SESSION 8

Programming Languages for Objects

- Introduction to Correctness and Robustness
- Writing Correct Programs
- Exceptions and try..catch
- Assertions and Annotations
- Analysis of Algorithms

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.

About fifteen 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.

Of course, there have been more recent problems. For example, computer software error contributed to the Northeast Blackout of 2003, one of the largest power outages in history. In 2006, the Airbus A380 was delayed by software incompatibility problems, at a cost of perhaps billions of dollars. In 2007, a software problem grounded thousands of planes at the Los Angeles International Airport. On May 6, 2010, a flaw in an automatic trading program apparently resulted in a 1000-point drop in the Dow Jones Industrial Average.

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);
}
```

(Be sure you understand why we can't just test "`if (3*N+1 > 2147483647)`".) 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 at the start of the year 2000 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 zero by zero or taking the square root of a negative number is `Double.NaN`. You can test whether a number `x` is this special not-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 next three sections.

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 {
   System.out.print("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 $A*x^2 + B*x + C = 0$, 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*B-4*A*C >= 0 and that A != 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*B-4*A*C >= 0 is a precondition of the program segment. The condition that A != 0 is another precondition. A precondition is defined to be a 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 {
   System.out.println("Enter A, B, and C.  B*B-4*A*C must be >=
0.");
   System.out.print("A = ");
   A = TextIO.getlnDouble();
   System.out.print("B = ");
   B = TextIO.getlnDouble();
   System.out.print("C = ");
   C = TextIO.getlnDouble();
   if (A == 0 || B*B - 4*A*C < 0)
      System.out.println("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 >= 0` and that `A != 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 real 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.

```
System.out.println("Enter your values for A, B, and C.");
System.out.print("A = ");
A = TextIO.getlnDouble();
System.out.print("B = ");
B = TextIO.getlnDouble();
System.out.print("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);
   System.out.println("A solution of A*X*X + B*X + C = 0 is " + x);
}
else if (A == 0) {
   System.out.println("The value of A cannot be zero.");
}
else {
   System.out.println("Since B*B - 4*A*C is less than zero, the");
   System.out.println("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 <= 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 section 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 `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, as well as before the start of the loop.

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 not acceptable.) 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. You might want to run the program and try this out.

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) ) {
           System.out.println(
                 "Error:  Expected to find a number, but found " +
ch);
           return -1;
       }
       measurement = TextIO.getDouble();

       skipBlanks();
       if (TextIO.peek() == '\n') {
           System.out.println(
                 "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 {
           System.out.println("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*.

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 (as a postcondition of the code that precedes the array reference) 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 in Section 3.7 that Java provides a neater, more structured alternative technique for dealing with errors that can occur while a program is running. The technique 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 Chapter 12. 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

