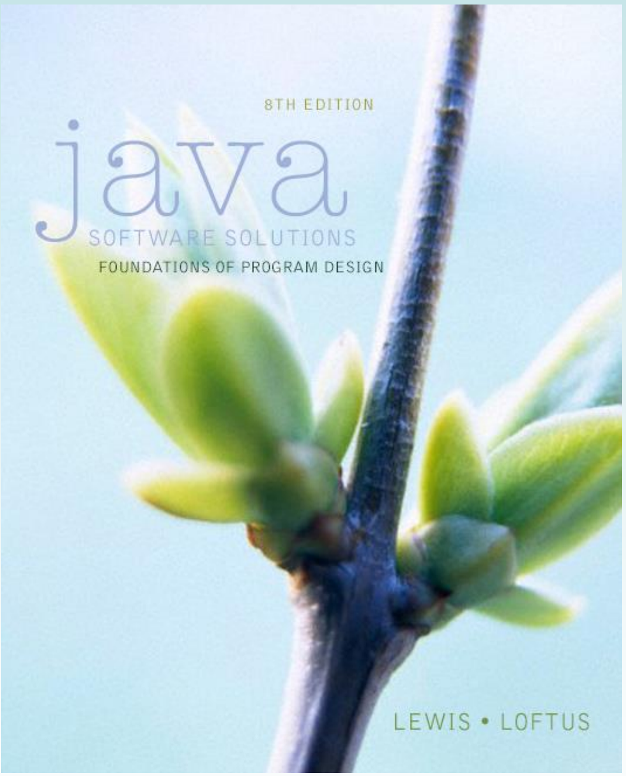
Chapter 12
Recursion



Java Software Solutions

Foundations of Program Design

8^{th} Edition

(Modified)

John Lewis
William Loftus

Addison-Wesley
is an imprint of
PEARSON

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 the following list of numbers:

```
24, 88, 40, 37
```

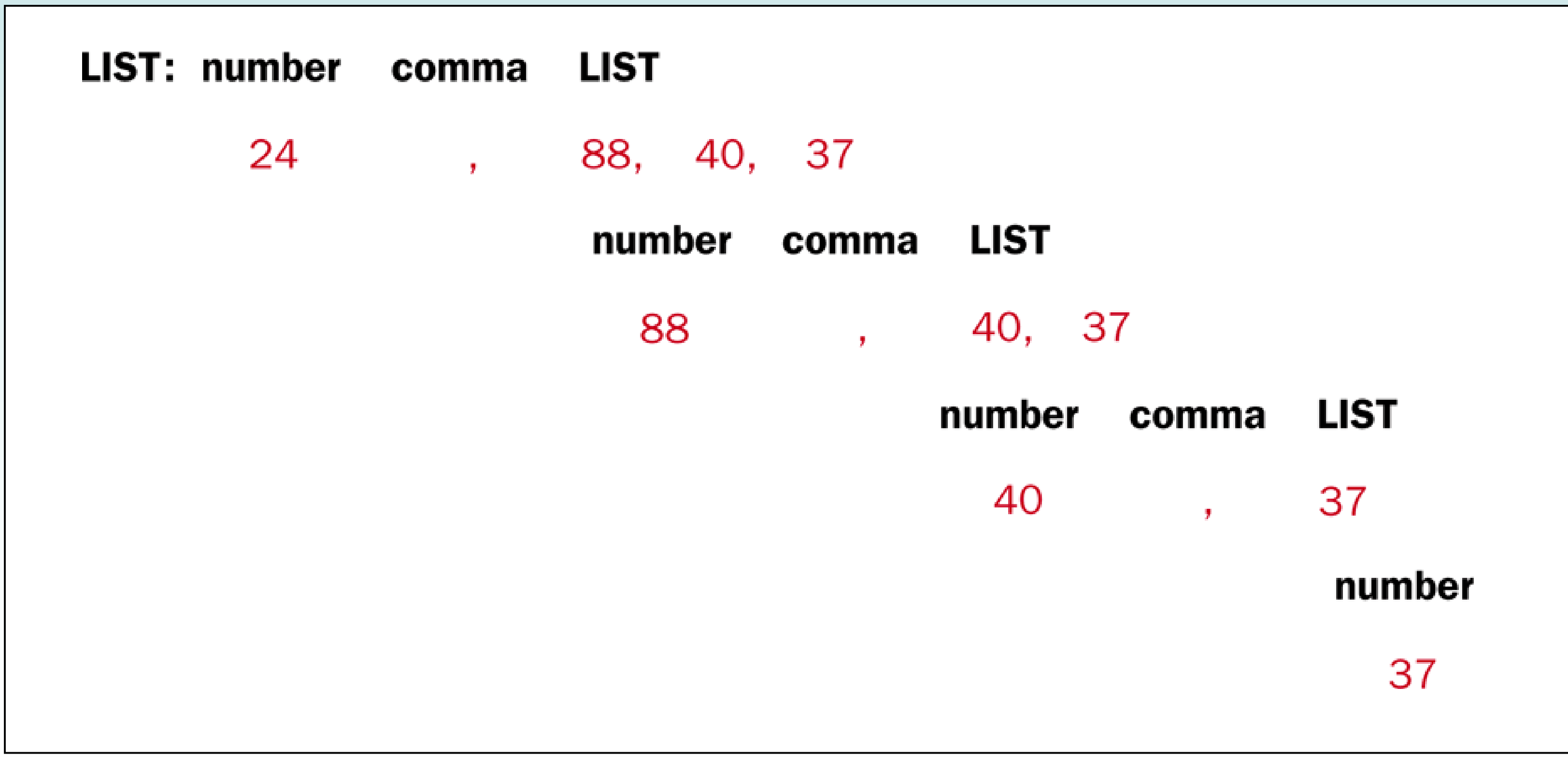
- Such a list can be defined as follows:

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:



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

Recursive Factorial

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

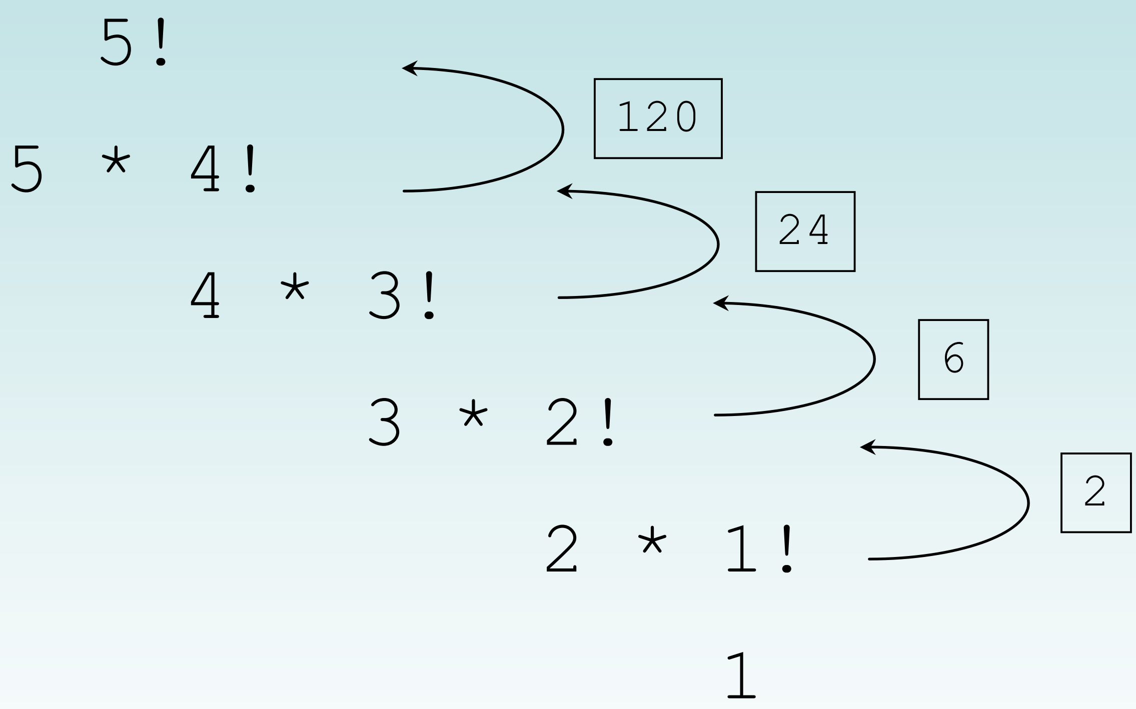
- This definition can be expressed recursively as:

```
1!  =  1
N!  =  N * (N-1)!
```

- A factorial is defined in terms of another factorial

- Eventually, the base case of 1! is reached

Recursive Factorial



Quick Check

Write a recursive definition of 5 * n, where n > 0.

Quick Check

Write a recursive definition of 5 * n, where n > 0.

```
5 * 1    =    5
5 * n    =    5 + (5 * (n-1))
```

Outline

Recursive Thinking

Recursive Programming

Using Recursion

Recursion in Graphics

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:

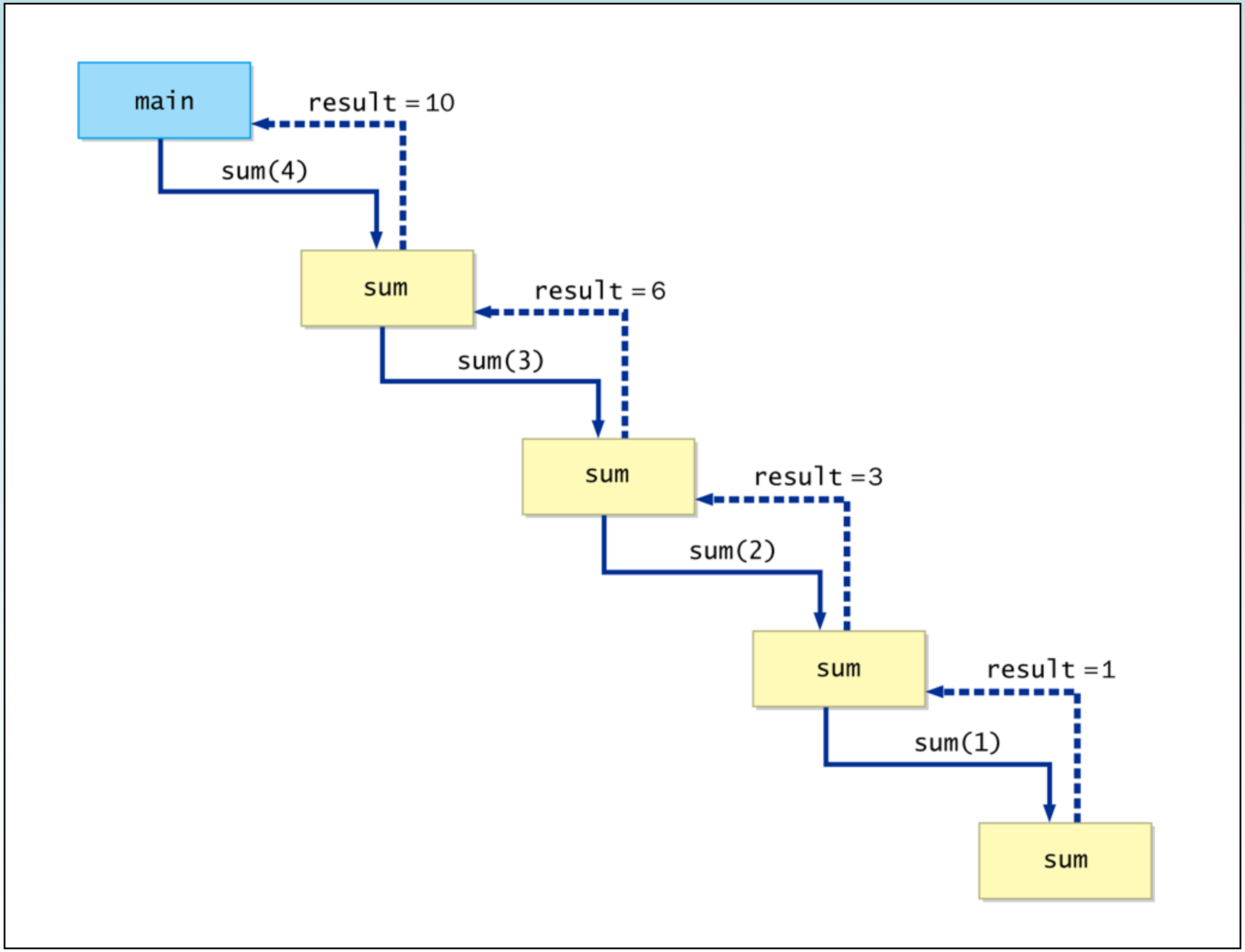
$$
\begin{aligned}
\sum_{i=1}^{N} i &= N + \sum_{i=1}^{N-1} i = N + N-1 + \sum_{i=1}^{N-2} i \\
&= N + N-1 + N-2 + \sum_{i=1}^{N-3} i \\
&\;\;\vdots \\
&= N + N-1 + N-2 + \cdots + 2 + 1
\end{aligned}
$$

Sum of 1 to N

- The summation could be implemented recursively as follows:

```
// 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(n-1);

   return result;
}
```

Sum of 1 to N



main
result = 10
sum(4)
sum
result = 6
sum(3)
sum
result = 3
sum(2)
sum
result = 1
sum(1)
sum

