUL)

Figure 1-6. ASCII code used in personal computers and printers

1.6 The Internet

You basically need three things to make computers talk to each other:

1. A wire (or in more recent technology, a radio) link between them;
2. Hardware adapters at each *host* (the term used for a computer connected to a network), switches at the network junctions, and other hardware that controls the network;
3. A common language, called a *protocol*, so that the hosts can understand each other.

Of the three, the protocol is probably the most important. Once a reliable and flexible protocol is designed, the hardware and the connections will somehow follow and fall into place. Actually a protocol standard defines hundreds, even thousands of distinct protocols that function at many different levels, or, as network designers say, layers. (Figure 1-7).

The protocols in the bottom layer deal directly with the hardware — the network technology itself and various devices that are connected to it: switches, high-speed adapters, routers, and so on. These hardware protocols are constantly evolving due to changing and competing new technologies and standards set by manufacturers and professional organizations.

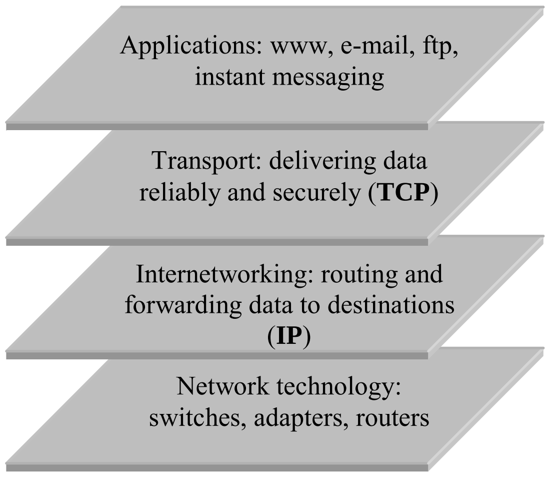

Applications: www, e-mail, ftp, instant messaging
Transport: delivering data reliably and securely (**TCP**)
Internetworking: routing and forwarding data to destinations (**IP**)
Network technology: switches, adapters, routers

Figure 1-7. TCP/IP protocols in layered network architecture

The next layer deals with routing and forwarding: how to make sure that the information from one host eventually reaches another host. These internetworking protocols must know about the general layout of the network (who is connected to whom), and be efficient and robust for reliable connections. The Internet's internetworking layer is called *IP* (the *Internet Protocol*).

The layer above internetworking, the transport protocol, is responsible for properly handling information from specific applications used on the network and for meeting the requirements of these applications: reliability, security, data compression, and so on. *TCP* (the *Transmission Control Protocol*) is the Internet's transport protocol. The *TCP/IP* combination is what really defines the Internet.

Finally, at the very top layer, there are protocols for network applications such as e-mail (using SMTP — Simple Mail Transfer Protocol), the World Wide Web (using HTTP — HyperText Transfer Protocol), file transfer (using FTP — File Transfer Protocol), instant messaging, remote terminal emulation (*telnet*), and other applications.

If we had to set a birth date for the Internet, it would probably be early September 1969 when the first computer network was tested. The network had only four nodes: University of California in Los Angeles (UCLA), Stanford Research Institute (SRI), University of California in Santa Barbara (UCSB), and the University of Utah in Salt Lake City. Here is how Dr. Leonard Kleinrock [1], a computer science professor at UCLA and one of the Internet pioneers, describes the event. Kleinrock and his group of graduate students hoped to log onto the Stanford computer and try to send it some data. They would start by typing "login" and seeing if the letters appeared on the remote terminal.

> "We set up a telephone connection between us and the guys at SRI...We typed the L and we asked on the phone, "Do you see the L?" "Yes, we see the L," came the response. "We typed the O, and we asked, "Do you see the O." "Yes, we see the O." Then we typed the G, and the system crashed! Yet a revolution had begun." [1]

More precisely, the world's first instance of host-to-host, packet-switched data communications between networked computers had taken place.

The Internet has certainly made a lot of headway since. In 2010, over 75 percent of people in the United States and almost 2 billion people worldwide (28.7%) were online [1]. The best way to explore the Internet and its history is certainly not through a book, but by getting online and browsing [1, 2]. A *browser* is a program that helps its user navigate through the Internet and presents the information that comes from the Internet back to the user. *Firefox* and *Internet Explorer* are the two most popular browsers [1]. Information comes from the Internet in many different formats. The most common one is HTML documents — text files with embedded formatting tags in them. You can find a basic HTML tutorial in Appendix C. Other files use standard formats for representing images, sounds, and so on. A browser and its helper modules, called *plug-ins*, know how to handle different formats of data and show (or play) the data to the user.

In a matter of just a few years the Internet has become a vast repository of knowledge and information (and misinformation) of all kinds and from all sources. It would be very difficult to find anything in this ocean without some guidance. *Portals* are popular web sites that arrange a large number of Internet links by category and help users navigate to the subjects they need. *Search engines* are programs based on large Internet data centers that analyze and index the contents of web pages and find web sites relevant to user queries.

As more and more individuals and businesses use computing services and resources on remote servers over the Internet, a new trend is rapidly emerging: *cloud computing*. Cloud computing may refer to different things. In its simplest form it can mean using a calendar or a calculator that runs on a remote web site or using a word processor or spreadsheet software remotely through the Internet and keeping your documents on a remote server, too, so that you can access them anywhere from any computer. Cloud computing can also mean structuring a whole enterprise's information technology needs around remote services provided by a third party vendor.

The idea is that with the Internet, computing companies become services like water and electricity are delivered to your home or business in quantity that you need. That way a business does not have to invest in expensive hardware and software; it just buys a service that meets its needs and can be easily scaled up or down according to the demand. Not everyone knows, for example, that amazon.com, besides selling books and consumer goods, sells cloud computing services through its Amazon Web Services (AWS) division [1]. Netflix, a popular movie delivery company, is an AWS's customer.

1.7 Summary

Digital electronics represents information using two states: "on" or "off," "high" or "low," "1" or "0." Digital devices, called gates, implement simple logical operations on signals: AND, OR, NOT. All other logical and arithmetic operations can be implemented using these three simple operations.

The heart of a computer is a CPU (central processing unit) that can perform logical and arithmetic operations. An executable program is a sequence of CPU instructions in machine code. It must be loaded into RAM (random-access memory) before it can run. All instructions and data addresses are encoded in binary sequences of 0s and 1s. The CPU fetches instructions and data from RAM, interprets operation codes, and executes the instructions.

RAM is arranged into bytes; each byte is 8 bits; each bit can hold either 1 or 0. The size of RAM is measured in kilobytes (1 KB = 2^{10} = 1024 bytes), megabytes (1 MB = 2^{20} = 1,048,576 bytes), or gigabytes (1 GB is over one billion bytes). The contents of RAM are erased when the power is turned off.

Mass storage devices have a larger memory capacity — hundreds of gigabytes or terabytes (1 TB is over one trillion bytes) — and they can hold the information permanently, but access to them is slower. Data in mass storage is arranged into files by the operating system software. The files are stored in a branching system of directories (represented as folders). The operating system also loads and runs applications, provides a GUI (graphical user interface) to users, and provides system services to programs (such as reading and writing files, supporting input devices, etc.).

All kinds of information are represented in the computer memory as sequences of 0s and 1s. Integers are represented as binary numbers. Real numbers use standard floating-point binary representation. Characters are represented as their codes in one of the standard coding systems. The most common codes are ASCII and Unicode. The latter includes thousands of characters from virtually all world alphabets.

The Internet is a network of millions of computers connected in many different ways. The Internet is based on the TCP/IP (Transmission Control Protocol / Internet Protocol), which controls information routing on the network and supports various network applications. Higher-level protocols are used in Internet applications such as e-mail, the World Wide Web, file transfer, and remote terminal emulation. A browser is a program on your computer that processes your requests for Internet connections and delivers and displays the received Internet information, web pages in particular. Portals are popular web sites that list many Internet links, arranged by category. Search engines are Internet indexing services that collect and index information from the Internet and help you find web sites relevant to your requests.

🕮 Exercises 🕮

The exercises for this chapter are in the book (*Java Methods: Object-Oriented Programming and Data Structures, 2nd AP Edition*, ISBN 978-0-9824775-7-1, Skylight Publishing, 2011 [1]).

Second AP* Edition
— with GridWorld

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
9 Bartlet Street, Suite 70
Andover, MA 01810

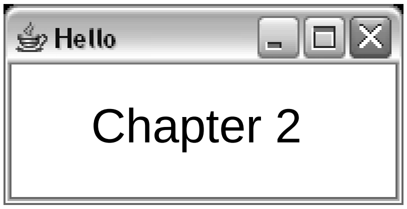
web: http://www.skylit.com
e-mail: sales@skylit.com
support@skylit.com

Library of Congress Control Number: 2010915303

ISBN   978-0-9824775-7-1

1 2 3 4 5 6 7 8 9 10   16 15 14 13 12 11

Printed in the United States of America



Chapter 2

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.

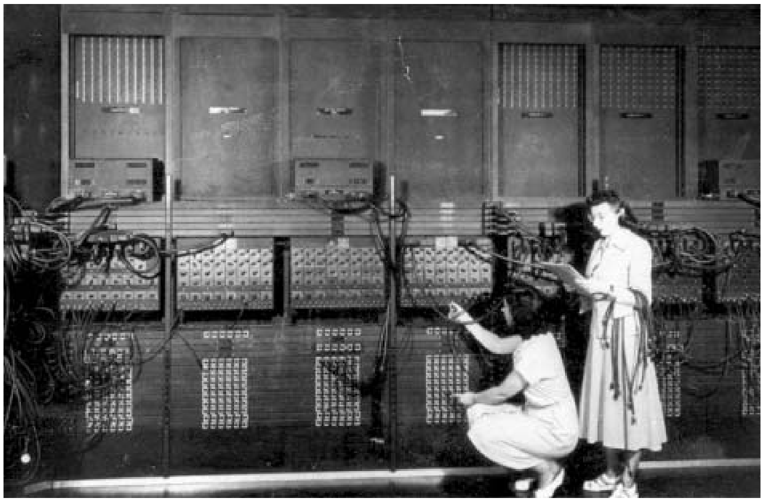


Figure 2-1. Two technicians wiring the right side of ENIAC

(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 Chapter 4). Often one or two programmers wrote the entire program and knew all about it, while no one else could understand it. Computer users were happy just to have access to a computer and were willing to learn cryptic instructions and formats for using programs.

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

Software applications that run on a desktop computer are loaded with features and must be very interactive and “*user-friendly*,” (that is, have an intuitive and fairly conventional user interface). They must also be *portable* (that is, able to run on different computer systems) and 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 teams follow standard 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 applets 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 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.

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 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).

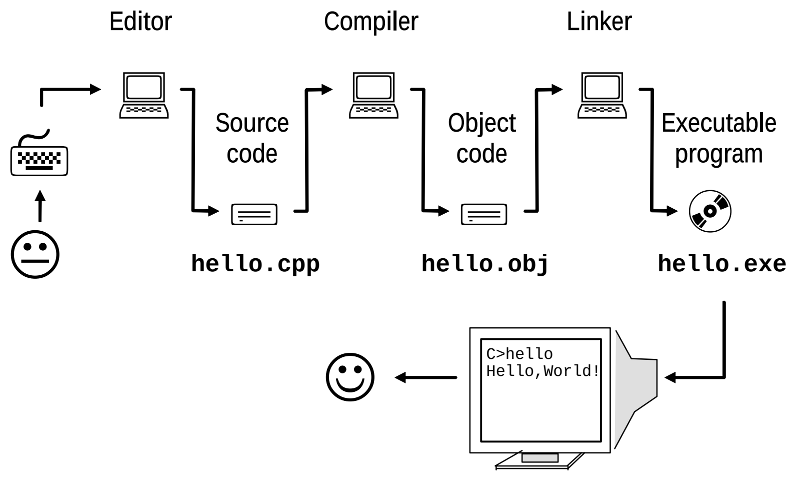


Figure 2-2. Software development cycle for a compiled program: edit-compile-link-run

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.

Java also uses a compiler, but, as we will explain shortly, the Java compiler does not generate object code.

In an alternative approach, instead of compiling, a program in a high-level language can be *interpreted* by a software tool called an *interpreter*. 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 and an interpreter, and the program has to be interpreted again each time it is run. It is like a live concert as opposed to a studio recording, and a live performance needs all the instruments each time.

❖ ❖ ❖

A particular programming language is usually established as either a compiled language or an interpreted language (that is, 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 <u>interpreted</u> 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 one of the main purposes of Java is 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, are 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 2-3). 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).

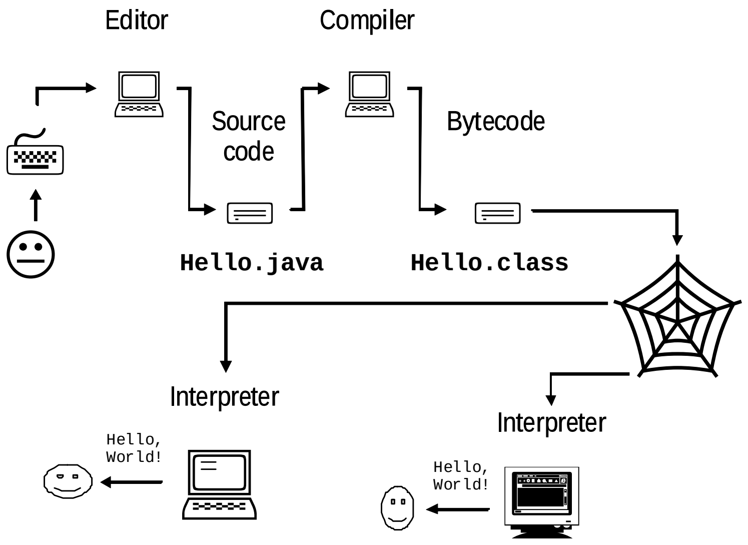


Figure 2-3. Java software development and distribution through the Internet

To speed up the loading of applets, a new software technology has emerged, 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. (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. We do not even have to use a browser to test an applet: the standard Java Development Kit (JDK) has a program, called *Applet Viewer*, that runs applets.

❖ ❖ ❖

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 2-4).

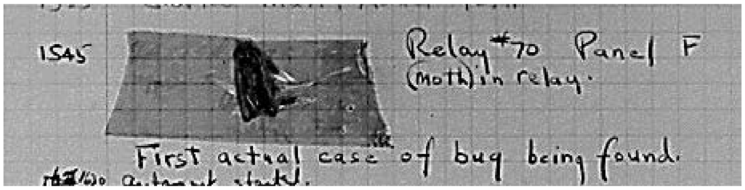


Figure 2-4. The term "bug" was popularized by Grace Hopper,❀hopper a legendary computer pioneer, who was the first to come up with the idea of a compiler and who created COBOL. One of Hopper's favorite stories was about a moth that was found trapped between the points of a relay, which caused a malfunction 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 language and are caught by the compiler. Logic errors are caused by flawed logic in the program; they are not caught by the compiler 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. 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, etc.) 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. In Chapter 20, we will talk about Java collections classes, which are part of the `java.util` package from the standard Java library. Java programmers can take advantage of dozens of standard packages that are already available for free; new packages are being developed all the time. At the same time, the plenitude of available packages and components puts an additional burden on the software engineer, who must be familiar with the standard packages and keep track of the new ones.

2.4 *Lab:* Three Ways to Say Hello

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

In this section, we will use the most basic set of tools, JDK (Java Development Kit). JDK comes from Sun Microsystems, Inc., the makers and owners of Java.

JDK includes a compiler, an interpreter, the *Applet Viewer* program, other utility programs, the standard Java library, documentation, and examples.

JDK itself does not have an IDE (Integrated Development Environment), but Sun and many third-party vendors and various universities and other organizations offer IDEs for running Java. *Eclipse, BlueJ, JCreator* are some examples, but there are dozens of others. This book's companion web site, `www.skylit.com/javamethods`, has a list of several development environments and *FAQs* (*Frequently Asked Questions*) about installing and using some of them.

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 Sun'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, the interpreter, or *Applet Viewer*. The compiler is called `javac.exe,` the interpreter is called `java.exe`, and *Applet Viewer* is called `appletviewer.exe`. These programs reside in the `bin` subfolder of the folder where your JDK is installed. This might be, for example, `C:\Program Files\Java\jdk1.6.0_21\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 `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.

In the little program above, `HelloWorld` is the name of a class as well as 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 application (but <u>not</u> an applet) must have a `main` method. 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` 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 (you'll find it under *All Programs/Accessories* on your *Start* menu). Navigate to the folder that contains your program (for example, `mywork`) using the `cd` (change directory) command, and set the path:

```
 C:\Documents and Settings\Owner>cd \mywork
 C:\mywork> path C:\program files\java\jdk1.6.0_21\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.

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 this program in using 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!");
  }
}
```

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 `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!
```

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 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 2-5). 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.

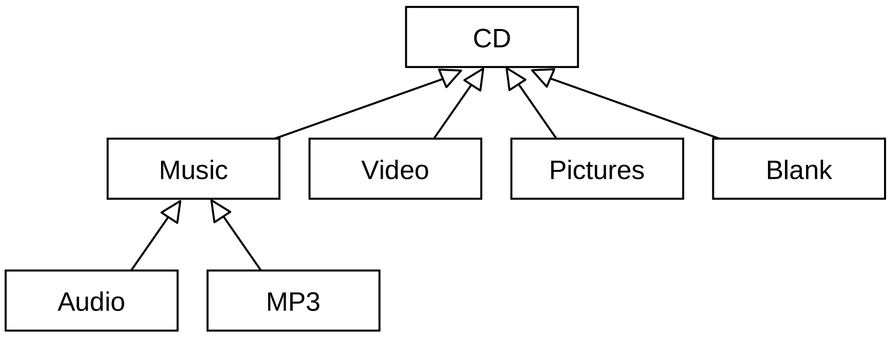


Figure 2-5. A hierarchy of classes that represent compact disks with different content

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 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 other languages are its built-in support for GUI and graphics and its support for object-oriented programming.**

In this section we will consider four more examples: two applications (one with a simple GUI object, another with graphics), and two applets (one with graphics and one with 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 2-6. In this program, the `main` method creates one object, which we call `window`. The type of this object is described as `HelloGui`; that is, `window` is an object of the `HelloGui` class. This program uses only one object of this class. `main` then sets `window`'s size and position (in pixels) and displays it on the screen. Our class has a *constructor*, which is a special procedure for constructing objects of this class. Constructors always have the same name as the class. Here the constructor calls the superclass's constructor to set the text displayed in the window's title bar and adds a label object to the window's content pane.

```
/**
 *  This program displays a message in a window.
 */

import java.awt.*;
import javax.swing.*;

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 2-6. `JM\Ch02\HelloGui\HelloGui.java`

The code in Figure 2-6 is a little cryptic, but still we can see roughly what's going on. Do not retype the program — just copy `HelloGui.java` form the `JM\Ch02\HelloGui` folder into your current work folder. Set up a project in your favorite IDE, and add the class, `HelloGui`. 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 a text label. The new class, `HelloGraphics`, is shown in Figure 2-7.

```
//  This program shows simple graphics in a window.

import java.awt.*;
import javax.swing.*;

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 = 25, y = 40:
    g.drawRect(20, 40, 150, 45);

    g.setColor(Color.BLUE);

    // Draw a string of text starting at x = 60, y = 25:
    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 2-7. **`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 box. `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 should include both classes.

3. Hello, Applet

Applets are small programs embedded in web pages and distributed over the Internet. From a programmer's point of view, the difference between an applet and a GUI application is minor. Instead of extending `JFrame`, your applet's class extends `JApplet`, a *Swing* library class that represents applet objects. An applet does not need a `main` method because the browser (or *Applet Viewer*) automatically constructs the applet object and displays it as part of a web document. Instead of a constructor, your applet class uses the `init` method to initialize your applet. Figure 2-8 shows the `HelloApplet` class adapted from the `HelloGraphics` class. This applet redefines `JApplet`'s `paint` method to show graphics.

```
/*
 * This applet shows a string of text inside a box.
 */

import java.awt.*;
import javax.swing.*;

public class HelloApplet extends JApplet
{
  public void init()
  {
    Container c = getContentPane();
    c.setBackground(Color.WHITE);
  }

  public void paint(Graphics g)
  {
    super.paint(g);    // call JApplet's paint method
                       //  to paint the background
    g.setColor(Color.RED);
    g.drawRect(25, 40, 150, 45);  // draw a rectangle 150 by 45
    g.setColor(Color.BLUE);
    g.drawString("Hello, Applet!", 60, 65);
  }
}
```

Figure 2-8. **`JM\Ch02\HelloGui\HelloApplet.java`**

The code for this applet is shorter than `HelloGraphics`. But now we need another file that describes a web page that presents the applet. The contents and layout of web pages are usually described in HTML (Hypertext Mark-Up Language). You can find a brief HTML tutorial in Appendix C.🕸 Here we can use a very simple HTML file (Figure 2-9). Let's call it `TestApplet.html`. As you can see, some of the information — the size of the applet's content pane — has shifted from Java code into the HTML file. The title bar is no longer used because an applet does not run in a separate window — it is embedded into a browser's (or *Applet Viewer*'s) window. An applet does not have an exit button either (the browser's or *Applet Viewer*'s window has one).

```
<html>

<head>
<title>My First Java Applet</title>
</head>

<body>
<applet code="HelloApplet.class" width="300" height="100"
   alt="Java class failed">
Java is disabled
</applet>
</body>

</html>
```

Figure 2-9. `JM\Ch02\HelloGui\TestApplet.html`

You can either test your applet directly in your IDE or open it in your Internet browser. If you have a website, you can upload the `TestApplet.html` page to your site, along with the `HelloApplet.class` file, for the whole world to see.

You can adapt `TestApplet.html` to run another applet by replacing `HelloApplet.class` in it with the name of your new applet class and adjusting the applet's size, if necessary.

4. Hello, Action

And now, just for fun, let's put some action into our applet (Figure 2-10). Compile the `Banner` class from `JM\Ch02\HelloGui` and open the `TestBanner.html` file (from the same folder) in the *Applet Viewer* or in the browser to test this applet.

Look at the code in `Banner.java`. The `init` method in this applet creates a `Timer` object called `clock` 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 banner and repaints the screen.

You might notice that unfortunately the animation effect in this applet is not very smooth: the screen flickers whenever the banner moves. One of the advantages of Java's *Swing* package is that it can help deal with this problem. We will learn how to do it in later chapters.

```
/* This applet displays a message moving horizontally
   across the screen. */

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

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

  public void init()
  {
    Container c = getContentPane();
    c.setBackground(Color.WHITE);
    xPos = c.getWidth();
    yPos = c.getHeight() / 2;
    Timer clock = new Timer(30, this);  // fires every 30 milliseconds
    clock.start();
  }

  // Called automatically after a repaint request
  public void paint(Graphics g)
  {
    super.paint(g);
    g.drawString("Hello, World!", xPos, yPos);
  }

  // Called automatically when the timer fires
  public void actionPerformed(ActionEvent e)
  {
    Container c = getContentPane();

    // Adjust the horizontal position of the banner:
    xPos--;
    if (xPos < -100)
    {
      xPos = c.getWidth();
    }

    // Set the vertical position of the banner:
    yPos = c.getHeight() / 2;

    repaint();
  }
}
```

Figure 2-10. **`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://download.oracle.com/javase/6/docs/api/index.html`) or, if you have the JDK documentation installed on your computer, open the file `<`*`JDK base folder`*`>/docs/api/index.html` (for example, `C:/Program Files/Java/jdk1.6.0_21/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.■ (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. ⇔

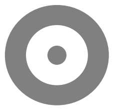
(b) Remove the parentheses around `n + n` and test the program again. How does the + operator work for text strings and for numbers? ✓

Sections 2.5-2.7

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

13. (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. ⸩

14.■ Modify *HelloApplet* (`JM\Ch02\HelloGui\HelloApplet.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. ⸩ ✓

15.■ Modify the *Banner* applet (`JM\Ch02\HelloGui\Banner.java`) to show a solid black box moving from right to left across the applet's window.

16.◆ Using the *Banner* applet (`JM\Ch02\HelloGui\Banner.java`) as a prototype, write an applet 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:

```
          Container c = getContentPane();
          int xPos = c.getWidth() / 2 - 30;
          int yPos = c.getHeight() / 2;
```

Then use a conditional statement to display the appropriate message:

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

Second AP[*] Edition
— with GridWorld

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
9 Bartlet Street, Suite 70
Andover, MA 01810

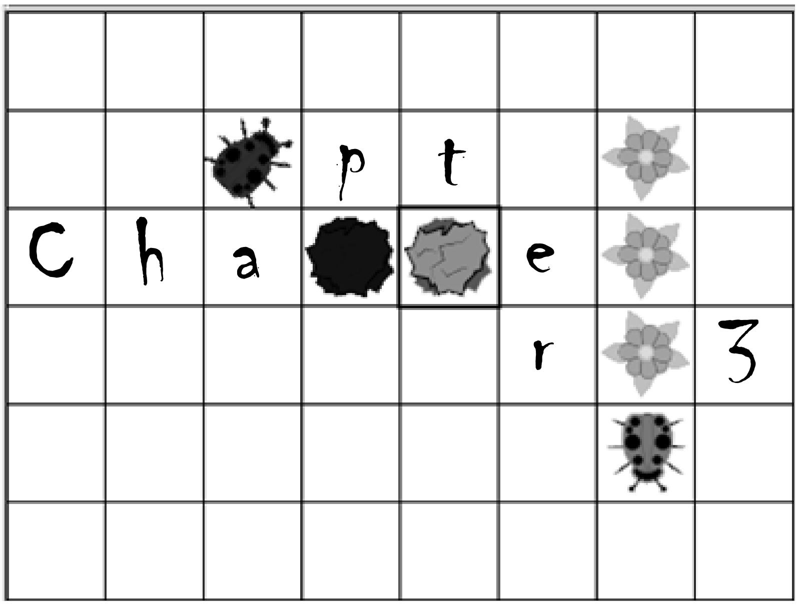
web: http://www.skylit.com
e-mail: sales@skylit.com
support@skylit.com

Library of Congress Control Number: 2010915303

ISBN   978-0-9824775-7-1

1 2 3 4 5 6 7 8 9 10   16 15 14 13 12 11

Printed in the United States of America



Objects and Classes

3.1 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 (now called a software engineer) 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, CRC card, instance variable or field, constructor, method, public vs. private, encapsulation and information hiding, inheritance, IS-A and HAS-A relationships.

In this chapter we will refer to the GridWorld case study, developed by the College Board's AP Computer Science Development Committee for AP Computer Science courses and exams. GridWorld is a *framework*, a set of Java classes that can be used to create animations and games that involve "actors" in a rectangular grid. The GridWorld materials and code are available free of charge under the GNU license at the College Board's web site.

See `www.skylit.com/javamethods/faqs/` for instructions on how to download the GridWorld materials and configure and run GridWorld projects.

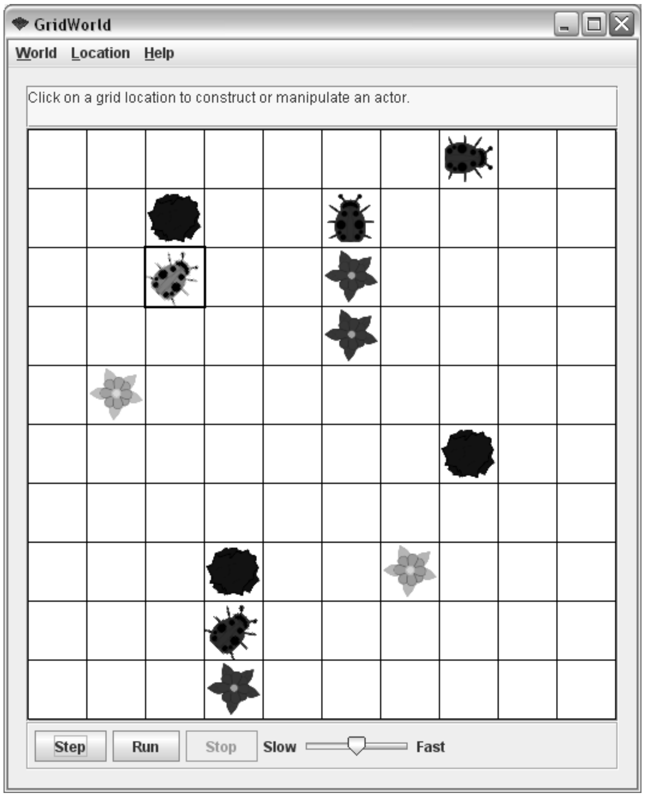
In this chapter we will work only with Part 1 and Part 2 of GridWorld. We will return to GridWorld again in Chapters 11 and 13.

3.2 *Case Study:* GridWorld

The window in Figure 3-1 comes from the *BugRunner* program in the GridWorld case study. 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 models an object or concept from the real world.

Some of the objects in a program may model real-world objects, such as a rock, a flower, or a bug. 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 rectangular grid or a location.



Click on a grid location to construct or manipulate an actor.

Figure 3-1. A window from the *BugRunner* program in GridWorld

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, in GridWorld, a "bug" object has such attributes as location, direction, and color, and such behaviors as moving and turning. The location or direction of a bug changes after each step.

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 3-1, for example, you can see four "bugs," three "rocks" and five "flowers," that is, four objects of the `Bug` class, three objects of the `Rock` class, and five objects of the `Flower` class. The flowers have different locations and colors; the bugs have different locations, directions, and colors.

In GridWorld, bugs, flowers, rocks, and other inhabitants of the grid are called *actors*.

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, in GridWorld, a flower, a rock, or another "actor" is displayed as an image. The images are stored in separate image files, and it is possible to change them without rebuilding the program. In fact, a `Bug` object will remain a `Bug` even if a program does not display it all and just prints out some information about its location and direction. 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.

Read Part 1 of the GridWorld Student Manual. Set up a project with the `BugRunner` class (located in GridWorld's `firstProject` folder) and the GridWorld library, `gridworld.jar`. (See `www.skylit.com/javamethods/faqs/` for instructions on how to set up GridWorld projects.) Experiment with the program and notice how different types of "actors" behave in the program. A bug moves forward when it can; otherwise it turns 45 degrees clockwise. A bug leaves a flower in its wake when it moves forward. A flower stays in one place, but it gets darker as it gets older. A rock just sits there and does nothing. Each of these actors "knows" the grid to which it belongs and its own location and direction in the grid.

❖ ❖ ❖

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 flowers of different colors are objects of the same class. But if we wanted to have carnivorous flowers that could catch and "eat" bugs, we would probably need a separate class to describe such objects. 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, `CarnivorousFlower` can be a subclass of `Flower`. In GridWorld, a `Bug`, a `Rock`, a `Flower` are different types of "actors" and `Bug`, `Rock`, and `Flower` are all subclasses of the class `Actor`. `Actor` is their *superclass*. More on this later (Section 3.6).

The rest of the visible objects in Figure 3-1 are GUI components. There is a menu bar, a text area for messages, and the control panel at the bottom with three buttons and a slider.

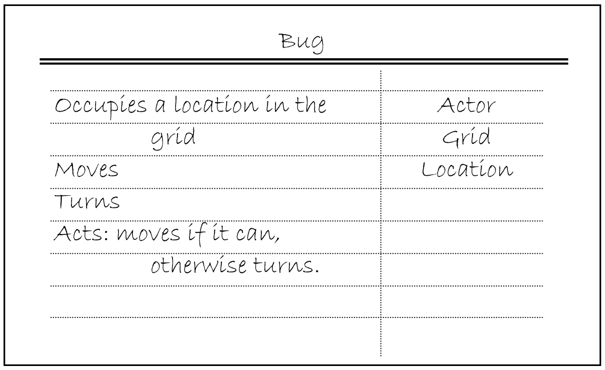
Finally, we see an object that represents the whole window in which the program is running. This object holds the grid and the GUI components.

The *BugRunner* program also uses several abstract types of objects. For example, the location of an actor is represented by an object of the class `Location`, which has the attributes row and column. A color is represented by an object of the Java library class `Color`. Another invisible object is a timer that sets the pace when the user presses the `Run` button. The `Timer` class also comes from the Java library.

❖ ❖ ❖

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 3-2).



Bug

Occupies a location in the grid	Actor
	Grid
Moves	Location
Turns	
Acts: moves if it can, otherwise turns.	

Figure 3-2. A CRC card for the class `Bug`

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.

3.3 Classes

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

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*. 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 3-3 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 3-3. ***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. When the program is running, different objects of the same class may have different values of attributes. 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, `Actor`, `Rock`, `Bug`, `Flower`, `Location`, `ActorWorld`, and `BugRunner` are some of the GridWorld classes. Their source code can be found in `GridWorldCode\framework\info\gridworld` subfolders.

All in all, the GridWorld framework includes 24 classes and one *interface* (we will talk about interfaces in Chapter 11). In addition, `GridWorldCode\projects` subfolders hold the source code for eight project classes, including `BugRunner`, `BoxBug`, and `BoxBugRunner`.

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*.

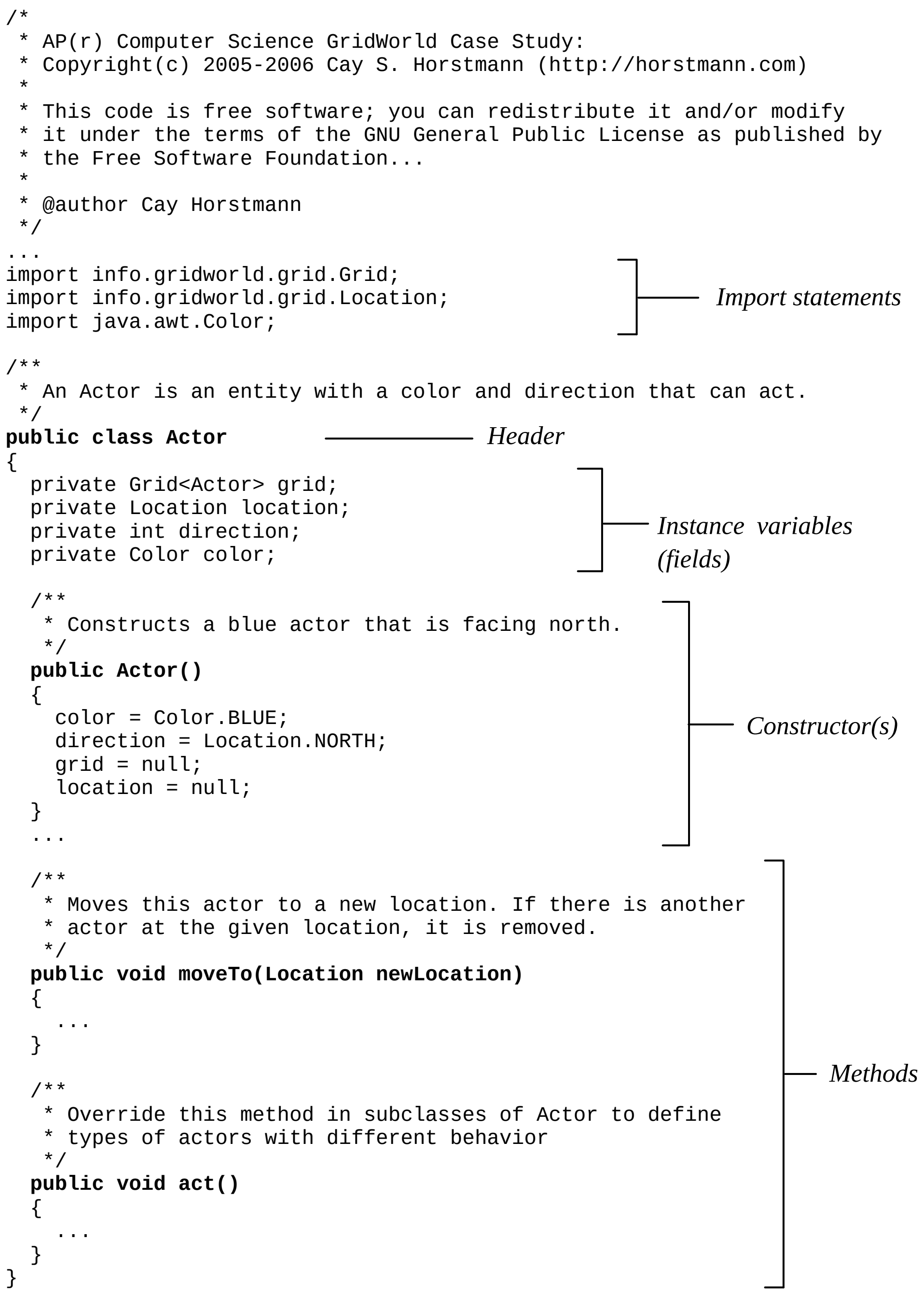
The class describes all these features in a very formal and precise manner.

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 23) 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 class as a whole, not to particular objects. It is called automatically when the program is started.

❖ ❖ ❖

Figure 3-4 shows a schematic view of a class's source code (adapted from GridWorld's `Actor` class).



```
/*
 * AP(r) Computer Science GridWorld Case Study:
 * Copyright(c) 2005-2006 Cay S. Horstmann (http://horstmann.com)
 *
 * This code is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation...
 *
 * @author Cay Horstmann
 */
...
import info.gridworld.grid.Grid;
import info.gridworld.grid.Location;
import java.awt.Color;

/**
 * An Actor is an entity with a color and direction that can act.
 */
public class Actor
{
  private Grid<Actor> grid;
  private Location location;
  private int direction;
  private Color color;

  /**
   * Constructs a blue actor that is facing north.
   */
  public Actor()
  {
    color = Color.BLUE;
    direction = Location.NORTH;
    grid = null;
    location = null;
  }
  ...

  /**
   * Moves this actor to a new location. If there is another
   * actor at the given location, it is removed.
   */
  public void moveTo(Location newLocation)
  {
    ...
  }

  /**
   * Override this method in subclasses of Actor to define
   * types of actors with different behavior
   */
  public void act()
  {
    ...
  }
}
```

Figure 3-4. A schematic view of a class's source code

You can see the following elements:

1. An comment at the top

   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.

   The compiler automatically finds the definitions of the needed classes, as long as they are located in the same folder as this class. But you need to tell the compiler where to find Java library classes and classes from different packages that reside in other folders or in "jar" files. A *Java archive* (`.jar`) file is a file that contains several pre-compiled classes (`.class` files) in compressed form. For example, all of the GridWorld framework classes are collected in one jar file, `gridworld.jar`. A jar file maintains the same structure of folders as in folders that are not compressed. Java library classes are collected in `.jar` files, too.

   If you examine GridWorld's source code, you will see that the `Actor.java` file resides in the `framework/info/gridworld/actor` folder. The `Grid` and `Location` classes used by the `Actor` class reside in a different folder, `framework/info/gridworld/grid`. So the `Actor` class needs `import` statements for these two classes:

```
import info.gridworld.grid.Grid;
import info.gridworld.grid.Location;
```

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

```
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. Or we could put

```
import java.awt.*;
```

Then the compiler would search the entire `java.awt` package for the classes that it couldn't find elsewhere.

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

```
private info.gridworld.grid.Location location;
private java.awt.Color color;
```

instead of

```
private Location location;
private Color color;
```

This would clutter your code.

3. The class header

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

```
public class Actor
```

— states that the name of this class is `Actor` 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.

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.

❖ ❖ ❖

When a Java application starts, control is passed to the main method. In the *BugRunner* application, the main method resides in the BugRunner class. BugRunner is different from the classes we described above (Actor, Bug, Flower, Rock): it is not used to define objects, and it has no instance variables or constructors; main is its only method (Figure 3-5).

```
... (comment)

import info.gridworld.actor.ActorWorld;
import info.gridworld.actor.Bug;
import info.gridworld.actor.Rock;

... (comment)
public class BugRunner
{
  public static void main(String[] args)
  {
    ActorWorld world = new ActorWorld();
    world.add(new Bug());
    world.add(new Rock());
    world.show();
  }
}
```

Figure 3-5. Fragments from BugRunner.java

The word static in public static void main(...) indicates that the method main is not called for any particular object: it belongs to the class as a whole. In fact, main is called automatically when a Java application is started. A Java application can have only one main method.

We could place main in any class, but it is cleaner to put it in a separate class. The name of the class, BugRunner, is chosen by its author; it could be called BugTest or FirstProject instead. BugRunner's main method creates a world (an object of the type ActorWorld) and adds one Bug object and one Rock object to world (at random locations) by calling world's add method for each of the added actors. It then displays world by calling its method show and waits for a command from the user.

3.4 *Lab:* Interacting with Actors

The author of GridWorld implemented in it the *direct manipulation interface* feature, which allows you to invoke constructors and call methods of different actors interactively, by clicking on occupied and empty cells of the grid.

This feature is described in GridWorld's help, accessible from the `Help` menu. If you click on an empty square in the grid, a menu will pop up that lists all the constructors for different types of actors that are currently in the grid. You can choose one of the constructors, and the corresponding type of actor will be added to the grid at that location. If you click on a square that already contains an actor, a menu pops up that lists all the methods for that type of actor. If you choose one, that method will be executed. You can also choose (click on) an actor and press `Delete` to remove that actor from the grid.

This interactive feature is provided for study purposes only — in general it is not required (or easy to implement) in a typical OOP program.

Experiment with the *BugRunner* program. Add a few bugs and flowers of different colors interactively, when the program is already running. Also add an `Actor` object. (`Actor`'s constructor is listed among other constructors because `Actor` is the superclass for more specific types of actors, such as `Bug`, `Flower`, etc.) Make one of the bugs move, turn, and "act" by invoking the respective methods from the pop-up menu. Change the color of the `Actor` by invoking its `setColor` method. Observe how the `Actor` "acts" by invoking its `act` method. Then delete some of the actors from the grid.

3.5 Fields, Constructors, and Methods

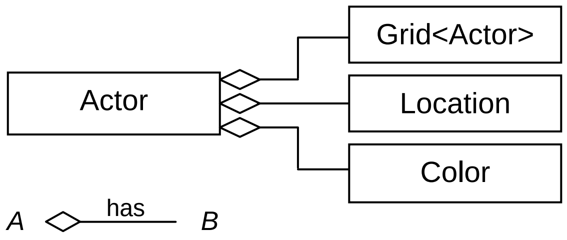
As we said earlier, the definition of a class describes all the *instance variables* of objects of this class. Instance variables are also called *data fields* or simply *fields*, from the analogy with fields in a form that can be filled in with different values.

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

```
   private Grid<Actor> grid;
   private Location location;
   private int direction;
   private Color color;
```

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 represented by 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`, `char`, etc., called *primitive data types*.)

The `grid` field in `Actor` refers to the grid to which this actor belongs. The type of this field is `Grid<Actor>`. `location` is another field; its type is `Location`: that is, it is an object of the `Location` class. `color` is another field; its type is `Color`. An OOP designer would say that an `Actor` *HAS-A* (has a) `Grid<Actor>`, HAS-A `Location`, and 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:

```
     Actor alice = new Actor();
```

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:

```
public Actor()
{
  color = Color.BLUE;
  direction = Location.NORTH;
  grid = null;          // will be set later when this actor is
  location = null;      //    added to a grid
}
```

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, or "args" for short).

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

For example, the `Bug` class defines two constructors:

```
public Bug()
{
  setColor(Color.RED);
}
```

and

```
public Bug(Color bugColor)
{
  setColor(bugColor);
}
```

The first one is a no-args constructor; it creates a red bug; the second takes one parameter, color, and creates a bug of that color. (Both constructors call `Actor`'s `setColor` method to set the color of the actor.)

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.