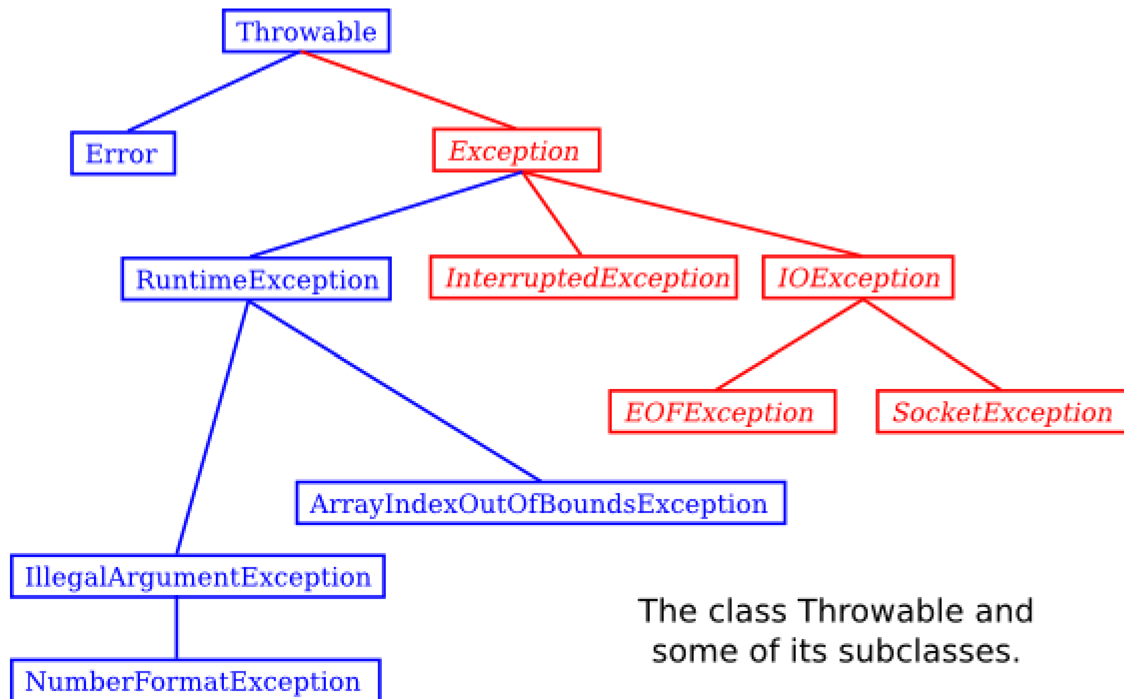
many other predefined subclasses. In addition, a programmer can create new exception classes to represent new types of exception.

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 red:



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 the method `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 default response to the exception 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 has many options. It will take a while to go through them. For one thing, a `try..catch` statement can have 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.

Of course, catching *RuntimeException* would catch many more types of exception than the two that we are interested in. It is possible to combine several specific exception types in 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 ( NullPointerException | ArrayIndexOutOfBoundsException err )
{
   System.out.println("Sorry, an error has occurred.");
   System.out.println("The error was: " + err);
}
```

Here, the two exception types are combined with a "|", the vertical line character that is also used in the boolean **or** operator. This example will catch errors of type *NullPointerException* or *ArrayIndexOutOfBoundsException*, and no other types.

The example I've been using here is not realistic, because 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. The next variation is the possibility of a finally clause at the end of a try statement. With this addition, 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-names variable-name ) {
   statements
}
```

where **exception-class-names** can be a single exception class or several classes separated by "`|`". 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
   communicate over the connection
}
catch ( IOException e ) {
   report the error
}
finally {
   if the connection was successfully opened
      close the connection
}
```

The `finally` clause ensures that the network connection will definitely be closed, whether or not an error occurs during the communication. 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.

The pattern of obtaining a resource, then using the resource, and then releasing the resource is very common. Note that the resource can only be released if no error occurred while obtaining it. And, if it was successfully obtained, then it should be closed whether or not an error occurs while using it. This pattern is so common that it leads to one last option in the `try` statement

syntax. With this option, you only need code to obtain the resource, and you don't need to worry about releasing it. That will happen automatically at the end of the `try` statement.

In order for this to work, the resource must be represented by an object that implements an interface named *AutoCloseable*, which defines a single method named `close()`, with no parameters. Standard Java classes that represent things like files and network connections already implement *AutoClosable*. So does the *Scanner* class, which was introduced in Subsection 2.4.6. In that section, I showed how to use a Scanner to read from `System.in`. Although I didn't do it in that section, it's considered good form to close a Scanner after using it. Here is an example that uses the resource pattern in a `try` statement to make sure that the Scanner is closed automatically:

```
try( Scanner in = new Scanner(System.in) ) {
    // Use the Scanner to read from standard input
}
catch (Exception e) {
    // ... some error occurred while using the Scanner
}
```

The statement that allocates the resource goes in parentheses after the word "try". The statement must have the form of a variable declaration that includes an initialization of the variable. The variable is local to the `try` statement. (You can actually declare several variables in the parentheses, separated by semicolons.) In this example, we can be sure that `in.close()` will definitely be called by the system at the end of the `try` statement, as long as the Scanner was successfully initialized.

This is all getting quite complicated, and I won't continue the discussion here. The sample program *TryStatementDemo.java* demonstrates a `try` statement with all its options, and it includes a lot of comments to help you understand what can happen when you run the program.

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 4.3.8. 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 != 0` and `B*B-4*A*C> = 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 *IllegalArgumentExceptions* 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. Exceptions that require mandatory handling are called checked exceptions. The compiler will check that such exceptions are handled by the program.

Suppose that some statement in the body of a subroutine can generate a checked exception, one 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 no caller of the subroutine can ever 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 the 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*. These checked exceptions 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 checked exceptions 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 checked exceptions 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 checked exception class, which **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.");
```

Since *ParseError* is defined as a subclass of *Exception*, it is a checked exception. If the throw statement does not occur in a try statement that catches the error, then the subroutine that contains the throw 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 *ParseErrors* would not be checked exceptions.

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 methods and classes that are meant to be used in more than one program. In this case, the person writing the method or class often has no reasonable way of handling the error, since that person has no way of knowing exactly how the method 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 method 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()` function 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 red:

```
/**
 * 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.
}
```

Assertions and Annotations

IN THIS SHORT SECTION, we look briefly at two features of Java that are not covered or used elsewhere in this textbook, assertions and annotations. They are included here for completeness, but they are mostly meant for more advanced programming.

8.4.1 Assertions

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 after all -- 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, with assertions turned on. 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 anyplace 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. Oracle'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.) One might say that assertions are for **you**, to help you in debugging your code, while exceptions are for people who use your code, to alert them that they are misusing it.

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.

8.4.2 Annotations

The term "annotation" refers to notes added to or written alongside a main text, to help you understand or appreciate the text. An annotation might be a note that you make to yourself in the margin of a book. It might be a footnote added to an old novel by an editor to explain the historical context of some event. The annotation is metadata or "metatext," that is, text written *about* the main text rather than as *part of* the main text itself.

Comments on a program are actually a kind of annotation. Since they are ignored by the compiler, they have no effect on the meaning of the program. They are there to explain that

meaning to a human reader. It is possible, of course, for another computer program (not the compiler) to process comments. That's what done in the case of Javadoc comments, which are processed by a program that uses them to create API documentation. But comments are only one type of metadata that might be added to programs.

In Java 5.0, a new feature called annotations was added to the Java language to make it easier to create new kinds of metadata for Java programs. This has made it possible for programmers to devise new ways of annotating programs, and to write programs that can read and use their annotations.

Java annotations have no direct effect on the program that they annotate. But they do have many potential uses. Some annotations are used to make the programmer's intent more explicit. Such annotations might be checked by a compiler to make sure that the code is consistent with the programmer's intention. For example, `@Override` is a standard annotation that can be used to annotate method definitions. It means that the method is intended to override (that is replace) a method with the same signature that was defined in some superclass. A compiler can check that the superclass method actually exists; if not, it can inform the programmer. An annotation used in this way is an aid to writing correct programs, since the programmer can be warned about a potential error in advance, instead of having to hunt it down later as a bug.

To annotate a method definition with the `@Override` annotation, simply place it in front of the definition. Syntactically, annotations are modifiers that are used in much the same way as built-in modifiers like "public" and "final." For example,

```
@Override public void WindowClosed(WindowEvent evt) { ... }
```

If there is no "`WindowClosed(WindowEvent)`" method in any superclass, then the compiler can issue an error. In fact, this example is based on a hard-to-find bug that I once introduced when trying to override a method named "windowClosed" with a method that I called "WindowClosed" (with an upper case "W"). If the `@Override` annotation had existed at that time -- and if I had used it -- the compiler could have rejected my code and saved me the trouble of tracking down the bug.

