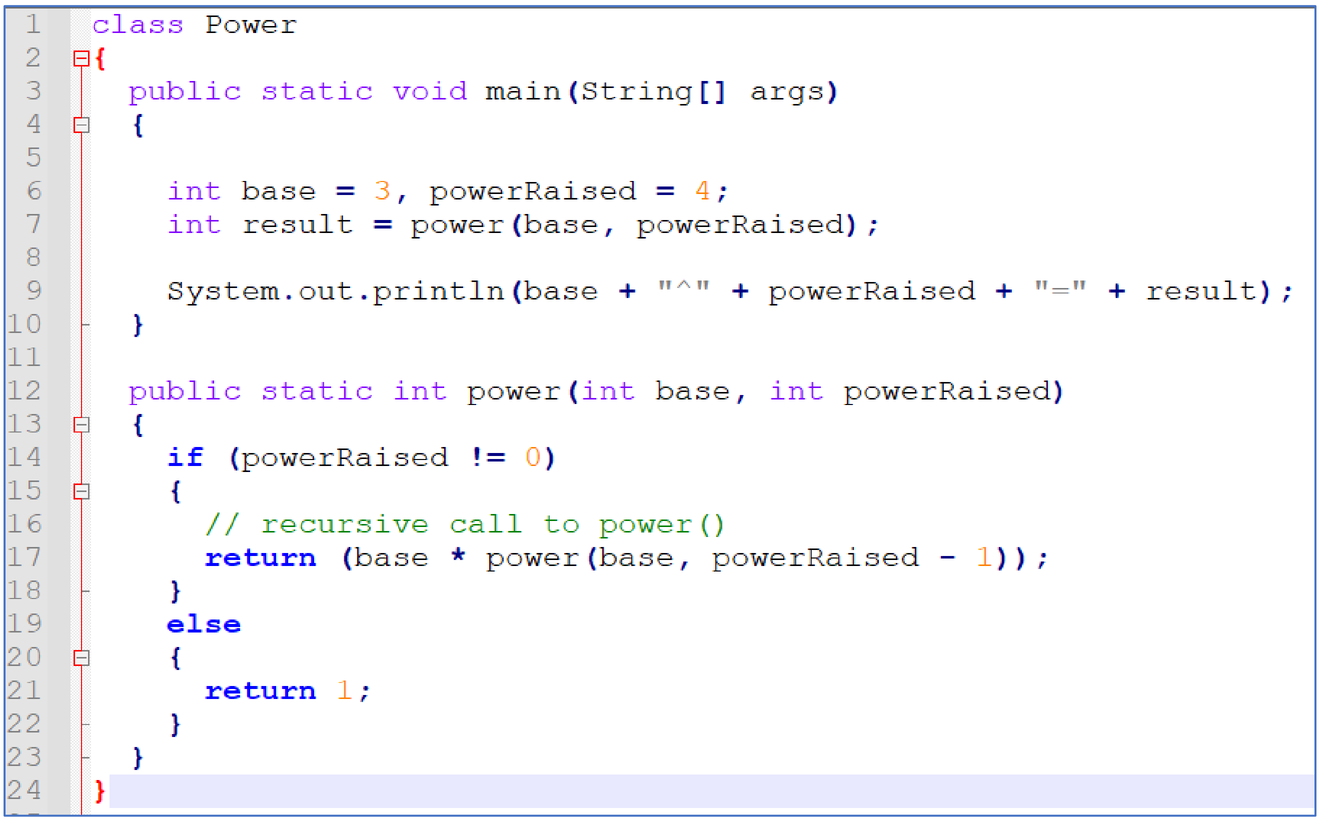
Chapter 18 Recursion

- Write a method for computing x^y doing repetitive multiplication. X and y are of type integer and are to be given as command line arguments. Raise and handle exception(s) for invalid values of x and y. (07).



```java
class Power
{
  public static void main(String[] args)
  {

    int base = 3, powerRaised = 4;
    int result = power(base, powerRaised);

    System.out.println(base + "^" + powerRaised + "=" + result);
  }

  public static int power(int base, int powerRaised)
  {
    if (powerRaised != 0)
    {
      // recursive call to power()
      return (base * power(base, powerRaised - 1));
    }
    else
    {
      return 1;
    }
  }
}
```

Recursion is a technique that leads to elegant solutions to problems that are difficult to program using simple loops.

search word problem

Suppose you want to find all the files under a directory that contain a particular word. How do you solve this problem? There are several ways to do so. An intuitive and effective solution is to use recursion by searching the files in the subdirectories recursively.

