

## Chapter 8

# Correctness and Robustness

IN PREVIOUS CHAPTERS, we have covered the fundamentals of programming. The chapters that follow will cover more advanced aspects of programming. The ideas that are presented will be a little more complex and the programs that use them a little more complicated. This chapter is a kind of turning point in which we look at the problem of getting such complex programs *right*.

Computer programs that fail are much too common. Programs are fragile. A tiny error can cause a program to misbehave or crash. Most of us are familiar with this from our own experience with computers. And we've all heard stories about software glitches that cause spacecraft to crash, telephone service to fail, and, in a few cases, people to die.

Programs don't have to be as bad as they are. It might well be impossible to guarantee that programs are problem-free, but careful programming and well-designed programming tools can help keep the problems to a minimum. This chapter will look at issues of correctness and robustness of programs. It also looks more closely at exceptions and the `try...catch` statement, and it introduces *assertions*, another of the tools that Java provides as an aid in writing correct programs.

This chapter also includes sections on two topics that are only indirectly related to correctness and robustness. Section 8.5 will introduce *threads* while Section 8.6 looks briefly at the *Analysis of Algorithms*. Both of these topics do fit into this chapter in its role as a turning point, since they are part of the foundation for more advanced programming.

### 8.1 Introduction to Correctness and Robustness

A PROGRAM is *correct* if it accomplishes the task that it was designed to perform. It is *robust* if it can handle illegal inputs and other unexpected situations in a reasonable way. For example, consider a program that is designed to read some numbers from the user and then print the same numbers in sorted order. The program is correct if it works for any set of input numbers. It is robust if it can also deal with non-numeric input by, for example, printing an error message and ignoring the bad input. A non-robust program might crash or give nonsensical output in the same circumstance.

Every program should be correct. (A sorting program that doesn't sort correctly is pretty useless.) It's not the case that every program needs to be completely robust. It depends on who will use it and how it will be used. For example, a small utility program that you write for your own use doesn't have to be particularly robust.

The question of correctness is actually more subtle than it might appear. A programmer

works from a specification of what the program is supposed to do. The programmer's work is correct if the program meets its specification. But does that mean that the program itself is correct? What if the specification is incorrect or incomplete? A correct program should be a correct implementation of a complete and correct specification. The question is whether the specification correctly expresses the intention and desires of the people for whom the program is being written. This is a question that lies largely outside the domain of computer science.

### 8.1.1 Horror Stories

Most computer users have personal experience with programs that don't work or that crash. In many cases, such problems are just annoyances, but even on a personal computer there can be more serious consequences, such as lost work or lost money. When computers are given more important tasks, the consequences of failure can be proportionately more serious.

Just a few years ago, the failure of two multi-million dollar space missions to Mars was prominent in the news. Both failures were probably due to software problems, but in both cases the problem was not with an incorrect program as such. In September 1999, the Mars Climate Orbiter burned up in the Martian atmosphere because data that was expressed in English units of measurement (such as feet and pounds) was entered into a computer program that was designed to use metric units (such as centimeters and grams). A few months later, the Mars Polar Lander probably crashed because its software turned off its landing engines too soon. The program was supposed to detect the bump when the spacecraft landed and turn off the engines then. It has been determined that deployment of the landing gear might have jarred the spacecraft enough to activate the program, causing it to turn off the engines when the spacecraft was still in the air. The unpowered spacecraft would then have fallen to the Martian surface. A more robust system would have checked the altitude before turning off the engines!

There are many equally dramatic stories of problems caused by incorrect or poorly written software. Let's look at a few incidents recounted in the book *Computer Ethics* by Tom Forester and Perry Morrison. (This book covers various ethical issues in computing. It, or something like it, is essential reading for any student of computer science.)

In 1985 and 1986, one person was killed and several were injured by excess radiation, while undergoing radiation treatments by a mis-programmed computerized radiation machine. In another case, over a ten-year period ending in 1992, almost 1,000 cancer patients received radiation dosages that were 30% less than prescribed because of a programming error.

In 1985, a computer at the Bank of New York started destroying records of on-going security transactions because of an error in a program. It took less than 24 hours to fix the program, but by that time, the bank was out \$5,000,000 in overnight interest payments on funds that it had to borrow to cover the problem.

The programming of the inertial guidance system of the F-16 fighter plane would have turned the plane upside-down when it crossed the equator, if the problem had not been discovered in simulation. The Mariner 18 space probe was lost because of an error in one line of a program. The Gemini V space capsule missed its scheduled landing target by a hundred miles, because a programmer forgot to take into account the rotation of the Earth.

In 1990, AT&T's long-distance telephone service was disrupted throughout the United States when a newly loaded computer program proved to contain a bug.

These are just a few examples. Software problems are all too common. As programmers, we need to understand why that is true and what can be done about it.

### 8.1.2 Java to the Rescue

Part of the problem, according to the inventors of Java, can be traced to programming languages themselves. Java was designed to provide some protection against certain types of errors. How can a language feature help prevent errors? Let's look at a few examples.

Early programming languages did not require variables to be declared. In such languages, when a variable name is used in a program, the variable is created automatically. You might consider this more convenient than having to declare every variable explicitly. But there is an unfortunate consequence: An inadvertent spelling error might introduce an extra variable that you had no intention of creating. This type of error was responsible, according to one famous story, for yet another lost spacecraft. In the **FORTRAN** programming language, the command “**DO 20 I = 1,5**” is the first statement of a counting loop. Now, spaces are insignificant in **FORTRAN**, so this is equivalent to “**DO20I=1,5**”. On the other hand, the command “**DO20I=1.5**”, with a period instead of a comma, is an assignment statement that assigns the value 1.5 to the variable **DO20I**. Supposedly, the inadvertent substitution of a period for a comma in a statement of this type caused a rocket to blow up on take-off. Because **FORTRAN** doesn't require variables to be declared, the compiler would be happy to accept the statement “**DO20I=1.5**.” It would just create a new variable named **DO20I**. If **FORTRAN** required variables to be declared, the compiler would have complained that the variable **DO20I** was undeclared.

While most programming languages today do require variables to be declared, there are other features in common programming languages that can cause problems. Java has eliminated some of these features. Some people complain that this makes Java less efficient and less powerful. While there is some justice in this criticism, the increase in security and robustness is probably worth the cost in most circumstances. The best defense against some types of errors is to design a programming language in which the errors are impossible. In other cases, where the error can't be completely eliminated, the language can be designed so that when the error does occur, it will automatically be detected. This will at least prevent the error from causing further harm, and it will alert the programmer that there is a bug that needs fixing. Let's look at a few cases where the designers of Java have taken these approaches.

An array is created with a certain number of locations, numbered from zero up to some specified maximum index. It is an error to try to use an array location that is outside of the specified range. In Java, any attempt to do so is detected automatically by the system. In some other languages, such as **C** and **C++**, it's up to the programmer to make sure that the index is within the legal range. Suppose that an array, **A**, has three locations, **A[0]**, **A[1]**, and **A[2]**. Then **A[3]**, **A[4]**, and so on refer to memory locations beyond the end of the array. In Java, an attempt to store data in **A[3]** will be detected. The program will be terminated (unless the error is “caught”, as discussed in Section 3.7). In **C** or **C++**, the computer will just go ahead and store the data in memory that is not part of the array. Since there is no telling what that memory location is being used for, the result will be unpredictable. The consequences could be much more serious than a terminated program. (See, for example, the discussion of buffer overflow errors later in this section.)

Pointers are a notorious source of programming errors. In Java, a variable of object type holds either a pointer to an object or the special value **null**. Any attempt to use a **null** value as if it were a pointer to an actual object will be detected by the system. In some other languages, again, it's up to the programmer to avoid such null pointer errors. In my old Macintosh computer, a **null** pointer was actually implemented as if it were a pointer to memory location zero. A program could use a null pointer to change values stored in memory near location zero. Unfortunately, the Macintosh stored important system data in those locations. Changing that

data could cause the whole system to crash, a consequence more severe than a single failed program.

Another type of pointer error occurs when a pointer value is pointing to an object of the wrong type or to a segment of memory that does not even hold a valid object at all. These types of errors are impossible in Java, which does not allow programmers to manipulate pointers directly. In other languages, it is possible to set a pointer to point, essentially, to any location in memory. If this is done incorrectly, then using the pointer can have unpredictable results.

Another type of error that cannot occur in Java is a memory leak. In Java, once there are no longer any pointers that refer to an object, that object is “garbage collected” so that the memory that it occupied can be reused. In other languages, it is the programmer’s responsibility to return unused memory to the system. If the programmer fails to do this, unused memory can build up, leaving less memory for programs and data. There is a story that many common programs for older Windows computers had so many memory leaks that the computer would run out of memory after a few days of use and would have to be restarted.

Many programs have been found to suffer from *buffer overflow errors*. Buffer overflow errors often make the news because they are responsible for many network security problems. When one computer receives data from another computer over a network, that data is stored in a buffer. The buffer is just a segment of memory that has been allocated by a program to hold data that it expects to receive. A buffer overflow occurs when more data is received than will fit in the buffer. The question is, what happens then? If the error is detected by the program or by the networking software, then the only thing that has happened is a failed network data transmission. The real problem occurs when the software does not properly detect buffer overflows. In that case, the software continues to store data in memory even after the buffer is filled, and the extra data goes into some part of memory that was not allocated by the program as part of the buffer. That memory might be in use for some other purpose. It might contain important data. It might even contain part of the program itself. This is where the real security issues come in. Suppose that a buffer overflow causes part of a program to be replaced with extra data received over a network. When the computer goes to execute the part of the program that was replaced, it’s actually executing data that was received from another computer. That data could be anything. It could be a program that crashes the computer or takes it over. A malicious programmer who finds a convenient buffer overflow error in networking software can try to exploit that error to trick other computers into executing his programs.

For software written completely in Java, buffer overflow errors are impossible. The language simply does not provide any way to store data into memory that has not been properly allocated. To do that, you would need a pointer that points to unallocated memory or you would have to refer to an array location that lies outside the range allocated for the array. As explained above, neither of these is possible in Java. (However, there could conceivably still be errors in Java’s standard classes, since some of the methods in these classes are actually written in the C programming language rather than in Java.)

It’s clear that language design can help prevent errors or detect them when they occur. Doing so involves restricting what a programmer is allowed to do. Or it requires tests, such as checking whether a pointer is `null`, that take some extra processing time. Some programmers feel that the sacrifice of power and efficiency is too high a price to pay for the extra security. In some applications, this is true. However, there are many situations where safety and security are primary considerations. Java is designed for such situations.

### 8.1.3 Problems Remain in Java

There is one area where the designers of Java chose not to detect errors automatically: numerical computations. In Java, a value of type **int** is represented as a 32-bit binary number. With 32 bits, it's possible to represent a little over four billion different values. The values of type **int** range from -2147483648 to 2147483647. What happens when the result of a computation lies outside this range? For example, what is  $2147483647 + 1$ ? And what is  $2000000000 * 2$ ? The mathematically correct result in each case cannot be represented as a value of type **int**. These are examples of *integer overflow*. In most cases, integer overflow should be considered an error. However, Java does not automatically detect such errors. For example, it will compute the value of  $2147483647 + 1$  to be the negative number, -2147483648. (What happens is that any extra bits beyond the 32-nd bit in the correct answer are discarded. Values greater than 2147483647 will “wrap around” to negative values. Mathematically speaking, the result is always “correct modulo  $2^{32}$ ”.)

For example, consider the  $3N+1$  program, which was discussed in Subsection 3.2.2. Starting from a positive integer  $N$ , the program computes a certain sequence of integers:

```
while ( N != 1 ) {
    if ( N % 2 == 0 ) // If N is even...
        N = N / 2;
    else
        N = 3 * N + 1;
    System.out.println(N);
}
```

But there is a problem here: If  $N$  is too large, then the value of  $3*N+1$  will not be mathematically correct because of integer overflow. The problem arises whenever  $3*N+1 > 2147483647$ , that is when  $N > 2147483646/3$ . For a completely correct program, we should check for this possibility **before** computing  $3*N+1$ :

```
while ( N != 1 ) {
    if ( N % 2 == 0 ) // If N is even...
        N = N / 2;
    else {
        if ( N > 2147483646/3 ) {
            System.out.println("Sorry, but the value of N has become");
            System.out.println("too large for your computer!");
            break;
        }
        N = 3 * N + 1;
    }
    System.out.println(N);
}
```

The problem here is not that the original algorithm for computing  $3N+1$  sequences was wrong. The problem is that it just can't be correctly implemented using 32-bit integers. Many programs ignore this type of problem. But integer overflow errors have been responsible for their share of serious computer failures, and a completely robust program should take the possibility of integer overflow into account. (The infamous “Y2K” bug was, in fact, just this sort of error.)

For numbers of type **double**, there are even more problems. There are still overflow errors, which occur when the result of a computation is outside the range of values that can be represented as a value of type **double**. This range extends up to about 1.7 times  $10$  to the

power 308. Numbers beyond this range do not “wrap around” to negative values. Instead, they are represented by special values that have no real numerical equivalent. The special values `Double.POSITIVE_INFINITY` and `Double.NEGATIVE_INFINITY` represent numbers outside the range of legal values. For example, `20 * 1e308` is computed to be `Double.POSITIVE_INFINITY`. Another special value of type **double**, `Double.NaN`, represents an illegal or undefined result. (“NaN” stands for “Not a Number”.) For example, the result of dividing by zero or taking the square root of a negative number is `Double.NaN`. You can test whether a number `x` is this special non-a-number value by calling the **boolean**-valued function `Double.isNaN(x)`.

For real numbers, there is the added complication that most real numbers can only be represented approximately on a computer. A real number can have an infinite number of digits after the decimal point. A value of type **double** is only accurate to about 15 digits. The real number  $1/3$ , for example, is the repeating decimal `0.333333333333...`, and there is no way to represent it exactly using a finite number of digits. Computations with real numbers generally involve a loss of accuracy. In fact, if care is not exercised, the result of a large number of such computations might be completely wrong! There is a whole field of computer science, known as *numerical analysis*, which is devoted to studying algorithms that manipulate real numbers.

So you see that not all possible errors are avoided or detected automatically in Java. Furthermore, even when an error is detected automatically, the system’s default response is to report the error and terminate the program. This is hardly robust behavior! So, a Java programmer still needs to learn techniques for avoiding and dealing with errors. These are the main topics of the rest of this chapter.

## 8.2 Writing Correct Programs

CORRECT PROGRAMS DON’T just happen. It takes planning and attention to detail to avoid errors in programs. There are some techniques that programmers can use to increase the likelihood that their programs are correct.

### 8.2.1 Provably Correct Programs

In some cases, it is possible to **prove** that a program is correct. That is, it is possible to demonstrate mathematically that the sequence of computations represented by the program will always produce the correct result. Rigorous proof is difficult enough that in practice it can only be applied to fairly small programs. Furthermore, it depends on the fact that the “correct result” has been specified correctly and completely. As I’ve already pointed out, a program that correctly meets its specification is not useful if its specification was wrong. Nevertheless, even in everyday programming, we can apply some of the ideas and techniques that are used in proving that programs are correct.

The fundamental ideas are *process* and *state*. A state consists of all the information relevant to the execution of a program at a given moment during its execution. The state includes, for example, the values of all the variables in the program, the output that has been produced, any input that is waiting to be read, and a record of the position in the program where the computer is working. A process is the sequence of states that the computer goes through as it executes the program. From this point of view, the meaning of a statement in a program can be expressed in terms of the effect that the execution of that statement has on the computer’s state. As a simple example, the meaning of the assignment statement “`x = 7;`” is that after this statement is executed, the value of the variable `x` will be 7. We can be absolutely

sure of this fact, so it is something upon which we can build part of a mathematical proof.