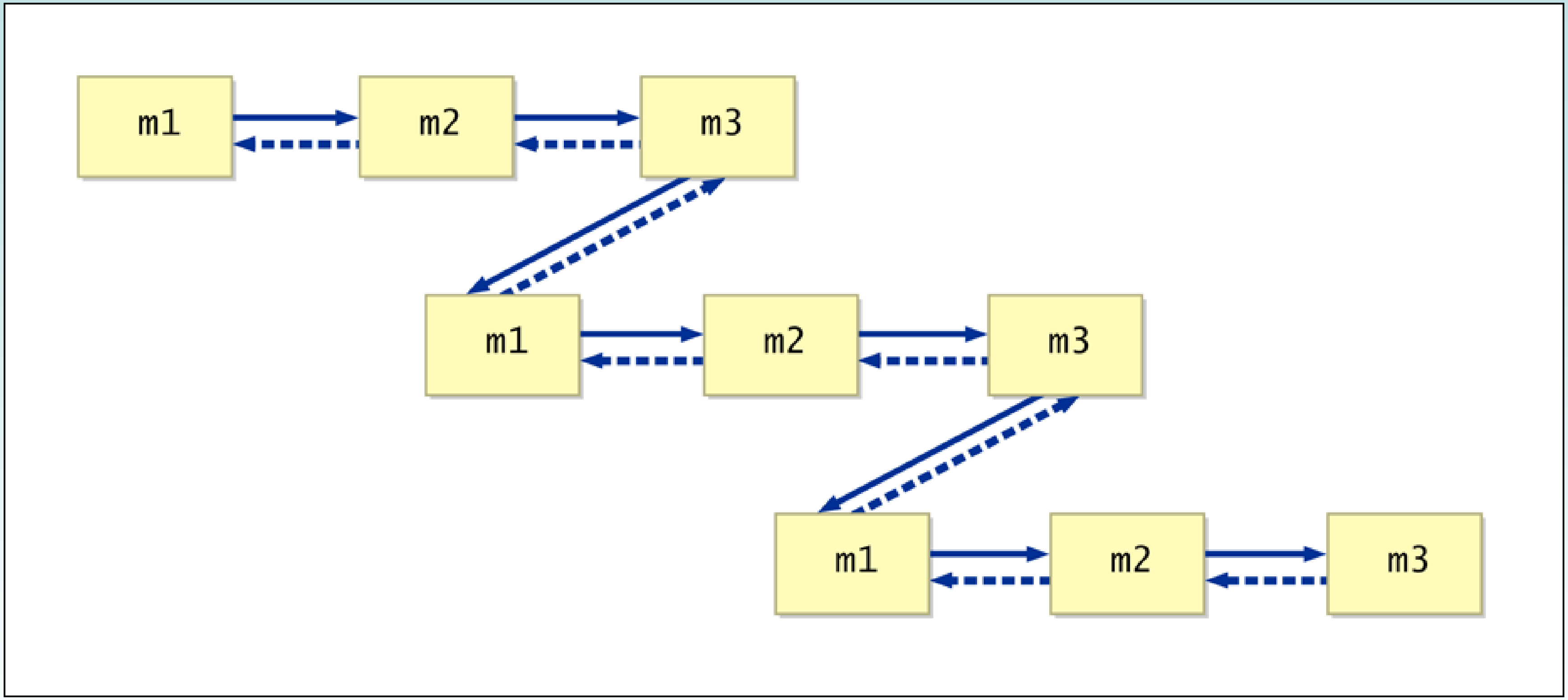
Recursive Programming

- 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

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 `m1` could invoke `m2`, which invokes `m3`, which in turn invokes `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



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`

```java
//********************************************************************
//  MazeSearch.java       Author: Lewis/Loftus
//
//  Demonstrates recursion.
//********************************************************************

public class MazeSearch
{
   //-----------------------------------------------------------------
   //  Creates a new maze, prints its original form, attempts to
   //  solve it, and prints out its final form.
   //-----------------------------------------------------------------
   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);
   }
}
```

```
//**********                                          **************
//  MazeSear
//
//  Demonstr
//**********                                          **************

public class
{
   //------                                          -------------
   //  Creat                                          mpts to
   //  solve
   //------                                          -------------
   public st
   {
      Maze l

      System

      if (la
         Sys                                          raversed!");
      else
         Sys                                          );

      System
   }
}
```

Output

```
111011000111
1011101111001
0000101010100
1110111010111
1010000111001
1011111101111
1000000000000
1111111111111

The maze was successfully traversed!

7770110001111
3077707771001
0000707070300
7770777070333
7070000773003
7077777703333
7000000000000
7777777777777
```

```java
//********************************************************************
//  Maze.java       Author: Lewis/Loftus
//
//  Represents a maze of characters. The goal is to get from the
//  top left corner to the bottom right, following a path of 1s.
//********************************************************************

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

continued

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

      return done;
   }

   //-----------------------------------------------------------------
   //  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;
   }
```

continued

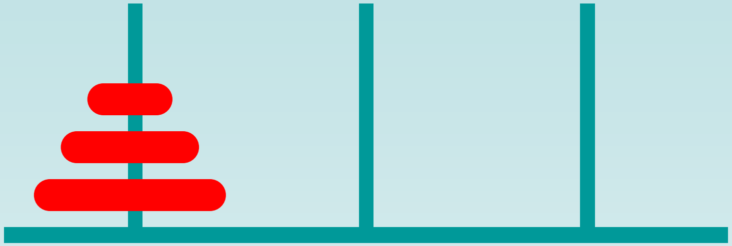
continued

```
   //-----------------------------------------------------------------
   //  Returns the maze as a string.
   //-----------------------------------------------------------------
   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;
   }
}
```

Towers of Hanoi

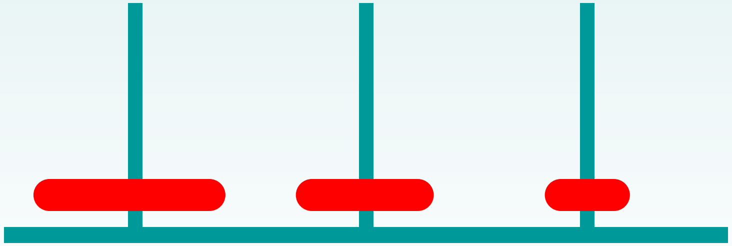
- The *Towers of Hanoi* is a puzzle made up of three vertical pegs and several disks that slide onto the pegs
- The disks are of varying size, initially placed on one peg with the largest disk on the bottom with increasingly smaller ones on top
- The goal is to move all of the disks from one peg to another 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

