Chapter 12
Recursion

Java Software Solutions
Foundations of Program Design
Seventh Edition

John Lewis
William Loftus
Recursion

• Recursion is a fundamental programming technique that can provide an elegant solution certain kinds of problems

• Chapter 12 focuses on:
  – thinking in a recursive manner
  – programming in a recursive manner
  – the correct use of recursion
  – recursion examples
Outline

Recursive Thinking
Recursive Programming
Using Recursion
Recursion in Graphics
Recursive Thinking

• A recursive definition is one which uses the word or concept being defined in the definition itself

• When defining an English word, a recursive definition is often not helpful

• But in other situations, a recursive definition can be an appropriate way to express a concept

• Before applying recursion to programming, it is best to practice thinking recursively
Recursive Definitions

• Consider a list of numbers:

  24, 88, 40, 37

• A list can be defined as follows:

  A List is a: number
  or a: number comma List

• That is, a List is defined to be a single number, or a number followed by a comma followed by a List

• The concept of a List is used to define itself
Recursive Definitions

- The recursive part of the LIST definition is used several times, terminating with the non-recursive part:

```
LIST: number comma LIST
   24 , 88, 40, 37
   number comma LIST
       88 , 40, 37
   number comma LIST
       40 , 37
   number
       37
```
Peano's def. of Natural Numbers

• The following two axioms define the natural numbers
  – 0 is a natural number
  – For every natural number $n$, $S(n)$ is a natural number

• The number 1 can be defined as $S(0)$, 2 as $S(S(0))$ (which is also $S(1)$), and, in general, any natural number $n$ as $S^n(0)$

• The next two axioms define their properties:
  – For every natural number $n$, $S(n) = 0$ is false. That is, there is no natural number whose successor is 0
  – For all natural numbers $m$ and $n$, if $S(m) = S(n)$, then $m = n$. That is, $S$ is an injection.
Infinite Recursion

• All recursive definitions have to have a non-recursive part called the base case

• If they didn't, there would be no way to terminate the recursive path

• Such a definition would cause infinite recursion

• This problem is similar to an infinite loop, but the non-terminating "loop" is part of the definition itself
Quiz

What is printing the program described in this flowchart?
Recursive Definition: Factorial

• N!, for any positive integer N, is defined to be the product of all integers between 1 and N inclusive

\[ N! = N \times (N-1) \times (N-2) \times \ldots \times 2 \times 1 \]

• This definition can be expressed recursively as:

\[ 1! = 1 \]
\[ N! = N \times (N-1)! \]

• A factorial is defined in terms of another factorial

• Eventually, the base case of 1! is reached
Recursive Factorial

5!
5 \times 4!
4 \times 3!
3 \times 2!
2 \times 1!

1

120
24
6
2
1

Copyright © 2012 Pearson Education, Inc.
Quick Check

Write a recursive definition of

\[ f: \ n \mapsto 5 \times n \]

where \( n > 0 \).
Quick Check

Write a recursive definition of

\[ f: n \mapsto 5 \times n \]

where \( n > 0 \).

\[ 5 \times 1 = 5 \]

\[ 5 \times n = 5 + (5 \times (n-1)) \]

\[ f(1) = 5 \]

\[ f(n) = 5 + f(n-1) \]
Quick Check

Write a recursive definition of $e^n$, where $n \geq 0$. 
Quick Check

Write a recursive definition of $e^n$, where $n \geq 0$.

\[ e^0 = 1 \]

\[ e^n = e \times e^{n-1} \]
A (non recursive) Method

```java
private class SliderListener implements ChangeListener {
    private int red, green, blue;

    // Gets the value of each slider, then updates the labels and the color panel.
    public void stateChanged (ChangeEvent event) {
        red = rSlider.getValue();
        green = gSlider.getValue();
        blue = bSlider.getValue();

        rLabel.setText ("Red: " + red);
        gLabel.setText ("Green: " + green);
        bLabel.setText ("Blue: " + blue);

        colorPanel.setBackground (new Color (red, green, blue));
    }
}
```

(stateChanged()) is NOT used in the definition of stateChanged()
Recursive Programming

• A recursive method is a method that invokes itself.

• A recursive method must be structured to handle both the base case and the recursive case.

