CHAPTER 11

Recursion

11.1 RECURSIVE void METHODS 579
Example: Vertical Numbers 579
Tracing a Recursive Call 582
A Closer Look at Recursion 585
Pitfall: Infinite Recursion 586
Stacks for Recursion ✜ 588
Pitfall: Stack Overflow ✜ 589
Recursion versus Iteration 590

11.2 RECURSIVE METHODS THAT RETURN A VALUE 591
General Form for a Recursive Method That Returns a Value 591
Example: Another Powers Method 591

11.3 THINKING RECURSIVELY 596
Recursive Design Techniques 596
Binary Search ✜ 597
Efficiency of Binary Search ✜ 603

CHAPTER SUMMARY 605
ANSWERS TO SELF-TEST EXERCISES 605
PROGRAMMING PROJECTS 609

11 Recursion

After a lecture on cosmology and the structure of the solar system, William James was accosted by a little old lady.

"Your theory that the sun is the center of the solar system, and the earth is a ball which rotates around it has a very convincing ring to it, Mr. James, but it's wrong. I've got a better theory," said the little old lady.

"And what is that, madam?" inquired James politely.

"That we live on a crust of earth which is on the back of a giant turtle."

Not wishing to demolish this absurd little theory by bringing to bear the masses of scientific evidence he had at his command, James decided to gently dissuade his opponent by making her see some of the inadequacies of her position.

"If your theory is correct, madam," he asked, "what does this turtle stand on?"

"You're a very clever man, Mr. James, and that's a very good question" replied the little old lady, "but I have an answer to it. And it is this: the first turtle stands on the back of a second, far larger, turtle, who stands directly under him."

"But what does this second turtle stand on?" persisted James patiently.

To this the little old lady crowed triumphantly. "It's no use, Mr. James — it's turtles all the way down."

J. R. Ross, *Constraints on Variables in Syntax*

Introduction

recursive method

A method definition that includes a call to itself is said to be **recursive**. Like most modern programming languages, Java allows methods to be recursive, and if used with a little care, this can be a useful programming technique. In this chapter we introduce the basic techniques needed for defining successful recursive methods. There is nothing in this chapter that is truly unique to Java. If you are already familiar with recursion you can safely skip this chapter. No new Java elements are introduced in this chapter.

Prerequisites

Except for the last subsection on binary search, this chapter uses material only from Chapters 1–5. The last subsection entitled "Binary Search" also uses the basic material on one-dimensional arrays from Chapter 6.

You may postpone all or part of this chapter if you wish. Nothing in the rest of this book requires any of this chapter.

11.1 Recursive void Methods

I remembered too that night which is at the middle of the Thousand and One Nights when Scheherazade (through a magical oversight of the copyist) begins to relate word for word the story of the Thousand and One Nights, establishing the risk of coming once again to the night when she must repeat it, and thus to infinity.

Jorge Luis Borges, *The Garden of Forking Paths*

When you are writing a method to solve a task, one basic design technique is to break the task into subtasks. Sometimes it turns out that at least one of the subtasks is a smaller example of the same task. For example, if the task is to search a list for a particular value, you might divide this into the subtask of searching the first half of the list and the subtask of searching the second half of the list. The subtasks of searching the halves of the list are "smaller" versions of the original task. Whenever one subtask is a smaller version of the original task to be accomplished, you can solve the original task by using a recursive method. We begin with a simple example to illustrate this technique. (For simplicity our examples are static methods; however, recursive methods need not be static.)

RECURSION

In Java, a method definition may contain an invocation of the method being defined. In such cases the method is said to be **recursive**.

Example

VERTICAL NUMBERS

Display 11.1 contains a demonstration program for a recursive method named `writeVertical`, which takes one (nonnegative) `int` argument and writes that `int` to the screen with the digits going down the screen one per line. For example, the invocation

```
writeVertical(1234);
```

would produce the output

```
1
2
3
4
```

Display 11.1 A Recursive void Method

```
1   public class RecursionDemo1
2   {
3       public static void main(String[] args)
4       {
5           System.out.println("writeVertical(3):");
6           writeVertical(3);

7           System.out.println("writeVertical(12):");
8           writeVertical(12);

9           System.out.println("writeVertical(123):");
10          writeVertical(123);
11      }

12      public static void writeVertical(int n)
13      {
14          if (n < 10)
15          {
16              System.out.println(n);
17          }
18          else //n is two or more digits long:
19          {
20              writeVertical(n/10);
21              System.out.println(n%10);
22          }
23      }
24  }
```

Sample Dialogue

```
writeVertical(3):
3
writeVertical(12):
1
2
writeVertical(123):
1
2
3
```

The task to be performed by `writeVertical` may be broken down into the following two subtasks:

Simple Case: If `n < 10`, then write the number `n` to the screen.

After all, if the number is only one digit long, the task is trivial.

Recursive Case: If `n >= 10`, then do two subtasks:

1. Output all the digits except the last digit.
2. Output the last digit.

For example, if the argument were 1234, the first part would output

```
1
2
3
```

and the second part would output 4. This decomposition of tasks into subtasks can be used to derive the method definition.

Subtask 1 is a smaller version of the original task, so we can implement this subtask with a recursive call. Subtask 2 is just the simple case we listed above. Thus, an outline of our algorithm for the method `writeVertical` with parameter n is given by the following pseudocode:

```
if (n < 10)
{
    System.out.println(n);
}
else //n is two or more digits long:
{
    writeVertical(the number n with the last digit removed);
    System.out.println(the last digit of n);
}
```

recursive subtask

If you observe the following identities, it is easy to convert this pseudocode to a complete Java method definition:

`n/10` is the number n with the last digit removed.
`n%10` is the last digit of n.

For example, `1234/10` evaluates to `123` and `1234%10` evaluates to `4`.

The following is the complete code for the method:

```
public static void writeVertical(int n)
{
    if (n < 10)
    {
        System.out.println(n);
    }
    else //n is two or more digits long:
    {
        writeVertical(n/10);
        System.out.println(n%10);
    }
}
```

TRACING A RECURSIVE CALL

Let's see exactly what happens when the following method call is made (as in Display 11.1):

```
writeVertical(123);
```

When this method call is executed, the computer proceeds just as it would with any method call. The argument `123` is substituted for the parameter `n` and the body of the method is executed. After the substitution of `123` for `n`, the code to be executed is equivalent to

```
if (123 < 10)
{
    System.out.println(123);
}
else //n is two or more digits long:
{
    writeVertical(123/10);
    System.out.println(123%10);
}
```

Computation will stop here until the recursive call returns.

Since `123` is not less than `10`, the `else` part is executed. However, the `else` part begins with the method call

```
writeVertical(n/10);
```

which (since `n` is equal to `123`) is the call

```
writeVertical(123/10);
```

which is equivalent to

```
writeVertical(12);
```

When execution reaches this recursive call, the current method computation is placed in suspended animation and this recursive call is executed. When this recursive call is finished, the execution of the suspended computation will return to this point and the suspended computation will continue from this point.