(Annotations are a fairly advanced feature, and I might not have mentioned them in this textbook, except that the `@Override` annotation can show up in code generated by Eclipse and other integrated development environments.)

There are two other standard annotations. One is `@Deprecated`, which can be used to mark deprecated classes, methods, and variables. (A deprecated item is one that is considered to be obsolete, but is still part of the Java language for backwards compatibility for old code.) Use of this annotation would allow a compiler to generate warnings when the deprecated item is used.

The other standard annotation is `@SurpressWarnings`, which can be used by a compiler to turn off warning messages that would ordinarily be generated when a class or method is compiled. `@SuppressWarnings` is an example of an annotation that has a parameter. The

parameter tells what class of warnings are to be suppressed. For example, when a class or method is annotated with

```
@SuppressWarnings("deprecation")
```

then no warnings about the use of deprecated items will be emitted when the class or method is compiled. There are other types of warning that can be suppressed; unfortunately the list of warnings and their names is not standardized and will vary from one compiler to another.

Note, by the way, that the syntax for annotation parameters -- especially for an annotation that accepts multiple parameters -- is not the same as the syntax for method parameters. I won't cover the annotation syntax here.

Programmers can define new annotations for use in their code. Such annotations are ignored by standard compilers and programming tools, but it's possible to write programs that can understand the annotations and check for their presence in source code. It is even possible to create annotations that will be retained at run-time and become part of the running program. In that case, a program can check for annotations in the actual compiled code that is being executed, and take actions that depend on the presence of the annotation or the values of its parameters.

Annotations can help programmers to write correct programs. To use an example from the Java documentation, they can help with the creation of "boilerplate" code -- that is, code that has a very standardized format and that can be generated mechanically. Often, boilerplate code is generated based on other code. Doing that by hand is a tedious and error-prone process. A simple example might be code to save certain aspects of a program's state to a file and to restore it later. The code for reading and writing the values of all the relevant state variables is highly repetitious. Instead of writing that code by hand, a programmer could use an annotation to mark the variables that are part of the state that is to be saved. A program could then be used to check for the annotations and generate the save-and-restore code. In fact, it would even be possible to do without that code altogether, if the program checks for the presence of the annotation at run time to decide which variables to save and restore.

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 $O(n^2)$ or $O(n)$ or $O(\log(n))$. 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 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 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*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 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 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*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*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)*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 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 O(n).

Note that to say that an algorithm has run time O(f(n)) is to say that its run time is no bigger than some constant times f(n) (for large values of n). 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 O(n), it would also be correct to say that the run time is $O(n^2)$ or even $O(n^{10})$. 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 Ω(f(n)), read "Omega of f of n." "Omega" is the name of a letter in the Greek alphabet, and Ω is the upper case version of that letter. (To be technical, saying that the run time of an algorithm is Ω(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).) O(f(n)) tells you something about the maximum amount of time that you might have to wait for an algorithm to finish; Ω(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 Ω(n) as well as O(n). When an algorithm has a run time that is both Ω(f(n)) and O(f(n)), its run time is said to be Θ(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 Θ(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 $O(n^2)$, that is, the run time is no more than some constant times n^2. Since it is both $\Omega(n^2)$ and $O(n^2)$, the run time of the simpleBubbleSort algorithm is $\Theta(n^2)$.

You should be aware that some people use the notation O(f(n)) as if it meant Θ(f(n)). That is, when they say that the run time of an algorithm is 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 Θ(f(n)). Properly speaking, 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 O(f(n)) or Θ(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 O(f(n)) or Θ(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 $a*n^3$ **grows faster** than the function $b*n^2$ as n gets larger. (Mathematically, the limit of the ratio of $a*n^3$ to $b*n^2$ 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, ..., 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*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 grows of various functions:

n	log(n)	n*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 Θ 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.