In fact, it is often possible to look at a program and deduce that some fact must be true at a given point during the execution of a program. For example, consider the `do` loop:

```
do {
    TextIO.put("Enter a positive integer: ");
    N = TextIO.getlnInt();
} while (N <= 0);
```

After this loop ends, we can be absolutely sure that the value of the variable `N` is greater than zero. The loop cannot end until this condition is satisfied. This fact is part of the meaning of the `while` loop. More generally, if a `while` loop uses the test “`while (<condition>)`”, then after the loop ends, we can be sure that the `<condition>` is false. We can then use this fact to draw further deductions about what happens as the execution of the program continues. (With a loop, by the way, we also have to worry about the question of whether the loop will ever end. This is something that has to be verified separately.)

A fact that can be proven to be true after a given program segment has been executed is called a **postcondition** of that program segment. Postconditions are known facts upon which we can build further deductions about the behavior of the program. A postcondition of a program as a whole is simply a fact that can be proven to be true after the program has finished executing. A program can be proven to be correct by showing that the postconditions of the program meet the program’s specification.

Consider the following program segment, where all the variables are of type **double**:

```
disc = B*B - 4*A*C;
x = (-B + Math.sqrt(disc)) / (2*A);
```

The quadratic formula (from high-school mathematics) assures us that the value assigned to `x` is a solution of the equation  $Ax^2 + Bx + C = 0$ , provided that the value of `disc` is greater than or equal to zero and the value of `A` is not zero. **If** we can assume or guarantee that  $B^2 - 4AC \geq 0$  and that  $A \neq 0$ , then the fact that `x` is a solution of the equation becomes a postcondition of the program segment. We say that the condition,  $B^2 - 4AC \geq 0$  is a **precondition** of the program segment. The condition that  $A \neq 0$  is another precondition. A precondition is defined to be condition that must be true at a given point in the execution of a program in order for the program to continue correctly. A precondition is something that you want to be true. It’s something that you have to check or force to be true, if you want your program to be correct.

We’ve encountered preconditions and postconditions once before, in Subsection 4.6.1. That section introduced preconditions and postconditions as a way of specifying the contract of a subroutine. As the terms are being used here, a precondition of a subroutine is just a precondition of the code that makes up the definition of the subroutine, and the postcondition of a subroutine is a postcondition of the same code. In this section, we have generalized these terms to make them more useful in talking about program correctness.

Let’s see how this works by considering a longer program segment:

```
do {
    TextIO.putln("Enter A, B, and C.  B*B-4*A*C must be >= 0.");
    TextIO.put("A = ");
    A = TextIO.getlnDouble();
    TextIO.put("B = ");
    B = TextIO.getlnDouble();
    TextIO.put("C = ");
```

```

    C = TextIO.getlnDouble();
    if (A == 0 || B*B - 4*A*C < 0)
        TextIO.putln("Your input is illegal. Try again.");
} while (A == 0 || B*B - 4*A*C < 0);

disc = B*B - 4*A*C;
x = (-B + Math.sqrt(disc)) / (2*A);

```

After the loop ends, we can be sure that  $B*B - 4*A*C \geq 0$  and that  $A \neq 0$ . The preconditions for the last two lines are fulfilled, so the postcondition that  $x$  is a solution of the equation  $A*x^2 + B*x + C = 0$  is also valid. This program segment correctly and provably computes a solution to the equation. (Actually, because of problems with representing numbers on computers, this is not 100% true. The **algorithm** is correct, but the **program** is not a perfect implementation of the algorithm. See the discussion in Subsection 8.1.3.)

Here is another variation, in which the precondition is checked by an `if` statement. In the first part of the `if` statement, where a solution is computed and printed, we know that the preconditions are fulfilled. In the other parts, we know that one of the preconditions fails to hold. In any case, the program is correct.

```

TextIO.putln("Enter your values for A, B, and C.");
TextIO.put("A = ");
A = TextIO.getlnDouble();
TextIO.put("B = ");
B = TextIO.getlnDouble();
TextIO.put("C = ");
C = TextIO.getlnDouble();

if (A != 0 && B*B - 4*A*C >= 0) {
    disc = B*B - 4*A*C;
    x = (-B + Math.sqrt(disc)) / (2*A);
    TextIO.putln("A solution of  $A*X*X + B*X + C = 0$  is " + x);
}
else if (A == 0) {
    TextIO.putln("The value of A cannot be zero.");
}
else {
    TextIO.putln("Since  $B*B - 4*A*C$  is less than zero, the");
    TextIO.putln("equation  $A*X*X + B*X + C = 0$  has no solution.");
}

```

Whenever you write a program, it's a good idea to watch out for preconditions and think about how your program handles them. Often, a precondition can offer a clue about how to write the program.

For example, every array reference, such as `A[i]`, has a precondition: The index must be within the range of legal indices for the array. For `A[i]`, the precondition is that  $0 \leq i < A.length$ . The computer will check this condition when it evaluates `A[i]`, and if the condition is not satisfied, the program will be terminated. In order to avoid this, you need to make sure that the index has a legal value. (There is actually another precondition, namely that `A` is not `null`, but let's leave that aside for the moment.) Consider the following code, which searches for the number `x` in the array `A` and sets the value of `i` to be the index of the array element that contains `x`:



```

i = 0;
while (A[i] != x) {
    i++;
}

```

As this program segment stands, it has a precondition, namely that `x` is actually in the array. If this precondition is satisfied, then the loop will end when `A[i] == x`. That is, the value of `i` when the loop ends will be the position of `x` in the array. However, if `x` is not in the array, then the value of `i` will just keep increasing until it is equal to `A.length`. At that time, the reference to `A[i]` is illegal and the program will be terminated. To avoid this, we can add a test to make sure that the precondition for referring to `A[i]` is satisfied:

```

i = 0;
while (i < A.length && A[i] != x) {
    i++;
}

```

Now, the loop will definitely end. After it ends, `i` will satisfy **either** `i == A.length` or `A[i] == x`. An `if` statement can be used after the loop to test which of these conditions caused the loop to end:

```

i = 0;
while (i < A.length && A[i] != x) {
    i++;
}

if (i == A.length)
    System.out.println("x is not in the array");
else
    System.out.println("x is in position " + i);

```

### 8.2.2 Robust Handling of Input

One place where correctness and robustness are important—and especially difficult—is in the processing of input data, whether that data is typed in by the user, read from a file, or received over a network. Files and networking will be covered in Chapter 11, which will make essential use of material that will be covered in the next two sections of this chapter. For now, let's look at an example of processing user input.

Examples in this textbook use my *TextIO* class for reading input from the user. This class has built-in error handling. For example, the function `TextIO.getDouble()` is guaranteed to return a legal value of type **double**. If the user types an illegal value, then *TextIO* will ask the user to re-enter their response; your program never sees the illegal value. However, this approach can be clumsy and unsatisfactory, especially when the user is entering complex data. In the following example, I'll do my own error-checking.

Sometimes, it's useful to be able to look ahead at what's coming up in the input without actually reading it. For example, a program might need to know whether the next item in the input is a number or a word. For this purpose, the *TextIO* class includes the function `TextIO.peek()`. This function returns a **char** which is the next character in the user's input, but it does not actually read that character. If the next thing in the input is an end-of-line, then `TextIO.peek()` returns the new-line character, `'\n'`.

Often, what we really need to know is the next **non-blank** character in the user's input. Before we can test this, we need to skip past any spaces (and tabs). Here is a function that does

this. It uses `TextIO.peek()` to look ahead, and it reads characters until the next character in the input is either an end-of-line or some non-blank character. (The function `TextIO.getAnyChar()` reads and returns the next character in the user's input, even if that character is a space. By contrast, the more common `TextIO.getChar()` would skip any blanks and then read and return the next non-blank character. We can't use `TextIO.getChar()` here since the object is to skip the blanks **without** reading the next non-blank character.)

```
/**
 * Reads past any blanks and tabs in the input.
 * Postcondition: The next character in the input is an
 *                end-of-line or a non-blank character.
 */
static void skipBlanks() {
    char ch;
    ch = TextIO.peek();
    while (ch == ' ' || ch == '\t') {
        // Next character is a space or tab; read it
        // and look at the character that follows it.
        ch = TextIO.getAnyChar();
        ch = TextIO.peek();
    }
} // end skipBlanks()
```

(In fact, this operation is so common that it is built into the most recent version of `TextIO`. The method `TextIO.skipBlanks()` does essentially the same thing as the `skipBlanks()` method presented here.)

An example in Subsection 3.5.3 allowed the user to enter length measurements such as “3 miles” or “1 foot”. It would then convert the measurement into inches, feet, yards, and miles. But people commonly use combined measurements such as “3 feet 7 inches”. Let's improve the program so that it allows inputs of this form.

More specifically, the user will input lines containing one or more measurements such as “1 foot” or “3 miles 20 yards 2 feet”. The legal units of measure are inch, foot, yard, and mile. The program will also recognize plurals (inches, feet, yards, miles) and abbreviations (in, ft, yd, mi). Let's write a subroutine that will read one line of input of this form and compute the equivalent number of inches. The main program uses the number of inches to compute the equivalent number of feet, yards, and miles. If there is any error in the input, the subroutine will print an error message and return the value -1. The subroutine assumes that the input line is not empty. The main program tests for this before calling the subroutine and uses an empty line as a signal for ending the program.

Ignoring the possibility of illegal inputs, a pseudocode algorithm for the subroutine is

```
inches = 0    // This will be the total number of inches
while there is more input on the line:
    read the numerical measurement
    read the units of measure
    add the measurement to inches
return inches
```

We can test whether there is more input on the line by checking whether the next non-blank character is the end-of-line character. But this test has a precondition: Before we can test the next non-blank character, we have to skip over any blanks. So, the algorithm becomes

```

inches = 0
skipBlanks()
while TextIO.peek() is not '\n':
    read the numerical measurement
    read the unit of measure
    add the measurement to inches
    skipBlanks()
return inches

```

Note the call to `skipBlanks()` at the end of the `while` loop. This subroutine must be executed before the computer returns to the test at the beginning of the loop. More generally, if the test in a `while` loop has a precondition, then you have to make sure that this precondition holds at the **end** of the `while` loop, before the computer jumps back to re-evaluate the test.

What about error checking? Before reading the numerical measurement, we have to make sure that there is really a number there to read. Before reading the unit of measure, we have to test that there is something there to read. (The number might have been the last thing on the line. An input such as “3”, without a unit of measure, is illegal.) Also, we have to check that the unit of measure is one of the valid units: inches, feet, yards, or miles. Here is an algorithm that includes error-checking:

```

inches = 0
skipBlanks()

while TextIO.peek() is not '\n':

    if the next character is not a digit:
        report an error and return -1
    Let measurement = TextIO.getDouble();

    skipBlanks()    // Precondition for the next test!!
    if the next character is end-of-line:
        report an error and return -1
    Let units = TextIO.getWord()

    if the units are inches:
        add measurement to inches
    else if the units are feet:
        add 12*measurement to inches
    else if the units are yards:
        add 36*measurement to inches
    else if the units are miles:
        add 12*5280*measurement to inches
    else
        report an error and return -1

    skipBlanks()

return inches

```

As you can see, error-testing adds significantly to the complexity of the algorithm. Yet this is still a fairly simple example, and it doesn’t even handle all the possible errors. For example, if the user enters a numerical measurement such as `1e400` that is outside the legal range of values of type **double**, then the program will fall back on the default error-handling in *TextIO*. Something even more interesting happens if the measurement is “1e308 miles”. The number `1e308` is legal, but the corresponding number of inches is outside the legal range of

values for type **double**. As mentioned in the previous section, the computer will get the value `Double.POSITIVE_INFINITY` when it does the computation.

Here is the subroutine written out in Java:

```
/**
 * Reads the user's input measurement from one line of input.
 * Precondition:  The input line is not empty.
 * Postcondition: If the user's input is legal, the measurement
 *               is converted to inches and returned.  If the
 *               input is not legal, the value -1 is returned.
 *               The end-of-line is NOT read by this routine.
 */
static double readMeasurement() {

    double inches; // Total number of inches in user's measurement.

    double measurement; // One measurement,
                        // such as the 12 in "12 miles"
    String units;       // The units specified for the measurement,
                        // such as "miles"

    char ch; // Used to peek at next character in the user's input.

    inches = 0; // No inches have yet been read.

    skipBlanks();
    ch = TextIO.peek();

    /* As long as there is more input on the line, read a measurement and
       add the equivalent number of inches to the variable, inches.  If an
       error is detected during the loop, end the subroutine immediately
       by returning -1. */

    while (ch != '\n') {

        /* Get the next measurement and the units.  Before reading
           anything, make sure that a legal value is there to read. */

        if ( ! Character.isDigit(ch) ) {
            TextIO.putln(
                "Error:  Expected to find a number, but found " + ch);
            return -1;
        }
        measurement = TextIO.getDouble();

        skipBlanks();
        if (TextIO.peek() == '\n') {
            TextIO.putln(
                "Error:  Missing unit of measure at end of line.");
            return -1;
        }
        units = TextIO.getWord();
        units = units.toLowerCase();

        /* Convert the measurement to inches and add it to the total. */

        if (units.equals("inch")
            || units.equals("inches") || units.equals("in")) {
            inches += measurement;
        }
    }
}
```

```

    }
    else if (units.equals("foot")
        || units.equals("feet") || units.equals("ft")) {
        inches += measurement * 12;
    }
    else if (units.equals("yard")
        || units.equals("yards") || units.equals("yd")) {
        inches += measurement * 36;
    }
    else if (units.equals("mile")
        || units.equals("miles") || units.equals("mi")) {
        inches += measurement * 12 * 5280;
    }
    else {
        TextIO.putln("Error: \"" + units
            + "\" is not a legal unit of measure.");
        return -1;
    }

    /* Look ahead to see whether the next thing on the line is
       the end-of-line. */

    skipBlanks();
    ch = TextIO.peek();

} // end while

return inches;

} // end readMeasurement()

```

The source code for the complete program can be found in the file *LengthConverter2.java*.

## 8.3 Exceptions and try..catch

GETTING A PROGRAM TO WORK under ideal circumstances is usually a lot easier than making the program *robust*. A robust program can survive unusual or “exceptional” circumstances without crashing. One approach to writing robust programs is to anticipate the problems that might arise and to include tests in the program for each possible problem. For example, a program will crash if it tries to use an array element `A[i]`, when `i` is not within the declared range of indices for the array `A`. A robust program must anticipate the possibility of a bad index and guard against it. One way to do this is to write the program in a way that ensures that the index is in the legal range. Another way is to test whether the index value is legal before using it in the array. This could be done with an `if` statement:

```

if (i < 0 || i >= A.length) {
    ... // Do something to handle the out-of-range index, i
}
else {
    ... // Process the array element, A[i]
}

```

There are some problems with this approach. It is difficult and sometimes impossible to anticipate all the possible things that might go wrong. It's not always clear what to do when an error is detected. Furthermore, trying to anticipate all the possible problems can turn what would otherwise be a straightforward program into a messy tangle of `if` statements.

### 8.3.1 Exceptions and Exception Classes

We have already seen that Java (like its cousin, C++) provides a neater, more structured alternative method for dealing with errors that can occur while a program is running. The method is referred to as *exception handling*. The word “exception” is meant to be more general than “error.” It includes any circumstance that arises as the program is executed which is meant to be treated as an exception to the normal flow of control of the program. An exception might be an error, or it might just be a special case that you would rather not have clutter up your elegant algorithm.