The recursive call

```
writeVertical(12);
```

is handled just like any other method call. The argument `12` is substituted for the parameter `n` and the body of the method is executed. After substituting `12` for `n`, there are two computations, one suspended and one active, as follows:

```
if (123 < 10)
{
    System
}
else //n i
{
    writeV
    System
}
```

```
if (12 < 10)
{
    System.out.println(12);
}
else //n is two or more digits long:
{
    writeVertical(12/10);
    System.out.println(12%10);
}
```

Computation will stop here until the recursive call returns.

Since `12` is not less than `10`, the `else` part is executed. However, as you already saw, the `else` part begins with a recursive call. The argument for the recursive call is `n/10`, which in this case is equivalent to `12/10`. So, this second computation of the method `writeVertical` is suspended and the following recursive call is executed:

```
writeVertical(12/10);
```

which is equivalent to

```
writeVertical(1);
```

At this point, there are two suspended computations waiting to resume and the computer begins to execute this new recursive call, which is handled just like all the previous recursive calls. The argument `1` is substituted for the parameter `n` and the body of the method is executed. At this point, the computation looks like the following:

```
if (123 < 10)
{
    S
}
else
{
    w
    S
}
```

```
if (12 < 10)
{

}
else
{


}
```

```
if (1 < 10)
{
    System.out.println(1);
}
else //n is two or more digits long:
{
    writeVertical(1/10);
    System.out.println(1%10);
}
```

No recursive call this time

output the digit 1

When the body of the method is executed this time, something different happens. Since `1` is less than `10`, the Boolean expression in the `if-else` statement is `true`, so the statement before the `else` is executed. That statement is simply an output statement that writes the argument `1` to the screen, so the call `writeVertical(1)` writes `1` to the screen and ends without any recursive call.

When the call `writeVertical(1)` ends, the suspended computation that is waiting for it to end resumes where that suspended computation left off, as shown by the following:

```
if (123 < 10)
{
    S
}
else
{
    w
    S
}
```

```
if (12 < 10)
{
    System.out.println(12);
}
else //n is two or more digits long:
{
    writeVertical(12/10);
    System.out.println(12%10);
}
```

Computation resumes here.

output the digit 2

When this suspended computation resumes, it executes an output statement that outputs the value `12%10`, which is `2`. That ends that computation, but there is yet another suspended computation waiting to resume. When this last suspended computation resumes, the situation is

```
if (123 < 10)
{
    System.out.println(123);
}
else //n is two or more digits long:
{
    writeVertical(123/10);
    System.out.println(123%10);
}
```

Computation resumes here.

output the digit 3

When this last suspended computation resumes, it outputs the value `123%10`, which is `3`, and the execution of the original method call ends. And, sure enough, the digits `1`, `2`, and `3` have been written to the screen one per line, in that order.

A CLOSER LOOK AT RECURSION

The definition of the method `writeVertical` uses recursion. Yet, we did nothing new or different in evaluating the method call `writeVertical(123)`. We treated it just like any of the method calls we saw in previous chapters. We simply substituted the argument `123` for the parameter `n` and then executed the code in the body of the method definition. When we reached the recursive call

```
writeVertical(123/10);
```

we simply repeated this process one more time.

how recursion works

The computer keeps track of recursive calls in the following way. When a method is called, the computer plugs in the arguments for the parameter(s) and begins to execute the code. If it should encounter a recursive call, then it temporarily stops its computation, because it must know the result of the recursive call before it can proceed. It saves all the information it needs to continue the computation later on, and proceeds to evaluate the recursive call. When the recursive call is completed, the computer returns to finish the outer computation.

how recursion ends

The Java language places no restrictions on how recursive calls are used in method definitions. However, in order for a recursive method definition to be useful, it must be designed so that any call of the method must ultimately terminate with some piece of code that does not depend on recursion. The method may call itself, and that recursive call may call the method again. The process may be repeated any number of times. However, the process will not terminate unless eventually one of the recursive calls does not depend on further recursion. The general outline of a successful recursive method definition is as follows:

- One or more cases in which the method accomplishes its task by using recursive call(s) to accomplish one or more smaller versions of the task.
- One or more cases in which the method accomplishes its task without the use of any recursive calls. These cases without any recursive calls are called **base cases** or **stopping cases**.

base case
stopping case

Often an `if-else` statement determines which of the cases will be executed. A typical scenario is for the original method call to execute a case that includes a recursive call. That recursive call may in turn execute a case that requires another recursive call. For some number of times, each recursive call produces another recursive call, but eventually one of the stopping cases should apply. *Every call of the method must eventually lead to a stopping case, or else the method call will never end because of an infinite chain of recursive calls.* (In practice, a call that includes an infinite chain of recursive calls will usually terminate abnormally rather than actually running forever.)

The most common way to ensure that a stopping case is eventually reached is to write the method so that some (positive) numeric quantity is decreased on each recursive call and to provide a stopping case for some "small" value. This is how we designed the method `writeVertical` in Display 11.1 When the method `writeVertical` is called, that call produces a recursive call with a smaller argument. This continues with

each recursive call producing another recursive call until the argument is less than 10. When the argument is less than 10, the method call ends without producing any more recursive calls and the process works its way back to the original call and the process ends.

General Form of a Recursive Method Definition

The general outline of a successful recursive method definition is as follows:

- One or more cases that include one or more recursive calls to the method being defined. These recursive calls should solve "smaller" versions of the task performed by the method being defined.
- One or more cases that include no recursive calls. These cases without any recursive calls are called **base cases** or **stopping cases**.

Pitfall

Infinite Recursion

In the example of the method `writeVertical` discussed in the previous subsections, the series of recursive calls eventually reached a call of the method that did not involve recursion (that is, a stopping case was reached). If, on the other hand, every recursive call produces another recursive call, then a call to the method will, in theory, run forever. This is called **infinite recursion**. In practice, such a method will typically run until the computer runs out of resources and the program terminates abnormally.

Examples of infinite recursion are not hard to come by. The following is a syntactically correct Java method definition, which might result from an attempt to define an alternative version of the method `writeVertical`:

```
public static void newWriteVertical(int n)
{
        newWriteVertical(n/10);
        System.out.println(n%10);
}
```

If you embed this definition in a program that calls this method, the program will compile with no error messages and you can run the program. Moreover, the definition even has a certain reasonableness to it. It says that to output the argument to `newWriteVertical`, first output all but the last digit and then output the last digit. However, when called, this method will produce an infinite sequence of recursive calls. If you call `newWriteVertical(12)`, that execution will stop to execute the recursive call `newWriteVertical(12/10)`, which is equivalent to `newWriteVertical(1)`. The execution of that recursive call will, in turn, stop to execute the recursive call

```
newWriteVertical(1/10);
```

which is equivalent to

```
newWriteVertical(0);
```

That, in turn, will stop to execute the recursive call `newWriteVertical(0/10);` which is also equivalent to

```
newWriteVertical(0);
```

and that will produce another recursive call to again execute the same recursive method call `newWriteVertical(0);` and so on, forever. Since the definition of `newWriteVertical` has no stopping case, the process will proceed forever (or until the computer runs out of resources).