• Each call to the method sets up a new execution environment, with new parameters and local variables.

• As with any method call, when the method completes, control returns to the method that invoked it (which may be an earlier invocation of itself).
Sum of 1 to N

• Consider the problem of computing the sum of all the numbers between 1 and any positive integer N

• This problem can be recursively defined as:

\[
\sum_{i=1}^{N} i = N + \sum_{i=1}^{N-1} i = N + (N - 1) + \sum_{i=1}^{N-2} i \\
\vdots \\
= N + (N - 1) + (N - 2) + \cdots + 2 + 1
\]
Sum of 1 to N

• The summation could be implemented recursively as follows:

```java
// This method returns the sum of 1 to num
public int sum (int num)
{
    int result;

    if (num == 1)
        result = 1;
    else
        result = num + sum (num-1);

    return result;
}
```
Recursive Programming

• Note that just because we can use recursion to solve a problem, doesn't mean we should

• We usually would not use recursion to solve the summation problem, because the iterative version is easier to understand

• However, for some problems, recursion provides an elegant solution, often cleaner than an iterative version

• You must carefully decide whether recursion is the correct technique for any problem
Quiz

• Write a recursive method that computes the factorial of a non-negative int number \( n \):
  \[
  \text{factorial}(0)=1, \quad \text{factorial}(n) = n \times \text{factorial}(n-1)
  \]
Factorial

public int factorial(int n) {
    if (n == 0)
        return 1;
    return n * factorial(n-1);
}

factorial(4)
  factorial(3)
    factorial(2)
      factorial(1)
        factorial(0)
          return 1
          return 1*1 = 1
        return 1*1 = 1
        return 2*1 = 2
        return 3*2 = 6
      return 4*6 = 24
    return 2*1 = 2
    return 3*2 = 6
  return 4*6 = 24
  return 24
Indirect Recursion

• A method invoking itself is considered to be *direct recursion*

• A method could invoke another method, which invokes another, etc., until eventually the original method is invoked again

• For example, method \textit{m1} could invoke \textit{m2}, which invokes \textit{m3}, which in turn invokes \textit{m1} again

• This is called *indirect recursion*, and requires all the same care as direct recursion

• It is often more difficult to trace and debug
Indirect Recursion
Quiz

• What does the following recursive function return?

```java
public String mystery(String s) {
    int N = s.length();
    if (N <= 1)
        return s;
    String a = s.substring(0, N/2);
    String b = s.substring(N/2, N);
    return mystery(b) + mystery(a);
}
```
Quiz

• What does the following recursive function return?

```java
public String mystery(String s) {
    int N = s.length();
    if (N <= 1)
        return s;
    String a = s.substring(0, N/2);
    String b = s.substring(N/2, N);
    return mystery(b) + mystery(a);
}
```

The reverse of the input string.
Mathematical Induction

- Recursive programming is directly related to mathematical induction, a technique for proving facts about discrete functions.

- Proving that a statement involving an integer $N$ is true for all $N$ by mathematical induction involves two steps:
  - **The base case:** to prove the statement true for some specific value or values of $N$ (usually 0 or 1).
  - **The induction step:** assume that a statement is true for all positive integers less than $N$, then use that fact to prove it true for $N$. 
Proof by Induction Example

• Prove that:
  – $1 + 2 + 3 + 4 + \ldots + N = (N+1)N/2$

• Base case:
  – $1 = (1+1)1/2$ TRUE

• Induction step:
  – Assume that it is true for $N-1$
    • $1+ \ldots + (N-1) = N(N-1)/2$
  – Then:
    • $1+ \ldots + (N-1) + N = N(N-1)/2 + N$
    • $= (N^2 - N + 2N)/2$
    • $= (N^2 + N)/2 = (N + 1) N /2$ Q.E.D.
Without using Induction

• Prove that:
  – $1 + 2 + 3 + 4 + \ldots + N = S_n = \frac{(N+1)N}{2}$

• $(1 + 2 + 3 + 4 + \ldots + N) + (1 + 2 + 3 + 4 + \ldots + N) = 2S_n$

• $(1 + 2 + 3 + 4 + \ldots + N) + (N + (N-1) + \ldots + 1) = 2S_n$

• $N(N+1) = 2S_n$

• $S_n = \frac{(N+1)N}{2}$
Quiz

• Consider the fibonacci sequence:
  – f(0) = 0, f(1) = 1, f(n) = f(n-1) + f(n-2)
  – 0, 1, 1, 2, 3, 5, 8, 13, ..