When an exception occurs during the execution of a program, we say that the exception is *thrown*. When this happens, the normal flow of the program is thrown off-track, and the program is in danger of crashing. However, the crash can be avoided if the exception is *caught* and handled in some way. An exception can be thrown in one part of a program and caught in a different part. An exception that is not caught will generally cause the program to crash. (More exactly, the thread that throws the exception will crash. In a multithreaded program, it is possible for other threads to continue even after one crashes. We will cover threads in Section 8.5. In particular, GUI programs are multithreaded, and parts of the program might continue to function even while other parts are non-functional because of exceptions.)

By the way, since Java programs are executed by a Java interpreter, having a program crash simply means that it terminates abnormally and prematurely. It doesn't mean that the Java interpreter will crash. In effect, the interpreter catches any exceptions that are not caught by the program. The interpreter responds by terminating the program. In many other programming languages, a crashed program will sometimes crash the entire system and freeze the computer until it is restarted. With Java, such system crashes should be impossible—which means that when they happen, you have the satisfaction of blaming the system rather than your own program.

Exceptions were introduced in Section 3.7, along with the `try...catch` statement, which is used to catch and handle exceptions. However, that section did not cover the complete syntax of `try...catch` or the full complexity of exceptions. In this section, we cover these topics in full detail.

\* \* \*

When an exception occurs, the thing that is actually “thrown” is an object. This object can carry information (in its instance variables) from the point where the exception occurs to the point where it is caught and handled. This information always includes the *subroutine call stack*, which is a list of the subroutines that were being executed when the exception was thrown. (Since one subroutine can call another, several subroutines can be active at the same time.) Typically, an exception object also includes an error message describing what happened to cause the exception, and it can contain other data as well. All exception objects must belong to a subclass of the standard class `java.lang.Throwable`. In general, each different type of exception is represented by its own subclass of *Throwable*, and these subclasses are arranged in a fairly complex class hierarchy that shows the relationship among various types of exception. *Throwable* has two direct subclasses, *Error* and *Exception*. These two subclasses in turn have

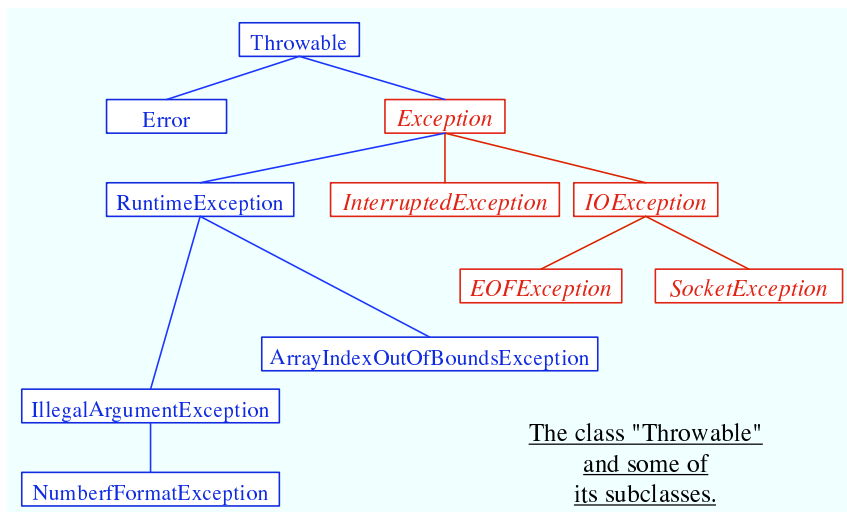
many other predefined subclasses. In addition, a programmer can create new exception classes to represent new types of exception.

Most of the subclasses of the class *Error* represent serious errors within the Java virtual machine that should ordinarily cause program termination because there is no reasonable way to handle them. In general, you should not try to catch and handle such errors. An example is a *ClassFormatError*, which occurs when the Java virtual machine finds some kind of illegal data in a file that is supposed to contain a compiled Java class. If that class was being loaded as part of the program, then there is really no way for the program to proceed.

On the other hand, subclasses of the class *Exception* represent exceptions that are meant to be caught. In many cases, these are exceptions that might naturally be called “errors,” but they are errors in the program or in input data that a programmer can anticipate and possibly respond to in some reasonable way. (However, you should avoid the temptation of saying, “Well, I’ll just put a thing here to catch all the errors that might occur, so my program won’t crash.” If you don’t have a reasonable way to respond to the error, it’s best just to let the program crash, because trying to go on will probably only lead to worse things down the road—in the worst case, a program that gives an incorrect answer without giving you any indication that the answer might be wrong!)

The class *Exception* has its own subclass, *RuntimeException*. This class groups together many common exceptions, including all those that have been covered in previous sections. For example, *IllegalArgumentException* and *NullPointerException* are subclasses of *RuntimeException*. A *RuntimeException* generally indicates a bug in the program, which the programmer should fix. *RuntimeExceptions* and *Errors* share the property that a program can simply ignore the possibility that they might occur. (“Ignoring” here means that you are content to let your program crash if the exception occurs.) For example, a program does this every time it uses an array reference like `A[i]` without making arrangements to catch a possible *ArrayIndexOutOfBoundsException*. For all other exception classes besides *Error*, *RuntimeException*, and their subclasses, exception-handling is “mandatory” in a sense that I’ll discuss below.

The following diagram is a class hierarchy showing the class *Throwable* and just a few of its subclasses. Classes that require mandatory exception-handling are shown in *italic*:



The class *Throwable* includes several instance methods that can be used with any exception object. If `e` is of type *Throwable* (or one of its subclasses), then `e.getMessage()` is a function

that returns a *String* that describes the exception. The function `e.toString()`, which is used by the system whenever it needs a string representation of the object, returns a *String* that contains the name of the class to which the exception belongs as well as the same string that would be returned by `e.getMessage()`. And `e.printStackTrace()` writes a stack trace to standard output that tells which subroutines were active when the exception occurred. A stack trace can be very useful when you are trying to determine the cause of the problem. (Note that if an exception is **not** caught by the program, then the system automatically prints the stack trace to standard output.)

### 8.3.2 The try Statement

To catch exceptions in a Java program, you need a **try** statement. We have been using such statements since Section 3.7, but the full syntax of the **try** statement is more complicated than what was presented there. The **try** statements that we have used so far had a syntax similar to the following example:

```
try {
    double determinant = M[0][0]*M[1][1] - M[0][1]*M[1][0];
    System.out.println("The determinant of M is " + determinant);
}
catch ( ArrayIndexOutOfBoundsException e ) {
    System.out.println("M is the wrong size to have a determinant.");
    e.printStackTrace();
}
```

Here, the computer tries to execute the block of statements following the word “**try**”. If no exception occurs during the execution of this block, then the “**catch**” part of the statement is simply ignored. However, if an exception of type *ArrayIndexOutOfBoundsException* occurs, then the computer jumps immediately to the **catch** clause of the **try** statement. This block of statements is said to be an *exception handler* for *ArrayIndexOutOfBoundsException*. By handling the exception in this way, you prevent it from crashing the program. Before the body of the **catch** clause is executed, the object that represents the exception is assigned to the variable `e`, which is used in this example to print a stack trace.

However, the full syntax of the **try** statement allows more than one **catch** clause. This makes it possible to catch several different types of exception with one **try** statement. In the above example, in addition to the possible *ArrayIndexOutOfBoundsException*, there is a possible *NullPointerException* which will occur if the value of `M` is `null`. We can handle both possible exceptions by adding a second **catch** clause to the **try** statement:

```
try {
    double determinant = M[0][0]*M[1][1] - M[0][1]*M[1][0];
    System.out.println("The determinant of M is " + determinant);
}
catch ( ArrayIndexOutOfBoundsException e ) {
    System.out.println("M is the wrong size to have a determinant.");
}
catch ( NullPointerException e ) {
    System.out.print("Programming error!  M doesn't exist." + );
}
```

Here, the computer tries to execute the statements in the **try** clause. If no error occurs, both of the **catch** clauses are skipped. If an *ArrayIndexOutOfBoundsException* occurs, the computer



executes the body of the first `catch` clause and skips the second one. If a *NullPointerException* occurs, it jumps to the second `catch` clause and executes that.

Note that both *ArrayIndexOutOfBoundsException* and *NullPointerException* are subclasses of *RuntimeException*. It's possible to catch all *RuntimeExceptions* with a single `catch` clause. For example:

```
try {
    double determinant = M[0][0]*M[1][1] - M[0][1]*M[1][0];
    System.out.println("The determinant of M is " + determinant);
}
catch ( RuntimeException err ) {
    System.out.println("Sorry, an error has occurred.");
    System.out.println("The error was: " + err);
}
```

The `catch` clause in this `try` statement will catch any exception belonging to class *RuntimeException* or to any of its subclasses. This shows why exception classes are organized into a class hierarchy. It allows you the option of casting your net narrowly to catch only a specific type of exception. Or you can cast your net widely to catch a wide class of exceptions. Because of subclassing, when there are multiple `catch` clauses in a `try` statement, it is possible that a given exception might match several of those `catch` clauses. For example, an exception of type *NullPointerException* would match `catch` clauses for *NullPointerException*, *RuntimeException*, *Exception*, or *Throwable*. In this case, only the **first** `catch` clause that matches the exception is executed.

The example I've given here is not particularly realistic. You are not very likely to use exception-handling to guard against null pointers and bad array indices. This is a case where careful programming is better than exception handling: Just be sure that your program assigns a reasonable, non-null value to the array `M`. You would certainly resent it if the designers of Java forced you to set up a `try..catch` statement every time you wanted to use an array! This is why handling of potential *RuntimeExceptions* is not mandatory. There are just too many things that might go wrong! (This also shows that exception-handling does not solve the problem of program robustness. It just gives you a tool that will in many cases let you approach the problem in a more organized way.)

\* \* \*

I have still not completely specified the syntax of the `try` statement. There is one additional element: the possibility of a **finally clause** at the end of a `try` statement. The complete syntax of the `try` statement can be described as:

```
try {
    <statements>
}
<optional-catch-clauses>
<optional-finally-clause>
```

Note that the `catch` clauses are also listed as optional. The `try` statement can include zero or more `catch` clauses and, optionally, a `finally` clause. The `try` statement **must** include one or the other. That is, a `try` statement can have either a `finally` clause, or one or more `catch` clauses, or both. The syntax for a `catch` clause is

```
catch ( <exception-class-name> <variable-name> ) {
    <statements>
}
```

and the syntax for a **finally** clause is

```
finally {  
    <statements>  
}
```

The semantics of the **finally** clause is that the block of statements in the **finally** clause is guaranteed to be executed as the last step in the execution of the **try** statement, whether or not any exception occurs and whether or not any exception that does occur is caught and handled. The **finally** clause is meant for doing essential cleanup that under no circumstances should be omitted. One example of this type of cleanup is closing a network connection. Although you don't yet know enough about networking to look at the actual programming in this case, we can consider some pseudocode:

```
try {  
    open a network connection  
}  
catch ( IOException e ) {  
    report the error  
    return // Don't continue if connection can't be opened!  
}  
  
// At this point, we KNOW that the connection is open.  
  
try {  
    communicate over the connection  
}  
catch ( IOException e ) {  
    handle the error  
}  
finally {  
    close the connection  
}
```

The **finally** clause in the second **try** statement ensures that the network connection will definitely be closed, whether or not an error occurs during the communication. The first **try** statement is there to make sure that we don't even try to communicate over the network unless we have successfully opened a connection. The pseudocode in this example follows a general pattern that can be used to robustly obtain a resource, use the resource, and then release the resource.

### 8.3.3 Throwing Exceptions

There are times when it makes sense for a program to deliberately throw an exception. This is the case when the program discovers some sort of exceptional or error condition, but there is no reasonable way to handle the error at the point where the problem is discovered. The program can throw an exception in the hope that some other part of the program will catch and handle the exception. This can be done with a **throw statement**. You have already seen an example of this in Subsection 4.3.5. In this section, we cover the **throw** statement more fully. The syntax of the **throw** statement is:

```
throw <exception-object> ;
```

The *<exception-object>* must be an object belonging to one of the subclasses of `Throwable`. Usually, it will in fact belong to one of the subclasses of `Exception`. In most cases, it will be a newly constructed object created with the `new` operator. For example:

```
throw new ArithmeticException("Division by zero");
```

The parameter in the constructor becomes the error message in the exception object; if `e` refers to the object, the error message can be retrieved by calling `e.getMessage()`. (You might find this example a bit odd, because you might expect the system itself to throw an *ArithmeticException* when an attempt is made to divide by zero. So why should a programmer bother to throw the exception? Recall that if the numbers that are being divided are of type `int`, then division by zero will indeed throw an *ArithmeticException*. However, no arithmetic operations with floating-point numbers will ever produce an exception. Instead, the special value `Double.NaN` is used to represent the result of an illegal operation. In some situations, you might prefer to throw an *ArithmeticException* when a real number is divided by zero.)

An exception can be thrown either by the system or by a `throw` statement. The exception is processed in exactly the same way in either case. Suppose that the exception is thrown inside a `try` statement. If that `try` statement has a `catch` clause that handles that type of exception, then the computer jumps to the `catch` clause and executes it. The exception has been *handled*. After handling the exception, the computer executes the `finally` clause of the `try` statement, if there is one. It then continues normally with the rest of the program, which follows the `try` statement. If the exception is not immediately caught and handled, the processing of the exception will continue.

When an exception is thrown during the execution of a subroutine and the exception is not handled in the same subroutine, then that subroutine is terminated (after the execution of any pending `finally` clauses). Then the routine that called that subroutine gets a chance to handle the exception. That is, if the subroutine was called inside a `try` statement that has an appropriate `catch` clause, then *that* `catch` clause will be executed and the program will continue on normally from there. Again, if the second routine does not handle the exception, then it also is terminated and the routine that called *it* (if any) gets the next shot at the exception. The exception will crash the program only if it passes up through the entire chain of subroutine calls without being handled. (In fact, even this is not quite true: In a multithreaded program, only the thread in which the exception occurred is terminated.)

A subroutine that might generate an exception can announce this fact by adding a clause “`throws <exception-class-name>`” to the header of the routine. For example:

```
/**
 * Returns the larger of the two roots of the quadratic equation
 * A*x*x + B*x + C = 0, provided it has any roots. If A == 0 or
 * if the discriminant, B*B - 4*A*C, is negative, then an exception
 * of type IllegalArgumentException is thrown.
 */
static public double root( double A, double B, double C )
    throws IllegalArgumentException {
    if (A == 0) {
        throw new IllegalArgumentException("A can't be zero.");
    }
    else {
        double disc = B*B - 4*A*C;
        if (disc < 0)
            throw new IllegalArgumentException("Discriminant < zero.");
    }
}
```

```

        return (-B + Math.sqrt(disc)) / (2*A);
    }
}

```

As discussed in the previous section, the computation in this subroutine has the preconditions that  $A \neq 0$  and  $B*B-4*A*C \geq 0$ . The subroutine throws an exception of type *IllegalArgumentException* when either of these preconditions is violated. When an illegal condition is found in a subroutine, throwing an exception is often a reasonable response. If the program that called the subroutine knows some good way to handle the error, it can catch the exception. If not, the program will crash—and the programmer will know that the program needs to be fixed.

A **throws** clause in a subroutine heading can declare several different types of exception, separated by commas. For example:

```

void processArray(int[] A) throws NullPointerException,
                                   ArrayIndexOutOfBoundsException { ...

```

### 8.3.4 Mandatory Exception Handling

In the preceding example, declaring that the subroutine `root()` can throw an *IllegalArgumentException* is just a courtesy to potential readers of this routine. This is because handling of *IllegalArgumentException*s is not “mandatory.” A routine can throw an *IllegalArgumentException* without announcing the possibility. And a program that calls that routine is free either to catch or to ignore the exception, just as a programmer can choose either to catch or to ignore an exception of type *NullPointerException*.

