Chapter 18, Recursion

Java How to Program, 10/e

OBJECTIVES

In this chapter you'll:

- Learn the concept of recursion.
- Write and use recursive methods.
- Determine the base case and recursion step in a recursive algorithm.
- Learn how recursive method calls are handled by the system.
- Learn the differences between recursion and iteration, and when to use each.
- Learn what fractals are and how to draw them using recursion.
- Learn what recursive backtracking is and why it's an effective problem-solving technique.

18.1 Introduction

- For some problems, it's useful to have a method call itself.
  - Known as a recursive method.
  - Can call itself either directly or indirectly through another method.
- Figure 18.1 summarizes the recursion examples and exercises in this book.

Chapter	Recursion examples and exercises in this book
18	Factorial Method (Figs. 18.3 and 18.4) Fibonacci Method (Fig. 18.5) Towers of Hanoi (Fig. 18.11) Fractals (Figs. 18.18 and 18.19) What Does This Code Do? (Exercise 18.7, Exercise 18.12 and Exercise 18.13) Find the Error in the Following Code (Exercise 18.8) Raising an Integer to an Integer Power (Exercise 18.9) Visualizing Recursion (Exercise 18.10) Greatest Common Divisor (Exercise 18.11) Determine Whether a String Is a Palindrome (Exercise 18.14) Eight Queens (Exercise 18.15) Print an Array (Exercise 18.16) Print an Array Backward (Exercise 18.17) Minimum Value in an Array (Exercise 18.18) Star Fractal (Exercise 18.19) Maze Traversal Using Recursive Backtracking (Exercise 18.20) Generating Mazes Randomly (Exercise 18.21) Mazes of Any Size (Exercise 18.22) Time to Calculate a Fibonacci Number (Exercise 18.23)

Fig. 18.1 | Summary of the recursion examples and exercises in this text. (Part 1 of 2.)

Chapter	Recursion examples and exercises in this book
19	Merge Sort (Fig. 19.6) Linear Search (Exercise 19.8) Binary Search (Exercise 19.9) Quicksort (Exercise 19.10)
21	Binary-Tree Insert (Fig. 21.17) Preorder Traversal of a Binary Tree (Fig. 21.17) Inorder Traversal of a Binary Tree (Fig. 21.17) Postorder Traversal of a Binary Tree (Fig. 21.17) Print a Linked List Backward (Exercise 21.20) Search a Linked List (Exercise 21.21)

Fig. 18.1 | Summary of the recursion examples and exercises in this text. (Part 2 of 2.)

18.2 Recursion Concepts

- When a recursive method is called to solve a problem, it actually is capable of solving only the *simplest case(s)*, or base case(s).
  - If the method is called with a *base case*, it returns a result.
- If the method is called with a more complex problem, it divides the problem into two conceptual pieces
  - a piece that the method knows how to do and
  - a piece that it does not know how to do.
- To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or smaller version of it.
- Because this new problem resembles the original problem, the method calls a fresh copy of itself to work on the smaller problem
  - this is a recursive call
  - also called the recursion step

18.2 Recursion Concepts (cont.)

- The recursion step normally includes a `return` statement, because its result will be combined with the portion of the problem the method knew how to solve to form a result that will be passed back to the original caller.
- The recursion step executes while the original method call is still active.
- For recursion to eventually terminate, each time the method calls itself with a simpler version of the original problem, the sequence of smaller and smaller problems must converge on a base case.
  - When the method recognizes the base case, it returns a result to the previous copy of the method.
  - A sequence of returns ensues until the original method call returns the final result to the caller.

18.2 Recursion Concepts (cont.)

- A recursive method may call another method, which may in turn make a call back to the recursive method.
  - This is known as an indirect recursive call or indirect recursion.

18.3 Example Using Recursion: Factorials

- Factorial of a positive integer *n*, written *n*! (pronounced “n factorial”), which 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.
- The factorial of integer `number` (where `number` ≥ 0) can be calculated iteratively (nonrecursively) using a `for` statement as follows:
  - `factorial = 1;`
  - ```
    for ( int counter = number; counter >= 1; counter-- )
       factorial *= counter;
    ```
- Recursive declaration of the factorial calculation for integers greater than 1 is arrived at by observing the following relationship:
  - $n! = n \cdot (n-1)!$
- Figure 18.3 uses recursion to calculate and print the factorials of the integers from 0 through 21.
    ```

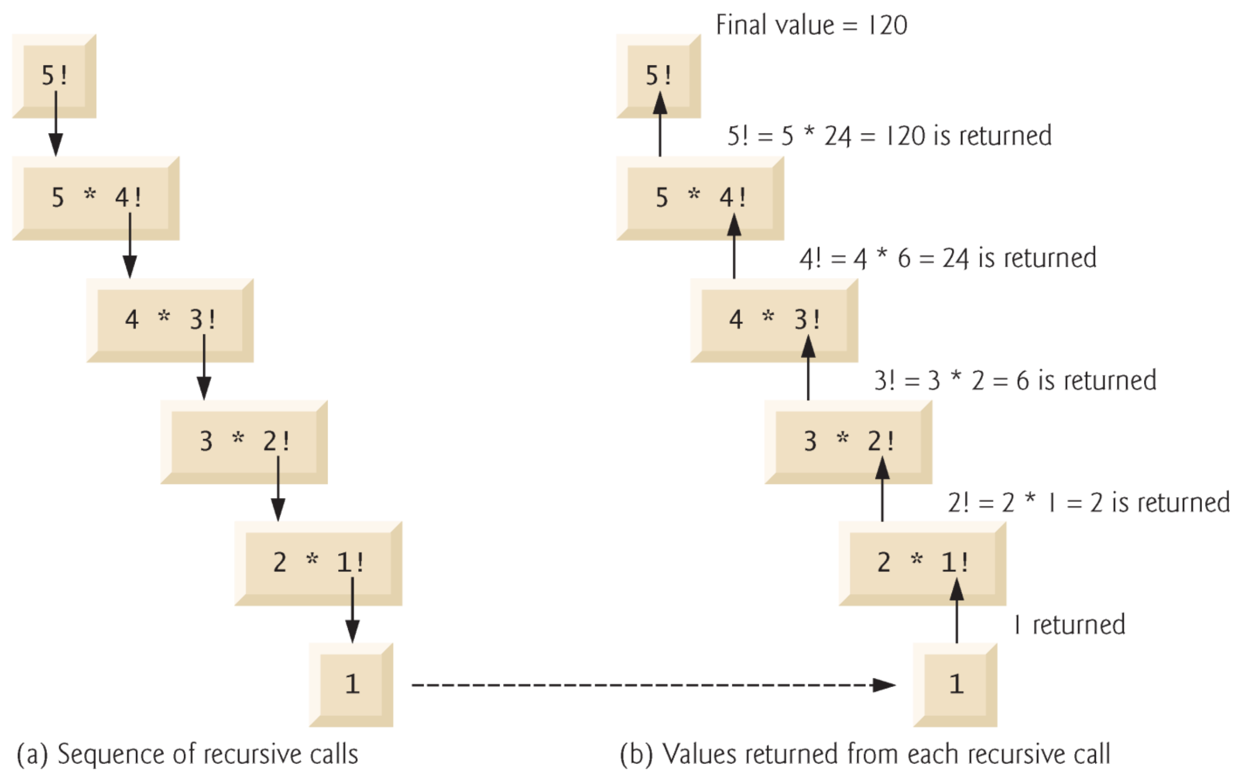

Fig. 18.2 | Recursive evaluation of 5!.