Self-Test Exercises

1. What is the output of the following program?

```
public class Exercise1
{
    public static void main(String[] args)
    {
        cheers(3);
    }

    public static void cheers(int n)
    {
        if (n == 1)
        {
            System.out.println("Hurray");
        }
        else
        {
            System.out.println("Hip ");
            cheers(n - 1);
        }
    }
}
```

2. Write a recursive `void` method that has one parameter which is a integer and that writes to the screen the number of asterisks `'*'` given by the argument. The output should be all on one line. You can assume the argument is positive.

3. Write a recursive `void` method that has one parameter, which is a positive integer. When called, the method writes its argument to the screen backward. That is, if the argument is `1234`, it outputs the following to the screen:

```
4321
```

4. Write a recursive `void` method that takes a single (positive) `int` argument `n` and writes the integers `1`, `2`, . . . , `n` to the screen.
5. Write a recursive `void` method that takes a single (positive) `int` argument `n` and writes integers `n`, `n−1`, . . . , `3`, `2`, `1` to the screen. Hint: Notice that you can get from the code for exercise 4 to that for this exercise (or vice versa) by an exchange of as little as two lines.

STACKS FOR RECURSION ✣

stack

To keep track of recursion, and a number of other things, most computer systems use a structure called a *stack*. A **stack** is a very specialized kind of memory structure that is analogous to a stack of paper. In this analogy, there is an inexhaustible supply of extra blank sheets of paper. To place some information in the stack, it is written on one of these sheets of paper and placed on top of the stack of papers. To place more information in the stack, a clean sheet of paper is taken, the information is written on it, and this new sheet of paper is placed on top of the stack. In this straightforward way, more and more information may be placed on the stack.

Getting information out of the stack is also accomplished by a very simple procedure. The top sheet of paper can be read, and when it is no longer needed, it is thrown away. There is one complication: Only the top sheet of paper is accessible. In order to read, say, the third sheet from the top, the top two sheets must be thrown away. Since the last sheet that is put on the stack is the first sheet taken off the stack, a stack is often called a **last-in/first-out** memory structure, abbreviated **LIFO**.

last-in/first-out

recursion

Using a stack, the computer can easily keep track of recursion. Whenever a method is called, a new sheet of paper is taken. The method definition is copied onto this sheet of paper, and the arguments are plugged for the function parameters. Then the computer starts to execute the body of the function definition. When it encounters a recursive call, it stops the computation it is doing on that sheet in order to compute the value returned by the recursive call. But, before computing the recursive call, it saves enough information so that, when it does finally determine the value returned by the recursive call, it can continue the stopped computation. This saved information is written on a sheet of paper and placed on the stack. A new sheet of paper is used for the recursive call. The computer writes a second copy of the method definition on this new sheet of paper, plugs in the arguments for the method parameters, and starts to execute the recursive call. When it gets to a recursive call within the recursively called copy, it repeats the process of saving information on the stack and using a new sheet of paper for the new recursive call. This process is illustrated in the subsection entitled "Tracing a Recursive Call." Even though we did not call it a stack at the time, the illustrations of computations placed one on top of the other illustrate the actions of the stack.

This process continues until some recursive call to the method completes its computation without producing any more recursive calls. When that happens, the computer turns its attention to the top sheet of paper on the stack. This sheet contains the

partially completed computation that is waiting for the recursive computation that just ended. So, it is possible to proceed with that suspended computation. When that suspended computation ends, the computer discards that sheet of paper and the suspended computation that is below it on the stack becomes the computation on top of the stack. The computer turns its attention to the suspended computation that is now on the top of the stack, and so forth. The process continues until the computation on the bottom sheet is completed. Depending on how many recursive calls are made and how the function definition is written, the stack may grow and shrink in any fashion. Notice that the sheets in the stack can only be accessed in a last-in/first-out fashion, but that is exactly what is needed to keep track of recursive calls. Each suspended version is waiting for the completion of the version directly above it on the stack.

Of course, computers do not have stacks of paper. This is just an analogy. The computer uses portions of memory rather than pieces of paper. The contents of one of these portions of memory ("sheets of paper") is called a **stack frame** or **activation record**. These stack frames are handled in the last-in/first-out manner we just discussed. (These stack frames do not contain a complete copy of the function definition, but merely reference a single copy of the function definition. However, a stack frame contains enough information to allow the computer to act as if the stack frame contains a complete copy of the function definition.)

stack frame

Stack

A **stack** is a last-in/first-out memory structure. The first item referenced or removed from a stack is always the last item entered into the stack. Stacks are used by computers to keep track of recursion (and for other purposes).

Pitfall

Stack Overflow

There is always some limit to the size of the stack. If there is a long chain in which a method makes a recursive call to itself, and that call results in another recursive call, and that call produces yet another recursive call, and so forth, then each recursive call in this chain will cause another suspended computation to be placed on the stack. If this chain is too long, then the stack will attempt to grow beyond its limit. This is an error condition known as a **stack overflow**. If you receive an error message that says *stack overflow*, it is likely that some method call has produced an excessively long chain of recursive calls. One common cause of stack overflow is infinite recursion. If a method is recursing infinitely, then it will eventually try to make the stack exceed any stack size limit.

RECURSION VERSUS ITERATION

Recursion is not absolutely necessary. In fact, some programming languages do not allow it. Any task that can be accomplished using recursion can also be done in some other way without using recursion. For example, Display 11.2 contains a nonrecursive version of the method given in Display 11.1. The nonrecursive version of a method typically uses a loop (or loops) of some sort in place of recursion. For that reason, the nonrecursive version is usually referred to as an **iterative version**. If the definition of the method `writeVertical` given in Display 11.1 is replaced by the version given in Display 11.2, then the output will be the same. As is true in this case, a recursive version of a method can sometimes be much simpler than an iterative version. The full program with the iterative version of the method is given in the file `IterativeDemo1` on the accompanying CD.

iterative version

extra code on CD

efficiency

A recursively written method will usually run slower and use more storage than an equivalent iterative version. The computer must do extra work manipulating the stack to keep track of the recursion. However, since the system does all this for you automatically, using recursion can sometimes make your job as a programmer easier, and can sometimes produce code that is easier to understand.