For those exception classes that require mandatory handling, the situation is different. If a subroutine can throw such an exception, that fact **must** be announced in a **throws** clause in the routine definition. Failing to do so is a syntax error that will be reported by the compiler.

On the other hand, suppose that some statement in the body of a subroutine can generate an exception of a type that requires mandatory handling. The statement could be a **throw** statement, which throws the exception directly, or it could be a call to a subroutine that can throw the exception. In either case, the exception **must** be handled. This can be done in one of two ways: The first way is to place the statement in a **try** statement that has a **catch** clause that handles the exception; in this case, the exception is handled within the subroutine, so that any caller of the subroutine will never see the exception. The second way is to declare that the subroutine can throw the exception. This is done by adding a “**throws**” clause to the subroutine heading, which alerts any callers to the possibility that an exception might be generated when the subroutine is executed. The caller will, in turn, be forced either to handle the exception in a **try** statement or to declare the exception in a **throws** clause in its own header.

Exception-handling is mandatory for any exception class that is not a subclass of either *Error* or *RuntimeException*. Exceptions that require mandatory handling generally represent conditions that are outside the control of the programmer. For example, they might represent bad input or an illegal action taken by the user. There is no way to **avoid** such errors, so a robust program has to be prepared to handle them. The design of Java makes it impossible for programmers to ignore the possibility of such errors.

Among the exceptions that require mandatory handling are several that can occur when using Java’s input/output routines. This means that you can’t even use these routines unless you understand something about exception-handling. Chapter 11 deals with input/output and uses mandatory exception-handling extensively.

### 8.3.5 Programming with Exceptions

Exceptions can be used to help write robust programs. They provide an organized and structured approach to robustness. Without exceptions, a program can become cluttered with `if` statements that test for various possible error conditions. With exceptions, it becomes possible to write a clean implementation of an algorithm that will handle all the normal cases. The exceptional cases can be handled elsewhere, in a `catch` clause of a `try` statement.

When a program encounters an exceptional condition and has no way of handling it immediately, the program can throw an exception. In some cases, it makes sense to throw an exception belonging to one of Java's predefined classes, such as *IllegalArgumentException* or *IOException*. However, if there is no standard class that adequately represents the exceptional condition, the programmer can define a new exception class. The new class must extend the standard class *Throwable* or one of its subclasses. In general, if the programmer does **not** want to require mandatory exception handling, the new class will extend *RuntimeException* (or one of its subclasses). To create a new exception class that **does** require mandatory handling, the programmer can extend one of the other subclasses of *Exception* or can extend *Exception* itself.

Here, for example, is a class that extends *Exception*, and therefore requires mandatory exception handling when it is used:

```
public class ParseError extends Exception {
    public ParseError(String message) {
        // Create a ParseError object containing
        // the given message as its error message.
        super(message);
    }
}
```

The class contains only a constructor that makes it possible to create a *ParseError* object containing a given error message. (The statement “`super(message)`” calls a constructor in the superclass, *Exception*. See Subsection 5.6.3.) Of course the class inherits the `getMessage()` and `printStackTrace()` routines from its superclass. If `e` refers to an object of type *ParseError*, then the function call `e.getMessage()` will retrieve the error message that was specified in the constructor. But the main point of the *ParseError* class is simply to exist. When an object of type *ParseError* is thrown, it indicates that a certain type of error has occurred. (*Parsing*, by the way, refers to figuring out the syntax of a string. A *ParseError* would indicate, presumably, that some string that is being processed by the program does not have the expected form.)

A `throw` statement can be used in a program to throw an error of type *ParseError*. The constructor for the *ParseError* object must specify an error message. For example:

```
throw new ParseError("Encountered an illegal negative number.");
```

or

```
throw new ParseError("The word '" + word
                    + "' is not a valid file name.");
```

If the `throw` statement does not occur in a `try` statement that catches the error, then the subroutine that contains the `throw` statement must declare that it can throw a *ParseError* by adding the clause “`throws ParseError`” to the subroutine heading. For example,

```
void getUserData() throws ParseError {
    . . .
}
```

This would not be required if *ParseError* were defined as a subclass of *RuntimeException* instead of *Exception*, since in that case exception handling for *ParseErrors* would not be mandatory.

A routine that wants to handle *ParseErrors* can use a `try` statement with a `catch` clause that catches *ParseErrors*. For example:

```
try {
    getUserData();
    processUserData();
}
catch (ParseError pe) {
    . . . // Handle the error
}
```

Note that since *ParseError* is a subclass of *Exception*, a `catch` clause of the form “`catch (Exception e)`” would also catch *ParseErrors*, along with any other object of type *Exception*.

Sometimes, it's useful to store extra data in an exception object. For example,

```
class ShipDestroyed extends RuntimeException {
    Ship ship; // Which ship was destroyed.
    int where_x, where_y; // Location where ship was destroyed.
    ShipDestroyed(String message, Ship s, int x, int y) {
        // Constructor creates a ShipDestroyed object
        // carrying an error message plus the information
        // that the ship s was destroyed at location (x,y)
        // on the screen.
        super(message);
        ship = s;
        where_x = x;
        where_y = y;
    }
}
```

Here, a *ShipDestroyed* object contains an error message and some information about a ship that was destroyed. This could be used, for example, in a statement:

```
if ( userShip.isHit() )
    throw new ShipDestroyed("You've been hit!", userShip, xPos, yPos);
```

Note that the condition represented by a *ShipDestroyed* object might not even be considered an error. It could be just an expected interruption to the normal flow of a game. Exceptions can sometimes be used to handle such interruptions neatly.

\* \* \*

The ability to throw exceptions is particularly useful in writing general-purpose subroutines and classes that are meant to be used in more than one program. In this case, the person writing the subroutine or class often has no reasonable way of handling the error, since that person has no way of knowing exactly how the subroutine or class will be used. In such circumstances, a novice programmer is often tempted to print an error message and forge ahead, but this is almost never satisfactory since it can lead to unpredictable results down the line. Printing an error message and terminating the program is almost as bad, since it gives the program no chance to handle the error.

The program that calls the subroutine or uses the class needs to know that the error has occurred. In languages that do not support exceptions, the only alternative is to return some special value or to set the value of some variable to indicate that an error has occurred. For

example, the `readMeasurement()` function in Subsection 8.2.2 returns the value `-1` if the user's input is illegal. However, this only does any good if the main program bothers to test the return value. It is very easy to be lazy about checking for special return values every time a subroutine is called. And in this case, using `-1` as a signal that an error has occurred makes it impossible to allow negative measurements. Exceptions are a cleaner way for a subroutine to react when it encounters an error.

It is easy to modify the `readMeasurement()` subroutine to use exceptions instead of a special return value to signal an error. My modified subroutine throws a *ParseError* when the user's input is illegal, where *ParseError* is the subclass of *Exception* that was defined above. (Arguably, it might be reasonable to avoid defining a new class by using the standard exception class *IllegalArgumentException* instead.) The changes from the original version are shown in *italic*:

```
/**
 * Reads the user's input measurement from one line of input.
 * Precondition:  The input line is not empty.
 * Postcondition: If the user's input is legal, the measurement
 *               is converted to inches and returned.
 * @throws ParseError if the user's input is not legal.
 */
static double readMeasurement() throws ParseError {

    double inches; // Total number of inches in user's measurement.

    double measurement; // One measurement,
                        // such as the 12 in "12 miles."
    String units;       // The units specified for the measurement,
                        // such as "miles."

    char ch; // Used to peek at next character in the user's input.

    inches = 0; // No inches have yet been read.

    skipBlanks();
    ch = TextIO.peek();

    /* As long as there is more input on the line, read a measurement and
       add the equivalent number of inches to the variable, inches. If an
       error is detected during the loop, end the subroutine immediately
       by throwing a ParseError. */

    while (ch != '\n') {

        /* Get the next measurement and the units. Before reading
           anything, make sure that a legal value is there to read. */

        if ( ! Character.isDigit(ch) ) {
            throw new ParseError("Expected to find a number, but found " + ch);
        }
        measurement = TextIO.getDouble();

        skipBlanks();
        if (TextIO.peek() == '\n') {
            throw new ParseError("Missing unit of measure at end of line.");
        }
        units = TextIO.getWord();
        units = units.toLowerCase();
    }
}
```

```

    /* Convert the measurement to inches and add it to the total. */
    if (units.equals("inch")
        || units.equals("inches") || units.equals("in")) {
        inches += measurement;
    }
    else if (units.equals("foot")
        || units.equals("feet") || units.equals("ft")) {
        inches += measurement * 12;
    }
    else if (units.equals("yard")
        || units.equals("yards") || units.equals("yd")) {
        inches += measurement * 36;
    }
    else if (units.equals("mile")
        || units.equals("miles") || units.equals("mi")) {
        inches += measurement * 12 * 5280;
    }
    else {
        throw new ParseError("\"" + units
                               + "\" is not a legal unit of measure.");
    }

    /* Look ahead to see whether the next thing on the line is
       the end-of-line. */

    skipBlanks();
    ch = TextIO.peek();

} // end while

return inches;

} // end readMeasurement()

```

In the main program, this subroutine is called in a `try` statement of the form

```

try {
    inches = readMeasurement();
}
catch (ParseError e) {
    . . . // Handle the error.
}

```

The complete program can be found in the file *LengthConverter3.java*. From the user's point of view, this program has exactly the same behavior as the program *LengthConverter2* from the previous section. Internally, however, the programs are significantly different, since *LengthConverter3* uses exception-handling.

## 8.4 Assertions

IN THIS SHORT SECTION, we look at **assertions**, another feature of the Java programming language that can be used to aid in the development of correct and robust programs.

Recall that a precondition is a condition that must be true at a certain point in a program, for the execution of the program to continue correctly from that point. In the case where



there is a chance that the precondition might not be satisfied—for example, if it depends on input from the user—then it’s a good idea to insert an `if` statement to test it. But then the question arises, What should be done if the precondition does not hold? One option is to throw an exception. This will terminate the program, unless the exception is caught and handled elsewhere in the program.

In many cases, of course, instead of using an `if` statement to *test* whether a precondition holds, a programmer tries to write the program in a way that will *guarantee* that the precondition holds. In that case, the test should not be necessary, and the `if` statement can be avoided. The problem is that programmers are not perfect. In spite of the programmer’s intention, the program might contain a bug that screws up the precondition. So maybe it’s a good idea to check the precondition—at least during the debugging phase of program development.

Similarly, a postcondition is a condition that is true at a certain point in the program as a consequence of the code that has been executed before that point. Assuming that the code is correctly written, a postcondition is guaranteed to be true, but here again testing whether a desired postcondition is **actually** true is a way of checking for a bug that might have screwed up the postcondition. This is something that might be desirable during debugging.

The programming languages C and C++ have always had a facility for adding what are called **assertions** to a program. These assertions take the form “`assert(<condition>)`”, where `<condition>` is a **boolean**-valued expression. This condition expresses a precondition or postcondition that should hold at that point in the program. When the computer encounters an assertion during the execution of the program, it evaluates the condition. If the condition is false, the program is terminated. Otherwise, the program continues normally. This allows the programmer’s belief that the condition is true to be tested; if it is not true, that indicates that the part of the program that preceded the assertion contained a bug. One nice thing about assertions in C and C++ is that they can be “turned off” at compile time. That is, if the program is compiled in one way, then the assertions are included in the compiled code. If the program is compiled in another way, the assertions are not included. During debugging, the first type of compilation is used. The release version of the program is compiled with assertions turned off. The release version will be more efficient, because the computer won’t have to evaluate all the assertions.

Although early versions of Java did not have assertions, an assertion facility similar to the one in C/C++ has been available in Java since version 1.4. As with the C/C++ version, Java assertions can be turned on during debugging and turned off during normal execution. In Java, however, assertions are turned on and off at run time rather than at compile time. An assertion in the Java source code is always included in the compiled class file. When the program is run in the normal way, these assertions are ignored; since the condition in the assertion is not evaluated in this case, there is little or no performance penalty for having the assertions in the program. When the program is being debugged, it can be run with assertions enabled, as discussed below, and then the assertions can be a great help in locating and identifying bugs.

\* \* \*

An **assertion statement** in Java takes one of the following two forms:

```
assert <condition> ;
```

or

```
assert <condition> : <error-message> ;
```

where `<condition>` is a **boolean**-valued expression and `<error-message>` is a string or an expression of type *String*. The word “`assert`” is a reserved word in Java, which cannot be used as an

identifier. An assertion statement can be used anywhere in Java where a statement is legal.

If a program is run with assertions disabled, an assertion statement is equivalent to an empty statement and has no effect. When assertions are enabled and an assertion statement is encountered in the program, the *<condition>* in the assertion is evaluated. If the value is **true**, the program proceeds normally. If the value of the condition is **false**, then an exception of type `java.lang.AssertionError` is thrown, and the program will crash (unless the error is caught by a `try` statement). If the `assert` statement includes an *<error-message>*, then the error message string becomes the message in the *AssertionError*.

So, the statement `assert <condition> : <error-message>;` is similar to

```
if ( <condition> == false )
    throw new AssertionError( <error-message> );
```

except that the `if` statement is executed whenever the program is run, and the `assert` statement is executed only when the program is run with assertions enabled.

The question is, when to use assertions instead of exceptions? The general rule is to use assertions to test conditions that should definitely be true, if the program is written correctly. Assertions are useful for testing a program to see whether or not it is correct and for finding the errors in an incorrect program. After testing and debugging, when the program is used in the normal way, the assertions in the program will be ignored. However, if a problem turns up later, the assertions are still there in the program to be used to help locate the error. If someone writes to you to say that your program doesn't work when he does such-and-such, you can run the program with assertions enabled, do such-and-such, and hope that the assertions in the program will help you locate the point in the program where it goes wrong.

Consider, for example, the `root()` method from Subsection 8.3.3 that calculates a root of a quadratic equation. If you believe that your program will always call this method with legal arguments, then it would make sense to write the method using assertions instead of exceptions:

```
/**
 * Returns the larger of the two roots of the quadratic equation
 * A*x*x + B*x + C = 0, provided it has any roots.
 * Precondition: A != 0 and B*B - 4*A*C >= 0.
 */
static public double root( double A, double B, double C ) {
    assert A != 0 : "Leading coefficient of quadratic equation cannot be zero.";
    double disc = B*B - 4*A*C;
    assert disc >= 0 : "Discriminant of quadratic equation cannot be negative.";
    return (-B + Math.sqrt(disc)) / (2*A);
}
```

The assertions are not checked when the program is run in the normal way. If you are correct in your belief that the method is never called with illegal arguments, then checking the conditions in the assertions would be unnecessary. If your belief is not correct, the problem should turn up during testing or debugging, when the program is run with the assertions enabled.

If the `root()` method is part of a software library that you expect other people to use, then the situation is less clear. Sun's Java documentation advises that assertions should **not** be used for checking the contract of public methods: If the caller of a method violates the contract by passing illegal parameters, then an exception should be thrown. This will enforce the contract whether or not assertions are enabled. (However, while it's true that Java programmers *expect* the contract of a method to be enforced with exceptions, there are reasonable arguments for using assertions instead, in some cases.)

On the other hand, it never hurts to use an assertion to check a postcondition of a method. A postcondition is something that is supposed to be true after the method has executed, and it can be tested with an `assert` statement at the end of the method. If the postcondition is false, there is a bug in the method itself, and that is something that needs to be found during the development of the method.