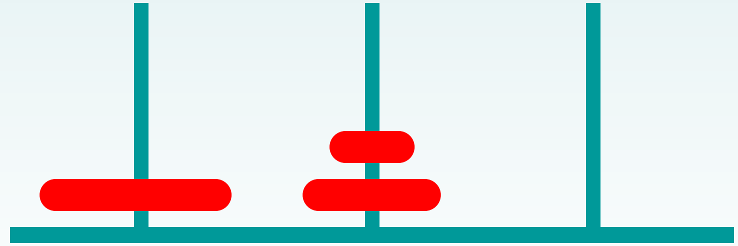


Original Configuration

Move 1

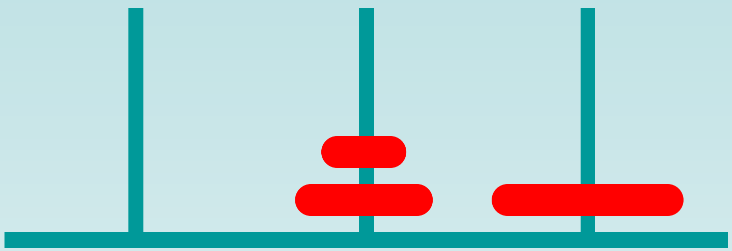


Move 2

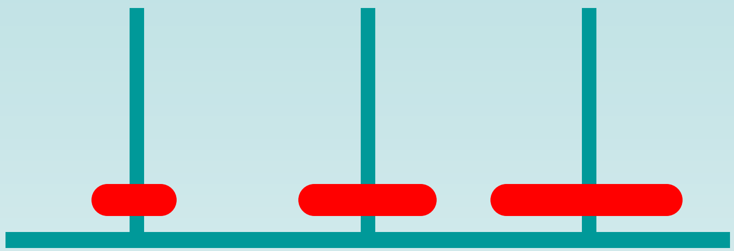


Move 3

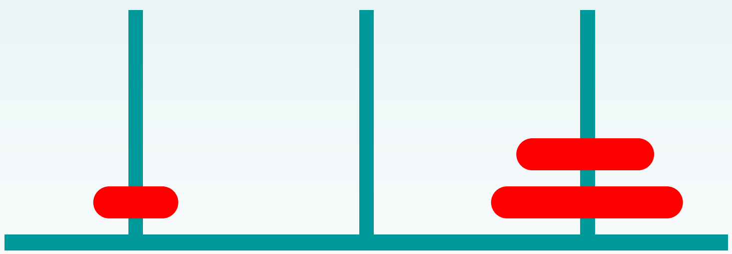
Towers of Hanoi



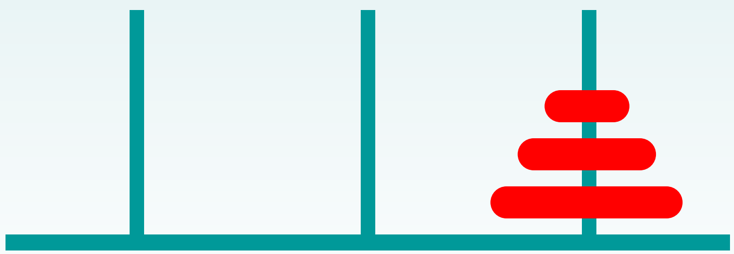
Move 4



Move 5



Move 6



Move 7 (done)

Towers of Hanoi

- An iterative solution to the Towers of Hanoi is quite complex
- A recursive solution is much shorter and more elegant:
  - Move topmost N-1 disks from original peg to spare peg
  - Move bottom (largest) disk to destination peg
  - Move N-1 disks from spare peg to destination peg

```java
//********************************************************************
//  SolveTowers.java       Author: Lewis/Loftus
//
//  Demonstrates recursion.
//********************************************************************

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

      towers.solve();
   }
}
```

```java
//*************                                    ********************
//  SolveTowers.
//
//  Demonstrates
//*************                                    ********************

public class Sol
{
   //----------                                    --------------------
   //  Creates a                                   it.
   //----------                                    --------------------
   public static
   {
      TowersOfHa                                   );

      towers.sol
   }
}
```

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
```

```java
//********************************************************************
//  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

```java
   //-----------------------------------------------------------------
   //  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);
   }
}
```

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`

```java
//********************************************************************
//  TiledPictures.java       Author: Lewis/Loftus
//
//  Demonstrates the use of recursion.
//********************************************************************

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);
   }
```