Display 11.2 Iterative Version of the Method in Display 11.1

```
1   public static void writeVertical(int n)
2   {
3       int nsTens = 1;
4       int leftEndPiece = n;
5       while (leftEndPiece > 9)
6       {
7           leftEndPiece = leftEndPiece/10;
8           nsTens = nsTens*10;
9       }
10      //nsTens is a power of ten that has the same number
11      //of digits as n. For example, if n is 2345, then
12      //nsTens is 1000.

13      for (int powerOf10 = nsTens;
14              powerOf10 > 0; powerOf10 = powerOf10/10)
15      {
16          System.out.println(n/powerOf10);
17          n = n%powerOf10;
18      }
19  }
```

Self-Test Exercises

6. If your program produces an error message that says *stack overflow,* what is a likely source of the error?
7. Write an iterative version of the method `cheers` defined in Self-Test Exercise 1.
8. Write an iterative version of the method defined in Self-Test Exercise 2.
9. Write an iterative version of the method defined in Self-Test Exercise 3.
10. Trace the recursive solution you made to Self-Test Exercise 4.
11. Trace the recursive solution you made to Self-Test Exercise 5.

11.2 Recursive Methods that Return a Value

To iterate is human, to recurse divine.

Anonymous

GENERAL FORM FOR A RECURSIVE METHOD THAT RETURNS A VALUE

The recursive methods you have seen thus far are all `void` methods, but recursion is not limited to `void` methods. A recursive method can return a value of any type. The technique for designing recursive methods that return a value is basically the same as what you learned for `void` methods. An outline for a successful recursive method definition that returns a value is as follows:

- One or more cases in which the value returned is computed in terms of calls to the same method (that is, using recursive calls). As was the case with `void` methods, the arguments for the recursive calls should intuitively be "smaller."
- One or more cases in which the value returned is computed without the use of any recursive calls. These cases without any recursive calls are called **base cases** or **stopping cases** (just as they were with `void` methods).

This technique is illustrated in the next Programming Example.

Example

ANOTHER POWERS METHOD

In Chapter 5 we introduced the static method pow of the class `Math`. This method pow computes powers. For example, `Math.pow(2.0,3.0)` returns $2.0^{3.0}$, so the following sets the variable `result` equal to `8.0`:

```
double result = Math.pow(2.0, 3.0);
```

The method pow takes two arguments of type `double` and returns a value of type `double`. Display 11.3 contains a recursive definition for a static method that is similar but that works with the type `int` rather than `double`. This new method is called `power`. For example, the following will set the value of `result2` equal to 8, since 2^3 is 8:

```
int result2 = power(2, 3);
```

Outside the defining class, this would be written

```
int result2 = RecursionDemo2.power(2, 3);
```

Display 11.3 The Recursive Method power

```
1   public class RecursionDemo2
2   {
3       public static void main(String[] args)
4       {
5           for (int n = 0; n < 4; n++)
6               System.out.println("3 to the power " + n
7                   + " is " + power(3, n));
8       }

9       public static int power(int x, int n)
10      {
11          if (n < 0)
12          {
13              System.out.println("Illegal argument to power.");
14              System.exit(0);
15          }

16          if (n > 0)
17              return ( power(x, n - 1)*x );
18          else // n == 0
19               return (1);
20      }
21  }
```

SAMPLE DIALOGUE

```
3 to the power 0 is 1
3 to the power 1 is 3
3 to the power 2 is 9
3 to the power 3 is 27
```

Our main reason for defining the method power is to have a simple example of a recursive method, but there are situations in which the method power would be preferable to the method pow. The method pow returns a value of type double, which is only an approximate quantity. The method power returns a value of type int, which is an exact quantity. In some situations, you might need the additional accuracy provided by the method power.

The definition of the method power is based on the following formula:

x^n is equal to x^{n-1} * x

Translating this formula into Java says that the value returned by power(x, n) should be the same as the value of the expression

```
power(x, n - 1)*x
```

The definition of the method power given in Display 11.3 does return this value for power(x, n), provided n > 0.

The case where n is equal to 0 is the stopping case. If n is 0, then power(x, n) simply returns 1 (since x^0 is 1).

Let's see what happens when the method power is called with some sample values. First consider the simple expression:

```
power(2, 0)
```

When the method is called, the value of x is set equal to 2, the value of n is set equal to 0, and the code in the body of the method definition is executed. Since the value of n is a legal value, the if-else statement is executed. Since this value of n is not greater than 0, the return statement after the else is used, so the method call returns 1. Thus, the following would set the value of result3 equal to 1:

```
int result3 = power(2, 0);
```

Now let's look at an example that involves a recursive call. Consider the expression

```
power(2, 1)
```

When the method is called, the value of x is set equal to 2, the value of n is set equal to 1, and the code in the body of the method definition is executed. Since this value of n is greater than 0, the following return statement is used to determine the value returned:

```
return ( power(x, n - 1)*x );
```

which in this case is equivalent to

```
return ( power(2, 0)*2 );
```

At this point, the computation of power(2, 1) is suspended, a copy of this suspended computation is placed on the stack, and the computer then starts a new method call to compute the value

of power(2, 0). As you have already seen, the value of power(2, 0) is 1. After determining the value of power(2, 0), the computer replaces the expression power(2, 0) with its value of 1 and resumes the suspended computation. The resumed computation determines the final value for power(2, 1) from the above return statement as

power(2, 0)*2 is 1*2 which is 2

so the final value returned for power(2, 1) is 2. So, the following would set the value of result4 equal to 2:

```
int result4 = power(2, 1);
```

Larger numbers for the second argument will produce longer chains of recursive calls. For example, consider the statement

```
System.out.println(power(2, 3));
```

The value of power(2, 3) is calculated as follows:

```
power(2, 3) is power(2, 2)*2
  power(2, 2) is power(2, 1)*2
    power(2, 1) is power(2, 0)*2
      power(2, 0) is 1 (stopping case)
```

When the computer reaches the stopping case, power(2, 0), there are three suspended computations. After calculating the value returned for the stopping case, it resumes the most recently suspended computations to determine the value of power(2, 1). After that, the computer completes each of the other suspended computations, using each value computed as a value to plug into another suspended computation, until it reaches and completes the computation for the original call power(2, 3). The details of the entire computation are illustrated in Display 11.4.