\* \* \*

To have any effect, assertions must be **enabled** when the program is run. How to do this depends on what programming environment you are using. (See Section 2.6 for a discussion of programming environments.) In the usual command line environment, assertions are enabled by adding the option `-enableassertions` to the `java` command that is used to run the program. For example, if the class that contains the main program is *RootFinder*, then the command

```
java -enableassertions RootFinder
```

will run the program with assertions enabled. The `-enableassertions` option can be abbreviated to `-ea`, so the command can alternatively be written as

```
java -ea RootFinder
```

In fact, it is possible to enable assertions in just part of a program. An option of the form `-ea:<class-name>` enables only the assertions in the specified class. Note that there are no spaces between the `-ea`, the `“:”`, and the name of the class. To enable all the assertions in a package and in its sub-packages, you can use an option of the form `-ea:<package-name>...`. To enable assertions in the “default package” (that is, classes that are not specified to belong to a package, like almost all the classes in this book), use `-ea:...` . For example, to run a Java program named “MegaPaint” with assertions enabled for every class in the packages named “paintutils” and “drawing”, you would use the command:

```
java -ea:paintutils... -ea:drawing... MegaPaint
```

If you are using the Eclipse integrated development environment, you can specify the `-ea` option by creating a *run configuration*. Right-click the name of the main program class in the Package Explorer pane, and select “Run As” from the pop-up menu and then “Run...” from the submenu. This will open a dialog box where you can manage run configurations. The name of the project and of the main class will be already be filled in. Click the “Arguments” tab, and enter `-ea` in the box under “VM Arguments”. The contents of this box are added to the `java` command that is used to run the program. You can enter other options in this box, including more complicated `enableassertions` options such as `-ea:paintutils...`. When you click the “Run” button, the options will be applied. Furthermore, they will be applied whenever you run the program, unless you change the run configuration or add a new configuration. Note that it is possible to make two run configurations for the same class, one with assertions enabled and one with assertions disabled.

## 8.5 Introduction to Threads

LIKE PEOPLE, COMPUTERS can *multitask*. That is, they can be working on several different tasks at the same time. A computer that has just a single central processing unit can’t literally do two things at the same time, any more than a person can, but it can still switch its attention back and forth among several tasks. Furthermore, it is increasingly common for computers to have more than one processing unit, and such computers can literally work on several tasks simultaneously. It is likely that from now on, most of the increase in computing power will

come from adding additional processors to computers rather than from increasing the speed of individual processors. To use the full power of these multiprocessing computers, a programmer must do *parallel programming*, which means writing a program as a set of several tasks that can be executed simultaneously. Even on a single-processor computer, parallel programming techniques can be useful, since some problems can be tackled most naturally by breaking the solution into a set of simultaneous tasks that cooperate to solve the problem.

In Java, a single task is called a *thread*. The term “thread” refers to a “thread of control” or “thread of execution,” meaning a sequence of instructions that are executed one after another—the thread extends through time, connecting each instruction to the next. In a multithreaded program, there can be many threads of control, weaving through time in parallel and forming the complete fabric of the program. (Ok, enough with the metaphor, already!) Every Java program has at least one thread; when the Java virtual machine runs your program, it creates a thread that is responsible for executing the `main` routine of the program. This main thread can in turn create other threads that can continue even after the main thread has terminated. In a GUI program, there is at least one additional thread, which is responsible for handling events and drawing components on the screen. This GUI thread is created when the first window is opened. So in fact, you have already done parallel programming! When a `main` routine opens a window, both the main thread and the GUI thread can continue to run in parallel. Of course, parallel programming can be used in much more interesting ways.

Unfortunately, parallel programming is even more difficult than ordinary, single-threaded programming. When several threads are working together on a problem, a whole new category of errors is possible. This just means that techniques for writing correct and robust programs are even more important for parallel programming than they are for normal programming. (That’s one excuse for having this section in this chapter—another is that we will need threads at several points in future chapters, and I didn’t have another place in the book where the topic fits more naturally.) Since threads are a difficult topic, you will probably not fully understand everything in this section the first time through the material. Your understanding should improve as you encounter more examples of threads in future sections.

### 8.5.1 Creating and Running Threads

In Java, a thread is represented by an object belonging to the class `java.lang.Thread` (or to a subclass of this class). The purpose of a *Thread* object is to execute a single method. The method is executed in its own thread of control, which can run in parallel with other threads. When the execution of the method is finished, either because the method terminates normally or because of an uncaught exception, the thread stops running. Once this happens, there is no way to restart the thread or to use the same *Thread* object to start another thread.

There are two ways to program a thread. One is to create a subclass of *Thread* and to define the method `public void run()` in the subclass. This `run()` method defines the task that will be performed by the thread; that is, when the thread is started, it is the `run()` method that will be executed in the thread. For example, here is a simple, and rather useless, class that defines a thread that does nothing but print a message on standard output:

```
public class NamedThread extends Thread {
    private String name; // The name of this thread.
    public NamedThread(String name) { // Constructor gives name to thread.
        this.name = name;
    }
    public void run() { // The run method prints a message to standard output.
```

```

        System.out.println("Greetings from thread '" + name + "'!");
    }
}

```

To use a *NamedThread*, you must of course create an object belonging to this class. For example,

```
NamedThread greetings = new NamedThread("Fred");
```

However, creating the object does not automatically start the thread running. To do that, you must call the `start()` method in the thread object. For the example, this would be done with the statement

```
greetings.start();
```

The purpose of the `start()` method is to create a new thread of control that will execute the *Thread* object's `run()` method. The new thread runs in parallel with the thread in which the `start()` method was called, along with any other threads that already existed. This means that the code in the `run()` method will execute at the same time as the statements that follow the call to `greetings.start()`. Consider this code segment:

```

NamedThread greetings = new NamedThread("Fred");
greetings.start();
System.out.println("Thread has been started.");

```

After `greetings.start()` is executed, there are two threads. One of them will print "Thread has been started." while the other one wants to print "Greetings from thread 'Fred!'". It is important to note that *these messages can be printed in either order*. The two threads run simultaneously and will compete for access to standard output, so that they can print their messages. Whichever thread happens to be the first to get access will be the first to print its message. In a normal, single-threaded program, things happen in a definite, predictable order from beginning to end. In a multi-threaded program, there is a fundamental indeterminacy. You can't be sure what order things will happen in. This indeterminacy is what makes parallel programming so difficult!

Note that calling `greetings.start()` is **very** different from calling `greetings.run()`. Calling `greetings.run()` will execute the `run()` method in the same thread, rather than creating a new thread. This means that all the work of the `run()` will be done before the computer moves on to the statement that follows the call to `greetings.run()` in the program. There is no parallelism and no indeterminacy.

\* \* \*

I mentioned that there are two ways to program a thread. The first way was to define a subclass of *Thread*. The second is to define a class that implements the interface `java.lang.Runnable`. The *Runnable* interface defines a single method, `public void run()`. An object that implements the *Runnable* interface can be passed as a parameter to the constructor of an object of type *Thread*. When that thread's `start` method is called, the thread will execute the `run()` method in the *Runnable* object. For example, as an alternative to the *NamedThread* class, we could define the class:

```

public class NamedRunnable implements Runnable {
    private String name; // The name of this thread.
    public NamedRunnable(String name) { // Constructor gives name to object.
        this.name = name;
    }
}

```

```

    public void run() { // The run method prints a message to standard output.
        System.out.println("Greetings from thread '" + name + "'!");
    }
}

```

To use this version of the class, we would create a *NamedRunnable* object and use that object to create an object of type *Thread*:

```

NamedRunnable greetings = new NamedRunnable("Fred");
Thread greetingsThread = new Thread(greetings);
greetingsThread.start();

```

Finally, I'll note that it is sometimes convenient to define a thread using an anonymous inner class (Subsection 5.7.3). For example:

```

Thread greetingsFromFred = new Thread() {
    public void run() {
        System.out.println("Greetings from Fred!");
    }
};
greetingsFromFred.start();

```

\* \* \*

To help you understand how multiple threads are executed in parallel, we consider the sample program *ThreadTest1.java*. This program creates several threads. Each thread performs exactly the same task. The task is to count the number of integers less than 1000000 that are prime. (The particular task that is done is not important for our purposes here.) On my computer, this task takes a little more than one second of processing time. The threads that perform this task are defined by the following static nested class:

```

/**
 * When a thread belonging to this class is run it will count the
 * number of primes between 2 and 1000000. It will print the result
 * to standard output, along with its ID number and the elapsed
 * time between the start and the end of the computation.
 */
private static class CountPrimesThread extends Thread {
    int id; // An id number for this thread; specified in the constructor.
    public CountPrimesThread(int id) {
        this.id = id;
    }
    public void run() {
        long startTime = System.currentTimeMillis();
        int count = countPrimes(2,1000000); // Counts the primes.
        long elapsedTime = System.currentTimeMillis() - startTime;
        System.out.println("Thread " + id + " counted " +
            count + " primes in " + (elapsedTime/1000.0) + " seconds.");
    }
}

```

The main program asks the user how many threads to run, and then creates and starts the specified number of threads:

```

public static void main(String[] args) {
    int numberOfThreads = 0;
    while (numberOfThreads < 1 || numberOfThreads > 25) {
        System.out.print("How many threads do you want to use (1 to 25) ? ");
        numberOfThreads = TextIO.getlnInt();
        if (numberOfThreads < 1 || numberOfThreads > 25)
            System.out.println("Please enter a number between 1 and 25 !");
    }
    System.out.println("\nCreating " + numberOfThreads
                       + " prime counting threads...");
    CountPrimesThread[] worker = new CountPrimesThread[numberOfThreads];
    for (int i = 0; i < numberOfThreads; i++)
        worker[i] = new CountPrimesThread( i );
    for (int i = 0; i < numberOfThreads; i++)
        worker[i].start();
    System.out.println("Threads have been created and started.");
}

```

It would be a good idea for you to compile and run the program or to try the applet version, which can be found in the on-line version of this section.

When I ran the program with one thread, it took 1.18 seconds for my computer to do the computation. When I ran it using six threads, the output was:

```

Creating 6 prime counting threads...
Threads have been created and started.
Thread 1 counted 78498 primes in 6.706 seconds.
Thread 4 counted 78498 primes in 6.693 seconds.
Thread 0 counted 78498 primes in 6.838 seconds.
Thread 2 counted 78498 primes in 6.825 seconds.
Thread 3 counted 78498 primes in 6.893 seconds.
Thread 5 counted 78498 primes in 6.859 seconds.

```

The second line was printed immediately after the first. At this point, the main program has ended but the six threads continue to run. After a pause of about seven seconds, all six threads completed at about the same time. The order in which the threads complete is not the same as the order in which they were started, and the order is indeterminate. That is, if the program is run again, the order in which the threads complete will probably be different.

On my computer, six threads take about six times longer than one thread. This is because my computer has only one processor. Six threads, all doing the same task, take six times as much processing as one thread. With only one processor to do the work, the total elapsed time for six threads is about six times longer than the time for one thread. On a computer with two processors, the computer can work on two tasks at the same time, and six threads might complete in as little as three times the time it takes for one thread. On a computer with six or more processors, six threads might take no more time than a single thread. Because of overhead and other reasons, the actual speedup will probably be smaller than this analysis indicates, but on a multiprocessor machine, you should see a definite speedup. What happens when you run the program on your own computer? How many processors do you have?

Whenever there are more threads to be run than there are processors to run them, the computer divides its attention among all the runnable threads by switching rapidly from one thread to another. That is, each processor runs one thread for a while then switches to another thread and runs that one for a while, and so on. Typically, these “context switches” occur about 100 times or more per second. The result is that the computer makes progress on all

the tasks, and it looks to the user as if all the tasks are being executed simultaneously. This is why in the sample program, in which each thread has the same amount of work to do, all the threads complete at about the same time: Over any time period longer than a fraction of a second, the computer's time is divided approximately equally among all the threads.

When you do parallel programming in order to spread the work among several processors, you might want to take into account the number of available processors. You might, for example, want to create one thread for each processor. In Java, you can find out the number of processors by calling the function

```
Runtime.getRuntime().availableProcessors()
```

which returns an **int** giving the number of processors that are available to the Java Virtual Machine. In some cases, this might be less than the actual number of processors in the computer.

### 8.5.2 Operations on Threads

The *Thread* class includes several useful methods in addition to the `start()` method that was discussed above. I will mention just a few of them.

If `thrd` is an object of type *Thread*, then the **boolean**-valued function `thrd.isAlive()` can be used to test whether or not the thread is alive. A thread is “alive” between the time it is started and the time when it terminates. After the thread has terminated it is said to be “dead”. (The rather gruesome metaphor is also used when we refer to “killing” or “aborting” a thread.)

The static method `Thread.sleep(milliseconds)` causes the thread that executes this method to “sleep” for the specified number of milliseconds. A sleeping thread is still alive, but it is not running. While a thread is sleeping, the computer will work on any other runnable threads (or on other programs). `Thread.sleep()` can be used to insert a pause in the execution of a thread. The `sleep` method can throw an exception of type *InterruptedException*, which is an exception class that requires mandatory exception handling (see Subsection 8.3.4). In practice, this means that the `sleep` method is usually used in a `try..catch` statement that catches the potential *InterruptedException*:

```
try {
    Thread.sleep(lengthOfPause);
}
catch (InterruptedException e) {
}
```

One thread can interrupt another thread to wake it up when it is sleeping or paused for some other reason. A *Thread*, `thrd`, can be interrupted by calling its method `thrd.interrupt()`, but you are not likely to do this until you start writing rather advanced applications, and you are not likely to need to do anything in response to an *InterruptedException* (except to catch it). It's unfortunate that you have to worry about it at all, but that's the way that mandatory exception handling works.

Sometimes, it's necessary for one thread to wait for another thread to die. This is done with the `join()` method from the *Thread* class. Suppose that `thrd` is a *Thread*. Then, if another thread calls `thrd.join()`, that other thread will go to sleep until `thrd` terminates. If `thrd` is already dead when `thrd.join()` is called, then it simply has no effect—the thread that called `thrd.join()` proceeds immediately. The method `join()` can throw an *InterruptedException*, which must be handled. As an example, the following code starts several threads, waits for them all to terminate, and then outputs the elapsed time:



```

CountPrimesThread[] worker = new CountPrimesThread[numberOfThreads];
long startTime = System.currentTimeMillis();
for (int i = 0; i < numberOfThreads; i++) {
    worker[i] = new CountPrimesThread();
    worker[i].start();
}
for (int i = 0; i < numberOfThreads; i++) {
    try {
        worker[i].join(); // Sleep until worker[i] has terminated.
    }
    catch (InterruptedException e) {
    }
}
// At this point, all the worker threads have terminated.
long elapsedTime = System.currentTimeMillis() - startTime;
System.out.println("Elapsed time: " + (elapsedTime/1000.0) + " seconds.");

```

An observant reader will note that this code assumes that no *InterruptedException* will occur. To be absolutely sure that the thread `worker[i]` has terminated in an environment where *InterruptedExceptions* are possible, you would have to do something like:

```

while (worker[i].isAlive()) {
    try {
        worker[i].join();
    }
    catch (InterruptedException e) {
    }
}

```

### 8.5.3 Mutual Exclusion with “synchronized”

Programming several threads to carry out independent tasks is easy. The real difficulty arises when threads have to interact in some way. One way that threads interact is by sharing resources. When two threads need access to the same resource, such as a variable or a window on the screen, some care must be taken that they don’t try to use the same resource at the same time. Otherwise, the situation could be something like this: Imagine several cooks sharing the use of just one measuring cup, and imagine that Cook A fills the measuring cup with milk, only to have Cook B grab the cup before Cook A has a chance to empty the milk into his bowl. There has to be some way for Cook A to claim exclusive rights to the cup while he performs the two operations: Add-Milk-To-Cup and Empty-Cup-Into-Bowl.