H-tree problem

H-trees, depicted in Figure 18.1, are used in a very large-scale integration (VLSI) design as a clock distribution network for routing timing signals to all parts of a chip with equal propagation delays. How do you write a program to display H-trees? A good approach is to use recursion.

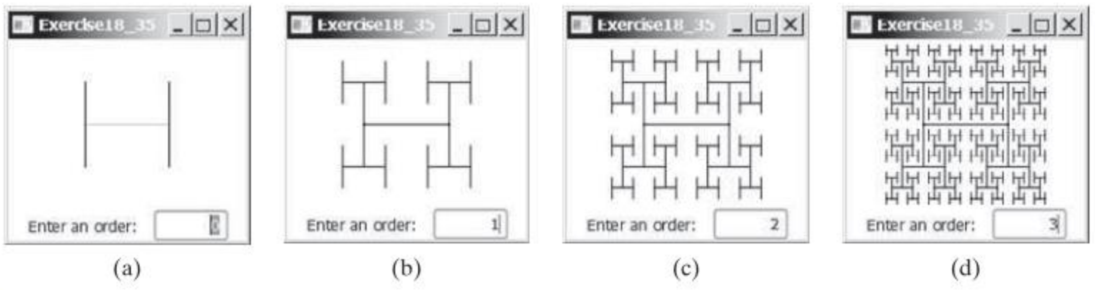


FIGURE 18.1 An H-tree can be displayed using recursion.

To use recursion is to program using recursive methods—that is, to use methods that invoke themselves.

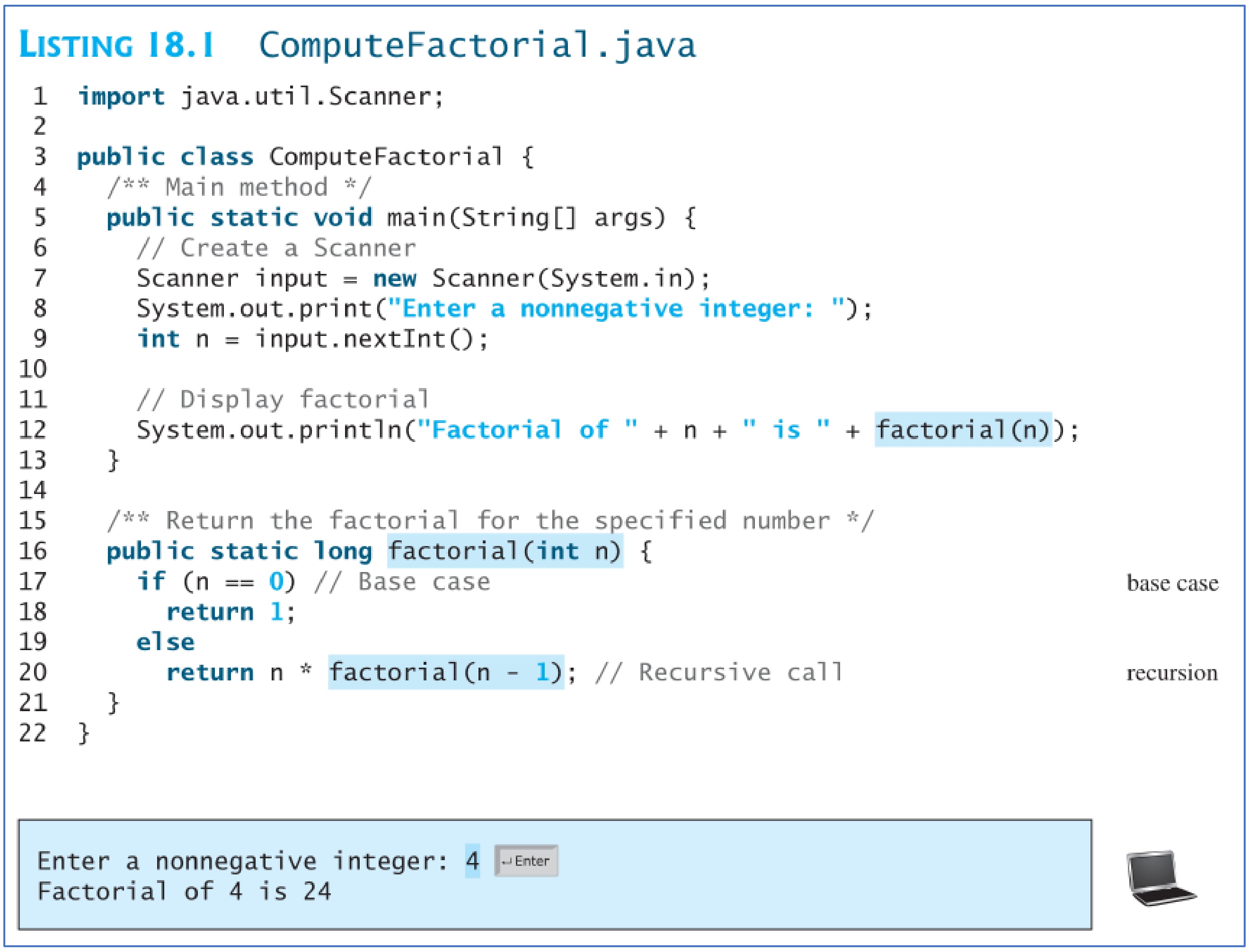
A recursive method is one that invokes itself.