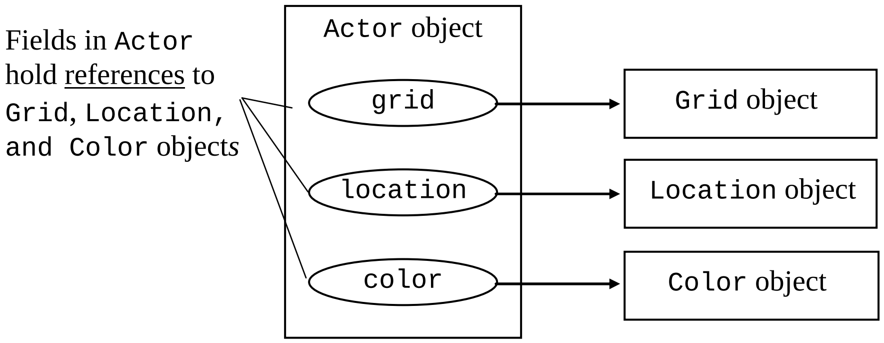
For example, if we wanted to add a green bug to `world` at location (2, 3) we could write:

```
    Location loc = new Location(2, 3);
    Bug bob = new Bug(Color.GREEN);
    world.add(loc, bob);
```

or simply

```
    world.add(new Location(2, 3), new Bug(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:


Fields in Actor hold references to Grid, Location, and Color objects
Actor object
grid
location
color
Grid object
Location object
Color object

Eventually 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).

Since `Bug` is a subclass of `Actor`, `Bug`'s constructors first call `Actor`'s no-args constructor to initialize `Actor`'s instance variables. These calls are implicit — they are not shown in `Bug`'s code.

But it is possible to call a superclass's constructor explicitly and pass parameters to it using the keyword `super`.

❖ ❖ ❖

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 `alice` of the `Actor` class and added it to a grid, we can call its `moveTo` method:

```
    alice.moveTo(loc);
```

This statement moves `alice` to the `Location loc`. The compiler understands this statement because we have defined the `moveTo` method in the `Actor` 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, if `bob` is a `Bug`, we can call its `turn` method:

```
    bob.turn();
```

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, `Actor`'s `getLocation` method returns this actor's location:

```
public Location getLocation()
{
  return location;
}
```

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

For example, `Bug`'s `move` and `turn` 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, `Bug`'s `act` method calls its `canMove` method and then its `move` or `turn` methods:

```
public void act()
{
  if (canMove())
    move();
  else
    turn();
}
```

`Flower`'s `act` method calls `Flower`'s `getColor` and `setColor` methods (inherited from `Actor`) and `Color`'s `getRed`, `getGreen`, and `getBlue` methods (which return the red, green, and blue components of the color):

```
public void act()
{
  Color c = getColor();

  int red = (int) (c.getRed() * (1 - DARKENING_FACTOR));
  int green = (int) (c.getGreen() * (1 - DARKENING_FACTOR));
  int blue = (int) (c.getBlue() * (1 - DARKENING_FACTOR));

  setColor(new Color(red, green, blue));
}
```

❖ ❖ ❖

The `Actor` class provides a well-defined functionality through its constructors and public methods. The user of the `Actor` class (possibly a different programmer) does not need to know all the details of how the class `Actor` works, only how to construct its objects and what they can do. In fact, all the instance variables in `Actor` are declared `private` so that programmers writing classes that use `Actor` 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:

- `Actor`'s programmer can change the structure of the fields in the `Actor` class, and the rest of the project won't be affected, as long as `Actor`'s constructors and public methods have the same specifications and work as before;
- `Actor`'s programmer can document the `Actor` 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. The `Actor` class is too complicated to be tested in its entirety outside GridWorld, but we can create a small program that tests some of `Actor`'s features, for example, its `getDirection` and `setDirection` methods:

```
import info.gridworld.actor.Actor;

public class TestActor
{
  public static void main(String[] args)
  {
    ____________________________  // create an Actor called alice

    System.out.println(alice.getDirection());

    _____________________________ // call alice's setDirection method
                                  //    with the parameter 90

    System.out.println(alice.getDirection());
  }
}
```

Type in the above code, filling in the blanks, and save it in a file `TestActor.java`. Create a project that includes the `TestActor` class and the GridWorld library, and test your program. Explain the output. Now try to call `setDirection` with the parameter 500. Explain the output.

3.6 Inheritance

A `Bug` does not have a method to reverse direction. It would be easy to add a method `turnAround` to the `Bug` 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. For example, if you only had `gridworld.jar` and `BugRunner.java`, but not the source code for other classes, you could still run GridWorld projects, but you couldn't change `Bug.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*.

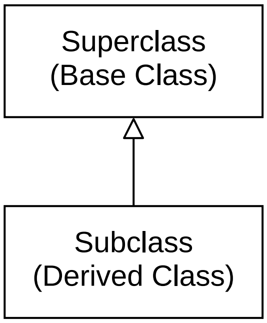
Inheritance is one of the fundamental OOP concepts, and all OOP languages support it.

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

For example,

```
public class BoxBug extends Bug
{
  ...
}
```

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:

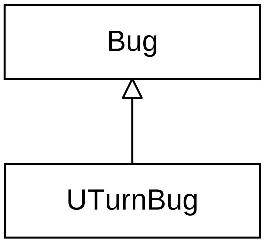


In Java you can extend a class without having its source code.

A subclass *inherits* all the methods and fields of its superclass. Constructors are not inherited; a subclass has to provide its own.

In Java every class extends the library class `Object` by default. So all the classes in a program belong to one large hierarchy of classes with `Object` at the top. `Object` supplies a few common methods, including `toString` and `getClass`.

In our example, we create a new class `UTurnBug`, which extends `Bug`:



`UTurnBug` is a short class (Figure 3-6). It has two constructors that parallel `Bug`’s constructors and adds one method, `turnAround`. It also overrides (redefines) the `Bug`’s `act` method. We should not duplicate `Bug`’s or `Actor`’s other methods and fields in the `UTurnBug` class, as they are inherited from `Bug` and `Actor`.

```
/*
 * A subclass of Bug that adds the turnAround method and
 * redefines Bug's act method so that this bug
 * makes a U-turn when it can't move
 */

import info.gridworld.actor.Bug;
import java.awt.Color;

public class UTurnBug extends Bug
{
  public UTurnBug()
  {
    setColor(Color.YELLOW);
  }

  public UTurnBug(Color bugColor)
  {
    setColor(bugColor);
  }

  public void turnAround()
  {
    turn(); turn(); turn(); turn();
            // Or: setDirection(getDirection() + 180);
  }

  // Overrides Bug's act method
  public void act()
  {
    if (canMove())
        move();
    else
        turnAround();
  }
}
```

Figure 3-6. `JM\Ch03\GridWorld\UTurnBug.java`

Set up a GridWorld project with the `BugRunner` class and the `UTurnBug` class (from `JM\Ch03\GridWorld\UTurnBug.java`). Add a statement to `BugRunner` to add a `UTurnBug` to `world`. Run the program and test `UTurnBug`'s `turnAround` method, first interactively, by invoking it from the pop-up menu, then by running the program through several steps.

❖ ❖ ❖

In the `UTurnBug` example, `UTurnBug`'s constructors take the same number and types of parameters as `Bug`'s constructors. In general, this does not have to be the case. `BoxBug`, for example, has only one constructor, and it takes one parameter, an integer (which specifies the number of steps before a turn).

Also, notice a paradox. Our `UTurnBug` bug inherits all the fields from `Bug`, which in turn has inherited all the fields from `Actor`. So a `UTurnBug` has a `direction` field. However, this and other fields are declared `private` in `Actor`. This means that the programmer who wrote the `UTurnBug` class did not have direct access to them (even if he is the same programmer who wrote `Actor`!). Recall that the `Actor` class is fully encapsulated and all its fields are private. So the statement

```
    direction = direction + 180;
```

won't work in `UTurnBug`. What do we do? `Actor`'s subclasses and, in fact, any other classes that use `Actor` might need access to the values stored in the private fields.

To resolve the issue, the `Actor` class provides public methods that simply return the values of its private fields: `getDirection`, `getLocation`, `getColor`, `getGrid`.

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. In our example, we could use

```
    setDirection(getDirection() + 180);
```

Methods like `setDirection` are called *setters* or *modifiers*.

❖ ❖ ❖

Inheritance represents the *IS-A relationship* between objects. A `Bug` IS-A (is an) `Actor`. A `UTurnBug` IS-A `Bug`. 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 an `Actor`-type object, you can plug in a `Bug`-type or a `UTurnBug`-type object instead, because a `Bug` is an `Actor`, and a `UTurnBug` is an `Actor`, too (by virtue of being a `Bug`).

For example, the class `ActorWorld` has a method `add`:

```
public void add(Actor a)
{
  ...
}
```

Since `Bug`, `Flower`, `Rock`, and `UTurnBug` are all subclasses of `Actor`, you can pass a `Bug`, a `Flower`, a `Rock`, or a `UTurnBug` object to `ActorWorld`'s `add` method:

```
world.add(new UTurnBug());
```

In Java, a collection of objects can only hold objects of a specified type. For example, the `grid` field in `Actor` is defined as a `Grid<Actor>` object — `grid` holds `Actor`s. We wouldn't be able to place bugs, flowers, and rocks into the same grid if `Bug`, `Flower`, and `Rock` were not subclasses of `Actor`.

For the same reason, the following statements will compile with no problems:

```
Actor daisy = new Flower();
Bug boxy = new BoxBug();
Actor pacer = new UTurnBug();
```

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

❖ ❖ ❖

Figure 3-7 shows a UML diagram of some of the GridWorld classes and their relationships.

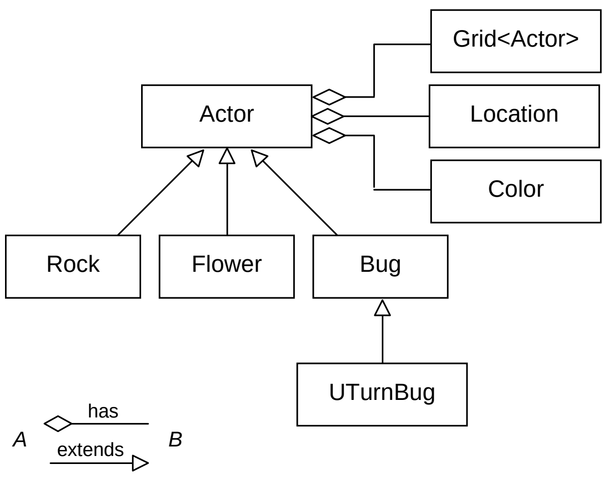


Figure 3-7. **`Actor` has a `Grid<Actor>`, `Location`, and `Color`; `Rock`, `Flower`, and `Bug` extend `Actor`; `UTurnBug` extends `Bug`**

❖ ❖ ❖

We have made our first steps in OOP. But a lot remains to be learned. We will discuss more advanced concepts and continue with GridWorld in Chapter 11.

3.7 *Lab:* Random Bugs

Read Part 2 of the GridWorld *Student Manual*. Set up and run the *BoxBug* project with the `BoxBug` and `BoxBugRunner` classes (provided in the GridWorld's `projects/boxBug` folder) and `gridworld.jar`. Review `BoxBug`'s source code.

In this lab you will create a new variation of `Bug`, a `RandomBug`. A `RandomBug` is similar to a regular `Bug`: if it can move, it does. Then, regardless of whether it has moved or not, it changes direction randomly (by a multiple of 45 degrees).

Follow these steps:

1. Copy `UTurnBug.java` into `RandomBug.java` and update the class's header and constructor names in the code. `RandomBug` should extend `Bug`.

2. Replace the `turnAround` method with a `turn(angle)` method:

   ```
   public void turn(int angle)
   {
     ...
   }
   ```

   Use `UTurnBug`'s code (Figure 3-6) as a prototype for setting the direction of the bug.

   `RandomBug` already has a `turn` method, inherited from `Bug`. That method takes no parameters. It is acceptable and sometimes desirable to give the same name to two or several methods in a class, as long as these methods take different numbers and/or types of parameters. This is called *method overloading*.

3. Modify the `act` method to make the bug move if it can. Then the bug should turn by a random angle, which is a multiple of 45 degrees (or — the same thing — simply turn in a random direction). The statement

   ```
     int angle = 45 * (int)(Math.random() * 8);
   ```

   sets `angle` to a random multiple of 45, from 0 to 315. (We will explain how it works later, in Chapter 6; for now just use it as written above.)

4. Copy `BugRunner.java` into `RandomBugRunner.java`, change the class's name in the class's header, and edit the `main` method to add a couple of `RandomBug` objects to `world`. Create a GridWorld project with `RandomBugRunner` and `RandomBug` and test your program.

Also complete Exercises 1 and 2 on page 13 of the GridWorld Student Manual.

3.8 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`. 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* or *accessors*. Methods that set the values of private fields are called *setters* or *modifiers*.

A class definition does not have to start from scratch: it can *extend* the definition of another class, adding fields and/or methods 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 3.1-3.5

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 *BugRunner* 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/6/docs/api/index.html`), or, if you have the JDK documentation installed on your computer, open the file `<`*`JDK base folder`*`>/docs/api/index.html` (for example, `C:/Program Files/Java/jdk1.6.0_21/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 `gridworld.jar` and rename it into `gridworld.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 in its `info/gridworld/actor` subfolder?

7. The constructor of GridWorld's `Location` class takes two integer parameters. Which of them is the row and which is the column of the location? Are rows and columns counted from 0 or from 1? Find out by running *BugRunner* and positioning the cursor over the grid.

8. Modify GridWorld's `BugRunner` class to place into the grid a red bug at location (1, 2) and a green bug at the upper left corner of the grid. Add a gray rock at a random location. ⸨ Hint: see `BoxBugRunner.java` for an example of how to place an actor at a specified location. ⸩ ✓

9.■ Modify `BugRunner` to place into the grid three bugs in a row, next to each other, all facing east. ⸨ Hint: turn each bug a couple of times before adding it to the grid. ⸩

10.■ Write a simple console application that creates a `Bug` and "prints it out":

```
Bug bob = new Bug();
System.out.println(bob);
```

What is displayed? How does the program know how to "print" a bug? Examine `Bug.java` and `Actor.java` to find an explanation. ✓

11.◆ 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.

12.◆ The class `Circle` (`Circle.java` in `JM\Ch03\Exercises`) describes a circle with a given radius. The radius has the type `double`, which is a primitive data type used for representing real numbers. The `CircleTest.java` class in `JM\Ch03\Exercises` is a tiny console application that prompts the user to enter a number for the radius, creates a `Circle` object of that radius, and displays its area by calling the `Circle`'s `getArea` method.

Create a class `Cylinder` with two private fields: `Circle base` and `double height`. Is it fair to say that a `Cylinder` HAS-A `Circle`? Provide a constructor that takes two `double` 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.

13.◆ `JM\Ch03\Exercises\CoinTest` 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 3.6-3.8

14. 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. ______ ✓

15. Question 12 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. ✓

16. Which of the following assignment statements will compile without errors?

(a) `Actor alice1 = new Actor();` ______
(b) `Actor alice2 = new Bug();` ______
(c) `Actor alice3 = new Flower();` ______
(d) `Bug bob1 = new Actor();` ______
(e) `Flower rose1 = new Actor();` ______
(f) `Flower rose2 = new Flower(Color.RED);` ______
(g) `BoxBug boxy1 = new BoxBug();` ______
(h) `BoxBug boxy2 = new BoxBug(5);` ______
(i) `Bug boxy3 = new BoxBug(5);` ______

Explain your answers and write a program to test them.

17.■ Write a class `WiltingFlower` as a subclass of `Flower`. Provide one constructor that takes `WiltingFlower`'s life span as a parameter and saves it in a private field. Another field, `age`, should be initialized to 0. On each step (each call to the `act` method) the age of the `WiltingFlower` should increase by 1. Once the age exceeds the life span, the flower should "die" (call its `removeSelfFromGrid` method). ⋲ Hint: do not start from scratch — adapt `WiltingFlower` from `BoxBug`. ⋺

18.■ The BoxBug class has one constructor, which takes one integer parameter:

```
/**
 * Constructs a box bug that traces a square of a given
 * side length
 * @param length the side length
 */
public BoxBug(int length)
{
  steps = 0;
  sideLength = length;
}
```

(The side of the square traced by the bug is measured between the centers of the grid cells where the bug turns. When `sideLength` is set to `0`, the bug keeps turning in one cell and never moves.)

Add three more constructors to this class: (1) a no-args constructor, which sets bug's color to `Color.red` and `sideLength` to 0; (2) a constructor that takes one parameter, `bugColor`, and sets the bug's color to `bugColor` and `sideLength` to 0; and (3) a constructor that takes two parameters, `bugColor` and `length`, and sets bug's color to `bugColor` and `sideLength` to `length`. Run the BoxBugRunner program and test the added constructors interactively, using GridWorld's direct manipulation interface feature.

19.■ Make rocks "roll." Derive a class `RollingRock` from `Rock`. Provide two constructors similar to the `Rock` constructors, but make them set `RollingRock`'s direction to southeast (135 degrees). Add a `roll` method, identical to `Bug`'s `move` method (only do not add a flower to the grid when the rock rolls). Provide an `act` method: make it simply always call `roll`. Supply a class `RockRunner` that places a few `RollingRock` objects into the grid and test your program.

20.◆ Derive a class `CarefulBug` from `Bug`. A `CarefulBug` acts as follows. Like a `BoxBug`, it counts the steps (calls to its `act` method). On count zero, it turns 45 degrees counterclockwise and increments the steps count by 1. On count one, it returns back to its original direction and increments the steps count by 1. On count two it turns 45 degrees clockwise and increments the steps count by 1. On count three it returns to its original direction and increments the steps count by 1. On count four it acts like a regular bug (if it can move, it moves, otherwise it turns) and resets the steps count to zero.

Provide two constructors similar to `Bug`'s constructors. They should also set the steps count to 0. Add a method `turnLeft` that turns the bug 45 degrees counterclockwise. Override `Bug`'s `act` method, calling `turnLeft`, `turn`, `canMove`, and `move` as necessary.

Add an orange `CarefulBug` to `world` in the `BoxRunner` program to test your `CarefulBug` class.

⋚ Hint:

```
if (steps == 0)  // If steps is equal to 0
{
  ...            //   do this
}
else if (steps == 1)
{
  ...
}
...
```

⋛

chapter ← 4

Algorithms

4.1 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 on a computer.

The paradox of this definition, however, is that algorithms existed long before computers. One of the most famous, Euclid's Algorithm for finding the greatest common factor of two integers, dates back to about 300 BC. You may also recall the algorithm for long division of numbers, often used in the pre-calculator era. 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 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, to introduce methods for describing algorithms (flowcharts, pseudocode) and to review the main algorithmic devices: decisions, iterations, and recursion. At this stage, we are not going to deal with any complicated algorithms. We will discuss searching and sorting algorithms in Chapter 13.

4.2 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 4-1).

```
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
```

Figure 4-1. 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 millions 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 number 1 and the sum 0; add the square of that number to the sum; take the next number; 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 current number and another that holds the accumulating sum of squares.

Think of a variable as a "slate" on which you can write some information. You can read the value currently written on the slate. When necessary you can erase that value and write a new one. In a computer implementation, a variable occupies a memory location, and there are CPU instructions to read a value stored in memory or to write a new value into it. In descriptions of algorithms, as in computer programs, we give variables names for convenience. In this algorithm, let us call our two variables *i* and *sum*. At the beginning, we want to set *sum* to 0 and *i* to 1. Let's record these steps using a more formal notation, called pseudocode:

```
sum ← 0
i  ← 1
```

Now we want to compute i^2 and add it to *sum*. For extra clarity, let's introduce another variable that will hold i^2. Let's call it *sq*. So our next steps are

```
sq ← i * i
sum ← sum + sq
```

The last statement indicates that we are updating the value of *sum*. This is shorthand for the following instructions: read the value written in *sum*; add *sq* to it; write the result back into *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 *i* by 1:

```
i  ← i + 1
```

The above three steps constitute the core of our algorithm. We need to repeat these steps as long as *i* remains less than or equal to 100. Once *i* becomes greater than 100, the process stops and *sum* contains the result. Thus, for each value of *i*, 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. A CPU has special instructions (called conditional branching instructions) that direct the CPU to take different paths depending on the result of the previous operation. It can, for example, compute 100 - *i* and then continue the computation if the result is non-negative or jump to a different set of instructions if the result is negative.

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 100 - *i* with *n* - *i* in the decision test.

Figure 4-2 (a) shows pseudocode for the final algorithm. Pseudocode can be somewhat informal as long as knowledgeable people understand what it means.

Figure 4-2 (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.

(a)

```
Input: n

sum ← 0

i ← 1

Repeat the following
three steps while
i ≤ n:

  sq = i * i
  sum ← sum + sq
  i ← i + 1

Output: sum
```

(b)

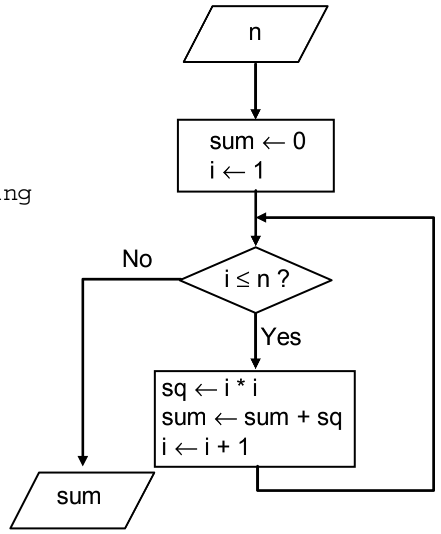


Figure 4-2. Pseudocode and flowchart for an algorithm that calculates $1^2 + 2^2 + ... + n^2$

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 4-2 the algorithm iterates *n* times through three steps, incrementing *i* by 1 in each iteration.

The second property of a good algorithm is that it is rather general. The brute-force "algorithm" in Figure 4-1 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 4-2, a truly usable algorithm, works for any *n* without changing anything. The value of *n* serves as 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 4-3 shows how the same algorithm might be coded in Pascal and C++ functions and in a Java *class* with the `addSquares` *method*.

In Pascal

```
function addSquares(n : integer)
        : integer;
  var
    i, sum : integer;
  begin
    sum := 0;
    for i := 1 to n do begin
      sum := sum + i * i
    end;
    addSquares := sum;
  end;
```

In Java

```
public class MyMath
{
  public static int
           addSquares(int n)
  {
    int sum = 0;

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

In C or C++

```
int addSquares(int n)
{
    int i, sum = 0;

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

Figure 4-3. Pascal, C++, and Java, implementations of the sum-of-squares algorithm

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 4-2, 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 $(i+1)^2 = i^2 + (2i+1)$ and $[2i+1] = [2(i-1)+1] + 2$. If we keep the value of $2i + 1$ in a variable *k*, we can update the value of *k* and the value of *sq* without multiplications (Figure 4-4).

```
Input: n

sum ← 0
sq ← 1
i ← 1
k ← 3

Repeat the following four
steps while i ≤ n:
  sum ← sum + sq
  sq ← sq + k
  k ← k + 2
  i ← i + 1

Output: sum
```

Figure 4-4. A version of the sum-of-squares 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:

Iteration #	**0**	**1**	**2**	**3**	**...**
i	1	2	3	4	...
k	3	5	7	9	...
sq	1	4	9	16	...
sum	0	1	5	14	...

4.3 Iterations

As we have seen in the previous example, the ability to iterate over the same sequence of steps is crucial for creating compact algorithms. The same instructions are executed on each iteration, but something must change. Otherwise the iterations would never stop and the program would go into an "infinite loop." What actually changes are the values of at least some of the variables, until a specified condition is met, making the iterations stop.

High-level programming languages usually support iterations in one form or another. Java has a "while" loop —

```
while (<this condition holds>)
{
  ... // do something
}
```

— as in:

```
while (i <= n)
{
  sum += i * i;   // add i * i to sum
  i++;            // increment i by 1
}
```

There is also a "for" loop —

```
for (<initial setup>; <as long as this condition holds>;
                  <adjust variable(s) at the end of each iteration>)
{
  ... // do something
}
```

— as in:

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

We will discuss the complete syntax rules for Java's iterative statements in Chapter 8.

❖ ❖ ❖

When you analyze loops, it is of course important to understand how the values of variables change. But it is also important to see what stays the same.

A *loop invariant* is a relationship among the variables that is relevant to the purpose of the loop and that holds before the loop is entered and after each iteration through the loop.

Loop invariants help us ascertain that the loop is doing exactly what it is supposed to do. For example, you might notice that the algorithm in Figure 4-4 has two loop invariants: $k = 2i + 1$ and $sq = i^2$. In fact, this algorithm is specifically designed to have these invariants. Knowing these loop invariants helps us see that our algorithm works.

4.4 Recursion

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.

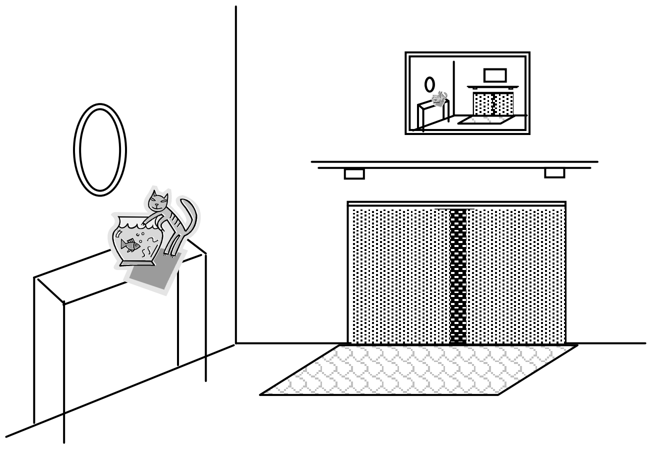


Figure 4-5. Some of the graphics elements are "primitives"; others are "pictures"

Consider the picture in Figure 4-5. 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... The overall complexity of the "whole" picture is greater than the complexity of each picture inside it, but the pictures inside 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.

Recursion is a very powerful method for describing and implementing algorithms. 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 the *base* or *stopping cases*) when a computation is obvious and recursion is not required.

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 $n = 1$.

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).

Recursion may seem very tricky to an untrained eye, and some people never completely come to grips with it. But it is easy for computers.

A computer program handles recursion as a form of iterations, which however are hidden from the programmer.

In high-level languages, such as Java or C++, recursion is implemented by means of functions calling themselves. In Java, for instance, the above recursive algorithm can be implemented as shown in Figure 4-6.

Computers have a hardware mechanism, called the *stack*, that facilitates recursive function calls. It is described in Chapter 21. Take a look at it now if you want to learn the technical details. At this point, suffice it to say that the compiler knows how to handle recursive functions without getting confused.

```
public class MyMath
{
  public static int addSquares(int n)
  {
    if (n == 1)
      return 1;
    else
      return addSquares(n-1) + n * n;
  }
}
```

Figure 4-6. Recursive method in Java for computing $1^2 + 2^2 + ... + n^2$ (not recommended for this task)

Some programming languages, such as LISP or Scheme, almost always suggest the use of recursion instead of explicit iterations. In Java, recursion is used in standard packages but programmers do not have to program recursive functions themselves very often. Still, knowing how recursion works is very useful for understanding certain common algorithms such as Mergesort (Section 13.7) or dealing with branching or nested structures (such as the structure of pictures within pictures in the above example).

You may notice that we said a recursive algorithm was "not recommended" for solving the sum-of-squares problem. Some people, once they understand it, find recursion so elegant that they are tempted to use it everywhere. Indeed, as Figure 4-6 shows, 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.

4.5 *Case Study:* Euclid's GCF Algorithm

Figure 4-9 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 replace the greater number with the difference of the two numbers, the greatest common factor remains the same. In other words, if $a > b$, then

$$\text{GCF}(a, b) = \text{GCF}(a - b, b).$$

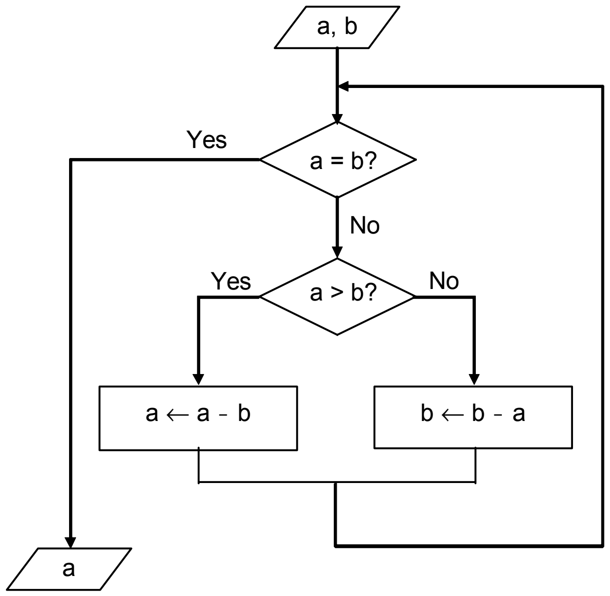


Figure 4-7. Euclid's algorithm for finding the greatest common factor of two positive integers

Let us take an example, $a = 30$, $b = 42$, and 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 4-8. 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.

In a recursive implementation (Figure 4-9), the while loop is replaced by recursive calls. But first we have to check for the base case, $a = b$. In that case the GCF is obviously *a* (or *b*). Again, the parameters passed to the recursive calls remain positive and the larger of them is reduced, so the recursive calls converge to the base case.

```
public class MyMath
{
  ...

  /**
   * Returns GCF of a and b (iterative)
   * precondition: a > 0, b > 0
   */
  public static int gcf(int a, int b)
  {
    while (a != b) // while a not equal to b
    {
      if (a > b)
        a -= b;    // subtract b from a
      else
        b -= a;    // subtract a from b
    }

    return a;
  }
  ...
}
```

Figure 4-8. Euclid's Algorithm in Java

```
public class MyMath
{
  ...

  /**
   * Returns GCF of a and b (recursive)
   * precondition: a > 0, b > 0
   */
  public static int gcf(int a, int b)
  {
    if (a == b) // base case: if a equals b
      return a;

    if (a > b)
      return gcf(a - b, b);
    else // if b > a
      return gcf(a, b - a);
  }
  ...
}
```

Figure 4-9. Recursive implementation of Euclid's Algorithm in Java

There is an operator in Java, `a % b`, called the *modulo division operator*, that computes the remainder when `a` is divided by `b`. For example:

```
21 % 9 = 3 ( = 21 - 2 * 9 )
9 % 21 = 9 ( = 9 - 0 * 21 )
```

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 can use the modulo operator to make the code for both `gcf` methods more efficient.

4.6 Working with Lists

We deal with lists all the time — lists of things to do, shopping lists, guest lists, restaurant menus, and, sometimes, the proverbial "laundry lists." Computers deal with lists, too: lists of files, lists of words, lists of bank or credit card transactions, lists of people, students, courses, customers, subscribers, departing and arriving flights, and, yes, dry cleaning orders.

In abstract terms, a list is a data structure in which the items are numbered and we know how to get to the *i*-th item. One of the basic operations on a list is *traversal*, which means processing all the items in order (for example, printing out the list). A list traversal algorithm is straightforward:

```
Start with the first item
While more items remain,
  process the next item
```

Such an algorithm presumably maintains a variable that refers to the "current" item in the list and shifts that variable to the next item once the current item has been processed. Java (starting with release 5.0) has a so-called "for-each" loop (also called the "enhanced for loop") that automates this process. For example:

```
    for (String word : list) // for each word
                             // (character string) in list
    {
      System.out.println(word);  // print word on a separate line
    }
```

We humans can process short lists instantaneously. You can glance at a guest list for a small birthday party and immediately see whether you are invited or not. However, a computer needs to examine each item in the list to find a given target value or to ascertain that that value is not in the list. This type of process is called *Sequential Search*. If a list has 1,000,000 items arranged in random order, it will take, on average, 500,000 comparisons to find a particular item, and, if the target value is not in the list, then the computer will need to scan all 1,000,000 items to make sure that is so. But if the list items are arranged in order (for example, words in alphabetical order or numbers in increasing order), there is a much more efficient search algorithm, called *Binary Search*.

The idea of Binary Search is familiar to anyone who has played the number-guessing game. Your friend thinks of a number between 1 and 100. You need to guess the number by asking questions like "Is your number greater than *x*?" How many questions, in the worst case, will you need to ask to guess the number? Seven, right? If I pick a number between 1 and 1,000,000, then you'll need at most 20 questions!

The Binary Search algorithm in an ordered list is similar. Start looking for the target value at the midpoint of the list (Figure 4-10). If your target is greater than the middle item, continue searching in the right half; if it is less than the middle item, continue searching in the left half. Keep dividing the remaining search range into approximately equal halves until you find the target value or there is nothing left to search. This type of algorithm is called "divide and conquer." We will return to the Binary Search algorithm and its implementation in Java in Chapter 13.

In the Sequential Search algorithm, the number of comparisons in the worst case is roughly proportional to n, where n is the length of the list. In Binary Search, the number of comparisons is proportional to $\log_2(n)$. Suppose one comparison takes T_s time in Sequential Search and T_b time in Binary Search. Even if $T_b > T_s$, the Binary Search algorithm is still considered much better than Sequential Search because, for large n,

$$\frac{T_b \cdot \log(n)}{T_S \cdot n} \rightarrow 0$$

(approaches 0) regardless of the values of the constant factors T_s and T_b. It is said that Sequential Search is an $O(n)$ algorithm and Binary Search is an $O(\log n)$ algorithm. The "big-O" notation and its use for comparing algorithms is explained in Chapter 18.

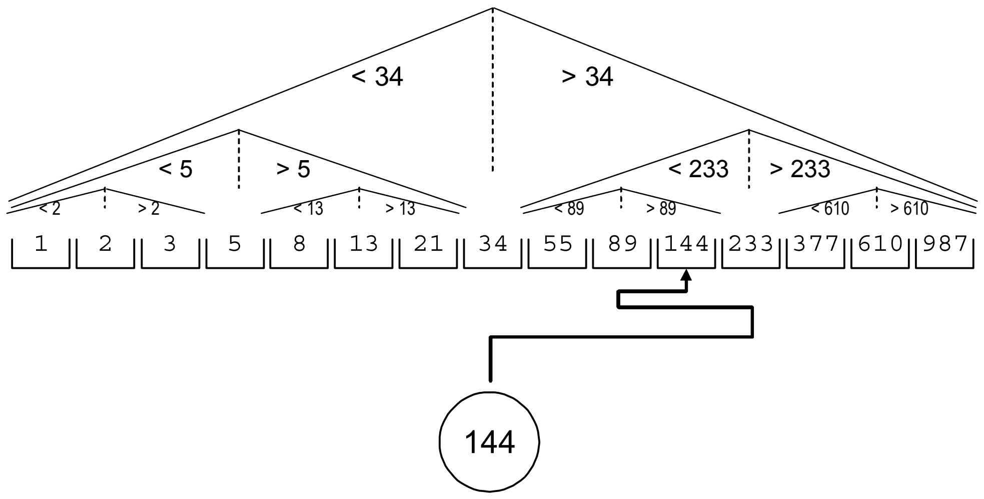


Figure 4-10. A Binary Search example

Another common task is arranging the list items in order (provided we have some method for comparing the items). This task is called *sorting*. Here is a simple algorithm for sorting a list of items:

```
Set n to the total number of items in the list
While n is greater than 1 repeat the following steps:
    Find the largest item among the first n items
    Swap that largest item with the n-th item
    Decrement n by 1
```

This algorithm is called *Selection Sort* (because we repeatedly select the largest item among the first *n* in the list). To better see how this works, you can enact this algorithm with a group of friends, arranging them, say, in order of their birthdays. Again, this is just a quick preview. A more detailed discussion of this and other sorting algorithms and their implementations in Java has to wait until we get to Chapter 13.

4.7 *Case Study:* File Manager

Recursion is especially suitable for handling nested structures. Consider for example the file system on your computer (Figure 4-11). 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 and choose "Properties," you will see a screen that shows the total number of files in the folder and all its subfolders and the total number of bytes that this folder and all its files and all its subfolders take. How does your operating system do that?

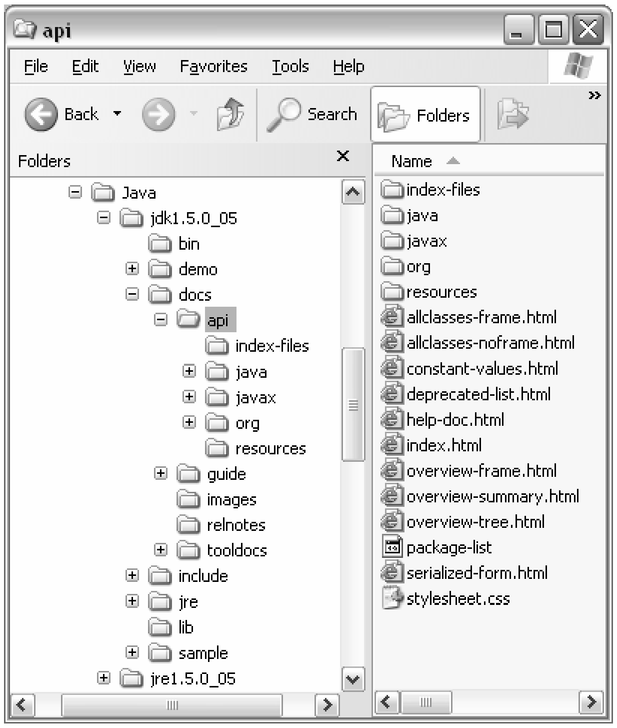


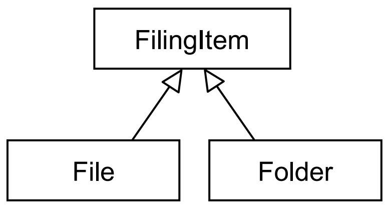
Figure 4-11. A snapshot of the *Windows Explorer* screen

It is not easy to come up with an iterative algorithm to scan through all the files in all subfolders. 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 pseudocode:

```
function  fileCount(folder)
{
  count ← 0

  for each item in folder
  {
    if this item is a file
      count ← count + 1
    else (if this item is a folder)
      count  ← count + fileCount(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. We can define a small hierarchy of classes:



A `FilingItem` object can be a `File` or a `Folder`. As we say in OOP, a `File` IS-A (is a) `FilingItem` and a `Folder` IS-A `FilingItem`. A `Folder` also HAS-A list of `FilingItem`s:

```
public class Folder extends FilingItem
{
  private List<FilingItem> items;
  ...
}
```

Let us define a `fileCount` method for both a `File` and a `Folder`. These methods have the same name but work differently. For a `File`, `fileCount` simply returns 1:

```
public class File extends FilingItem
{
  ...

  public int fileCount()
  {
    return 1;
  }
  ...
}
```

For a `Folder`, `fileCount` is recursive: it returns the sum of all counts for all items in its `items` list:

```
public class Folder extends FilingItem
{
  private List<FilingItem> items;
  ...
  public int fileCount()
  {
    int sum = 0;

    for (FilingItem item : items)     // for each item in items
      sum += item.fileCount();        // add item's file count to sum

    return sum;
  }
  ...
}
```

Java will automatically call the correct `fileCount` method for each item in the `items` list, be it a `File` or a `Folder`. This feature is called *polymorphism*. Polymorphism is discussed later, in Chapter 11.

4.8 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 methods for describing algorithms.

The three major devices used in algorithms are *decisions*, *iterations*, and *recursion*. A decision allows the algorithm to take different paths depending on the result of the previous operation. Iterations and/or recursion help fold the task into one or several mini-tasks that are repeated multiple times. Any high-level programming language supports decisions, iterations, and recursion in some form. Java, for example, has the `if` statement for decisions, the `while` and `for` loops for iterations, and the liberty for a method to recursively call itself.

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 until the values of the variables meet a specified condition, making the iterations stop.

A relationship between variables that is relevant to the purpose of the loop and that holds before the first iteration and after each iteration is called *loop invariant*. Loop invariants are helpful for proving that the code is correct.

Iterations are essential for handling *lists*. A list is a data structure in which the items are numbered and we know how to get to the *i*-th item. List traversal is a procedure in which we process all elements in the order in which they occur in the list. A sequential search for a particular value in a list of 1,000,000 elements, arranged in random order, takes, on average, 500,000 comparisons, if the target value is in the list, or 1,000,000 comparisons to establish that the target value is not in the list. For a list whose elements are arranged in some order (such as words in alphabetical order or numbers in increasing order) there is a much more efficient "divide and conquer" algorithm, called *Binary Search*. A binary search in a list of 1,000,000 elements needs only 20 comparisons!

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 similar but smaller structures, and a recursive definition of a procedure that performs a certain task relies on applying the same procedure to a similar but smaller task. A recursive function calls itself, but the parameters passed to this recursive call must be somehow "reduced" with each call. A recursive function 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.

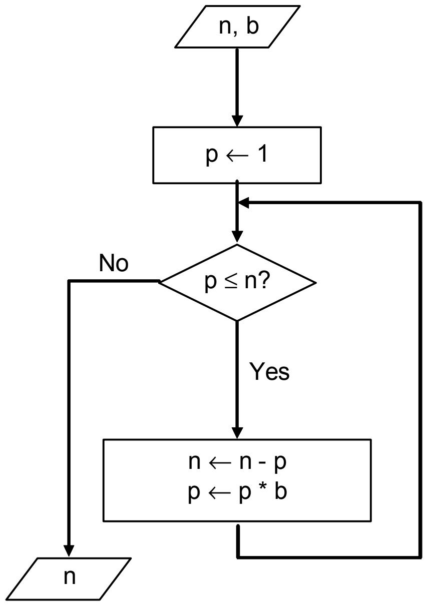
Exercises

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*. (This sum converges to $\frac{\pi^2}{6}$ as *n* increases.) ✓

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. (This sum converges to the natural logarithm of 2.) ≧

3. 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. Show your algorithm in pseudocode or draw a flowchart. ✓

4. What is the output from the algorithm shown in the flowchart below when the input is $n = 37$, $b = 2$?



5.■ Recall that $1 + 3 + ... + (2p - 1) = p^2$ for any integer $p \geq 1$. Using this property, come up with a simple iterative algorithm that finds out whether a given number is a perfect square. Your algorithm can compare numbers and use addition but don't use any other arithmetic operations. Show your algorithm in pseudocode or draw a flowchart.

6.■ 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();

    < missing code >
  }
}
```

7.■ In a bag of *n* coins, one is radioactive. In one trial we are allowed to measure the radioactivity of any pile of coins. (One coin makes the whole pile test positive for radioactivity.) Describe an efficient algorithm for finding the radioactive coin in the bag. How many trials are needed to find the radioactive coin in a bag of 1000 coins using your algorithm? ✓

8.◆ Suppose you have a precise balance and three coins. If two of the coins are of the same weight and the third one is lighter (a fake), it takes only one trial to identify the lighter one. How many trials are needed to find the lighter coin among 81 coins if the rest are all of the same weight? ✓

9. Consider the following recursive method:

```
  public int mysterySum(int n)
  {
    if (n == 0) // if n equals to 0
      return 0;
    else
      return 3 + mysterySum(n - 1);
  }
```

What value is returned when `mysterySum(5)` is called? ✓

10.■ The questions below are paper-and-pencil exercises. You can use pseudocode or Java, but don't worry about the correct Java syntax.

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 this method, write a sketch of an iterative method `printTriangle` (or pseudocode) 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

```
*
**
***
****
*****
```

(b) Write a recursive version of the `printTriangle` method, without iterations.

(c) Modify the recursive version of `printTriangle`, so that `printTriangle(n)` displays

```
*****
****
***
**
*
```

11.■ The following recursive method calculates 3^n:

```
// precondition: n >= 0
public int power3(int n)
{
  if (n == 0) // if n equals to 0
    return 1;
  else
  {
    int p = power3(n/2);
      // when n is odd, n/2 is truncated to an integer
      // e.g., 7/2 gives 3 and 1/2 gives 0

    p *= p;  // multiply p by itself

    if (n % 2 == 1)  // if n is odd
      p *= 3; // multiply p by 3

    return p;
  }
}
```

How many multiplications will be performed when the program calls `power3(15)`?

12. Let's pretend for a moment that Java does not support multiplication. Write an iterative version and a recursive version of the following method (in pseudocode, if you wish):

```
/**
 *  Returns the product of a and b
 *  precondition: a >= 0, b >= 0
 */

 public int product(int a, int b)
 {
   ...
 }
```

13.■ Consider the following recursive method:

```
public int 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)■ 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. $2^{\left(2^n\right)}$

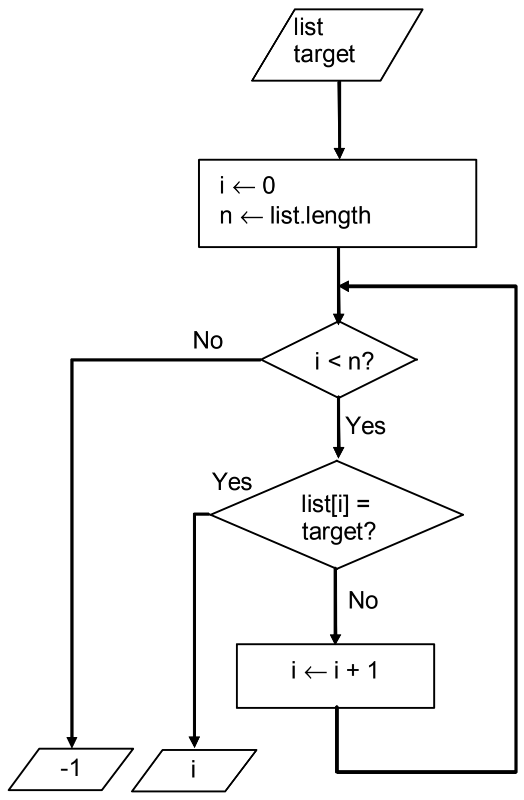
14. The numbers $\binom{0}{n}, \binom{1}{n}, \binom{2}{n} \dots \binom{n}{n}$ in the expansion

$$(x+y)^n = \binom{0}{n}x^n + \binom{1}{n}x^{n-1}y + \binom{2}{n}x^{n-2}y^2 + \dots + \binom{n-1}{n}xy^{n-1} + \binom{n}{n}y^n$$

are called binomial coefficients. For example,
$(x+y)^2 = x^2 + 2xy + y^2$, so $\binom{0}{2}=1, \binom{1}{2}=2, \binom{2}{2}=1$.
$(x+y)^3 = x^3 + 3x^2y + 3xy^2 + y^3$, so $\binom{0}{3}=1, \binom{1}{3}=3, \binom{2}{3}=3, \binom{3}{3}=1$.
$(x+y)^4 = x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4$, so
$\binom{0}{4}=1, \binom{1}{4}=4, \binom{2}{4}=6, \binom{3}{4}=4, \binom{4}{4}=1$.
Binomial coefficients have the following properties: for any positive integer n, $\binom{0}{n}=\binom{n}{n}=1$, and for any integers n and k, such that $0<k<n$, $\binom{k}{n}=\binom{k-1}{n-1}+\binom{k}{n-1}$. Complete the recursive method `binCoeff` below, which computes a specified binomial coefficient (or write it in pseudocode).

```
/**
 *  Returns the value of the binomomial coefficient (n, k)
 *  precondition: 0 <= k <= n
 */
 public int binCoeff(int n, int k)
 {
   ...
 }
```

15. The flowchart below describes the Sequential Search algorithm:



The algorithm finds the index *i* of the first element whose value is equal to *target*. If no such value is found, the algorithm returns -1. `list[i]` in the flowchart refers to the list's *i*-th element. <u>In C, C++, and Java, elements of a list are counted starting from 0</u>.

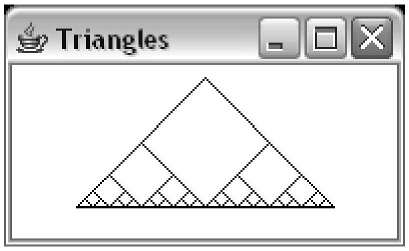
(a) Draw a flowchart for the algorithm that looks for two consecutive elements in the list, `list[i]` and `list[i+1]`, equal to each other. The algorithm should return the index of the first one, if such a pair is found, or -1, if no two consecutive values are equal.

(b)■ Draw a flowchart for the algorithm that finds the index of the element of a list with the maximum value. If several values are equal to the maximum value, the output should be the index of the first one of them.