Something similar happens with threads, even with something as simple as adding one to a counter. The statement

```
count = count + 1;
```

is actually a sequence of three operations:

- Step 1. Get the value of `count`
- Step 2. Add 1 to the value.
- Step 3. Store the new value in `count`

Suppose that several threads perform these three steps. Remember that it's possible for two threads to run at the same time, and even if there is only one processor, it's possible for that processor to switch from one thread to another at any point. Suppose that while one thread is between Step 2 and Step 3, another thread starts executing the same sequence of steps. Since the first thread has not yet stored the new value in `count`, the second thread reads the **old** value of `count` and adds one to that old value. After both threads have executed Step 3, the value of `count` has gone up only by 1 instead of by 2! This type of problem is called a **race condition**. This occurs when one thread is in the middle of a multi-step operation, and another thread changes some value or condition that the first thread is depending upon. (The first thread is “in a race” to complete all the steps before it is interrupted by another thread.) Another example of a race condition can occur in an `if` statement. Suppose the following statement, which is meant to avoid a division-by-zero error is executed by a thread:

```
if ( A != 0 )
    B = C / A;
```

If the variable `A` is shared by several threads, and if nothing is done to guard against the race condition, then it is possible that a second thread will change the value of `A` to zero between the time that the first thread checks the condition `A != 0` and the time that it does the division. This means that the thread ends up dividing by zero, even though it just checked that `A` was not zero!

To fix the problem of race conditions, there has to be some way for a thread to get **exclusive access** to a shared resource. This is not a trivial thing to implement, but Java provides a high level and relatively easy-to-use approach to exclusive access. It's done with **synchronized methods** and with the **synchronized statement**. These are used to protect shared resources by making sure that only one thread at a time will try to access the resource. Synchronization in Java actually provides only **mutual exclusion**, which means that exclusive access to a resource is only guaranteed if **every** thread that needs access to that resource uses synchronization. Synchronization is like a cook leaving a note that says, “I'm using the measuring cup.” This will get the cook exclusive access to the cup—but only if all the cooks agree to check the note before trying to grab the cup.

Because this is a difficult topic, I will start with a simple example. Suppose that we want to avoid the race condition that occurs when several threads all want to add 1 to a counter. We can do this by defining a class to represent the counter and by using synchronized methods in that class:

```
public class ThreadSafeCounter {
    private int count = 0; // The value of the counter.

    synchronized public void increment() {
        count = count + 1;
    }

    synchronized public int getValue() {
        return count;
    }
}
```

If `tsc` is of type `ThreadSafeCounter`, then any thread can call `tsc.increment()` to add 1 to the counter in a completely safe way. The fact that `tsc.increment()` is **synchronized** means that only one thread can be in this method at a time; once a thread starts executing this

method, it is guaranteed that it will finish executing it without having another thread change the value of `tsc.count` in the meantime. There is no possibility of a race condition. Note that the guarantee depends on the fact that `count` is a **private** variable. This forces all access to `tsc.count` to occur in the **synchronized** methods that are provided by the class. If `count` were **public**, it would be possible for a thread to bypass the synchronization by, for example, saying `tsc.count++`. This could change the value of `count` while another thread is in the middle of the `tsc.increment()`. Synchronization does **not** guarantee exclusive access; it only guarantees **mutual exclusion** among all the threads that are properly synchronized.

The *ThreadSafeCounter* class does not prevent all possible race conditions that might arise when using a counter. Consider the `if` statement:

```
if ( tsc.getValue() == 0 )
    doSomething();
```

where `doSomething()` is some method that requires the value of the counter to be zero. There is still a race condition here, which occurs if a second thread increments the counter between the time the first thread tests `tsc.getValue() == 0` and the time it executes `doSomething()`. The first thread needs exclusive access to the counter during the execution of the whole `if` statement. (The synchronization in the *ThreadSafeCounter* class only gives it exclusive access during the time it is evaluating `tsc.getValue()`.) We can solve the race condition by putting the `if` statement in a **synchronized** statement:

```
synchronized(tsc) {
    if ( tsc.getValue() == 0 )
        doSomething();
}
```

Note that the **synchronized** statement takes an object—`tsc` in this case—as a kind of parameter. The syntax of the **synchronized** statement is:

```
synchronized( <object> ) {
    <statements>
}
```

In Java, mutual exclusion is always associated with an object; we say that the synchronization is “on” that object. For example, the `if` statement above is “synchronized on `tsc`.” A synchronized instance method, such as those in the class *ThreadSafeCounter*, is synchronized on the object that contains the instance method. In fact, adding the **synchronized** modifier to the definition of an instance method is pretty much equivalent to putting the body of the method in a **synchronized** statement, `synchronized(this) {...}`. It is also possible to have synchronized static methods; a synchronized static method is synchronized on a special class object that represents the class that contains the static method.

The real rule of synchronization in Java is: **Two threads cannot be synchronized on the same object at the same time**; that is, they cannot simultaneously be executing code segments that are synchronized on that object. If one thread is synchronized on an object, and a second thread tries to synchronize on the same object, the second thread is forced to wait until the first thread has finished with the object. This is implemented using something called a **lock**. Every object has a lock, and that lock can be “held” by only one thread at a time. To enter a synchronized statement or synchronized method, a thread must obtain the associated object’s lock. If the lock is available, then the thread obtains the lock and immediately begins executing the synchronized code. It releases the lock after it finishes executing the synchronized code. If Thread A tries to obtain a lock that is already held by Thread B, then Thread A has

to wait until Thread B releases the lock. In fact, Thread A will go to sleep, and will not be awoken until the lock becomes available.

\* \* \*

As a simple example of shared resources, we return to the prime-counting problem. Suppose that we want to count all the primes in a given range of integers, and suppose that we want to divide the work up among several threads. Each thread will be assigned part of the range of integers and will count the primes in its assigned range. At the end of its computation, the thread has to add its count to the overall total number of primes found. The variable that represents the total is shared by all the threads. If each thread just says

```
total = total + count;
```

then there is a (small) chance that two threads will try to do this at the same time and that the final total will be wrong. To prevent this race condition, access to `total` has to be synchronized. My program uses a synchronized method to add the counts to the total:

```
synchronized private static void addToTotal(int x) {
    total = total + x;
    System.out.println(total + " primes found so far.");
}
```

The source code for the program can be found in *ThreadTest2.java*. This program counts the primes in the range 3000001 to 6000000. (The numbers are rather arbitrary.) The `main()` routine in this program creates between 1 and 5 threads and assigns part of the job to each thread. It then waits for all the threads to finish, using the `join()` method as described above, and reports the total elapsed time. If you run the program on a multiprocessor computer, it should take less time for the program to run when you use more than one thread. You can compile and run the program or try the equivalent applet in the on-line version of this section.

\* \* \*

Synchronization can help to prevent race conditions, but it introduces the possibility of another type of error, **deadlock**. A deadlock occurs when a thread waits forever for a resource that it will never get. In the kitchen, a deadlock might occur if two very simple-minded cooks both want to measure a cup of milk at the same time. The first cook grabs the measuring cup, while the second cook grabs the milk. The first cook needs the milk, but can't find it because the second cook has it. The second cook needs the measuring cup, but can't find it because the first cook has it. Neither cook can continue and nothing more gets done. This is deadlock. Exactly the same thing can happen in a program, for example if there are two threads (like the two cooks) both of which need to obtain locks on the same two objects (like the milk and the measuring cup) before they can proceed. Deadlocks can easily occur, unless great care is taken to avoid them. Fortunately, we won't be looking at any examples that require locks on more than one object, so we will avoid that source of deadlock.

#### 8.5.4 Wait and Notify

Threads can interact with each other in other ways besides sharing resources. For example, one thread might produce some sort of result that is needed by another thread. This imposes some restriction on the order in which the threads can do their computations. If the second thread gets to the point where it needs the result from the first thread, it might have to stop and wait for the result to be produced. Since the second thread can't continue, it might as well go to sleep. But then there has to be some way to notify the second thread when the result is

ready, so that it can wake up and continue its computation. Java, of course, has a way to do this kind of waiting and notification: It has `wait()` and `notify()` methods that are defined as instance methods in class *Object* and so can be used with any object. The reason why `wait()` and `notify()` should be associated with objects is not obvious, so don't worry about it at this point. It does, at least, make it possible to direct different notifications to a different recipients, depending on which object's `notify()` method is called.

The general idea is that when a thread calls a `wait()` method in some object, that thread goes to sleep until the `notify()` method in the same object is called. It will have to be called, obviously, by another thread, since the thread that called `wait()` is sleeping. A typical pattern is that Thread A calls `wait()` when it needs a result from Thread B, but that result is not yet available. When Thread B has the result ready, it calls `notify()`, which will wake Thread A up so that it can use the result. It is not an error to call `notify()` when no one is waiting; it just has no effect. To implement this, Thread A will execute code similar to the following, where `obj` is some object:

```
if ( resultIsAvailable() == false )
    obj.wait(); // wait for noification that the result is available
useTheResult();
```

while Thread B does something like:

```
generateTheResult();
obj.notify(); // send out a notification that the result is available
```

Now, there is a really nasty race condition in this code. The two threads might execute their code in the following order:

1. Thread A checks `resultIsAvailable()` and finds that the result is not ready, so it decides to execute the `obj.wait()` statement, but before it does,
2. Thread B finishes generating the result and calls `obj.notify()`
3. Thread A calls `obj.wait()` to wait for notification that the result is ready.

In Step 3, Thread A is waiting for a notification that will never come, because `notify()` has already been called. This is a kind of deadlock that can leave Thread A waiting forever. Obviously, we need some kind of synchronization. The solution is to enclose both Thread A's code and Thread B's code in `synchronized` statements, and it is very natural to synchronize on the same object, `obj`, that is used for the calls to `wait()` and `notify()`. In fact, since synchronization is almost always needed when `wait()` and `notify()` are used, Java makes it an absolute requirement. In Java, a thread can legally call `obj.wait()` or `obj.notify()` only if that thread holds the synchronization lock associated with the object `obj`. If it does not hold that lock, then an exception is thrown. (The exception is of type *IllegalMonitorStateException*, which does not require mandatory handling and which is typically not caught.) One further complication is that the `wait()` method can throw an *InterruptedException* and so should be called in a `try` statement that handles the exception.

To make things more definite, let's consider a *producer/consumer* problem where one thread produces a result that is consumed by another thread. Assume that there is a shared variable named `sharedResult` that is used to transfer the result from the producer to the consumer. When the result is ready, the producer sets the variable to a non-null value. The producer can check whether the result is ready by testing whether the value of `sharedResult` is null. We will use a variable named `lock` for synchronization. The the code for the producer thread could have the form:

```

makeResult = generateTheResult(); // Not synchronized!
synchronized(lock) {
    sharedResult = makeResult;
    lock.notify();
}

```

while the consumer would execute code such as:

```

synchronized(lock) {
    while ( sharedResult == null ) {
        try {
            lock.wait();
        }
        catch (InterruptedException e) {
        }
    }
    useResult = sharedResult;
}
useTheResult(useResult); // Not synchronized!

```

The calls to `generateTheResult()` and `useTheResult()` are not synchronized, which allows them to run in parallel with other threads that might also synchronize on `lock`. Since `sharedResult` is a shared variable, all references to `sharedResult` should be synchronized, so the references to `sharedResult` must be inside the `synchronized` statements. The goal is to do as little as possible (but not less) in synchronized code segments.

If you are uncommonly alert, you might notice something funny: `lock.wait()` does not finish until `lock.notify()` is executed, but since both of these methods are called in `synchronized` statements that synchronize on the same object, shouldn't it be impossible for both methods to be running at the same time? In fact, `lock.wait()` is a special case: When the consumer thread calls `lock.wait()`, it gives up the lock that it holds on the synchronization object, `lock`. This gives the producer thread a chance to execute the `synchronized(lock)` block that contains the `lock.notify()` statement. After the producer thread exits from this block, the lock is returned to the consumer thread so that it can continue.

The producer/consumer pattern can be generalized and made more useful without making it any more complex. In the general case, multiple results are produced by one or more producer threads and are consumed by one or more consumer threads. Instead of having just one `sharedResult` object, we keep a list of objects that have been produced but not yet consumed. Producer threads add objects to this list. Consumer threads remove objects from this list. The only time when a thread is blocked from running is when a consumer thread tries to get a result from the list, and no results are available. It is easy to encapsulate the whole producer/consumer pattern in a class (where I assume that there is a class *ResultType* that represents the result objects):

```

/**
 * An object of type ProducerConsumer represents a list of results
 * that are available for processing. Results are added to the list
 * by calling the produce method and are removed by calling consume.
 * If no result is available when consume is called, the method will
 * not return until a result becomes available.
 */
private static class ProducerConsumer {

    private ArrayList<ResultType> items = new ArrayList<ResultType>();

```

```

        // This ArrayList holds results that have been produced and are waiting
        // to be consumed. See Subsection 7.3.3 for information on ArrayList.

public void produce(ResultType item) {
    synchronized(items) {
        items.add(item); // Add item to the list of results.
        items.notify(); // Notify any thread waiting in consume() method.
    }
}

public ResultType consume() {
    ResultType item;
    synchronized(items) {
        // If no results are available, wait for notification from produce().
        while (items.size() == 0) {
            try {
                items.wait();
            }
            catch (InterruptedException e) {
            }
        }
        // At this point, we know that at least one result is available.
        item = items.remove(0);
    }
    return item;
}
}

```

For an example of a program that uses a *ProducerConsumer* class, see *ThreadTest3.java*. This program performs the same task as *ThreadTest2.java*, but the threads communicate using the producer/consumer pattern instead of with a shared variable.

Going back to our kitchen analogy for a moment, consider a restaurant with several waiters and several cooks. If we look at the flow of customer orders into the kitchen, the waiters “produce” the orders and leave them in a pile. The orders are “consumed” by the cooks; whenever a cook needs a new order to work on, she picks one up from the pile. The pile of orders, of course, plays the role of the list of result objects in the producer/consumer pattern. Note that the only time that a cook has to wait is when she needs a new order to work on, and there are no orders in the pile. The cook must wait until one of the waiters places an order in the pile. We can complete the analogy by imagining that the waiter rings a bell when he places the order in the pile—ringing the bell is like calling the `notify()` method to notify the cooks that an order is available.

A final note on `notify`: It is possible for several threads to be waiting for notification. A call to `obj.notify()` will wake only one of the threads that is waiting on `obj`. If you want to wake all threads that are waiting on `obj`, you can call `obj.notifyAll()`. And a final note on `wait`: There is another version of `wait()` that takes a number of milliseconds as a parameter. A thread that calls `obj.wait(milliseconds)` will wait only up to the specified number of milliseconds for a notification. If a notification doesn’t occur during that period, the thread will wake up and continue without the notification. In practice, this feature is most often used to let a waiting thread wake periodically while it is waiting in order to perform some periodic task, such as causing a message “Waiting for computation to finish” to blink.

### 8.5.5 Volatile Variables

And a final note on communication among threads: In general, threads communicate by sharing variables and accessing those variables in synchronized methods or synchronized statements. However, synchronization is fairly expensive computationally, and excessive use of it should be avoided. So in some cases, it can make sense for threads to refer to shared variables without synchronizing their access to those variables.

However, a subtle problem arises when the value of a shared variable is set by one thread and used in another. Because of the way that threads are implemented in Java, the second thread might not see the changed value of the variable immediately. That is, it is possible that a thread will continue to see the **old** value of the shared variable for some time after the value of the variable has been changed by another thread. This is because threads are allowed to **cache** shared data. That is, each thread can keep its own local copy of the shared data. When one thread changes the value of a shared variable, the local copies in the caches of other threads are not immediately changed, so the other threads continue to see the old value.