base case or stopping condition

The method knows how to solve the simplest case, which is referred to as the base case or the stopping condition.

The subproblem is essentially the same as the original problem, but it is simpler or smaller.

Because the sub-problem has the same property as the original problem, you can call the method with a different argument, which is referred to as a recursive call.



LISTING 18.1 ComputeFactorial.java

```java
import java.util.Scanner;

public class ComputeFactorial {
  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a nonnegative integer: ");
    int n = input.nextInt();

    // Display factorial
    System.out.println("Factorial of " + n + " is " + factorial(n));
  }

  /** Return the factorial for the specified number */
  public static long factorial(int n) {
    if (n == 0) // Base case
      return 1;
    else
      return n * factorial(n - 1); // Recursive call
  }
}
```

base case

recursion

```
Enter a nonnegative integer: 4 ↵Enter
Factorial of 4 is 24
```

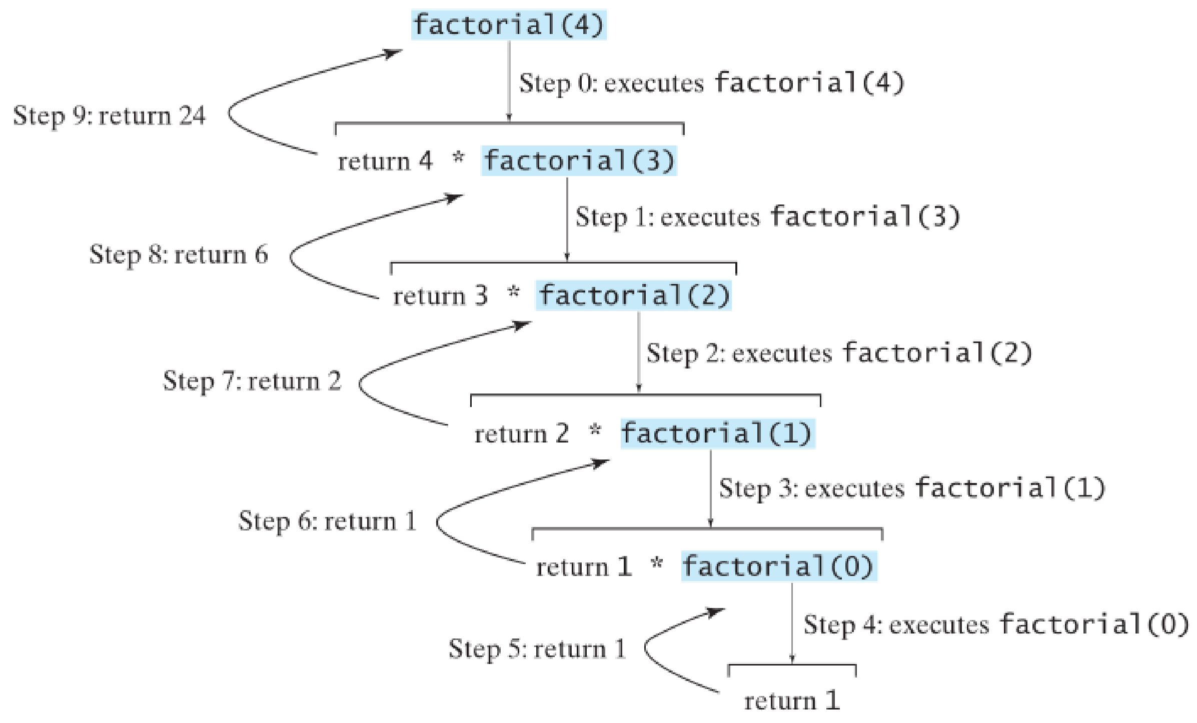
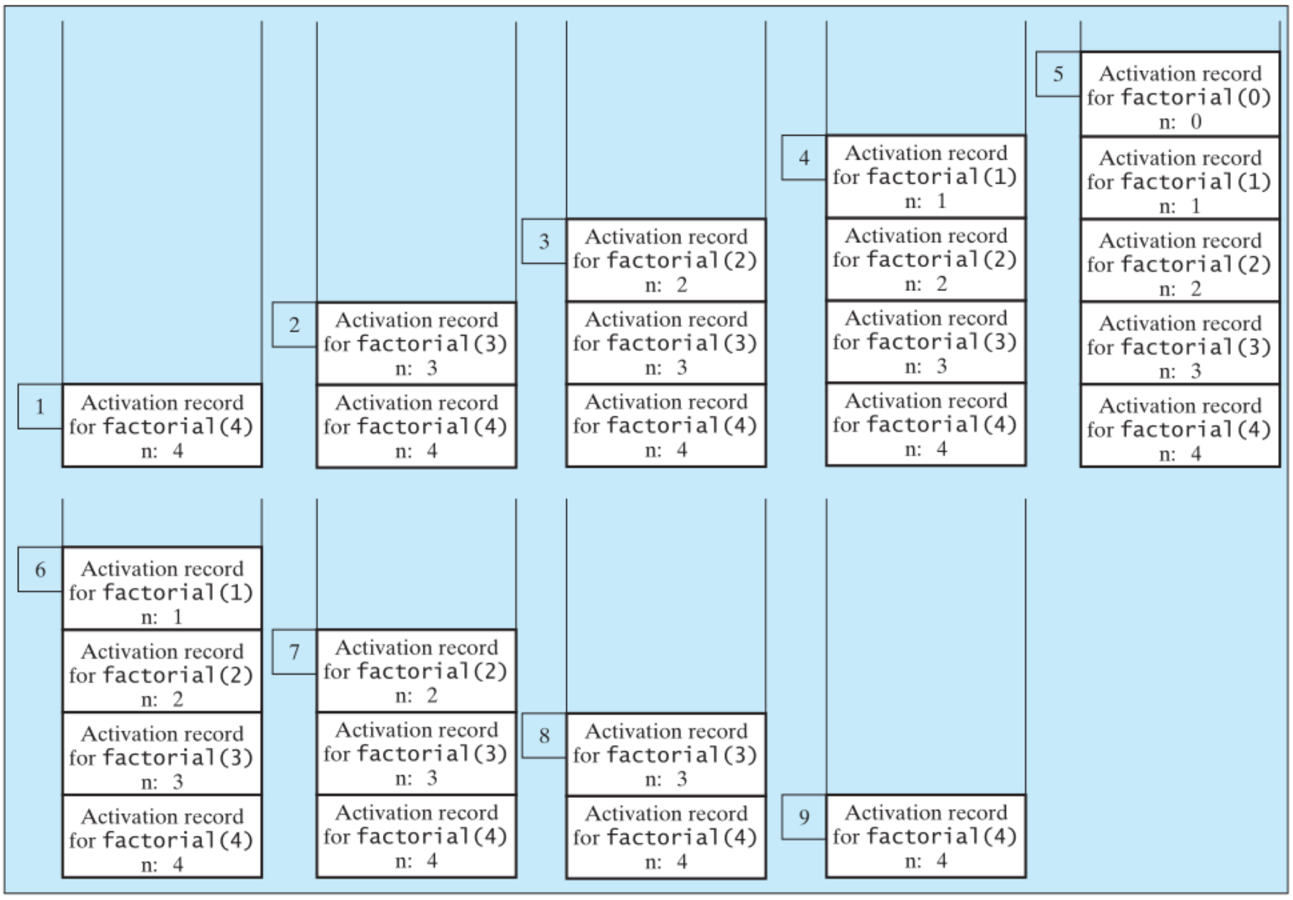


FIGURE 18.2 Invoking `factorial(4)` spawns recursive calls to `factorial`.



1 Activation record for factorial(4) n: 4
2 Activation record for factorial(3) n: 3
Activation record for factorial(4) n: 4
3 Activation record for factorial(2) n: 2
Activation record for factorial(3) n: 3
Activation record for factorial(4) n: 4
4 Activation record for factorial(1) n: 1
Activation record for factorial(2) n: 2
Activation record for factorial(3) n: 3
Activation record for factorial(4) n: 4
5 Activation record for factorial(0) n: 0
Activation record for factorial(1) n: 1
Activation record for factorial(2) n: 2
Activation record for factorial(3) n: 3
Activation record for factorial(4) n: 4
6 Activation record for factorial(1) n: 1
Activation record for factorial(2) n: 2
Activation record for factorial(3) n: 3
Activation record for factorial(4) n: 4
7 Activation record for factorial(2) n: 2
Activation record for factorial(3) n: 3
Activation record for factorial(4) n: 4
8 Activation record for factorial(3) n: 3
Activation record for factorial(4) n: 4
9 Activation record for factorial(4) n: 4

FIGURE 18.3 When `factorial(4)` is being executed, the `factorial` method is called recursively, causing stack space to dynamically change.