Continued ⇨

(c)◆ Draw a flowchart for the algorithm that determines whether a list has two duplicate (equal) values. The output should be *true* if found, *false* if not found. ⋞ Hint: for each *i* set *target* to `list[i]`, and start looking for a duplicate at `list[j]` where $j = i + 1$. ⋟

16.◆ The program *Ornament* (`JM\Ch04\Exercises\Ornament.java`) displays an ornament made of nested right equilateral triangles, as shown below:



The half-length of the base of the largest triangle is 64, and the coordinates of its midpoint are (100, 100). The half-length of the base of the smallest triangle shown is 4. Fill in the blanks in the `drawTriangles` method. ⋞ Hint: a statement

```
g.drawLine(x1, y1, x2, y2);
```

draws a line in the graphics context `g` from point `(x1, y1)` to point `(x2, y2)`. ⋟

```
/**
 *    Chapter 5
 */
```

Java Syntax and Style

5.1 Prologue

In Chapter 3 we discussed the main ideas of OOP methodology and worked through the general structure of a small program. There's much more than that, however, to writing working Java code. 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 in algorithms: 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.

In the following sections 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.

5.2 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.

The second format is to place a comment after a double slash mark 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 <...>, *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 (<u>H</u>yper<u>T</u>ext <u>M</u>arkup <u>L</u>anguage) 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 an important element (a class, a method, or a field) in order to be processed by *Javadoc*.

Documentation 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 applet.
 *  <p>
 *  All <code>MyMath</code> methods work with real numbers of
 *  the <code>double</code> type.
 *
 *  @author      Al Jibris
 */
```

Or:

```
  /**
   *  Returns the greatest common factor of positive integers
   *  <code>a</code> and <code>b</code>.
   *  <p>
   *  (This is a recursive implementation of Euclid's Algorithm.)
   *
   *  @param  a  the first number
   *  @param  b  the second number
   *  @return    the greatest common factor of a and b
   */
  public int gcf(int a, int b)
  {
    ...
  }
```

Note how HTML formatting tags may be embedded in a documentation 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 Sun'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.

5.3 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 5-1 shows a list of the Java *reserved words*, loosely organized by category.

Primitive 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`	
`double`	Control statements:	`this`	Not used in this book:
`boolean`		`super`	
`void`	`if`	`abstract`	`continue`
`enum`	`else`	`instanceof`	`package`
	`for`		`native`
Built-in constants:	`while`		`volatile`
	`do`		`transient`
`true`	`switch`		`synchronized`
`false`	`case`		`assert`
`null`	`default`		
	`break`		
	`return`		

Figure 5-1. 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 5-2 shows fragments of the `HelloGui` class with all the reserved words highlighted.

```
/**
 *  This program displays a message
 *  in a window.
 */

import java.awt.*;
import javax.swing.*;

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

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

Figure 5-2. 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 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 her own Java classes, their fields and methods, and 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. Names of "universal" or important constants may use all caps.

For example:

```
public class VendingMachine            // class name
{
  private int depositedAmount;         // variable
  ...
  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 usually 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 for a short time, 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 <...>, *The 17 Bits of Style.*

5.4 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 5-3. 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 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.*; import javax.swing.*; public
class HelloGui extends JFrame { public HelloGui() {
super("Hello World with GUI"); Container c=
getContentPane();c.setBackground(Color.WHITE);
c.setLayout(new FlowLayout()); c.add(new JLabel
("Hello, GUI!")); } 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(EXIT_ON_CLOSE);
window.setVisible(true); }}
```

Figure 5-3. `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 six-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 `paint`'s name in the `HelloGraphics` class:

```
public void pain(Graphics g)
{
  < ... code >
}
```

The class still compiles fine and the program runs, but instead of redefining the `paint` 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 only the gray background.

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 6 in the `Banner` class (Figure <...> on page <...>).

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;


         ...
 Line 5:  public class Banner extends JApplet
 Line 6:  //  accidentally omitted implements ActionListener
          ...
 Line 16:     Timer clock = new Timer(30, this);
```

When we compile the program, the compiler can tell that something is not right and reports an error on Line 16:

```
 C:\Mywork\Banner.java:16: cannot find symbol
 symbol  : constructor Timer(int,Banner)
 location: class javax.swing.Timer
     Timer clock = new Timer(30, this);
                   ^
 1 error
```

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

Appendix <...>* lists a few common compiler error messages and their causes.

Notwithstanding the compiler's somewhat limited capacity to pinpoint your syntax errors, you can never blame the compiler for errors. You may be sure that there is something wrong with your code if it 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 "syntax" 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 i, sum = 0;
  for (i = 1; i <= n; i++);
    sum += i * i;
  return sum;
}
```

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

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

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

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

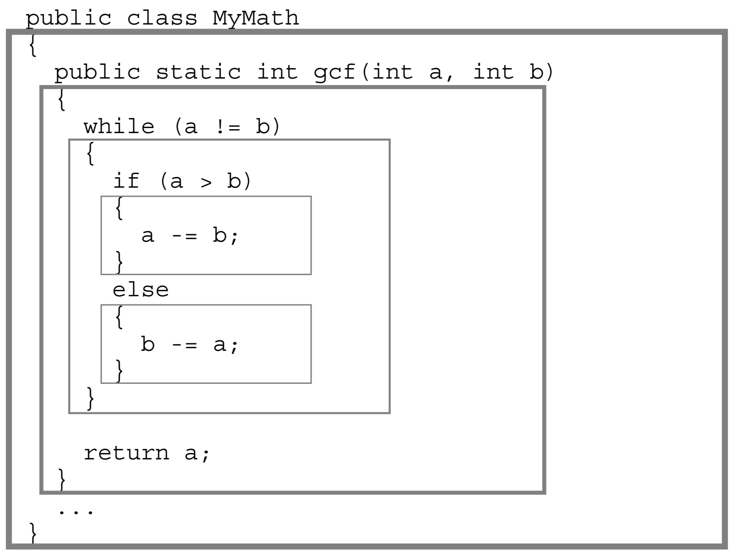
You think your code will iterate *n* times through the `sum += ...` statement. Guess what: instead it will iterate *n* times through nothing, an empty statement. As a result, `addSquares(5)` will return 36, rather than 55.

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

5.5 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 certain array declarations as explained in Chapter 12).



```
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 5-4. Nested blocks, marked by braces, within a class

Braces divide the code into *nested blocks* (Figure 5-4). Statements inside a block are usually indented by a fixed number of spaces or one tab. 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.

5.6 *Lab:* Correcting Syntax Errors

Figure 5-5 shows a Java program that is supposed to display an orange disk moving across the "sky." However, the code in Figure 5-5 has several syntax errors. Find and fix them. Do not retype the program — just copy `MovingDisk.java` form `JM\Ch05\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.

```java
import java.awt.*;
import java.awt.event*;
import javax.swing.*;

public class MovingDisk extands JFrame
  implements ActionListener
{
  private int time;

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

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

    Color sky;
    if (y > 150) sky = Color.Black
    else sky = Color.Cyan;
    Container c = getContentPane();
    c.setBackground(sky);
    super.paint(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)
  {
    MovingDisk w = new movingDisk();
    w.setSize(300, 150);
    w.setDefaultCloseOperation(EXIT_ON_CLOSE);
    w.setResizable(false);
    w.setVisible(true);
  }
}
```

Figure 5-5. `JM\Ch05\Syntax\MovingDisk.java` **(with syntax errors)**

First 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.

When your program finally compiles, try to run it. If you get an "exception" error message, try to get some clues from it, look at the code again carefully, and fix the problem.

5.7 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 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 "documentation" comments help the *Javadoc* program automatically generate documentation in the HTML format.

A program's text contains some *reserved words*, 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

1. Name three good uses for comments.

2. Add a *Javadoc*-style comments to the `paint` method in the `MovingDisk` class from Section 5.6 (`JM\Ch05\Syntax\MovingDisk.java`).

3. Consider the *Moving Disk* program.

(a) Find the fifteen different reserved words used in `MovingDisk.java`. ✓

(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* applet (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 = c.getWidth();
}
```

Try to compile and run the applet and explain the result. Why does it compile with no errors? Why is the message not moving across the screen?

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

```
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\Ch05\Exercises\Morning.java`). Compile and run the program. Note that this program uses another class, `EasySound`, whose source file, `EasySound.java` (from `JM\Ch05\Exercises`) should be added to your project. Copy the file `roost.wav` into the folder where your IDE places the compiled classes.

(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.

int chapter = 6;

Data Types, Variables, and Arithmetic

6.1 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. As we mentioned earlier, 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;
```

<u>does not</u> represent an algebraic equality, 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 <u>not</u> 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`, and `char`) or a class type (such as `String`, `Foot`, `Walker`). The programmer specifies the variable's data type based on the kind of data it will contain. A variable's data type determines the amount of space allocated for it in memory and the kind of operations that can be done to it. A variable of type `int`, for example, contains an integer value; 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 memory. When an object is created with the `new` operator, that operator 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 Chapter 9.

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 s*cope* of variables and symbolic constants
- Java's arithmetic operators

6.2 Declaring Fields and Local Variables

Variables in Java fall into three main categories: *fields* (which are also called *instance variables*), *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 fragments of code in Figure 6-1 show several declarations of fields in the `Walker` class and a constructor that sets their values.

```
public class Walker
{
  private Foot leftFoot, rightFoot;
  private int stepLength;
  private int stepsCount;

  // Constructor
  public Walker(int x, int y, Image leftPic, Image rightPic)
  {
    stepLength = Foot.LENGTH * 3;
    leftFoot =  new Foot(x, y - Foot.LENGTH / 2, leftPic);
    rightFoot = new Foot(x, y + Foot.LENGTH / 2, rightPic);
  }

  ...
}
```

Declarations of fields

Figure 6-1. A fragment from the `Walker` 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 and `null` for objects.

For example:

```
   private int stepsCount;   // stepsCount is set to 0
   private Foot leftFoot;    // leftFoot 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 `setup` method in `WalkingGroup` uses two local variables, `x` and `y`:

```
   // Sets up this group of participants
   public void setup(int floorDir, Dance steps1, Dance steps2)
   {
     int x = 2 * Foot.LENGTH;
     int y = danceFloor.getHeight() / 2;
     if (floorDir == 0)
     ...
 }
```

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 are not used in declarations of local variables.

Local variables must be declared and assigned a value before they 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.

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 stepLength;
Foot leftFoot;
Walker amy;
```

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

```
Foot leftFoot, rightFoot;
int stepLength, stepsCount;
double x, y, z;
```

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

```
int stepLength = Foot.LENGTH * 3;
EasySound beat = new EasySound("beat.wav");
```

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

```
private final int stepLength = Foot.LENGTH * 3;
```

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 6.6).

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

```
Walker person = new Walker (x, y, leftShoe, rightShoe);
```

instead of

```
person = new Walker (x, y, leftShoe, rightShoe);
```

in the `setup` method, the compiler will treat this statement as a declaration of a local variable `person` and leave the field `person` uninitialized (`null`). When later another method attempts to call one of `person`'s methods, 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 its object. 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.

6.3 Primitive Data Types

In Java, values of primitive data types are not objects. This is a concession to more traditional 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, and 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 (as opposed to some other languages, such as C++) each data type has a fixed size, regardless of the particular computer model or brand of Java interpreter. Table 6-1 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`.

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

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 6-1. The primitive data types

6.4 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 10 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 operators to objects. Strings are an exception: when + is applied to two strings, it concatenates them. For example,

```
String str = "Chapter" + "6";
```

is the same as

```
String str = "Chapter6";
```

6.5 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.            (doubles)
"leftshoe.gif", "1776", "y"     (Strings)
```

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."

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: `""`.

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

```
    System.out.print("\nDon\'t let me down\nDon\'t let me down\n");
                      ^^   ^^             ^^   ^^             ^^
```

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

❖ ❖ ❖

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

```
      public final int LENGTH = 48;
      private final int stepLength = Foot.LENGTH * 3;
```

(Sometimes, programmers write names of symbolic constants using all capital letters for better visibility.)

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 some expression.

In Java, a constant doesn't have to be initialized right away in its declaration: its "final" value can be set in a constructor.

For example:

```
   private final int stepLength;
   ...

 // Constructor
 public Walker(int x, int y, Image leftPic, Image rightPic)
 {
   stepLength = Foot.LENGTH * 3;
   ...
 }
```

It may seem, at first, that symbolic constants are redundant and we can simply use their literal values throughout the program. For example, instead of writing

```
  private final int DELAY = 30;
    ...
    t = new Timer(DELAY, this);
```

we could simply write

```
    t = new Timer(30, this);
```

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 quick tweak to the constant declaration will change the constant's value throughout the program.

Consider, for instance, the code that we put together for constructing a `Walker` object:

```
  // Constructor
  public Walker(int x, int y, Image leftPic, Image rightPic)
  {
    leftFoot =  new Foot(x, y - Foot.LENGTH, leftPic);
    rightFoot = new Foot(x, y + Foot.LENGTH, rightPic);
    stepLength = Foot.LENGTH * 3;
  }
```

Surely we could write the same code with a specific number 24 plugged in for `Foot.LENGTH`:

```
    ...
    stepLength = 72;
    leftFoot =  new Foot(x, y - 24, leftPic);
    rightFoot = new Foot(x, y + 24, rightPic);
    ...
```

At a first glance it might seem simpler. But if we needed to adjust the scale of the foot, we would have to change all these numbers. A programmer making the change would have to figure out what all the numbers mean and recalculate them. Luckily, in this case all the statements that depend on the size are grouped together, so at least they are not too hard to find. It could be much worse: they could be 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 defined 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

```
 final int semiCircleDegrees = 180;    // pedantic
 final int HoursInDay = 24;
```

6.6 Scope of Variables

As we have discussed above, each 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 <u>conspicuous</u> names.

`x`, `y`, or `a` are bad choices — they should be saved for throwaway local variables. `person` and `stepsCount` 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 <u>never</u> try to economize on declarations of temporary local variables within methods by making them fields. Everything should be declared where it belongs.

6.7 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 <u>types</u> 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`, 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 towards 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 `ints` into `doubles`. 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`, 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* (i.e., 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 above:

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

Your code will be more robust and 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)(Math.random() * 6);
    // 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 .5 to a positive number or subtract .5 from a negative number first, and then cast it into an `int`. For example:

```
    int pieChartDegrees = (int)((double)count / totalCount * 360 + .5);
```

or

```
    int pieChartDegrees = (int)(360.0 * count / totalCount + .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 6.9.

6.8 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 += i * i;
```

is the same as:

```
    sum = sum + i * i;
```

The `+=` form may seem cryptic at the beginning, but, once you get used to it, it becomes attractive — not only because it is more concise, but also because it emphasizes the fact that the same variable is being modified. The latter form 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 = "leftshoe";
 fileName += ".gif";     // fileName now refers to "leftshoe.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.

6.9 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 other operand is 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 <u>object</u> 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 hexadecimal address in memory. Something like this: `"Foot@11a698a"`. Programmers often override the default and provide more useful `toString` methods for their classes. 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 + " = " + 1.0 / 2);
```

display

```
1/2 = 0.5
```

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,

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

is equivalent to

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

6.10 *Lab:* Pie Chart

Figure 6-2 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.

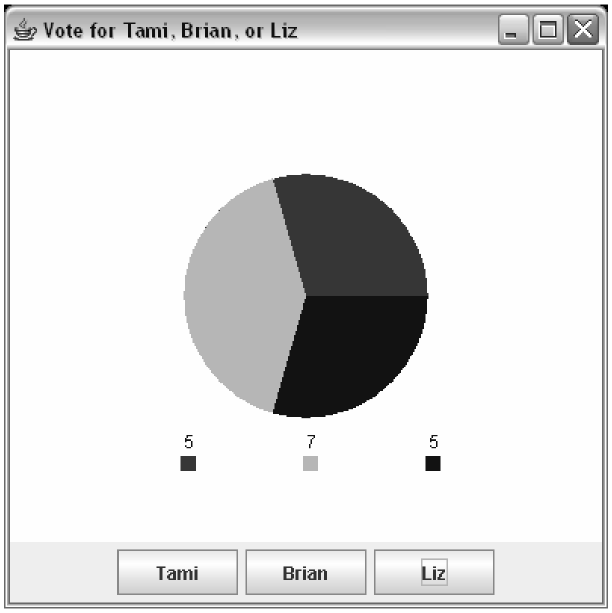


Figure 6-2. The *Poll* program

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\Ch06\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. For example, the following method

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

   should display

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

4. Compile the `PollDisplayPanel` class and fix the syntax errors, if any.

5. Create a simple test class with a `main` method similar to the one shown in the above example. Compile and run your test program.

6. 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.

7. Fill in the blanks in the `drawPieChart` and `drawLegend` methods.

8. Compile and test the *Poll* program.

6.11 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.

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 *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 9).

Java has `byte`, `int`, `short`, and `long` *primitive data types* for representing integers of various sizes. We will 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 7.

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.

Exercises

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) `i` 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. Which one of the following statements prints a backslash on the screen?

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. 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);`

7. (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`?

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

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

9. 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));
```

10. If `double x` has a negative value, write an expression that rounds `x` to the nearest integer.

11. Given

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

write 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`. ⋟ ✓

12.■ 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;
...
```

✓

13.■ 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. ✓

14.■ 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).

15.■ `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.");
```

16. 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\Ch06\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.

17.■ 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 costs for a given number of binders:

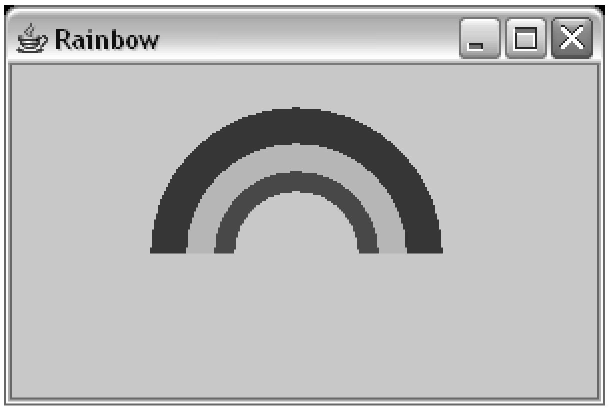
```
public double computeShippingCost(int nJars)
{
  int nCartons = (nJars + 11) / 12;

  int totalOunces = ________________________ ;

  int lbs = ___________________________ ;

  return ______________________________ ;
}
```

18.■ 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 16 into a dog-to-human-age converter). ✓

19.◆ 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\Ch06\Exercises`.

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

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

3. Find and fix three syntax errors in `Rainbow.java` so that it compiles with no errors.

Continued

4. 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). In `Rainbow`'s constructor, set the window's background to `skyColor` rather than `Color.WHITE`.

5. In the `paint` method, declare <u>local</u> 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 content pane, 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 `paint` method to draw the largest (red) semicircle. Test your program.

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.

10. Test your program.

`if (chapter == 7)`

Boolean Expressions and `if-else` Statements

7.1 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 jumps 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.

In Sections 7.9 and 7.12 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 some code for this case study with a lot of `if-else` statements in it.

7.2 `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`.

7.3 `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;
```

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

```
boolean over21;

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

7.4 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 and `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 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

```
15.0 / 3.0 == 5.0
```

the numbers on the left and right may be very close but not exactly equal due to rounding errors.

❖ ❖ ❖

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

For example, in

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

the `==` operator 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 <u>values</u> may be the same. As we'll explain in Chapter 10, 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)) ...
```

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.

7.5 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 <u>or</u> not games (or neither)."

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))
```

7.6 Order of Operators

In general, all unary operators have higher precedence then 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!
```

as opposed to:

```
     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:

Highest	`!` (unary)- (cast) `++` `--`
↑	`*` `/` `%`
	`+` `-`
	`<` `<=` `>` `>=` `==` `!=`
↓	`&&`
Lowest	`||`

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!

7.7 Short-Circuit Evaluation

In the binary logical operations `&&` and `||`, 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 expression *`condition1`* `&&` *`condition2`* is always false, no matter what the value of *`condition2`* is. If *`condition1`* is true, then *`condition1`* `||` *`condition2`* is always 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 `boolean`s, 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 17.

7.8 `if-else-if` and Nested `if-else`

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

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

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 "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:

```
         •
       /   \
    if(…)  else
     /       \
    •         •
   / \       / \
if(…)else if(…)else
 /     \   /     \
```

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
{
  if (accidents)
    surcharge = 1.1;  // Surcharge 10%
  else
    surcharge = .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 intractable. 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;
                    //              }
```

7.9 *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 usually represented by dots (Figure 7-1).

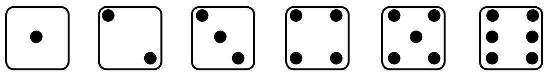


Figure 7-1. Dots configuration on a die

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 in the `Craps.jar` file in `JM\Ch07\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 7-2). The program allows you to play as many games as you want.

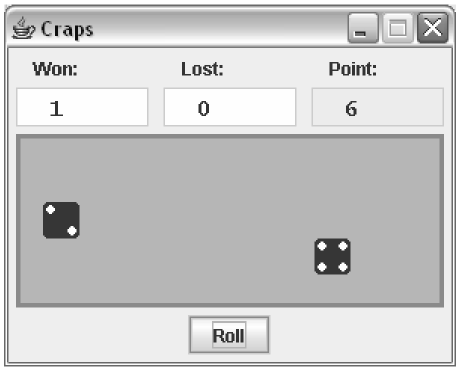


Figure 7-2. The *Craps* program

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 7-2) 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 7-3).

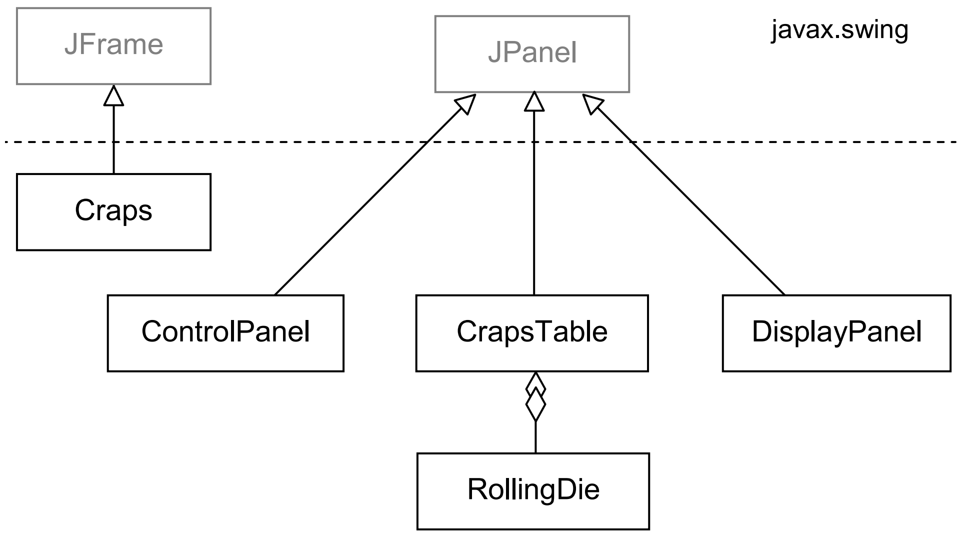
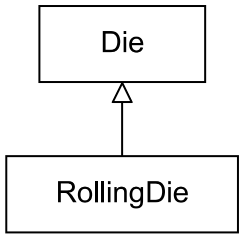


Figure 7-3. GUI classes in the *Craps* program

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 <u>extend</u> `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 7-4 shows the overall design for the *Craps* program that I have come up with.

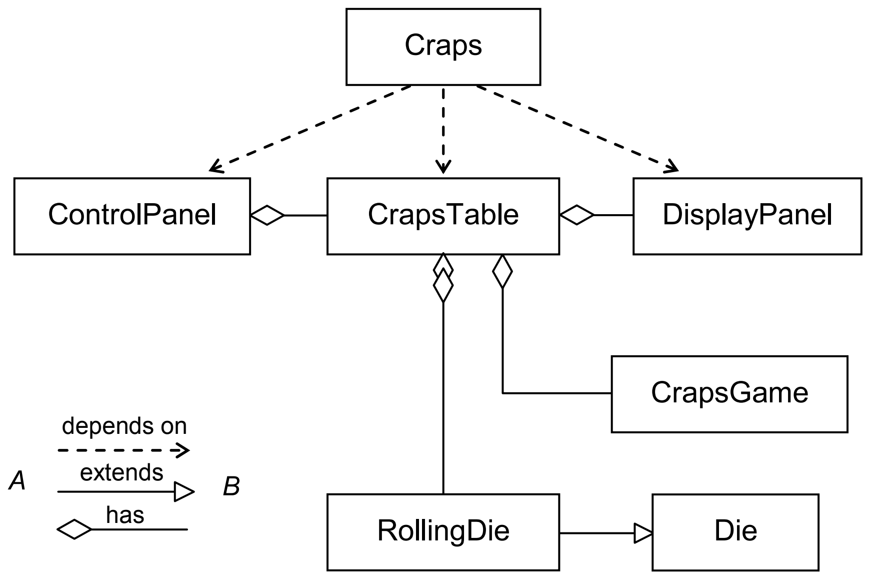


Figure 7-4. *Craps* classes and their relationships

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. We will talk about design patterns in Chapter 26. 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 <u>cohesive</u>.

Second, dependencies between classes should be minimized. The reason we can draw a class diagram in Figure 7-4 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 <u>minimized</u>.

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 7-5 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() { ... }
```

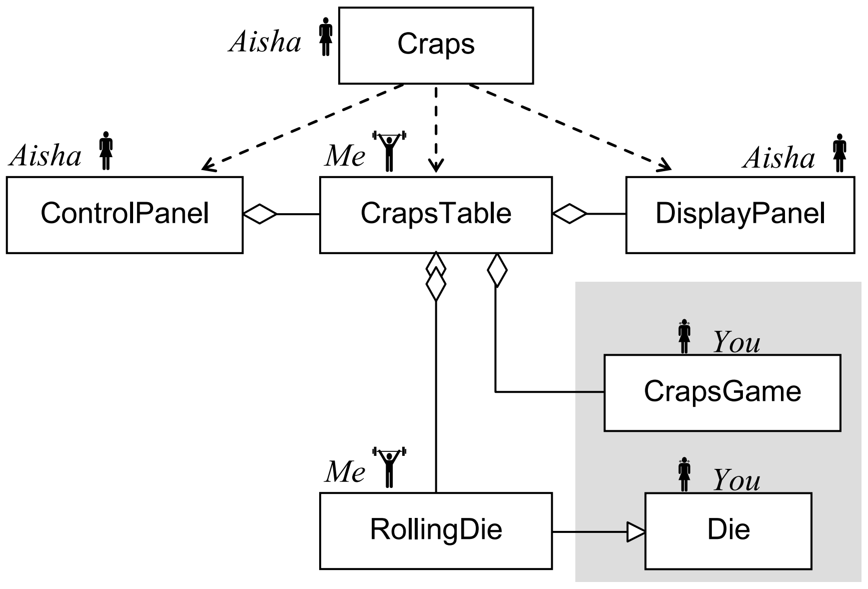


Figure 7-5. Task assignments in the *Craps* project

❖ ❖ ❖

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:

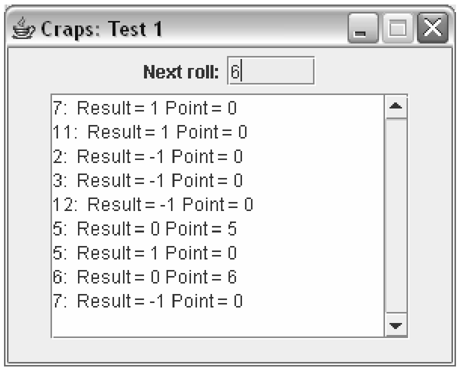
```
        int result = game.processRoll(total);
        int point = game.getPoint();
```

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 7-6). 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.



```
7:  Result = 1 Point = 0
11:  Result = 1 Point = 0
2:  Result = -1 Point = 0
3:  Result = -1 Point = 0
12:  Result = -1 Point = 0
5:  Result = 0 Point = 5
5:  Result = 1 Point = 0
6:  Result = 0 Point = 6
7:  Result = -1 Point = 0
```

Figure 7-6. The preliminary program for testing `CrapsGame`

1. Copy `CrapsTest1.java` and `CrapsGame.java` from `JM\Ch07\Craps` to your work folder. Fill in the blanks in the `CrapsGame` class, compile it, combine it with the `CrapsTest1` class into a program, and test it 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\Ch07\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.

7.10 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;       // Evaluate the expression
                          //   and save its value in x
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` are integer or character literal or symbolic constants. When a switch is compiled, the compiler creates a table of these values and the associated addresses of the corresponding "cases" (code fragments). When the switch is executed, the program first evaluates `expression` to an integer. Then it finds it in the table and jumps to the corresponding case. If the value is not in the table, the program jumps to "default." 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 type (integer or `char`). In most programs it is really not an expression but simply one variable, as in `switch(x)`.

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 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."

6. 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 special comment in the code.

7.11 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 is usually capitalized; 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 defintion 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;
      ...
  }
```

7.12 *Case Study and Lab:* Rolling Dice Continued

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 7-7) 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.

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

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

  // Rolls the dice
  public void rollDice()
  {
    die1.roll();
    die2.roll();
    int total = die1.getNumDots() + die2.getNumDots();
    int result = game.processRoll(pts);
    int point = game.getPoint();
    display.update(result, point);
  }

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

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

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`.

You're certainly welcome to take a look at Aisha's GUI implementation (in `JM\Ch07\Craps\Source.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 7-8) 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 7-8. A fragment from `JM\Ch07\Craps\RollingDie.java`

Copy `RollingDie.java`from `JM\Ch07\Craps` into your work folder and fill in the blanks in its `drawDots` method. (Figure 7-1 shows the desired configurations of dots on a die.) Collect all the files for the *Craps* program together: `Craps.jar` (from `JM\Ch07\Craps`); `CrapsGame.java` and `die.java` (your solutions from the lab in Section 7.9); and `RollingDie.java`. Compile them, and run the program.

7.13 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 multiway branching, 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. 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

1. Write a method that returns the value of the larger of the integers `x` and `y` (or either one, if they are equal): ✓

```
public static int max(int x, int y)
{
  ...
}
```

2.■ 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. ⋟ ✓

3. Which of the following expressions are equivalent (that is, have the same value for all possible values of the variables `a` and `b`) to `!(a || !b)`?

A. `a || !b`
B. `!a || b`
C. `!a && b`
D. `!a && !b`
E. `a && !b`

4. 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)) ...`

5. 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))`

6. Rewrite the following condition to avoid a possible arithmetic exception error:

```
if (Math.sqrt(x) < 3 && x > 7) ...
```

7. 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 (i.e. $a, b, c \neq 0$ and $a/b = b/c$). ⋞ Hint: recall that comparing `double` values for exact match may not work — use integer cross products instead. ⋟

8. Given an integer variable `x`, write a Boolean expression that evaluates to `true` if and only if the decimal representation of the value of `x` ends with exactly two zeroes (no more and no less). ✓

9. (a) Restore appropriate indentation and optional braces in the following code fragment using `if-else-if` sequences and nested `if-else` statements as appropriate:

```
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 > ;
```

10. 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';
```

11. 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');
```

12.■ 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.

13.■ 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\Ch07\Exercises\Dates.java`), which prompts the user to enter two dates, reads the input, and displays the result of the comparison. ✓

14.■ 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(int hours, double rate)
{
  double wages;

  < ... missing code >

  return wages;
}
```

Add your code to the `Wages.java` program, available in `JM\Ch07\Exercises`, and test it.

15.■ A "Be Prepared" test prep book costs $18.95; "Next Best" costs $21.95. A site called apzone.com offers a special deal: both for $37.95. If you buy three or more copies (in any mix of these two titles), they are $15.95 each. If you buy 12 or more copies, you pay only $14.00 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.

16.■ 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`.

17.■ `size1` and `size2` are the sizes of two files, and `space` is the amount of available space on a CD. Write a method that takes these integer numbers as arguments and figures out the largest combination of files that fits on a CD. The method should return 3 if both files fit together, the file number (1 or 2) corresponding to the longest file that fits by itself (`1` if the files are the same size), or 0 if neither file fits on the CD. Your method must have only one `return` statement.

```
public int findBestFit(int size1, int size2, int space)
{
  < ... missing code >
}
```

18. Generalize the `Die` class from Section 7.9 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*.

19. 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.

20.◆ Finish the program in `JM\Ch07\Exercises\Rps.java` that plays the "Rock, Paper, Scissors" game. You need to supply code for the `nextPlay` method. Use nested `switch` statements.

21.■ 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) // new roll
  ...
else   // keep rolling
  ...
```

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 < 8)
  chapter++;
```

Iterative Statements: **`while`**, **`for`**, **`do-while`**

8.1 Prologue

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 element 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 12 and 19. Java has a convenient "for each" loop for traversing a collection of values.

In this chapter we will learn the Java syntax for `while`, for, and `do-while` loops and how to use `break` and `return` in loops.

8.2 The `while` and `for` Loops

The general form of the `while` statement 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 <u>not</u> 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 of the MyMath class (JM\Ch08\MyMath\MyMath.java) 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 i = 1;

  while (i <= n)
  {
    sum += i;
    i++;
  }

  return sum;
}
```

We can discern three elements that must be present, in one form or another, 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, i is initially set to 1 in the declaration int i = 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 i <= n. If n is zero or negative, the condition will be false on the very first test (since i 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 `i`:

```
    ...
    i++;        // increment i
```

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 <u>always executed once</u> before the first pass through the loop, *condition* is tested <u>before each pass</u> through the loop, and *change* is a statement executed <u>at the end of each pass</u> 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. We don't feel strongly about either style, so we will use both, depending on the situation or our mood.

Note that a variable that controls the loop can be declared right in the `for` statement. For example:

```
for (int i = 1; i <= 100; i++)
{
  sum += i * i;
}
```

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(i);
```

after the closing brace, you will get a syntax error: undefined variable `i`.

The `sumUpTo` method can be rewritten with a `for` loop as follows:

```
static public int sumUpTo(int n)
{
  int sum = 0;

  for (int i = 1; i <= n; i++)
    sum += i;

  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;
}
```

8.3 The `do-while` Loop

The `do-while` loop differs from the `while` loop in that the condition is tested after 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, entered, or initialized within the body of the loop. The following example comes from `main` in the `MyMath` class (`JM\Ch08\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);

  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();
}
```

8.4 `return` and `break` in Loops

We saw in Section 7.10 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. (If you'd like, you can add it to the `MyMath` class in `JM\Ch08\MyMath`.) A prime is an integer that is greater than 1 and has no factors besides 1 and itself. 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 `isPrime` method below employs `break` to reduce the number of iterations:

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

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`, for example, 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 <u>one</u> `return` in each method and who find a `break` or `return` inside a loop objectionable.

8.5 Nested Loops

A nested loop is a loop within a loop. For example,

```
for (int i = 1; i <= n; i++)
{
  for (int j = 1; j <= i; j++)
    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.numRows(); r++)
{
  for (int c = 0; c < grid.numCols(); c++)
  {
    System.out.print(grid.getValue(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 Chapter 12.

The following method 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 proven nor disproven.

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;
  }
```

8.6 *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.

Write a program 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 <u>even</u> 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. The 42nd Mersenne prime, $2^{25,964,951}-1$, was found on February 18, 2005. (It has 7,816,230 digits.)

Write a program to find the first six Mersenne primes, and use them to calculate the 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!

8.7 Summary

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

1. The population of Mexico in 2003 was 105 million. Write a program that calculates and prints out the year in which the population of Mexico will reach 120 million, assuming a constant growth rate of 2.3% per year. Use a `while` loop.

2. 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).

3. 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.) ✓

4. Write a program to test the algorithms from Questions 1 and 2 in Chapter 4 (page <...>).

5. 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
```

6.■ (a) Modify the `isPrime` method on page <...> 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). ✓

7. Recall that $1+3+...+(2p-1)=p^2$ 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 (see Question <...> for Chapter 4 on page <...>). ✓

8.■ (a) Write a method that calculates and returns the sum of all the digits of a given non-negative integer.

(b) Pretending that the modulo division operator does not exist in Java, write a `boolean` recursive 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.

9. Finish the program `GradeAvg.java` (in `JM\Ch08\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. ⋚

10. Write a method

```
public 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.

11.■ Given a positive number a, the sequence of values

$$x_0 = \frac{a}{2}$$

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) \quad (n \ge 0)$$

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 r of the square root of a,
  // such that |r^2 - a| < 0.01.
  // precondition: a is a positive number
public static double sqrtEst(double a)
{
  double r = a/2;
  double diff;

  do
  {
    ...

  } while ( ... );

  return r;
}
```

12.◆ 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.

13.■ 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 question.

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 answer to the "Next customer?" question:

```
Scanner keyboard = new Scanner(System.in);
char answer;
...

  int quantity = keyboard.nextInt();
  keyboard.nextLine();  // skip the rest of the line
  ...

  String input = keyboard.nextLine().trim();
  if (input.length() == 1)
    answer = input.charAt(0);
  else
    answer = ' ';
```

⋟

14.◆ 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 + 0 pennies
```

(There are 18 different representations for 30 cents and 242 for $1.00.)

15.■ Find and read a few informative websites dedicated to *Fibonacci numbers*. The following recursive method 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 one loop. Add your method to the `MyMath` class in `JM\Ch08` and test it.

Chapter chapter9 = new Chapter(9);

Implementing Classes and Using Objects

9.1 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 Dave, the developer of `Fraction`, and Claire, who wrote a small client class `TestFractions`, to help us co-teach this chapter. They 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 9-1 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 9.9.

<u>Claire</u>: Hi!

<u>Dave</u>: That `Fraction` class — it wasn't a big deal.

<u>Claire</u>: Wow, I didn't realize it had so much stuff in it! ⇦

```
// Represents a fraction with an int numerator and int denominator
// and provides methods for adding and multiplying fractions.

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 9-1 `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);
  }

  // Returns the value of this fraction as a double
  public double getValue()
  {
    return (double)num / (double)denom;
  }
```

Figure 9-1 `Fraction.java` Continued

```
// 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);
    }

    if (n % d == 0)
      return d;
    else if (d % n == 0)
      return n;
    else
      return gcf(n % d, d % n);
  }
}
```

Figure 9-1. `JM\Ch09\Fraction\Fraction.java`

9.2 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).

<u>Dave</u>: 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 even data types without affecting client classes. I also made the `gcf` and `reduce` methods private.

<u>Claire</u>: 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(20, 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.

<u>Dave</u>: I learned in school to provide as little information to clients as possible. This is called "information hiding." ⇦

The reason for making a class's members 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 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*.

Dave: I didn't provide any accessor or modifier methods for the private fields `num` and `denom` because I didn't want anyone to fiddle with their values outside the class. As to the `gcf` method, you're right: I should have made it `public static`. I'll change that later.

Claire: Good idea, whatever that "static" is.

Dave: But my `reduce` method will remain private. ⇦

The concepts of public and private apply to the class as a whole (or, if you wish, to the developer of the class), 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 9.3. 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.

9.3 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 `booleans`, `null` for objects.

A constructor may take parameters of specified types and use them for initializing the new object. If a class has more than one constructor, then they must differ in the number or types of their parameters.

Dave: 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.

Claire: 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 no-args constructor is <u>not</u> supplied.

Normally constructors should prevent programs from creating invalid objects.

Dave: 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.

Claire: What do I get if this "exception" happens?

Dave: 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."

Dave: 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`.

Dave: 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 name for a method. This is a potential source of bugs that are hard to catch.

Dave: 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 extends JFrame
{
  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. I compile, run, and all I see is a blank window! ⇦

9.4 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 die1, die2;
```

— 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 five references are not explicitly initialized: they do not yet refer to valid objects. Their values are `null`, which 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 is to set the variable 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);
```

Dave: 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 9-2-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, equal to f1 (Figure 9-2).

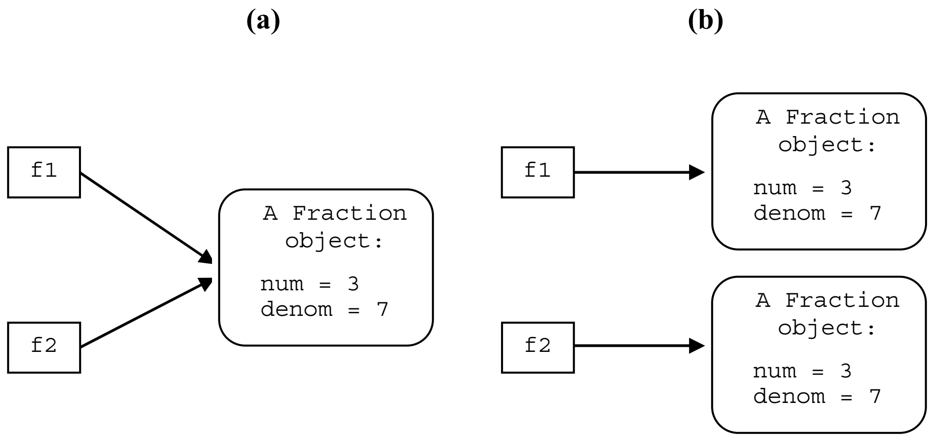


Figure 9-2. Copying references vs. copying objects

9.5 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. You also specify the type of its return value:

```
public [or private] [static]
  returntype methodName(type1 paramName1, ..., typeN paramNameN)
```

This header is called the method's *signature*. *returntype*: can be any primitive data type (such as `int`, `double`, `boolean`, `char`, etc.) or any type of object 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. The parameters can be of any primitive type or any type of objects. Some methods have no parameters at all, only empty parentheses.

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)
```

Dave: I have defined the `add` and `multiply` methods in my `Fraction` class and also a `toString` method.

Claire: 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.

Dave: 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)
{
  ...
}
```

Claire: 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?

Dave: 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 the default `toString`. Without it, `System.out.println(new Fraction(1, 3))` instead of `1/3` would print some garbage, something like `Fraction@126b249`.

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`.

Claire: Whoa, hold it! Overloaded? polymorphism? I think I'm the one who is getting overloaded here! ⇦

Overloaded methods will be explained shortly, in Section 9.10. The `Object` class, class hierarchies, and polymorphism are discussed in Chapter 11.

❖ ❖ ❖

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 as local variables. You cannot use these same names for local variables that you declare in that method.

9.6 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. We will discuss static fields and methods in detail in Section 9.11. For now it is sufficient to understand how to call such 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 of a class 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 often omitted.

Dave: I could have 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.

Dave: 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.

Dave: 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 its methods or constructor parameters. 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. ⇦

`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.

9.7 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;
craps.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`.

Claire: 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.

Dave: In Pascal 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.

Exactly the opposite happens with parameters that are objects of some class type (such as `String`, `RollingDie`, `Fraction`). 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.

Dave: 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.

❖ ❖ ❖

Claire: 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!

Dave: 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, sets it to their sum, and returns a reference to it. ⇦

9.8 `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 a more concise way to write

```
public static boolean inRange(int x, int a, int b)
{
  if (a <= x && x <= b)
    return true;
  else
    return false;
}
```

<u>Dave</u>: 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;
}
```

Claire: 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.

Dave: 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);
  }
```

Claire: 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));
```

Dave: You can even call its method right away:

```
double x = f1.add(f2).getValue();
        // first calls add, then calls getValue for the sum
```

⇦

We will encounter objects returned from methods again when we deal 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;
  ...
}
```

9.9 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 VCRs. 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 red 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 9-3 shows a snapshot of this program. You can play with it by clicking on the `SnackBar.jar` file in `JM\Ch09\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\Ch09\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.

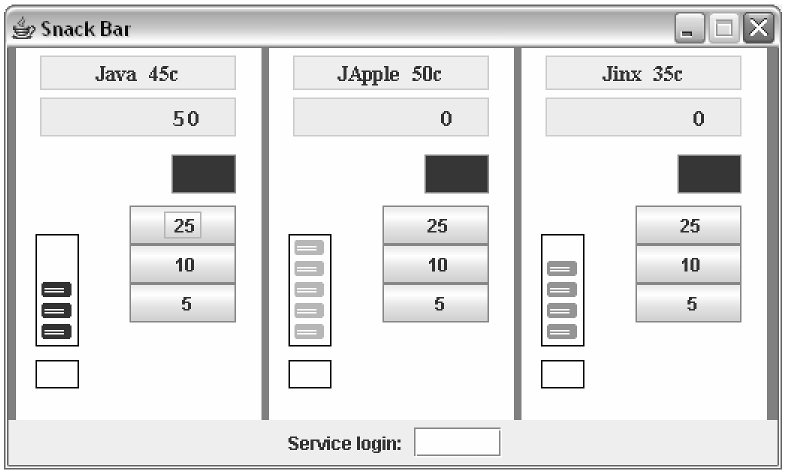


Figure 9-3. The *Snack Bar* program with three vending machines

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 object knows only what it really needs to know about other objects.

The complete `VendingMachine` class code is included in `JM\Ch09\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 red "sale" 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 7 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 9-4 shows the three classes involved in the `SnackBar` program: `SnackBar`, `VendingMachine`, and `Vendor`. 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`.