```java
// Fig. 18.3: FactorialCalculator.java
// Recursive factorial method.

public class FactorialCalculator
{
   // recursive method factorial (assumes its parameter is >= 0)
   public static 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);
   }

   // output factorials for values 0-21
   public static void main(String[] args)
   {
      // calculate the factorials of 0 through 21
      for (int counter = 0; counter <= 21; counter++)
         System.out.printf("%d! = %d%n", counter, factorial(counter));
   }
} // end class FactorialCalculator
```

Fig. 18.3 | Recursive factorial method. (Part 1 of 2.)

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

```
...
12! = 479001600 —— 12! causes overflow for int variables
...
20! = 2432902008176640000
21! = -4249290049419214848 —— 21! causes overflow for long variables
```

Fig. 18.3 | Recursive factorial method. (Part 2 of 2.)

Common Programming Error 18.1

Either omitting the base case or writing the recursion step incorrectly so that it does not converge on the base case can cause a logic error known as ***infinite recursion****, where recursive calls are continuously made until memory is exhausted or the method-call stack overflows. This error is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.*

18.3 Example Using Recursion: Factorials (Cont.)

- We use type `long` so the program can calculate factorials greater than 12!.
- The `factorial` method produces large values so quickly that we exceed the largest `long` value when we attempt to calculate 21!.
- Package `java.math` provides classes BigInteger and BigDecimal explicitly for arbitrary precision calculations that cannot be performed with primitive types.

18.4 Reimplementing Class `FactorialCalculator` Using Class `BigInteger`

- Figure 18.4 reimplements class `FactorialCalculator` using `BigInteger` variables.
- To demonstrate larger values than what `long` variables can store, we calculate the factorials of the numbers 0–50.

```java
// Fig. 18.4: FactorialCalculator.java
// Recursive factorial method.
import java.math.BigInteger;

public class FactorialCalculator
{
   // recursive method factorial (assumes its parameter is >= 0)
   public static BigInteger factorial(BigInteger number)
   {
      if (number.compareTo(BigInteger.ONE) <= 0) // test base case
         return BigInteger.ONE; // base cases: 0! = 1 and 1! = 1
      else // recursion step
         return number.multiply(
            factorial(number.subtract(BigInteger.ONE)));
   }

   // output factorials for values 0-50
   public static void main(String[] args)
   {
      // calculate the factorials of 0 through 50
      for (int counter = 0; counter <= 50; counter++)
         System.out.printf("%d! = %d%n", counter,
            factorial(BigInteger.valueOf(counter)));
   }
} // end class FactorialCalculator
```

Fig. 18.4 | Factorial calculations with a recursive method. (Part 1 of 2.)

```
0! = 1
1! = 1
2! = 2
3! = 6
...
21! = 51090942171709440000 —— 21! and larger values no longer cause overflow
22! = 1124000727777607680000
...
47! = 258623241511168180642964355153611979969197632389120000000000
48! = 12413915592536072670862289047373375038521486354677760000000000
49! = 608281864034267560872252163321295376887552831379210240000000000
50! = 30414093201713378043612608166064768844377641568960512000000000000
```

Fig. 18.4 | Factorial calculations with a recursive method. (Part 2 of 2.)

18.4 Reimplementing Class FactorialCalculator Using Class BigInteger (Cont.)

- BigInteger method compareTo compares the BigInteger number that calls the method to the method's BigInteger argument.
  - Returns -1 if the BigIteger that calls the method is less than the argument, 0 if they are equal or 1 if the BigInteger that calls the method is greater than the argument.
- BigInteger constant ONE represents the integer value 1.
- BigInteger methods multiply and subtract implement multiplication and subtraction. Similar methods are provided for other arithmetic operations

18.5 Example Using Recursion: Fibonacci Series

- The Fibonacci series, begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two.
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, …
- This series occurs in nature and describes a form of spiral.
- The ratio of successive Fibonacci numbers converges on a constant value of 1.618…,
  - called the golden ratio or the golden mean.
- The Fibonacci series may be defined recursively as follows:
  - fibonacci(0) = 0
    fibonacci(1) = 1
    fibonacci(*n*) = *fibonacci*(*n* – 1) + *fibonacci*(*n* – 2)

18.5 Example Using Recursion: Fibonacci Series (cont.)

- *Two base cases for*
  - `fibonacci(0)` is defined to be 0
  - `fibonacci(1)` to be 1
- Fibonacci numbers tend to become large quickly.
  - We use type `BigInteger` as the parameter type and the return type of method `fibonacci`.

```java
// Fig. 18.5: FibonacciCalculator.java
// Recursive fibonacci method.
import java.math.BigInteger;

public class FibonacciCalculator
{
   private static BigInteger TWO = BigInteger.valueOf(2);

   // recursive declaration of method fibonacci
   public static BigInteger fibonacci(BigInteger number)
   {
      if (number.equals(BigInteger.ZERO) ||
           number.equals(BigInteger.ONE)) // base cases
         return number;
      else // recursion step
         return fibonacci(number.subtract(BigInteger.ONE)).add(
            fibonacci(number.subtract(TWO)));
   }

```

Fig. 18.5 | Recursive fibonacci method. (Part 1 of 3.)

```java
    // displays the fibonacci values from 0-40
    public static void main(String[] args)
    {
       for (int counter = 0; counter <= 40; counter++)
          System.out.printf("Fibonacci of %d is: %d%n", counter,
            fibonacci(BigInteger.valueOf(counter)));
    }
 } // end class FibonacciCalculator
```

Fig. 18.5 | Recursive fibonacci method. (Part 2 of 3.)

