Recursion
OBJECTIVES

In this chapter you will learn:

- The concept of recursion.
- How to write and use recursive methods.
- How to determine the base case and recursion step in a recursive algorithm.
- How recursive method calls are handled by the system.
- The differences between recursion and iteration, and when it is appropriate to use each.
15.1 Introduction

- Earlier programs structured as methods that call one another in a disciplined, hierarchical manner

- Recursive methods
  - Call themselves
  - Useful for some problems to define a method to call itself
  - Can be called directly or indirectly through another method
15.2 Recursion Concepts

• Recursive problem-solving elements
  – Base case
    • Recursive method capable of solving only simplest case—the base case
    • If method is called with base case, method returns result
  – If method is called with more complex problem, problem divided into two pieces—a piece the method knows how to do and a piece the method does not know how to do (called recursive call or recursion step)
  – Recursive call/recursion step
    • Must resemble original problem but be slightly simpler or smaller version
    • Method calls fresh copy of itself to work on smaller problem
    • Normally includes return statement

• Indirect recursion
  – Recursive method calls another method that eventually makes call back to recursive method
15.3 Example Using Recursion: Factorials

- Factorial of $n$, or $n!$ is the product
  \[ n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1 \]
  With $1!$ equal to 1 and $0!$ Defined to be 1.
- Can be solved recursively or iteratively (nonrecursively)
- Recursive solution uses following relationship:
  \[ n! = n \cdot (n - 1)! \]
- Infinite recursion – recursive calls are continuously made until memory has been exhausted
  - Caused by either omitting base case or writing recursion step that does not converge on base case
Fig. 15.2 | Recursive evaluation of $5!$. 

(a) Sequence of recursive calls.

(b) Values returned from each recursive call.
```java
// Fig. 15.3: FactorialCalculator.java
// Recursive factorial method.

public class FactorialCalculator {
    // recursive method factorial
    public long factorial( long number ) {
        if ( number <= 1 ) // test for base case
            return 1; // base cases: 0! = 1 and 1! = 1
        else // recursion step
            return number * factorial( number - 1 );
    } // end method factorial

    // output factorials for values 0-10
    public void displayFactorials() {
        // calculate the factorials of 0 through 10
        for ( int counter = 0; counter <= 10; counter++ )
            System.out.printf( "%d! = %d\n", counter, factorial( counter ) );
    } // end method displayFactorials
} // end class FactorialCalculator
```

Outline

FactorialCalculator.java

Base case returns 1

Recursion step breaks problem into two parts: one the method knows how to do, one the method does not

Portion method

Recursive call: Portion method does not know how to do; smaller version of original problem

Original call to recursive method
public class FactorialTest
{
    // calculate factorials of 0-10
    public static void main( String args[] )
    {
        FactorialCalculator factorialCalculator = new FactorialCalculator();
        factorialCalculator.displayFactorials();
    } // end main
} // end class FactorialTest

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
15.4 Example Using Recursion: Fibonacci Series

- Fibonacci series begins with 0 and 1 and has property that each subsequent Fibonacci number is the sum of previous two Fibonacci numbers.
- Series occurs in nature, ratio of successive Fibonacci numbers converges on golden ratio or golden mean
- Fibonacci series defined recursively as:
  
  - fibonacci(0) = 0
  - fibonacci(1) = 1
  - fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)
- Recursive solution for calculating Fibonacci values results in explosion of recursive method calls
public class FibonacciCalculator
{
    // recursive declaration of method fibonacci
    public long fibonacci( long number )
    {
        if ( ( number == 0 ) || ( number == 1 ) ) // base cases
            return number;
        else // recursion step
            return fibonacci( number - 1 ) + fibonacci( number - 2 );
    } // end method fibonacci

    public void displayFibonacci()
    {
        for ( int counter = 0; counter <= 10; counter++ )
            System.out.printf( "Fibonacci of %d is: %d\n", counter,
                fibonacci( counter ) );
    } // end method displayFibonacci
} // end class FibonacciCalculator
// Fig. 15.6: FibonacciTest.java
// Testing the recursive fibonacci method.

public class FibonacciTest
{
    public static void main( String args[] )
    {
        FibonacciCalculator fibonacciCalculator = new FibonacciCalculator();
        fibonacciCalculator.displayFibonacci();
    } // end main
} // end class FibonacciTest

Fibonacci of 0 is: 0
Fibonacci of 1 is: 1
Fibonacci of 2 is: 1
Fibonacci of 3 is: 2
Fibonacci of 4 is: 3
Fibonacci of 5 is: 5
Fibonacci of 6 is: 8
Fibonacci of 7 is: 13
Fibonacci of 8 is: 21
Fibonacci of 9 is: 34
Fibonacci of 10 is: 55
Fig. 15.7 | Set of recursive calls for fibonacci(3).
15.5 Recursion and the Method Call Stack

• Method call stack used to keep track of method calls and local variables within a method call
• Just as with nonrecursive programming, recursive method calls are placed at the top of the method call stack
• As recursive method calls return, their activation records are popped off the stack and the previous recursive calls continue executing
• Current method executing is always method whose activation record is at top of stack
15.6 Recursion vs. Iteration

• Any problem that can be solved recursively can be solved iteratively

• Both iteration and recursion use a control statement
  – Iteration uses a repetition statement
  – Recursion uses a selection statement