When a synchronized method or statement is entered, threads are forced to update their caches to the most current values of the variables in the cache. So, using shared variables in synchronized code is always safe.

It is still possible to use a shared variable **outside** of synchronized code, but in that case, the variable must be declared to be **volatile**. The **volatile** keyword is a modifier that can be added to a variable declaration, as in

```
private volatile int count;
```

If a variable is declared to be **volatile**, no thread will keep a local copy of that variable in its cache. Instead, the thread will always use the official, main copy of the variable. This means that any change made to the variable will immediately be available to all threads. This makes it safe for threads to refer to **volatile** shared variables even outside of synchronized code. (Remember, though, that synchronization is still the only way to prevent race conditions.)

When the **volatile** modifier is applied to an object variable, only the variable itself is declared to be volatile, not the contents of the object that the variable points to. For this reason, **volatile** is generally only used for variables of simple types such as primitive types and enumerated types.

A typical example of using volatile variables is to send a signal from one thread to another that tells the second thread to terminate. The two threads would share a variable

```
volatile boolean terminate = false;
```

The run method of the second thread would check the value of **terminate** frequently and end when the value of **terminate** becomes true:

```
public void run() {
    while (true) {
        if (terminate)
            return;

        . // Do some work
        .
    }
}
```

This thread will run until some other thread sets the value of **terminate** to true. Something like this is really the only clean way for one thread to cause another thread to die.



(By the way, you might be wondering why threads should use local data caches in the first place, since it seems to complicate things unnecessarily. Caching is allowed because of the structure of multiprocessing computers. In many multiprocessing computers, each processor has some local memory that is directly connected to the processor. A thread's cache is stored in the local memory of the processor on which the thread is running. Access to this local memory is much faster than access to other memory, so it is more efficient for a thread to use a local copy of a shared variable rather than some "master copy" that is stored in non-local memory.)

## 8.6 Analysis of Algorithms

THIS CHAPTER HAS CONCENTRATED mostly on correctness of programs. In practice, another issue is also important: *efficiency*. When analyzing a program in terms of efficiency, we want to look at questions such as, "How long does it take for the program to run?" and "Is there another approach that will get the answer more quickly?" Efficiency will always be less important than correctness; if you don't care whether a program works correctly, you can make it run very quickly indeed, but no one will think it's much of an achievement! On the other hand, a program that gives a correct answer after ten thousand years isn't very useful either, so efficiency is often an important issue.

The term "efficiency" can refer to efficient use of almost any resource, including time, computer memory, disk space, or network bandwidth. In this section, however, we will deal exclusively with time efficiency, and the major question that we want to ask about a program is, how long does it take to perform its task?

It really makes little sense to classify an individual program as being "efficient" or "inefficient." It makes more sense to compare two (correct) programs that perform the same task and ask which one of the two is "more efficient," that is, which one performs the task more quickly. However, even here there are difficulties. The running time of a program is not well-defined. The run time can be different depending on the number and speed of the processors in the computer on which it is run and, in the case of Java, on the design of the Java Virtual Machine which is used to interpret the program. It can depend on details of the compiler which is used to translate the program from high-level language to machine language. Furthermore, the run time of a program depends on the size of the problem which the program has to solve. It takes a sorting program longer to sort 10000 items than it takes it to sort 100 items. When the run times of two programs are compared, it often happens that Program A solves small problems faster than Program B, while Program B solves large problems faster than Program A, so that it is simply not the case that one program is faster than the other in all cases.

In spite of these difficulties, there is a field of computer science dedicated to analyzing the efficiency of programs. The field is known as *Analysis of Algorithms*. The focus is on algorithms, rather than on programs as such, to avoid having to deal with multiple implementations of the same algorithm written in different languages, compiled with different compilers, and running on different computers. Analysis of Algorithms is a mathematical field that abstracts away from these down-and-dirty details. Still, even though it is a theoretical field, every working programmer should be aware of some of its techniques and results. This section is a very brief introduction to some of those techniques and results. Because this is not a mathematics book, the treatment will be rather informal.

One of the main techniques of analysis of algorithms is *asymptotic analysis*. The term "asymptotic" here means basically "the tendency in the long run." An asymptotic analysis of

an algorithm's run time looks at the question of how the run time depends on the size of the problem. The analysis is asymptotic because it only considers what happens to the run time as the size of the problem increases without limit; it is not concerned with what happens for problems of small size or, in fact, for problems of any fixed finite size. Only what happens in the long run, as the problem size increases without limit, is important. Showing that Algorithm A is asymptotically faster than Algorithm B doesn't necessarily mean that Algorithm A will run faster than Algorithm B for problems of size 10 or size 1000 or even size 1000000—it only means that if you keep increasing the problem size, you will eventually come to a point where Algorithm A is faster than Algorithm B. An asymptotic analysis is only a first approximation, but in practice it often gives important and useful information.

\* \* \*

Central to asymptotic analysis is **Big-Oh notation**. Using this notation, we might say, for example, that an algorithm has a running time that is  $\mathcal{O}(n^2)$  or  $\mathcal{O}(n)$  or  $\mathcal{O}(\log(n))$ . These notations are read “Big-Oh of  $n$  squared,” “Big-Oh of  $n$ ,” and “Big-Oh of  $\log n$ ” (where  $\log$  is a logarithm function). More generally, we can refer to  $\mathcal{O}(f(n))$  (“Big-Oh of  $f$  of  $n$ ”), where  $f(n)$  is some function that assigns a positive real number to every positive integer  $n$ . The “ $n$ ” in this notation refers to the size of the problem. Before you can even begin an asymptotic analysis, you need some way to measure problem size. Usually, this is not a big issue. For example, if the problem is to sort a list of items, then the problem size can be taken to be the number of items in the list. When the input to an algorithm is an integer, as in the case of an algorithm that checks whether a given positive integer is prime, the usual measure of the size of a problem is the number of bits in the input integer rather than the integer itself. More generally, the number of bits in the input to a problem is often a good measure of the size of the problem.

To say that the running time of an algorithm is  $\mathcal{O}(f(n))$  means that for large values of the problem size,  $n$ , the running time of the algorithm is no bigger than some constant times  $f(n)$ . (More rigorously, there is a number  $C$  and a positive integer  $M$  such that whenever  $n$  is greater than  $M$ , the run time is less than or equal to  $C \cdot f(n)$ .) The constant takes into account details such as the speed of the computer on which the algorithm is run; if you use a slower computer, you might have to use a bigger constant in the formula, but changing the constant won't change the basic fact that the run time is  $\mathcal{O}(f(n))$ . The constant also makes it unnecessary to say whether we are measuring time in seconds, years, CPU cycles, or any other unit of measure; a change from one unit of measure to another is just multiplication by a constant. Note also that  $\mathcal{O}(f(n))$  doesn't depend at all on what happens for small problem sizes, only on what happens in the long run as the problem size increases without limit.

To look at a simple example, consider the problem of adding up all the numbers in an array. The problem size,  $n$ , is the length of the array. Using `A` as the name of the array, the algorithm can be expressed in Java as:

```
total = 0;
for (int i = 0; i < n; i++)
    total = total + A[i];
```

This algorithm performs the same operation, `total = total + A[i]`,  $n$  times. The total time spent on this operation is  $a \cdot n$ , where  $a$  is the time it takes to perform the operation once. Now, this is not the only thing that is done in the algorithm. The value of `i` is incremented and is compared to `n` each time through the loop. This adds an additional time of  $b \cdot n$  to the run time, for some constant  $b$ . Furthermore, `i` and `total` both have to be initialized to zero; this adds some constant amount  $c$  to the running time. The exact running time would then be  $(a+b) \cdot n + c$ , where the constants  $a$ ,  $b$ , and  $c$  depend on factors such as how the code is compiled

and what computer it is run on. Using the fact that  $c$  is less than or equal to  $c*n$  for any positive integer  $n$ , we can say that the run time is less than or equal to  $(a+b+c)*n$ . That is, the run time is less than or equal to a constant times  $n$ . By definition, this means that the run time for this algorithm is  $\mathcal{O}(n)$ .

If this explanation is too mathematical for you, we can just note that for large values of  $n$ , the  $c$  in the formula  $(a+b)*n+c$  is insignificant compared to the other term,  $(a+b)*n$ . We say that  $c$  is a “lower order term.” When doing asymptotic analysis, lower order terms can be discarded. A rough, but correct, asymptotic analysis of the algorithm would go something like this: Each iteration of the `for` loop takes a certain constant amount of time. There are  $n$  iterations of the loop, so the total run time is a constant times  $n$ , plus lower order terms (to account for the initialization). Disregarding lower order terms, we see that the run time is  $\mathcal{O}(n)$ .

\* \* \*

Note that to say that an algorithm has run time  $\mathcal{O}(f(n))$  is to say that its run time is no bigger than some constant times  $f(n)$  (for large values of  $n$ ).  $\mathcal{O}(f(n))$  puts an **upper limit** on the run time. However, the run time could be smaller, even much smaller. For example, if the run time is  $\mathcal{O}(n)$ , it would also be correct to say that the run time is  $\mathcal{O}(n^2)$  or even  $\mathcal{O}(n^{10})$ . If the run time is less than a constant times  $n$ , then it is certainly less than the same constant times  $n^2$  or  $n^{10}$ .

Of course, sometimes it’s useful to have a **lower limit** on the run time. That is, we want to be able to say that the run time is greater than or equal to some constant times  $f(n)$  (for large values of  $n$ ). The notation for this is  $\Omega(f(n))$ , read “Omega of  $f$  of  $n$ .” “Omega” is the name of a letter in the Greek alphabet, and  $\Omega$  is the upper case version of that letter. (To be technical, saying that the run time of an algorithm is  $\Omega(f(n))$  means that there is a positive number  $C$  and a positive integer  $M$  such that whenever  $n$  is greater than  $M$ , the run time is greater than or equal to  $C*f(n)$ .)  $\mathcal{O}(f(n))$  tells you something about the maximum amount of time that you might have to wait for an algorithm to finish;  $\Omega(f(n))$  tells you something about the minimum time.

The algorithm for adding up the numbers in an array has a run time that is  $\Omega(n)$  as well as  $\mathcal{O}(n)$ . When an algorithm has a run time that is both  $\Omega(f(n))$  and  $\mathcal{O}(f(n))$ , its run time is said to be  $\Theta(f(n))$ , read “Theta of  $f$  of  $n$ .” (Theta is another letter from the Greek alphabet.) To say that the run time of an algorithm is  $\Theta(f(n))$  means that for large values of  $n$ , the run time is between  $a*f(n)$  and  $b*f(n)$ , where  $a$  and  $b$  are constants (with  $b$  greater than  $a$ , and both greater than 0).

Let’s look at another example. Consider the algorithm that can be expressed in Java in the following method:

```
/**
 * Sorts the n array elements A[0], A[1], ..., A[n-1] into increasing order.
 */
public static simpleBubbleSort( int[] A, int n ) {
    for (int i = 0; i < n; i++) {
        // Do n passes through the array...
        for (int j = 0; j < n-1; j++) {
            if ( A[j] > A[j+1] ) {
                // A[j] and A[j+1] are out of order, so swap them
                int temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}
```

```

        }
    }
}

```

Here, the parameter  $n$  represents the problem size. The outer **for** loop in the method is executed  $n$  times. Each time the outer **for** loop is executed, the inner **for** loop is executed  $n-1$  times, so the **if** statement is executed  $n*(n-1)$  times. This is  $n^2-n$ , but since lower order terms are not significant in an asymptotic analysis, it's good enough to say that the **if** statement is executed about  $n^2$  times. In particular, the test  $A[j] > A[j+1]$  is executed about  $n^2$  times, and this fact by itself is enough to say that the run time of the algorithm is  $\Omega(n^2)$ , that is, the run time is at least some constant times  $n^2$ . Furthermore, if we look at other operations—the assignment statements, incrementing  $i$  and  $j$ , etc.—none of them are executed more than  $n^2$  times, so the run time is also  $\mathcal{O}(n^2)$ , that is, the run time is no more than some constant times  $n^2$ . Since it is both  $\Omega(n^2)$  and  $\mathcal{O}(n^2)$ , the run time of the `simpleBubbleSort` algorithm is  $\Theta(n^2)$ .

You should be aware that some people use the notation  $\mathcal{O}(f(n))$  as if it meant  $\Theta(f(n))$ . That is, when they say that the run time of an algorithm is  $\mathcal{O}(f(n))$ , they mean to say that the run time is about **equal** to a constant times  $f(n)$ . For that, they should use  $\Theta(f(n))$ . Properly speaking,  $\mathcal{O}(f(n))$  means that the run time is less than a constant times  $f(n)$ , possibly much less.

\* \* \*

So far, my analysis has ignored an important detail. We have looked at how run time depends on the problem size, but in fact the run time usually depends not just on the size of the problem but on the specific data that has to be processed. For example, the run time of a sorting algorithm can depend on the initial order of the items that are to be sorted, and not just on the number of items.

To account for this dependency, we can consider either the **worst case** run time analysis or the **average case** run time analysis of an algorithm. For a worst case run time analysis, we consider all possible problems of size  $n$  and look at the **longest** possible run time for all such problems. For an average case analysis, we consider all possible problems of size  $n$  and look at the **average** of the run times for all such problems. Usually, the average case analysis assumes that all problems of size  $n$  are equally likely to be encountered, although this is not always realistic—or even possible in the case where there is an infinite number of different problems of a given size.

In many cases, the average and the worst case run times are the same to within a constant multiple. This means that as far as asymptotic analysis is concerned, they are the same. That is, if the average case run time is  $\mathcal{O}(f(n))$  or  $\Theta(f(n))$ , then so is the worst case. However, later in the book, we will encounter a few cases where the average and worst case asymptotic analyses differ.

\* \* \*

So, what do you really have to know about analysis of algorithms to read the rest of this book? We will not do any rigorous mathematical analysis, but you should be able to follow informal discussion of simple cases such as the examples that we have looked at in this section. Most important, though, you should have a feeling for exactly what it means to say that the running time of an algorithm is  $\mathcal{O}(f(n))$  or  $\Theta(f(n))$  for some common functions  $f(n)$ . The main point is that these notations do not tell you anything about the actual numerical value of the running time of the algorithm for any particular case. They do not tell you anything at all

about the running time for small values of  $n$ . What they do tell you is something about the **rate of growth** of the running time as the size of the problem increases.

Suppose you compare two algorithms that solve the same problem. The run time of one algorithm is  $\Theta(n^2)$ , while the run time of the second algorithm is  $\Theta(n^3)$ . What does this tell you? If you want to know which algorithm will be faster for some particular problem of size, say, 100, nothing is certain. As far as you can tell just from the asymptotic analysis, either algorithm could be faster for that particular case—or in **any** particular case. But what you can say for sure is that if you look at larger and larger problems, you will come to a point where the  $\Theta(n^2)$  algorithm is faster than the  $\Theta(n^3)$  algorithm. Furthermore, as you continue to increase the problem size, the relative advantage of the  $\Theta(n^2)$  algorithm will continue to grow. There will be values of  $n$  for which the  $\Theta(n^2)$  algorithm is a thousand times faster, a million times faster, a billion times faster, and so on. This is because for any positive constants  $a$  and  $b$ , the function  $a \cdot n^3$  **grows faster** than the function  $b \cdot n^2$  as  $n$  gets larger. (Mathematically, the limit of the ratio of  $a \cdot n^3$  to  $b \cdot n^2$  is infinite as  $n$  approaches infinity.)