```
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
...
Fibonacci of 37 is: 24157817
Fibonacci of 38 is: 39088169
Fibonacci of 39 is: 63245986
Fibonacci of 40 is: 102334155
```

Fig. 18.5 | Recursive fibonacci method. (Part 3 of 3.)

18.5 Example Using Recursion: Fibonacci Series (cont.)

- `BigInteger` constants `ZERO` and `ONE` represent the values 0 and 1, respectively.
- Iif `number` is greater than 1, the recursion step generates *two* recursive calls, each for a slightly smaller problem than the original call to `fibonacci`.
- `BigInteger` methods `add` and `subtract` are used to help implement the recursive step.

18.5 Example Using Recursion: Fibonacci Series (cont.)

- Figure 18.6 shows how method `fibonacci` evaluates `fibonacci(3)`.
- The Java language specifies that the order of evaluation of the operands is from left to right.
- Thus, the call `fibonacci(2)` is made first and the call `fibonacci(1)` second.
- Each invocation of the `fibonacci` method that does not match one of the base cases (0 or 1) results in two more recursive calls to the `fibonacci` method.
- Calculating the Fibonacci value of 20 with the program in Fig. 18.5 requires 21,891 calls to the `fibonacci` method; calculating the Fibonacci value of 30 requires 2,692,537 calls!

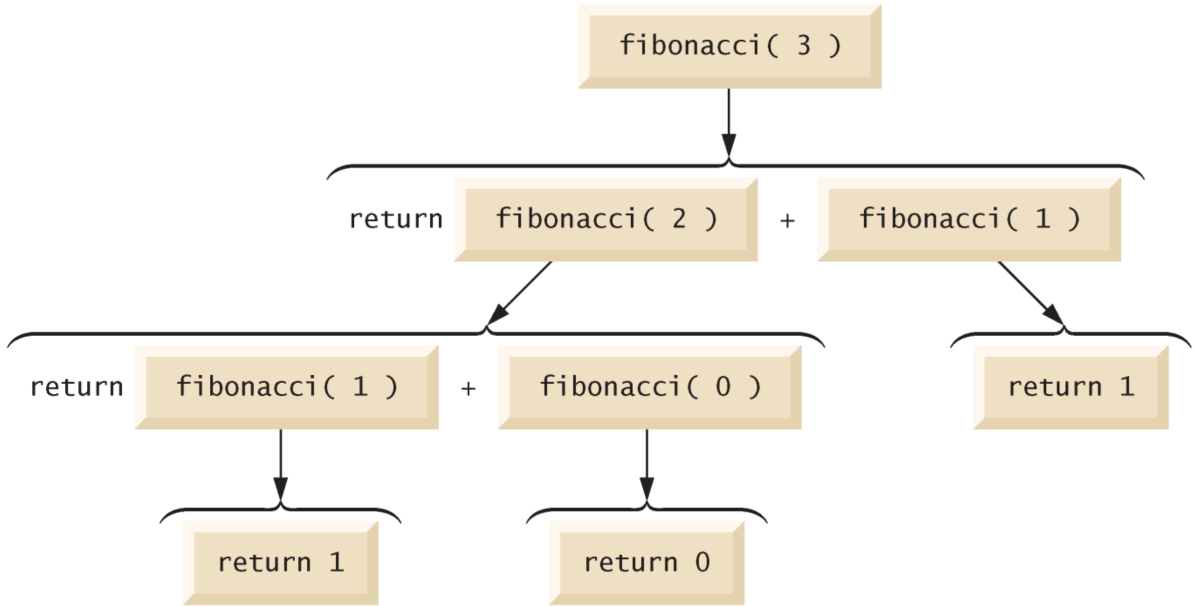

Fig. 18.6 | Set of recursive calls for `fibonacci`(3).

Performance Tip 18.1

Avoid Fibonacci-style recursive programs, because they result in an exponential "explosion" of method calls.

18.6 Recursion and the Method-Call Stack

- The *method-call stack* and *stack frames* keep track of recursive method calls.

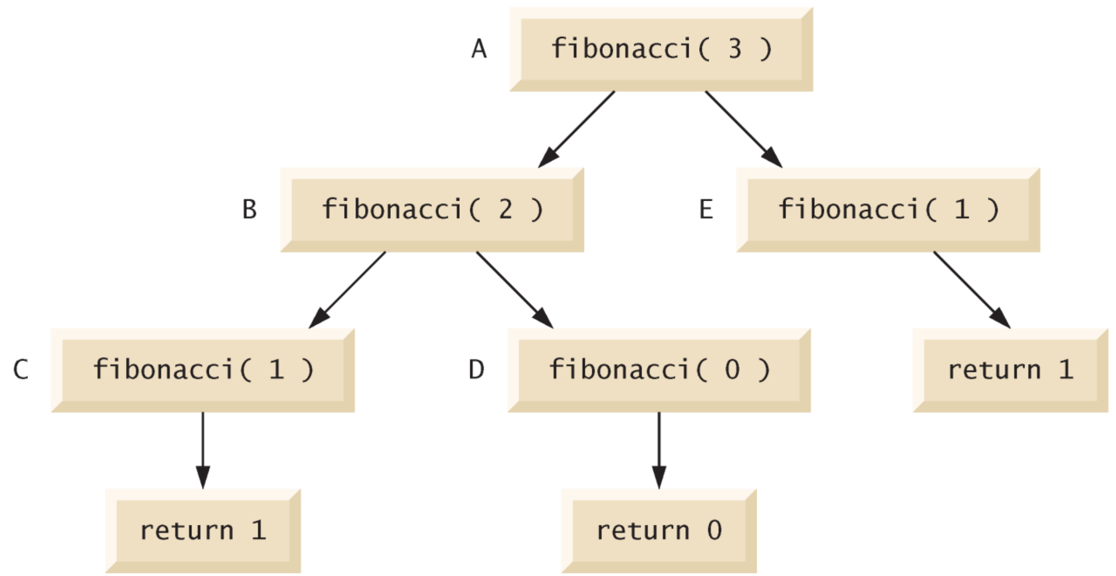

Fig. 18.7 | Method calls made within the call `fibonacci(3)`.

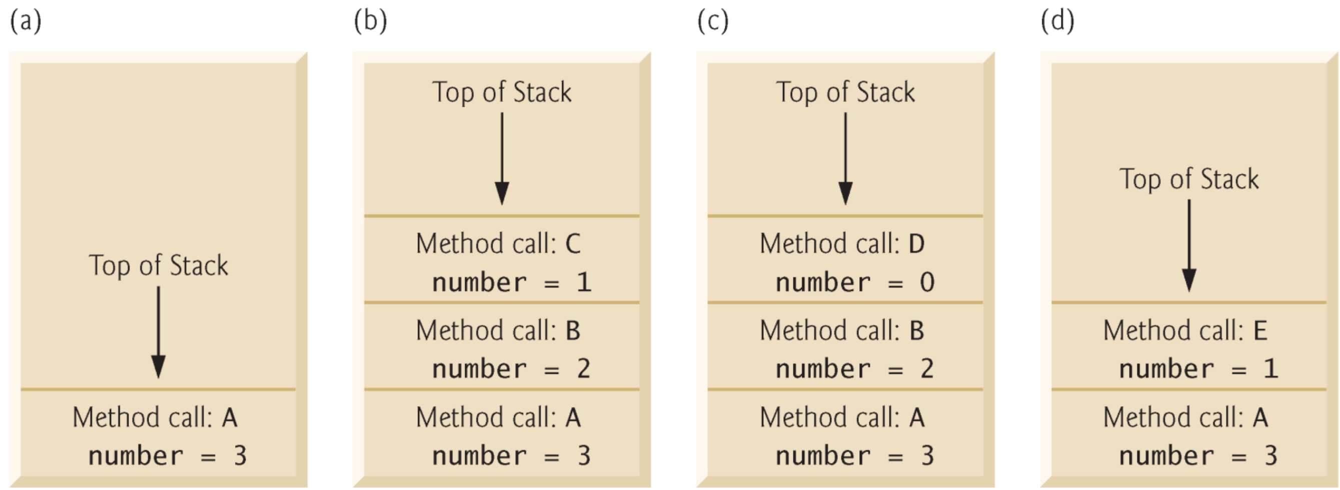

Fig. 18.8 | Method calls on the program-execution stack.

18.7 Recursion vs. Iteration

- Both iteration and recursion are *based on a control statement*:
  - Iteration uses a repetition statement (e.g., `for`, `while` or `do…while`)
  - Recursion uses a selection statement (e.g., `if`, `if…else` or `switch`)
- Both iteration and recursion involve *repetition*:
  - Iteration explicitly uses a repetition statement
  - Recursion achieves repetition through repeated method calls
- Iteration and recursion each involve a *termination test*:
  - Iteration terminates when the loop-continuation condition fails
  - Recursion terminates when a base case is reached.

18.7 Recursion vs. Iteration (cont.)

- Both iteration and recursion can occur infinitely:
  - An infinite loop occurs with iteration if the loop-continuation test never becomes false
  - Infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case, or if the base case is not tested.

```java
// Fig. 18.9: 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;
   }

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

Fig. 18.9 | Iterative factorial method. (Part 1 of 2.)

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

Fig. 18.9 | Iterative factorial method. (Part 2 of 2.)

18.7 Recursion vs. Iteration (cont.)

- Recursion repeatedly invokes the mechanism, and consequently the overhead, of method calls.
  - Can be expensive in terms of both processor time and memory space.
- Each recursive call causes another copy of the method (actually, only the method's variables, stored in the activation record) to be created
  - this set of copies can consume considerable memory space.
- Since iteration occurs within a method, repeated method calls and extra memory assignment are avoided.

Software Engineering Observation 18.1

Any problem that can be solved recursively can also be solved iteratively. 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.

Performance Tip 18.2

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

Common Programming Error 18.2

Accidentally having a nonrecursive method call itself either directly or indirectly through another method can cause infinite recursion.

18.8 Towers of Hanoi

- Towers of Hanoi
  - Move stack of disks from one peg to another under the constraints that exactly one disk is moved at a time and at no time may a larger disk be placed above a smaller disk.
  - Three pegs are provided, one being used for temporarily holding disks.
- Moving n disks can be viewed in terms of moving only n – 1 disks (hence the recursion) as follows:
  - 1.Move *n* – 1 disks from peg 1 to peg 2, using peg 3 as a temporary holding area.
  - 2.Move the last disk (the largest) from peg 1 to peg 3.
  - 3.Move *n* – 1 disks from peg 2 to peg 3, using peg 1 as a temporary holding area.
- The process ends when the last task involves moving *n* = 1 disk (i.e., the base case). This task is accomplished by moving the disk, without using a temporary holding area.