• Prove by induction that:
  – For all n > 0, f(3n) is even
Quiz

- Consider the Fibonacci sequence:
  - \( f(0) = 0, f(1) = 1, f(n) = f(n-1) + f(n-2) \)
  - 0, 1, 1, 2, 3, 5, 8, 13, ..

- Prove by induction that:
  - For all \( n > 0 \), \( f(3n) \) is even

- Base case \( n=1 \)
  - \( f(3) = 2 \) TRUE

- Induction step
  - if \( f(3n) \) is even we must prove that \( f(3(n+1)) \) is even
  - \( f(3(n+1)) = f(3n+2) + f(3n+1) = f(3n+1) + f(3n) + f(3n+1) = 2f(3n+1) + f(3n) \) THIS is EVEN
Recursion can be inefficient

```c
int fibonacci(int n) {
    if (n < 2)
        return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

How many calls to `fibonacci` for computing `fibonacci(5)`?
Recursion can be inefficient

```c
int fibonacci(int n) {
    if (n < 2)
        return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

How many calls to `fibonacci` for computing `fibonacci(5)`?

\[
\begin{array}{cccccc}
 f(5) & f(4) & f(3) & f(3) & f(2) & f(2) & f(1) \\
 f(3) & f(2) & f(2) & f(1) & f(1) & f(0) & f(0) \\
 f(2) & f(1) & f(1) & f(0) & f(1) & f(0) & \\
 f(1) & f(0) & & & & & \\
\end{array}
\]

15 calls
Iterative version of Fibonacci

```c
int itFibonacci(int n) {
    int result = 0, prec = 1;
    for (int i = 1; i <= n; i++) {
        result += prec;  // f(n+1) = f(n) + f(n-1)
        prec = result - prec;  // f(n) = f(n+1) - f(n-1)
    }
    return result;
}
```

f(0) = 0, f(1) = 1, f(n) = f(n-1) + f(n-2)
0, 1, 2, 3, 5, 8, 13, ...
Outline

Recursive Thinking
Recursive Programming
Using Recursion
Recursion in Graphics
Maze Traversal

• We can use recursion to find a path through a maze

• From each location, we can search in each direction

• The recursive calls keep track of the path through the maze

• The base case is an invalid move or reaching the final destination

• See MazeSearch.java
• See Maze.java
public class MazeSearch
{
    public static void main (String[] args)
    {
        Maze labyrinth = new Maze();

        System.out.println (labyrinth);

        if (labyrinth.traverse (0, 0))
            System.out.println ("The maze was successfully traversed!");
        else
            System.out.println ("There is no possible path.");

        System.out.println (labyrinth);
    }
}
public class MazeSearch {
   public static void main(String[] args) {
      Maze labyrinth = new Maze();
      System.out.println(labyrinth);
      if (labyrinth.traverse(0, 0))
         System.out.println("The maze was successfully traversed!");
      else
         System.out.println("There is no possible path.");
      System.out.println(labyrinth);
   }
}

Output
1110110001111
1011101111001
0000101010100
1110110101111
1010001111001
1011111101111
1000000000000
1111111111111

The maze was successfully traversed!
public class Maze
{
    private final int TRIED = 3;
    private final int PATH = 7;

    private int[][] grid = {
        {1,1,1,0,1,1,0,0,0,1,1,1,1},
        {1,0,1,1,1,0,1,1,1,1,0,0,1},
        {0,0,0,0,1,0,1,0,1,0,1,0,0},
        {1,1,1,0,1,1,1,0,1,0,1,1,1},
        {1,0,1,0,0,0,0,1,1,1,0,0,1},
        {1,0,1,1,1,1,1,1,0,1,1,1,1},
        {1,0,0,0,0,0,0,0,0,0,0,0,0},
        {1,1,1,1,1,1,1,1,1,1,1,1,1}
    };
continued
public boolean traverse (int row, int column)
{
    boolean done = false;