This means that for “large” problems, a  $\Theta(n^2)$  algorithm will definitely be faster than a  $\Theta(n^3)$  algorithm. You just don’t know—based on the asymptotic analysis alone—exactly how large “large” has to be. In practice, in fact, it is likely that the  $\Theta(n^2)$  algorithm will be faster even for fairly small values of  $n$ , and absent other information you would generally prefer a  $\Theta(n^2)$  algorithm to a  $\Theta(n^3)$  algorithm.

So, to understand and apply asymptotic analysis, it is essential to have some idea of the rates of growth of some common functions. For the power functions  $n$ ,  $n^2$ ,  $n^3$ ,  $n^4$ ,  $\dots$ , the larger the exponent, the greater the rate of growth of the function. Exponential functions such as  $2^n$  and  $10^n$ , where the  $n$  is in the exponent, have a growth rate that is faster than that of any power function. In fact, exponential functions grow so quickly that an algorithm whose run time grows exponentially is almost certainly impractical even for relatively modest values of  $n$ , because the running time is just too long. Another function that often turns up in asymptotic analysis is the logarithm function,  $\log(n)$ . There are actually many different logarithm functions, but the one that is usually used in computer science is the so-called logarithm to the base two, which is defined by the fact that  $\log(2^x) = x$  for any number  $x$ . (Usually, this function is written  $\log_2(n)$ , but I will leave out the subscript 2, since I will only use the base-two logarithm in this book.) The logarithm function grows very slowly. The growth rate of  $\log(n)$  is much smaller than the growth rate of  $n$ . The growth rate of  $n \cdot \log(n)$  is a little larger than the growth rate of  $n$ , but much smaller than the growth rate of  $n^2$ . The following table should help you understand the differences among the rates of growth of various functions:

$n$	$\log(n)$	$n \cdot \log(n)$	$n^2$	$n / \log(n)$
16	4	64	256	4.0
64	6	384	4096	10.7
256	8	2048	65536	32.0
1024	10	10240	1048576	102.4
1000000	20	19931568	1000000000000	50173.1
1000000000	30	29897352854	1000000000000000000	33447777.3

The reason that  $\log(n)$  shows up so often is because of its association with multiplying and dividing by two: Suppose you start with the number  $n$  and divide it by 2, then divide by 2 again, and so on, until you get a number that is less than or equal to 1. Then the number of divisions is equal (to the nearest integer) to  $\log(n)$ .

As an example, consider the binary search algorithm from Subsection 7.4.1. This algorithm searches for an item in a sorted array. The problem size,  $n$ , can be taken to be the length of the array. Each step in the binary search algorithm divides the number of items still under consideration by 2, and the algorithm stops when the number of items under consideration is less than or equal to 1 (or sooner). It follows that the number of steps for an array of length  $n$  is at most  $\log(n)$ . This means that the worst-case run time for binary search is  $\Theta(\log(n))$ . (The average case run time is also  $\Theta(\log(n))$ .) By comparison, the linear search algorithm, which was also presented in Subsection 7.4.1 has a run time that is  $\Theta(n)$ . The  $\Theta$  notation gives us a quantitative way to express and to understand the fact that binary search is “much faster” than linear search.

In binary search, each step of the algorithm divides the problem size by 2. It often happens that some operation in an algorithm (not necessarily a single step) divides the problem size by 2. Whenever that happens, the logarithm function is likely to show up in an asymptotic analysis of the run time of the algorithm.

Analysis of Algorithms is a large, fascinating field. We will only use a few of the most basic ideas from this field, but even those can be very helpful for understanding the differences among algorithms.

## Exercises for Chapter 8

1. Write a program that uses the following subroutine, from Subsection 8.3.3, to solve equations specified by the user.

```

/**
 * Returns the larger of the two roots of the quadratic equation
 *  $Ax^2 + Bx + C = 0$ , provided it has any roots. If  $A == 0$  or
 * if the discriminant,  $B^2 - 4AC$ , is negative, then an exception
 * of type IllegalArgumentException is thrown.
 */
static public double root( double A, double B, double C )
    throws IllegalArgumentException {
    if (A == 0) {
        throw new IllegalArgumentException("A can't be zero.");
    }
    else {
        double disc = B*B - 4*A*C;
        if (disc < 0)
            throw new IllegalArgumentException("Discriminant < zero.");
        return (-B + Math.sqrt(disc)) / (2*A);
    }
}

```

Your program should allow the user to specify values for A, B, and C. It should call the subroutine to compute a solution of the equation. If no error occurs, it should print the root. However, if an error occurs, your program should catch that error and print an error message. After processing one equation, the program should ask whether the user wants to enter another equation. The program should continue until the user answers no.

2. As discussed in Section 8.1, values of type `int` are limited to 32 bits. Integers that are too large to be represented in 32 bits cannot be stored in an `int` variable. Java has a standard class, `java.math.BigInteger`, that addresses this problem. An object of type *BigInteger* is an integer that can be arbitrarily large. (The maximum size is limited only by the amount of memory on your computer.) Since *BigIntegers* are objects, they must be manipulated using instance methods from the *BigInteger* class. For example, you can't add two *BigIntegers* with the `+` operator. Instead, if N and M are variables that refer to *BigIntegers*, you can compute the sum of N and M with the function call `N.add(M)`. The value returned by this function is a new *BigInteger* object that is equal to the sum of N and M.

The `BigInteger` class has a constructor `new BigInteger(str)`, where `str` is a string. The string must represent an integer, such as "3" or "39849823783783283733". If the string does not represent a legal integer, then the constructor throws a *NumberFormatException*.

There are many instance methods in the *BigInteger* class. Here are a few that you will find useful for this exercise. Assume that N and M are variables of type `BigInteger`.

- `N.add(M)` — a function that returns a *BigInteger* representing the sum of N and M.
- `N.multiply(M)` — a function that returns a *BigInteger* representing the result of multiplying N times M.

- `N.divide(M)` — a function that returns a *BigInteger* representing the result of dividing `N` by `M`, discarding the remainder.
- `N.signum()` — a function that returns an ordinary **int**. The returned value represents the sign of the integer `N`. The returned value is 1 if `N` is greater than zero. It is -1 if `N` is less than zero. And it is 0 if `N` is zero.
- `N.equals(M)` — a function that returns a **boolean** value that is **true** if `N` and `M` have the same integer value.
- `N.toString()` — a function that returns a *String* representing the value of `N`.
- `N.testBit(k)` — a function that returns a **boolean** value. The parameter `k` is an integer. The return value is **true** if the `k`-th bit in `N` is 1, and it is **false** if the `k`-th bit is 0. Bits are numbered from right to left, starting with 0. Testing “if (`N.testBit(0)`)” is an easy way to check whether `N` is even or odd. `N.testBit(0)` is **true** if and only if `N` is an odd number.

For this exercise, you should write a program that prints  $3N+1$  sequences with starting values specified by the user. In this version of the program, you should use *BigIntegers* to represent the terms in the sequence. You can read the user’s input into a *String* with the `TextIO.getln()` function. Use the input value to create the *BigInteger* object that represents the starting point of the  $3N+1$  sequence. Don’t forget to catch and handle the *NumberFormatException* that will occur if the user’s input is not a legal integer! You should also check that the input number is greater than zero.

If the user’s input is legal, print out the  $3N+1$  sequence. Count the number of terms in the sequence, and print the count at the end of the sequence. Exit the program when the user inputs an empty line.

3. A Roman numeral represents an integer using letters. Examples are XVII to represent 17, MCMLIII for 1953, and MMMCCCIII for 3303. By contrast, ordinary numbers such as 17 or 1953 are called Arabic numerals. The following table shows the Arabic equivalent of all the single-letter Roman numerals:

M	1000	X	10
D	500	V	5
C	100	I	1
L	50		

When letters are strung together, the values of the letters are just added up, with the following exception. When a letter of smaller value is followed by a letter of larger value, the smaller value is subtracted from the larger value. For example, IV represents  $5 - 1$ , or 4. And MCMXCV is interpreted as  $M + CM + XC + V$ , or  $1000 + (1000 - 100) + (100 - 10) + 5$ , which is 1995. In standard Roman numerals, no more than three consecutive copies of the same letter are used. Following these rules, every number between 1 and 3999 can be represented as a Roman numeral made up of the following one- and two-letter combinations:

M	1000	X	10
CM	900	IX	9
D	500	V	5
CD	400	IV	4
C	100	I	1
XC	90		



L	50
XL	40

Write a class to represent Roman numerals. The class should have two constructors. One constructs a Roman numeral from a string such as “XVII” or “MCMXCV”. It should throw a *NumberFormatException* if the string is not a legal Roman numeral. The other constructor constructs a Roman numeral from an **int**. It should throw a *NumberFormatException* if the **int** is outside the range 1 to 3999.

In addition, the class should have two instance methods. The method `toString()` returns the string that represents the Roman numeral. The method `toInt()` returns the value of the Roman numeral as an **int**.

At some point in your class, you will have to convert an **int** into the string that represents the corresponding Roman numeral. One way to approach this is to gradually “move” value from the Arabic numeral to the Roman numeral. Here is the beginning of a routine that will do this, where **number** is the **int** that is to be converted:

```
String roman = "";
int N = number;
while (N >= 1000) {
    // Move 1000 from N to roman.
    roman += "M";
    N -= 1000;
}
while (N >= 900) {
    // Move 900 from N to roman.
    roman += "CM";
    N -= 900;
}
.
. // Continue with other values from the above table.
.
```

(You can save yourself a lot of typing in this routine if you use arrays in a clever way to represent the data in the above table.)

Once you’ve written your class, use it in a main program that will read both Arabic numerals and Roman numerals entered by the user. If the user enters an Arabic numeral, print the corresponding Roman numeral. If the user enters a Roman numeral, print the corresponding Arabic numeral. (You can tell the difference by using `TextIO.peek()` to peek at the first character in the user’s input. If that character is a digit, then the user’s input is an Arabic numeral. Otherwise, it’s a Roman numeral.) The program should end when the user inputs an empty line.

4. The source code file `Expr.java` defines a class, *Expr*, that can be used to represent mathematical expressions involving the variable **x**. The expression can use the operators **+**, **-**, **\***, **/**, and **^** (where **^** represents the operation of raising a number to a power). It can use mathematical functions such as **sin**, **cos**, **abs**, and **ln**. See the source code file for full details. The *Expr* class uses some advanced techniques which have not yet been covered in this textbook. However, the interface is easy to understand. It contains only a constructor and two public methods.

The constructor `new Expr(def)` creates an *Expr* object defined by a given expression. The parameter, **def**, is a string that contains the definition. For example,

`new Expr("x^2")` or `new Expr("sin(x)+3*x")`. If the parameter in the constructor call does not represent a legal expression, then the constructor throws an *IllegalArgumentEx-ception*. The message in the exception describes the error.

If `func` is a variable of type `Expr` and `num` is of type `double`, then `func.value(num)` is a function that returns the value of the expression when the number `num` is substituted for the variable `x` in the expression. For example, if `Expr` represents the expression `3*x+1`, then `func.value(5)` is `3*5+1`, or 16. If the expression is undefined for the specified value of `x`, then the special value `Double.NaN` is returned.

Finally, `func.toString()` returns the definition of the expression. This is just the string that was used in the constructor that created the expression object.

For this exercise, you should write a program that lets the user enter an expression. If the expression contains an error, print an error message. Otherwise, let the user enter some numerical values for the variable `x`. Print the value of the expression for each number that the user enters. However, if the expression is undefined for the specified value of `x`, print a message to that effect. You can use the `boolean`-valued function `Double.isNaN(val)` to check whether a number, `val`, is `Double.NaN`.

The user should be able to enter as many values of `x` as desired. After that, the user should be able to enter a new expression. In the on-line version of this exercise, there is an applet that simulates my solution, so that you can see how it works.

5. This exercise uses the class *Expr*, which was described in Exercise 8.4 and which is defined in the source code file *Expr.java*. For this exercise, you should write a GUI program that can graph a function,  $f(x)$ , whose definition is entered by the user. The program should have a text-input box where the user can enter an expression involving the variable `x`, such as `x^2` or `sin(x-3)/x`. This expression is the definition of the function. When the user presses return in the text input box, the program should use the contents of the text input box to construct an object of type *Expr*. If an error is found in the definition, then the program should display an error message. Otherwise, it should display a graph of the function. (Note: A `JTextField` generates an `ActionEvent` when the user presses return.)

The program will need a *JPanel* for displaying the graph. To keep things simple, this panel should represent a fixed region in the  $xy$ -plane, defined by  $-5 \leq x \leq 5$  and  $-5 \leq y \leq 5$ . To draw the graph, compute a large number of points and connect them with line segments. (This method does not handle discontinuous functions properly; doing so is very hard, so you shouldn't try to do it for this exercise.) My program divides the interval  $-5 \leq x \leq 5$  into 300 subintervals and uses the 301 endpoints of these subintervals for drawing the graph. Note that the function might be undefined at one of these  $x$ -values. In that case, you have to skip that point.

A point on the graph has the form  $(x,y)$  where  $y$  is obtained by evaluating the user's expression at the given value of  $x$ . You will have to convert these real numbers to the integer coordinates of the corresponding pixel on the canvas. The formulas for the conversion are:

```
a = (int)( (x + 5)/10 * width );
b = (int)( (5 - y)/10 * height );
```

where `a` and `b` are the horizontal and vertical coordinates of the pixel, and `width` and `height` are the width and height of the panel.

You can find an applet version of my solution in the on-line version of this exercise.

6. Exercise 3.2 asked you to find the integer in the range 1 to 10000 that has the largest number of divisors. Now write a program that uses multiple threads to solve the same problem. By using threads, your program will take less time to do the computation when it is run on a multiprocessor computer. At the end of the program, output the elapsed time, the integer that has the largest number of divisors, and the number of divisors that it has. The program can be modeled on the sample prime-counting program *ThreadTest2.java* from Subsection 8.5.3.

## Quiz on Chapter 8

1. What does it mean to say that a program is *robust*?
2. Why do programming languages require that variables be declared before they are used? What does this have to do with correctness and robustness?
3. What is a *precondition*? Give an example.
4. Explain how preconditions can be used as an aid in writing correct programs.
5. Java has a predefined class called *Throwable*. What does this class represent? Why does it exist?
6. Write a method that prints out a  $3N+1$  sequence starting from a given integer, *N*. The starting value should be a parameter to the method. If the parameter is less than or equal to zero, throw an *IllegalArgumentException*. If the number in the sequence becomes too large to be represented as a value of type **int**, throw an *ArithmeticException*.
7. Rewrite the method from the previous question, using **assert** statements instead of exceptions to check for errors. What the difference between the two versions of the method when the program is run?
8. Some classes of exceptions require *mandatory exception handling*. Explain what this means.
9. Consider a subroutine `processData()` that has the header
 

```
static void processData() throws IOException
```

 Write a `try..catch` statement that calls this subroutine and prints an error message if an *IOException* occurs.
10. Why should a subroutine throw an exception when it encounters an error? Why not just terminate the program?
11. Suppose that a program uses a single thread that takes 4 seconds to run. Now suppose that the program creates two threads and divides the same work between the two threads. What can be said about the expected execution time of the program that uses two threads?
12. Consider the *ThreadSafeCounter* example from Subsection 8.5.3:

```
public class ThreadSafeCounter {
    private int count = 0; // The value of the counter.

    synchronized public void increment() {
        count = count + 1;
    }

    synchronized public int getValue() {
        return count;
    }
}
```

The `increment()` method is synchronized so that the caller of the method can complete the three steps of the operation “Get value of count,” “Add 1 to value,” “Store new value in count” without being interrupted by another thread. But `getValue()` consists of a single, simple step. Why is `getValue()` synchronized? (This is a deep and tricky question.)