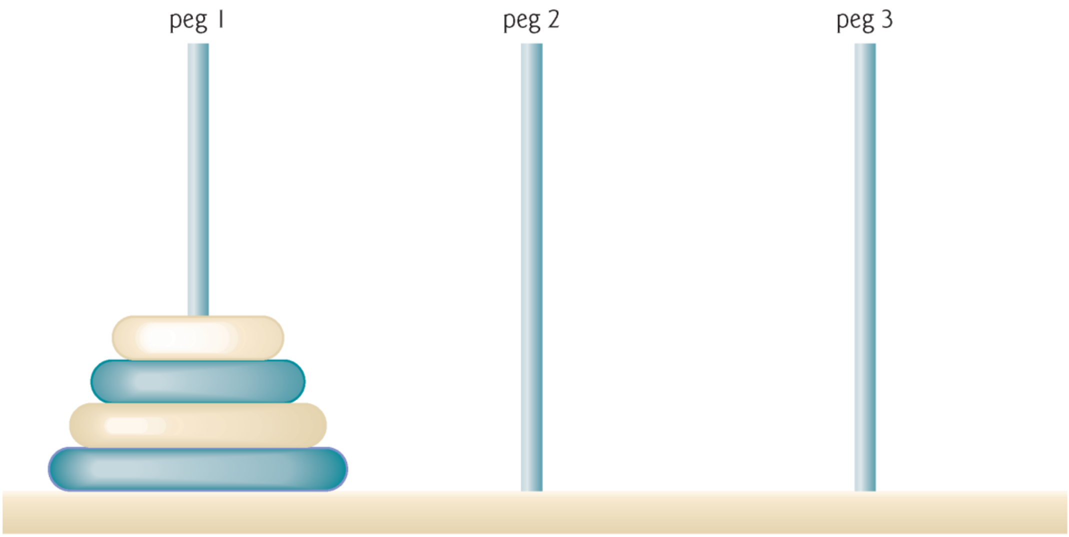

Fig. 18.10 | Towers of Hanoi for the case with four disks.

```java
// Fig. 18.11: TowersOfHanoi.java
// Towers of Hanoi solution with a recursive method.
public class TowersOfHanoi
{
   // recursively move disks between towers
   public static void solveTowers(int disks, int sourcePeg,
      int destinationPeg, int tempPeg)
   {
      // base case -- only one disk to move
      if (disks == 1)
      {
         System.out.printf("%n%d --> %d", sourcePeg, destinationPeg);
         return;
      }

      // recursion step -- move (disk - 1) disks from sourcePeg
      // to tempPeg using destinationPeg
      solveTowers(disks - 1, sourcePeg, tempPeg, destinationPeg);

      // move last disk from sourcePeg to destinationPeg
      System.out.printf("%n%d --> %d", sourcePeg, destinationPeg);

```

Fig. 18.11 | Towers of Hanoi solution with a recursive method. (Part 1 of 2.)

```java
      // move (disks - 1) disks from tempPeg to destinationPeg
      solveTowers(disks - 1, tempPeg, destinationPeg, sourcePeg);
   }

   public static void main(String[] args)
   {
      int startPeg = 1; // value 1 used to indicate startPeg in output
      int endPeg = 3; // value 3 used to indicate endPeg in output
      int tempPeg = 2; // value 2 used to indicate tempPeg in output
      int totalDisks = 3; // number of disks

      // initial nonrecursive call: move all disks.
      solveTowers(totalDisks, startPeg, endPeg, tempPeg);
   }
} // end class TowersOfHanoi
```

```
1 --> 3
1 --> 2
3 --> 2
1 --> 3
2 --> 1
2 --> 3
1 --> 3
```

Fig. 18.11 | Towers of Hanoi solution with a recursive method. (Part 2 of 2.)

18.9 Fractals

- A fractal is a geometric figure that can be generated from a pattern repeated recursively (Fig. 18.12).
- The figure is modified by recursively applying the pattern to each segment of the original figure.
- Fractals have a self-similar property—when subdivided into parts, each resembles a reduced-size copy of the whole.
- Many fractals yield an exact copy of the original when a portion of the fractal is magnified—such a fractal is said to be strictly self-similar.
- See our Recursion Resource Center (`www.deitel.com/recursion/`) for websites that demonstrate fractals.

18.9.1 Koch Curve Fractal

- Strictly self-similar Koch Curve fractal (Fig. 18.12).
  - It is formed by removing the middle third of each line in the drawing and replacing it with two lines that form a point, such that if the middle third of the original line remained, an equilateral triangle would be formed.
- Formulas for creating fractals often involve removing all or part of the previous fractal image.
- Start with a straight line (Fig. 18.12(a)) and apply the pattern, creating a triangle from the middle third (Fig. 18.12(b)).
- Then apply the pattern again to each straight line, resulting in Fig. 18.12(c).
- Each time the pattern is applied, the fractal is at a new level, or depth (sometimes the term order is also used).
- After only a few iterations, this fractal begins to look like a portion of a snowflake (Fig. 18.12(e and f)).

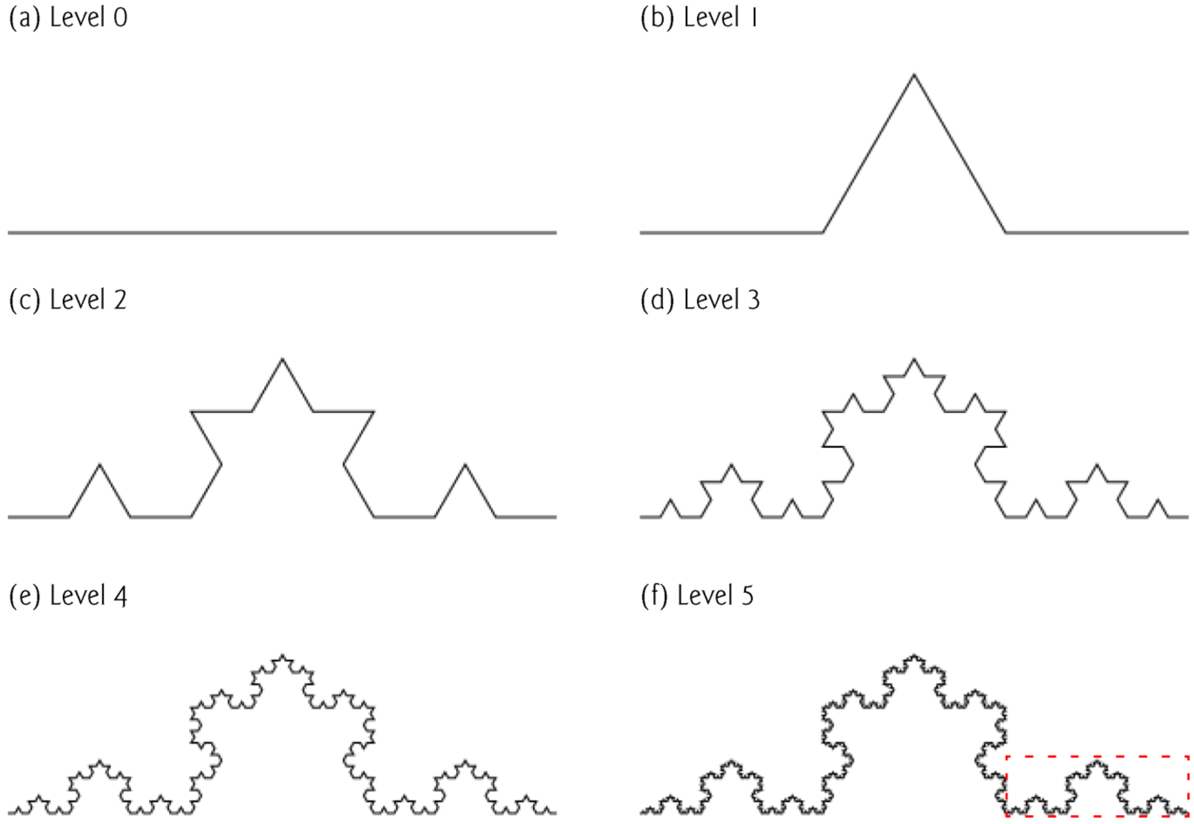

Fig. 18.12 | Koch Curve fractal.

18.9.1 Koch Curve Fractal (cont.)

- The Koch Snowflake fractal is similar to the Koch Curve but begins with a triangle rather than a line.
- The same pattern is applied to each side of the triangle, resulting in an image that looks like an enclosed snowflake.