LISTING 18.2 ComputeFibonacci.java

```java
import java.util.Scanner;

public class ComputeFibonacci {
  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);
    System.out.print("Enter an index for a Fibonacci number: ");
    int index = input.nextInt();

    // Find and display the Fibonacci number
    System.out.println("The Fibonacci number at index "
      + index + " is " + fib(index));
  }

  /** The method for finding the Fibonacci number */
  public static long fib(long index) {
    if (index == 0) // Base case
      return 0;
```

base case

```java
    else if (index == 1) // Base case
      return 1;
    else  // Reduction and recursive calls
      return fib(index - 1) + fib(index - 2);
  }
}
```

base case

recursion

```
Enter an index for a Fibonacci number: 1 ↵Enter
The Fibonacci number at index 1 is 1
```

```
Enter an index for a Fibonacci number: 6 ↵Enter
The Fibonacci number at index 6 is 8
```

```
Enter an index for a Fibonacci number: 7 ↵Enter
The Fibonacci number at index 7 is 13
```

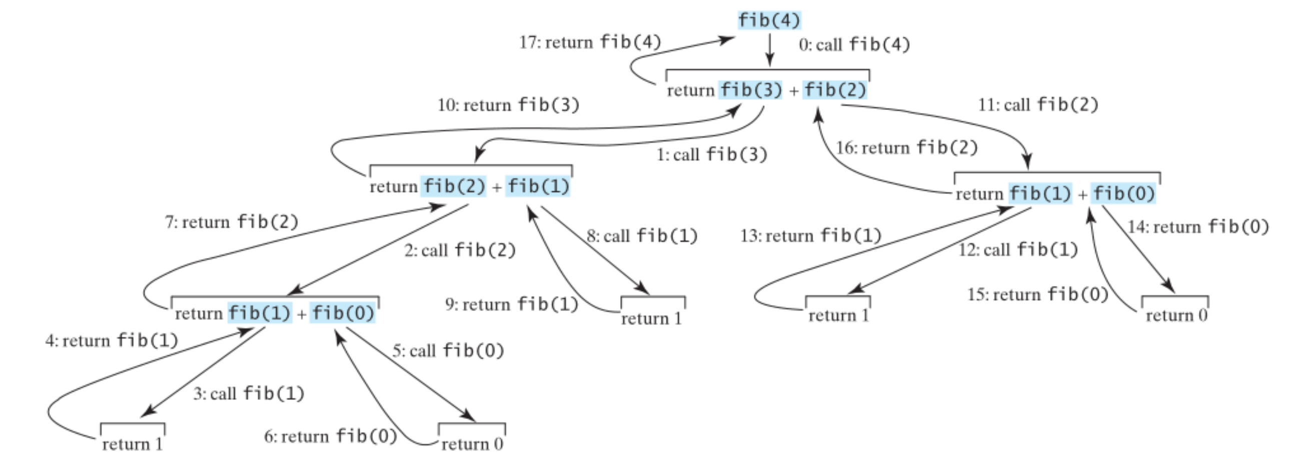


FIGURE 18.4 Invoking fib(4) spawns recursive calls to fib.

18.4 Problem Solving Using Recursion

If you think recursively, you can solve many problems using recursion.

recursion characteristics

The method is implemented using an if-else or a switch statement that leads to different cases.

One or more base cases (the simplest case) are used to stop recursion.

Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

Recursion is everywhere. It is fun to think recursively.

```
public static void drinkCoffee(Cup cup) {
  if (!cup.isEmpty()) {
    cup.takeOneSip(); // Take one sip
    drinkCoffee(cup);
  }
}
```

drinkCoffee and nPrintln methods are void and they do not return a value.

```
public static void nPrintln(String message, int times) {
  if (times >= 1) {
    System.out.println(message);
    nPrintln(message, times - 1);
  } // The base case is times == 0
}
```

LISTING 18.3 RecursivePalindromeUsingSubstring.java

```java
public class RecursivePalindromeUsingSubstring {
  public static boolean isPalindrome(String s) {
    if (s.length() <= 1) // Base case
      return true;
    else if (s.charAt(0) != s.charAt(s.length() - 1)) // Base case
      return false;
    else
      return isPalindrome(s.substring(1, s.length() - 1));
  }

  public static void main(String[] args) {
    System.out.println("Is moon a palindrome? "
```

method header
base case
base case
recursive call

```java
      + isPalindrome("moon"));
    System.out.println("Is noon a palindrome? "
      + isPalindrome("noon"));
    System.out.println("Is a a palindrome? " + isPalindrome("a"));
    System.out.println("Is aba a palindrome? " +
      isPalindrome("aba"));
    System.out.println("Is ab a palindrome? " + isPalindrome("ab"));
  }
}
```

```
Is moon a palindrome? false
Is noon a palindrome? true
Is a a palindrome? true
Is aba a palindrome? true
Is ab a palindrome? false
```

18.5 Recursive Helper Methods

Sometimes you can find a solution to the original problem by defining a recursive function to a problem similar to the original problem.