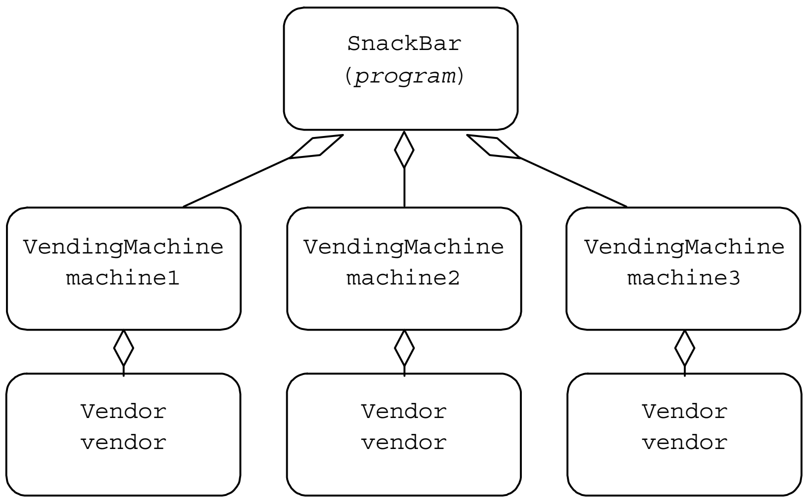


Figure 9-4. *Snack Bar* program’s objects

As you have probably guessed, your job is to implement the `Vendor` class from the written specifications (Figure 9-5). 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
{
  < ... missing fields >

  //  Constructor
  //  Parameters:
  //    int price of a single item in cents
  //    int number of items to place in stock
  ... Vendor ...
  {
    < ... missing code >
  }

  //  Sets the quantity of items in stock.
  //  Parameters:
  //    int number of items to place in stock
  //  Return:
  //    None
  ... setStock ...
  {
    < ... missing code >
  }

  //  Returns the number of items currently in stock.
  //  Parameters:
  //    None
  //  Return:
  //    int number of items currently in stock
  ... getStock ...
  {
    < ... missing code >
  }
```

Figure 9-5 `Vendor.java` *Continued*

```
  //  Adds a specified amount (in cents) to the deposited amount.
  //  Parameters:
  //    int number of cents to add to the deposit
  //  Return:
  //    None
  ... addMoney ...
  {
    < ... missing code >
  }


  //  Returns the currently deposited amount (in cents).
  //  Parameters:
  //    None
  //  Return:
  //    int number of cents in the current deposit
  ... getDeposit ...
  {
    < ... missing code >
  }

  //  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 and returns true;
  //  otherwise refunds the whole deposit (moves it into change)
  //  and returns false.
  //  Parameters:
  //    None
  //  Return:
  //    boolean successful sale or failure
  ... makeSale ...
  {
    < ... missing code >
  }

  //  Returns and zeroes out the amount of change (from the last
  //  sale or refund).
  //  Parameters:
  //    None
  //  Return:
  //    int number of cents in the current change
  ... getChange ...
  {
    < ... missing code >
  }
}
```

Figure 9-5. `JM\Ch09\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, 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\Ch09\SnackBar`. Complete the `Vendor` class and test the program thoroughly.

9.10 Overloaded Methods

It is not surprising that methods in different classes may have exactly the same names. Since a method is always called for a particular object (or a particular class), there is no confusion. For example:

```
person.stop();    // calls stop in Walker class
t.stop();         // calls stop in Timer class

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.

Dave: 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.

Claire: 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.

9.11 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 = .97,
                               speedFactor = .04,
                               speedLimit = 2.;

   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 <u>all</u> 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. 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 look at Dave's `gcf` method — it returns a GCF of two integers. Or `System`'s `exit` method.

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.

Dave: 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)
```

Claire: That was my idea! But why is it static?

Dave: 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 Sun didn't put a `gcf` method into their `Math` class.

Claire: You mean I can now write simply

```
int r = Fraction.gcf(a, b);
```

Dave: 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.

Claire: I had this problem recently when I wrote

```
public class Test
{
  public void testConstructors()
  { ... }

  public void testArithmetic()
  { ... }

  public static void main(String[] args)
  {
    testConsructors();
    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()
{ ... }
```

Dave: 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.testConsructors();
    obj.testArithmetic();
  }
}
```

⇦

It is very common for `main` to create the first object in the program and then let it do the rest of the work. Look at the `HelloGui` example on page <...>.

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.

Dave: 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.

Claire: 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?

Dave: 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 members, 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."

Claire: You mean I can't write

```
Math xyz = new Math();
```

?

Dave: 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 *instance* (non-static) methods and some *class* (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.

9.12 *Case Study:* Snack Bar Continued

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). 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();
machine1.reload();
machine2.reload();
machine3.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.

9.13 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 any type of values, 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 and types 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.) 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 when the object is passed to `print` or `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 and types 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

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 constructors `Complex(double a, double b)` and `Complex(double a)`, 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. 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.■ The program *Temperature* (`JM\Ch09\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 Celsius is 5/9 degree Fahrenheit. For example, 68°F is 5/9·(68 - 32) = 20°C ⪼

7.■ Add an integer parameter `size` to `RollingDie`'s constructor (see Section <...>) 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.

8. 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 `InvalidArgumentException`.

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 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.Point` _______
`java.awt.Color` _______
`java.util.Calendar` _______

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 "European" 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 `equals(Time other)` that returns `true` if this time equals `other` 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) Test your `Time` class using the provided `TestTime` console application class (`JM\Ch09\Exercises\TestTime.java`).

14. 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).

(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 *SnackBar* program, so that the program reports the amount of change received by the customer in specific coin denominations (e.g. `"Change 65c = 2q + 1d + 1n"`). ⋲ Hint: modify the statements in the `actionPerformed` method of the `VendingMachine` class after `vendor.getChange` is called. ⋺

15. 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);
  }
}
```

16. Figure out how to use the `Vendor` class from the *SnackBar* 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.

17.■ (a) Implement a class `Complex` (which represents a complex number $a + b \cdot i$) with two fields of the type `double` and two constructors as described in Question <...>. `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) Add a `toString` method to your `Complex` class that returns a string representation of the number in the form *a*+*bi*.

(d) Test your `abs` and `add` methods in a console application. ✓

(e) 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. Can you pass a `double` rather than a `Complex` to this method as a parameter? Add an overloaded version of `add` that would allow you to do that.

18. 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();
  }
}
```
✓

19. Rewrite the `FCConverter` class from Question (...), 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.

20.■ Some of the Java library classes provide a static method `valueOf`, which converts its parameter into an object of that class and returns that object. For example, `String.valueOf(int x)` returns a string representation of `x`. (the same as `"" + x`). 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`'s method `round` to round the numerator.

21.◆ (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` and `other` teams 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 two 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 three 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.

22.◆ 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 10"

Strings

10.1 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 constructors and 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 string*s (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.

In this chapter we will cover some string properties and methods that help us use strings in our programs. In particular, we will discuss:

- `String` constructors
- The immutability property
- Commonly used `String` methods
- How to convert numbers into strings and strings into numbers
- A few methods of the `Character` class that identify digits and letters

10.2 Literal Strings

Literal strings are written as text in double quotes. The text may contain escape characters (see Section <...>). Recall that the backslash character '\' is used as the "escape" character: `\n` stands for "newline," `\'` represents a single quote, `\"` represents a double quote, and `\\` represents a backslash inside a literal string. For example:

```
String hi = "T\'s up\n";
String pathName = "C:\\Ch10\\funny.txt";
                                    // meaning C:\Ch10\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 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.

10.3 `String` Constructors and Immutability

The `String` class has nine 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.

The other seven constructors create strings from character and byte arrays. They are potentially useful, but not before we learn about arrays (Chapter 12).

There is a big difference between an empty string and an uninitialized `String` reference.

Empty strings are initialized to `""` or created with the empty string constructor, as in

```
String s1 = "";             // s1 is set to an empty string
String s2 = new String();   // s2 is set to an empty string
```

An uninitialized `String` reference is `null`:

```
private String name;        // instance variable name 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 is constructed, it cannot be changed! If you look carefully at `String`'s methods, summarized in Figure 10-3, 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.toLower(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 10-1).

```
String bandName = "Foo Fighters";
char c = bandName.charAt(0);
bandName = Character.toLower(c) + bandName.substring(1);
```

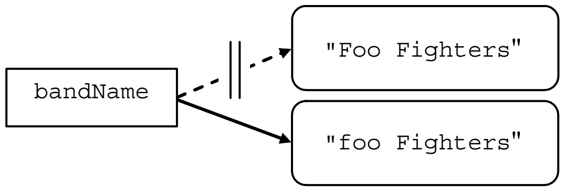


Figure 10-1. A `String` variable reassigned to a new value

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 10-2) 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 new copy of s1
```

— you can use

```
String s2 = s1;  // s2 refers to the same string as s1
```

But if you build a new string for every little change, a program that frequently changes long strings, represented by `String` objects, may become slow.

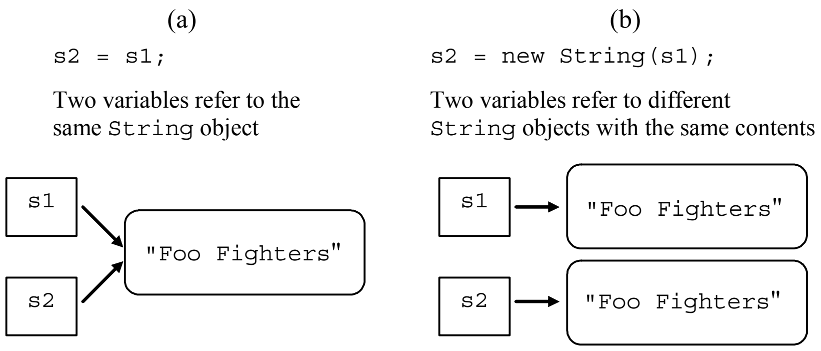

(a)
s2 = s1;
Two variables refer to the same String object
s1
s2
"Foo Fighters"
(b)
s2 = new String(s1);
Two variables refer to different String objects with the same contents
s1
s2
"Foo Fighters"
"Foo Fighters"

Figure 10-2.   Assigning references vs. copying strings

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.toLower(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 10.8. Make sure you've read it before writing commercial applications!

10.4 `String` Methods

The more frequently used `String` methods are summarized in Figure 10-3. 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.

```
int n      = s.length();
char ch    = s.charAt(pos);
String s2 = s.substring(fromPos);
String s2 = s.substring(fromPos, toPos);
String s2 = s.concat(str);
```

```
int result     = s.compareTo(s2);
int result     = s.compareToIgnoreCase(s2);
boolean match = s.equals(s2);
boolean match = s.equalsIgnoreCase(s2);
```

```
int k = s.indexOf(ch);
int k = s.indexOf(ch, fromPos);
int k = s.indexOf(str);
int k = s.indexOf(str, fromPos);
int k = s.lastIndexOf(ch);
int k = s.lastIndexOf(ch, fromPos);
int k = s.lastIndexOf(str);
int k = s.lastIndexOf(str, fromPos);
```

```
String s2 = s.trim();
String s2 = s.replace(oldChar, newChar);
String s2 = s.toUpperCase();
String s2 = s.toLowerCase();
```

Figure 10-3. Commonly used `String` methods

`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.

This convention goes back to C, 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 positions of characters in strings, they fall in the range from 0 to string length - 1.

Substrings

The `String` class has two (overloaded) `substring` methods. 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"
```

Note: 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 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` may be as inefficient as `s = s + s2`.

As we said in Section <...>, 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 (ASCII) 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 intermediate operation 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 <u>substring</u> rather than a single character. For example:

```
String s = "Internet", s2 = "net";
int pos1 = s.indexOf("net");         // pos1 gets the value 5
int pos2 = s.indexOf(s2, 6);         // pos2 gets the value -1
int pos3 = s.lastIndexOf(s2);        // pos3 gets the value 5
int pos4 = s.lastIndexOf("net", 6);  // pos4 gets the value 5
```

<u>Comparisons</u>

You cannot use relational operators (`==`, `!=`, `<`, `>`, `<=`, `>=`) to compare strings.

Recall that relational operators `==` and `!=` when applied to objects compare the objects' <u>references</u> (i.e., their addresses), <u>not their values</u>. 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<...>). 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 <u>before</u> 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 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
                    // all 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();
```

10.5 Converting Numbers into Strings and Strings into Numbers

As we discussed in Section <...>, 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.

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`, 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`of *Integer* class, 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.getValue() + obj2.getValue());
System.out.println(sum);
```

displays 8. Similarly, the `Double` class has a method `doubleValue`of *Double* class and the `Character` class has a method `charValue`.of *Character* class

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. For example:

```
int n = -123;
String s = Integer.toString(n);  // s gets the value "-123";
```

The second way is to use the static method `valueOf` of the `String` class.

```
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 three leading zeroes and four digits after the decimal point. Then 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 moneyFormat1:
double totalSales = 12345679.9;
String s1 = money1.format(totalSales);
    //  s1 gets the value "12345678.90"
String s2 = money2.format(totalSales);
    //  s2 gets the value "$12,345,679"

// Create a DecimalFormat object specifying 2 digits
//  (with a leading zero, if necessary):
DecimalFormat twoDigits = new DecimalFormat("00");

// Convert minutes into a string using timeFormat:
int minutes = 7;
String s2 = twoDigits.format(minutes);
```

```
    //  s2 gets the value "07"
```

If, for example, `totalSales` is 123.5 and you need to print something like

```
Total sales are $123.50
```

you could write

```
System.out.print("Total sales are " +
                          money2.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 = 2, day = 14, year = 2007;
  double amount = 123.5;
  System.out.printf("Date: %02d/%02d/%d  Amount: %7.2f\n",
                                        month, day, year, amount);
```

displays

```
 Date: 02/14/2007  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. `\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 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 = 2, day = 14, year = 2007;
    double amount = 123.5;
    String s = String.format("Date: %02d/%02d/%d  Amount: %7.2f\n",
                                              month, day, year, amount);
```

The above statements set `s` equal to `"Date: 02/14/2007  Amount:  123.50"`.

❖ ❖ ❖

The reverse operation — converting a string of digits (with a sign, if necessary) 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 10-4 shows how all this may be coded.

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
```

```
    Scanner input = new Scanner(System.in);
    int n = 0;

    while (n <= 0)
    {
      System.out.print("Enter a positive integer: ");
      String str = input.next();
      input.nextLine();  // skip the rest of the line
      try   // try once 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 10-4. Converting input into an `int` with exception handling

10.6 `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:

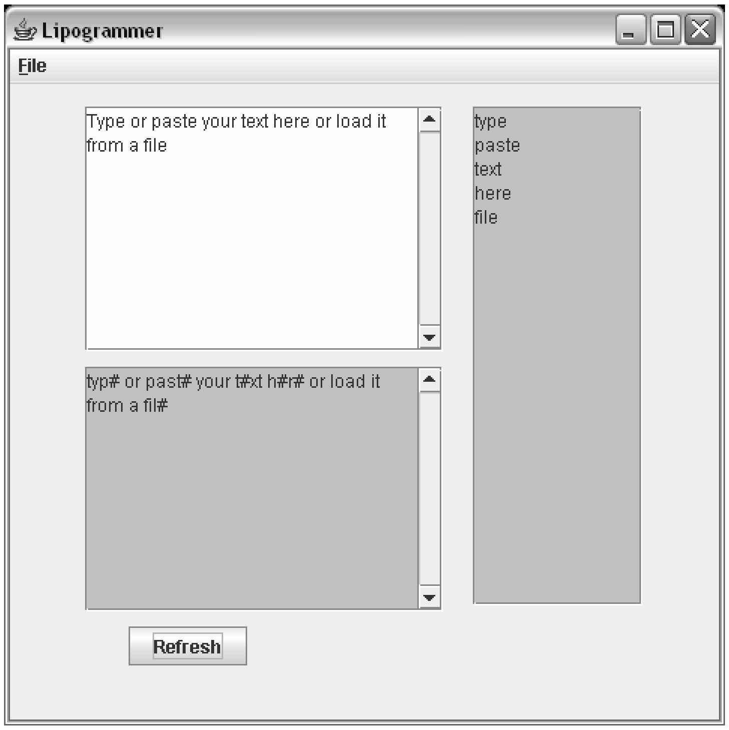
```
char c1 = Character.toUpperCase('a');  // c1 is set to 'A'
char c2 = Character.toUpperCase('*');  // c2 is set to '*'

Scanner input = new Scanner(System.in);
String firstName = input.next();

// Capitalize firstName:
char c = firstName.charAt(0);
firstName = input.toUpperCase(c) + firstName.substring(1);
```

10.7 *Lab: Lipogrammer*

According to *Wikipedia* (the Internet encyclopedia),

> 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 10-5, 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.



Type or paste your text here or load it from a file

typ# or past# your t#xt h#r# or load it from a fil#

type
paste
text
here
file

Figure 10-5. The *Lipogrammer* program

The `LipogramAnalyzer` should have the following constructor and two public methods:

```
// Constructor: saves the text string
public LipogramAnalyzer(String text)

// Returns the text string with all characters
// equal to letter replaced with '#'.
public String mark(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.
public String allWordsWith(char letter)
```

Hint: use 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 and the GUI classes `Lipogrammer.java` and `LipogrammerMenu.java` in the `JM\Ch10\Lipogrammer`. Test the program.

10.8 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 three constructors:

```
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 10-6 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, `char[]`, or `Object`. `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` equal to the substring between `fromPos` and `toPos-1`. `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 characters held in the buffer.

```
StringBuffer sb = new StringBuffer(10); // sb is empty

int len = sb.length();                  // len is set to 0
int size = sb.capacity();               // size 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(2004);                        // sb holds "win2004"
String str = sb.toString();             // str is set to "win2004"
```

Figure 10-6. Examples of common `StringBuffer` methods

10.9 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 10-3 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 a number. Your program should be able to catch the exception, alert the user, and keep running (see Figure 10-4).

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 `charAt`, `substring`, `indexOf`, and some other `String` methods, the `StringBuffer` class has `setCharAt`, `append`, and `insert` methods.

Exercises

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. (a) A string `dateStr` represents a date in the format “mm/dd/yyyy” (for example, `"05/31/2009"`). Write a statement or a fragment of code that changes `dateStr` to the format “dd-mm-yyyy” (e.g., `"31-05-2009"`). ✓

(b)■ Make the method in Part (a) more general, so that it can handle dates written with or without leading zeroes (for example, it should convert `"5/3/2009"` into `"03-05-2009"`). ✓

(c) Use the program in `JM\Ch10\Exercises\StringTest.java` to test this code and for other exercises.

4. 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"`.

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 0’s are replaced with 1’s and all the 1’s with 0’s. 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 <...> above. ⦄

10. Write a method that tries to find opening and closing comment marks (`"/*"` and `"*/"`) in a string. If both are found, 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, 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 recursive 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 and `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 (see Appendix <…>). 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("Strings are <b>immutable</b>")` should return a string equal to `"Strings are immutable"`.

15. Write 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

```
String validFileName(String fileName)
```

that validates the input, appends the default extension `".TXT"` if no extension is given (i.e., 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\Ch10\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. ⋞ Hint: recall that the `Character` class has static methods `boolean isLetterOrDigit(ch)` and `char toUpperCase(ch)`. ⋟

19.■ The program *Cooney* (`JM\Ch10\Exercises\Cooney.class` 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: use `JM\Ch10\Exercises\StringTest.java` as a basis for your program. ⋟

20.▪ An ISBN (International Standard Book Number) has ten digits. The first nine digits may have values from '0' to '9'; they identify the country in which the book was printed, the publisher, and the individual book. The tenth digit is a "check digit" assigned in such a way that the number $d_1d_2d_3d_4d_5d_6d_7d_8d_9d_{10}$ has the property:

$$(10d_1 + 9d_2 + 8d_3 + 7d_4 + 6d_5 + 5d_6 + 4d_7 + 3d_8 + 2d_9 + d_{10}) \text{ mod } 11 = 0$$

"mod" stands for modulo division (same as `%` in Java). If d_{10} needs the value 10 to balance the check digit equation, then the character 'X' is used. For example, 096548534X is a valid ISBN.

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 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 in a simple program. ⪡ 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.■ (a) 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 `String` fields: one to hold the answer word; another to hold the partially filled string, initially set to a string of dashes of the same length as the answer; and the third to hold all the letters already tried (with no duplicates), initially empty. Another field, an `int`, should hold the number of failed guesses.

Provide a constructor that initializes the answer to a given string and all the other fields appropriately. Provide accessor methods for all four fields. Finally, provide an `int` method `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.

(b) Write a simple console application to play the game. Prompt the user to guess a letter. Allow the letter to be in upper or lower case and to have any number of tabs or spaces around the letter. After each guess, display the correct guesses (in the right places, with dashes marking the missing letters), the letters tried so far, and the number of failed attempts.

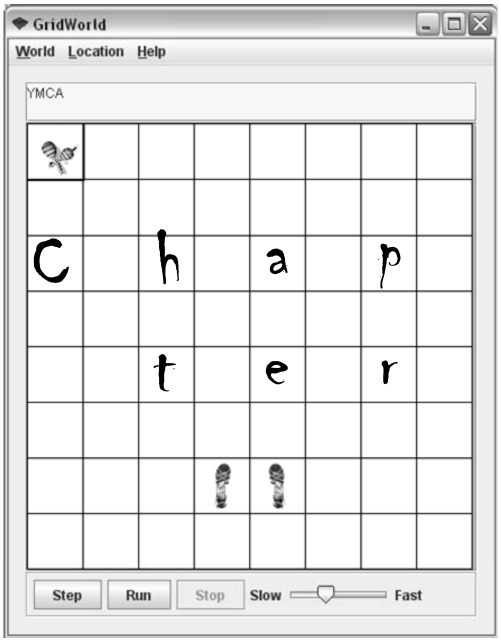
22.◆ Write and test a method

```
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
```



Class Hierarchies and Interfaces

11.1 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 11-1).

```
public class D extends B
```

declares that *D* is a subclass of *B*.

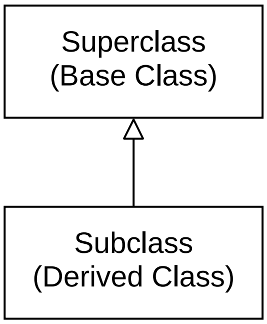


Figure 11-1. Inheritance terminology and notation in class diagrams: a subclass `extends` superclass

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 11-2, for example, `Bug extends`

`Actor` (that is, `Bug` is a *subclass* of `Actor`). The class `Actor` is a generic class that represents any kind of "creature" or thing that can inhabit the grid in GridWorld. `Bug` represents a particular type of actor — a creature that acts in a particular way (moves if it can, otherwise turns). We say that a `Bug` IS-A(n) `Actor`.

As you can see in Figure 11-2, 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.

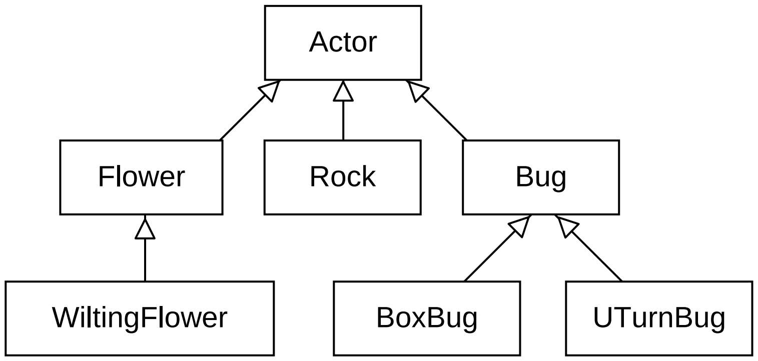


Figure 11-2. A hierarchy of classes

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 and introduce *abstract classes* and *interfaces* — Java tools for managing different levels of abstraction.

11.2 Class Hierarchies

GridWorld's `BoxBug`, `UTurnBug` and other variations of `Bug` (introduced in Chapter 3) are all subclasses of `Bug`. Why couldn't they be direct subclasses of `Actor` instead? The main reason is that the `Bug` class contains code useful for more specific types of "bugs": the `canMove`, `move`, `turn`, and `act` methods. This code is inherited in subclasses of `Bug`. If `BoxBug`, `UTurnBug`, etc, were direct subclasses of `Actor`, each of them would have to duplicate `Bug`'s code.

Duplicate 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.

You have to be a little careful, though. For example, in one of the Chapter 3 exercises we asked you to define a class `RollingRock`. We recommended that you copy `Bug`'s `move` method into your `RollingRock` class. Why not just derive `RollingRock` from `Bug` instead? Technically this would be possible. However, it would break down the logic of the IS-A relationship between classes: a `RollingRock` is not a kind of `Bug` — a `RollingRock` is a kind of `Rock`.

A `RollingRock` is a kind of `Rock` but needs a `move` method, like a `Bug`. It would be nice if a class could inherit methods from several superclasses. However, such *multiple inheritance* is not supported in Java. This looks like a serious limitation. There is a way around it: we can add the same functionality to different types of objects by defining a *wrapper class*. This solution is rather technical, though. Wrapper classes are discussed in Chapter 27.

❖ ❖ ❖

All GridWorld actors share many common features and behaviors. These common features and behaviors are not duplicated in the individual actor classes — they are collected in one common superclass `Actor`. Besides avoiding duplication of code, such an arrangement has another advantage: objects of `Actor`'s subclasses get a secondary, more generic type (`Actor`), which comes in handy in client classes. For example, the class `ActorWorld` has a method

```
public void add(Location loc, Actor occupant)
{
  occupant.putSelfInGrid(getGrid(), loc);
}
```

When we call this method, `occupant` can be any kind of `Actor` (a `Bug`, a `Flower`, etc.). The compiler allows that because objects of subclasses of `Actor` have a secondary, more generic type — `Actor`. Any actor (that is, any object of any subclass of `Actor`) has the `putSelfInGrid` method (inherited from `Actor`). If `Bug`, `Flower`, `Rock`, etc. were not subclasses of `Actor`, we would need a separate `add` method in `ActorWorld` for each of them.

Another example is `ActorWorld`'s `step` method:

```
public void step()
{
  ...
  for (Actor a : actors)
  {
    // only act if another actor hasn't removed a
    if (a.getGrid() == gr)
      a.act();
  }
}
```

`step` calls `act` for each actor in the grid. Different types of actors (`Bugs`, `Flowers`, `Rocks` — objects that belong to different subclasses of `Actor`) can be intermixed in the same grid. Each has an `act` method, but these methods may be different. Java automatically calls the appropriate `act` method for each actor in the grid — the OOP feature known as *polymorphism*. (Polymorphism is discussed later, in Section 11.7.)

❖ ❖ ❖

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.

For a method like `step` to compile, however, Java requires that the `Actor` class have some kind of `act` method. Different subclasses of the `Actor` class override (redefine) this method. But what do we put in `Actor`'s `act`?

We could potentially just leave it empty. For example:

```
public class Actor
{
  ...

  public void act()
  {
    // empty method: the method is defined, but does not do anything
  }

  ...
}
```

This works, but it is inconvenient: an `Actor` would just sit in the grid but we wouldn't see any action. In the implementation of GridWorld, `Actor`'s `act` method rotates the actor 180 degrees:

```
public class Actor
{
  ...

  public void act()
  {
    setDirection(getDirection() + Location.HALF_CIRCLE);
  }

  ...
}
```

If we forget to override the `act` method in an `Actor`'s subclass or if we do it incorrectly (for example, we misspell "act" or make it take a parameter), at least we will see some action and will be able to diagnose the problem quickly.

Another possibility would be to make `Actor`'s `act` throw an `UnsupportedOperationException`:

```
  public void act()
  {
    throw new UnsupportedOperationException();
  }
```

But Java offers us a better solution: we can declare a method *abstract*.

11.3 Abstract Classes

Java allows us to leave some of the methods in a class declared but undefined. Such methods are called *abstract*. The way to declare an abstract method is to use the keyword `abstract` in its header and to supply no code for it at all, not even empty braces. For example:

```
   public abstract void act();
```

An abstract method is better than an `UnsupportedOperationException` or an empty method, because when we declare a method `abstract`, the compiler checks whether each subclass properly defines this method before we can even try to run our program.

If a class has at least one abstract method, it must be declared `abstract`.

For example:

```
  public abstract class Actor
  {
    ...

    public abstract void act();

    ...
  }
```

So why is `Actor` in GridWorld not abstract? Cay Horstmann, GridWorld's lead author, explains:

> The `act` method of the `Actor` class flips the actor, so that you can see if you forgot to override that method. Of course, there would be an even better way of ensuring that the method has been overridden: to declare it abstract. Making `Actor` into an abstract class would clearly be the better design. However, abstract classes are typically introduced at the end of the [AP Computer Science] A course... The

committee wanted to make it easy to use GridWorld throughout the A course. We therefore decided to use the conceptually simpler concrete actor class.[*]

❖ ❖ ❖

Java doesn't allow us to create objects of an abstract class.

If you make `Actor` abstract and try to create an `Actor` object —

```
    Actor alice = new Actor();
```

— the compiler will give you an error message, something like this:

```
 Actor 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 classes at the bottom are concrete.

OO designers sometimes define rather elaborate hierarchies. Figure 11-3 shows a fragment of the hierarchy of Java library classes.

You can clearly see how the newer *Swing* GUI classes, starting with `JComponent`, were added under `Container`, while all the old *AWT* classes (`Button`, etc.) were left untouched. The hierarchy has the class `Object` at the very top.

[*] Cay S. Horstmann, *The Design of the* GridWorld *Case Study*. `http://apcentral.collegeboard.com/apc/public/repository/ap-sf-computer-science-gridworld.pdf`, p. 9.

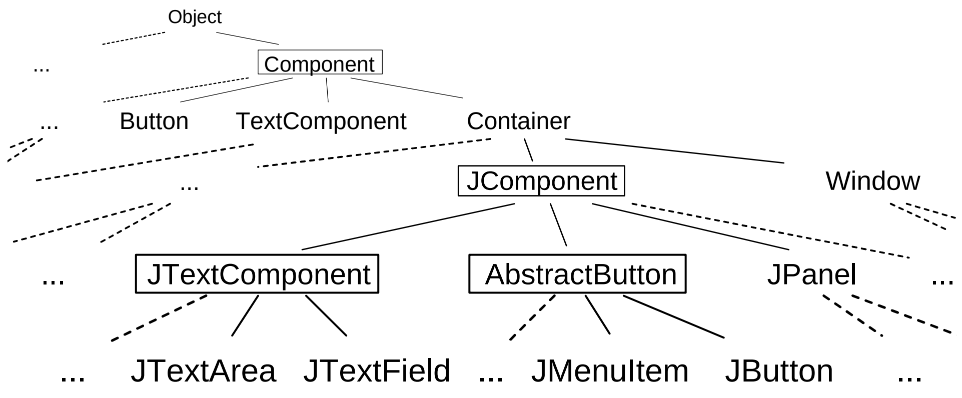


Figure 11-3. A fragment of Java library GUI class hierarchy (abstract classes are boxed)

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 that provides a few generic methods, such as `toString` and `equals`, but these methods are usually overridden in classes down the hierarchy.

11.4 Invoking Superclass's Constructors

In Section 3.6 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 `Actor` class, for example, has the method `getColor`, which returns `Actor`'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 `Actor` class has a *modifier* method `setColor`. `Bug`'s constructors call that method. For example:

```
   public Bug()
   {
     setColor(Color.RED);
   }
```

But a more common solution would be 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 class UTurnBug extends Bug
 {
   public UTurnBug()
   {
     super(Color.YELLOW);  // Call Bug's constructor that takes
                           //    one parameter of the type Color
   }
   ...
 }
```

In Chapter 3 we used instead

```
public class UTurnBug extends Bug
{
  public UTurnBug()
  {
    setColor(Color.YELLOW);
  }
   ...
 }
```

We did that because we simply didn't want to introduce `super` too early.[*]

If your constructor has a `super(…)` statement, then it must be the first statement in the constructor.

[*] GridWorld's authors decided not to use `super` at all in GridWorld's code, even though `super` is tested on AP exams. Feel free to use `super` in GridWorld exercises.

What happens when `super` is not used, like in the second, "setColor" version?

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.

If it doesn't, the compiler displays an error message.

For example,

```
  public UTurnBug()
  {
    setColor(Color.YELLOW);
  }
```

is exactly the same as

```
  public UTurnBug()
  {
    super();     // call Bug's no-args constructor (default)
    setColor(Color.YELLOW);
  }
```

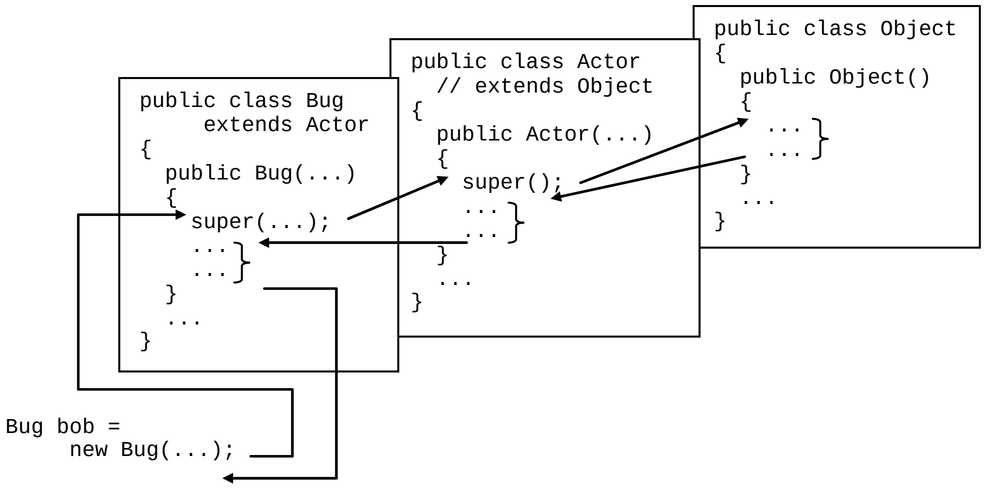
This works because the `Bug` class does have a no-args constructor. The version with `super(Color.YELLOW)` is stylistically better; also it does not require that `Bug` have a no-args constructor.

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 <u>its</u> superclass's constructor, and so on, up the inheritance line, all the way up to `Object` (Figure 11-4). `UTurnBug`'s constructor, for example, calls `Bug`'s constructor, which calls `Actor`'s constructor, which calls `Object`'s constructor.

After all the calls to its superclasses' constructors 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.



```
public class Bug
     extends Actor
{
  public Bug(...)
  {
    super(...);
    ...
    ...
  }
  ...
}
```

```
public class Actor
  // extends Object
{
  public Actor(...)
  {
    super();
    ...
    ...
  }
  ...
}
```

```
public class Object
{
  public Object()
  {
    ...
    ...
  }
  ...
}
```

```
Bug bob =
     new Bug(...);
```

Figure 11-4. 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.

11.5 Calling Superclass's Methods

In Question 17 in Chapter 3 we asked you to write a class `WiltingFlower` as a subclass of `Flower`. On each step (each call to its `act` method) the age of the `WiltingFlower` increases by one. Once the age exceeds the life span, the `WiltingFlower` object "dies."

We asked you to provide a constructor that takes `WiltingFlower`'s life span as a parameter. Actually it is better to provide two constructors: one that takes a life span as a parameter; and another that takes two parameters, a life span and a color. Now, that we have discussed `super`, we can do this properly:

```
public class WiltingFlower extends Flower
{
  private int lifeSpan;
  private int age;

  public WiltingFlower(int span)
  {
    // super(); -- called automatically
    lifeSpan = span;
    age = 0;
  }

  public WiltingFlower(int span, Color color)
  {
    super(color);
    lifeSpan = span;
    age = 0;
  }

  ...
}
```

The first constructor calls `Flower`'s no-args constructor automatically, by default. The latter sets the flower's color to `DEFAULT_COLOR`, which is defined in `Flower`'s code as `Color.PINK`:

```
    private static final Color DEFAULT_COLOR = Color.PINK;
```

That's why `WiltingFlowers` created without a color parameter are pink, just like regular `Flowers`.

So far so good. But our `WiltingFlower` just sits there for a number of steps, doing nothing, then dies:

```
public void act()
{
  age++;
  if (age > lifeSpan)
  {
    removeSelfFromGrid();
  }
}
```

It would be nice if it somehow showed its age. Recall that regular `Flowers` darken with age. How can we make `WiltingFlowers` do the same? We could, of course, just copy the code that darkens the color from `Flower` into `WiltingFlower`. But duplication of code is never a good thing. It would be better if we somehow were able to call `Flower`'s `act` method from `WiltingFlower`.

It turns out we can.

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 act()
{
  age++;
  if (age > lifeSpan)
  {
    removeSelfFromGrid();
  }
  else
    super.act();
}
```

`super`-dot is necessary only if your class defines a method with the same name and parameters as the superclass and we need to distinguish between the two methods.

Typically this happens when a method in your class overrides a superclass's method, as in the above example. Without the `super`-dot prefix in the above code, `act` would call itself recursively, and the program would eventually crash.

Another example: suppose that when we print a `WiltingFlower`, we want to display its age, in addition to the standard information displayed for any actor. To accomplish this we can override `Actor`'s `toString` method in `WiltingFlower`:

```
public class WiltingFlower extends Flower
{
  ...

  public String toString()
  {
    return super.toString() + " age = " + age;
  }
}
```

Note that even though `toString` is defined in `Actor`, it is inherited by `Flower`, and the `super`-dot prefix works as usual: `super.toString()` calls `Flower`'s `toString`, which is simply `Actor`'s `toString`. In general, `super`-dot refers to the nearest explicitly defined method up the inheritance line.

11.6 *Case Study:* A GridWorld Dance

If you have found our discussion of class hierarchies and abstract classes in the previous sections a little, well... abstract, that's understandable. OOP has a complicated conceptual layer and a lot of terminology. Let us see how we can make it work in a project.

Figure 11-5 shows a screenshot from one of the "dance numbers" in our GridWorld Dance mini-case study. You can see several "dancers" all set up for a "YMCA" dance.

Set up a project with `DanceRunner.java`, `GridWorldDance.jar` and all the `.gif` and `.wav` files from the `JM\Ch11\GridWorldDance` folder. Don't forget to add `gridworld.jar` to your project as a required library. Run `DanceRunner` to see a dance number in action. If the sound on your computer is enabled, you will also hear the beat.

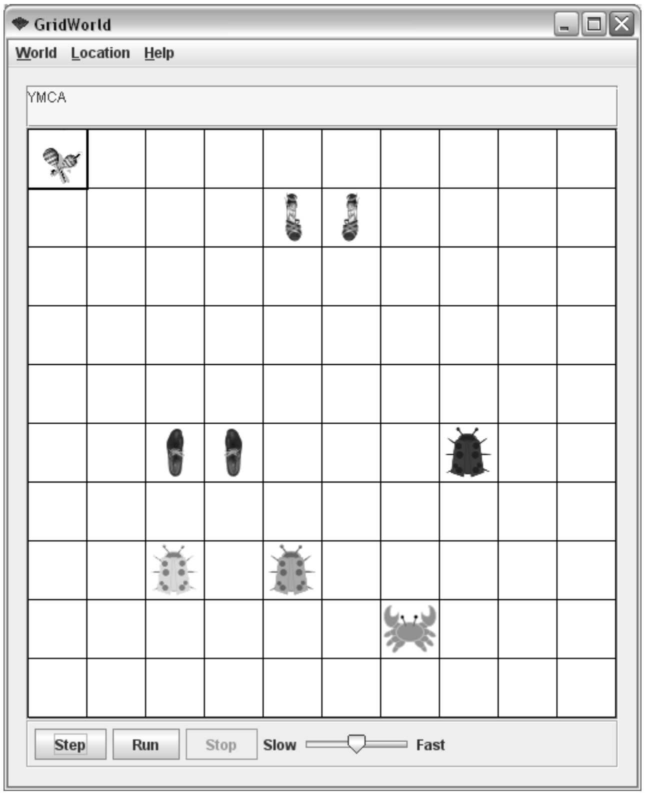


Figure 11-5. GridWorld “dance”

Run `DanceRunner` again and you might see a different dance number. We have programmed three dance numbers: `CongaNumber`, `WaltzNumber` and `YMCANumber`. `DanceRunner` chooses randomly among the three:

```
public class DanceRunner
{
  public static void main(String[] args)
  {
    int k = (int)(3 * Math.random());
    switch (k)
    {
      case 0:
        (new WaltzNumber()).main(args); break;
      case 1:
        (new YMCANumber()).main(args); break;
      case 2:
        (new CongaNumber()).main(args); break;
    }
  }
}
```

If you are patient and run it several times, you will see all three dance numbers.

`YMCANumber.java`, `WaltzNumber.java` and `CongaNumber.java` are located in `JM\Ch11\GridWorldDance`, too. As you can see, each of these "dance number" classes has its own `main` method. For example:

```
public class WaltzNumber extends DanceNumber
{
  ...

  public static void main(String[] args)
  {
    DanceNumber number = new WaltzNumber();
    number.run(new ActorWorld(), new Waltz());
  }
}
```

So if you add them to your project, you can run each of them individually.

Normally, different dance numbers would be integrated into one GUI application, and you would be able to choose a dance number from a menu or a pull-down box. Doing that would require extending GridWorld's GUI or adding a separate control window somewhere on the screen. We have decided against this to keep things relatively simple.

The complete source code for our GridWorld dance project can be found in `JM\Ch11\GridWorldDance\source.zip`.

❖ ❖ ❖

Question 4 on Page 15 of the GridWorld Student Manual asks you to write a `DancingBug` class. A `DancingBug` in that exercise is rather limited: once it receives instructions for a dance (passed to its constructor in an array of steps), it can't "learn" a new dance. In our mini-case study we build upon the `DancingBug` idea. Now that we have a more advanced grasp of OOP, we should be able to teach some of the `GridWorld` inhabitants different dances and more sophisticated steps. Our real purpose, of course, is to review what you have learned so far about class hierarchies, abstract classes, and calling superclass's constructors and methods using `super`, and to provide examples of *polymorphism* for the discussion in Section 11.7 and of an *interface* for Section 11.8.

In the center of our design is an abstract class `Dancer`, which extends GridWorld's `Actor` (Figure 11-6). Different types of `Dancers` are represented by concrete subclasses of `Dancer`. Among them are `DancingBug` and `DancingCrab`, and more exotic "creatures": `LeftShoe`, `RightShoe`, `LeftSandal`, `RightSandal`, and `Maracas`. The latter stays in the corner and moves a little, but mostly provides the beat (using our `EasySound` class).

Notice how the top part of the class diagram in Figure 11-6 — the subsystem of classes that deal with different dances and dance numbers — is separate from the bottom part — the subsystem that deals with dancers. The two subsystems are connected mainly through the `Dance` *interface*. (Interfaces are the subject of Section 11.8.)

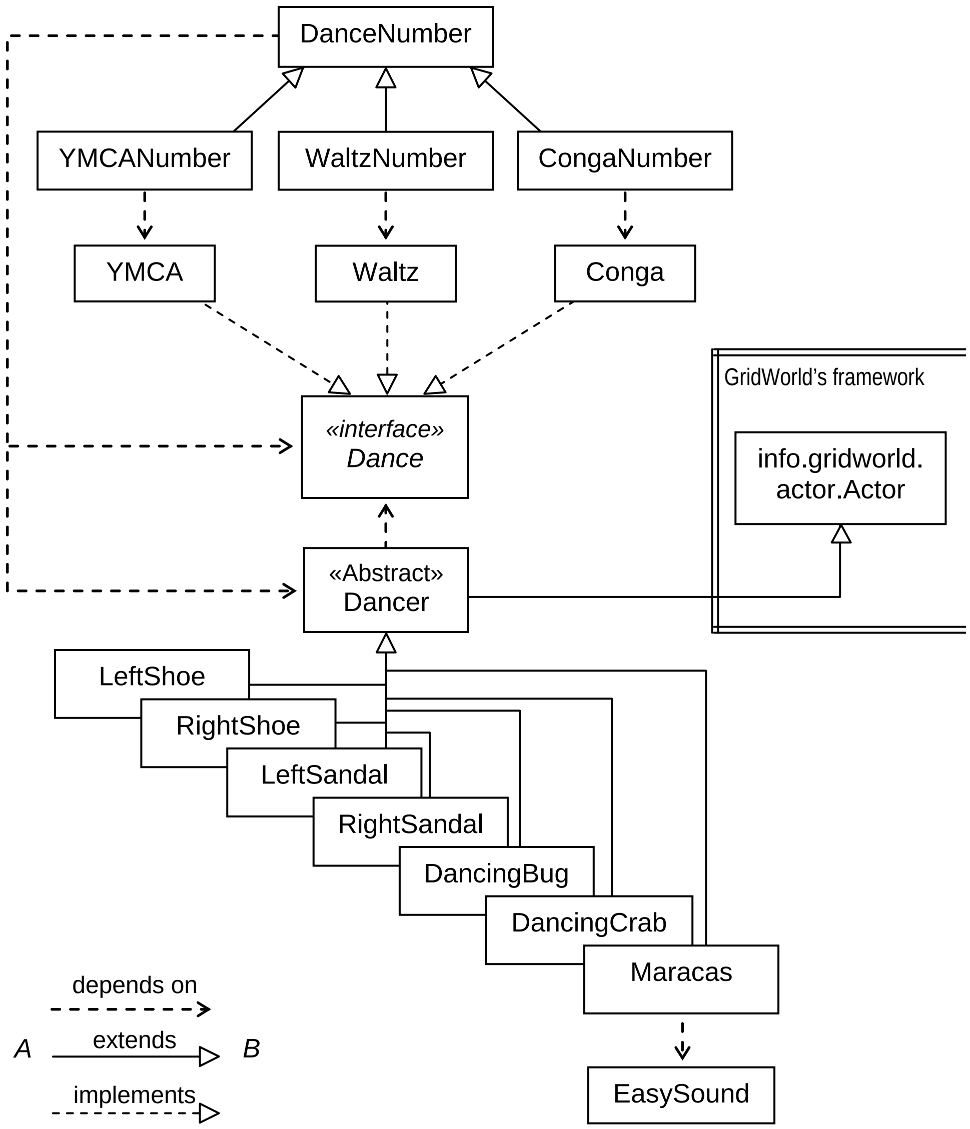


Figure 11-6. GridWorld Dance project: a `Dancer` "learns" a `Dance`, then performs the necessary steps in its `act` method

As we said, `Dancer` is an abstract class. It has one abstract method:

```
public abstract void learn(Dance dance);
```

All its concrete subclasses define this method. Different dancers might need to learn different steps to "perform" the same dance. For example, `LeftShoe`'s and `RightShoe`'s steps are different from each other. Also the man's steps are different from the woman's steps in some dances. In our "waltz" number, one `LeftSandal` and one `rightSandal` perform the female part, and one `LeftShoe` and one `RightShoe` perform the male part. In our implementation, the concrete classes for different dancers are very short and similar to each other. For example:

```
public class LeftShoe extends Dancer
{
  public void learn(Dance dance)
  {
    setSteps(dance.getSteps(1));
  }
}

public class RightShoe extends Dancer
{
  public void learn(Dance dance)
  {
    setSteps(dance.getSteps(2));
  }
}
```

The `learn` methods above use an integer parameter to request a different set of steps from `dance`: `dance.getSteps(1)` vs. `dance.getSteps(2)`.

