14.1 Introduction to Recursion

**CONCEPT:** A recursive method is a method that calls itself.

You have seen instances of methods calling other methods. Method A can call method B, which can then call method C. It's also possible for a method to call itself. A method that calls itself is a *recursive method*. Look at the `message` method in Code Listing 14-1.

**Code Listing 14-1** (*EndlessRecursion.java*)

```java
/**
 * This class has a recursive method.
 */

public class EndlessRecursion {
    public static void message() {
        System.out.println("This is a recursive method.");
        message();
    }
}
```
This method displays the string "This is a recursive method.", and then calls itself. Each time it calls itself, the cycle is repeated. Can you see a problem with the method? There's no way to stop the recursive calls. This method is like an infinite loop because there is no code to stop it from repeating.

Like a loop, a recursive method must have some way to control the number of times it repeats. The class in Code Listing 14-2 has a modified version of the message method. It passes an integer argument, which holds the number of times the method should call itself.

**Code Listing 14-2**  
(Recursive.java)

```java
/**
 * This class has a recursive method, message, that displays
 * a message n times.
 */

public class Recursive
{
  public static void message(int n)
  {
    if (n > 0)
    {
      System.out.println("This is a recursive method.");
      message(n - 1);
    }
  }
}
```

This method contains an if statement that controls the repetition. As long as the n parameter is greater than zero, the method displays the message and calls itself again. Each time it calls itself, it passes n - 1 as the argument. For example, look at the program in Code Listing 14-3.

**Code Listing 14-3**  
(RecursionDemo.java)

```java
/**
 * This class demonstrates the Recursive.message method.
 */

public class RecursionDemo
{
  public static void main(String[] args)
  {
    Recursive.message(5);
  }
}
```
In line 9, the main method in this class calls the \texttt{Recursive.message} method with argument 5, which causes the method to call itself five times. The first time the method is called, the \texttt{if} statement displays the message and then calls itself with 4 as the argument. Figure 14-1 illustrates this.

\textbf{Figure 14-1} First two calls of the method

The diagram in Figure 14-1 illustrates two separate calls of the message method. Each time the method is called, a new instance of the \texttt{n} parameter is created in memory. The first time the method is called, the \texttt{n} parameter is set to 5. When the method calls itself, a new instance of \texttt{n} is created, and the value 4 is passed into it. This cycle repeats until, finally, zero is passed to the method. This is illustrated in Figure 14-2.

As you can see from Figure 14-2, the method is called a total of six times. The first time it is called from the \texttt{main} method of the \texttt{RecursionDemo} class, and the other five times it calls itself. The number of times that a method calls itself is known as the \textit{depth of recursion}. In this example, the depth of recursion is five. When the method reaches its sixth call, the \texttt{n} parameter is set to 0. At that point, the \texttt{if} statement's conditional expression is false, so the method returns. Control of the program returns from the sixth instance of the method to the point in the fifth instance directly after the recursive method call. This is illustrated in Figure 14-3.

Because there are no more statements to be executed after the method call, the fifth instance of the method returns control of the program back to the fourth instance. This repeats until all instances of the method return.

\textbf{Program Output}

This is a recursive method.
This is a recursive method.
This is a recursive method.
This is a recursive method.
This is a recursive method.
916 Chapter 14 Recursion

Figure 14-2 Total of six calls to the message method

The method is first called from the main method of the RecursionDemo class.

First call of the method
Value of n: 5

Second call of the method
Value of n: 4

Third call of the method
Value of n: 3

Fourth call of the method
Value of n: 2

Fifth call of the method
Value of n: 1

Sixth call of the method
Value of n: 0

Figure 14-3 Control returns to the point after the recursive method call

public static void message(int n)
{
    if (n > 0)
    {
        System.out.println("This is a recursive method.");
        message(n - 1);
    }
}

14.2 Solving Problems with Recursion

CONCEPT: A problem can be solved with recursion if it can be broken down into successive smaller problems that are identical to the overall problem.

The Recursive and RecursionDemo classes shown in the previous section demonstrate the mechanics of a recursive method. Recursion can be a powerful tool for solving repetitive problems and is an important topic in upper-level computer science courses. What might not be clear to you yet is how to use recursion to solve a problem.