This new method is called a recursive helper method. The original problem can be solved by invoking the recursive helper method.

It is a common design technique in recursive programming to define a second method that receives additional parameters. Such a method is known as a recursive helper method.

LISTING 18.4 RecursivePalindrome.java

helper method

base case

base case

```
public class RecursivePalindrome {
  public static boolean isPalindrome(String s) {
    return isPalindrome(s, 0, s.length() - 1);
  }

  private static boolean isPalindrome(String s, int low, int high) {
    if (high <= low) // Base case
      return true;
    else if (s.charAt(low) != s.charAt(high)) // Base case
      return false;
    else
      return isPalindrome(s, low + 1, high - 1);
  }
```

```

  public static void main(String[] args) {
    System.out.println("Is moon a palindrome? "
      + isPalindrome("moon"));
    System.out.println("Is noon a palindrome? "
      + isPalindrome("noon"));
    System.out.println("Is a a palindrome? " + isPalindrome("a"));
    System.out.println("Is aba a palindrome? " + isPalindrome("aba"));
    System.out.println("Is ab a palindrome? " + isPalindrome("ab"));
  }
}
```

LISTING 18.5 RecursiveSelectionSort.java

helper method

base case

```java
public class RecursiveSelectionSort {
  public static void sort(double[] list) {
    sort(list, 0, list.length - 1); // Sort the entire list
  }

  private static void sort(double[] list, int low, int high) {
    if (low < high) {
      // Find the smallest number and its index in list[low .. high]
      int indexOfMin = low;
      double min = list[low];
      for (int i = low + 1; i <= high; i++) {
        if (list[i] < min) {
          min = list[i];
          indexOfMin = i;
        }
      }

      // Swap the smallest in list[low .. high] with list[low]
      list[indexOfMin] = list[low];
      list[low] = min;

```

recursive call

```java
      // Sort the remaining list[low+1 .. high]
      sort(list, low + 1, high);
    }
  }
}
```