• Iteration and recursion both involve a termination test
  – Iteration terminates when the loop-continuation condition fails
  – Recursion terminates when a base case is reached

• Recursion can be expensive in terms of processor time and memory space, but usually provides a more intuitive solution
Software Engineering Observation 15.1

Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach is normally preferred over an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that is easier to understand and debug. A recursive approach can often be implemented with fewer lines of code. Another reason to choose a recursive approach is that an iterative one might not be apparent.
// Fig. 15.10: FactorialCalculator.java

// Iterative factorial method.

public class FactorialCalculator {

    // recursive declaration of method factorial
    public long factorial( long number ) {
        long result = 1;

        // iterative declaration of method factorial
        for ( long i = number; i >= 1; i-- )
            result *= i;

        return result;
    } // end method factorial

    // output factorials for values 0-10
    public void displayFactorials() {
        // calculate the factorials of 0 through 10
        for ( int counter = 0; counter <= 10; counter++ )
            System.out.printf( "%d! = %d\n", counter, factorial( counter ) );
    } // end method displayFactorials

} // end class FactorialCalculator

// Iterative solution uses counter-controlled repetition
// Fig. 15.11: FactorialTest.java
// Testing the iterative factorial method.

public class FactorialTest
{
    // calculate factorials of 0-10
    public static void main( String args[] )
    {
        FactorialCalculator factorialCalculator = new FactorialCalculator();
        factorialCalculator.displayFactorials();
    } // end main
} // end class FactorialTest

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
Performance Tip 15.2

Avoid using recursion in situations requiring high performance. Recursive calls take time and consume additional memory.
15.7 String Permutations

- Permutations of a string of text – all the different strings that can be created by rearranging characters of original string
- Words created from permutations are known as anagrams
- Recursive solution: Remove one character, find permutations of remaining characters (recursive case), combine permutations with character that was removed
- Base case: Finding permutations for just one character – character itself is the only permutation
- Any string provides $n!$ permutations for $n$ characters
1 // Fig. 15.12: Permutation.java
2 // Recursive method to find all permutations of a String.
3
4 public class Permutation
5 {
6     // recursive declaration of method permuteString
7     private void permuteString(
8         String beginningString, String endingString
9     )
10     {
11         // base case: if string to permute is length 1
12         // (beginningString) with endingString, which is only one character
13         if ( endingString.length() <= 1 )
14             System.out.println( beginningString + endingString );
15         else // recursion step: permute endingString
16             {
17                 // for each character in endingString
18                 for ( int i = 0; i < endingString.length(); i++ )
19                     {
20                         try
21                             {
22                                 // create new string to permute by eliminating
23                                 // character at index i
24                                 String newString = endingString.substring( 0, i ) +
25                                         endingString.substring( i + 1 );
26                             }
27                         catch ( IndexOutOfBoundsException e )
28                             { System.out.println( e ); } // catch exceptions
29                     }
30                 }
31             }
32     }
33 }
34
35 // Fig. 15.13: PermutationTest.java
36 // Test program for the Permutation class.
37
38 public class PermutationTest
39 {
40     public static void main( String args[] )
41     {
42         Permutation permutation = new Permutation();
43         permutation.permuteString( "abc" );
44     }
45 }

// recursive call with a new string to permute
// and a beginning string to concatenate,
// includes the character at index i
permuteString( beginningString +
    endingString.charAt( i ), newString );
}

} // end try

catch ( StringIndexOutOfBoundsException exception )
{
    exception.printStackTrace();
}

} // end method permuteString

} // end class Permutation
// Fig. 15.13: PermutationTest.java
// Testing the recursive method to permute strings.
import java.util.Scanner;

public class PermutationTest {
    public static void main(String args[])
    {
        Scanner scanner = new Scanner(System.in);
        Permutation permutationObject = new Permutation();

        System.out.print("Enter a string: ");
        String input = scanner.nextLine(); // retrieve String to permute

        // permute String
        permutationObject.permuteString("", input);
    } // end main
} // end class PermutationTest
Outline

PermutationTest

.java

(2 of 2)
15.8 Towers of Hanoi

• Classic problem – Priests in Far East are attempting to move a stack of disks from one peg to another. One disk must be moved at a time, at no time may a larger disk be placed above a smaller disk

• Recursive solution:
  – Move n – 1 disks from peg 1 to peg 2, using peg 3 as temporary holding area
  – Move the last disk (the largest) from peg 1 to peg 3
  – Move the n – 1 disks from peg 2 to peg 3, using peg 1 as a temporary holding area

• Base case: When only one disk needs to be moved – no temporary holding area needed, disk is simply moved
Fig. 15.14 | Towers of Hanoi for the case with four disks.
15.9 Fractals

• Fractal – a geometric figure that often can be generated from a pattern repeated recursively an infinite number of times

• Pattern applied to each segment of original figure

• Benoit Mandelbrot introduced term “fractal,” along with specifics of how fractals are created and their practical applications
  – Help us better understand patterns in nature, the human body and the universe
  – Popular art form
15.9 Fractals

• Self-similar property – fractals have this property in the case that, when subdivided into parts, each resembles a reduced-size copy of the whole

• If part is exact copy of original, fractal is said to be strictly self similar

• Each time pattern is applied, fractal is said to be at new level or depth

• Fractal examples: Koch Curve, Koch Snowflake
Fig. 15.17 | Koch Curve fractal.