    if (valid (row, column))
    {
        grid[row][column] = TRIED;  // this cell has been tried

        if (row == grid.length-1 && column == grid[0].length-1)
            done = true;  // the maze is solved — base case
        else
        {
            done = traverse (row+1, column);  // down
            if (!done)
                done = traverse (row, column+1);  // right
            if (!done)
                done = traverse (row-1, column);  // up
            if (!done)
                done = traverse (row, column-1);  // left
        }

        if (done)  // this location is part of the final path
            grid[row][column] = PATH;
    }

    return done;
}
continued

// Determines if a specific location is valid.
private boolean valid (int row, int column)
{
    boolean result = false;

    // check if cell is in the bounds of the matrix
    if (row >= 0 && row < grid.length &&
        column >= 0 && column < grid[row].length)

        // check if cell is not blocked and not previously tried
        if (grid[row][column] == 1)
            result = true;

    return result;
}
public String toString ()
{
    String result = "\n";
    for (int row=0; row < grid.length; row++)
    {
        for (int column=0; column < grid[row].length; column++)
            result += grid[row][column] + "";
        result += "\n";
    }
    return result;
}
Quiz

• Trace the calls to `traverse()` and `valid()` for the maze
  `row0=11, row1=01`
Quiz

• Trace the calls to `traverse()` and `valid()` for the maze row0=11, row1=01
Towers of Hanoi

• The *Towers of Hanoi* is a puzzle made up of three vertical pegs and several disks that slide onto the pegs.

• The disks are of varying size, initially placed on one peg with the largest disk on the bottom with increasingly smaller ones on top.
Towers of Hanoi

• The goal is to move all of the disks from one peg to another

• Under the following rules:
  – Move only one disk at a time
  – A larger disk cannot be put on top of a smaller one
Towers of Hanoi

Original Configuration

Move 1

Move 2

Move 3

Target
Towers of Hanoi

Move 4

Move 5

Move 6

Move 7 (done)
Recursive Solution

• To solve a N-tower
  1. Solve the (N-1)-tower: move the (N-1)-tower in the middle peg
  2. Move the largest disc to target peg
  3. Solve the (N-1)-tower: move the (N-1)-tower from the middle peg to the target peg
Towers of Hanoi

- An iterative solution to the Towers of Hanoi is quite complex
- A recursive solution is much shorter and more elegant

- See `SolveTowers.java`
- See `TowersOfHanoi.java`
public class SolveTowers
{
    // Creates a TowersOfHanoi puzzle and solves it.
    public static void main (String[] args)
    {
        TowersOfHanoi towers = new TowersOfHanoi (4);
        towers.solve();
    }
}
public class SolveTowers {

    // Creates a TowersOfHanoi puzzle and solves it.
    public static void main(String[] args) {
        TowersOfHanoi towers = new TowersOfHanoi(4);
        towers.solve();
    }
}

Output

Move one disk from 1 to 2
Move one disk from 1 to 3
Move one disk from 2 to 3
Move one disk from 1 to 2
Move one disk from 3 to 1
Move one disk from 3 to 2
Move one disk from 1 to 2
Move one disk from 1 to 3
Move one disk from 2 to 3
Move one disk from 2 to 1
Move one disk from 3 to 1
Move one disk from 2 to 3
Move one disk from 1 to 2
Move one disk from 1 to 3
Move one disk from 2 to 3
/**
 * TowersOfHanoi.java       Author: Lewis/Loftus
 *
 * Represents the classic Towers of Hanoi puzzle.
 */

public class TowersOfHanoi
{
    private int totalDisks;

    //---
    //  Sets up the puzzle with the specified number of disks.
    //---
    public TowersOfHanoi (int disks)
    {
        totalDisks = disks;
    }

    //---
    //  Performs the initial call to moveTower to solve the puzzle.
    //  Moves the disks from tower 1 to tower 3 using tower 2.
    //---
    public void solve ()
    {
        moveTower (totalDisks, 1, 3, 2);
    }

continued
continued

    //----------------------------------------------------------------------------
    // Moves the specified number of disks from one tower to another
    // by moving a subtower of n-1 disks out of the way, moving one
    // disk, then moving the subtower back. Base case of 1 disk.
    //----------------------------------------------------------------------------
    private void moveTower (int numDisks, int start, int end, int temp)
    {
        if (numDisks == 1)
            moveOneDisk (start, end);
        else
        {
            moveTower (numDisks-1, start, temp, end);
            moveOneDisk (start, end);
            moveTower (numDisks-1, temp, end, start);
        }
    }