LISTING 18.6 Recursive Binary Search Method

helper method

base case

recursive call

base case

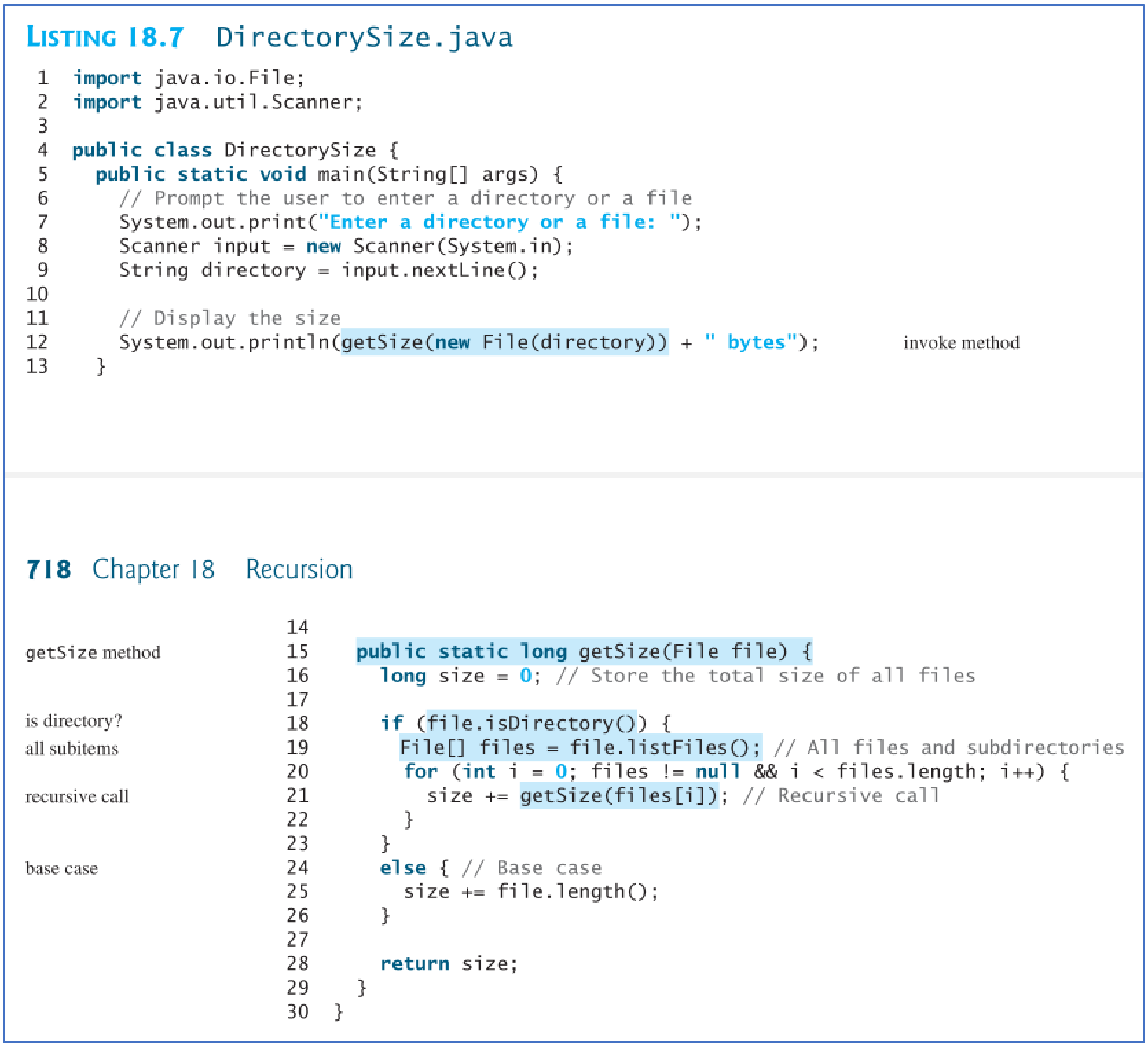
recursive call

```java
public class RecursiveBinarySearch {
  public static int recursiveBinarySearch(int[] list, int key) {
    int low = 0;
    int high = list.length - 1;
    return recursiveBinarySearch(list, key, low, high);
  }

  private static int recursiveBinarySearch(int[] list, int key,
      int low, int high) {
    if (low > high) // The list has been exhausted without a match
      return -low - 1;

    int mid = (low + high) / 2;
    if (key < list[mid])
      return recursiveBinarySearch(list, key, low, mid - 1);
    else if (key == list[mid])
      return mid;
    else
      return recursiveBinarySearch(list, key, mid + 1, high);
  }
}
```

Listing 18.7 gives a program that prompts the user to enter a directory or a file and displays its size



LISTING 18.7 DirectorySize.java

```java
import java.io.File;
import java.util.Scanner;

public class DirectorySize {
  public static void main(String[] args) {
    // Prompt the user to enter a directory or a file
    System.out.print("Enter a directory or a file: ");
    Scanner input = new Scanner(System.in);
    String directory = input.nextLine();

    // Display the size
    System.out.println(getSize(new File(directory)) + " bytes");
  }
```

invoke method

```java

  public static long getSize(File file) {
    long size = 0; // Store the total size of all files

    if (file.isDirectory()) {
      File[] files = file.listFiles(); // All files and subdirectories
      for (int i = 0; files != null && i < files.length; i++) {
        size += getSize(files[i]); // Recursive call
      }
    }
    else { // Base case
      size += file.length();
    }

    return size;
  }
}
```