18.9.2 (Optional) Case Study: Lo Feather Fractal

- ***The "Lo Feather Fractal"***
  - Program to create a strictly self-similar fractal.
  - Named for Sin Han Lo, a Deitel & Associates colleague who created it.
- The fractal will eventually resemble one-half of a feather (see the outputs in Fig. 18.19).
- The base case, or fractal level of 0, begins as a line between two points, A and B (Fig. 18.13).
- To create the next higher level, we find the midpoint (C) of the line.
- To calculate the location of point C, use the following formula:

```
xC = (xA + xB) / 2;
yC = (yA + yB) / 2;
```

18.9.2 (Optional) Case Study: Lo Feather Fractal (cont.)

- To create this fractal, we also must find a point D that lies left of segment AC and creates an isosceles right triangle ADC.
- To calculate point D's location, use the following formulas:
  - ```
    xD = xA + (xC - xA) / 2 - (yC - yA) / 2;
    yD = yA + (yC - yA) / 2 + (xC - xA) / 2;
    ```
- We now move from level 0 to level 1 as follows: First, add points C and D (as in Fig. 18.14).
- Then, remove the original line and add segments DA, DC and DB.
- The remaining lines will curve at an angle, causing our fractal to look like a feather.
- For the next level of the fractal, this algorithm is repeated on each of the three lines in level 1.
- For each line, the formulas above are applied, where the former point D is now considered to be point A, while the other end of each line is considered to be point B.
    ```

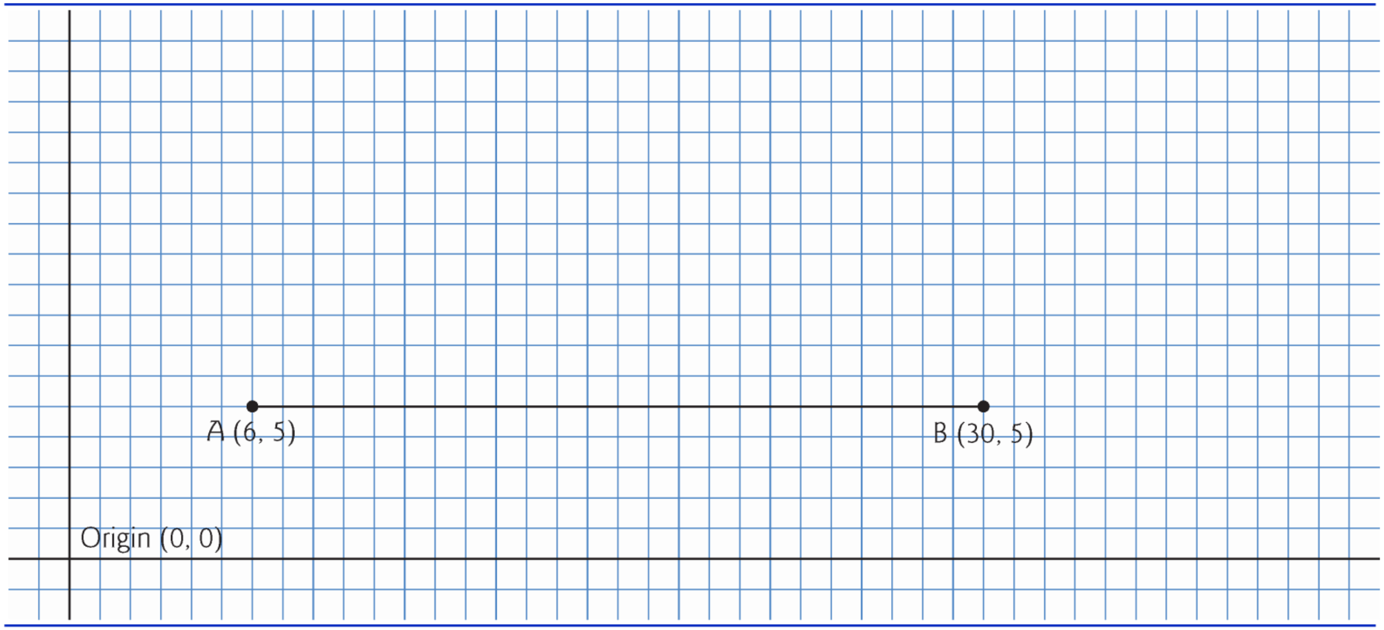

Fig. 18.13 | "Lo feather fractal" at level 0.

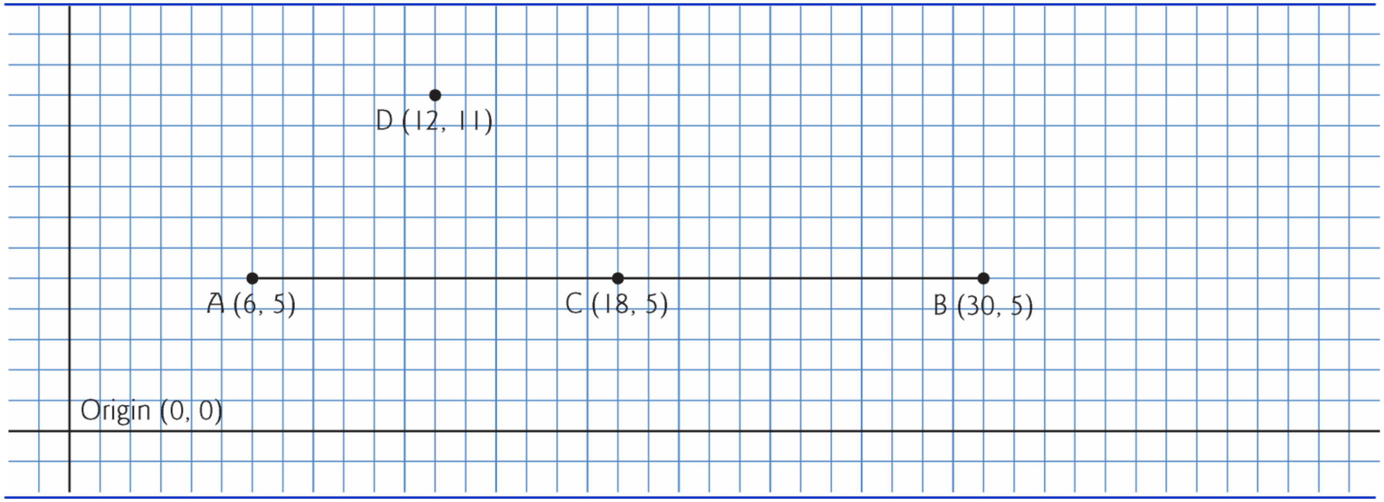

Fig. 18.14 | Determining points C and D for level 1 of the "Lo feather fractal."

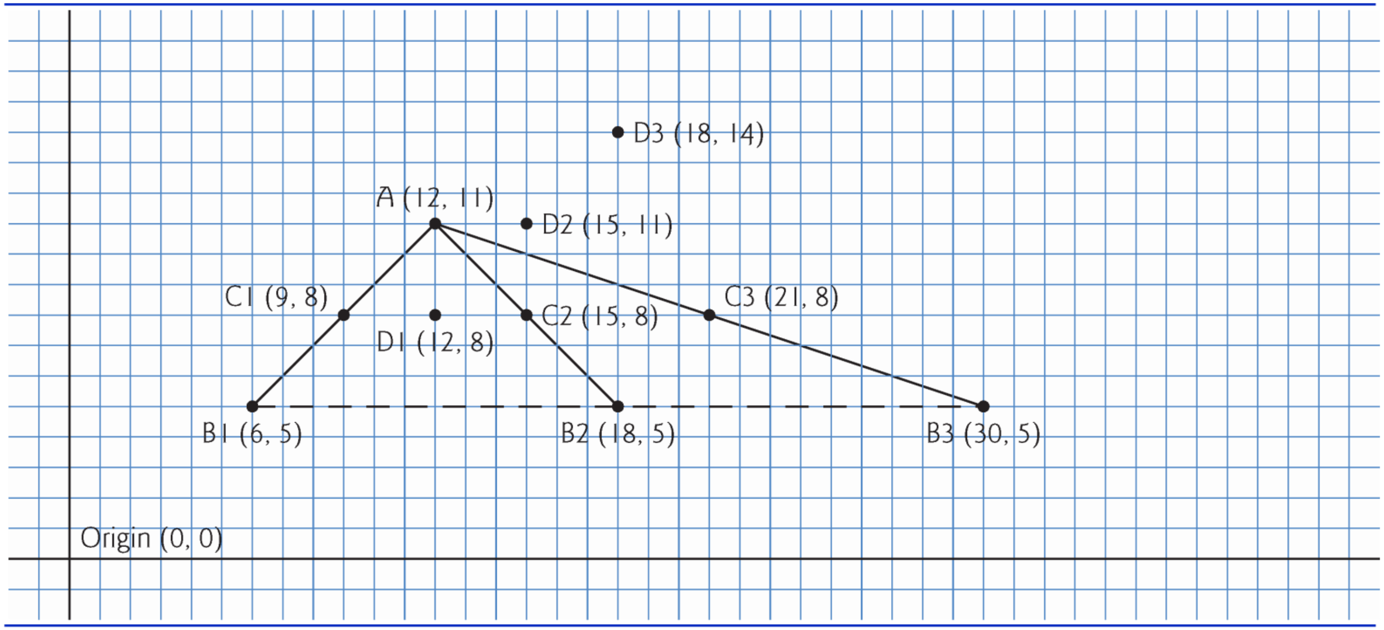

Fig. 18.15 | "Lo feather fractal" at level 1, with C and D points determined for level 2. [*Note:* The fractal at level 0 is included as a dashed line as a reminder of where the line was located in relation to the current fractal.]

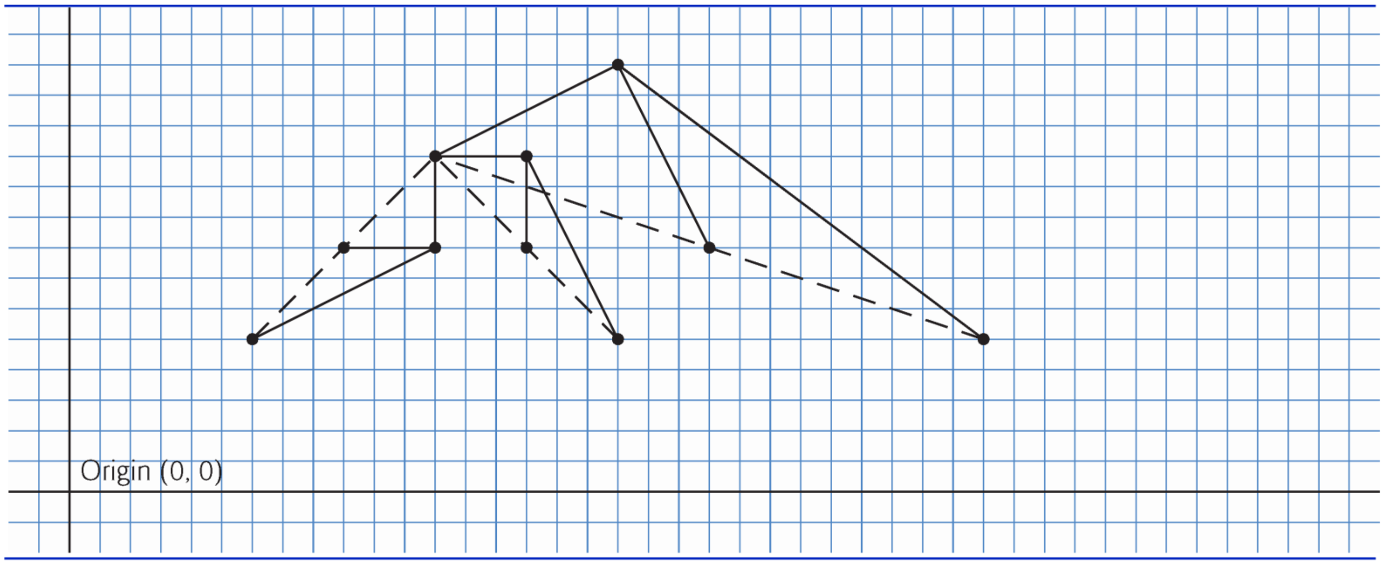

Fig. 18.16 | "Lo feather fractal" at level 2, with dashed lines from level 1 provided.

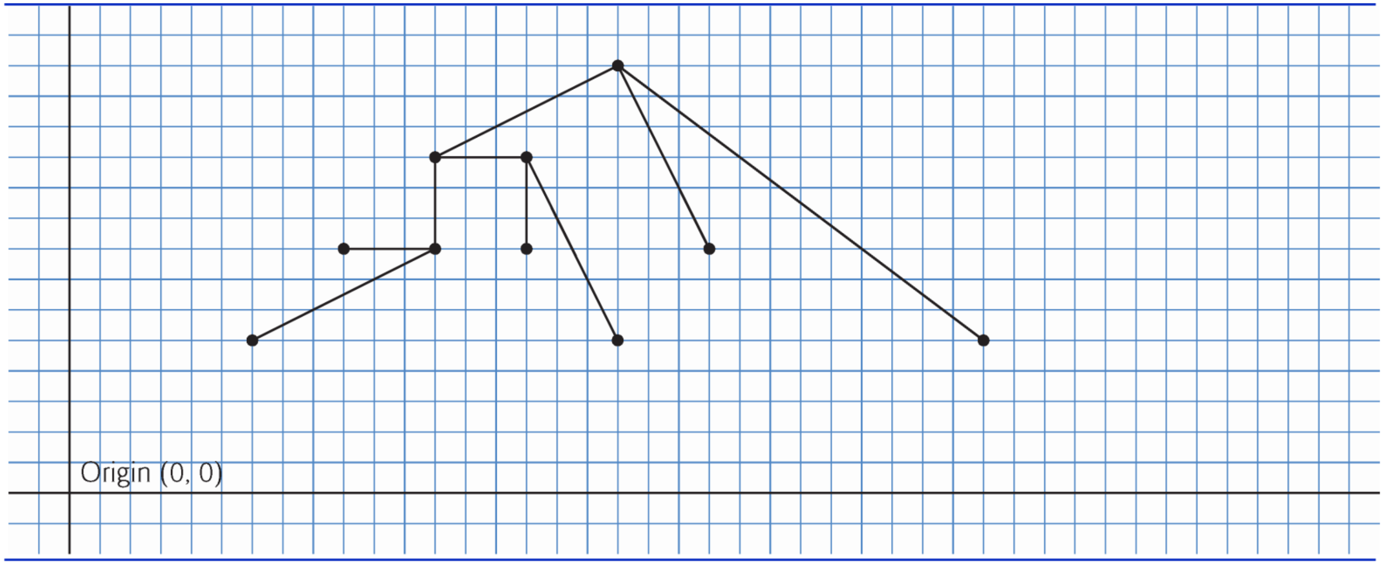

Fig. 18.17 | "Lo feather fractal" at level 2.