To be honest, a difference in one number does not warrant making separate classes: we could have simply set that number's value in a constructor. We defined separate classes for different `Dancers` in order to take advantage of GridWorld's device that automatically finds and displays the appropriate image from a file that matches an actor's class name. We have placed `DancingBug.gif`, `DancingCrab.gif`, `LeftShoe.gif`, `RightShoe.gif`, `LeftSandal.gif`, `RightSandal.gif`, and `Maracas.gif` image files in the same folder as their respective Java classes. (`DancingBug.gif` and `DancingCrab.gif` are simply copies of GridWorld's `BoxBug.gif` and `CrabCritter.gif`. We have renamed them to match the respective class names.)

The "shoe" and "sandal" classes have no explicitly defined constructors — they rely on `Dancer`'s no-args constructor, which is called automatically. The latter sets the `Dancer`'s color to `null` (after `Actor`'s no-args constructor sets it to `Color.BLUE`):

```
  public Dancer()
  {
    setColor(null);
  }
```

This tells GridWorld's GUI classes to leave the image color alone (we don't necessarily want blue shoes). The `Dancer` class has another constructor, which sets the `Dancer`'s color to a specified color:

```
  public Dancer(Color color)
  {
    setColor(color);
  }
```

The `DancingBug` class takes advantage of it:

```
  public DancingBug(Color color)
  {
    super(color);
  }
```

Most of the subclasses of `Dancer` do not override its `act` method. This method knows how to interpret a sequence of dance steps.

One exception is the `Maracas` dancer. Its primary task is to provide the beat, so, in its `learn` method, instead of requesting a sequence of steps from `dance`, it requests the sequence of sounds for the beat and defines its own simple steps (move right and turn, then return to the previous location and direction):

```
public class Maracas extends Dancer
{
  ...
  public void learn(Dance dance)
  {
    beat = dance.getBeat();
    stepsCount = 0;
    setSteps("21 57 ");
  }
  ...
```

In its `act` method, a `Maracas` object plays the beat, then calls the superclass's (`Dancer`'s) `act` method to perform the steps:

```
  public void act()
  {
    if (beat != null)
    {
      int i = beat[stepsCount];
      if (i > 0)
        beatSound[i].play();
      stepsCount++;
      if (stepsCount >= beat.length)
        stepsCount = 0;
    }
    super.act();
  }
```

❖ ❖ ❖

We decided to represent a sequence of steps for a dance simply as a string of characters. Each step is represented by a triplet of consecutive characters. `Dancer`’s `act` method cuts out the next triplet of characters from the string and passes it to the `takeStep` method:

```
public abstract class Dancer extends Actor
{
  ...
  public void act()
  {
    if (getGrid() != null && steps != null)
    {
      takeStep(steps.substring(stepsCount, stepsCount + 3));
      stepsCount += 3;
      if (stepsCount >= steps.length() - 2)
        stepsCount = 0;
    }
  }
  ...
```

`takeStep` interprets the step. If the first character is a digit, it indicates the direction in which the dancer moves. `'0'` means forward, `'2'` means sideways to the right, `'4'` means back, `'6'` means sideways to the left, `'1'`, `'3'`, `'5'`, and `'7'` mean the respective diagonal moves. If the first character is not a digit, the dancer does not move in this step:

```
private void takeStep(String step)
{
  char d = step.charAt(0);
  if (Character.isDigit(d))
  {
    int dir = Character.digit(d, 10) * 45;
    Location loc = getLocation();
    Location next = loc.getAdjacentLocation(getDirection() + dir);
    if (getGrid().isValid(next))
      moveTo(next);
    else
      removeSelfFromGrid();
  }
  ...
```

The second character indicates the direction in which the dancer turns after the move. `'1'` means clockwise by 45 degrees from the current direction, and so on. `'0'` or any character that is not a digit means do not turn:

```
  ...
  d = step.charAt(1);
  if (Character.isDigit(d))
  {
    int dir = Character.digit(d, 10) * 45;
    setDirection(getDirection() + dir);
  }
}
```

The third character in a step is a separator and is not used (reserved for "future extension").

11.7 Polymorphism

`ActorWorld`'s `step` method calls `act` for each actor in the grid:

```
public void step()
{
  ...
  for (Actor a : actors)
  {
    // only act if another actor hasn't removed a
    if (a.getGrid() == gr)
      a.act();
  }
}
```

Different types of actors can be mixed in the same grid. Each has an `act` method, and these methods may be different, but the appropriate `act` method is called for each actor in the grid.

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).

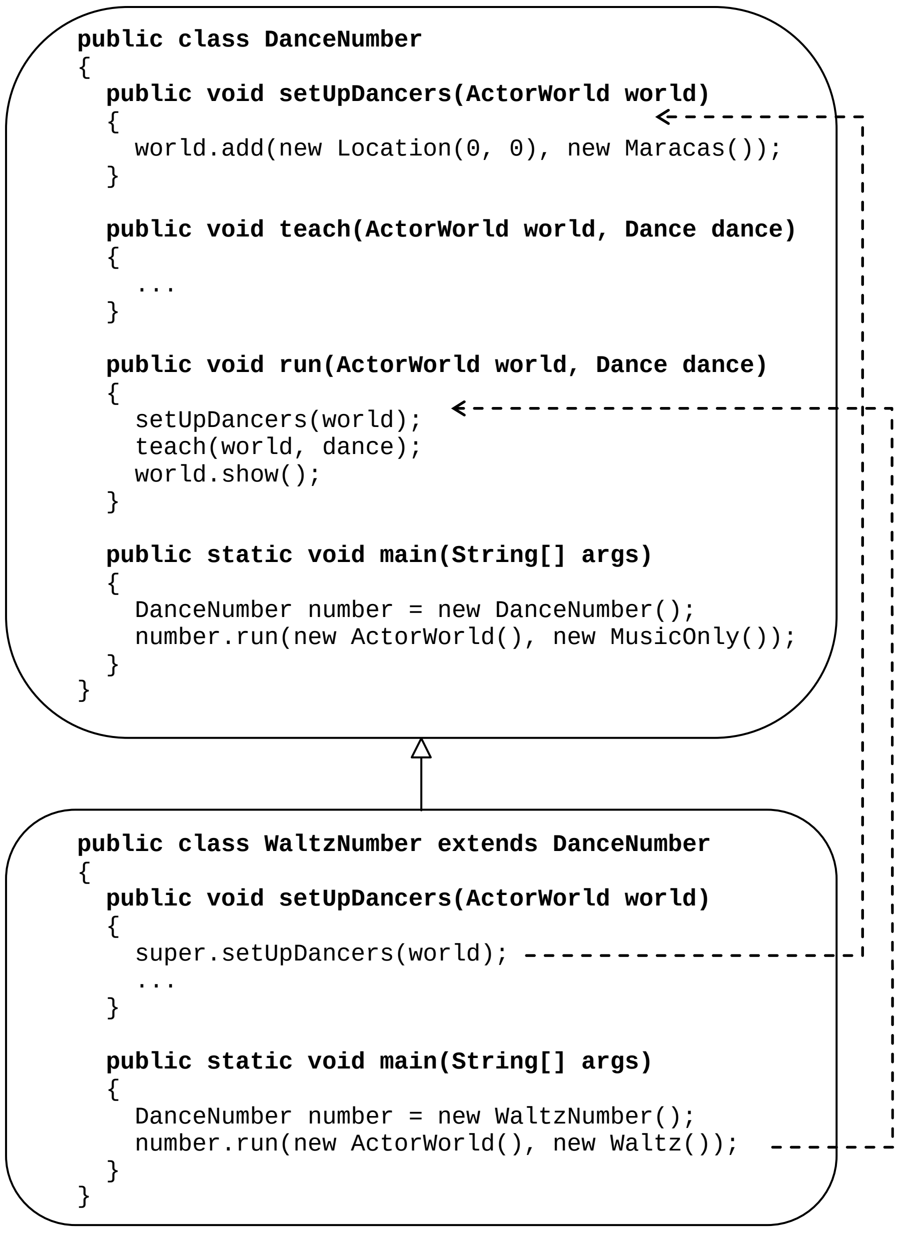
Another example of polymorphism is the `teach` method in the `DanceNumber` class from our GridWorld Dance case study in the previous section. This method "teaches" each dancer in `world` the specified dance by calling the dancer's `learn` method:

```
private void teach(ActorWorld world, Dance dance)
{
  world.setMessage(dance.getName());

  Grid<Actor> gr = world.getGrid();
  for (Location loc : gr.getOccupiedLocations())
  {
    Actor a = gr.get(loc);
    if (a instanceof Dancer)
      ((Dancer)a).learn(dance);
  }
}
```

The appropriate `learn` method is called for every `Dancer` in the grid despite the fact that the dancer's specific type is hidden.

❖ ❖ ❖

Notice a slightly unusual feature of our GridWorld Dance design: each of our "dance number" classes has a `main` method and, at the same time, is a subclass of `DanceNumber`. Why not? `DanceNumber` has a `main`, too: it runs a dance number with music only. Figure 11-7 shows the relationship between `DanceNumber` and its subclass `YMCANumber`. `YMCANumber`'s `main` method calls `DanceNumber`'s `run`; `YMCANumber`'s `setUpDancers` method overrides `DanceNumber`'s `setUpDancers` but calls it using `super`.



```
public class DanceNumber
{
  public void setUpDancers(ActorWorld world)
  {
    world.add(new Location(0, 0), new Maracas());
  }

  public void teach(ActorWorld world, Dance dance)
  {
    ...
  }

  public void run(ActorWorld world, Dance dance)
  {
    setUpDancers(world);
    teach(world, dance);
    world.show();
  }

  public static void main(String[] args)
  {
    DanceNumber number = new DanceNumber();
    number.run(new ActorWorld(), new MusicOnly());
  }
}
```

```
public class WaltzNumber extends DanceNumber
{
  public void setUpDancers(ActorWorld world)
  {
    super.setUpDancers(world);
    ...
  }

  public static void main(String[] args)
  {
    DanceNumber number = new WaltzNumber();
    number.run(new ActorWorld(), new Waltz());
  }
}
```

Figure 11-7. `YMCANumber` extends `DanceNumber`; each has a `main` method and can be used as a runnable class

There is no need to override `DanceNumber`'s `run` and `teach` methods in its subclasses — these methods are the same for all dance numbers. But `DanceNumber` does not know how to arrange dancers for a particular dance, so its subclasses override its `setUpDancers` method. For example:

```
public class WaltzNumber extends DanceNumber
{
  public void setUpDancers(ActorWorld world)
  {
    super.setUpDancers(world);

    world.add(new Location(4, 5), new LeftShoe());
    world.add(new Location(4, 6), new RightShoe());

    Actor leftSandal = new LeftSandal();
    leftSandal.setDirection(180);
    world.add(new Location(2, 6), leftSandal);

    Actor rightSandal = new RightSandal();
    rightSandal.setDirection(180);
    world.add(new Location(2, 5), rightSandal);

    world.add(new Location(6, 3), new DancingBug());
    world.add(new Location(7, 4), new DancingCrab());
  }
  ...
```

The call to `super.setUpDancers(world)` above simply adds a `Maracas` object in the upper left corner of the grid.

When we run `DanceNumber` directly, nothing unusual happens: its `main` creates an object of the `DanceNumber` class and calls its `run` method. `run` in turn calls its `setUpDancers` and `teach` methods (Figure 11-7).

A more interesting situation happens when we execute `main` in `WaltzNumber` or another subclass of `DanceNumber`. `WaltzNumber`'s `main` calls the `run` method inherited from `DanceNumber`. `run` in turn calls `setUpDancers`. But which one? Both `DanceNumber` and `WaltzNumber` have a `setUpDancers` method. It would appear on the surface that `run` would call `DanceNumber`'s `setUpDancers` because they are defined in the same class. Not so! It doesn't matter in which class a method is defined; what matters is which object's method is called at run time. Since `run` is called as a method of a `WaltzNumber` object, <u>that object's</u> `setUpDancers` is called, too. If that were not the case, out dance numbers would always run music only. This is an intricate example of polymorphism.

Polymorphism is implemented in Java using the mechanism called *late* or *dynamic method binding*. When `DanceNumber` is compiled, the compiler does not know which `setUpDancers` method will be called from `run`, so the decision is left until run time. The compiler does not hardcode the address of `setUpDancers` in that call; instead it puts a reference to the `setUpDancers` method in a table of entry points for methods. The compiler creates such a table for 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 `run` is called for an object of the `WaltzNumber` class, that class's table of entry points for methods is used, including its own `setUpDancers` method.

11.8 Interfaces

Look again at Figure 11-6: in the GridWorld Dance project, `Dance` serves to connect the subsystem of dances and the subsystem of dancers. Not only does `Dance` serves as an interface, it actually is an `interface`:

```
public interface Dance
{
  String getName();
  String getSteps(int m);
  int[] getBeat();
}
```

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).

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.

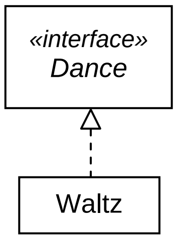
The `Dance` interface, for example, specifies three methods. A concrete class that implements `Dance` must have all three of these methods defined (Figure 11-8).

```
public class Waltz implements Dance
{
  private static String[] steps = { < ...values not shown > };
  private static int[] beat = {1, 2, 2};

  < ... possibly constructors >

  public String getName()
  {
    return "Waltz";
  }

  public String getSteps(int m) { return steps[m-1]; }
  public int[] getBeat() { return beat; }

  < ... possibly other methods >
}
```

Figure 11-8. A sketch of a concrete class that implements the `Dance` interface

Note the level of abstraction that interfaces provide. When you write classes for individual dancers (`DancingBug`, etc.), you do not really need to know what a `Dance` object is or how it is implemented — only that a dance has `getSteps` and `getBeat` methods.

In class diagrams, the fact that a class implements an interface is shown by an arrow with a triangular head and a dotted line, from the class to the interface:



Also, in diagrams and in Java API documentation, the names of interfaces are italicized.

Table 11-1 lists a few similarities and differences between a class and an interface.

Class	Interface
A superclass provides a secondary data type to objects of its subclasses.	**An interface provides a secondary data type to objects of classes that implement that interface.**
An abstract class cannot be instantiated.	**An interface cannot be instantiated.**
A concrete subclass of an abstract class must define all the inherited abstract methods. (An abstract subclass of an abstract class can leave some of the methods abstract.)	**A concrete class that implements an interface must define all the methods specified by the interface. (An abstract class that implements an interface can leave some of the interface's methods abstract.)**
A class can extend another class. A subclass can add methods and override some of its superclass's methods.	**An interface can extend another interface** (called its *superinterface*) by adding declarations of abstract methods. A *subinterface* cannot override methods of its *superinterface* (because there is nothing to override: all methods are abstract.)
A class can extend only one class.	A class can implement any number of interfaces.
A class can have fields.	An interface cannot have fields, except, possibly, some `public static final` constants.
A class defines its own constructors or is supplied with a default no-args constructor by the compiler.	An interface has no constructors.
A concrete class has all its methods defined. An abstract class usually has one or more abstract methods.	All methods declared in an interface are abstract.
Every class is a part of a hierarchy of classes with `Object` at the top.	An interface may belong to a small hierarchy of interfaces, but this is not very common.

Table 11-1. Similarities and differences between a class and an interface (similarities are shown in bold)

The two main similarities are:

1. **A concrete class that extends an abstract class and/or implements an interface must supply code for all the abstract methods of its superclass and/or of the interface.**
2. **Like a superclass, an interface provides a secondary data type to the objects of classes that implement that interface, and polymorphism works for interface types.**

❖ ❖ ❖

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. Class hierarchies in Java have this 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`.

We come across such situations frequently in OOP, and interfaces help. If a class implements several interfaces, that class supplies the code for all methods of all 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.

❖ ❖ ❖

Polymorphism fully applies to interface data types.

Below are two examples of how polymorphism works with interfaces.

Example 1:

We can pass a `Waltz` object to a `Dancer`'s `learn` method as a parameter. The statement that `Waltz implements Dance` tells the compiler that this is allowed, even though `learn` expects a parameter of the `Dance` type. After all, a `Waltz` IS-A `Dance`. What's important is that a `Waltz` object has all the methods that any `Dance` is supposed to have, including the method `getSteps`, called by the `Dancer`, and that it says `implements Dance` in its header. (We can be also sure that a `Waltz` object has the method `getBeat` method in case the `Maracas` needs it.) If `dance` is declared as a variable of the `Dance` type and it happens to refer to a `Waltz` object, then `Waltz`'s `getSteps` will be called automatically when the call `dance.getSteps(...)` is made.

Example 2:

```
public interface Edible
{
  String getFoodGroup();
  int getCaloriesPerServing();
}
```

The `Edible` interface serves as an interface between classes that represent different food items and client classes that work with food items. For instance, we can write a class `Breakfast` with a general method `eat`:

```
public class Breakfast
{
  private int myTotalCalories = 0;
  ...
  public void eat(Edible obj, int servings)
  {
    myTotalCalories += obj.getCaloriesPerServing() * servings;
  }
  ...
}
```

The `eat` method takes an `Edible` object as a parameter. `eat` works properly whether the object passed to it is an `Apple`, a `Pancake`, or any other object of a class that implements `Edible`. The `getCaloriesPerServing` method will be called automatically for an `obj` of a particular type due to polymorphism.

Suppose `Apple` is defined as

```
public class Apple implements Edible
{
  < fields and constructors not shown >

  public String getFoodGroup() { return "Fruit"; }
  public int getCaloriesPerServing() { return 70; }

  < other methods not shown >
}
```

Then `eat` will add `70*servings` to `myTotalCalories`.

❖ ❖ ❖

Names given to interfaces sometimes sound like adjectives because an interface supplies an additional characteristic to the object whose class implements that interface. If `Apple` implements `Edible`, it makes sense to say that an `Apple` is `Edible`.

Interfaces are widely used in the Java library. For example, the classes `ArrayList` and `LinkedList` from the `java.util` package both implement the `List` interface. (We will discuss lists in Chapter 21.) The `java.util` package alone includes about 20 interfaces (see some of them in Figure 20-1 on Page 467).

An interface is also used in GridWorld's `grid` package (Figure 11-9).

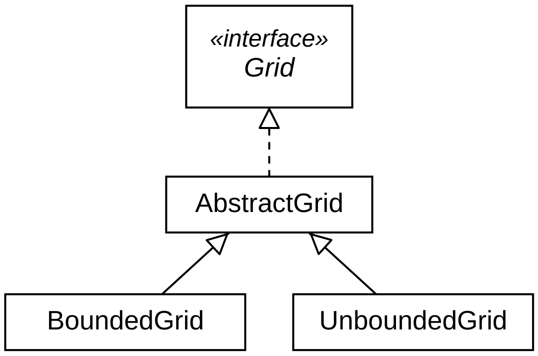


Figure 11-9. GridWorld's `Grid` interface and the classes that implement it

The `Grid` interface has eleven methods; they are listed in Appendix B of the GridWorld Student Manual. `BoundedGrid` and `UnboundedGrid` implement `Grid`, but some of their common code is factored out into an abstract superclass `AbstractGrid`:

```
public abstract class AbstractGrid<E> implements Grid<E>
```

(*E* is the type parameter for objects held in the grid.)

If a class *X* implements an interface *I*, the compiler understands that all subclasses of *X* (and classes down the inheritance line from *X*) also implement *I*. It is allowed but not necessary to write `implements I` in a subclass of *X* — this is understood.

For example:

```
public class BoundedGrid<E> extends AbstractGrid<E>
    // implements Grid<E> -- this line will compile but is redundant
```

GridWorld's authors could have made `Grid` an abstract class and eliminated `AbstractGrid`. This would work in GridWorld programs, but the present design with an interface is more flexible: it lets us implement a grid in a completely different way if we need to, not as a subclass of `AbstractGrid` (potentially as a subclass of some other class).

11.9 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 (and from all of the interfaces that the class and its ancestors implement).

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 (and all of the methods of the interfaces that that class and its ancestors are said to `implement`). 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 can serve to isolate different subsystems from each other, limiting their interaction to a small number of methods. 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

Sections 11.1-11.5

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("Murray Smith", "Gardening");
System.out.println(diploma1);
System.out.println();

Diploma diploma2 = new DiplomaWithHonors("Lisa Smith",
                                 "Evolutionary Psychology");
System.out.println(diploma2);
System.out.println();
```

display

```
This certifies that Murray Smith
has completed a course in Gardening

This certifies that Lisa Smith
has completed a course in Evolutionary Psychology
*** 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. In GridWorld, write and test a class `SlowBug`. On each third step (call to its `act` method) a `SlowBug` acts like a regular `Bug`. On other steps it does nothing. Your class should be very short, just a dozen lines — do not duplicate any part of `Bug`'s code in it. Provide two constructors, similar to `Bug`'s constructors. (Pretend that `Actor`'s `setColor` method does not exist — use `super` instead.) Define only one method, `act`, which overrides `Bug`'s `act` and calls it when necessary. Do not call any other `Bug` methods. ✓

7. *Dionaea* is a carnivorous plant that catches and digests insects. Write a GridWorld class `Dionaea` derived from `Flower`. Provide two constructors, similar to the ones in `Flower`, but make `Dionaea`'s default color `null` (to preserve the colors in the `Dionaea` image file). Do not use the `setColor` method. Define a method `turn`, as in `Bug`, and methods `canEat` and `eat`, similar to `Bug`'s `canMove` and `move` methods. A `Dionaea` "can eat" if there is a `Bug` in the square directly in front of it; `Dionaea`'s `eat` method removes the bug from the grid by calling its `removeSelfFromGrid` method. Define the method `act`, so that a `Dionaea` eats if it can or turns by 45 degrees clockwise if it can't. Change the `BugRunner` class into a `BugKiller` class. Have its `main` method place a few `Dionaea`, `Bug`, and `Rock` objects into the grid. Test your program. ⋞ Hint: `Dionaea`'s image in the program will be the same as `Flower`'s image unless you create an image file `Dionaea.gif` and place it in the same folder as `Dionaea.class`. ⋟

Sections 11.6-11.9

8.■ The class `Triangle` (`JM\Ch11\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.

Continued ⇗

(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? ✓

9. Add a `Bystander` class to the GridWorld Dance project. Make it a subclass of `RightShoe` and override its `learn` method. A `Bystander` does not learn the specific dance steps; instead it stays in the same location and turns back and forth by 45 degrees. Create a subclass of `CongaNumber` that adds a `Bystander` in the corner. ⋚ Hint: `"-1 -7 "` describes the sequence of "dance" steps for a `Bystander`. ⋛

10.◆ Add and test your own "dance" class and dance number class to the GridWorld Dance project (but don't spend too much time on the choreography!)

11. 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.

12. Identify the following in Figure 20-1 on Page 467: pairs of classes that implement the same interface, a class that implements two interfaces, and an interface that is a subinterface of another interface.

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.■ Explain why the `toString` method is never listed in any interface.

15.■ Suppose in our GridWorld Dance project we renamed `Dancer.java` into `AbstractDancer.java` and then made a global change replacing "Dancer" with "AbstractDancer" in `AbstractDancer`' code and all seven of its subclasses (`DancingBug`, `DancingCrab`, `LeftShoe`, `RightShoe`, `LeftSandal`, `RightSandal`, and `Maracas`). Suppose we then created and added to the project an interface `Dancer` with one method, `void learn(Dance d)`. The project compiles fine, but when you run it, the dancers show up but refuse to move. Can you add one line to one of the classes to fix the problem?

double[] chapter12 = {12.1, 12.2, 12.3, 12.4, 12.5, 12.6, 12.7, 12.8, 12.9, 12.10, 12.11, 12.12};

Arrays and `ArrayLists`

12.1 Prologue

Java programmers can declare several consecutive memory locations of the same data type under one name. Such memory blocks are called *arrays,* 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.readInt();
score2 = input.readInt();
...
```

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 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 an array's elements using indices
- Access an array's length
- Pass arrays to methods
- Use the `java.util.ArrayList` class
- Traverse an array or an `ArrayList` using a "for each" loop
- Declare and create two-dimensional arrays

12.2 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 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 elements are object references, you have to initialize each element by setting it to a valid reference before that element can be used. For example:

```
colors[0] = new Color(207, 189, 250);
colors[1] = Color.BLUE;
```

and so on.

❖ ❖ ❖

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", "Nick"};

Color[] rainbowColors =
{
  Color.RED, 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 six 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. You can only reassign the name of the original array to another array and throw away the first one.

❖ ❖ ❖

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. In Java (as in C and C++), 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];
int i;

< ... other statements >

  coordinates[2 * i] = x;
  coordinates[2 * i + 1] = y;
  i += 2;
```

❖ ❖ ❖

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 between `0` and *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[k] = < 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:

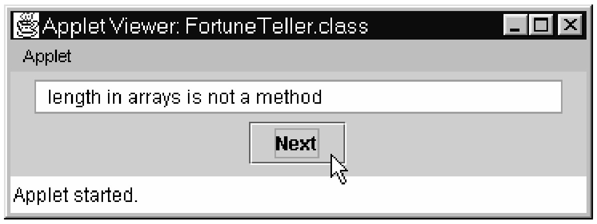
```
// Swaps a[i] and a[j]
public void swap(int a[], int i, int j)
{
  int temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}
```

A method can also create a new array and return it (as a reference). For example:

```
public int[] readScores()  // return type is an integer array
{
  int[] scores = new int[10];

  for (int k = 0; k < 10; k++)
    scores[k] = readOneScore();

  return scores;
}
```

12.3 Lab: *Fortune Teller*

The applet in Figure 12-1 is a “fortune teller.” When the user presses the “Next” button, the applet displays a message randomly chosen from an array of messages. The applet is implemented in one class, `FortuneTeller`.



length in arrays is not a method

Figure 12-1. The *Fortune Teller* applet

Set up a project with the `FortuneTeller.java` and `TestFortune.html` files from `JM\Ch12\Fortunes`. (`TestFortune.html` describes a web page that holds the applet.) Fill in the blanks in the applet’s code, 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\Ch07\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.

12.4 The `ArrayList` Class

Java arrays are convenient and safe. However, they 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.

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 12-2). The capacity is the length of the currently allocated array that holds the list values. The size is the number of elements 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.

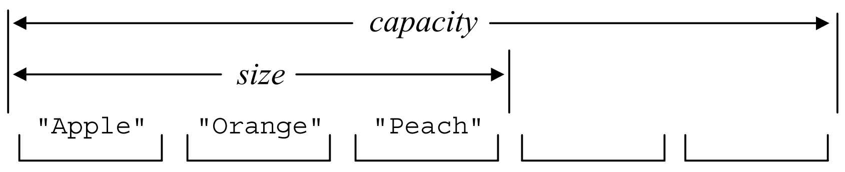
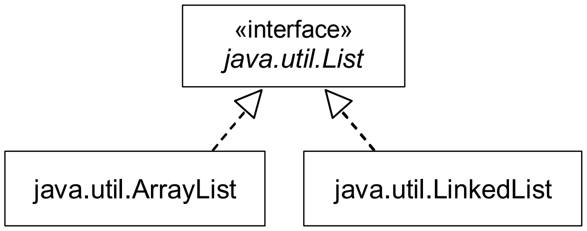


Figure 12-2. The size and capacity of an `ArrayList`

The `ArrayList` class provides the `get` and `set` methods, which access and set the *i*-th element, respectively. These methods check that $0 \le i < size$. 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 falls into the Java *collection framework* and implements the `java.util.List` interface.

Another library class, `java.util.LinkedList`, also implements `java.util.List`:



`LinkedList` has the same methods as `ArrayList`, but it uses a different data structure to store the list elements. This is explained in more detail in Chapter 20.

If your class uses `java.util.List` and `java.util.ArrayList`, add

```
import java.util.List;
import java.util.ArrayList;
```

or

```
import java.util.*;
```

at the top of your source file.

❖ ❖ ❖

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<Walker> family;
```

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 names rearranged
// in random order
public ArrayList<String> shuffle(ArrayList<String> names)
{
  ArrayList<String> result = new ArrayList<String>();
  ...
  return result;
}
```

We hope you have good typing skills!

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 and is the same as

```
ArrayList<Double> samples = new ArrayList<Double>();
samples.add(new Double(5.0));
```

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.)

12.5 `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);
```

```
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 object 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 [elmt1, elmt2, ... elmtN]
```

Figure 12-3. Commonly used `ArrayList<E>` methods

The `ArrayList` class implements over two dozen methods (specified in the `List` interface), but we will discuss only a subset of more commonly used methods. These are shown in Figure 12-3. (Java API docs use the letter *E* to refer to the type of `ArrayList` elements. So it would be more accurate to say that Figure 12-3 shows some of the methods of `ArrayList<`*`E`*`>`.)

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)`, <u>inserts</u> `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 ≤ `i` ≤ *`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. This is accomplished by calling `obj`'s `equals` method, something like this:

```
    if (obj.equals(list.get(i))
      ...
```

Therefore, you should make sure a reasonable `equals` method is defined for your objects if you plan to place them into an `ArrayList`. We will explain how to define `equals` in Chapter 13.

12.6 `ArrayList`'s Pitfalls

You have to be rather careful with the `add` and `remove` methods: keep in mind that they change the indices of the subsequent elements and the size of the list.

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 <u>the same</u> object (that is, `obj1 == obj2`).

It is important to understand that an object can change <u>after</u> it has been added to a list (unless that object is immutable) and that the same object can belong to several `ArrayList`s.

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:

<u>Version 1:</u>

```
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", "Ben", "Claire"};
    List<Person> guests = makeGuestList(names);
    System.out.println(guests);
```

Version 1 displays

```
[Alice, Ben, Claire]
```

as expected (Figure 12-4-a). Version 2, however, displays

```
[Claire, Claire, Claire]
```

because the list contains three references to the same object (Figure 12-4-b). Adding this object to the list does not shield it from being modified by the subsequent `setName` calls.

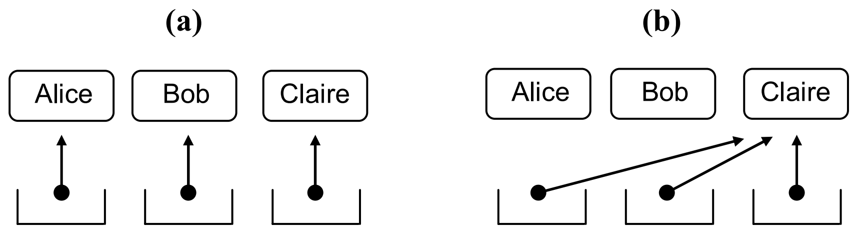


Figure 12-4. (a) A list with references to different objects; (b) a list with references to the same object

12.7 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 (such as an `ArrayList`) 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
}
```

Or, for an `ArrayList`,

```
ArrayList<String> names = new ArrayList<String>();
...
for (int i = 0; i < names.size(); i++)
{
  String str = names.get(i);
  ...  // process str
}
```

Starting with version 5.0, Java offers a convenient "for each" loop for traversing an array or a `List` (or another type of Java `Collection`). The above code fragments can be rewritten with a "for each" loop as follows:

```
for (String str : names)
{
  ...  // process str
}
```

This works for both an array and an `ArrayList`. The "for each" syntax requires that the loop variable be declared inside the `for` statement, as in `String str`.

Note that if you use a "for each" loop to traverse an array with elements of a <u>primitive data type</u>, you <u>cannot</u> change their values because the loop variable holds a copy of the element, not a reference to the original.

The "for each" loop does not give you access to the index of the "visited" element. For example, it will work fine in a method like `contains` but not in a method like `indexOf`. Use a regular `for` loop if you need access to the indices.

❖ ❖ ❖

A frequent reason for traversing a list is looking for 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
static public 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
static public 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. Similar code works for an `ArrayList<String>`:

```
// Returns the position of the alphabetically first word in words
static public int findFirstWord(ArrayList<String> words)
{
  int iMin = 0;

  for (int i = 1; i < words.size(); i++)
  {
    String word = words.get(i);
    if (word.compareToCaseBlind(words.get(iMin) < 0)
      iMin = i;
  }

  return iMin;
}
```

12.8 Inserting and Removing Elements

`ArrayList`'s `add` and `remove` methods take care of inserting and removing elements. For regular arrays, you have to do it yourself. To insert an element, 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 12-5 illustrates the process.

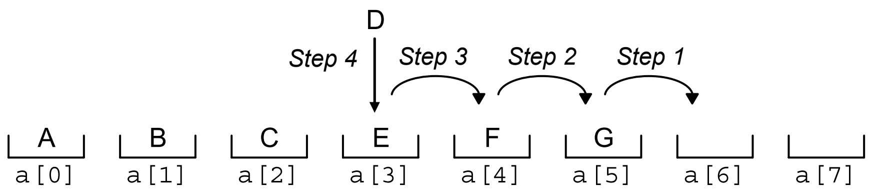


Figure 12-5. Inserting an element in the middle of an array

The following code serves as an example:

```
String word;
int insertPos;

<... other statements >

if (count < maxCount)  // count is the number of words
{
  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.

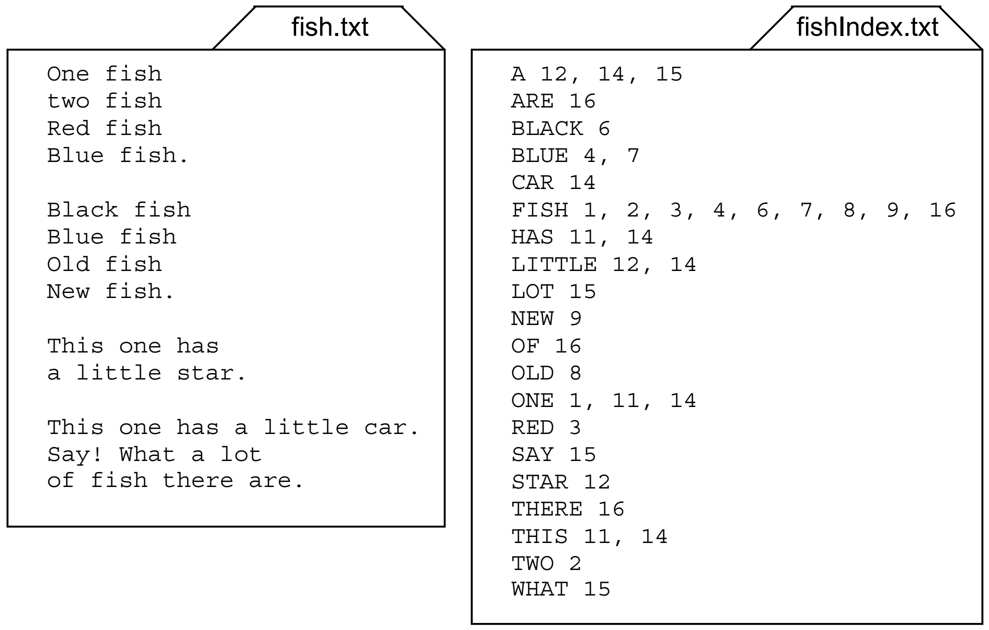
12.9 *Lab:* Creating an Index for a Document

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 12-6 shows an example.

The *Index Maker* program consists of three classes (Figure 12-7). It also uses `ArrayList` in two ways: `IndexEntry` has an `ArrayList<Integer>` field that holds the line numbers, and `DocumentIndex` extends `ArrayList<IndexEntry>`.

The `IndexMaker` class is the main class. We have provided this class for you in `JM\Ch12\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.


fish.txt

```
One fish
two fish
Red fish
Blue fish.

Black fish
Blue fish
Old fish
New fish.

This one has
a little star.

This one has a little car.
Say! What a lot
of fish there are.
```

fishIndex.txt

```
A 12, 14, 15
ARE 16
BLACK 6
BLUE 4, 7
CAR 14
FISH 1, 2, 3, 4, 6, 7, 8, 9, 16
HAS 11, 14
LITTLE 12, 14
LOT 15
NEW 9
OF 16
OLD 8
ONE 1, 11, 14
RED 3
SAY 15
STAR 12
THERE 16
THIS 11, 14
TWO 2
WHAT 15
```

Figure 12-6. A sample text file and its index

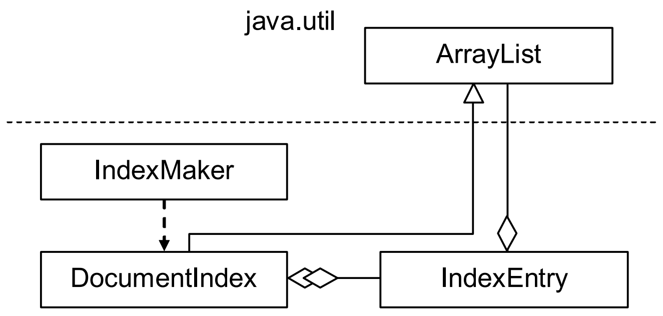


Figure 12-7. `IndexMaker` classes

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. You will need to convert `num` into an `Integer` to call `numsList`'s `contains` method.
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 12-6).

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 in Figure 12-6. 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 methods:

1. `void addWord(String word, int num)` — `addWord` 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.

12.10 Two-Dimensional Arrays

Programmers use two-dimensional arrays to represent rectangular tables of elements of the same data type. For instance, a 2-D array may hold positions in a board game, a table of spreadsheet cells, elements of a matrix, or pixel values in an image.

The following example shows two ways to declare and initialize a 2-D 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 2-D 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 2-D array is represented essentially as a 1-D array of 1-D 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 2-D array to have different numbers of "columns." In this book, we will deal only with "rectangular" 2-D arrays that have the same number of elements in all rows.

If `m` is a 2-D array, then `m.length` is the number of rows in the array, `m[0]` is the first row (a 1-D 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`.

If we want to traverse a two-dimensional array, it is convenient to use *nested* `for` loops. 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] = '*';
  }
}
```

Braces are optional here since the body of each loop consists of only one statement. You could just as well write

```
for (r = 0; r < rows; r++)
  for (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`.)

The following code fragment prints out the values of a 2-D array `m`:

```
for (int r = 0; r < m.length; r++)
{
  for (int c = 0; c < m[r].length; c++)
  {
    System.out.print(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 2-D 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 r, c, firstArow, firstAcol;

for (r = 0; r < rows; r++)
{
  for (c = 0; c < cols; c++)
  {
    if (grid[r][c] == 'A')
      break;
  }
}
firstArow = r;
firstAcol = c;
```

Unfortunately, it will only find the first occurrence of 'A' in the last row (if any).

❖ ❖ ❖

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.

12.11 *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 (Figure 12-8). 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\Ch12\Chomp` to run the *Chomp* 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...

* The proof looks 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,

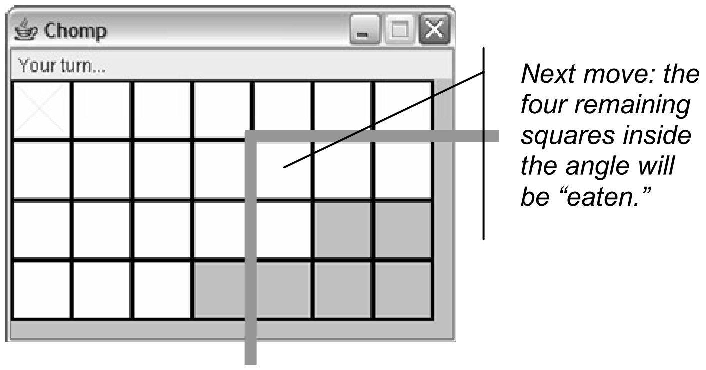

Chomp
Your turn...
Next move: the four remaining squares inside the angle will be "eaten."

Figure 12-8. The *Chomp* game program

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.

❖ ❖ ❖

Let us begin by looking at the overall structure of this program (Figure 12-9). The program consists of eight classes and two interfaces, `Player` and `Strategy`. 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 `Strategy` methods (but not of any particular Chomp strategy).

---

this argument is called "strategy stealing." Unfortunately, this proof gives no clue as to what the winning strategy might be.

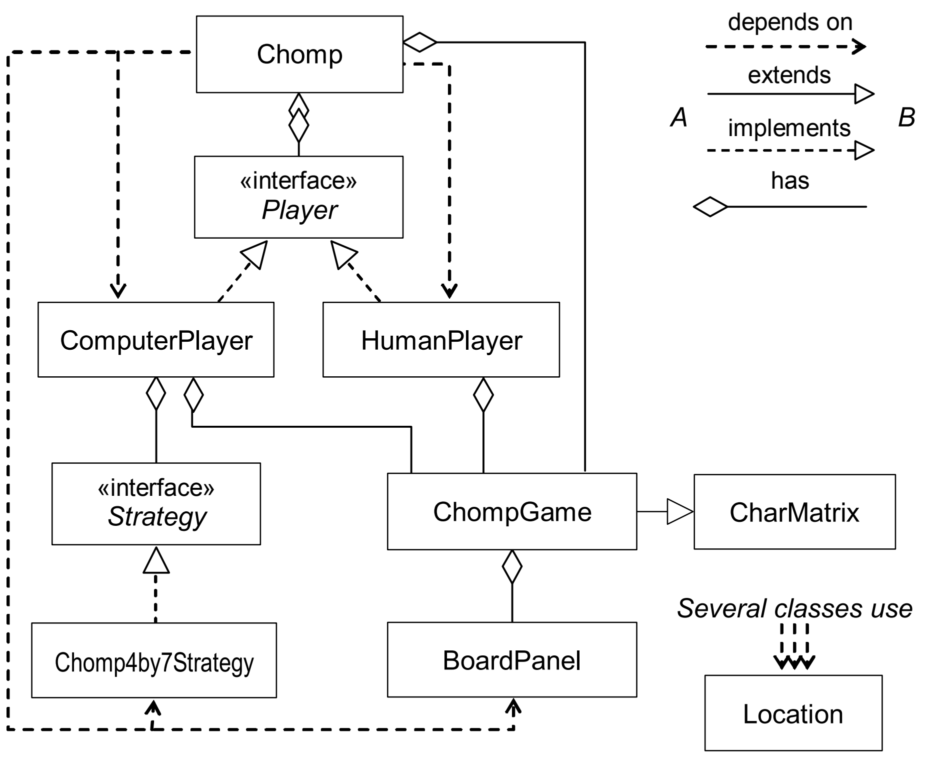


Figure 12-9. The classes in the *Chomp* program

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 2-D 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 rather large. More importantly, general methods dealing with a character matrix would be mixed together in the same class with much more specific methods that deal with the rules of Chomp. Such a class would become intractable and hardly 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.

❖ ❖ ❖

Each of the two players implements the interface `Player`. However, the use of an interface here is different from the way we used the `Dance` interface in the *Dance Studio* project in Chapter 11. There we used `Dance` to isolate the "dance" subsystem from the rest of the program and to be able to pass different types of `Dance` parameters to a `Dancer`'s `learn` method. Here, as you can see in Figure 12-9, `Chomp` is well aware of both players, so `Player` does not isolate anything from anything. 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 players in the future if we want.

A secondary common 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.

But the `Strategy` interface works exactly the same way as `Dance` in *Dance Studio*: it isolates the subsystem that deals with different Chomp strategies from the rest of the program. A more advanced version of the *Chomp* program will need to support different strategies for different levels of play and board sizes. 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.

❖ ❖ ❖

You might have noticed that our designs of *Craps*, *Dance Studio* 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 all three projects mentioned above, we used the *Model-View-Controller* (*MVC*) design pattern. 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." In *Dance Studio*, `DanceGroup` is the "model," `DanceFloor` is the "view," and `ControlPanel` and `Band` are the "controllers."

Dance Studio and *Chomp* also follow the *Strategy* design pattern. This design pattern applies when a "player" needs to follow different "strategies" at different times. Rather then making the player itself create all the strategies and choose among them, we provide an interface or an abstract class to represent a strategy and a "setStrategy" method in the player object. A "strategy" and a "player" do not have to be interpreted literally, of course. For example, in *Dance Studio*, a particular dance is a "strategy" for a dancer and the `Dancer`'s `learn` method is used to set the `Dancer`'s "strategy."

We will explain several common design patterns in more detail in Chapter 26.

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 the rest of the *Chomp's* classes provided in `JM\Ch12\Chomp`.

12.12 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 `a` that holds the number of elements in `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 2-D 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 2-D array is treated basically as a 1-D array of 1-D arrays, its rows. If `m` is a 2-D 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).

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 types — they are converted to objects of the corresponding wrapper class.

The most commonly used `ArrayList` methods are shown in Figure 12-3. 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` and `indexOf` methods help to find a given value in the list.

`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. Both `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.

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). Usually elements are not added or removed during traversal.

The "for each" loop —

```
for (E elmt : list)
{
  ...
}
```

provides a convenient way to traverse an array or a list from beginning to end. Use nested conventional `for` loops to traverse a 2-D array:

```
for (int r = 0; r < m.length; r++)
{
  for (int c = 0; c < m[0].length; c++)
  {
    ...
  }
}
```

Exercises

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) The program, when running, 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 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 (i.e. `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 n = v.length, count = 0;

  for (int i = 0; i < n; i++)
  {
    if (v[i] != 0) break;
    count++;
  }
  return count;
}
```

7. 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 arguments, `m` and `n`, swaps them if necessary to make `m > n`, calculates the Pythagorean triple using the above expressions, 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.

8. Complete the following method:

```
// Returns an array filled with values
//  1, 2, ..., n-1, n, n-1, ..., 2, 1.
public static int[] createWedge(int n)
{
  ...
}
```

9. 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 – 3	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 either `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. ⋝

10.◆ Change the `SnackBar` class from the lab in Chapter 9 (`JM\Ch09\SnackBar\SnackBar.java`) to be able to support any number of vending machines. Assume that the number of machines is passed to the program as a command-line argument. If not given, then the program should use the default number, three machines. Assign brand names and colors to the machines randomly from the three names and colors specified in the `SnackBar` class.