First, it should be noted that recursion is never absolutely required to solve a problem. Any problem that can be solved recursively can also be solved iteratively, with a loop. In fact, recursive algorithms are usually less efficient than iterative algorithms. This is because a method call requires several actions to be performed by the JVM. These actions include allocating memory...
for parameters and local variables, and storing the address of the program location where control returns after the method terminates. These actions, which are sometimes referred to as overhead, take place with each method call. Such overhead is not necessary with a loop.

Some repetitive problems, however, are more easily solved with recursion than with iteration. Whereas an iterative algorithm might result in faster execution time, the programmer might be able to design a recursive algorithm faster.

In general, a recursive method works like this:

- If the problem can be solved now, without recursion, then the method solves it and returns.
- If the problem cannot be solved now, then the method reduces it to a smaller but similar problem and calls itself to solve the smaller problem.

In order to apply this approach, we first identify at least one case in which the problem can be solved without recursion. This is known as the base case. Second, we determine a way to solve the problem in all other circumstances using recursion. This is called the recursive case. In the recursive case, we must always reduce the problem to a smaller version of the original problem. By reducing the problem with each recursive call, the base case will eventually be reached and the recursion will stop.

Let's take an example from mathematics to examine an application of recursion. In mathematics, the notation \( n! \) represents the factorial of the number \( n \). The factorial of a non-negative number can be defined by the following rules:

\[
\begin{align*}
\text{If } n = 0 & \text{ then } n! = 1 \\
\text{If } n > 0 & \text{ then } n! = 1 \times 2 \times 3 \times \ldots \times n
\end{align*}
\]

Let's replace the notation \( n! \) with \text{factorial}(n), which looks a bit more like computer code, and rewrite these rules as:

\[
\begin{align*}
\text{If } n = 0 & \text{ then } \text{factorial}(n) = 1 \\
\text{If } n > 0 & \text{ then } \text{factorial}(n) = n \times \text{factorial}(n - 1)
\end{align*}
\]

These rules state that when \( n \) is 0, its factorial is 1. When \( n \) is greater than 0, its factorial is the product of all the positive integers from 1 up to \( n \). For instance, \text{factorial}(6) is calculated as \( 1 \times 2 \times 3 \times 4 \times 5 \times 6 \).

When designing a recursive algorithm to calculate the factorial of any number, we first identify the base case, which is the part of the calculation that we can solve without recursion. That is the case where \( n \) is equal to 0:

\[
\text{If } n = 0 \text{ then } \text{factorial}(n) = 1
\]

This tells how to solve the problem when \( n \) is equal to 0, but what do we do when \( n \) is greater than 0? That is the recursive case, or the part of the problem that we use recursion to solve. This is how we express it:

\[
\text{If } n > 0 \text{ then } \text{factorial}(n) = n \times \text{factorial}(n - 1)
\]

This states that if \( n \) is greater than 0, the factorial of \( n \) is \( n \) times the factorial of \( n - 1 \). Notice how the recursive call works on a reduced version of the problem, \( n - 1 \). So, our recursive rule for calculating the factorial of a number might look like this:

\[
\begin{align*}
\text{If } n = 0 & \text{ then } \text{factorial}(n) = 1 \\
\text{If } n > 0 & \text{ then } \text{factorial}(n) = n \times \text{factorial}(n - 1)
\end{align*}
\]
The following code shows how this might be implemented in a Java method.

```java
private static int factorial(int n) {
    if (n == 0) {
        return 1;  // Base case
    } else {
        return n * factorial(n - 1);
    }
}
```

The program in Code Listing 14-4 demonstrates the method.