```java
// Fig. 18.18: Fractal.java
// Fractal user interface.
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JColorChooser;

public class Fractal extends JFrame
{
   private static final int WIDTH = 400;  // define width of GUI
   private static final int HEIGHT = 480; // define height of GUI
   private static final int MIN_LEVEL = 0;
   private static final int MAX_LEVEL = 15;

   // set up GUI
   public Fractal()
   {
      super("Fractal");

```

Fig. 18.18 | Fractal user interface. (Part 1 of 6.)

```java
      // set up levelJLabel to add to controlJPanel
      final JLabel levelJLabel = new JLabel("Level: 0");

      final FractalJPanel drawSpace = new FractalJPanel(0);

      // set up control panel
      final JPanel controlJPanel = new JPanel();
      controlJPanel.setLayout(new FlowLayout());

      // set up color button and register listener
      final JButton changeColorJButton = new JButton("Color");
      controlJPanel.add(changeColorJButton);
      changeColorJButton.addActionListener(
         new ActionListener() // anonymous inner class
         {
            // process changeColorJButton event
            @Override
            public void actionPerformed(ActionEvent event)
            {
               Color color = JColorChooser.showDialog(
                  Fractal.this, "Choose a color", Color.BLUE);

```

Fig. 18.18 | Fractal user interface. (Part 2 of 6.)