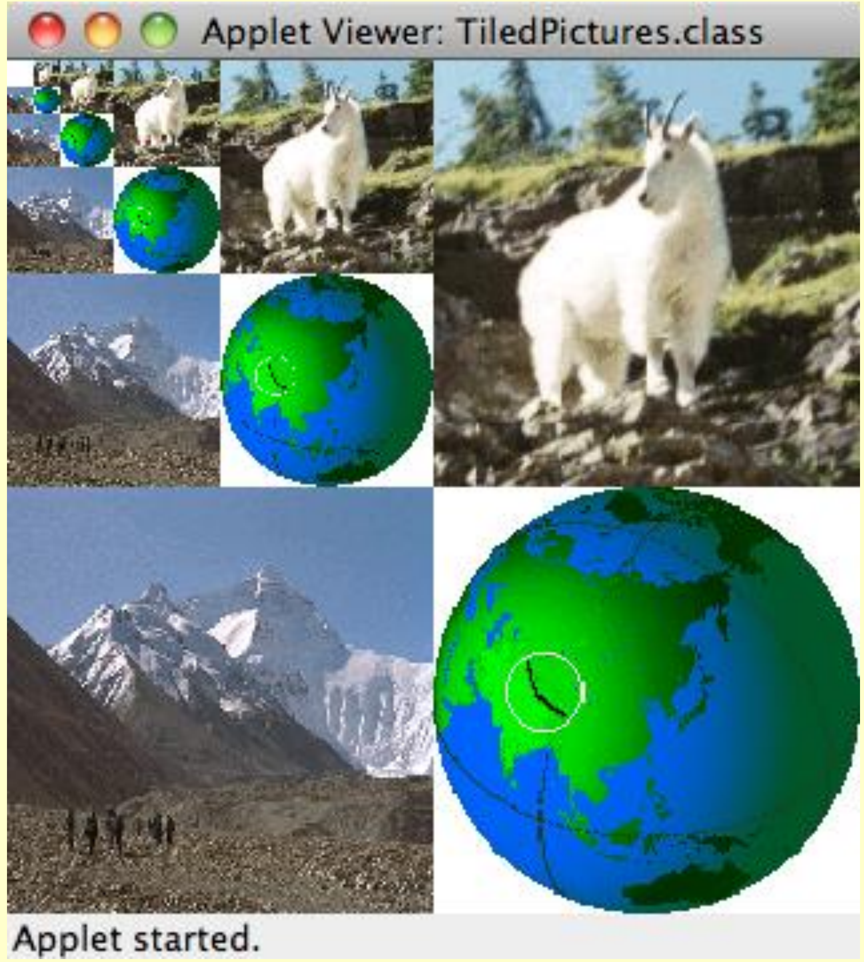
continue

continue

```
   //-----------------------------------------------------------------
   //  Performs the initial call to the drawPictures method.
   //-----------------------------------------------------------------
   public void paint(Graphics page)
   {
      drawPictures(APPLET_WIDTH, page);
   }
}
```

continue

```
   //---------
   //  Perform
   //---------
   public void
   {
      drawPict
   }
}
```



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`

```java
//********************************************************************
//  KochSnowflake.java       Author: Lewis/Loftus
//
//  Demonstrates the use of recursion in graphics.
//********************************************************************

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class KochSnowflake extends JApplet implements ActionListener
{
   private final int APPLET_WIDTH = 400;
   private final int APPLET_HEIGHT = 440;

   private final int MIN = 1, MAX = 9;

   private JButton increase, decrease;
   private JLabel titleLabel, orderLabel;
   private KochPanel drawing;
   private JPanel appletPanel, tools;
```

continue

continue

```
   //-----------------------------------------------------------------
   //  Sets up the components for the applet.
   //-----------------------------------------------------------------
   public void init()
   {
      tools = new JPanel();
      tools.setLayout(new BoxLayout(tools, BoxLayout.X_AXIS));
      tools.setPreferredSize(new Dimension(APPLET_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.setMargin(new Insets(0, 0, 0, 0));
      increase.addActionListener(this);

      decrease = new JButton(new ImageIcon("decrease.gif"));
      decrease.setPressedIcon(new ImageIcon("decreasePressed.gif"));
      decrease.setMargin(new Insets(0, 0, 0, 0));
      decrease.addActionListener(this);
```