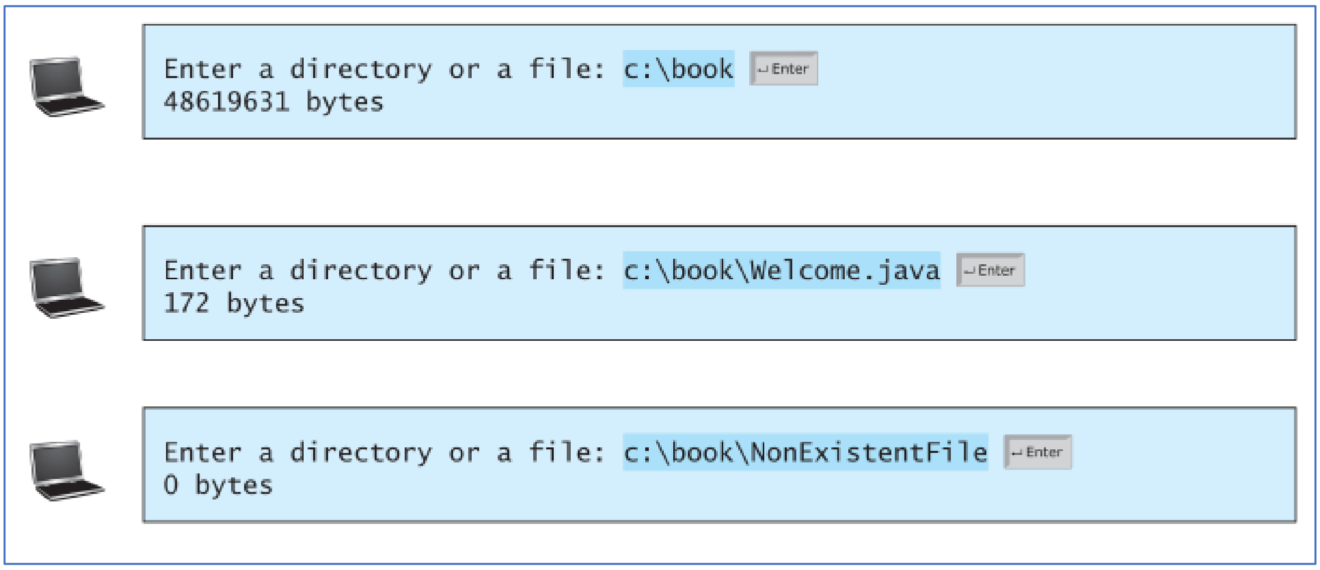
getSize method

is directory?

all subitems

recursive call

base case



```
Enter a directory or a file: c:\book ↵Enter
48619631 bytes
```

```
Enter a directory or a file: c:\book\Welcome.java ↵Enter
172 bytes
```

```
Enter a directory or a file: c:\book\NonExistentFile ↵Enter
0 bytes
```

18.9 Recursion vs. Iteration

Recursion is an alternative form of program control. It is essentially repetition without a loop.

Recursion bears substantial overhead.

Each time the program calls a method, the system must allocate memory for all of the method's local variables and parameters.

This can consume considerable memory and requires extra time to manage the memory.

Any problem that can be solved recursively can be solved nonrecursively with iterations.

Recursion has some negative aspects: it uses up too much time and too much memory.

Why, then, should you use it? In some cases, using recursion enables you to specify a clear, simple solution for an inherently recursive problem that would otherwise be difficult to obtain.

Examples are the directory-size problem, the Tower of Hanoi problem.

The decision whether to use recursion or iteration should be based on the nature of, and your understanding of, the problem you are trying to solve.

The rule of thumb is to use whichever approach can best develop an intuitive solution that naturally mirrors the problem.

If an iterative solution is obvious, use it. It will generally be more efficient than the recursive option.

18.10 Tail Recursion

A tail recursive method is efficient for reducing stack size.

A recursive method is said to be tail recursive if there are no pending operations to be performed on return from a recursive call.

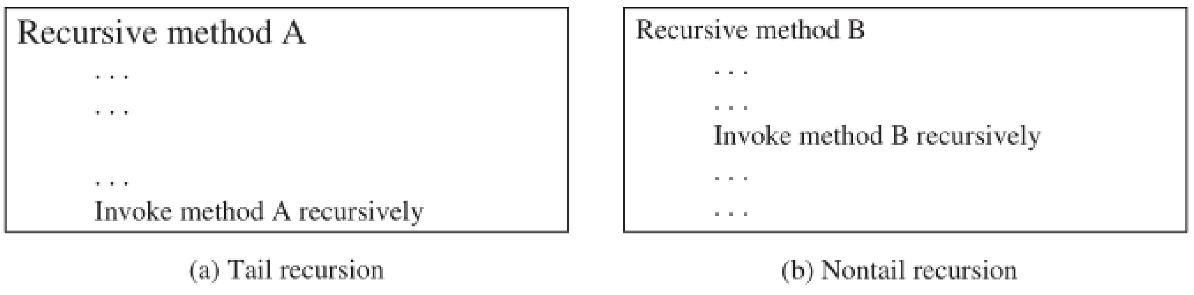


FIGURE 18.11 A tail-recursive method has no pending operations after a recursive call.

LISTING 18.10 ComputeFactorialTailRecursion.java

```
public class ComputeFactorialTailRecursion {
  /** Return the factorial for a specified number */
  public static long factorial(int n) {
    return factorial(n, 1); // Call auxiliary method
  }

  /** Auxiliary tail-recursive method for factorial */
  private static long factorial(int n, int result) {
    if (n == 0)
      return result;
    else
      return factorial(n - 1, n * result); // Recursive call
  }
}
```

original method

invoke auxiliary method

auxiliary method

recursive call