Self-Test Exercises

12. What is the output of the following program?

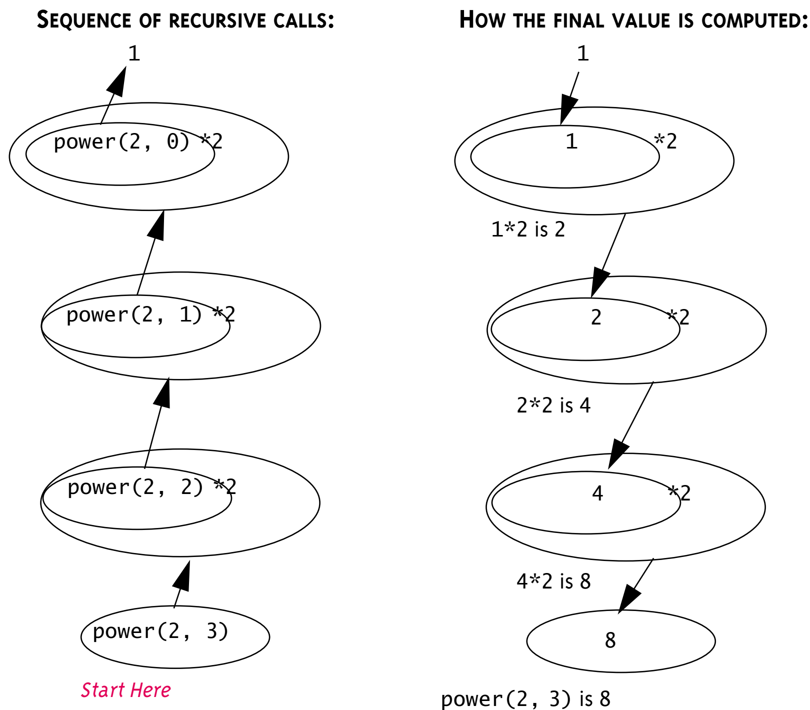
```
public class Exercise12
{
    public static void main(String[] args)
    {
        System.out.println(mystery(3));
    }

    public static int mystery(int n)
    {
        if (n <= 1)
            return 1;
```

```
            else
                return ( mystery(n - 1) + n );
        }
    }
```

13. What is the output of the following program? What well-known mathematical method is `rose`?

```
public class Exercise13
{
    public static void main(String[] args)
    {
        System.out.println(rose(4));
    }

    public static int rose(int n)
    {
        if (n <= 0)
```

Display 11.4 Evaluating the Recursive Method Call power(2,3)



```
                return 1;
            else
                return ( rose(n - 1) * n );
        }
    }
```

14. Redefine the method `power` (Display 11.3) so that it also works for negative exponents. To do this, you also have to change the type of the value returned to `double`. The method heading for the redefined version of `power` is as follows:

```
/**
 Precondition: If n < 0, then x is not 0.
 Returns x to the power n.
*/
public static double power(int x, int n)
```

Hint: x^{-n} is equal to $1/(x^n)$.

11.3 Thinking Recursively

There are two kinds of people in the world, those who divide the world into two kinds of people and those who do not.

Anonymous

RECURSIVE DESIGN TECHNIQUES

When defining and using recursive methods, you do not want to be continually aware of the stack and the suspended computations. The power of recursion comes from the fact that you can ignore that detail and let the computer do the bookkeeping for you. Consider the example of the method `power` in Display 11.3. The way to think of the definition of `power` is as follows:

`power(x, n)` returns `power(x, n - 1)*x`

Since x^n is equal to $x^{n-1}*x$, this is the correct value to return, provided that the computation will always reach a stopping case and will correctly compute the stopping case. So, after checking that the recursive part of the definition is correct, all you need to check is that the chain of recursive calls will always reach a stopping case and that the stopping case will always return the correct value. In other words, all that you need to do is check that the following three properties are satisfied:

criteria for methods that return a value

1. There is no infinite recursion. (A recursive call may lead to another recursive call, and that may lead to another, and so forth, but every such chain of recursive calls eventually reaches a stopping case.)
2. Each stopping case returns the correct value for that case.

3. For the cases that involve recursion: *if* all recursive calls return the correct value, *then* the final value returned by the method is the correct value.

   For example, consider the method `power` in Display 11.3:

1. **There is no infinite recursion:** The second argument to `power(x, n)` is decreased by one in each recursive call, so any chain of recursive calls must eventually reach the case `power(x, 0)`, which is the stopping case. Thus, there is no infinite recursion.
2. **Each stopping case returns the correct value for that case:** The only stopping case is `power(x, 0)`. A call of the form `power(x, 0)` always returns `1`, and the correct value for x^0 is `1`. So the stopping case returns the correct value.
3. **For the cases that involve recursion: *if* all recursive calls return the correct value, *then* the final value returned by the method is the correct value:** The only case that involves recursion is when `n > 1`. When `n > 1`, `power(x, n)` returns

   ```
   power(x, n – 1)*x.
   ```

   To see that this is the correct value to return, note that: *if* `power(x, n – 1)` returns the correct value, *then* `power(x, n – 1)` returns x^{n-1} and so `power(x, n)` returns

   $x^{n-1} * x$, which is x^n

   and that is the correct value for `power(x, n)`.

That's all you need to check to be sure that the definition of `power` is correct. (The above technique is known as *mathematical induction,* a concept that you may have heard about in a mathematics class. However, you do not need to be familiar with the term *mathematical induction* to use this technique.)

We gave you three criteria to use in checking the correctness of a recursive method that returns a value. Basically the same rules can be applied to a recursive `void` method. If you show that your recursive `void` method definition satisfies the following three criteria, then you will know that your `void` method performs correctly:

criteria for `void` methods

1. There is no infinite recursion.
2. Each stopping case performs the correct action for that case.
3. For each of the cases that involve recursion: *if* all recursive calls perform their actions correctly, *then* the entire case performs correctly.

BINARY SEARCH ✣

In this subsection we will develop a recursive method that searches an array to find out whether it contains a specified value. For example, the array may contain a list of the numbers for credit cards that are no longer valid. A store clerk needs to search the list to see if a customer's card is valid or invalid.

The indices of the array `a` are the integers `0` through `finalIndex`. To make the task of searching the array easier, we will assume that the array is sorted. Hence, we know the following:

```
a[0] ≤ a[1] ≤ a[2] ≤... ≤ a[finalIndex]
```

In fact, the binary search algorithm we will use requires that the array be sorted like this.

When searching an array, you are likely to want to know both whether the value is in the array and, if it is, where it is in the array. For example, if you are searching for a credit card number, then the array index may serve as a record number. Another array indexed by these same indices may hold a phone number or other information to use for reporting the suspicious card. Hence, if the sought-after value is in the array, we will have our method return an index of where the sought-after value is located. If the value is not in the array, our method will return `-1`. (The array may contain repeats, which is why we say "an index" and not "the index.")

Now let us proceed to produce an algorithm to solve this task. It will help to visualize the problem in very concrete terms. Suppose the list of numbers is so long that it takes a book to list them all. This is in fact how invalid credit card numbers are distributed to stores that do not have access to computers. If you are a clerk and are handed a credit card, you must check to see if it is on the list and hence invalid. How would you proceed? Open the book to the middle and see if the number is there. If it is not and it is smaller than the middle number, then work backward toward the beginning of the book. If the number is larger than the middle number, you work your way toward the end of the book. This idea produces our first draft of an algorithm:

algorithm
first version

```
mid = approximate midpoint between 0 and finalIndex;
if (key == a[mid])
   return mid;
else if (key < a[mid])
    search a[0] through a[mid - 1];
else if (key > a[mid])
    search a[mid + 1] through a[finalIndex];
```

Since the searchings of the shorter lists are smaller versions of the very task we are designing the algorithm to perform, this algorithm naturally lends itself to the use of recursion. The smaller lists can be searched with recursive calls to the algorithm itself.