11. 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`. _____ ✓

12. Write a method that takes an `ArrayList` and returns a new `ArrayList` in which the elements are stored in reverse order. The original list should remain unchanged. ✓

13. Write a method that removes the smallest value from an `ArrayList<Integer>`. ⋜ Hint: `Integer` has a method `compareTo(Integer other)` that returns the difference of this `Integer` and other. ⋝

14. 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, <u>not</u> `equals`. ⫷ Hint: the `contains` and `indexOf` methods cannot be used. ⫸

15.■ Write and test a method

```
public void removeDuplicates(ArrayList<String> words)
```

that removes duplicate words (that is, for which `word1.equals(word2)`) from the list. ⫷ Hint: start with the last element and use the `indexOf` and `remove` methods. ⫸

16. Can an `ArrayList<Object>` be its own element? Test this hypothesis.

17. Find and fix the bug in the following code:

```
char[] hello = {' ', 'h', 'e', 'l', 'l' 'o'};
int i = 0;

// Shift to the left and append '!':
while (i < 6)
{
  hello[i-1] = hello[i];
  i++;
}
hello[5] = '!';
```

18. Write 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)
```

19.■ (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. ✓

20.■ 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: the size of the array is >= 2.
public static double averageTopTwo(int[] scores)
{
  int i, size = 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 < size; i++)
  {
    if (scores[i] > scores[iMax1])
    {
      _____________________________________
    }
    else if ( _______________________________ )
    {
      _________________________
      ...
    }
  }
  return __________________________;
}
```

21.■ Write a method that removes the smallest and the largest elements from a given array of `int`s.

22.■ (a) Write and test a method that computes the value of a polynomial $P(x) = a_0 + a_1x + \ldots + a_nx^n$ for a given x: ✓

```
// Returns the value of the n-th degree polynomial
// P(x) = a[0] + a[1]*x + a[2]*x^2 + ... + a[n]*x^n
// where the length of a is n + 1
public double polynomial(double[] a, double x)
```

(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 needs the same number of additions but fewer multiplications. Modify the method you wrote in Part (a) for the second implementation.

23. (a)■ Write and test a class `Polynomial` that represents a polynomial $P(x) = a_0 + a_1x + \ldots + a_nx^n$. Use an `ArrayList` of `Double`s of size n + 1 to hold the coefficients. Provide a constructor

```
    public Polynomial(double[] coefficients)
```

that sets the coefficients by copying them from a given array. Make sure `Polynomial` objects are immutable. Provide one method to return the degree of the polynomial, n, and another to calculate this polynomial's value for a given x, using the formula from Question <...>.

(b)◆ Write a method that multiplies this polynomial by another polynomial and returns their product.

24.■ 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.

(a) Write the following method that calculates and returns the sum of two "large integers" *a* and *b*:

```
private final int N = 100;

// Calculates the sum of two "large integers" a and b,
// represented as arrays of digits, and returns their sum
// as an array of N digits.
// Precondition: the sum fits into N digits.
public static int[] add(int[] a, int[] b)
{
  ...
}
```

(b) Write a test program that defines and initializes (or lets the user enter) two arrays of digits and displays their "large" sum.

25.◆ A partition of a positive integer *n* is its representation as a sum of positive integers $n = p_1 + p_2 + ... + p_k$. Write a method that prints out all possible partitions of a given positive integer *n*. Consider only partitions where $p_1 \le p_2 \le ... \le p_k$. ⁝ Hint: use an `ArrayList` of `Integer`s of capacity *n* to represent a partition; write a recursive helper method `displayPartitions(int n, List list)`. If the sum of the values in `list` is *n*, just print these values. If the sum is less than *n*, try appending to the list another integer, in the range from the last value in the list (or 1, if the list was empty) to *n* - *sum*, and call `displayPartitions(n, list)` again. Don't forget to remove the last added value before trying the next one. ⁝

26. 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. ✓

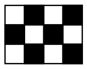
27. Write a method that returns the value of the largest positive element in a 2-D array, or 0 if all its elements are negative: ✓

```
// Returns the value of the largest positive element in
// the matrix m, or 0, if all its elements are negative.
private static double positiveMax(double[][] m)
```

28. Write a method

```
public void fillCheckerboard(Color[][] board)
```

that fills `board` with alternating black and white colors in a checkerboard pattern. For example:



29. Let us say that a matrix (a 2-D 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)
{
  ...
}
```

30.■ 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 generates the first *n* rows of Pascal's Triangle. The method should return a "jagged" 2-D array with *n* rows. The *k*-th row should be a 1-D array of length *k* that holds the values from the *k*-th row of Pascal's Triangle.

31.■ (a) Modify the *Chomp* program so 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? ✓

32. Turn the *Chomp* program into a game for three players: two human players and one computer player.

33.■ (a) In the *Chomp* program, make the board scaleable. Eliminate the `CELLSIZE` constant in the `BoardPanel` class and obtain the cell's width and height from the current dimensions of the panel when necessary.

(b) 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? ✓

13ACEHPRT

Searching and Sorting

13.1 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 (such as 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.

13.2 `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 the name of your class. 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. There strings are compared character by character for an exact match. Consider another example, the class `Country` in Figure 13-1. 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 fields.

In order to override `Object`'s `equals` method in your class, the signature of your `equals` method must be exactly the same as the signature of `equals` in `Object`. In particular, the declared type of the parameter `other` must be `Object`.

```
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 13-1. `JM\Ch13\Compare\Country.java`

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 in `Object`. You might say: "So what? I only intend to compare countries to each other, not to other types of objects." 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.

"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"))
  ...
```

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 (coutry.getName().equals("USA"))
  ...
```

or

```
Country usa = new Country("USA");
if (coutry.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 an 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 - 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 above example, 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 to be `Comparable` and polymorphically call `compareTo` for your objects. For example, the `java.util.Arrays` class has a `sort(Object[] arr)` method that expects the elements of `arr` to be `Comparable`. 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 `Countries` comparable. You can sort them 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. A comparator belongs to a class that implements the `java.util.Comparator<T>` interface and has a method

```
   public int compare(T obj1, T obj2)
```

where *`T`* is the name of a class.

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 - 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 13-2 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 13-2. `JM\Ch13\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 13-3). The output from the main method in the figure is

```
[Brazil: 190, China: 1321, India: 1110, Indonesia: 249, USA: 301]
[Brazil: 190, Indonesia: 249, USA: 301, India: 1110, China: 1321]
[China: 1321, India: 1110, USA: 301, Indonesia: 249, Brazil: 190]
```

```
import java.util.Arrays;

public class ComparatorTest
{
  public static void main(String[] args)
  {
    Country[] countries =
      { // population in millions
        new Country("China", 1321),
        new Country("India", 1110),
        new Country("USA", 301),
        new Country("Indonesia", 249),
        new Country("Brazil", 190),
      };

    // 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 13-3. `JM\Ch13\Compare\ComparatorTest.java`

13.3 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 always 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. We explain it more formally in Chapter 18.)

In Chapter 4 we introduced a more efficient algorith, called *Binary Search*, 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. In any event, each time we repeat the same procedure we narrow the range of our search by half (see Figure <...> on page <...>). 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 13-4 implements the Binary Search algorithm for an array of `Comparable` objects sorted in ascending order.

```
// 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 binarySearch(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, middle;

  while (left <= right)
  {
    // Take the index of the middle element between
    //   "left" and "right":

    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]
      left = middle + 1;
    else if (diff < 0) // target < a[middle]
      right = middle - 1;
    else   // target is equal to a[middle]
      return middle;
  }

  return -1;
}
```

Figure 13-4. `JM\Ch13\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.

13.4 *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 IllegalAgrumentException("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.
6. Define a new method `merge(SortedWordList additionalWords)`. This method should insert into this list in alphabetical order all the words from `additionalWords` that are not already in this list. `merge` should be efficient. You may not use any temporary arrays or lists. Each element from this list should move <u>at most once</u>, directly into its proper location. To achieve this while avoiding `IndexOutOfBounds` errors, you first need to add some dummy elements to the list. Save the current size of the list, then append to it n arbitrary strings, where n = `additionalWords.size()`. Call `super.add("")` to append an empty string or just call `addAll(additionalWords)` once. Now merge the lists, starting at the end of each list and at the end of the added space. At each step decide which of the two lists should supply the next element for the next vacant location.

7. Combine your class with `SortedListTest.java` in `JM\Ch13\SortedList` and test the program.

13.5 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*. We have already mentioned it in Chapter <...>; we present it here again for convenience:

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 13-5 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 here, 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[size-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 13-5. `JM\Ch13\Benchmarks\SelectionSort.java`

13.6 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 13-6 implements this algorithm for an array of `double`s.

```
public class InsertionSort
{
  // Sorts a[0], ..., a[size-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 backwards 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 13-6. `JM\Ch13\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.

13.7 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 arguments, and all the local variables of a method (see Section <...>). 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 13-7 shows a `Mergesort` class that can sort an array of `doubles`. 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[size-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);
    }
  }
```

Figure 13-7 `Mergesort.java` *Continued* ⇗

```
  // 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)
    {
      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 13-7. `JM\Ch13\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.

13.8 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 13-8 shows a Java implementation, in which the pivot is the element in the middle of the array.

To partition the array, you proceed from both ends of the array towards 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 update both indices. When the left and the right side meet at a certain position, you swap the pivot with one of the elements that have met.

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.

```
public class Quicksort
{
  // Sorts a[0], ..., a[size-1] in ascending order
  //   using the Quicksort algorithm
  public static void sort(double[] a)
  {
    recursiveSort(a, 0, a.length - 1);
  }
```

Figure 13-8 `Quicksort.java` *Continued* ⇨

```
  // 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 element 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--;
      }
    }

    // Place the pivot in its correct position:
    if (p < j)
    {
      swap (a, j, p);
      p = j;
    }
    else if (p > i)
    {
      swap (a, i, p);
      p = i;
    }

    // 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 13-8. `JM\Ch13\Benchmarks\Quicksort.java`

13.9 *Lab:* Benchmarks

A benchmark is an empirical test of performance. The program in Figure 13-9 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 total time it took to run all the trials.

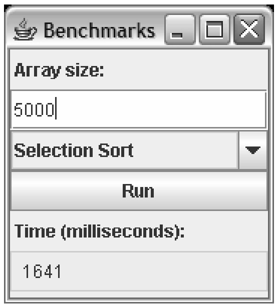


Figure 13-9. The *Benchmarks* program

The trials are performed by the `runSort` method, which returns the total time spent sorting an array filled with random values. Your task is to write this method.

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 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 random number generator 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 the same sequence of random numbers. If you want different sequences on different runs of your program, you need different seeds. A common technique is to use the current system time as a seed. In our program we call the `currentTimeMillis` method of the `System` class to obtain a value for the seed. (Recall that `System` is imported into all programs and has only static fields and methods.) Once we obtain a seed value, we initialize the random number generator with that seed at the beginning of `runSort` because we want to run all the sorting methods on exactly the same data.

Java's `util` package provides a class `Random` for generating random numbers. `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 the seed as a parameter. For example:

```
 Random generator = new Random(seed);
```

You should create a `Random` object at the beginning of `runSort` using this constructor. After that you can generate the next random `double` by calling this object's `nextDouble` method. For example:

```
        a[k] = generator.nextDouble();
```

The `runSort` method fills the array with random numbers and then sorts it. This is repeated `RUNS` times, and `runSort` returns the total time it took in milliseconds. Call the system time before and after each trial 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\Ch13\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 each of the 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 an *n* log *n* curve for Mergesort and Quicksort.

13.10 The `Arrays` and `Collections` Classes

No matter what you want to do in Java, 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` 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,

```
String[] names = {"Ann", "Kate", "Zoe"};
System.out.println(Arrays.toString(names));
```

produces the output

```
[Ann, Kate, Zoe];
```

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.

13.11 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 signature:

```
   public 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 - 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` (sort of like `"this - 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 to 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:

<u>Selection Sort</u>

Set *n* to the size of the array. While *n* is greater than 2, 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.

<u>Insertion Sort</u>

Keep the first *n* elements sorted. Starting at $n = 2$, repeat the following: find the place of `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.

<u>Mergesort</u>

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.

<u>Quicksort</u>

Choose a pivot element and partition the array, so that all the elements on the left side are less than or equal to the pivot and all the elements on the right side 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. It 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` methods work with `List`s, such as `ArrayList`, and other Java collections.

Exercises

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 9, `Comparable`. The natural ordering for `Fraction`s should be the same as for the rational numbers that they represent. Also redefine the `equals` method 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
```

Continued ⇗

(b) Override the `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*-coeffients; 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*.)

(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*.

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. What is the number of comparisons in Binary Search required in the worst case to find a target value in a sorted array of 80 elements? Consider two scenarios:

(a) We know for sure that the target 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 method that finds the number of X's in the string in $\log_2 n$ steps, where *n* is the length of the string.

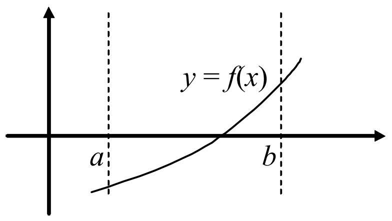
8.■ Write a recursive method that tries to find `target` among the elements `a[m]`, ..., `a[n-1]` of a given array (any array, not necessarily sorted).

```
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 ($m \ge n$). 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.

10.■ A divide-and-conquer algorithm can be used to find a zero (root) of a function. Suppose a function $f(x)$ is a continuous function on the interval $[a, b]$. Suppose $f(a) < 0$ and $f(b) > 0$:



The graph of the function must cross the x-axis at some point. We can split the segment into two halves and continue our search for a zero in the left or the right half, depending on the value of $f(x)$ in the middle point $x = (a + b) / 2$.

Write a program that finds x (to the nearest .001), such that $x = \cos(x)$. ⸨ Hint: consider the function $f(x) = x - \cos(x)$ on the interval $[0, \pi/2]$. ⸩ ✓

11.◆ 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?

12. 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. ______ ✓

13. Write a method `void shuffle(Object[] arr)` that shuffles the objects in `arr` in random order. ⸨ Hint: the algorithm is very similar to Selection Sort, only on each iteration, instead of finding the largest element, you choose a random one. ⸩

14. 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 implemented in Figure 13-5).

15. What are the values stored in an array `a` after five iterations in the `for` loop of the Insertion Sort (as implemented in Figure 13-6) if the initial values are 3, 7, 5, 8, 2, 0, 1, 9, 4, 3? ✓

16. 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? ✓

17. Add `Arrays`'s built-in sorting method to the *Benchmarks* program to see how it compares to our own sorting methods.

`System.out.println`

`("Chapter 14");`

Streams and Files

14.1 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. 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 CD-ROM 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 <u>logical</u> stream. For example, when we read from a disk file, the input is *buffered* so that bytes or characters are read into the program not directly from the disk but from an intermediate buffer in memory. 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.

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.

14.2 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 `Ch14` subdirectory (subfolder) in the `JM` directory (folder) on drive `C` is `C:\JM\Ch14\words.txt`. This is called an *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 `Ch14\Classes`, and `words.txt` were in `Ch14\Data`, we would write

```
     File wordsFile = new File("..\\Data\\words.txt");
```

`"\\"` stands for a single backslash. `".."` means "go one directory level up." (You can also use forward slashes, in the Unix style: `"../Data/words.txt"`.)

Figure 14-1 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 14-1. 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 14-2 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 14-2. A `JFileChooser` example

14.3 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 <u>not</u> a pathname, but rather a string to be used as an input stream.

If a `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 14-3.

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 14-3. 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` works with tokens 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 or `double`s). For example:

```
int num = input.nextInt();
input.nextLine();              // skip the rest of the line and newline
Sring 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 14-4 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 14-4. Reading a file character by character using a `BufferedReader`

14.4 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 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 <...>).

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.

Figure 14-5 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 14-5. Creating a text file

❖ ❖ ❖

If you want to append text to an existing file, use a `PrintWriter` “wrapped” around a `FileWriter` (Figure 14-6).

```
    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 14-6. Appending data to a text file

14.5 *Lab:* Dictionary Maker

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 applet. You will start with a text file, `words.txt`, which contains about 20,000 English words. 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 14-7. 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.

Chapter 18

Big-O Analysis of Algorithms

18.1 Prologue

This chapter opens our discussion of a series of data structures that constitute a standard set of tools in software design and development. These include lists, stacks, queues, binary trees, priority queues, hash tables, and other structures.

A data structure combines data organization with methods of accessing and manipulating the data.

For example, an array becomes a data structure for storing a list of values when we provide methods to find, insert, and remove a value. Similar functionality can be achieved with a *linked list* instead of an array (we will explain linked lists in Chapter 20.) At a very abstract level, we can think of a general "list" object: a list contains a number of values arranged in sequence; we can find a target value in a list, add values to the list, and remove values from the list. This abstract structure is called an *abstract data type* (ADT).

The data structures that we are going to study are not specific to Java — they can be implemented in any programming language. In Java, a data structure can be described as an interface and implemented as a class. There can be several implementations of the same abstract data type. `java.util.ArrayList` and `java.util.LinkedList`, for example, both implement the interface `java.util.List`.

Before we proceed with the data structures, though, we need to acquire some background knowledge of how to evaluate and compare the efficiency of different algorithms that work with these structures. Algorithms and data structures are often analyzed in terms of their time efficiency and space requirements. These are the concerns of a branch of computer science called *computational complexity* or *operations research*. In this book we concentrate on time efficiency.

As we saw in the *Benchmarks* program in Chapter 13, one obvious way to measure the time efficiency of an algorithm is to implement it in a computer program, run that program on various sets of input data, and measure the running time. It may seem that benchmarks leave little room for theory, but this first impression is incorrect. While benchmarks depend on the details of implementation (such as the actual code, the programming language, the optimizing capabilities of the compiler, and so on), as well as on the CPU speed and other hardware characteristics, it turns out it is possible to study the efficiency of algorithms excluding all these practical matters. The *big-O* concept serves this purpose. We already used it informally when we compared various searching and sorting algorithms. In this chapter we will discuss big-O in a more mathematically rigorous way.

The theoretical study of algorithms relies on an abstract model of a computer as a device with some defined capabilities, such as the capability to perform arithmetic operations and to store and retrieve data values in memory. The abstract model disregards all specific features of a particular computer system, such as the CPU speed and RAM size. The theoretical approach requires two simplifications. First, we have to stop measuring performance in real time (which depends on CPU speed, etc.). Nor do we measure it in terms of the number of required program instructions or statements (which depends on the language, compiler, implementation, etc.). Instead, we discuss performance in terms of some abstract "steps" that are necessary to complete the task.

What constitutes a "step" depends on the nature of the task. In a searching task, for example, we may define one step as one comparison between the target value and a data value in the list. In calculating a Fibonacci number iteratively, one step may be defined as one addition. The total number of required steps may depend on the size of the task, but it is assumed that each step takes the same amount of time. With this approach we cannot say how long a particular implementation of an algorithm might run on a particular computer system, but we can compare different algorithms that accomplish the same task.

The second simplification is that our theoretical analysis applies only when the task size is a large number. Let us denote the total number of steps that an algorithm requires to complete the task as $T(n)$. $T(n)$ is some function of the task size n. The theoretical approach focuses on the behavior of $T(n)$ for large n, which is called *asymptotic behavior*. In the following section we will see why knowing the asymptotic behavior of an algorithm is important and how it can be expressed in formal mathematical terms.

In the remainder of this chapter we will discuss a more formal definition of "big-O," consider some examples, and review the big-O performance of several sorting algorithms.

18.2 Big-O Notation

As a starting point for our discussion, let us compare two familiar searching algorithms in an array, Sequential Search and Binary Search. We will assume that the elements in the array are arranged in ascending order.

Recall that in the Sequential Search algorithm we simply try to match the target value against each array value in turn until we find a match or finish scanning the whole array. Suppose this algorithm is implemented as follows:

```
for (int i = 0; i < n; i++)
{
  if (a[i] == target)
    break;
}
```

The total running time includes the time it takes for the initialization and for the iterations through the loop. (In this example, the initialization is simply setting `i` equal to `0`.) Assuming that the average number of iterations is $n/2$, the average time may be expressed as:

$$t(n) = t_{\text{init}} + t_{\text{iter}} \cdot n/2$$

where t_{init} is the initialization time and t_{iter} is the time required for each iteration. In other words, the average time is a linear function of n:

$$t(n) = An + B$$

As n increases, An also increases, and the relative contribution of the constant term B eventually becomes negligible as compared to the linear term An, even if A is small and B is large. Mathematically, this means that the ratio

$$\frac{t(n)}{An} = \frac{An+B}{An} = 1 + \frac{B}{An}$$

becomes very close to 1 as n increases without bound.

Therefore, <u>for a large n</u> we can drop the constant term and say that the average time is approximately An. That means that the average time for the Sequential Search algorithm *grows linearly* with n (Figure 18-1-(a)).

Now let us consider the Binary Search algorithm applied to the same task. As we know from Chapter 13, if n is between 2^{h-1} and 2^h-1, the worst case will require h comparisons. Thus, the number of comparisons in the worst case is

$$T(n) = \log_2 n + 1 \text{ (truncated to an integer)}$$

For a value randomly chosen from the values in the array, the average number of steps in a binary search is (approximately, for large n) only one less than the worst case.

Again, the total average time consists of the initialization time and the time for average number of iterations through the loop:

$$t(n) = t_{\text{init}} + t_{\text{iter}} \cdot \log_2 n$$

Following the same reasoning as for the sequential search, we conclude that the execution time of the binary search is, for large *n*, approximately proportional to the logarithm of *n*:

$$t(n) = C \log_2 n$$

The coefficient *C* is determined by the time spent in one iteration through the loop. Figure 18-1-(b) shows the general shape of this curve. $\log_2 n$ approaches infinity as *n* increases, but it does so more slowly than the linear growth of a straight line.

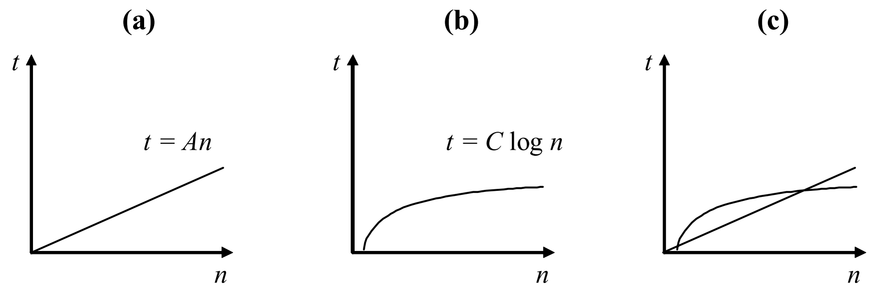


Figure 18-1. (a) Linear growth (b) Logarithmic growth (c) log growth is slower

Note that "one step" in the sequential search is not exactly the same as "one step" in the binary search, because besides comparing the values, we also need to modify some variables and control the iterations. Thus, the coefficients *A* and *C* may be different; for example, *C* may be larger than *A*. For some small *n*, a sequential search could potentially run faster than a binary search. But, no matter what the ratio of *A* to *C*, the linear curve eventually overtakes the logarithmic curve for large enough *n* (Figure 18-1 (c)).

In other words, asymptotically, binary search is faster than sequential search. Moreover, it is not just 5 times faster or 100 times faster. It is faster in principle: you can run sequential search on the fastest computer and binary search for the same task on the slowest computer, and still, if n is large enough, binary search will finish first.

This is an important theoretical result. The difference in the asymptotic behavior of these two searching algorithms provides an important new way of looking at their performance. Binary search time grows logarithmically and sequential search time linearly, so no matter what specific coefficients of the growth functions we use, linear time eventually surpasses logarithmic time.

In this context it makes sense to talk about the order of growth that characterizes the asymptotic behavior of a function, ignoring the particular constant coefficients. For example, $f(n)=n$ has a higher order of growth than $g(n)=\log_2 n$, which means that for any positive constant *k*, no matter how small,

$$k \cdot n > \log_2 n$$

when *n* is large enough. Two functions that differ only by a constant factor have the same order of growth.

The following definition of *big-O* (order of growth) notation helps us formalize this terminology and refine our ideas about the order of growth. "Big-O" is defined as follows:

Given two positive functions $t(n)$ and $g(n)$, we say that

$$t(n) = O\left(g(n)\right)$$

if there exist a positive constant *A* and some number *N* such that

$$t(n) \le Ag(n)$$

for all *n* > *N*.

The big-O definition basically means that $t(n)$ asymptotically (for large enough *n*) grows not faster than $g(n)$ (give or take a constant factor). In other words, the order of growth of $t(n)$ is not larger than $g(n)$.

So, in terms of order of growth, $f = O(g)$ is like "$f \le g$." In practice, when the performance of algorithms is stated in terms of big-O, it usually refers to the "tightest" possible upper bound. In this book, we have chosen to follow the widely accepted practice of using big-O in the sense of "order of growth of *f* = *g*." For example, in our analysis of the two searching algorithms we say that both the worst and average time is $O(n)$ for Sequential Search and $O\left(\log_2 n\right)$ for Binary Search.

❖ ❖ ❖

As we know from the Change of Base Theorem, for any a, $b > 0$, and a, $b \neq 1$

$$\log_b n = \frac{\log_a n}{\log_a b}$$

Therefore,

$$\log_a n = C \log_b n$$

where C is a constant equal to $\log_a b$.

Since functions that differ only by a positive constant factor have the same order of growth, $O(\log_2 n)$ is the same as $O(\log_a n)$. Therefore, when we talk about logarithmic growth, the base of the logarithm is not important, and we can say simply $O(\log n)$.

❖ ❖ ❖

One set of functions that are often used for describing the order of growth are, naturally, powers of n:

$$1, n, n^2, n^3, \ldots..$$

The order of growth for n^k is higher than n^{k-1} .

If a function is a sum of several terms, its order of growth is determined by the fastest growing term. In particular, if we have a polynomial

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0$$

its growth is of the order n^k:

$$p(n) = O\left(n^k\right)$$

Thus, any second-degree polynomial is $O\left(n^2\right)$. This is called *quadratic* growth.

Note that no one uses such things as $O(3n)$ or $O(n^2 / 2)$ because they are the same as $O(n)$ or $O(n^2)$, respectively.

❖ ❖ ❖

Let us consider a common example of code that requires $O\left(n^2\right)$ operations. Suppose we have a nested loop:

```
... // set up the outer loop
for (int i = 1;   i < n;   i++)
{
  ... // set up the inner loop
  for (int j = 0;   j < i;   j++)
  {
    ... // do something
  }
}
```

This kind of code may be used in a simple sorting method (Selection Sort), for finding duplicates in an array, or in some operations on matrices (such as transposing a matrix by flipping an *n* by *n* 2-D array symmetrically over its diagonal).

The outer loop runs for i from 1 to $n-1$, a total of $n-1$ times, and the inner loop runs for j from 0 to $i-1$, a total of i times. The code inside the inner loop will, therefore, execute a total of

$$1+2+...+(n-1)$$

times. Since 1, 2, 3, ... is an arithmetic sequence, its partial sum can be found by taking the number of terms and multiplying it by the average of the first and the last terms:

$$1+2+...+(n-1)=(n-1)\frac{1+(n-1)}{2}=\frac{(n-1)n}{2}$$

Even if we take into account the setup time for each loop, we still get a second-degree polynomial of *n*. Therefore, $t(n)=O\left(n^2\right)$.

❖ ❖ ❖

The time efficiency of almost all of the algorithms discussed in this book can be characterized by one of very few growth rate functions:

I. $O(1)$ — *constant time*. This means that the algorithm requires the same fixed number of steps regardless of the size of the task.

Examples:

A. Finding a median value in a sorted array
B. Calculating $1+2+3+...+n$ using the formula for the sum of an arithmetic sequence
C. Push and pop operations in an efficiently implemented stack; add and remove operations in a queue (Chapter 21)
D. Finding a key in a lookup table or a sparsely populated hash table (Chapter 24)

II. $O(n)$ — *linear time*. This means that the algorithm requires a number of steps proportional to the size of the task.

Examples:

A. Traversing a list with *n* elements, (for example, finding max or min)
B. Calculating *n*-factorial; finding the *n*-th Fibonacci number iteratively
C. Traversing a binary tree with *n* nodes (Chapter 23)

III. $O\left(n^2\right)$ — *quadratic time*. The number of operations is proportional to the size of the task squared.

Examples:

A. More simplistic sorting algorithms, such as Selection Sort of *n* elements
B. Comparing two 2-D arrays of size *n* by *n*
C. Finding duplicates in an unsorted list of *n* elements (implemented with a nested loop)

IV. $O(\log n)$ — *logarithmic time.*

Examples:

A. Binary search in a sorted list of *n* elements
B. Finding a target value in a binary search tree with *n* nodes (Chapter 23)
C. *Add* and *remove* operations in a priority queue, implemented as a heap, with *n* nodes (Chapter 25)

V. $O(n \cdot \log n)$ — *"n log n" time.*

Examples:

A. More advanced sorting algorithms, such as Mergesort and Quicksort

VI. $O(a^n)$ $(a > 1)$ — *exponential time.*

Examples:

A. Recursive Fibonacci implementation ($a \geq 3/2$; see Chapter 22)
B. The Towers of Hanoi ($a = 2$; see Lab <...>)
C. Generating all possible permutations of *n* symbols

The best time in the above list is obviously constant time, and the worst is exponential time, which overwhelms even the fastest computers even for relatively small *n*. *Polynomial* growth (linear, quadratic, cubic, etc.) is considered manageable as compared to exponential growth.

Figure 18-2 shows the asymptotic behavior of the functions from the above list. Using the "<" sign informally, we can say that

$$O(1) < O(\log n) < O(n) < O(n \cdot \log\ n) < O\left(n^2\right) < O\left(n^3\right) < O\left(a^n\right)$$

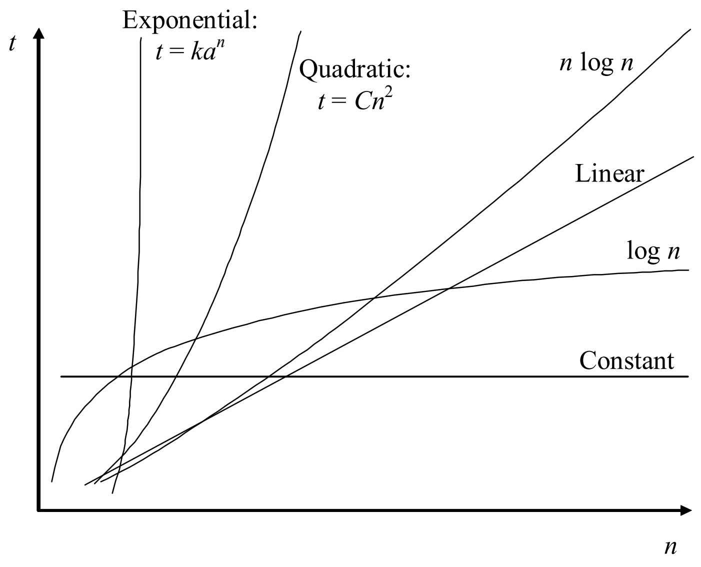


Figure 18-2. Rates of growth

The slow asymptotic growth of $\log_2 n$ (in comparison to linear growth) is especially dramatic. Every thousand-fold increase in the size of a task results in only a fixed, fairly small increment in the required number of operations. Consider the following:

$$\log_2 1000 \approx 10;\ \log_2 10^6 \approx 20;\ \log_2 10^9 \approx 30;\ \text{...etc.}$$

This property is used in many efficient "divide and conquer" algorithms such as Binary Search and is the basis for using binary search trees and heaps.

18.3 Sorting: a Big-O Review

In this section we quickly review the sorting algorithms we have studied and discuss their big-O properties more formally.

In general, the task of sorting is understood as rearranging the values of a list in ascending (or descending) order. The usual abstract formulation of this task assumes that we need to compare two values from the list to decide which one is smaller. What matters is the result of each comparison, not the values themselves. As we will see in Exercise <...> in Chapter 24, such methods as Radix Sort allow us to sort values using some kind of lookup tables without any comparisons at all. These methods rely on information about the range and "composition" of values in the list. For example, if we know that the values in the list are in the range from 0 to 9, we can simply count the number of occurrences for each value and then recreate the list in order. The performance of such algorithms is $O(n)$. But here we deal with the algorithms that require actual comparisons.

Mathematically speaking, we assume that we have a list of objects, $X = \{x_0, ..., x_{n-1}\}$, and that an ordering relation < is defined on X. This relation is called *total ordering* if for any two values a and b in X, exactly one of three possibilities is true: either $a < b$, or $b < a$, or $a = b$. An ordering relation must be *transitive*: if $a < b$ and $b < c$, then $a < c$. A sorting algorithm is a strategy for finding a permutation of the indices $p(0), p(1), ..., p(n-1)$ such that

$$x_{p(0)} \le x_{p(1)} \le ... \le x_{p(n-1)}$$

(A permutation is a one-to-one mapping from the set of integers $\{0, 1, 2, ... n-1\}$ onto itself.)

The sorting strategy is based on pairwise comparisons of values. The decision as to which next pair of values to compare may be based on the results of all the previous comparisons. In these terms, how fast can we sort? Clearly if we compare each value with each other value, it will take n^2 comparisons. As we know, Mergesort and Quicksort can do the job in only $O(n \cdot \log n)$ comparisons. Can we do any better? The answer is no.

If you are limited to "honest" comparison sorting, any sorting algorithm in its worst case scenario takes at least $O(n \cdot \log n)$ steps.

To prove this, envision our sorting strategy as a decision tree: in each node we compare two values, then go left or right depending on the result (Figure 18-3). The node at the end of each path from the root holds a different permutation of values. The number of comparisons in the worst case is the length of the longest path. If the length of the longest path in the tree is h, then the number of leaves (terminal nodes) does not exceed 2^h. The number of all possible permutations of n values is $n!$. So we must have

$$2^h \ge n!$$

Therefore

$h \geq \log_2 (n!)$

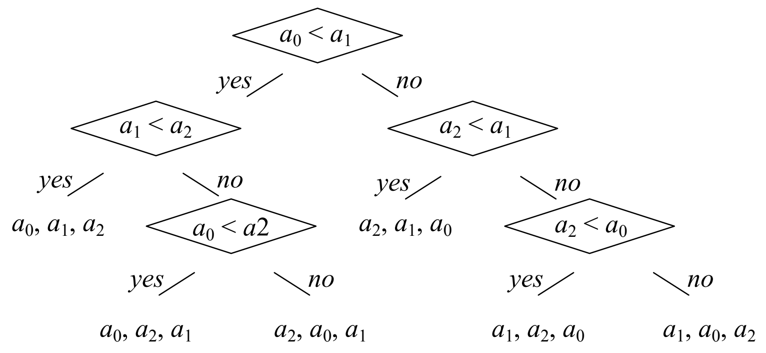


Figure 18-3. A sorting algorithm as a decision tree (for three values)

For example, for $n = 4$, $n! = 24$, so the best algorithm needs 5 comparisons; for $n = 5$ it needs 7 comparisons. An algorithm with 7 comparisons for 5 values indeed does exist, but it is not at all obvious. Asymptotically, $\log_2(n!) = n \cdot \log n$,[*] so faster sorting is not possible.

This is as far as we want to go with theory. Now let us review a few slower $O(n^2)$ algorithms and then a few more advanced $O(n \cdot \log n)$ algorithms.

Sorting by Counting

In the most simplistic approach, we can compare each value in X to each other value. This means $n \cdot (n-1)$ or $O(n^2)$ comparisons. Indeed, for each x_i in X a computer program can simply count how many other values x_j do not exceed x_i. (If $x_j = x_i$ we only count it when j < i.) Then the position of x_i in the sorted list, $p(i)$, is set to the resulting count. This is called sorting by counting:

[*] It is not hard to prove this fact using calculus. $\ln(n!) = \ln 1 + \ln 2 + ... + \ln n$, which can be viewed as a Riemann sum for $\int_1^n \ln x \, dx = (x \ln x - x)\Big|_1^n = n \ln n - n + 1 = O(n \cdot \log n)$.

```
for (int i = 0; i < n; i++)
{
  int count = 0;
  for (int j = 0; j < n; j++)
  {
    if (a[j] < a[i] || (a[j] == a[i]  && j < i))
      count++;
  }
  b[count] = a[i];
}
```

The algorithm is inefficient because it makes a lot of redundant comparisons. If we have already compared x_i to x_j, there is no need to compare x_j to x_i.

Selection Sort

A slight improvement is achieved in Selection Sort. It cuts the number of comparisons in half, but the big-O for the Selection Sort algorithm is still $O\left(n^2\right)$.

The number of comparisons in the counting algorithm and in Selection Sort does not depend on the initial arrangement of the values in the list. For example, if the list is already sorted, these algorithms still go through all the prescribed steps. There is no "best case" or "worst case" scenario: the number of executed steps is always the same.

Insertion Sort

Slightly more sophisticated sorting algorithms may take advantage of a particular initial arrangement of values in the list to reduce the number of comparisons. For example, Insertion Sort keeps the beginning fragment of the list sorted and inserts each next element into it in order. On average, Insertion Sort takes half the number of comparisons of Selection Sort, but it is still an $O\left(n^2\right)$ algorithm. In the worst case, when the array is sorted in reverse order, the number of comparisons is the same as in Selection Sort, $n(n-1)/2$. But if the array is already sorted, the inner `for` loop performs only one comparison. If the array is nearly sorted, with only a couple of elements out of place, then Insertion Sort takes only $O(n)$ comparisons.

Mergesort

In Mergesort, we split the array into two equal (or almost equal) halves. We sort each half recursively, then merge the sorted halves into one sorted array. All the comparisons are actually performed at the merging stage — the sorting stage merely calls the method recursively for each half of the array (see Figure <...> on page <...>).

This version of Mergesort predictably takes $O(n \cdot \log n)$ comparisons regardless of the initial configuration of the array. With a shortcut, discussed in Question <...> in the exercises to this chapter, it only takes n comparisons when the array is already sorted.

Quicksort

The idea of Quicksort is to choose one element, called the pivot, then partition 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. After partitioning, Quicksort is applied (recursively) to the left-of-pivot part and to the right-of-pivot part.

In general, you have to be careful with this type of recursion: you have to think not only about performance, but also about the required stack space. If everything goes smoothly, Quicksort partitions the array into approximately equal halves most of the time. Then the array length is divided roughly by two for each recursive call, recursion quickly converges to the base case, and the required stack space is only $O(\log n)$. But what happens if we always choose the first element of the array as the pivot and the array happens to be sorted? Then partitioning does not really work: one half is empty, while the other has all the elements except the pivot. In this case, the algorithm degenerates into a slow version of Selection Sort and takes $O(n^2)$ time. What's worse, the depth of recursion becomes $O(n)$, and, for a large array, it may overflow the stack. (There is a version of Quicksort that overcomes the possible stack overflow problem by handling the shorter half of the array recursively and the longer half iteratively.)

Heapsort

We will consider this interesting algorithm in Chapter 25, after we learn about heaps.

18.4 Summary

The efficiency of algorithms is usually expressed in terms of asymptotic growth as the size of the task increases toward infinity. The size of a task is basically an intuitive notion: it reflects the number of elements involved or other similar parameters. The asymptotic growth may be expressed using "big-O" notation, which gives an upper bound for the order of growth. In practice, the big-O estimate is usually expressed in terms of the "tightest" possible upper bound.

The most common orders of growth (in increasing order) are

$O(1)$ — constant;
$O(\log n)$ — logarithmic;
$O(n)$ — linear;
$O(n \cdot \log n)$ — "n log n";
$O(n^2)$ — quadratic;
$O(a^n)$ — exponential.

Logarithmic growth is dramatically slower than linear growth. This explains the efficiency of "divide and conquer" algorithms, such as Binary Search, and of binary search trees and heaps (Chapters 23 and 25). Discovering an $O(\log n)$ algorithm instead of an $O(n)$ algorithm or an $O(n \cdot \log n)$ instead of an $O(n^2)$ algorithm for some task is justifiably viewed as a breakthrough in time efficiency. Exponential growth is unmanageable: it quickly puts the task out of reach of existing computers, even for tasks of rather small size.

Sorting algorithms provide a fertile field for formulating and studying general properties of algorithms and for comparing their efficiency. Simplistic sorting algorithms, such as Sorting by Counting, Selection Sort, and Insertion Sort, accomplish the task in $O(n^2)$ comparisons, although Insertion Sort may work much faster when applied to an array that is already sorted or almost sorted. More advanced algorithms, such as Mergesort, Quicksort, and Heapsort, accomplish the task in $O(n \cdot \log n)$ comparisons.

Sorting is a very well studied subject. There are hundreds of other algorithms and variations. There is no single "best" sorting algorithm. In the real world, the choice depends on specific properties of the target data and additional constraints and requirements.

Table 18-1 shows the average, best, and worst cases for the algorithms that we have discussed. The worst case, in general, assumes that the algorithm is running on the set of data that takes the longest time to process.

	Best case	**Average case**	**Worst case**
Sequential Search	$O(1)$ — found right away	$O(n)$ — found on average in the middle	$O(n)$
Binary Search	$O(1)$ — found right away	$O(\log n)$	$O(\log n)$
Selection Sort	$\mathrm{O}(n^2)$	$\mathrm{O}(n^2)$	$\mathrm{O}(n^2)$
Insertion Sort	$\mathrm{O}(n)$ — array already sorted	$\mathrm{O}(n^2)$	$\mathrm{O}(n^2)$
Mergesort	$O(n \log n)$, or $O(n)$ in a slightly modified version when the array is sorted	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$ — pivot is consistently chosen far from the median value, as when the array is already sorted and the first element is chosen as pivot
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Table 18-1. Best, average, and worst case Big-O for sorting and searching

Exercises

1. Mark true or false and explain:

(a) In comparing the order of growth, $f = O(g)$ is like “$f \le g$.” ______ ✓
(b) We often informally say $f = O(g)$ when we actually mean that the order of growth of *f* and *g* is the same. ______
(c) $\log_2 n = O(\log_{10} n)$. ______ ✓
(d) If $f(n) = 1 + 2 + ... + n$, then $f(n) = O(n^2)$. ______
(e)■ $(\log_2 n)^2 = O(n)$. ______ ✓

2. What is the big-O of the number of operations required to perform the following tasks, assuming reasonably optimized implementation?

(a) Transposing a square matrix of size *n* ✓
(b) Reversing an array of *n* elements
(c) Finding the number of pairs of consecutive double letters in a character string of length *n* ✓
(d) Selection Sort of an array with *n* elements

3. What is the worst-case running time (big-O, in terms of *n*) of the following methods?

(a)

```
public int maxCluster(double[] v, double d)
{
  int n = v.length, maxCount = 0;

  for (int i = 0; i < n; i++)
  {
    int count = 0;
    for (int j = 0; j < n; j++)
      if (Math.abs(v[i] - v[j]) < d)
        count++;
    if (count > maxCount)
      maxCount = count;
  }
  return maxCount;
}
```

(b)

```
public boolean isPalindrome(String w)
{
  int n = w.length(), i = 0, j = n-1;
  while (i < j && w[i] == w[j])
  {
    i++;
    j--;
  }
  return i >= j;
}
```

(c)

```
public double pow(double x, int n)
{
  double y = 1;

  if (n > 0)
  {
    y = pow(x, n/2);
    y *= y;
    if (n % 2 != 0)
      y *= x;
  }
  return y;
}
```

4. Indicate whether the specified task requires logarithmic (*L*), polynomial (*P*) or exponential (*E*) time (assuming an optimal implementation):

(a) Counting how many numbers in a set of *n* values are equal to the average of any two or more values in the same set

(b) Concatenating all strings of length 5 stored in a square matrix

```
private String words[][] = new String[n][n];  ✓
```

(c) Finding the highest power of 2 that evenly divides a given positive integer *n*

5.■ The code below calculates the sum of the elements in a 2-D array `matrix` that belong to a "cross" formed by an intersection of a given row and column:

```
int sum = 0;

for (int r = 0; r < matrix.length; r++)
{
  for (int c = 0; c < matrix[0].length; c++)
  {
    if (r == row || c == col)
      sum += matrix[r][c];
  }
}
sum -= matrix[row][col];
```

Find the big-O for this code when `matrix` is an *n* by *n* array, and rewrite the code to improve its big-O.

6.◆ Suppose the array `s` contains a digitized seismogram recorded for 5 seconds during a remote earthquake. We want to find the moment of the seismic wave's arrival. Suppose the seismogram is digitized at *n* samples per second and the array contains 5*n* amplitudes of the signal. The code below averages the signal over one-second intervals and chooses the largest average to detect the arrival of the first wave:

```
double sum, max;
int kMax;

// Find the sum starting at i = 0:
sum = 0.0;
for (int i = 0; i < n; i++)
  sum += s[i];
max = sum;
kMax = 0;

for (int k = 1; k < 4*n; k++)
{
  // Find the sum starting at i = k:
  sum = 0.0;
  for (int i = k; i < k + n; k++)
    sum += s[i];
```

Continued ➯

```
      // Update max:
      if (sum > max)
      {
        max = sum;
        kMax = k;
      }
    }
    return kMax;
```

What is the big-O for this code in terms of *n*? Is there a way to improve it?

7.◆ Suppose we have an integer function $f(i)$ with an integer argument, and we define a sequence g_i as follows:

$$g_0 = 1;\ g_i = f(g_{i-1}) \text{ for } i > 0.$$

We want to know whether and when the values $g_0, g_1, \ldots$ start repeating. In other words, for a given *n* we want to know whether there exist *k* and *m*, $0 \le k < m \le n$, such that $g_k = g_m$, and if so to find the smallest such *k*.

Consider the following code written for this task:

```
private int g(int i)
{
  if (i == 0)
    return 1;
  else
    return f(g(i-1));
}

public int hasRepeats(int n)
{
  for (int k = 0; k < n; k++)
  {
    for (int m = k + 1; m <= n; m++)
    {
      if (g(k) == g(m))
        return k;
    }
  }
  return -1;
}
```

What is the big-O for this code's time and space requirements in terms of *n*? Can they be improved? Can this task be done in $O(n)$ time?

8. Mark true or false and explain:

(b) Quicksort is sensitive to data; the performance is $O(n \cdot \log n)$ only if most splits divide the array into two halves that are approximately equal in size. ______

(c) Quicksort requires a temporary array that is as large as the original array. ______ ✓

9. Complete each sentence with the word *always*, *sometimes*, or *never*:

(a) Selection Sort in an array of *n* elements _________________ works in $O\left(n^2\right)$ time. ✓

(c) Insertion Sort _________________ works faster than Quicksort. ✓

(d) Quicksort is _________________ slower than Mergesort.

10. Suppose the Insertion Sort algorithm for sorting an array has been modified: instead of sequential search, binary search is used to find the position for inserting the next element into the already sorted initial segment of the list. What is the big-O for the average number of comparisons (in terms of the length of the array *n*)? What is the big-O for the best and worst cases?

11. The Mergesort algorithm, as presented in Chapter 13 on page <...>, always takes $O\left(n \cdot \log n\right)$ comparisons. Add a shortcut —

```
if (...)
  return;
```

— so that the algorithm works in $O(n)$ time when the array is already sorted.

`map.put(19, "Chapter");`

The Java Collections Framework

19.1 Prologue

In software, a *framework* is a general system of components and architectural solutions that provides development tools to programmers for use with a relatively wide range of applications. A *collection* is... well, any collection of elements. The *Java collections framework* is a system of interfaces and classes in the `java.util` package that deal with different kinds of collections: lists, stacks, queues, priority queues, sets, and maps. Starting with Java 5.0, Java collections hold objects of a specified type. Java developers call them "*generic collections*" or "generics," because their code works for a collection with any specified types of elements. We would prefer to call them *type-specific collections*, because a collection holds elements of a specific type.

In this chapter we discuss the Java collections framework from the perspective of a user (that is, a software developer). We look at how each kind of collection can be used in a program, its efficiency in particular situations, and its API (Application Programming Interface), that is, its constructors and methods. We also show some coding examples. But we don't go into the implementation details of various kinds of collections. Data structures on which Java collections are based are discussed in the subsequent chapters.

The Java collections framework is vast. We will discuss only the more commonly used interfaces and classes, and only a subset of their constructors and methods. For further technical details, refer to the Java API.

Figure 19-1 shows what's in store. At the top of the collection framework lies the interface `Collection`. It defines a collection of elements in very abstract terms, with methods like `isEmpty`, `size`, `contains`, `add`, `remove`, and `iterator`. `Collection` has more specialized subinterfaces, `List` and `Set`. The `Map` interface defines maps. Strictly speaking, a map is not a collection but a *mapping* from one set of elements to another. For each interface `java.util` provides one or several classes that implement it. Different implementations are more suitable in different contexts. Some of the classes also use the `Iterator`, `Comparable`, and `Comparator` interfaces.

The `Stack` class seems to stand alone. It is a "legacy" class from the first release of Java. To be precise, `Stack` extends the class `Vector`, which is an ancient version of `ArrayList`, and `Stack` does implement `Collection`; likewise the `Queue` interface extends `Collection`. We have deliberately omitted these connections in Figure 19-1 for reasons explained later in this chapter.

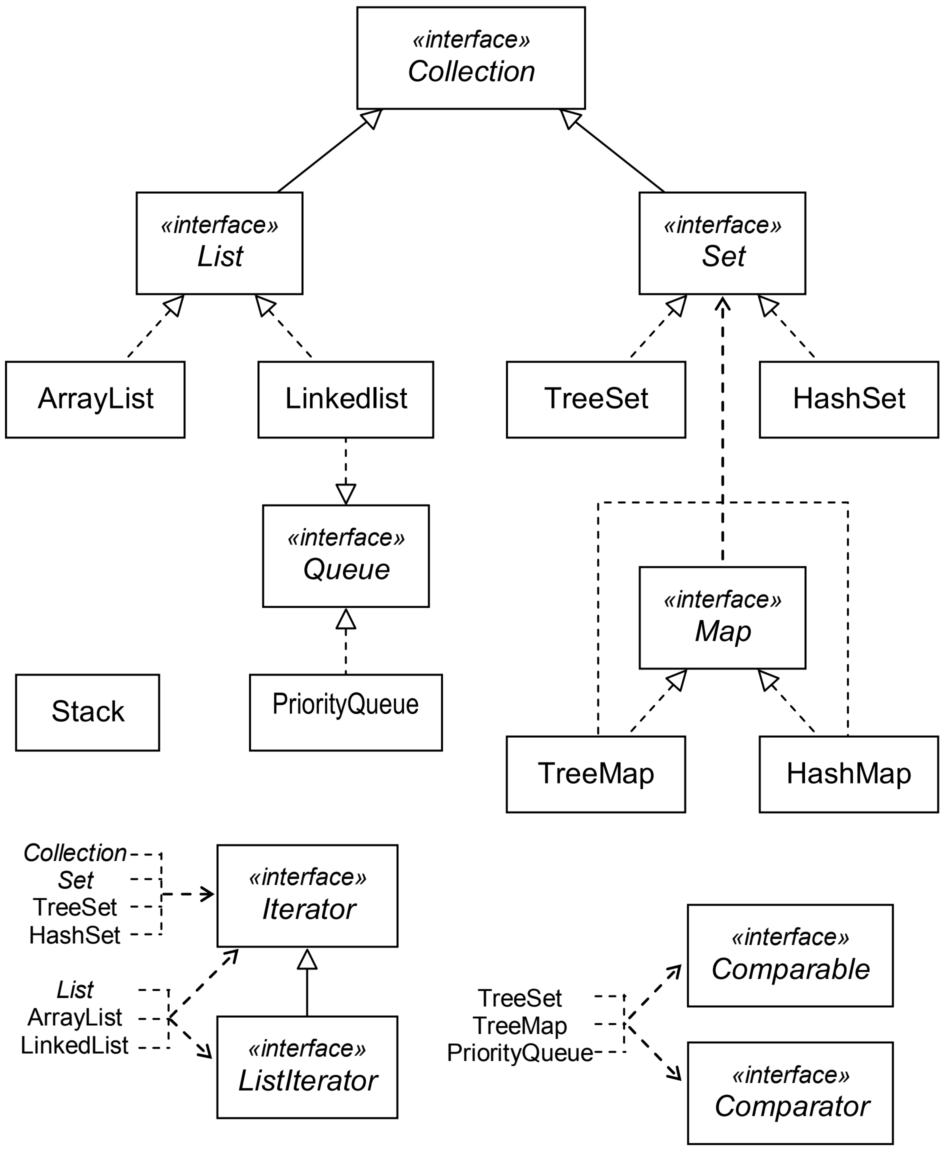


Figure 19-1. Part of the Java collections framework

❖ ❖ ❖

If your class uses Java collections, add

`import java.util.*;`

at the top of the source file.

19.2 `Collection<E>` and `Iterator<E>`

The `Collection<E>` interface represents any collection of elements of the type *`E`*. *`E`* is a type parameter — it can stand for any type of objects. Java API docs use the letter "E" for the name for this parameter, and so do we.

The commonly used `Collection<E>` methods are shown in Figure 19-2.