**Code Listing 14-4** *(FactorialDemo.java)*

```java
1 import java.util.Scanner;
2
3 /**
4 * This program demonstrates the recursive factorial method.
5 */
6
7 public class FactorialDemo {
8 {
9     public static void main(String[] args) {
10         int number;  // To hold a number
11         // Create a Scanner object for keyboard input.
12         Scanner keyboard = new Scanner(System.in);
13         // Get a number from the user.
14         System.out.print("Enter a nonnegative integer: ");
15         number = keyboard.nextInt();
16         // Display the factorial.
17         System.out.println(number + "! is " + factorial(number));
18     }
19     }
20
21 /**
22 * Recursive factorial method. This method returns the
23 * factorial of its argument, which is assumed to be a
24 * nonnegative number.
25 */
26
27 private static int factorial(int n) {
28     
29 }
In the example run of the program, the factorial method is called with the argument 4 passed into \( n \). Because \( n \) is not equal to 0, the if statement’s else clause executes the statement in line 35. Although this is a return statement, it does not immediately return. Before the return value can be determined, the value of \( \text{factorial}(n - 1) \) must be determined. The factorial method is called recursively until the fifth call, in which the \( n \) parameter will be set to zero. The diagram in Figure 14-4 illustrates the value of \( n \) and the return value during each call of the method.

**Figure 14-4** Recursive calls to the factorial method
Chapter 14 Recursion

This diagram illustrates why a recursive algorithm must reduce the problem with each recursive call. Eventually the recursion has to stop in order for a solution to be reached. If each recursive call works on a smaller version of the problem, then the recursive calls work toward the base case. The base case does not require recursion, so it stops the chain of recursive calls.

Usually, a problem is reduced by making the value of one or more parameters smaller with each recursive call. In our factorial method, the value of the parameter \( n \) gets closer to 0 with each recursive call. When the parameter reaches 0, the method returns a value without making another recursive call.

**Direct and Indirect Recursion**

The examples we have discussed so far show recursive methods that directly call themselves. This is known as *direct recursion*. There is also the possibility of creating *indirect recursion* in a program. This occurs when method \( A \) calls method \( B \), which in turn calls method \( A \). There can even be several methods involved in the recursion. For example, method \( A \) could call method \( B \), which could call method \( C \), which calls method \( A \).

**Checkpoint**

14.1 It is said that a recursive algorithm has more overhead than an iterative algorithm. What does this mean?
14.2 What is a base case?
14.3 What is a recursive case?
14.4 What causes a recursive algorithm to stop calling itself?
14.5 What is direct recursion? What is indirect recursion?

**14.3 Examples of Recursive Methods**

**Summing a Range of Array Elements with Recursion**

In this example we look at a method, \( \text{rangeSum} \), that uses recursion to sum a range of array elements. The method takes the following arguments: an \( \text{int} \) array that contains the range of elements to be summed, an \( \text{int} \) specifying the starting element of the range, and an \( \text{int} \) specifying the ending element of the range. Here is an example of how the method might be used:

```java
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
int sum;
sum = rangeSum(numbers, 3, 7);
```
This code specifies that `rangeSum` should return the sum of elements three through seven in the `numbers` array. The return value, which in this case would be 30, is stored in `sum`. Here is the definition of the `rangeSum` method:

```java
public static int rangeSum(int[] array, int start, int end)
{
    if (start > end)
        return 0;
    else
        return array[start] + rangeSum(array, start + 1, end);
}
```

This method's base case is when the `start` parameter is greater than the `end` parameter. If this is true, the method returns the value 0. Otherwise, the method executes the following statement:

```java
return array[start] + rangeSum(array, start + 1, end);
```

This statement returns the sum of `array[start]` plus the return value of a recursive call. Notice that in the recursive call, the starting element in the range is `start + 1`. In essence, this statement says "return the value of the first element in the range plus the sum of the rest of the elements in the range." The program in Code Listing 14-5 demonstrates the method.

**Code Listing 14-5** (RangeSum.java)

```java
1 /*
2 * This program demonstrates the recursive rangeSum method.
3 */
4
5 public class RangeSum
6 {
7    /**
8     * main method
9     */
10
11    public static void main(String[] args)
12    {
13        int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
14        System.out.print("The sum of elements 2 through 5 is "+
15                          rangeSum(numbers, 2, 5));
16    }
17
18    /**
19     * The rangeSum method returns the sum of a specified
20      * range of elements in array. The start parameter
21      * specifies the starting element and the end parameter
22     * specifies the ending parameter.
23     */
24
25```
26    public static int rangeSum(int[] array, int start, int end)  
27    {
28        if (start > end)  
29            return 0;
30        else
31            return array[start] + rangeSum(array, start + 1, end);  
32    }

Program Output  
The sum of elements 2 through 5 is 18

**Drawing Concentric Circles**  
In this example we look at the Circles applet, which uses recursion to draw concentric circles. Concentric circles are circles of different sizes, one inside another, all with a common center point. Figure 14-5 shows the applet's output. The applet code is shown in Code Listing 14-6.