    //----------------------------------------------------------------------------
    // Prints instructions to move one disk from the specified start
    // tower to the specified end tower.
    //----------------------------------------------------------------------------
    private void moveOneDisk (int start, int end)
    {
        System.out.println ("Move one disk from " + start + " to " +
                            end);
    }
    

Hanoi Tower Solution

Hanoi Tower execution time (seconds)

number of tiles
public int mystery(int x, int y) {
    if (x == y) return 0;
    else return mystery(x-1, y) + 1;
}

If the method is called as mystery(8, 3), what is returned?
A) 11  
B) 8    
C) 5    
D) 3    
E) 24
public int mystery(int x, int y) {
    if (x == y) return 0;
    else return mystery(x-1, y) + 1;
}

If the method is called as mystery(8, 3), what is returned?

C) 5

The method computes x - y if x > y. The method works as follows: each time the method is called recursively, it subtracts 1 from x until (x == y) is becomes true, and adds 1 to the return value. So, 1 is added each time the method is called, and the method is called once for each int value between x and y.
Quiz

Calling the previous method will result in infinite recursion if which condition below is initially true?

A) \((x == y)\)
B) \((x != y)\)
C) \((x > y)\)
D) \((x < y)\)
E) \((x == 0 && y != 0)\)
Quiz

Calling the previous method will result in infinite recursion if which condition below is initially true?

A) \( x == y \)
B) \( x != y \)
C) \( x > y \)
D) \( x < y \)
E) \( x == 0 && y != 0 \)

If \( x < y \) is true initially, then the else clause is executed resulting in the method being recursively invoked with a value of \( x - 1 \), or a smaller value of \( x \), so that \( x < y \) will be true again, and so for each successive recursive call, \( x < y \) will be true and the base case, \( x == y \), will never be true.
Quiz

What does the following method compute? Assume the method is called initially with i = 0

```java
public int mystery(String a, char b, int i) {
    if (i == a.length()) return 0;
    else if (b == a.charAt(i))
        return mystery(a, b, i+1) + 1;
    else return mystery(a, b, i+1);
}
```
Quiz

What does the following method compute? Assume the method is called initially with i = 0

```java
public int mystery(String a, char b, int i) {
    if (i == a.length()) return 0;
    else if (b == a.charAt(i))
        return mystery(a, b, i+1) + 1;
    else return mystery(a, b, i+1);
}
```

The number of times char b appears in String a. The method compares each character in String a with char b until i reaches the length of String a. 1 is added to the return value for each match.
Outline

Recursive Thinking
Recursive Programming
Using Recursion
Recursion in Graphics
Tiled Pictures

• Consider the task of repeatedly displaying a set of images in a mosaic
  – Three quadrants contain individual images
  – Upper-left quadrant repeats pattern

• The base case is reached when the area for the images shrinks to a certain size

• See TiledPictures.java
import java.awt.*;
import javax.swing.JApplet;

public class TiledPictures extends JApplet
{
    private final int APPLET_WIDTH = 320;
    private final int APPLET_HEIGHT = 320;
    private final int MIN = 20;  // smallest picture size

    private Image world, everest, goat;

    continue
continue

//-----------------------------------------------------------------
//  Loads the images.
//-----------------------------------------------------------------
public void init()
{
    world = getImage (getDocumentBase(), "world.gif");
    everest = getImage (getDocumentBase(), "everest.gif");
    goat = getImage (getDocumentBase(), "goat.gif");

    setSize (APPLET_WIDTH, APPLET_HEIGHT);
}

//-----------------------------------------------------------------
//  Draws the three images, then calls itself recursively.
//-----------------------------------------------------------------
public void drawPictures (int size, Graphics page)
{
    page.drawImage (everest, 0, size/2, size/2, size/2, this);
    page.drawImage (goat, size/2, 0, size/2, size/2, this);
    page.drawImage (world, size/2, size/2, size/2, size/2, this);