Our pseudocode is a bit too imprecise to be easily translated into Java code. The problem has to do with the recursive calls. There are two recursive calls shown:

```
search a[0] through a[mid - 1];
```

and

```
search a[mid + 1] through a[finalIndex];
```

more parameters

To implement these recursive calls, we need two more parameters. A recursive call specifies that a subrange of the array is to be searched. In one case it is the elements indexed by `0` through `mid - 1`. In the other case it is the elements indexed by `mid + 1` through `finalIndex`. The two extra parameters will specify the first and last indices of the search, so we will call them `first` and `last`. Using these parameters for the lowest and highest indices, instead of `0` and `finalIndex`, we can express the pseudocode more precisely as follows:

algorithm
first refinement

```
To search a[first] through a[last] do the following:
mid = approximate midpoint between first and last;
if (key == a[mid])
```

```
        return mid;
    else if (key < a[mid])
        return the result of searching a[first] through a[mid - 1];
    else if (key > a[mid])
        return the result of searching a[mid + 1] through a[last];
```

To search the entire array, the algorithm would be executed with `first` set equal to `0` and `last` set equal to `finalIndex`. The recursive calls will use other values for `first` and `last`. For example, the first recursive call would set `first` equal to `0` and `last` equal to the calculated value `mid` – 1.

stopping case

As with any recursive algorithm, we must ensure that our algorithm ends rather than producing infinite recursion. If the sought-after number is found on the list, then there is no recursive call and the process terminates, but we need some way to detect when the number is not on the list. On each recursive call, the value of `first` is increased or the value of `last` is decreased. If they ever pass each other and `first` actually becomes larger than `last`, then we will know that there are no more indices left to check and that the number `key` is not in the array. If we add this test to our pseudocode, we obtain a complete solution, as shown in Display 11.5.

algorithm final version

Now we can routinely translate the pseudocode into Java code. The result is shown in Display 11.6. The method `search` is an implementation of the recursive algorithm given in Display 11.5. A diagram of how the method performs on a sample array is given in Display 11.7. Display 11.8 illustrates how the method `search` is used.

Display 11.5 Pseudocode for Binary Search

ALGORITHM TO SEARCH a[first] THROUGH a[last]

```
/**
 Precondition:
 a[first]<= a[first + 1] <= a[first + 2] <=... <= a[last]
*/
```

TO LOCATE THE VALUE KEY:

```
if (first > last) //A stopping case
    return -1;
else
{
    mid = approximate midpoint between first and last;
    if (key == a[mid]) //A stopping case
        return mid;
    else if key < a[mid] //A case with recursion
        return the result of searching a[first] through a[mid - 1];
    else if key > a[mid] //A case with recursion
        return the result of searching a[mid + 1] through a[last];
 }
```

Display 11.6 Recursive Method for Binary Search

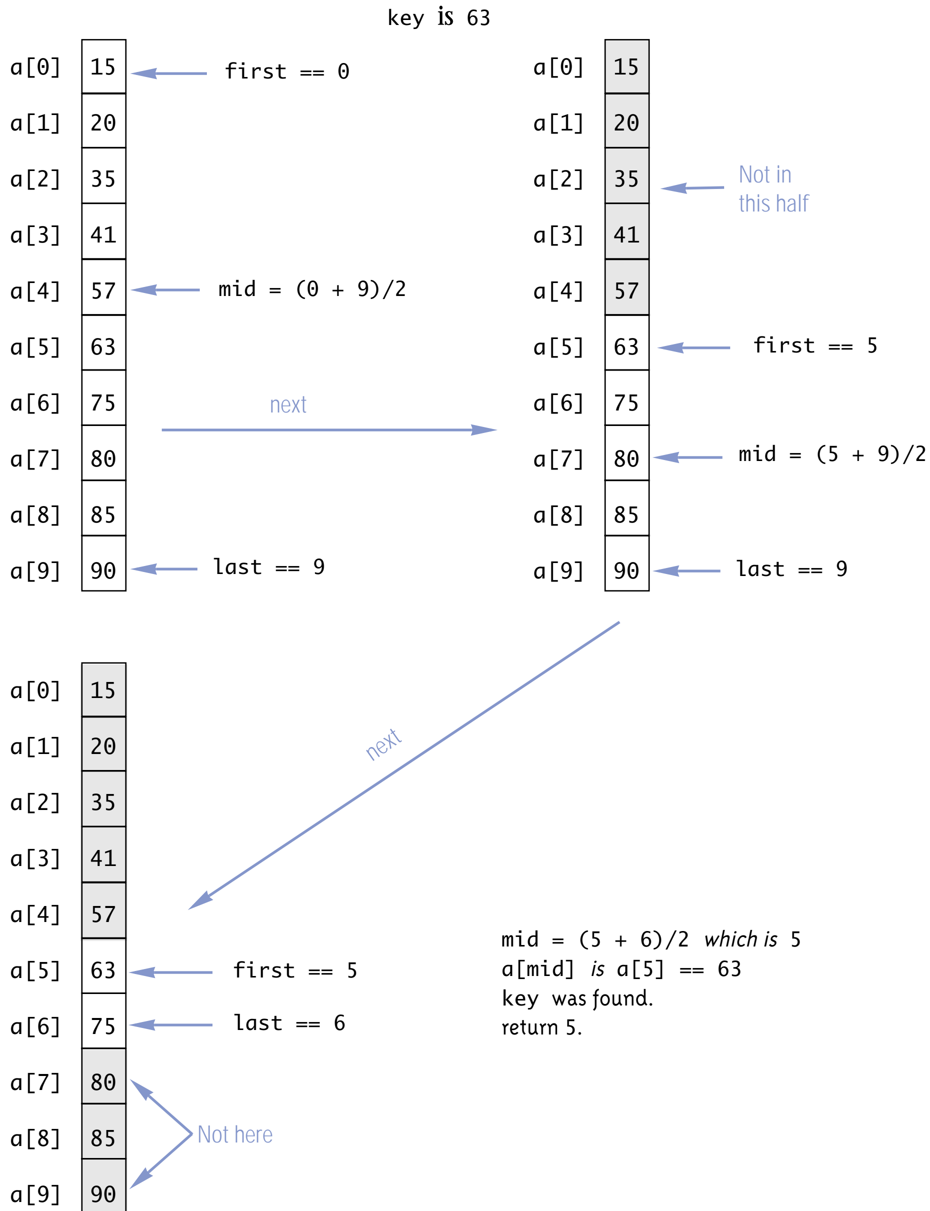
```
1   public class BinarySearch
2   {
3       /**
4        Searches the array a for key. If key is not in the array segment, then −1 is
5        returned. Otherwise returns an index in the segment such that key == a[index].
6        Precondition: a[first] <= a[first + 1]<= ... <= a[last]
7       */
8       public static int search(int[] a, int first, int last, int key)
9       {
10          int result = 0; //to keep the compiler happy.

11          if (first > last)
12              result = −1;
13          else
14          {
15              int mid = (first + last)/2;

16              if (key == a[mid])
17                  result = mid;
18              else if (key < a[mid])
19                  result = search(a, first, mid − 1, key);
20              else if (key > a[mid])
21                  result = search(a, mid + 1, last, key);
22          }
23          return result;
24      }
25  }
```