**Figure 14-5**  Circles applet
import javax.swing.*;
import java.awt.*;

/**
 * This applet uses a recursive method to draw
 * concentric circles.
 */

public class Circles extends JApplet
{
    /**
     * init method
     */

    public void init()
    {
        // Set the background color to white.
        setBackground(Color.WHITE);
    }

    /**
     * paint method
     */

    public void paint(Graphics g)
    {
        // Draw 10 concentric circles. The outermost circle's
        // enclosing rectangle should be at (5, 5), and it
        // should be 300 pixels wide by 300 pixels high.
        drawCircles(g, 10, 5, 300);
    }

    /**
     * The drawCircles method draws concentric circles.
     * It accepts the following arguments:
     * g, a Graphics object
     * n, the number of circles to draw
     * topXY, the top-left coordinates of the
     * outermost circle's enclosing rectangle
     * size, the width and height of the outermost
     * circle's enclosing rectangle
     */
private void drawCircles(Graphics g, int n, int topXY, int size)
{
    if (n > 0)
    {
        g.drawOval(topXY, topXY, size, size);
        drawCircles(g, n - 1, topXY + 15, size - 30);
    }
}

The drawCircles method, which is called from the applet's paint method, uses recursion to draw the concentric circles. The n parameter holds the number of circles to draw. If this parameter is set to 0, the method has reached its base case. Otherwise, it calls the g object's drawOval method to draw a circle. The topXY parameter holds the value to use as the X and Y coordinate of the enclosing rectangle's upper-left corner. The size parameter holds the value to use as the enclosing rectangle's width and height. After the circle is drawn, the drawCircles method is recursively called with parameter values adjusted for the next circle.

The Fibonacci Series

Some mathematical problems are designed to be solved recursively. One well-known example is the calculation of Fibonacci numbers. The Fibonacci numbers, named after the Italian mathematician Leonardo Fibonacci (born circa 1170), are the following sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

Notice that after the second number, each number in the series is the sum of the two previous numbers. The Fibonacci series can be defined as:

If \( n = 0 \) then \( \text{Fib}(n) = 0 \)
If \( n = 1 \) then \( \text{Fib}(n) = 1 \)
If \( n \geq 2 \) then \( \text{Fib}(n) = \text{Fib}(n - 1) + \text{Fib}(n - 2) \)

A recursive Java method to calculate the \( n \)th number in the Fibonacci series is shown here:

public static int fib(int n)
{
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}

Notice that this method actually has two base cases: when \( n \) is less than 0, and when \( n \) is equal to 1. In either case, the method returns a value without making a recursive call. The program in Code Listing 14-7 demonstrates this method by displaying the first 10 numbers in the Fibonacci series.
14.3 Examples of Recursive Methods

Code Listing 14-7  (FibNumbers.java)

```java
public class FibNumbers {
    public static void main(String[] args) {
        System.out.println("The first 10 numbers in the Fibonacci series are:");
        for (int i = 0; i < 10; i++)
            System.out.print(fib(i) + " ");
        System.out.println();
    }
    public static int fib(int n) {
        if (n == 0)
            return 0;
        else if (n == 1)
            return 1;
        else
            return fib(n - 1) + fib(n - 2);
    }
}
```

Program Output
The first 10 numbers in the Fibonacci series are:
0 1 1 2 3 5 8 13 21 34
Finding the Greatest Common Divisor

Our next example of recursion is the calculation of the greatest common divisor, or GCD, of two numbers. The GCD of two positive integers, \(x\) and \(y\), is:

- if \(y\) divides \(x\) evenly, then \(\text{gcd}(x, y) = y\)
- Otherwise, \(\text{gcd}(x, y) = \text{gcd}(y, \text{remainder of } x/y)\)

This definition states that the GCD of \(x\) and \(y\) is \(y\) if \(x/y\) has no remainder. This is the base case. Otherwise, the answer is the GCD of \(y\) and the remainder of \(x/y\). The program in Code Listing 14-8 shows a recursive method for calculating the GCD.

**Code Listing 14-8  (GCDdemo.java)**

```java
1 import java.util.Scanner;
2
3 /**
4 * This program demonstrates the recursive gcd method.
5 */
6
7 public class GCDdemo
8 {
9     /**
10      * main method
11     */
12
13     public static void main(String[] args)
14     {
15         int num1, num2; // Two numbers
16
17         // Create a Scanner object for keyboard input.
18         Scanner keyboard = new Scanner(System.in);
19
20         // Get two numbers from the user.
21         System.out.print("Enter an integer: ");
22         num1 = keyboard.nextInt();
23         System.out.print("Enter another integer: ");
24         num2 = keyboard.nextInt();
25
26         // Display the GCD.
27         System.out.println("The greatest common divisor " +
28                 "of these two numbers is " +
29                 gcd(num1, num2));
30     }
31
32     /**
33      * The gcd method returns the greatest common divisor
34      * of the arguments passed into x and y.
35     */
36```

```
A Recursive Binary Search Method

CONCEPT: The recursive binary search algorithm is more elegant and easier to understand than its iterative version.