```java
                // set default color, if no color is returned
                if (color == null)
                   color = Color.BLUE;

                drawSpace.setColor(color);
             }
          } // end anonymous inner class
       ); // end addActionListener

       // set up decrease level button to add to control panel and
       // register listener
       final JButton decreaseLevelJButton = new JButton("Decrease Level");
       controlJPanel.add(decreaseLevelJButton);
       decreaseLevelJButton.addActionListener(
          new ActionListener() // anonymous inner class
          {
             // process decreaseLevelJButton event
             @Override
             public void actionPerformed(ActionEvent event)
             {
                int level = drawSpace.getLevel();
                --level;

```

Fig. 18.18 | Fractal user interface. (Part 3 of 6.)

```
                // modify level if possible
                if ((level >= MIN_LEVEL)) &&
                   (level <= MAX_LEVEL))
                {
                   levelJLabel.setText("Level: " + level);
                   drawSpace.setLevel(level);
                   repaint();
                }
             }
          } // end anonymous inner class
       ); // end addActionListener

```

Fig. 18.18 | Fractal user interface. (Part 4 of 6.)

```
        // set up increase level button to add to control panel
        // and register listener
        final JButton increaseLevelJButton = new JButton("Increase Level");
        controlJPanel.add(increaseLevelJButton);
        increaseLevelJButton.addActionListener(
           new ActionListener() // anonymous inner class
           {
              // process increaseLevelJButton event
              @Override
              public void actionPerformed(ActionEvent event)
              {
                 int level = drawSpace.getLevel();
                 ++level;

                 // modify level if possible
                 if ((level >= MIN_LEVEL)) &&
                    (level <= MAX_LEVEL))
                 {
                    levelJLabel.setText("Level: " + level);
                    drawSpace.setLevel(level);
                    repaint();
                 }
              }
           } // end anonymous inner class
        ); // end addActionListener
```

Fig. 18.18 | Fractal user interface. (Part 5 of 6.)

```
      controlJPanel.add(levelJLabel);

      // create mainJPanel to contain controlJPanel and drawSpace
      final JPanel mainJPanel = new JPanel();
      mainJPanel.add(controlJPanel);
      mainJPanel.add(drawSpace);

      add(mainJPanel); // add JPanel to JFrame

      setSize(WIDTH, HEIGHT); // set size of JFrame
      setVisible(true); // display JFrame
   } // end Fractal constructor

   public static void main(String[] args)
   {
      Fractal demo = new Fractal();
      demo.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
} // end class Fractal
```

Fig. 18.18 | Fractal user interface. (Part 6 of 6.)

```java
// Fig. 18.19: FractalJPanel.java
// Drawing the "Lo feather fractal" using recursion.
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Dimension;
import javax.swing.JPanel;

public class FractalJPanel extends JPanel
{
   private Color color; // stores color used to draw fractal
   private int level;   // stores current level of fractal

   private static final int WIDTH = 400;  // defines width of JPanel
   private static final int HEIGHT = 400; // defines height of JPanel

   // set the initial fractal level to the value specified
   // and set up JPanel specifications
   public FractalJPanel(int currentLevel)
   {
      color = Color.BLUE;  // initialize drawing color to blue
      level = currentLevel; // set initial fractal level
      setBackground(Color.WHITE);
      setPreferredSize(new Dimension(WIDTH, HEIGHT));
   }
```

Fig. 18.19 | Drawing the "Lo feather fractal" using recursion. (Part 1 of 8.)

```java

   // draw fractal recursively
   public void drawFractal(int level, int xA, int yA, int xB,
      int yB, Graphics g)
   {
      // base case: draw a line connecting two given points
      if (level == 0)
         g.drawLine(xA, yA, xB, yB);
      else // recursion step: determine new points, draw next level
      {
         // calculate midpoint between (xA, yA) and (xB, yB)
         int xC = (xA + xB) / 2;
         int yC = (yA + yB) / 2;

         // calculate the fourth point (xD, yD) which forms an
         // isosceles right triangle between (xA, yA) and (xC, yC)
         // where the right angle is at (xD, yD)
         int xD = xA + (xC - xA) / 2 - (yC - yA) / 2;
         int yD = yA + (yC - yA) / 2 + (xC - xA) / 2;

```

Fig. 18.19 | Drawing the "Lo feather fractal" using recursion. (Part 2 of 8.)

```
      // recursively draw the Fractal
      drawFractal(level - 1, xD, yD, xA, yA, g);
      drawFractal(level - 1, xD, yD, xC, yC, g);
      drawFractal(level - 1, xD, yD, xB, yB, g);
   }
}

// start drawing the fractal
@Override
public void paintComponent(Graphics g)
{
   super.paintComponent(g);

   // draw fractal pattern
   g.setColor(color);
   drawFractal(level, 100, 90, 290, 200, g);
}

// set the drawing color to c
public void setColor(Color c)
{
   color = c;
}

```

Fig. 18.19 | Drawing the "Lo feather fractal" using recursion. (Part 3 of 8.)

```
    // set the new level of recursion
    public void setLevel(int currentLevel)
    {
       level = currentLevel;
    }