Notice that the method `search` solves a more general problem than the original task. Our goal was to design a method to search an entire array. Yet the method will let us search any interval of the array by specifying the indices `first` and `last`. This is common when designing recursive methods. Frequently, it is necessary to solve a more general problem in order to be able to express the recursive algorithm. In this case, we only wanted the answer in the case where `first` and `last` are set equal to `0` and `finalIndex`. However, the recursive calls will set them to values other than `0` and `finalIndex`.

In the subsection entitled "Tracing a Recursive Call," we gave three criteria that you should check to ensure that a recursive `void` method definition is correct. Let's check these three things for the method `search` given in Display 11.6:

1. **There is no infinite recursion:** On each recursive call the value of `first` is increased or the value of `last` is decreased. If the chain of recursive calls does not end in some other way, then eventually the method will be called with `first` larger than `last`, and that is a stopping case.

Display 11.7 **Execution of the Method search**



2. **Each stopping case performs the correct action for that case:** There are two stopping cases, when `first > last` and when `key == a[mid]`. Let's consider each case.

   If `first > last`, there are no array elements between `a[first]` and `a[last]` and so `key` is not in this segment of the array. (Nothing is in this segment of the array!) So,

Display 11.8 **Using the search Method**

```
1  public class BinarySearchDemo
2  {
3      public static void main(String[] args)
4      {
5          int[] a = {-2, 0, 2, 4, 6, 8, 10, 12, 14, 16};
6          int finalIndex = 9;

7          System.out.println("Array contains:");
8          for (int i = 0; i < a.length; i++)
9              System.out.print(a[i] + " ");
10         System.out.println();
11         System.out.println();

12         int result;
13         for (int key = -3; key < 5; key++)
14         {
15             result = BinarySearch.search(a, 0, finalIndex, key);
16             if (result >= 0)
17                 System.out.println(key + " is at index " + result);
18             else
19                 System.out.println(key + " is not in the array.");
20         }
21     }
22 }
```

Sample Dialogue

```
Array contains:
-2 0 2 4 6 8 10 12 14 16

-3 is not in the array.
-2 is at index 0
-1 is not in the array.
0 is at index 1
1 is not in the array.
2 is at index 2
3 is not in the array.
4 is at index 3
```

if `first > last`, the method `search` correctly returns −1, indicating that `key` is not in the specified range of the array.

If `key == a[mid]`, the algorithm correctly sets `location` equal to `mid`. Thus, both stopping cases are correct.

3. **For each of the cases that involve recursion, *if* all recursive calls perform their actions correctly, *then* the entire case performs correctly:** There are two cases in which there are recursive calls, when `key < a[mid]` and when `key > a[mid]`. We need to check each of these two cases.

   First suppose `key < a[mid]`. In this case, since the array is sorted, we know that if `key` is anywhere in the array, then `key` is one of the elements `a[first]` through `a[mid – 1]`. Thus, the method need only search these elements, which is exactly what the recursive call

   ```
   search(a, first, mid – 1, key)
   ```

   does. So if the recursive call is correct, then the entire action is correct.

   Next, suppose `key > a[mid]`. In this case, since the array is sorted, we know that if `key` is anywhere in the array, then `key` is one of the elements `a[mid + 1]` through `a[last]`. Thus, the method need only search these elements, which is exactly what the recursive call

   ```
   search(a, mid + 1, last, key)
   ```

   does. So if the recursive call is correct, then the entire action is correct. Thus, in both cases the method performs the correct action (assuming that the recursive calls perform the correct action).

The method `search` passes all three of our tests, so it is a good recursive method definition.

EFFICIENCY OF BINARY SEARCH ✜

The binary search algorithm is extremely fast compared to an algorithm that simply tries all array elements in order. In the binary search, you eliminate about half the array from consideration right at the start. You then eliminate a quarter, then an eighth of the array, and so forth. These savings add up to a dramatically fast algorithm. For an array of 100 elements, the binary search will never need to compare more than seven array elements to the key. A serial search could compare as many as 100 array elements to the key and on the average will compare about 50 array elements to the key. Moreover, the larger the array is, the more dramatic the savings will be. On an array with 1,000 elements, the binary search will only need to compare about 10 array elements to the key value, as compared to an average of 500 for the serial search algorithm.[1]

iterative version

An iterative version of the method `search` is given in Display 11.9. On some systems the iterative version will run more efficiently than the recursive version. The algorithm for the iterative version was derived by mirroring the recursive version. In the iterative version, the local variables `first` and `last` mirror the roles of the parameters in the recursive version, which are also named `first` and `last`. As this example illustrates, it often makes sense to derive a recursive algorithm even if you expect to later

---

[1] The binary search algorithm has worst-case running time that is logarithmic, that is $O(\log n)$. The serial search algorithm is linear, that is $O(n)$. If the terms used in this footnote are not familiar to you, you can safely ignore this footnote.

Display 11.9 Iterative Version of Binary Search

```
1   /**
2    Searches the array a for key. If key is not in the array segment, then −1 is
3    returned. Otherwise returns an index in the segment such that key == a[index].
4    Precondition: a[lowEnd] <= a[lowEnd + 1]<= ... <= a[highEnd]
5   */
6   public static int search(int[] a, int lowEnd, int highEnd, int key)
7   {
8       int first = lowEnd;
9       int last = highEnd;
10      int mid;

11      boolean found = false; //so far
12      int result = 0; //to keep compiler happy

13      while ( (first <= last) && !(found) )
14      {
15          mid = (first + last)/2;

16          if (key == a[mid])
17          {
18              found = true;
19              result = mid;
20          }
21          else if (key < a[mid])
22          {
23              last = mid − 1;
24          }
25          else if (key > a[mid])
26          {
27              first = mid + 1;
28          }
29      }

30      if (first > last)
31          result = −1;

32      return result;
33  }
```

extra code on CD

convert it to an iterative algorithm. You can see the iterative method from Display 11.9 embedded in a full demonstration in the files `IterativeBinarySearch.java` and `IterativeBinarySearchDemo.java` on the accompanying CD.

Most modern compilers will convert certain simple recursive method definitions to iterative ones before translating the code. In such cases there is no loss of efficiency for using the recursive version in place of an iterative version.