In Chapter 7 you learned about the binary search algorithm and saw an iterative example written in Java. The binary search algorithm can also be implemented recursively. For example, the procedure can be expressed as:

- If \( \text{array}[\text{middle}] \) equals the search value, then the value is found.
- Else if \( \text{array}[\text{middle}] \) is less than the search value, perform a binary search on the upper half of the array.
- Else if \( \text{array}[\text{middle}] \) is greater than the search value, perform a binary search on the lower half of the array.

When you compare the recursive algorithm to its iterative counterpart, it becomes evident that the recursive version is much more elegant and easier to understand. The recursive binary search algorithm is also a good example of repeatedly breaking a problem down into smaller pieces until it is solved. Here is the code for the method:

```java
public static int binarySearch(int[] array, int first, int last, int value)
{
    int middle; // Mid point of search

    // Test for the base case where the value is not found.
    if (first > last)
        return -1;

    // Calculate the middle position.
    middle = (first + last) / 2;
```
Chapter 14 Recursion

II

Search for the value.

if (array[middle] == value)
    return middle;
else if (array[middle] < value)
    return binarySearch(array, middle + 1, last, value);
else
    return binarySearch(array, first, middle - 1, value);
}

The first parameter, array, is the array to be searched. The next parameter, first, holds the subscript of the first element in the search range (the portion of the array to be searched). The next parameter, last, holds the subscript of the last element in the search range. The last parameter, value, holds the value to be searched for. Like the iterative version, this method returns the subscript of the value if it is found, or -1 if the value is not found. Code Listing 14-9 demonstrates the method.

Code Listing 14-9 (RecursiveBinarySearch.java)

```java
import java.util.Scanner;

public class RecursiveBinarySearch {
    public static void main(String [] args) {
        // The values in the following array are sorted
        // in ascending order.
        int numbers[] = {101, 142, 147, 189, 199, 207, 222,
                        234, 289, 296, 310, 319, 388, 394,
                        417, 429, 447, 521, 536, 600};

        int result;   // Result of the search
        int searchValue;  // Value to search for
        String again;    // User input

        // Create a Scanner object for keyboard input.
        Scanner keyboard = new Scanner(System.in);

        do
            { // Get a value to search for.
                System.out.print("Enter a value to search for: ");
                searchValue = keyboard.nextInt();

                // Search for the value
                result = binarySearch(numbers, 0,
                                       (numbers.length - 1), searchValue);
        
```
14.4 A Recursive Binary Search Method

```java
// Display the results.
if (result == -1)
{
    System.out.println(searchValue +
        " was not found.");
}
else
{
    System.out.println(searchValue +
        " was found at " +
        "element " + result);
}

// Consume the remaining newline.
keyboard.nextLine();

// Does the user want to search again?
System.out.print("Do you want to search again? " +
        "(Y or N): ");
again = keyboard.nextLine();
}
while (again.charAt(0) == 'y' || again.charAt(0) == 'Y');

/**
 * The binarySearch method performs a binary search on an
 * integer array. The array is searched for the number passed
 * to value. If the number is found, its array subscript is
 * returned. Otherwise, -1 is returned indicating the value was
 * not found in the array.
 */

public static int binarySearch(int[] array, int first,
        int last, int value)
{
    int middle;  // Mid-point of search

    // Test for the base case where the value is not found.
    if (first > last)
        return -1;

    // Calculate the middle position.
    middle = (first + last) / 2;

    // Search for the value.
    if (array[middle] == value)
        return middle;
```
else if (array[middle] < value)
    return binarySearch(array, middle + 1, last, value);
else
    return binarySearch(array, first, middle - 1, value);
}

---

Program Output with Example Input Shown in Bold

Enter a value to search for: 289 [Enter]
289 was found at element 8
Do you want to search again? (Y or N): y [Enter]
Enter a value to search for: 388 [Enter]
388 was found at element 12
Do you want to search again? (Y or N): y [Enter]
Enter a value to search for: 101 [Enter]
101 was found at element 0
Do you want to search again? (Y or N): y [Enter]
Enter a value to search for: 999 [Enter]
999 was not found.
Do you want to search again? (Y or N): n [Enter]

See Appendix K on the Student CD for a discussion of the recursive QuickSort algorithm.