continue

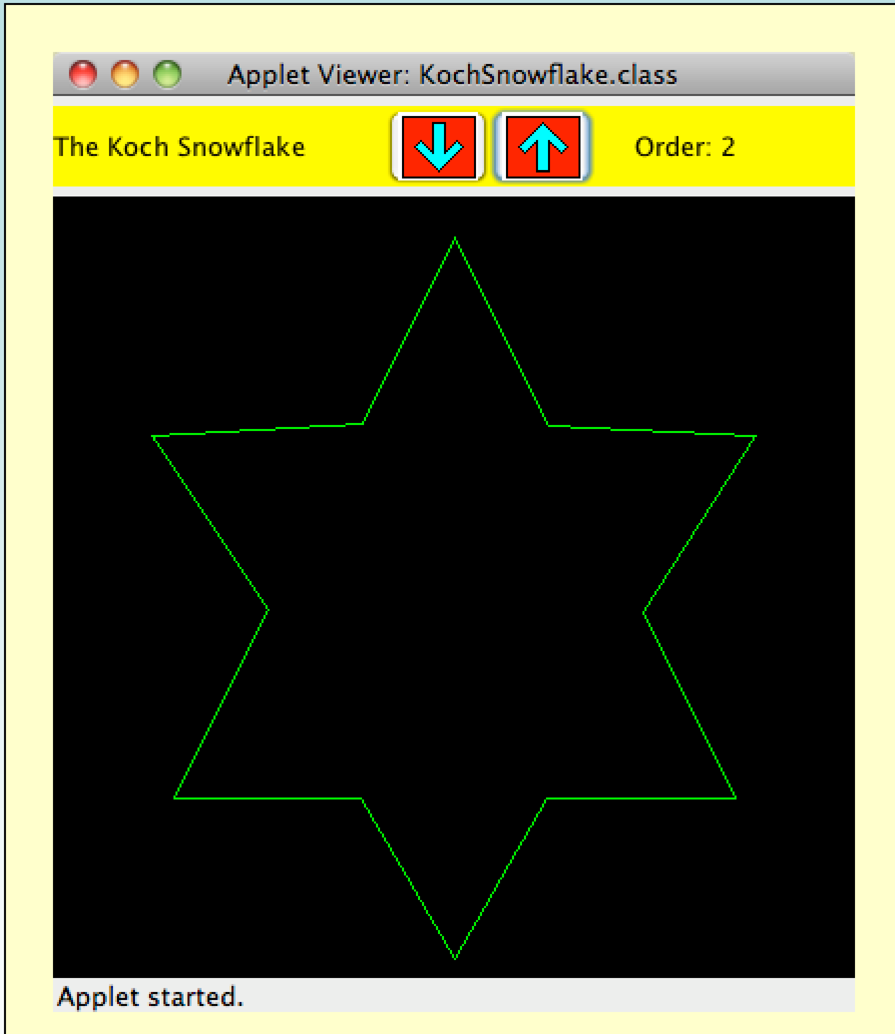
continue

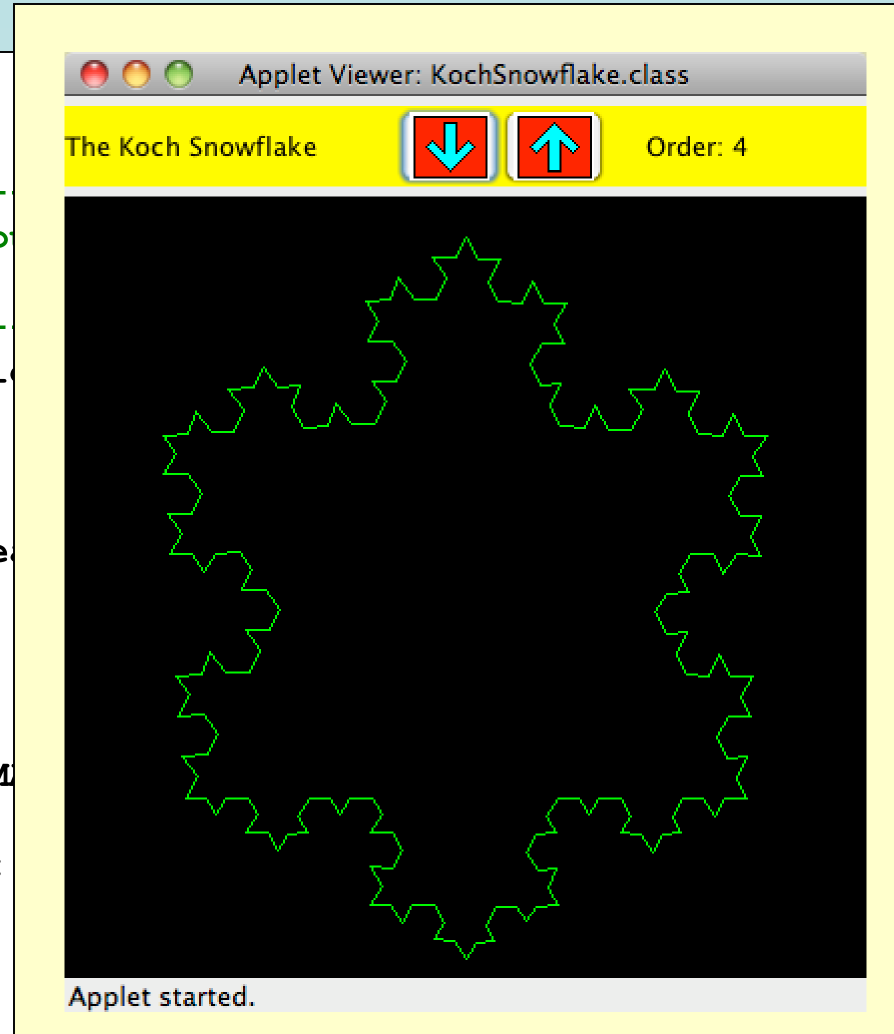
```
      orderLabel = new JLabel("Order: 1");
      orderLabel.setForeground(Color.black);

      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);

      appletPanel = new JPanel();
      appletPanel.add(tools);
      appletPanel.add(drawing);

      getContentPane().add(appletPanel);

      setSize(APPLET_WIDTH, APPLET_HEIGHT);
   }
```