```
boolean isEmpty();           // Returns true if this collection is
                             //   empty; otherwise returns false.
int size();                  // Returns the number of elements in
                             //   this collection.
boolean contains(Object obj);:
                             // Returns true if this collection contains
                             //   an element equal to obj; otherwise
                             //   returns false.
boolean add(E obj);          // Adds obj to this collection.  Returns
                             //   true if successful; otherwise
                             //   returns false.
boolean remove(E obj);       // Removes the first element that is
                             //   equal to obj, if any.  Returns
                             //   true if successful; otherwise
                             //   returns false.
Iterator<E> iterator();      // Returns an iterator for this collection.
```

Figure 19-2. Some of the methods of `java.util.Collection<E>`

The `contains` and `remove` methods rely on whatever `equals` method is defined for objects of the type *`E`*. It is a programmer's responsibility to override the `Object` class's `equals` method with a reasonable version of it for the particular type *`E`*.

The `add` method adds an object of the type *`E`* to the collection.

Java collections work only with objects — they do not support primitive data types. If you want a collection to hold values of a primitive data type, you need to create a collection of objects of the corresponding wrapper class and create a wrapper object for each value before you add it to the collection.

For example, if you want to hold `double`s in an `ArrayList`, you need to create an `ArrayList<Double>`:

```
ArrayList<Double> list = new ArrayList<Double>();
list.add(new Double(5.1));
```

Actually, if you don't do the wrapping explicitly, the compiler will do it for you. You can write simply

```
list.add(5.1);
```

Starting with Java 5.0, a value of a primitive data type is converted into an object of the corresponding wrapper type automatically, when appropriate. This feature is called *autoboxing* (or *autowrapping*).

The compiler does the reverse for you, too. For example, you can write

```
double x = list.get(i);
```

instead of

```
double x = list.get(i).doubleValue();
```

This is called *autounboxing*.

Autoboxing and autounboxing save a few keystrokes but if you overuse it, it may become a source of bugs.

❖ ❖ ❖

An *iterator* is an object that helps to traverse a particular collection. `Iterator<E>` is an interface, because different collections implement different iterators with different rules, and the order of traversal can vary. Classes that implement the `Iterator` interface are implemented as private *inner* classes in various collection classes — a topic that is outside the scope of this book. The programmer obtains an iterator from a particular collection by calling that collection's `iterator` method. You need to get a new iterator for every traversal.

At this point, all of this appears to be rather abstract, and in fact it is designed to be so. When we get to specific types of collections, we will see how their iterators work. In an `ArrayList`, for example, an iterator traverses the elements in order of their indices.

The `Iterator<E>` interface specifies three methods shown in Figure 19-3.

```
boolean hasNext();            // Returns true if there are more
                              //   elements  to visit in this traversal;
                              //   otherwise returns false.
E next();                     // Returns the next element in
                              //   this traversal.
void remove();                // Removes the last visited element.
```

Figure 19-3. Methods of `java.util.Iterator<E>`

Traversal of an `ArrayList<String>`, for example, can be coded as follows:

```
ArrayList<String> words = new ArrayList<String>();
...
Iterator<String> iter = words.iterator();
while(iter.hasNext())
{
  String word = iter.next();
  < ... process word >
}
```

Since Java 5.0, iterators are used less frequently because a “for each” loop can replace an iterator (as long as the iterator’s `remove` method is not needed). The above example can be rewritten more concisely with a “for each” loop:

```
ArrayList<String> words = new ArrayList<String>();
...
for(String word : words)
{
  < ... process word >
}
```

The compiler simply converts a “for each” loop into iterator calls. (That’s why compiler error messages associated with “for each” loops sometimes mention iterators.)

Why do we need iterators? Why, for example, couldn't a `Collection`-type class itself have provided something like `startIterations`, `next`, and `hasNext` methods? The answer is, it could, as long as you used simple iterations. But when you tried nested loops, this approach would fall apart. Suppose, for example, you need to find duplicate titles in an `ArrayList<String> movies`. To do this you have to run nested loops on the same list, comparing every pair. The idea of a list serving as its own iterator would lead to an error, something like this:

```
movies.startIterations();
while (movies.hasNext())
{
  String movie1 = movies.next();
  movies.startIterations();  // Oops, a bug -- the outer loop's
                             //   position is lost
  while (movies.hasNext())
  {
    String movie2 = movies.next();
    if (movie2 != movie1 && movie2.equals(movie1))
      System.out.println("Duplicate name: " + movie2);
  }
}
```

What you need are two separate iterators, one for the outer loop and one for the inner loop:

```
Iterator<String> iter1 = movies.iterator();
while (iter1.hasNext())
{
  String movie1 = iter1.next();
  Iterator<String> iter2 = movies.iterator();

  while (iter2.hasNext())
  {
    String movie2 = iter2.next();
    if (movie2 != movie1 && movie2.equals(movie1))
      System.out.println("Duplicate name: " + movie2);
  }
}
```

This is similar to using nested "for each" loops:

```
for (String movie1 : movies)
{
  for (String movie2 : movies)
  {
    if (movie2 != movie1 && movie2.equals(movie1))
      System.out.println("Duplicate name: " + movie2);
  }
}
```

19.3 Lists and `ListIterator<E>`

A list is a collection in which the elements are arranged in a sequence and numbered (indexed). The indices start from 0.

The `List<E>` interface extends the `Collection<E>` interface, adding methods that take advantage of indices; Figure 19-4 shows some of them.

```
boolean isEmpty();
int size();
boolean contains(Object obj);        Methods inherited from
boolean add(E obj);                  Collection<E>
boolean remove(E obj);
Iterator<E> iterator();

E get(int i);                 // Returns the i-th element.
E set(int i, E obj);          // Replaces the i-th element with obj and
                              //   returns the old value.
void add(int i, E obj);       // Inserts obj to become the i-th element.
                              //   Increments the size of the list by one.
E remove(int i);              // Removes the i-th element and returns its
                              //   value.  Decrements the size of the list
                              //   by one.
int indexOf(Object obj)       // Returns the index of the first element
                              //   equal to obj; if not found, returns -1.
ListIterator<E> listIterator();
                              // Returns a ListIterator for this list.

ListIterator<E> listIterator(int i);
                              // Returns a ListIterator, which starts
                              //   iterations at the i-th element.
```

Figure 19-4. Some of the methods of the `java.util.List<E>` interface

These methods should look very familiar: we already discussed them when we introduced the `ArrayList` class in Chapter 12. The only new methods are two overloaded versions of `listIterator`. These methods return a `ListIterator<E>` object, which is an enhanced iterator, specific to `Lists` (`ListIterator<E>` is a subinterface of `Iterator<E>`). `ListIterator<E>` methods are shown in Figure 19-5.

```
boolean hasNext();
E next();                                 Methods inherited from
void remove();                            Iterator<E>

int nextIndex();              // Returns the index of the element
                              //   that would be returned by a
                              //   subsequent call to next().
boolean hasPrevious();        // Returns true if there are more
                              //   elements  to visit in reverse
                              //   direction; otherwise returns false.
E previous();                 // Returns the previous element in
                              //   the list.
int previousIndex();          // Returns the index of the element
                              //   that would be returned by a
                              //   subsequent call to previous().
void add(E obj)               // Inserts obj at the current position
                              //   of this iterator.
void set(E obj)               // Sets the element last returned
                              //   by next() or previous() to obj.
```

Figure 19-5. Methods of `java.util.ListIterator<E>`

It may be useful to envision a list iterator as a logical "cursor" positioned before the list, after the list, or between two consecutive elements of the list (Figure 19-6). `list.listIterator(0)` positions the cursor before the first element of the list (the element with the index 0). `list.listIterator(list.size())` positions the cursor after the end of the list. `list.listIterator(i)` positions the cursor between the elements with indices `i-1` and `i`. The `next` method returns the element immediately after the cursor position, and the `previous` method returns the element immediately before the cursor position.

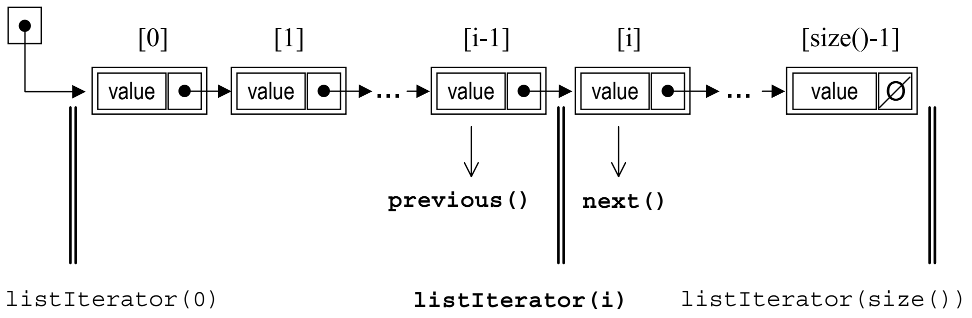


Figure 19-6. Logical "cursor" positioning for a `ListIterator`

As you can see, `ListIterator<E>` can traverse the list either forward or backward. To traverse the list from the end backward, all you have to do is obtain a `ListIterator` that starts past the end of your list, then call `previous`. For example:

```
List<String> movies = new LinkedList<String>;
...
ListIterator<String> iter = movies.listIterator(list.size());
while (iter.hasPrevious())
{
  String str = iter.previous();
  ...
}
```

The `ListIterator` interface also provides the `add` method, which inserts an element into the list (at the current "cursor" position); and the `set` method, which sets the value of the element last returned by `next` or `previous`.

The following example shows how the "remove duplicates" code from the earlier example can be rewritten more efficiently with `ListIterator`s, using "triangular" nested loops:

```
ListIterator<String> iter1 = movies.iterator();
while (iter1.hasNext())
{
  String movie1 = iter1.next();

  ListIterator<String> iter2 = movies.listIterator(iter1.nextIndex());
     // Start the inner loop at the current position
     //   of iter1

  while (iter2.hasNext())
  {
    String movie2 = iter2.next();
    if (movie1.equals(movie2))
      System.out.println("Duplicate name: " + movie1);
  }
}
```

❖ ❖ ❖

As we know, a list can be implemented as an array — that's what the Java library class `ArrayList` does. In an `ArrayList`, the list elements are stored in consecutive memory locations. This provides direct access to the *i*-th element of the array. This property, called *random access*, is important in many algorithms. For example, the Binary Search algorithm requires direct access to the element precisely in the middle between two given elements.

`ArrayList` has two drawbacks, however. First, inserting or removing an element at the beginning or somewhere in the middle of an array is inefficient — a lot of bytes may need to be moved if the array is large. Second, if the list outgrows the allocated space, we need to allocate a bigger array and copy all the values into it.

`java.util` offers another implementation of the `List<E>` interface: the `LinkedList<E>` class.

The `LinkedList<E>` class implements a list as a *doubly-linked* list.

In a doubly-linked list, each element is held within a larger structure, called a *node*. In addition to holding a reference to the element, each node holds references ("links") to the previous node and the next node. Nodes can be scattered anywhere in memory, but each node "knows" where to find the next one and the previous one. A `LinkedList` keeps references to the first node (*head*) and the last node (*tail*) of the list (Figure 19-7). We will discuss the implementation of linked lists in detail in Chapter 20.

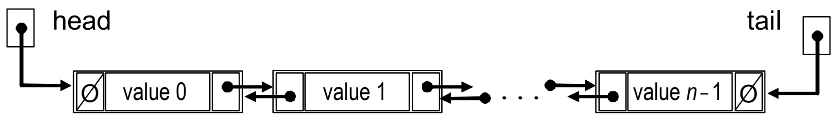


Figure 19-7. A doubly-linked list

A doubly-linked list makes it easy to insert or remove a node simply by rearranging a few links. This is a crucial property if we have a frequently updated list. Also, nodes of a linked list are allocated only when new values are added, so no memory is wasted for vacant nodes, and a list can grow as large as the total memory permits, without recopying. A drawback of a linked list, however, is that it is not a random-access collection: we have to traverse the list from the beginning or from the end to get to a particular element.

Table 19-1 compares the big-O running times for different methods in an `ArrayList` and a `LinkedList`. In an `ArrayList`, the `get(i)` and `set(i, obj)` methods work fast, $O(1)$, and they become the primary tools for handling the list. In a `LinkedList`, however, to get to the *i*-th element we have to start from the head or tail (whichever is closer to the *i*-th node) and traverse the nodes until we get to the *i*-th node. Thus, in a `LinkedList`, the `get` and `set` methods, take $O(n)$ time on average, where n is the size of the list. On the other hand, the `add(0, obj)` method takes $O(n)$ time in an `ArrayList`, because all the elements must shift to make room for the added first element. In a `LinkedList`, this method just rearranges a couple of links, so it takes constant time, $O(1)$. Appending an element at the end in both `ArrayList` and `LinkedList` takes $O(1)$ time (assuming the `ArrayList` has sufficient capacity and does not need to be reallocated).

	`ArrayList`	`LinkedList`
`get(i)` and `set(i, obj)`	$\boldsymbol{O(1)}$	$O(n)$
`add(i, obj)` and `remove(i)`	$O(n)$	$O(n)$
`add(0, obj)`	$O(n)$	$\boldsymbol{O(1)}$
`add(n-1, obj)`	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$
`contains(obj)`	$O(n)$	$O(n)$

Table 19-1. Average time big-O for some methods of `ArrayList` and `LinkedList`

One must be especially careful with `LinkedList` traversals: since `get(i)` takes $O(n)$ time, a simple traversal loop using indices —

```
List<E> list = new LinkedList<E>();
...
for (int i = 0; i < list.size(); i++)
{
  E obj = list.get(i);
  ...
}
```

— takes $O(n^2)$ time. (It starts over from the beginning of the list on each pass through the loop.) But a traversal with an iterator or a “for each” loop —

```
for (E obj : list)
{
  ...
}
```

runs in $O(n)$ time, because on each iteration the iterator simply proceeds from one node to the next.

For convenience, the `LinkedList` class provides six methods that are efficient for linked lists: `addFirst`, `addLast` (the same as `add(obj)`), `getFirst`, `getLast`, `removeFirst`, and `removeLast` (Figure 19-8). All of them take constant time, *O*(1). (`ArrayList` does not have these methods.)

```
void addFirst(E obj);         // The same as add(0, obj)
void addLast(E obj);          // The same as add(obj)
E getFirst();                 // The same as get(0)
E getLast();                  // The same as get(list.size() - 1)
E removeFirst();              // The same as remove(0)
E removeLast();               // The same as remove(list.size() - 1)
```

Figure 19-8. Additional methods in `java.util.LinkedList<E>`

In Chapter 12, we saw many examples of how `ArrayList`s are used. Each of these examples can be rewritten for a `LinkedList` by simply substituting "LinkedList" for "ArrayList" throughout. But in many cases the code will become much less efficient. As we said, a simple traversal loop

```
for (int i = 0; i < list.size(); i++)
  System.out.println(list.get(i));
```

runs in $O(n)$ time for an `ArrayList` and in $O(n^2)$ time for a `LinkedList`. You need to use a "for each" loop or an iterator for `LinkedList`s. To make your code more general, it is better to always use a "for each" loop or an iterator when traversing lists.

Figure 19-9 shows how `LinkedList`'s specific methods are used for merging two sorted `LinkedList`s into one.

```
// Merges two aphabetically sorted linked lists into one sorted list.
// Precondition: list1 and list2 are sorted alphabetically.
public static LinkedList<String> merge(
      LinkedList<String> list1, LinkedList<String> list2)
{
  LinkedList<String> result = new LinkedList<String>();

  while (!list1.isEmpty() && !list2.isEmpty())
  {
    if (list1.getFirst().compareTo(list2.getFirst()) < 0)
      result.addLast(list1.removeFirst());
    else
      result.addLast(list2.removeFirst());
  }

   // Copy the remaining elements of list1 or list2, if any
   while (!list1.isEmpty())
     result.addLast(list1.removeFirst());

  while (!list2.isEmpty())
    result.addLast(list2.removeFirst());

  return result;
}
```

Figure 19-9. Merging two sorted `LinkedLists`

19.4 Stacks

A *stack* stores data elements in such a way that the value stored last will be retrieved first. This method is sometimes called LIFO — Last-In-First-Out. A stack is controlled by two operations referred to as *push* and *pop*. Push adds an element to the top of the stack and pop removes the top element from the stack. Elements are said to be "on" the stack.

The stack mechanism is useful for temporary storage, especially for dealing with nested structures and branching processes: expressions within expressions, methods calling other methods, pictures within pictures, GUI components within GUI components, and so on. The stack mechanism helps your program untangle the nested structure and trace all of its substructures in the correct order.

A stack can be easily implemented as an `ArrayList`: push adds an element at the end of the list, and pop removes the last element. These operations take constant time, *O*(1), because no shifting of elements is necessary. A stack can be also implemented using a `LinkedList`: `push` would call `addLast`, and `pop` would call `removeLast` (or `addFirst` for `push` and `removeFirst` for `pop`).

In an efficiently implemented stack, both `push` and `pop` operations take constant time, *O*(1).

In the Java collections framework, a stack is implemented by the class `Stack`. Figure 19-10 shows some of its methods. Actually, as we noted in the prologue, `Stack` extends `Vector`, which is an older implementation of a dynamic array (a predecessor of `ArrayList`).

```
boolean isEmpty();          // Returns true if this stack is
                            //   empty; otherwise returns false.
E push (E obj)              // Pushes obj onto the stack.  (Returns obj.)

E pop()                     // Pops the top element from the stack
                            //   and retruns it.
E peek();                   // Returns the top element on the stack
                            //   without removing it from the stack.
```

Figure 19-10. Methods of `java.util.Stack<E>`

Therefore, `Stack` inherits all of the `Vector`'s methods. `Stack` also implements the `Collection` interface. So, in addition to the four stack-specific methods in Figure 19-10, it has many other methods. But using methods outside of the four stack methods and using iterators is considered "poor sportsmanship" in some situations (such as AP Computer Science exams, where you can loose points for that).

The stack methods themselves are straightforward, but algorithms that involve stacks may be harder to grasp. An example in Figure 19-11 is a simple kind: it shows how a stack is used to verify that parentheses and square brackets match in a given expression, represented as a string. It is not sufficient to count the opening and closing parentheses and brackets — they must come in the right order.

```
// Returns true if parentheses and square brackets
// match in expr; otherwise returns false
public static boolean matchParens(String expr)
{
  Stack<Character> stk = new Stack<Character>();

  for (int i = 0; i < expr.length(); i++)
  {
    char c = expr.charAt(i);

    if (c == '(' || c == '[')
    {
      stk.push(new Character(c));  // or simply stack.push(c);
    }                              //   due to autoboxing
    else if (c == ')' || c == ']')
    {
      char c0 = stk.pop().charValue(); // or simply char c0 = stk.pop();
                                       //   due to autounboxing
      if (c0 == '(' && c != ')')
        return false;
      if (c0 == '[' && c != ']')
        return false;
    }
  }

  if (!stk.isEmpty())
    return false;     // or simply return stk.isEmpty();

  return true;
}
```

Figure 19-11. An example of a method that uses a stack

19.5 Queues

In a *queue* collection, data elements are retrieved in the same order as they were stored. This access method is called FIFO — First-In-First-Out (as opposed to LIFO — Last-In-First-Out, the access method of a stack). The queue is controlled by two operations: *add* and *remove*. *add* inserts a value at the rear of the queue, and *remove* removes a value from the front of the queue.

The queue structure is usually used for processing events that are to be processed in the order of their arrival but not necessarily right away. The events are buffered in a queue while they await processing. One example is events handling in a Java program. Events generated by different objects (GUI components, timers, repaint requests, etc.) all go to the events queue. When the program is done with the previous event, it retrieves the next one from the queue and sends it to the appropriate "listener" for processing.

Queues are widely used at the system level for buffering commands or data between processes or devices. A personal computer has a keyboard queue implemented as a ring buffer (discussed in Chapter 21). When a key is pressed, its code does not go directly to the active program but is placed in the keyboard buffer until the program requests it. Printer output is also buffered: the characters are held in the output buffer until the printer is ready to receive them. An operating system maintains a queue of print jobs waiting to be sent to a printer while other programs are running.

In the Java collections framework, a queue is described by the `Queue` interface. Figure 19-12 shows a subset of its methods.

```
boolean isEmpty();            // Returns true if this queue is
                              //   empty; otherwise returns false.
boolean add (E obj)           // adds obj to the queue. Returns true
                              //   if successful, false otherwise.
E remove()                    // Removes the first element from the
                              //   queue and returns it.
E peek();                     // Returns the first element in the queue
                              //   without removing it from the queue.
```

Figure 19-12. Methods of `java.util.Queue<E>`

The Java collections framework has no separate implementations for a regular queue, because the `LinkedList` class, with its efficient `addLast` and `removeFirst` methods, serves as a fine implementation of a queue. The two methods `remove()` (with no parameters) and `peek` have been added to the `LinkedList` class to satisfy the requirements of the `Queue` interface. `remove` is simply an alias for `removeFirst`, and `peek` is an alias for `getFirst`.

In an efficient implementation of a queue, both `add` and `remove` operations take constant time, *O*(1).

In some situations (AP exams) it is considered bad style to use any `LinkedList` methods outside the `Queue` subset or to use iterators when a `LinkedList` is used as a queue, because when you work with a queue, you are supposed to only use the "pure" queue methods from Figure 19-12.

Figure 19-13 presents an example of code that involves queues.

```
// Uses Quicksort to sort the elements from q in ascending order;
// returns a new sorted queue
public static Queue<Integer> sort(Queue<Integer> q)
{
  if (q.isEmpty())  // base case
    return q;

  Queue<Integer> q1 = new LinkedList<Integer>();
  Queue<Integer> q2 = new LinkedList<Integer>();

  Integer pivot = q.remove();
  while (!q.isEmpty())
  {
    Integer num = q.remove();
    if (num.compareTo(pivot) < 0)
      q1.add(num);
    else
      q2.add(num);
  }
  q1 = sort(q1);    // recursive calls
  q2 = sort(q2);

  q1.add(pivot);

  while (!q2.isEmpty())
    q1.add(q2.remove());

  return q1;
}
```

Figure 19-13. Quicksort algorithm implemented with queues

19.6 Priority Queues

A priority queue is a different kind of queue: elements are removed not in order of their arrival but in order of their priority. It is assumed that any two elements in a priority queue can be compared with regard to their "priority." In the Java implementation, it is assumed that the elements of a priority queue are `Comparable` or a comparator is provided for them. By convention, the "smallest" element has the highest priority.

It would be easy to implement a priority queue using a sorted linked list. The `add` method would insert a new element in the proper place to keep the list sorted, and the `remove` method would remove the first element. In this implementation, `remove` would take *O*(1) time, but `add` would take *O*(*n*) time (where *n* is the number of elements in the priority queue). Alternatively, we could keep a priority queue in an unsorted linked list, add an element at the end, and find and remove the smallest element in `remove`. Then `add` would be *O*(1), but `remove` would be *O*(*n*). Either way, one of the operations could become a bottleneck.

There is a way to implement a priority queue in such a way that both `add` and `remove` take *O*(log *n*) time. Such an implementation uses a *heap*, which is a particular kind of a binary tree. (We will discuss heaps in detail in Chapter 25.) The `PriorityQueue<E>` class in the Java collections framework is such an implementation.

As you can see in Figure 19-1, `PriorityQueue<E>` implements `Queue<E>`, so it has the same methods as `Queue<E>`: `isEmpty`, `peek`, `add` and `remove` (Figure 19-12). But the meanings of the `peek` and `remove` methods are different: `peek` returns the smallest element, and `remove` removes and retunrs the smallest element (as opposed to the first element).

In the `PriorityQueue<E>` class, the `peek` operation takes *O*(1) time and both the `add` and `remove` operations take *O*(log *n*) time.

`PriorityQueue` has a no-args constructor, which assumes that the priority queue will hold `Comparable` objects. Another constructor —

```
public PriorityQueue(int initialCapacity, Compapartor<E> comparator)
```

— takes a comparator as a parameter and uses it to compare the elements.

It is possible to use a priority queue to sort a list by first inserting all the elements into the priority queue, then removing them one by one (Figure 19-14). Actually, if the priority queue is based on a heap, this sorting method is not so bad: it runs in *O*(*n* log *n*). In Chapter 25, we will discuss a more elaborate implementation of the *Heapsort* algorithm, which does not require a separate heap.

```
// Sorts arr
public static <E> void sort(E[] arr)
{
  PriorityQueue<E> q = new PriorityQueue<E>();

  for (E obj : arr)
    q.add(obj);

  for (int i = 0; i < arr.length; i++)
    arr[i] = q.remove();
}
```

Figure 19-14. A simplistic version of Heapsort

19.7 Sets

A *set* is a collection that has no duplicate elements. More formally `x1.equals(x2)` returns `false` for any two elements `x1` and `x2` in a set. The `Set<E>` interface implements `Collection<E>` but it doesn't add any new methods: it just has a different name to emphasize the no duplicates rule. Figure 19-15 lists some of the `Set` (`Collection`) methods again, for convenience.

```
boolean isEmpty();
int size();
boolean contains(E obj);
boolean add(E obj);
boolean remove(E obj);
Iterator<E> iterator();
```

Methods inherited from `Collection<E>`

Figure 19-15. Some of the methods of `java.util.Set<E>`

Since we want to avoid duplicate elements in a set, the `contains` method has to be efficient: we need to be able to search for a target element within a set quickly. The Java collections framework includes two classes that implement `Set<E>` with efficient searching: `TreeSet<E>` and `HashSet<E>`.

`TreeSet<E>` implements a set as a binary search tree (BST). (BSTs are discussed in detail in Chapter 23.) BSTs combine the idea of Binary Search with the idea of a linked structure, such as linked list, where insertion and removal of elements is relatively fast. The `TreeSet<E>` class assumes that objects of type *E* are `Comparable` or a comparator is provided for them and passed as a parameter to `TreeSet`'s constructor.

In a `TreeSet<E>`, the `contains`, `add`, and `remove` methods take *O*(log *n*) time, where *n* is the number of elements in the set.

An iterator for a `TreeSet<E>` returns its elements in increasing order.

In the second implementation, `HashSet<E>`, the elements of a set are held in a hash table. (We will discuss hash tables in detail in Chapter 24.) The basic idea is to assign all elements in the set to small subsets, called "buckets," based on the integer value returned by each element's `hashCode` method. `hashCode` works like a homeroom number for a student: it makes it is easy to take attendance and locate a particular student in the school building. The buckets are kept small, so it is easy to find a particular element in a bucket.

A `HashSet` does not make any demands on the comparability of the set elements, but it requires that the elements have a decent `hashCode` method that distributes them into buckets more or less evenly.

The `hashCode` method must agree with `equals`: if `x1.equals(x2)` is `true`, then it should be the case that `x1.hashCode() == x2.hashCode()`. In other words, equal elements should be consistently hashed into the same bucket in the table. `hashCode` is a method provided in `Object`, but classes whose objects are likely to be held in a `HashSet` usually override it.

In a `HashSet<E>`, the `contains`, `add`, and `remove` methods take constant time *O*(1).

An iterator for a `HashSet` returns the elements in no particular order.

You may be wondering: Why would we ever use a `TreeSet` when we have a faster implementation in `HashSet`? The answer is that `TreeSet` is not really much slower (recall Binary Search — it takes only 20 comparisons for 1,000,000 elements), and it offers its own advantage: its iterator returns the elements in ascending order. Also, a `HashSet` may waste some space.

An example in Figure 19-16 shows how a `TreeSet<Integer>` is used to generate a set of all primes that do not exceed a given integer. The code relies on the fact that a "for each" loop for a `TreeSet` traverses the set in ascending order. For the same reason, when we print out a `TreeSet`, the elements are shown in ascending order.

```
// Returns a set of all primes that do not exceed n
public static TreeSet<Integer> allPrimes(int n)
{
  TreeSet<Integer> primes = new TreeSet<Integer>();

  for (int k = 2; k <= n; k++)
  {
    boolean hasPrimeFactors = false;

    // Check whether one of the primes found so far
    //   is a factor of k:
    for (int p : primes)     // autounboxing
    {
      if (k % p == 0)
      {
        hasPrimeFactors = true;
        break;
      }

      if (p * p >= n)        // The values of p come from the TreeSet
      {                      //   in ascending order; there is no need
        break;               //   to check beyond sqrt(n), because
      }                      //   if p is a factor, so is n/p
    }

    if (!hasPrimeFactors)
      primes.add(k);         // autoboxing
  }

  return primes;
}
```

Figure 19-16.   An example of using a `TreeSet`

19.8 Maps

A *map* is not a collection; rather, it represents a mapping from a set of "keys" to a set of "values." In a mapping, each key is mapped onto a single value, but different keys can be mapped onto the same value (Figure 19-17). Maps are very common: account numbers can be mapped onto customers, user names can be mapped onto passwords, index entries can be mapped onto lists of page numbers, people can be mapped onto their birthdays, and so on.

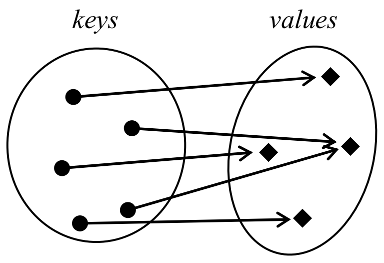


Figure 19-17. A mapping from a set of keys onto a set of values

In the Java collections framework, a map is represented by the `Map<K, V>` interface, where *K* is the type of key objects and *V* is the type of value objects. `Map`'s methods are shown in Figure 19-18.

Note that a map <u>does not</u> have an `iterator` method.

If you need to iterate over all the key-value pairs in a map, use an iterator for the set of keys. For example:

```
Map<String, Integer> map = new TreeMap<String, Integer>();
...
for (String key : map.keySet())
  System.out.println(key + " ==> " + map.get(key));
```

For a `TreeMap`, `keySet().iterator()` (or a "for each" loop for keys) will traverse the keys in ascending order.

```
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(K 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(K key);            // Removes the key-value pair from this map;
                            //   returns the value previously associated
                            //   with key.
boolean containsKey();      // 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 19-18. Some of the methods of `java.util.Map<K, V>`

❖ ❖ ❖

The Java collections framework offers two implementations of the `Map<K, V>` interface: `TreeMap<K, V>` and `HashMap<K, V>`. These implementations parallel the `TreeSet<K>` and `HashSet<K>` implementations for the set of keys: `TreeMap` is based on a binary search tree, and `HashMap` on a hash table.

The `get`, `put`, and `containsKey` methods take *O*(log *n*) time in a `TreeMap` and *O*(1) time in a `HashMap`, where *n* is the number of key-value pairs in the map.

If the truth be told, the `TreeMap` and `HashMap` implementations do not use separate sets for keys and for values. `TreeMap` uses a binary search tree of nodes; each node holds one (*key*, *value*) pair. `HashMap` 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 `TreeMap` or `HashMap`, respectively. Any changes to the set of keys will affect the map.

To avoid duplication of code, Java developers actually implemented the `TreeSet` and `HashSet` classes as special cases of `TreeMap` and `HashMap`, respectively, in which all keys are mapped onto the same object. `TreeSet` and `HashSet` have a field `map`, and their 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.

Figure 19-19 shows an example of code that uses a `TreeMap`.

```
// The names of a few people and their birthdays are
// kept in two "parallel" arrays:
String[] names = {"Tom", "Lena", "Zoe", "Aaron"};
String[] bdays = {"Jun 11", "May 5", "Jun 11", "Oct 22"};

// This representation is cumbersome and error prone. Let's put
// this data into a map, instead, where a name is the key and
// the birthday is the value:
Map<String, String> map = new TreeMap<String, String>();
      // We choose a TreeMap because we want the names
      // to be printed in alphabetical order.

for (int i = 0; i < names.length; i++)
  map.put(names[i], bdays[i]);

// Let's print out the map:
System.out.println(map);
     // The output will be:
     // {Aaron=Oct 22, Lena=May 5, Tom=Jun 11, Zoe=Jun 11}

// Now let's see if any two people in this map have the same
// birthday.  To do that, let's put all birthdays into a set,
// then compare its size with the size of the map:
HashSet<String> bdaysSet = new HashSet<String>();
     // No particular reason for choosing a HashSet
     // over a TreeSet here.

for (String name : map.keySet())
  bdaysSet.add(map.get(name));

System.out.println("At least two people have the same birthday: " +
        (bdaysSet.size() < map.size()));

// A "random" fact: the probability that in any random group
// of 23 people at least two share the same birthday is greater
// than 0.5.
```

Figure 19-19. An example of code that uses a map

❖ ❖ ❖

This concludes our brief tour of the Java collections framework. Table 19-2 summarizes our “findings.”

Collection<E>
boolean isEmpty()
int size()
boolean contains(E obj)
boolean add(E obj)
boolean remove(E obj)
Iterator<E> iterator()

List<E> extends Collection<E>	**ArrayList<E>**	**LinkedList<E>**
All Collection<E>'s methods, including:		
boolean add(E obj)	$O(1)$	$O(1)$
boolean remove(E obj)	$O(n)$	$O(n)$
boolean contains(E obj)	$O(n)$	$O(n)$
Methods specific to List<E>:		
E get(int i)	$\boldsymbol{O(1)}$	$O(n)$
E set(int i, E obj)	$\boldsymbol{O(1)}$	$O(n)$
void add(int i, E obj)	$O(n)$	$O(n)$
E remove(int i)	$O(n)$	$O(n)$
ListIterator<E> listIterator()		
ListIterator<E> listIterator(int i)		
Additional methods in LinkedList<E> only:		
E getFirst(E obj)	n/a	$\boldsymbol{O(1)}$
E getLast(E obj)		$\boldsymbol{O(1)}$
void addFirst(E obj)		$\boldsymbol{O(1)}$
void addLast(E obj)		$\boldsymbol{O(1)}$
E removeFirst(E obj)		$\boldsymbol{O(1)}$
E removeLast(E obj)		$\boldsymbol{O(1)}$

	Stack<E>
boolean isEmpty()	
E peek()	$O(1)$
E push(E obj)	$O(1)$
E pop()	$O(1)$

Table 19-2. The Java collections framework summary — continued

Queue<E>	**LinkedList<E>**	**PriorityQueue<E>** *E implements Comparable (or a comparator is provided).*
boolean isEmpty() E peek() boolean add(E obj) E remove()	 *O*(1) *O*(1) *O*(1)	 ***O*(1)** ***O*(log *n*)** ***O*(log *n*)**

Set<E>	**TreeSet<E>** *E implements Comparable (or a comparator is provided).* *Iterator traverses the set in order.*	**HashSet<E>** equals *and* hashCode *are redefined for E.* *Iterator traverses the set in no particular order.*
All Collection<E> methods, including: boolean add(E obj) boolean remove(E obj) boolean contains(E obj)	 *O*(log *n*) *O*(log *n*) *O*(log *n*)	 *O*(1) *O*(1) *O*(1)

Map<K,V>	**TreeMap<*K,V*>** *K implements Comparable (or a comparator is provided).*	**HashMap<*K,V*>** equals *and* hashCode *are redefined for K.*
boolean isEmpty() int size() V get(K key) V put(K key, V value) V remove(K key) boolean containsKey() Set<K> keySet()	 *O*(log *n*) *O*(log *n*) *O*(log *n*) *O*(log *n*) 	 *O*(1) *O*(1) *O*(1) *O*(1)

Table 19-2. The Java collections framework summary

19.9 *Case Study and Lab:* Stock Exchange

In this section we consider a larger, more realistic case study: a miniature stock exchange. A stock exchange is an organization for trading shares in publicly owned companies. In the OTC ("Over the Counter") system, stocks are traded electronically through a vast network of securities dealers connected to a computer network. There is no physical "stock exchange" location. In the past few years, thousands of investors have started trading stocks directly from their home computers through Internet-based online brokerage firms.

In this project we will program our own stock exchange and electronic brokerage, which we call *SafeTrade*. What do we mean by "safe"? Not too long ago, some unscrupulous online brokerage firms started encouraging a practice called day trading, in which traders hold a stock for a few hours or even minutes rather than months or years. As a result, quite a few people lost all their savings and got into debt. Actually, this case study would be more appropriately placed in Chapter 11: in the U.S. code of bankruptcy laws,[uscode] "Chapter 11" deals with reorganization due to bankruptcy. With our *SafeTrade* program, you stay safely offline and out of trouble and don't pay commissions to a broker.

We picked this project because it illustrates appropriate uses of many of the interfaces and classes in the Java collections framework. This project is large enough to warrant a meaningful team development effort.

❖ ❖ ❖

The stock exchange system keeps track of buy and sell orders placed by traders. It automatically executes orders when the highest "bid" price (order to buy stock at a certain price) meets the lowest "ask" price (offer to sell stock for a certain minimum price). There are also "market" orders to buy stock at the current "ask" price or to sell stock at the current "bid" price. Each stock is identified by its trading symbol. For example, Sun Microsystems is "SUNW" and Microsoft is "MSFT." In the real world, very small stock prices may include fractions of cents, but in *SafeTrade* we only go to whole cents.

SafeTrade's brokerage maintains a list of registered traders and allows them to log in and trade stocks. The program keeps track of all active buy and sell orders for each stock. A trader can request a quote for a stock; this includes the last sale price, the price and number of shares offered in the current highest bid and lowest ask, the day's high and low price for the stock, and the volume, which is the total number of shares traded during the day. In our model, a "day" is one run of the program. A stock is identified by its trading symbol, a string of one to five letters.

A trader can place buy and sell orders, specifying the price for a "limit" order or choosing a "market" order. Each order deals with only one stock. The order for a given stock holds six pieces of information: a reference to the trader who placed it, the stock symbol, the buy or sell indicator, the number of shares to be traded, the market or limit indicator, and the price for a limit order. *SafeTrade* acknowledges a placed order by sending a message back to the trader.

When a new order comes in, *SafeTrade* checks if it can be executed and, if so, executes the trade and reports it to both parties by sending messages to both traders. In *SafeTrade*, all orders are "partial" orders. This means that if an order cannot be executed for the total number of shares requested in it, the maximum possible number of shares changes hands and an order for the remaining shares remains active.

SafeTrade executes a market buy order at the price of the lowest ask, and a market sell order at the price of the highest bid. Normally, market orders can be executed immediately upon arrival. If both buy and sell orders are limit orders and the bid happens to be higher than the ask, the trade is executed at the ask price. In the unlikely event that there is only a market sell and a market buy, *SafeTrade* executes them at the last sale price. At the beginning of the day, the last sale price is supposed to be carried over from the previous day; in *SafeTrade* it is simply set by `Stock`'s constructor from a given parameter when the stock is "listed" with the stock exchange.

SafeTrade <u>does not</u> keep track of the availability of money or shares on the trader's account. If you want, you can add this functionality. For example, you can keep all transactions for a given trader in a list and have a separate field to hold his or her available "cash."

At a first glance, this appears to be a pretty large project. However, it turns out that with careful planning and an understanding of the requirements, the amount of code to be written is actually relatively small. The code is simple and splits into a number of small pieces, which can be handled either by one programmer or by a team of several programmers. We have contributed the main class and a couple of GUI classes.

Set up a project in your IDE with the `SafeTrade.java` and `SafeTrade.jar` files from `JM\Ch19\SafeTrade`. (Actually, `SafeTrade.jar` is a runnable jar file, so you can run *SafeTrade* by just double-clicking on `SafeTrade.jar`.)

One of the challenges of a project like this is testing. One of the team members should specialize in QA (Quality Assurance). While other team members are writing code, the QA person should develop a comprehensive test plan. He or she then tests the program thoroughly and works with programmers on fixing bugs.