14.5 The Towers of Hanoi

CONCEPT: The repetitive steps involved in solving the Towers of Hanoi game can be easily implemented in a recursive algorithm.

The Towers of Hanoi is a mathematical game that is often used in computer science textbooks to illustrate the power of recursion. The game uses three pegs and a set of discs with holes through their centers. The discs are stacked on one of the pegs as shown in Figure 14-6.

Figure 14-6 The pegs and discs in the Towers of Hanoi game

Notice that the discs are stacked on the leftmost peg, in order of size with the largest disc at the bottom. The game is based on a legend where a group of monks in a temple in Hanoi have a similar set of pegs with 64 discs. The job of the monks is to move the discs from the
first peg to the third peg. The middle peg can be used as a temporary holder. Furthermore, the monks must follow these rules while moving the discs:

- Only one disc can be moved at a time.
- A disc cannot be placed on top of a smaller disc.
- All discs must be stored on a peg except while being moved.

According to the legend, when the monks have moved all of the discs from the first peg to the last peg, the world will come to an end.

To play the game, you must move all of the discs from the first peg to the third peg, following the same rules as the monks. Let's look at some example solutions to this game, for different numbers of discs. If you have only one disc, the solution to the game is simple: Move the disc from peg 1 to peg 3. If you have two discs, the solution requires three moves:

- Move disc 1 to peg 2.
- Move disc 2 to peg 3.
- Move disc 1 to peg 3.

Notice that this approach uses peg 2 as a temporary location. The complexity of the moves continues to increase as the number of discs increase. To move three discs requires the seven moves shown in Figure 14-7.

**Figure 14-7** Steps for moving three pegs
The following statement describes the overall solution to the problem:

Move n discs from peg 1 to peg 3 using peg 2 as a temporary peg.

The following algorithm can be used as the basis of a recursive method that simulates the solution to the game. Notice that in this algorithm we use the variables A, B, and C to hold peg numbers.

To move n discs from peg A to peg C, using peg B as a temporary peg:

1. If n > 0 Then
   a. Move n - 1 discs from peg A to peg B, using peg C as a temporary peg.
   b. Move the remaining disc from peg A to peg C.
   c. Move n - 1 discs from peg B to peg C, using peg A as a temporary peg.
2. End If

The base case for the algorithm is reached when there are no more discs to move. The following code is for a method that implements this algorithm. Note that the method does not actually move anything, but displays instructions indicating all of the disc moves to make.

private void moveDiscs(int num, int fromPeg, int toPeg, int tempPeg)
{
    if (num > 0)
    {
        moveDiscs(num - 1, fromPeg, tempPeg, toPeg);
        System.out.println("Move a disc from peg " + fromPeg + " to peg " + toPeg);
        moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
    }
}

This method accepts arguments into the following four parameters:

- num: The number of discs to move.
- fromPeg: The peg to move the discs from.
- toPeg: The peg to move the discs to.
- tempPeg: The peg to use as a temporary peg.

If num is greater than 0, then there are discs to move. The first recursive call is:

moveDiscs(num - 1, fromPeg, tempPeg, toPeg);

This statement is an instruction to move all but one disc from fromPeg to tempPeg, using toPeg as a temporary peg. The next statement is:

System.out.println("Move a disc from peg " + fromPeg + " to peg " + toPeg);

This simply displays a message indicating that a disc should be moved from fromPeg to toPeg. Next, another recursive call is executed:

moveDiscs(num - 1, tempPeg, toPeg, fromPeg);