Self-Test Exercises

15. Write a recursive method definition for the following method:

```
/**
 Precondition: n >= 1
 Returns the sum of the squares of the numbers 1 through n.
*/
public static int squares(int n)
```

For example, `squares(3)` returns 14 because $1^2 + 2^2 + 3^2$ is 14.

Chapter Summary

- If a problem can be reduced to smaller instances of the same problem, then a recursive solution is likely to be easy to find and implement.
- A recursive algorithm for a method definition normally contains two kinds of cases: one or more cases that include at least one recursive call and one or more stopping cases in which the problem is solved without any recursive calls.
- When writing a recursive method definition, always check to see that the method will not produce infinite recursion.
- When you define a recursive method, use the three criteria given in the subsection "Recursive Design Techniques" to check that the method is correct.
- When you design a recursive method to solve a task, it is often necessary to solve a more general problem than the given task. This may be required to allow for the proper recursive calls, since the smaller problems may not be exactly the same problem as the given task. For example, in the binary search problem, the task was to search an entire array, but the recursive solution is an algorithm to search any portion of the array (either all of it or a part of it).

Answers to Self-Test Exercises

1. `Hip Hip Hurray`

2.
```
public static void stars(int n)
{
    System.out.print('*');
    if (n > 1)
```

```
        stars(n - 1);
}
```

The following is also correct, but is more complicated:

```
public static void stars(int n)
{
    if (n <= 1)
    {
        System.out.print('*');
    }
    else
    {
        stars(n - 1);
        System.out.print('*');
    }
}
```

3.
```
public static void backward(int n)
{
    if (n < 10)
    {
        System.out.print(n);
    }
    else
    {
        System.out.print(n%10);//write last digit
        backward(n/10);//write the other digits backward
    }
}
```

4.
```
public static void writeUp(int n)
{
    if (n >= 1)
    {
        writeUp(n - 1);
        System.out.print(n + " "); //write while the
                                   //recursion unwinds
    }
}
```

5.
```
public static void writeDown(int n)
{
    if (n >= 1)
    {
        System.out.print(n + " "); //write while the
                                   //recursion winds
        writeDown(n - 1);
    }
}
```

6. An error message that says *stack overflow* is telling you that the computer has attempted to place more stack frames on the stack than are allowed on your system. A likely cause of this error message is infinite recursion.

7.
```
public static void cheers(int n)
{
    while (n > 1)
    {
        System.out.print("Hip ");
        n--;
    }
    System.out.println("Hurray");
}
```

8.
```
public static void stars(int n)
{
    for (int count = 1; count <= n; count++)
        System.out.print('*');
}
```

9.
```
public static void backward(int n)
{
    while (n >= 10)
    {
        System.out.print(n%10);//write last digit
        n = n/10;//discard the last digit
    }
    System.out.print(n);
}
```

10. Trace for exercise 4: If `n = 3`, the code to be executed is

```
if (3 >= 1)
{
    writeUp(2);
    System.out.print(3 + " ");
}
```

The execution is suspended before the `System.out.println`. On the next recursion, `n = 2`; the code to be executed is

```
if (2 >= 1)
{
    writeUp(1);
    System.out.print(2 + " ");
}
```

The execution is suspended before the `System.out.println`. On the next recursion, `n = 1` and the code to be executed is

```
if (1 >= 1)
{
    writeUp(0);
    System.out.print(1 + " ");
}
```

The execution is suspended before the `System.out.println`. On the final recursion, `n = 0` and the code to be executed is

```
if (0 >= 1) // condition false, body skipped
{
    // skipped
}
```

The suspended computations are completed from the most recent to the least recent. The output is `1 2 3`.

11. Trace for exercise 5: If `n = 3`, the code to be executed is

```
    if (3 >= 1)
    {
        System.out.print(3 + " ");
        writeDown(2);
    }
```

Next recursion, `n = 2`, the code to be executed is

```
    if (2 >= 1)
    {
        System.out.print(2 + " ");
        writeDown(1)
    }
```

Next recursion, `n = 1`, the code to be executed is

```
    if (1 >= 1)
    {
        System.out.print(1 + " ");
        writeDown(0)
    }
```

Final recursion, `n = 0`, and the `if` statement does nothing, ending the recursive calls:

```
    if (0 >= 1) // condition false
    {
                // this clause is skipped
    }
```

The output is `3 2 1`.

12. `6`

13. The output is 24. The method rose is the factorial method, usually written $n!$ and defined as follows:

    $n!$ is equal to $n*(n-1)*(n-2)*...*1$

14.
```
public static double power(int x, int n)
{
    if (n < 0 && x == 0)
    {
        System.out.println(
                    "Illegal argument to power.");
        System.exit(0);
    }

    if (n < 0)
        return ( 1/power(x, -n));
    else if (n > 0)
        return ( power(x, n - 1)*x );
    else // n == 0
        return (1.0);
}
```

15.
```
public static int squares(int n)
{
    if (n <= 1)
        return 1;
    else
        return ( squares(n - 1) + n*n );
}
```

PROGRAMMING PROJECTS

1. Write a recursive method definition for a static method that has one parameter n of type int and that returns the nth Fibonacci number. The Fibonacci numbers are F_0 is 1, F_1 is 1, F_2 is 2, F_3 is 3, F_4 is 5, and in general

   $F_{i+2} = F_i + F_{i+1}$ for $i = 0, 1, 2, ...$

   Place the method in a class that has a main that tests the method.

2. The formula for computing the number of ways of choosing r different things from a set of n things is the following:

   $C(n, r) = n!/(r!*(n-r)!)$

The factorial method $n!$ is defined by

$$n! = n*(n-1)*(n-2)*...*1.$$

Discover a recursive version of the formula for $C(n, r)$ and write a recursive method that computes the value of the formula. Place the method in a class that has a `main` that tests the method.

3. *Towers of Hanoi.* There is a story about Buddhist monks who are playing this puzzle with 64 stone disks. The story claims that when the monks finish moving the disks from one post to a second via the third post, time will end.

   A stack of `n` disks of decreasing size (from bottom to top) is placed on one of three posts. The task is to move the disks one at a time from the first post to the second. To do this, any disk can be moved from any post to any other post, subject to the rule that you can never place a larger disk over a smaller disk. The (spare) third post is provided to make the solution possible. Your task is to write a recursive static method that gives instructions for a solution to this problem. We don't want to bother with graphics, so you should output a sequence of instructions that will solve the problem. The number of disks is a parameter to the method.

   Hint: If you could move up `n-1` of the disks from the first post to the third post using the second post as a spare, the last disk could be moved from the first post to the second post. Then, by using the same technique (whatever that may be), you can move the `n-1` disks from the third post to the second post, using the first disk as a spare. There! You have the puzzle solved. You only have to decide what the nonrecursive case is, what the recursive case is, and when to output instructions to move the disks.