❖ ❖ ❖

Our design process for *SafeTrade* consists of four parts. The first part is structural design, which determines which data structures will be used in the program. The second part is object-oriented design, which determines the types of objects to be defined and the classes and interfaces to be written. The third part is detailed design, which determines the fields, constructors, and methods in all the classes. The fourth part is developing a test plan. We are going to discuss the structural design first, then the OOD, and after that the detailed design and testing.

1. Structural design

Our structural design decisions are summarized in Table 19-3. We are lucky to have a chance to use many of the collections classes discussed in this chapter.

Data	***interface*** `=>` **class**
Registered traders	`Map => TreeMap<String, Trader>`
Logged-in traders	`Set => TreeSet<Trader>`
Mailbox for each trader	`Queue => LinkedList<String>`
Listed stocks	`Map => HashMap<String, Stock>`
Sell orders for each stock	`Queue => PriorityQueue<TradeOrder>` (with ascending price comparator)
Buy orders for each stock	`Queue => PriorityQueue<TradeOrder>` (with descending price comparator)

Table 19-3. Structural design decisions for *SafeTrade*

We have chosen to hold all registered traders in a `TreeMap`, keyed by the trader's screen name. A `HashMap` could potentially work faster, but the response time for a new registration is not very important, as long as it takes seconds, not hours. A

`HashMap` might waste some space, and we hope that thousands of traders will register, so we can't afford to waste any space in our database. For similar reasons, we have chosen a `TreeSet` over a `HashSet` to hold all currently logged-in traders.

A trader may log in, place a few orders, and log out. Meanwhile, *SafeTrade* may execute some of the trader's orders and send messages to the trader. But the trader may already not be there to read them. So the messages must be stored in the trader's "mailbox" until the trader logs in again and reads them. This is a perfect example of a queue. In *SafeTrade*, a mailbox for each trader is a `Queue<String>` (implemented as a `LinkedList<String>)`.

SafeTrade also needs to maintain data for each listed stock. A stock is identified by its trading symbol, so it is convenient to use a map where the stock symbol serves as the key for a stock and the whole `Stock` object serves as the value. The number of all listed stocks is limited to two or three thousand, and the list does not change very often. *SafeTrade* must be able to find a stock immediately for real-time quotes and especially to execute orders. Traders will get upset if they lose money because their order was delayed. Therefore, a good choice for maintaining listed stocks is a hash table, a `HashMap`.

Finally, *SafeTrade* must store all the buy and sell orders placed for each stock in such a way that it has quick access to the highest bid and the lowest ask. Both adding and executing orders must be fast. This is a clear case for using priority queues. We need two of them for each stock: one for sell orders and one for buy orders. For <u>sell</u> orders the order with the <u>lowest</u> ask price has the highest priority, while for <u>buy</u> orders the order with the <u>highest</u> bid price has the highest priority. Therefore, we need to write a comparator class and provide two differently configured comparator objects — one for the buy priority queue and one for the sell priority queue.

<u>2. Object-oriented design</u>

Figure 19-20 shows a class diagram for *SafeTrade*. The project involves nine classes and one interface. We have provided three classes and the interface: the application launcher class `SafeTrade`, the GUI classes `LoginWindow` and `TraderWindow`, and the `Login` interface. Your team's task is to write the remaining six classes. The following briefly describes the responsibilities of different types of objects.

Our classes

`SafeTrade`'s `main` method creates a `StockExchange` and a `Brokerage` and opens a `LoginWindow`. To make program testing easier, `main` also "lists" several stocks on the `StockExchange` —

```
    StockExchange exchange = new StockExchange();
    server.listStock("DS", "DanceStudios.com", 12.33);
    server.listStock("NSTL", "Nasty Loops Inc.", 0.25);
    server.listStock("GGGL", "Giggle.com", 28.00);
    server.listStock("MATI", "M and A Travel Inc.", 28.20);
    server.listStock("DDLC", "Dulce De Leche Corp.", 57.50);
    server.listStock("SAFT", "SafeTrade.com Inc.", 322.45);
```

— and registers and logs in a couple of traders at the `Brokerage`:

```
    Brokerage safeTrade = new Brokerage(exchange);
    safeTrade.addUser("stockman", "no");
    safeTrade.login("stockman", "no");
    safeTrade.addUser("mstrade", "no");
    safeTrade.login("mstrade", "no");
```

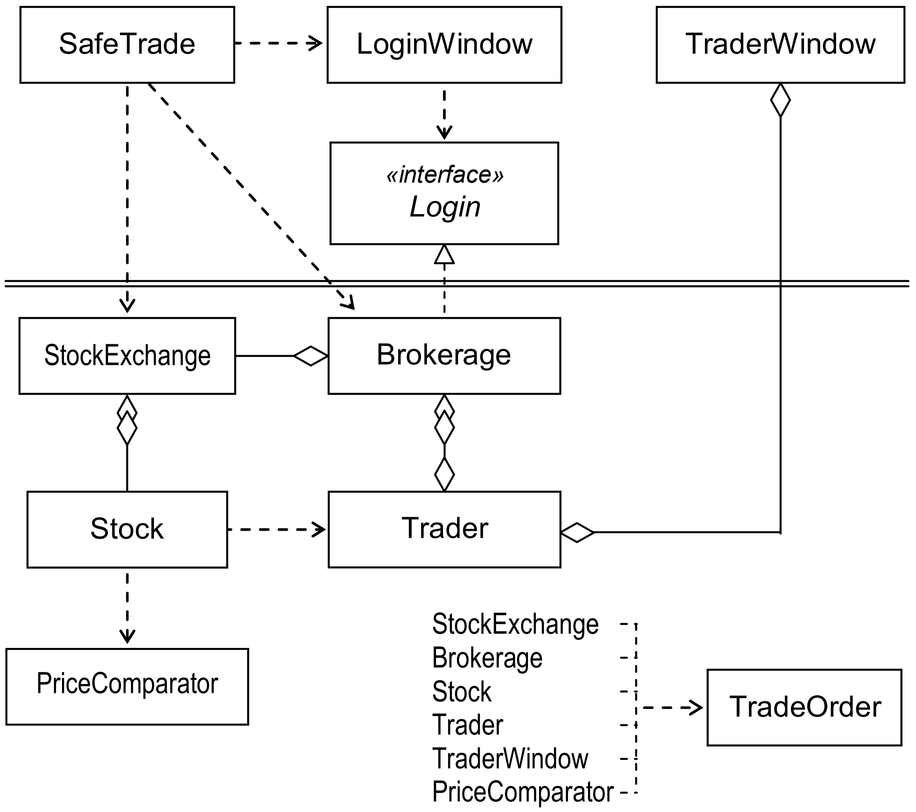
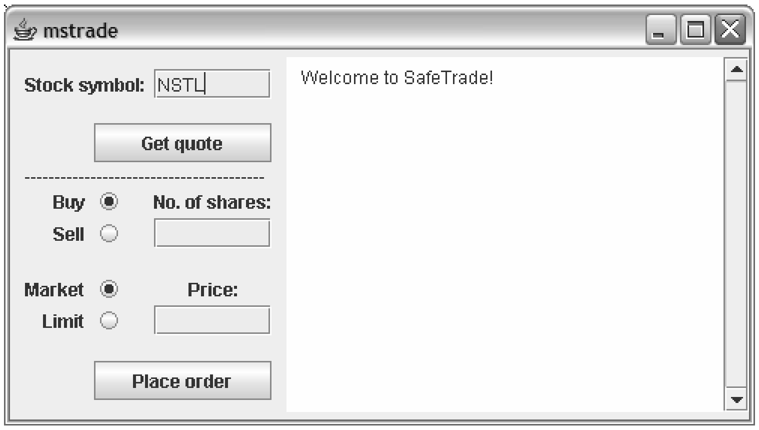


Figure 19-20. Class diagram for the *SafeTrade* project

The `LoginWindow` accepts a user name and a password from a user and can register a new trader.

The `Login` interface isolates `LoginWindow` from `Brokerage`, because logging in is a common function: we want to keep the `LoginWindow` class general and reusable in other projects.

The `TraderWindow` object is a GUI front end for a `Trader` (Figure 19-21). Each `Trader` creates one for itself. The `TraderWindow` collects the data about a quote request or a trade order and passes that data to the `Trader` by calling `Trader`'s methods.



Welcome to SafeTrade!

Figure 19-21. A trader window

Your classes:

The `StockExchange` keeps a `HashMap` of listed stocks, keyed by their symbols, and relays quote requests and trade orders to them.

The `Brokerage` keeps a `TreeMap` of registered traders and a `TreeSet` of logged-in traders. It receives quote requests and trade orders from traders and relays them to the `StockExchange`.

A `Stock` object holds the stock symbol, the company name, the lowest and highest sell prices, and the volume for the "day." It also has a priority queue for sell orders and another priority queue for buy orders for that stock.

The `Stock` class is more difficult than the other classes, because it includes an algorithm for executing orders.

A `PriceComparator` compares two `TradeOrder` objects based on their prices; an ascending or a descending comparator is passed as a parameter to the respective priority queue of orders when the priority queue is created.

A `Trader` represents one trader; it can request quotes and place orders with the brokerage. It can also receive messages and store them in its mailbox (a `Queue<String>`) and tell its `TraderWindow` to display them.

A `TradeOrder` is a "data carrier" object used by other objects to pass the data about a trade order to each other. Since all the other classes depend on `TradeOrder`, it makes sense to write it first. This is a simple class with fields and accessor methods that correspond to the data entry fields in a `TraderWindow` (Figure 19-21).

3. Detailed design

The detailed specs for the *SafeTrade* classes have been generated from the Javadoc comments in the source files and are provided in the `SafeTrade.zip` file in `JM\Ch19\SafeTrade`. Open `index.html` to see the documentation.

4. Testing

Proper testing for an application of this size is in many ways more challenging than writing the code. Entering and executing a couple of orders won't do. The QA specialist has to make a list of possible scenarios and to develop a strategy for testing them methodically. In addition, the tester must make a list of features to be tested. The most obvious are, for example: Do the "login" and "add user" screens work? Does the "Get quote" button work? When a trader logs out and then logs in again, are the messages preserved? And so on.

The `Stock` / `PriceComparator` subsystem can be tested separately. Write a simple console application for this. You will also need a *stub* class (a greatly simplified version of a class) for `Trader`, with a simple constructor, which sets the trader's name, and one method `receiveMessage(String msg)`, which prints out the trader's name and `msg`. Test the `StockExchange` class separately, using a stub class for `Stock`. Test the `Brokerage` / `Trader` subsystem separately using a stub class for `StockExchange`.

19.10 Summary

The commonly used features of the Java collections framework are summarized in Table 19-2. The collections framework provides developers with the necessary tools to implement a variety of data structures with ease.

While it is not possible to describe a large real-world project in a few pages, the *SafeTrade* case study gives you a flavor of what it might be like. Careful structural design helps a software architect to choose the data structures appropriate for a software system. The object-oriented approach helps us split a fairly large project into smaller manageable pieces that we can code and test independently. With little additional effort, we have made one of the classes (`LoginWindow`) reusable for future projects.

Although it is possible to test individual software components in a project to some extent, a comprehensive test plan is needed for testing a software application as a whole. Developing and implementing such a test plan may actually be more challenging than writing the code.

Exercises

1. Which of the following statements compile with no errors (assuming that the necessary library names are imported)?

(a) `List<String> list = new List<String>();`
(b) `List<String> list = new ArrayList<String>();`
(c) `List<String> list = new LinkedList<String>();`
(d) `ArrayList<String> list = new List<String>();`
(e) `ArrayList<String> list = new ArrayList<String>();`
(f) `LinkedList<String> list = new List<String>();`
(g) `LinkedList<String> list = new ArrayList<String>();`
(h)■ `LinkedList<Object> list = new LinkedList<String>();`

2. Write a method

```
public <E> void append(List<E> list1, List<E> list2)
```

that appends all the elements from `list2` to `list1` using indices. ✓

3. What is the primary reason for using iterators rather than indices with Java library classes that implement `java.util.List`?

4. Rewrite the method in Question 2 using an iterator or a "for each" loop.

5. Write a method

```
public LinkedList<String> mix(List<String> list1,
                                          List<String> list2)
```

that takes two lists of equal size and makes and returns a new list, alternating values from `list1` and `list2`. For example, if `list1` refers to the list `["A", "B"]` and `list2` refers to the list `["1", "2"]`, the combined list should be `["A", "1", "B", "2"]`. Your code should work in $O(n)$ time, where n is the size of `list1` and `list2`.

6.■ A list contains n+1 `Double` values a_0, ..., a_n. Write a method

```
public double sum2(List<Double> list)
```

that calculates $a_0a_1 + a_0a_2 + ... + a_0a_n + a_1a_2 + ... + a_{n-1}a_n$ — the sum of products for all pairs of elements. Do not use indices.

7. What is the output from the following code?

```
Stack<String> stack = new Stack<String>();

stack.push("A");
stack.push("B");
stack.push("C");
while (!stack.isEmpty())
{
  String s = stack.pop();
  System.out.print(s);
}
```

8. What is the output from the following code? ✓

```
Stack<String> stk = new Stack<String>();

stk.push("One");
stk.push("Two");
stk.push("Three");

while (!stk.isEmpty())
{
  String s = stk.pop();
  if (!stk.isEmpty())
  {
    s += ("-" + stk.pop());
    stk.push(s);
  }
  System.out.println(s);
}
```

9. A `Stack<Integer>` contains the values

```
(top) -1 3 7 -2 4 -6
```

What is its content after the following code is executed?

```
Stack<Integer> stackPos = new Stack<Integer>(),
      stackNeg = new Stack<Integer>();

while (!stack.isEmpty())
{
  Integer obj = stack.pop();
  if (obj.intValue() >= 0)
    stackPos.push(obj);
  else
    stackNeg.push(obj);
}

while (!stackPos.isEmpty())
  stack.push(stackPos.pop());

while (!stackNeg.isEmpty())
  stack.push(stackNeg.pop());
```

10.■ In the following code, `Point`'s `getX` and `getY` methods return `doubles`:

```
Point cursor;
Stack<Double> stk = new Stack<Double>();
...
// Save cursor position:
stk.push(cursor.getX()));
stk.push(cursor.getY()));

show(new LoginWindow());
...
// Restore cursor position:
double x = stk.pop();
double y = stk.pop();
cursor.move(x, y);
```

(a) Find and fix a bug in the code.
(b) Suggest a way to simplify the code. ✓

11.◆ A book's index contains entries and sub-entries nested to several levels. Sub-entries are indicated by deeper indentation. All the sub-entries of a given entry are preceded by the same number of spaces; that number is greater than the indentation at the previous level. For example (see `JM\Ch19\Exercises\index.dat`):

```
class
   abstract
   accessors
   constructors
     overloaded
     no-args
   modifiers
method
   private
   public
   static
stack
   for handling nested structures
   methods
     push
     pop
     peekTop
```

Write a program that reads an index file and verifies that all the entries and subentries are in alphabetical order. Skip empty lines.

12.◆ Write 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 2-D 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.dat`.

⋐ Hints: Use a stack. If there is only one way to proceed from the current position, then go there and update the total accumulated number of cookies. If there are two ways to proceed, save one of the possible two points (and its total) on the stack and proceed to the other point. If you have reached the lower right corner, update the maximum. If there is nowhere to go, examine the stack: pop a saved point, if any, and resume from there. ⋑

13. What is the output from the following code?

```
Queue<Integer> q = new LinkedList<Integer>();

for (int k = 1; k <= 3; k++)
{
  q.add(k-1);
  q.add(k+1);
}

while (!q.isEmpty())
{
  System.out.print(q.remove());
}
```

14.■ Complete a class `Student`:

```
public class Student
{
  private String name;
  private double GPA;
  ...
}
```

Then write a method

```
  public Queue<Student> cutAtGPA(Queue<Student> students,
                                               double minGPA)
```

that removes student records one by one from the `students` queue and adds those students whose GPA is not less than `minGPA` to a new "honors" queue. The method returns the "honors" queue.

15.■ A Morse code message is represented in a program as a queue of strings. Each string consists of dots and dashes. The message always ends with a special terminator string

```
  private final String terminator = "END";
```

Write and test a method

```
  public void replace(Queue<String> morseCode)
```

that replaces each question mark (represented by `"..--.."`) with a period (`".-.-.-"`), leaving all other codes unchanged. Do not use any temporary lists, queues, or other data structures.

16.■ Write a method

```
  public <E> Queue<E> copy(Queue<E> q)
```

that builds and returns a copy of `q`, leaving the original unchanged.

17.◆ A 6 by 6 game board contains arbitrarily arranged black and white squares. A path across this board may use only black squares and may move only down or to the right. Write and test a program that reads a board configuration from a file and finds and prints out all paths leading from the upper left corner (0,0) to the lower right corner (5,5). Use the `copy` method (from Question <...>) to hold partial paths and a `Stack` to hold the branching points and partial paths. A sample data file is provided in `JM\Ch19\Exercises\board.dat`.

18.■ In a `TreeSet` and a `HashSet`, which approach is used to determine whether two objects are equal?

A. `==` operator
B. `equals`
C. `compareTo` (or `compare` if a `TreeSet` was created with a comparator)
D. Both `equals` and `compareTo` (the program throws an exception if the two disagree)
E. Both `equals` and `hashCode` (a `HashSet` works properly only when the two agree)
F. Both `compareTo` and `hashCode`

Write a small program to test each hypothesis.

19. A program maintains a list of doctors on call for the current month. One doctor is on call for each day of the month. Which of the following structures is most appropriate for this task? ✓

A. A `TreeSet` of doctor names
B. A `TreeMap` keyed by doctor names
C. A `TreeMap` keyed by the day of the month
D. An `ArrayList` holding the names of doctors

20.■ (a) Define a class `Movie` that represents a movie record from the `movies.txt` file (in `JM\Ch19\Exercises`), with fields for the release year, title, director's name, and a list (`LinkedList`) of actors. Make `Movie` objects `Comparable` by title.

(b) Write a program that reads the `movies.txt` file into an array of `Movie` objects and sorts them by title using the `Arrays.sort` method.

(c) Sort the array by director's last name using a `Comparator`.

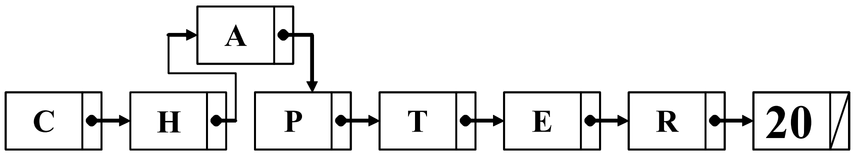
(d) Find out the total number of different actors listed in `movies.txt` by inserting them into a `TreeSet`.

(e) List all the actors from `movies.txt` in alphabetical order by last name using an iterator (or a "for each" loop) for the `TreeSet` of actors.

21.◆ Rewrite the *Index Maker* lab from Chapter 12 using sets and maps. Replace the `ArrayList` of numbers in an `IndexEntry` with a `TreeSet`. Derive `DocumentIndex` from `TreeMap<String,IndexEntry>` rather than from `ArrayList<IndexEntry>`. Modify appropriately the "for each" loop in `IndexMaker` that writes all the entries from the `DocumentIndex` into the output file.

22. What is the big-O of the running time in the following methods?

(a) `addFirst` in a `LinkedList` with *n* elements ✓
(b) `remove` in a `Queue`with *n* elements (implemented as a `LinkedList`)
(c) traversal of a `TreeSet` with *n* elements ✓
(d) `contains` in a `TreeSet` with *n* elements
(e)■ `remove` in a `PriorityQueue` with *n* elements



Lists and Iterators

20.1 Prologue

This chapter begins a series of chapters where we take a closer look at data structures. The Java collections framework provides a rich set of tools for incorporating various data structures into programs. In the previous chapter, we discussed a subset of these tools and when and how to use them. That's what most programmers need to know. However, we want to go a little beyond that and see how some of these data structures are actually implemented. We will proceed in a "do-it-yourself" manner, even though the Java library already has all the implementations. Of course we won't re-implement the library classes completely: we will just get a feel for how they work. We begin with lists.

The easiest implementation of a list is an array. As we've seen, the elements of an array are stored in consecutive locations in computer memory. We can calculate the address of each element from its index in the array. By contrast, the elements of a linked list may be scattered in various locations in memory, but each element contains a reference to the next element. The last element in the list points to nothing, so its reference to the next element is set to `null`.

Metaphorically, we can compare an array to a book: we can read its pages sequentially or we can open it to any page. A linked list is like a magazine article: at the end of the first installment it says, "continued on page 27." We read the second installment on page 27, and at the end it says, "continued on page 36," and so on, until we finally reach the ♦ symbol that marks the end of the article.

As we know from Chapter 19, the `java.util.LinkedList` class implements a list as a doubly-linked list. In this chapter we will learn the basics of implementing our own linked lists: a singly-linked list, a linked list with a tail, a doubly-linked list, and a circular list.

20.2 Singly-Linked List

In a *singly-linked list*, each element is stored in a larger structure, called a *node*. In addition to the element's value, the node contains a reference to the next node.

Let us say that the information stored in a node is represented by an object called `value` and that the reference to the next node is called `next`. We can encapsulate these fields in a class `ListNode` with one constructor, two accessors, and two modifiers (Figure 20-1).

```
// Represents a node of a singly-linked list.

public class ListNode
{
  private Object value;
  private ListNode next;

  // Constructor:
  public ListNode(Object initValue, ListNode initNext)
  {
    value = initValue;
    next = initNext;
  }

  public Object getValue() { return value; }
  public ListNode getNext() { return next; }

  public void setValue(Object theNewValue) { value = theNewValue; }
  public void setNext(ListNode theNewNext) { next = theNewNext; }
}
```

Figure 20-1. A class that represents a node in a singly-linked list (`JM\Ch20\Lists\ListNode.java`)[*]

Note two things about `ListNode`'s definition. First, "next" is a name chosen by the programmer: it is not required by Java syntax. We could have called it "link" or "nextNode" or whatever name we wanted. The name of the class, "ListNode," is also chosen by the programmer.

Second, the definition is self-referential: it refers to the `ListNode` data type inside the `ListNode` data type definition! The compiler is able to untangle this because `next` is just a reference to an object, not the object itself. A reference takes a fixed number of bytes regardless of the data type, so the compiler can calculate the total size of a `ListNode` without paying much attention to what type of reference `next` is.

[*] Adapted from The College Board's *AP Computer Science AB: Implementation Classes and Interfaces*.

We do not want to deal here with the rather complicated syntax for defining "generic" classes (which work with any specified type of objects). Therefore, the type of `value` in `ListNode` is `Object`, and we will use a cast to the appropriate type when we retrieve the value from a node.

As an example, let us consider a list of departing flights on an airport display. The flight information may be represented by an object of the type `Flight`:

```
public class Flight
{
  private int number;           // Flight number
  private String destination;   // Destination city
  ...                           // Other fields, constructors, methods
}
```

Suppose a program has to maintain a list of flights departing in the next few hours, and we have decided to implement it as a linked list. We can use the following statements to create a new node that holds information about a given flight:

```
    Flight flt = new Flight(...);
    ...
    ListNode node = new ListNode(flt, null);
```

To extract the flight info we need to cast the object returned by `getValue` back into the `Flight` type:

```
    flt = (Flight)node.getValue();
```

❖ ❖ ❖

A singly-linked list is accessed through a reference to its first node. We call it `head`. When a program creates a linked list, it usually starts with an empty list — the `head` reference is `null`:

```
   ListNode head = null;
```

To create the first node, we call `ListNode`'s constructor and store a reference to the new node in `head`:

```
   head = new ListNode(value0, null);
```

This results in a list with one node (Figure 20-2).

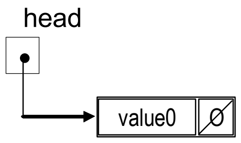


Figure 20-2. A linked list with only one node

A second node may be appended to the first:

```
ListNode node1 = new ListNode(value1, null);
head.setNext(node1);
```

Or, combining the above two statements into one:

```
head.setNext(new ListNode(value1, null));
```

This statement changes the `next` field in the `head` node from `null` to a reference to the second node. This is illustrated in Figure 20-3. Diagrams like this one help us understand how links in a linked list are reassigned in various operations.

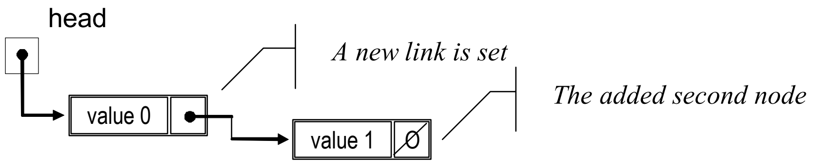


Figure 20-3. The second node is appended to the list

Figure 20-4 shows the field and constructors of the `SinglyLinkedList` class. (We are building this class just to learn how a linked list works; we will continue using the much more advanced `java.util.LinkedList` class in our projects.) `SinglyLinkedList` has one field, `ListNode head`, and two constructors. The no-args constructor creates an empty list. The constructor with one parameter, `Object[] items`, creates a list that contains all elements from the array `items` in the same order. When we build this list, we keep a reference to the last node, `tail`, append a new node to the tail node, then update `tail`.

```
// Implements a singly-linked list.
public class SinglyLinkedList
{
  private ListNode head;

  // Constructor: creates an empty list
  public SinglyLinkedList()
  {
    head = null;
  }

  // Constructor: creates a list that contains
  // all elements from the array values, in the same order
  public SinglyLinkedList(Object[] values)
  {
    ListNode tail = null;
    for (Object value : values) // for each value to insert
    {
      ListNode node = new ListNode(value, null);
      if (head == null)  // this is the first node
        head = node;
      else
        tail.setNext(node);  // append to the tail node
      tail = node;    // update tail
    }
  }
...
}
```

Figure 20-4. The constructors of the `SinglyLinkedList` class (`JM\Ch20\Lists\SinglyLinkedList.java`)

Figure 20-5 shows the resulting linked list.

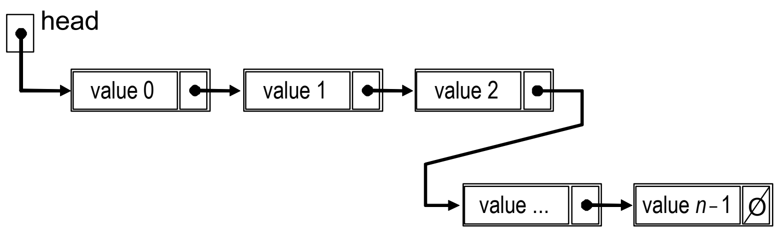


Figure 20-5. A singly-linked list

We can easily attach a new node at the head of the list. For example:

```
public void addFirst(Object value)
{
  ListNode node = new ListNode(value, null);
  node.setNext(head);
  head = node;
}
```

Or simply:

```
public void addFirst(Object value)
{
  head = new ListNode(value, head);
}
```

The above method creates a new node and sets its `next` field to the old value of `head`. It then sets `head` to refer to the newly created node (Figure 20-6).

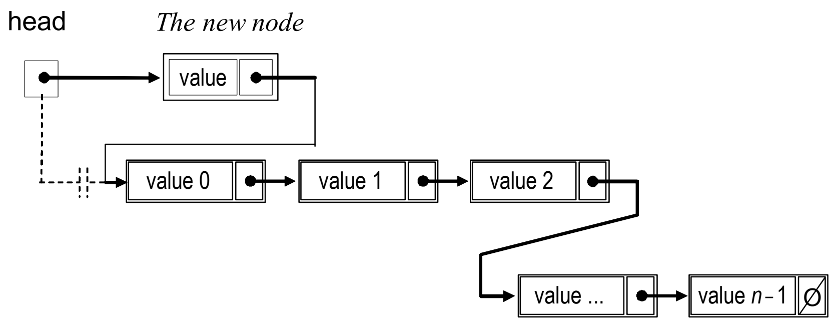


Figure 20-6. A new node inserted at the head of a linked list

But in an `addLast` method, you have to traverse the list to find the last node, then append the new node to the last one.

20.3 Traversals

It is easy to traverse a singly-linked list with a `for` loop that goes from one node to the next:

```
for (ListNode node = head; node != null; node = node.getNext())
{
  Object value = node.getValue();
  ...
}
```

❖ ❖ ❖

It is also not very hard to make iterators and "for each" loops work with our `SinglyLinkedList` class. To achieve this, we need to make our linked lists "iterable." Here is a recipe:

1. State that `SinglyLinkedList` implements `Iterable<Object>`. `Iterable<T>` is an interface in the `java.lang` package (built into Java). It specifies only one method:

   ```
   Iterator<T> iterator();
   ```

2. Add a method `iterator` to the `SinglyLinkedList` class. This method constructs an iterator for a `SinglyLinkedList`, passing the `head` of the list to the iterator's constructor as a parameter:

   ```
   public Iterator<Object> iterator()
   {
     return new SinglyLinkedListIterator(head);
   }
   ```

   The method returns the newly constructed iterator object.

3. Write the `SinglyLinkedListIterator` class, which implements `java.util.Iterator<Object>`. A simplified version of this class is shown in Figure 20-7.

```
// Implements an iterator for a SinglyLinkedList.

import java.util.Iterator;
import java.util.NoSuchElementException;

public class SinglyLinkedListIterator implements Iterator<Object>
{
  private ListNode nextNode;

  // Constructor
  public SinglyLinkedListIterator(ListNode head)
  {
    nextNode = head;
  }

  public boolean hasNext()
  {
    return nextNode != null;
  }

  public Object next()
  {
    if (nextNode == null)
      throw new NoSuchElementException();

    Object value = nextNode.getValue();
    nextNode = nextNode.getNext();
    return value;
  }

  // Not implemented.
  public void remove()
  {
    throw new UnsupportedOperationException();
  }
}
```

Figure 20-7. `JM\Ch20\Lists\SinglyLinkedListIterator.java`

In Figure 20-7 the iterator class is implemented as a regular public class. It would be stylistically more appropriate to make `SinglyLinkedListIterator` a *private inner* class in `SinglyLinkedList`, because `SinglyLinkedList` is the only class that uses this type of iterator. We have avoided inner classes in this book, because, from the OOP standpoint, the relationship between an object of an inner class and an object of its embedding outer class is not well defined. In particular, the code in an inner class has access to the private fields and methods of the embedding class. But while inner classes are philosophically confusing, their syntax is straightforward. Here, just replace `public class` with `private class` and embed the whole definition of `SinglyLinkedListIterator` inside the `SinglyLinkedList` class body. Something like this:

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class SinglyLinkedList implements Iterable<Object>
{
  < ... fields, constructors, and methods >

  private class SinglyLinkedListIterator implements Iterator<Object>
  {
    ...
  }
}
```

A more elaborate iterator implementation would have the `remove` method implemented (which means keeping track of not just the current node but also the two previous ones). It would also check that the list has not been modified outside the iterator.

20.4 *Lab:* Implementing a Singly-Linked List

Finish the `SinglyLinkedList` class (`SinglyLinkedList.java` in `JM\Ch20\Lists`). Implement all of the `java.util.List` methods shown in Figure <...> on page <...> (for elements of the `Object` type), except the two `listIterator` methods. Also implement a `toString` method: it should return a string in the same format as the library classes that implement `List`. Test your class thoroughly.

20.5 Linked List with a Tail

For a singly-linked list, adding an element at the beginning is an *O*(1) operation, while adding an element at the end is an *O*(*n*) operation (where *n* is the number of nodes), because you have to traverse the whole list to find the last node. (Incidentally, for arrays, it is exactly the opposite.) With a small modification, however, we can make appending an element at the end efficient, too. All we have to do is keep track of a reference to the last node (Figure 20-8) and update it appropriately when we add or remove nodes. Such a structure is called a *linked list with a tail*.

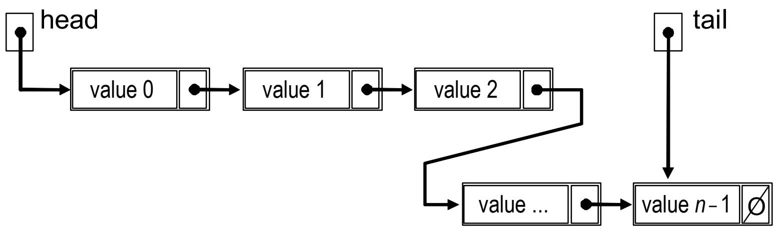


Figure 20-8. A singly-linked list with a tail

Write a class `LinkedListWithTail`. Define two fields: `head` and `tail`. `tail` should refer to the last node of the list. Provide a no-args constructor that creates an empty list, setting both `head` and `tail` to `null`. Implement the `isEmpty`, `peek`, `add`, and `remove` methods specified by the `java.util.Queue<Object>` interface. (Make sure that all these methods run in *O*(1) time.)

20.6 Doubly-Linked List and Circular List

We have already mentioned doubly-linked lists in Chapter 19 (Figure <...> on page <...>. In a doubly-linked list, each node holds references to both the next node and the previous node (Figure 20-9).

The advantage of a doubly-linked list over a singly-linked list is that we can traverse it in both directions, and we have direct access to the last node. A disadvantage is that it takes more space than a singly-linked list.

Like a linked-list with a tail, a doubly-linked list can be implemented with two fields, `head` and `tail`, that would hold references to the first and last nodes of the list, respectively. In an empty list, both `head` and `tail` are `null`.

```
// Represents a node in a doubly-linked list.
public class ListNode2
{
  private Object value;
  private ListNode2 previous;
  private ListNode2 next;

  // Constructor:
  public ListNode2(Object initValue, ListNode2 initPrevious,
                            ListNode2 initNext)
  {
    value = initValue;
    previous = initPrevious;
    next = initNext;
  }

  public Object getValue()  { return value; }
  public ListNode2 getPrevious() { return previous; }
  public ListNode2 getNext() { return next; }

  public void setValue(Object theNewValue) { value = theNewValue; }
  public void setPrevious(ListNode2 theNewPrev) { previous = theNewPrev; }
  public void setNext(ListNode2 theNewNext) { next = theNewNext; }
}
```

Figure 20-9. A class that represents a node of a doubly-linked list (`JM\Ch20\Lists\ListNode2.java`)

To add a value at the end of the list, we have to write something like this:

```
public void addLast(Object value)
{
  ListNode2 newNode = new ListNode2(value, tail, null);
  if (head == null)
  {
    head = newNode;
    tail = newNode;
  }
  else
  {
    tail.setNext(newNode);
  }
  tail = newNode;
}
```

❖ ❖ ❖

In a *circular* doubly-linked list, the "next" link in the last node refers back to the first node, and the "previous" link in the first node refers to the last node (Figure 20-10).

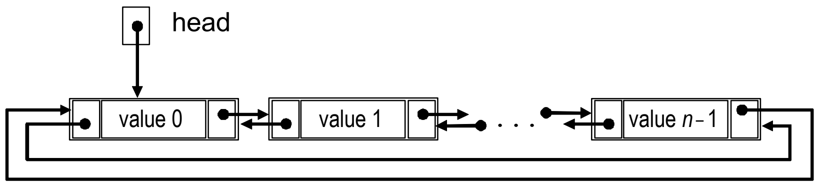


Figure 20-10. A doubly-linked circular list

In a circular list we do not need to keep a separate reference to the tail because we can access it as `head.getPrevious()`.

❖ ❖ ❖

In a doubly-linked list with separate references to the head and tail, we have to handle the case of an empty list separately. It is possible to streamline the code if we combine the `head` and `tail` references in one dummy node, which does not hold any valid value. Let's call this node `header`. Our list then becomes circular (Figure 20-11) and always has at least one node. This is how `java.util.LinkedList` is implemented.

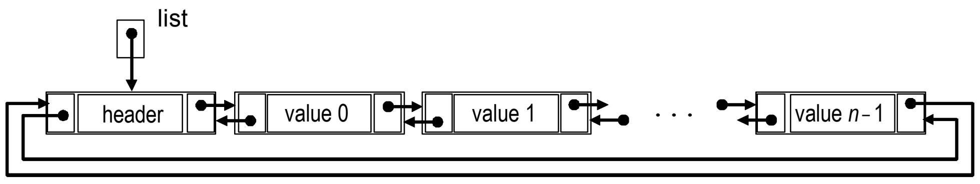


Figure 20-11. A circular doubly-linked list with a header node

In this implementation, an empty list has only one node, `header`, and its `next` and `previous` fields refer to `header` itself:

```
public DoublyLinkedList()  // Creates an empty list
{
  header = new ListNode2(null, null, null);
  header.setPrevious(header);
  header.setNext(header);
}
```

With a header node, the code for methods that insert and remove elements is a little shorter, because we don't have to handle the case of an empty list separately. For example, we can write a method to insert an element after a given node:

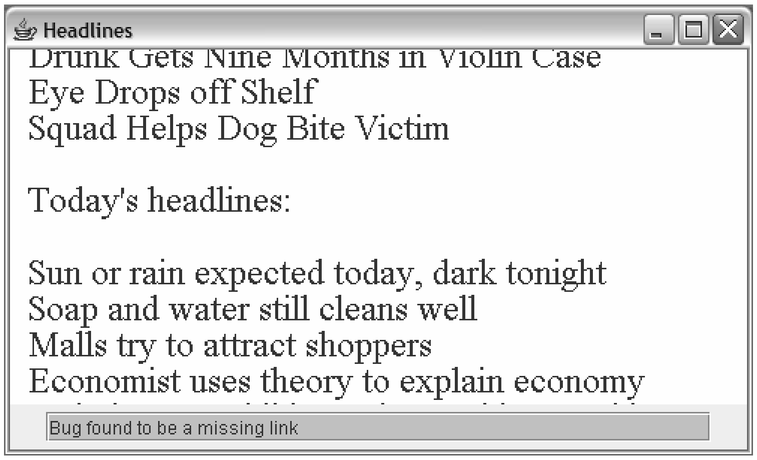
```
// Inserts value after node.
public void addAfter(ListNode2 node, Object value)
{
  ListNode2 newNode = new ListNode2(value, node, node.getNext());
  node.getNext().setPrevious(newNode);
  node.setNext(newNode);
}
```

This code also works for inserting the first or the last node. For example:

```
// Appends value to the list.
public void addLast(Object value)
{
  addAfter(header.getPrevious(), value);
}
```

20.7 *Lab:* Teletext

Figure 20-12 shows a snapshot from the *Teletext* program. The program continuously scrolls up a list of headlines. The user can add a headline, by typing it in the provided text input field. The line will be added after the blank line that follows "Today's headlines." The user can also delete a headline by entering "d." The next headline after the top one will be deleted. Run *Teletext* by clicking on the `Teletext.jar` file in `JM\Ch20\Teletext`.



```
Drunk Gets Nine Months in Violin Case
Eye Drops off Shelf
Squad Helps Dog Bite Victim

Today's headlines:

Sun or rain expected today, dark tonight
Soap and water still cleans well
Malls try to attract shoppers
Economist uses theory to explain economy
```

```
Bug found to be a missing link
```

Figure 20-12. A snapshot from the *Teletext* program

Figure 20-13 shows the class diagram for the *Teletext* program. The program keeps the headlines in a doubly-linked circular list. (Circular lists are not used very often; this is a rare occasion where we can benefit from one.) The list is implemented in the `TeletextList` class. Your task is to fill in the missing code in that class. `TeletextList.java` is located in `JM\Ch20\Teletext`.

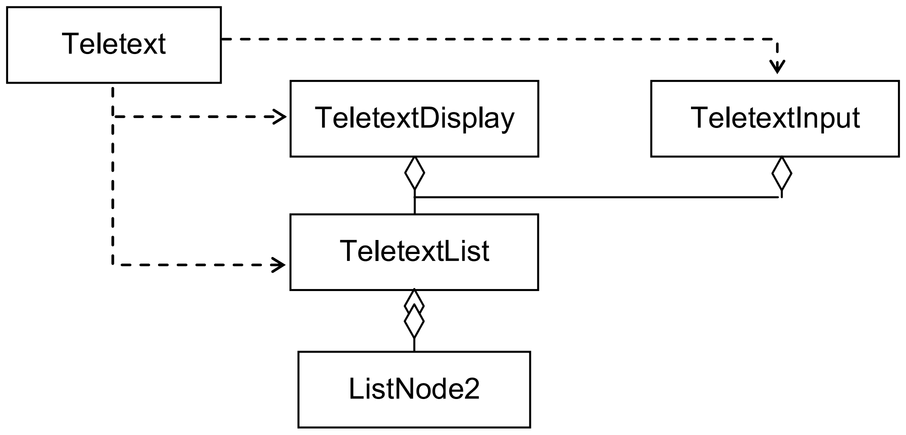


Figure 20-13. Class diagram for the *Teletext* program

20.8 Summary

In a *singly-linked list*, each element is stored in a larger structure, called a *node*. In addition to the element's value, a node contains a reference to the next node. The `next` reference in the last node is set to `null`. If `head` is a reference to the first node in the list, the statement

```
head = new ListNode(value, head);
```

appends a node holding `value` at the beginning of the list.

It is easy to traverse a linked list using a simple `for` loop:

```
for (ListNode node = head; node != null; node = node.getNext())
{
  Object value = node.getValue();
  ...
}
```

It is also not very hard to write a class that implements an "iterable" linked list for which iterators and "for each" loops work.

In a singly-linked list, inserting a node at the beginning takes $O(1)$ time, but appending a node at the end takes $O(n)$ time, where n is the number of nodes in the list. A *linked list with a tail* remedies this situation: it keeps a reference to the last node, so appending a node at the end takes $O(1)$ time. But removing the last node still takes $O(n)$ time because we need to traverse the whole list to properly update the tail reference.

In a *doubly-linked list*, each node, in addition to the list element, contains references to both the previous and the next node. A doubly-linked list can be traversed either forward or backward. Adding and removing elements at the beginning or at the end of such a list takes $O(1)$ time.

In a circular list, the last node is linked back to the first node.

The `java.util.LinkedList` class implements a list using a doubly-linked list (more precisely, a circular doubly-linked list with a header node).

Exercises

1. Fill in the blanks in the initialization of `node3`, `node2`, `node1` and `head`, so that `node1`, `node2`, and `node3` form a linked list (in that order) referred to by `head`. ✓

```
  ListNode node3 = new ListNode("Node 3",________ );
  ListNode node2 = new ListNode("Node 2",________ );
  ListNode node1 = new ListNode("Node 1",________ );
  ListNode head = ________ ;
```

2. Fill in the blanks in the following method:

```
// Returns true if the list referred to by head
// has at least two nodes; otherwise returns false
public boolean hasTwo(ListNode head)
{
  return _______________________________________ ;
}
```

3. Write a method

```
public ListNode removeFirst(ListNode head)
```

that unlinks the first node from the list and returns the head of the new list. Your method should throw a `NoSuchElementException` when the original list is empty. ✓

4. Write a method

```
public int size(ListNode head)
```

that returns the number of nodes in the list referred to by `head`:

(a) using a `for` loop
(b) using recursion.

5. `head` is the fist node of a non-empty linked list (without a tail). Write a method

```
public void add(ListNode head, Object value)
```

that appends a new node holding `value` at the end of the list. ✓

6. Fill in the blanks in the method below. This method takes the list referred to by `head`, builds a new list in which the nodes have the same information but are arranged in reverse order, and returns the head of the new list. The original list remains unchanged. Your solution must use a `for` loop (not recursion).

```
public ListNode reverseList(ListNode head)
{
  ListNode node, newNode, newHead = null;

  for ( ____________________________________________ )
  {
      ______________________________________________
      ...
  }

  return newHead;
}
```

7. Write a method

```
public ListNode concatenateStrings(ListNode head)
```

that takes the list referred to by `head`, builds a new list, and returns its head. The original list contains strings. The *k*-th node in the new list should contain the concatenation of all the strings from the original list from the first node up to and including the *k*-th node. For example, if the original list contains strings `"A"`, `"B"`, `"C"`, the new list should contain strings `"A"`, `"AB"`, `"ABC"`.

8.■ Write a method

```
public ListNode rotate(ListNode head)
```

that takes a linked list referred to by `head`, splits off the first node, and appends it at the end of the list. The method should accomplish this solely by rearranging links: do not allocate new nodes or move objects between nodes. The method should return the head of the rotated list.

9.■ A list referred to by `head` contains strings arranged alphabetically in ascending order. Write a method

```
public ListNode insertInOrder(ListNode head, String s)
```

that inserts `s` into the list, preserving the order. If `s` is already in the list, it is not inserted. The method should return the head of the updated list.

10.■ Write a method

```
public ListNode middleNode(ListNode head)
```

that returns the middle node (or one of the two middle nodes) of a linked list. Design this method using no recursion and only one loop.

11.◆ Let us say that a string matches an input pattern (another string) if the pattern is at least as long as the string and for every non-wildcard character in the pattern the string has the same character in the same position. (The wildcard character is `'?'`.) For example, both `"New York"` and `"New Jersey"` match the input pattern `"New ???????"`. Write a method

```
public ListNode moveToBack(ListNode head, String pattern)
```

that takes a list of strings referred to by `head` and moves all the strings that match `pattern` to the end of the list, preserving their order. Your method must work by rearranging links; do not allocate new nodes or use temporary arrays or lists. The method should return the head of the updated list.

12.■ Write a method

```
public void concat(LinkedListWithTail list1,
                        LinkedListWithTail list2)
```

that concatenates `list2` to `list1` in $O(1)$ time.

13.■ Implement Selection Sort for a linked list with a tail. (Assume that the list holds `Comparable` objects.) You method should run in $O(n^2)$ time, where n is the number of nodes in the list.

14.■ Write the following method:

```
// Removes the largest element from the list and returns
// the head of the modified list.
// Precondition: head refers to the first node of a
//               non-empty doubly-linked list with
//               Integer values.
public ListNode2 removeMax(ListNode2 head)
```

15.◆ Write a method

```
public void quickSort(ListNode2 fromNode, ListNode2 toNode)
```

that implements the Quicksort algorithm for a given segment of a doubly-linked list. `fromNode` refers to the first node of the segment, and `toNode` refers to the last node of the segment. (Assume that the list holds `Comparable` objects.) The links outside the segment should remain unchanged, and the segment should remain linked at the same place within the list. The method should run in $O(n \log n)$ time, where n is the number of nodes in the segment. Do not use any temporary arrays or lists.