This statement is an instruction to move all but one disc from tempPeg to toPeg, using fromPeg as a temporary peg. Code Listing 14-10 shows the Hanoi class, which uses this method.
**Code Listing 14-10** (Hanoi.java)

```java
/**
 * This class displays a solution to the Towers of
 * Hanoi game.
 */

public class Hanoi {
    private int numDiscs; // Number of discs

    /**
     * Constructor. The argument is the number of
     * discs to use.
     */
    public Hanoi(int n) {
        // Assign the number of discs.
        numDiscs = n;

        // Move the number of discs from peg 1 to peg 3
        // using peg 2 as a temporary storage location.
        moveDiscs(numDiscs, 1, 3, 2);
    }

    /**
     * The moveDiscs method accepts the number of
     * discs to move, the peg to move from, the peg
     * to move to, and the temporary peg as arguments.
     * It uses recursion to display the necessary
     * disc moves.
     */
    private void moveDiscs(int num, int fromPeg,
                           int toPeg, int tempPeg) {
        if (num > 0) {
            moveDiscs(num - 1, fromPeg, tempPeg, toPeg);
            System.out.println("Move a disc from peg " +
                               fromPeg + " to peg " + toPeg);
            moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
        }
    }
}
```

14.5 The Towers of Hanoi
The class constructor accepts an argument that is the number of discs to use in the game. It assigns this value to the numDiscs field, and then calls the moveDiscs method in line 22. In a nutshell, this statement is an instruction to move all the discs from peg 1 to peg 3, using peg 2 as a temporary peg. The program in Code Listing 14-11 demonstrates the class. It displays the instructions for moving three discs.

```java
/*
 * This class demonstrates the Hanoi class, which
 * displays the steps necessary to solve the Towers
 * of Hanoi game.
 */

public class HanoiDemo {
    public static void main(String[] args) {
        Hanoi towersOfHanoi = new Hanoi(3);
    }
}
```

Program Output
Move a disc from peg 1 to peg 3
Move a disc from peg 1 to peg 2
Move a disc from peg 3 to peg 2
Move a disc from peg 1 to peg 3
Move a disc from peg 2 to peg 1
Move a disc from peg 2 to peg 3
Move a disc from peg 1 to peg 3

### Common Errors to Avoid

The following list describes several errors that are commonly made when learning this chapter's topics.

- **Not coding a base case.** When the base case is reached, a recursive method stops calling itself. Without a base case, the method will continue to call itself infinitely.
- **Not reducing the problem with each recursive call.** Unless the problem is reduced (which usually means that the value of one or more critical parameters is reduced) with each recursive call, the method will not reach the base case. If the base case is not reached, the method will call itself infinitely.
- **Writing the recursive call in such a way that the base case is never reached.** You might have a base case and a recursive case that reduces the problem, but if the calculations are not performed in such a way that the base case is ultimately reached, the method will call itself infinitely.
Review Questions and Exercises

Multiple Choice and True/False

1. A method is called once from a program's main method, and then it calls itself four times. The depth of recursion is
   a. one
   b. four
   c. five
   d. nine

2. This is the part of a problem that can be solved without recursion.
   a. base case
   b. solvable case
   c. known case
   d. iterative case

3. This is the part of a problem that is solved with recursion.
   a. base case
   b. iterative case
   c. unknown case
   d. recursion case

4. This is when a method explicitly calls itself.
   a. explicit recursion
   b. modal recursion
   c. direct recursion
   d. indirect recursion

5. This is when method A calls method B, which calls method A.
   a. implicit recursion
   b. modal recursion
   c. direct recursion
   d. indirect recursion

6. This refers to the actions taken internally by the JVM when a method is called.
   a. overhead
   b. set up
   c. clean up
   d. synchronization

7. True or False: An iterative algorithm will usually run faster than an equivalent recursive algorithm.

8. True or False: Some problems can be solved only through recursion.

9. True or False: It is not necessary to have a base case in all recursive algorithms.

10. True or False: In the base case, a recursive method calls itself with a smaller version of the original problem.
Find the Error

1. Find the error in the following program.

   ```java
   public class FindTheError
   {
   public static void main(String[] args)
   {
   myMethod(0);
   }

   public static void myMethod(int num)
   {
   System.out.print(num + " ");
   myMethod(num + 1);
   }
   }
   ```

Algorithm Workbench

1. Write a method that accepts a `String` as an argument. The method should use recursion to display each individual character in the `String`.