    if (size > MIN)
        drawPictures (size/2, page);
}

continue
continue

//  Performs the initial call to the drawPictures method.
public void paint (Graphics page)
{
    drawPictures (APPLET_WIDTH, page);
}
}
```java
// Performs the initial call to the drawPictures method.
public void paint(Graphics page) {
    drawPictures(APPLET_WIDTH, page);
}
```
import javax.swing.JFrame;

public class TiledPicturesApp {

    public static void main(String[] args) {
        JFrame frame = new JFrame("Tiled Pictures");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(new TiledPicturesPanel());
        frame.pack();
        frame.setVisible(true);
    }
}

Application version of the previous applet
import java.awt.*;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.JPanel;

public class TiledPicturesPanel extends JPanel {

    private final int PANEL_WIDTH = 320;
    private final int PANEL_HEIGHT = 320;
    private final int MIN = 20; // smallest picture size

    private BufferedImage world, everest, goat;

}
public TiledPicturesPanel() {
try {
    world = ImageIO.read(new File("world.gif"));
    everest = ImageIO.read(new File("everest.gif"));
    goat = ImageIO.read(new File("goat.gif"));
} catch (IOException e) {
}
    setPreferredSize(new Dimension(PANEL_WIDTH, PANEL_HEIGHT));
}

continue
public void drawPictures(int size, Graphics page) {
    page.drawImage(everest, 0, size / 2, size / 2, size / 2, this);
    page.drawImage(goat, size / 2, 0, size / 2, size / 2, this);
    page.drawImage(world, size / 2, size / 2, size / 2, size / 2, this);

    if (size > MIN) {
        drawPictures(size / 2, page);
    }
}

public void paintComponent(Graphics page) {
    super.paintComponent(page);
    drawPictures(PANEL_WIDTH, page);
}

Fractals

• A *fractal* is a geometric shape made up of the same pattern repeated in different sizes and orientations

• The *Koch Snowflake* is a particular fractal that begins with an equilateral triangle

• To get a higher order of the fractal, the sides of the triangle are replaced with angled line segments

• See `KochSnowflake.java`

• See `KochPanel.java`
//********************************************************************
//  KochSnowflakeApp.java       Author: Lewis/Loftus
//
//  Demonstrates the use of recursion in graphics.
//********************************************************************

import javax.swing.JFrame;

public class KochSnowflakeApp {

    public static void main (String[] args)
    {
        JFrame frame = new JFrame("Koch Snowflake");
        frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

        frame.getContentPane().add(new KochMainPanel());
        frame.pack();
        frame.setVisible(true);
    }
}

This is an application; in the book you find an applet.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class KochMainPanel extends JPanel implements ActionListener {
    private final int PANEL_WIDTH = 400;
    private final int PANEL_HEIGHT = 440;

    private final int MIN = 1, MAX = 9;

    private JButton increase, decrease;
    private JLabel titleLabel, orderLabel;
    private KochPanel drawing;
    private Jpanel tools;

    continue
public KochMainPanel()
{
    tools = new JPanel();
    tools.setLayout (new BoxLayout(tools, BoxLayout.X_AXIS));
    tools.setPreferredSize (new Dimension (PANEL_WIDTH, 40));
    tools.setBackground (Color.yellow);
    tools.setOpaque (true);

    titleLabel = new JLabel ("The Koch Snowflake");
    titleLabel.setForeground (Color.black);

    increase = new JButton (new ImageIcon ("increase.gif"));
    increase.setPressedIcon (new ImageIcon ("increasePressed.gif"));
    increase.addActionListener (this);

    decrease = new JButton (new ImageIcon ("decrease.gif"));
    decrease.setPressedIcon (new ImageIcon ("decreasePressed.gif"));
    decrease.addActionListener (this);
continue

orderLabel = new JLabel ("Order: 1");
orderLabel.setForeground (Color.black);

tools.add (titleLabel);
tools.add (Box.createHorizontalStrut (40));
tools.add (decrease);
tools.add (increase);
tools.add (Box.createHorizontalStrut (20));
tools.add (orderLabel);

drawing = new KochPanel (1);

add (tools);
add (drawing);