    // returns level of recursion
    public int getLevel()
    {
       return level;
    }
} // end class FractalJPanel
```

Fig. 18.19 | Drawing the "Lo feather fractal" using recursion. (Part 4 of 8.)

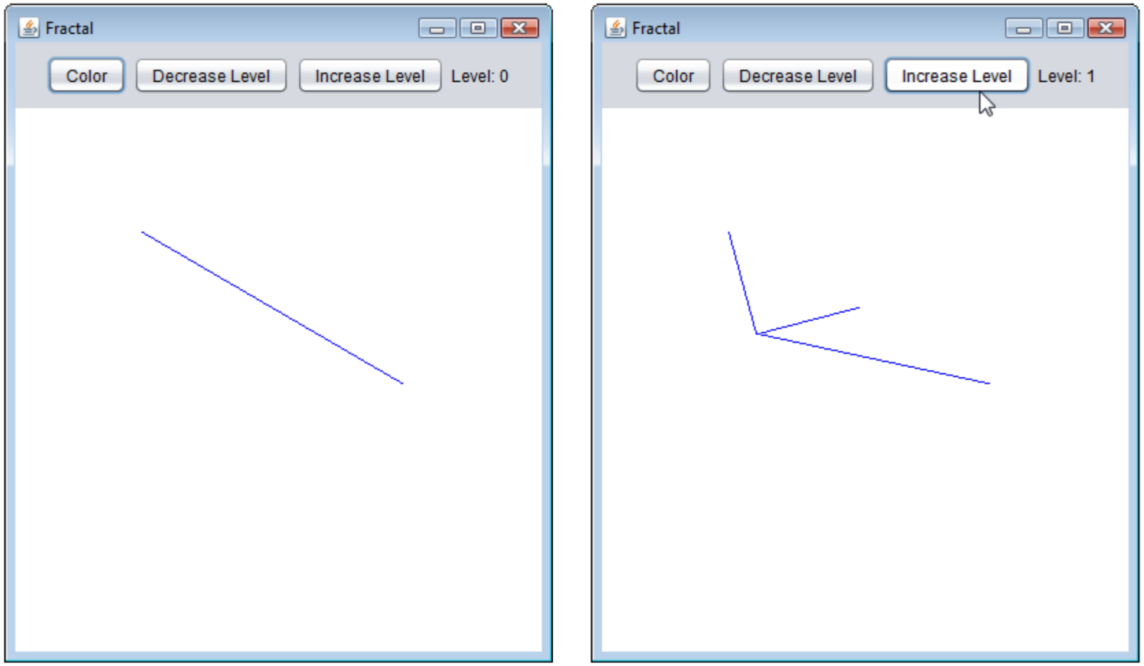

Fig. 18.19 | Drawing the "Lo feather fractal" using recursion. (Part 5 of 8.)

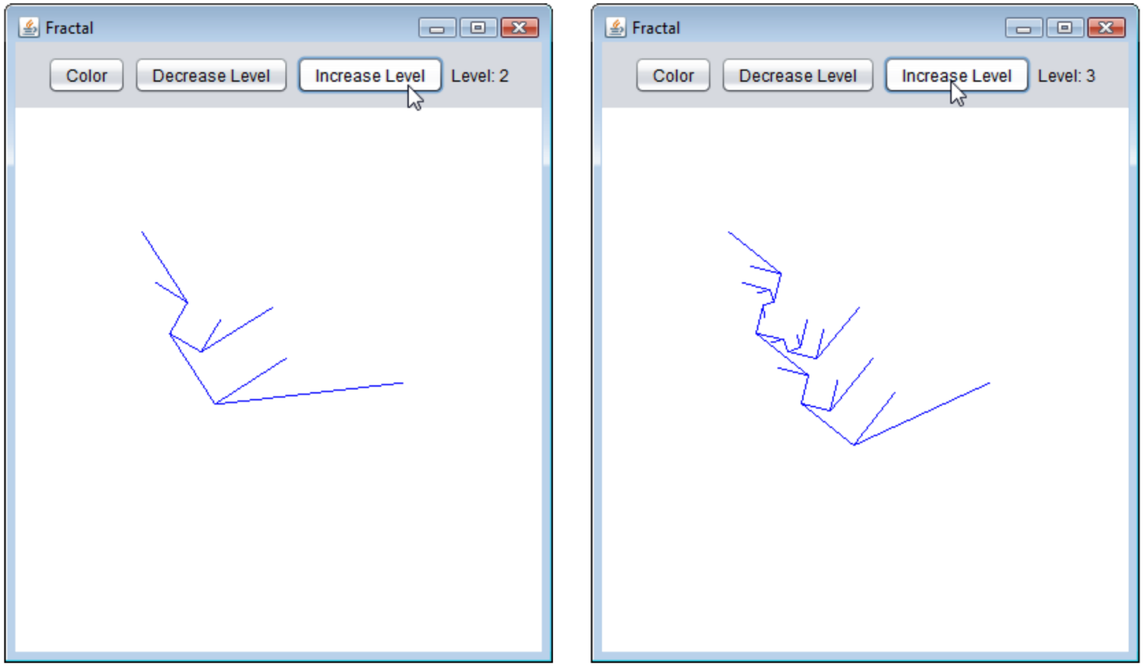

Fig. 18.19 | Drawing the "Lo feather fractal" using recursion. (Part 6 of 8.)

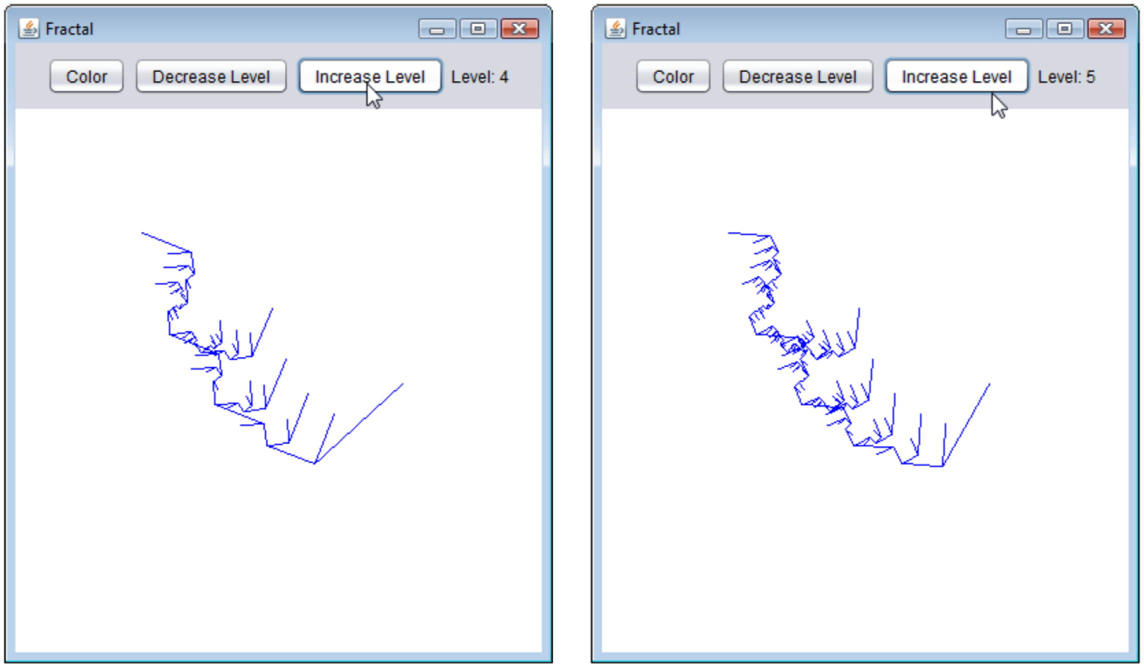

Fig. 18.19 | Drawing the "Lo feather fractal" using recursion. (Part 7 of 8.)

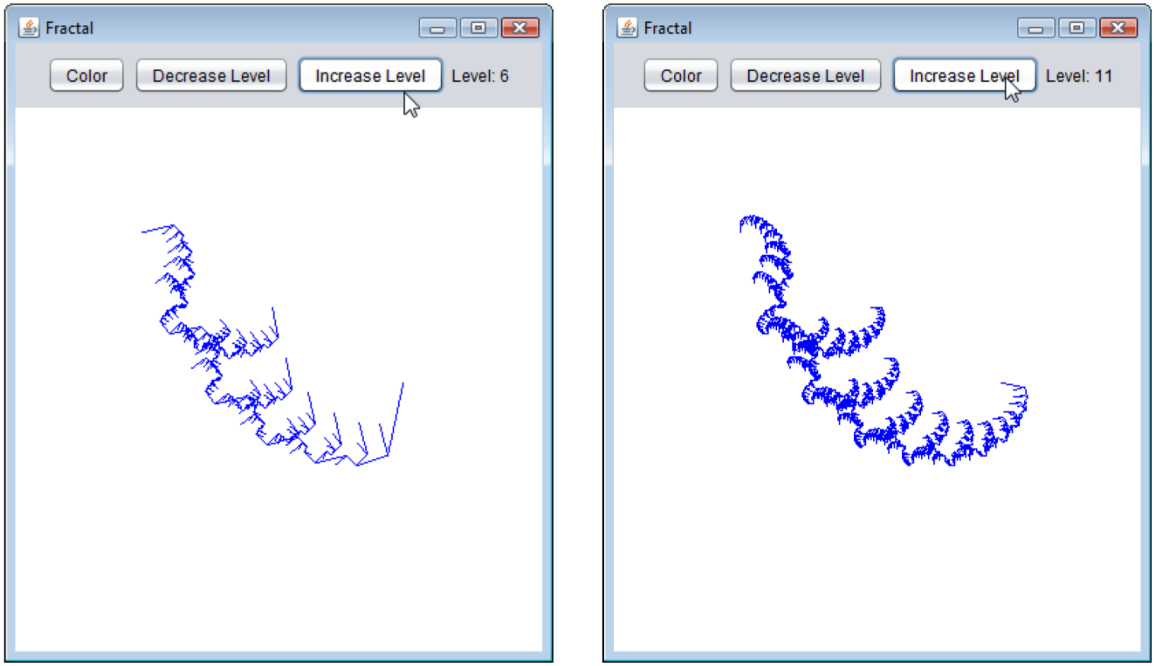

Fig. 18.19 | Drawing the "Lo feather fractal" using recursion. (Part 8 of 8.)

18.10 Recursive Backtracking

- Find a path through a maze, returning true if there is a possible solution to the maze.
- Involves moving through the maze one step at a time, where moves can be made by going down, right, up or left.
- From the current location, for each possible direction, the move is made in that direction and a recursive call is made to solve the remainder of the maze from the new location.
  - When a dead end is reached, back up to the previous location and try to go in a different direction.
  - If no other direction can be taken, back up again.
- Continue until you find a point in the maze where a move *can* be made in another direction.
  - Move in the new direction and continue with another recursive call to solve the rest of the maze.
- Using recursion to return to an earlier decision point is known as recursive backtracking.