2. Modify the method you wrote in Question 1 so it displays the `String` backwards.

3. What will the following program display?

   ```java
   public class Checkpoint
   {
   public static void main(String[] args)
   {
   int num = 0;
   showMe(num);
   }

   public static void showMe(int arg)
   {
   if (arg < 10)
   showMe(arg + 1);
   else
   System.out.println(arg);
   }
   }
   ```

4. What will the following program display?

   ```java
   public class ReviewQuestion4
   {
   public static void main(String[] args)
   {
   int num = 0;
   showMe(num);
   }
   ```
public static void showMe(int arg)
{
    System.out.println(arg);
    if (arg <= 10)
        showMe(arg + 1);
}

5. What will the following program display?
public class ReviewQuestion5
{
    public static void main(String[] args)
    {
        int x = 10;
        System.out.println(myMethod(x));
    }

    public static int myMethod(int num)
    {
        if (num <= 0)
            return 0;
        else
            return myMethod(num - 1) + num;
    }
}

6. Convert the following iterative method to one that uses recursion.
public static void sign(int n)
{
    while (n > 0)
    {
        System.out.println("No Parking");
        n--;
    }
}

7. Write an iterative version (using a loop instead of recursion) of the factorial method shown in this chapter.

Short Answer
1. What is the difference between an iterative algorithm and a recursive algorithm?
2. What is a recursive algorithm's base case? What is the recursive case?
3. What is the base case of each of the recursive methods listed in Algorithm Workbench Questions 3, 4, and 5?
4. What type of recursive method do you think would be more difficult to debug: one that uses direct recursion or one that uses indirect recursion? Why?

5. Which repetition approach is less efficient: a loop or a recursive method? Why?

6. When recursion is used to solve a problem, why must the recursive method call itself to solve a smaller version of the original problem?

7. How is a problem usually reduced with a recursive method?

**Programming Challenges**

1. **Recursive Multiplication**
   Write a recursive function that accepts two arguments into the parameters \( x \) and \( y \). The function should return the value of \( x \) times \( y \). Remember, multiplication can be performed as repeated addition:
   \[
   7 \times 4 = 4 + 4 + 4 + 4 + 4 + 4 + 4
   \]

2. **isMember Method**
   Write a recursive boolean method named `isMember`. The method should accept two arguments: an array and a value. The method should return `true` if the value is found in the array, or `false` if the value is not found in the array. Demonstrate the method in a program.

3. **String Reverser**
   Write a recursive method that accepts a string as its argument and prints the string in reverse order. Demonstrate the method in a program.

4. **maxElement Method**
   Write a method named `maxElement` that returns the largest value in an array that is passed as an argument. The method should use recursion to find the largest element. Demonstrate the method in a program.

5. **Palindrome Detector**
   A palindrome is any word, phrase, or sentence that reads the same forward and backwards. Here are some well-known palindromes:
   - Able was I, ere I saw Elba
   - A man, a plan, a canal, Panama
   - Desserts, I stressed
   - Kayak
   Write a boolean method that uses recursion to determine whether a `String` argument is a palindrome. The method should return `true` if the argument reads the same forward and backwards. Demonstrate the method in a program.

6. **Character Counter**
   Write a method that uses recursion to count the number of times a specific character occurs in an array of characters. Demonstrate the method in a program.
7. Recursive Power Method
Write a method that uses recursion to raise a number to a power. The method should accept two arguments: the number to be raised and the exponent. Assume that the exponent is a nonnegative integer. Demonstrate the method in a program.

8. Sum of Numbers
Write a method that accepts an integer argument and returns the sum of all the integers from 1 up to the number passed as an argument. For example, if 50 is passed as an argument, the method will return the sum of 1, 2, 3, 4, ..., 50. Use recursion to calculate the sum. Demonstrate the method in a program.

9. Ackermann's Function
Ackermann's function is a recursive mathematical algorithm that can be used to test how well a computer performs recursion. Write a method `ackermann(m, n)` that solves Ackermann's function. Use the following logic in your method:

- If \( m = 0 \) then return \( n + 1 \)
- If \( n = 0 \) then return `ackermann(m - 1, 1)`
- Otherwise, return `ackermann(m - 1, ackermann(m, n - 1))`

Test your method in a program that displays the return values of the following method calls:

- `ackermann(0, 0)`
- `ackermann(0, 1)`
- `ackermann(1, 1)`
- `ackermann(1, 2)`
- `ackermann(1, 3)`
- `ackermann(2, 2)`
- `ackermann(3, 2)`

10. Recursive Population Class
In Programming Challenge 6 of Chapter 5 you wrote a population class that predicts the size of a population of organisms after a number of days. Modify the class so it uses a recursive method instead of a loop to calculate the number of organisms.