setPreferredSize (PANEL_WIDTH, PANEL_HEIGHT);
}
public void actionPerformed(ActionEvent event) {
    int order = drawing.getOrder();

    if (event.getSource() == increase)
        order++;
    else
        order--;

    if (order >= MIN && order <= MAX)
    {
        orderLabel.setText("Order: " + order);
        drawing.setOrder(order);
        repaint();
    }
}
public void actionPerformed(ActionEvent event) {
    int order = drawing.getOrder();
    if (event.getSource() == increase)
        order++;
    else
        order--;
    if (order >= MIN && order <= MAX) {
        orderLabel.setText("Order: "+ order);
        drawing.setOrder(order);
        repaint();
    }
}

Applet started.

Copyright © 2012 Pearson Education, Inc.
Koch Snowflakes

$\langle x_1, y_1 \rangle$  \hspace{5cm}  $\langle x_5, y_5 \rangle$

Becomes

$\langle x_1, y_1 \rangle$  \hspace{5cm}  $\langle x_4, y_4 \rangle$  \hspace{5cm}  $\langle x_3, y_3 \rangle$

$\langle x_2, y_2 \rangle$  \hspace{5cm}  $\langle x_2, y_2 \rangle$  \hspace{5cm}  $\langle x_2, y_2 \rangle$

$\langle x_1, y_1 \rangle$  \hspace{5cm}  $\langle x_1, y_1 \rangle$  \hspace{5cm}  $\langle x_1, y_1 \rangle$

$= \sqrt{3/6} \cdot |P_5 - P_1|$

Copyright © 2012 Pearson Education, Inc.
public class KochPanel extends JPanel {
    private final int PANEL_WIDTH = 400;
    private final int PANEL_HEIGHT = 400;
    private final double SQ = Math.sqrt(3.0) / 6;
    private final int TOPX = 200, TOPY = 20;
    private final int LEFTX = 60, LEFTY = 300;
    private final int RIGHTX = 340, RIGHTY = 300;

    private int current; // current order
//--- draws the fractal recursively. The base case is order 1 for
// which a simple straight line is drawn. Otherwise three
// intermediate points are computed, and each line segment is
// drawn as a fractal.
//----------------------------------------------------------------------------
public void drawFractal(int order, int x1, int y1, int x5, int y5,
                          Graphics page) {
    int deltaX, deltaY, x2, y2, x3, y3, x4, y4;

    if (order == 1)
        page.drawLine(x1, y1, x5, y5);
    else
    {
        deltaX = x5 - x1;  // distance between end points
        deltaY = y5 - y1;

        x2 = x1 + deltaX / 3;  // one third
        y2 = y1 + deltaY / 3;

        x3 = (int) ((x1+x5)/2 + SQ * (y1-y5));  // tip of projection
        y3 = (int) ((y1+y5)/2 + SQ * (x5-x1));

        continue
    }
continue

    x4 = x1 + deltaX * 2/3;  // two thirds
    y4 = y1 + deltaY * 2/3;

    drawFractal (order-1, x1, y1, x2, y2, page);
    drawFractal (order-1, x2, y2, x3, y3, page);
    drawFractal (order-1, x3, y3, x4, y4, page);
    drawFractal (order-1, x4, y4, x5, y5, page);
}

//----------------------------------------------------------------------------
// Performs the initial calls to the drawFractal method.
//----------------------------------------------------------------------------
public void paintComponent (Graphics page)
{
    super.paintComponent (page);

    page.setColor (Color.green);

    page.setColor (Color.green);

    drawFractal (current, TOPX, TOPY, LEFTX, LEFTY, page);
    drawFractal (current, LEFTX, LEFTY, RIGHTX, RIGHTY, page);
    drawFractal (current, RIGHTX, RIGHTY, TOPX, TOPY, page);
}

continue
continue

//-----------------------------------------------------------------
//  Sets the fractal order to the value specified.
//-----------------------------------------------------------------
public void setOrder (int order)
{
    current = order;
}

//-----------------------------------------------------------------
//  Returns the current order.
//-----------------------------------------------------------------
public int getOrder ()
{
    return current;
}
}
Summary

- Chapter 12 has focused on:
  - thinking in a recursive manner
  - programming in a recursive manner
  - the correct use of recursion
  - recursion examples