continue

continue

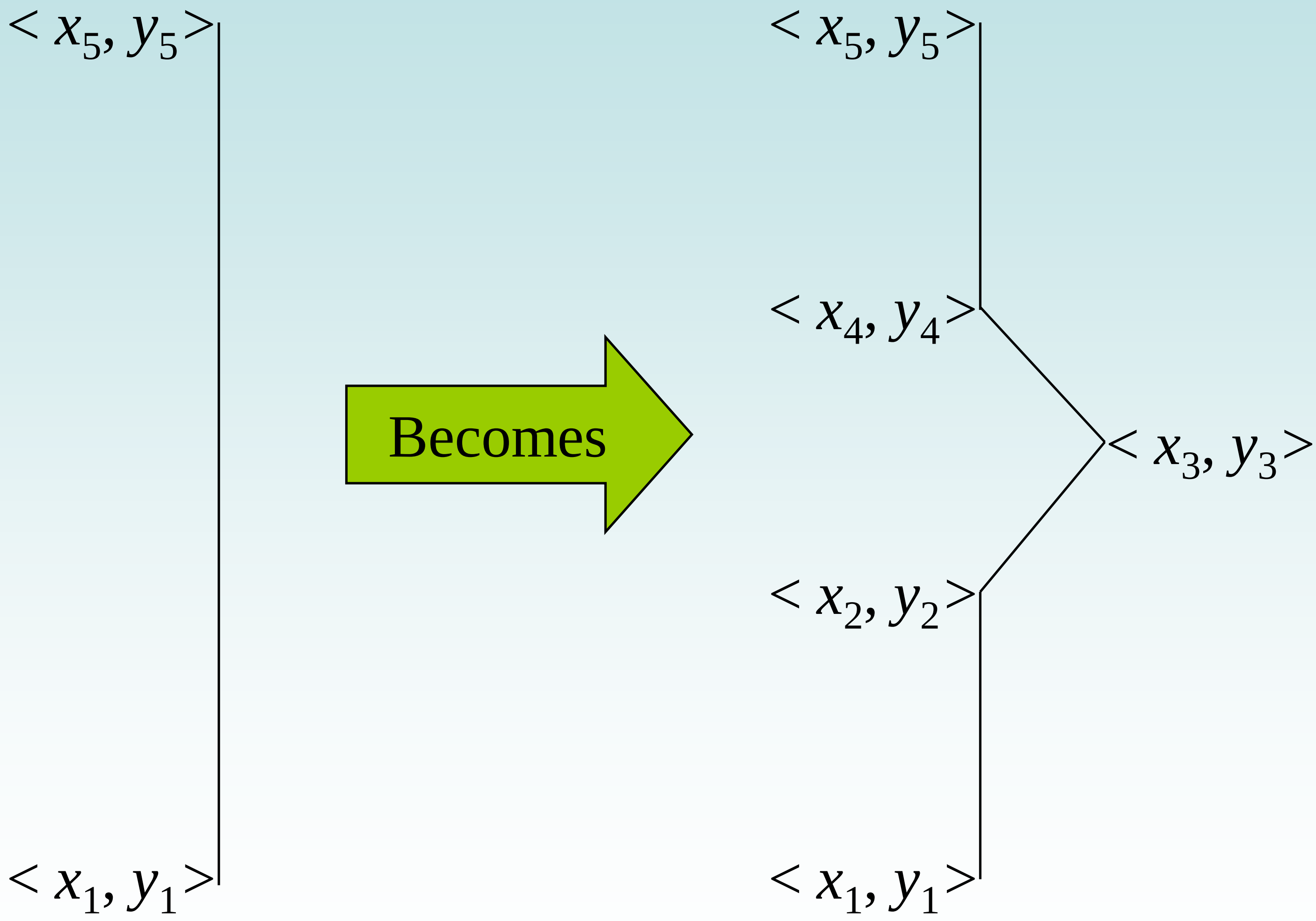
```
   //-----------------------------------------------------------------
   //  Determines which button was pushed, and sets the new order
   //  if it is in range.
   //-----------------------------------------------------------------
   public void actionPerformed(ActionEvent event)
   {
      int order = drawing.getOrder();

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

      if (order >= MIN && order <= MAX)
      {
         orderLabel.setText("Order: " + order);
         drawing.setOrder(order);
         repaint();
      }
   }
}
```


Applet Viewer: KochSnowflake.class
The Koch Snowflake
Order: 2
Applet started.


Applet Viewer: KochSnowflake.class
The Koch Snowflake
Order: 4
Applet started.

```
    }
}
```

Koch Snowflakes



```java
//********************************************************************
//  KochPanel.java       Author: Lewis/Loftus
//
//  Represents a drawing surface on which to paint a Koch Snowflake.
//********************************************************************

import java.awt.*;
import javax.swing.JPanel;

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
```

continue

continue

```
   //-----------------------------------------------------------------
   //  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);

      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