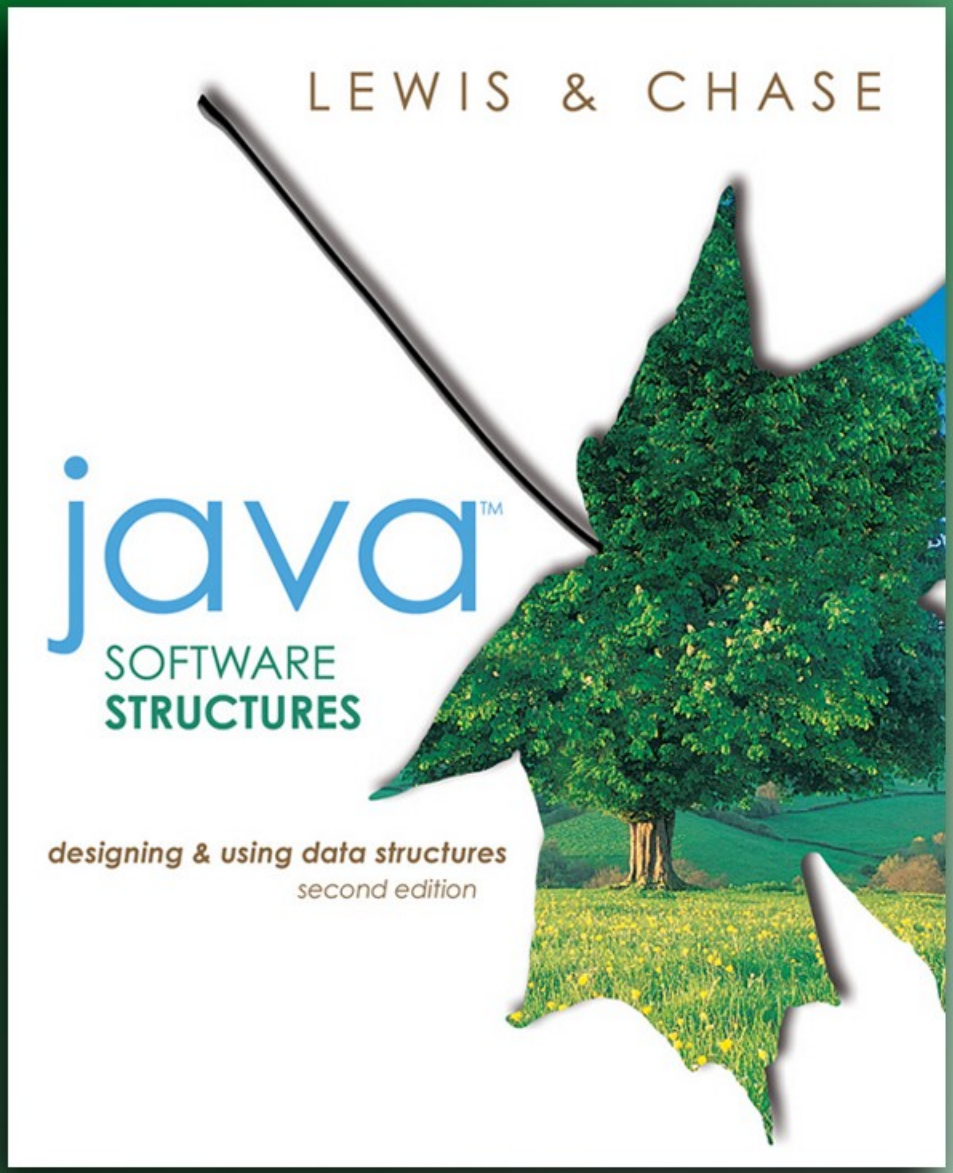
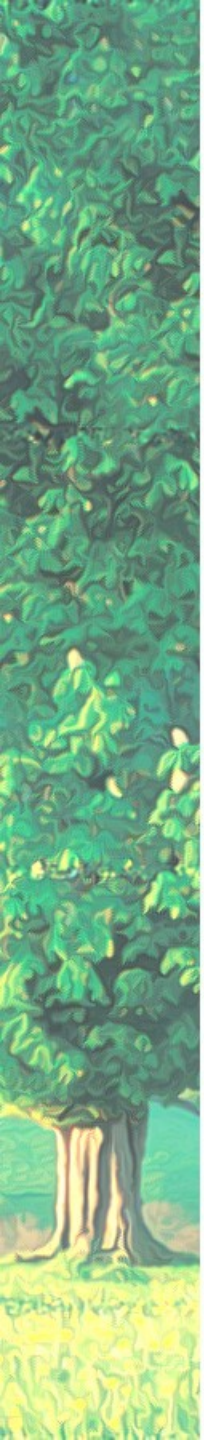


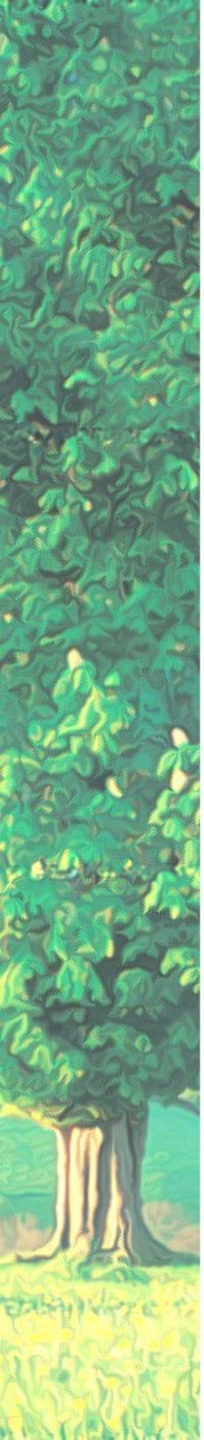
Chapter 10

Recursion



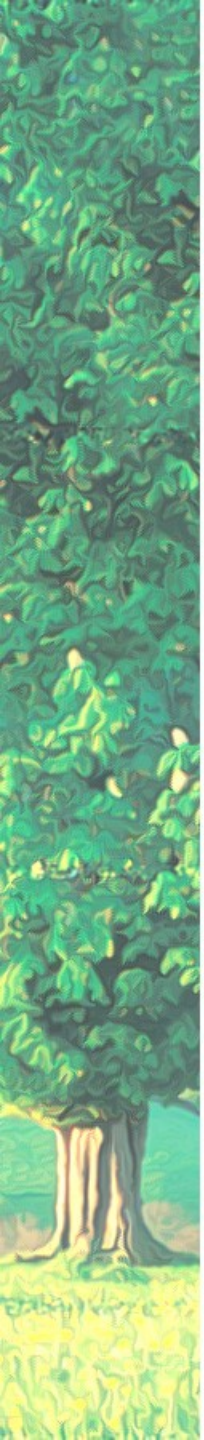
Chapter Objectives

- Explain the underlying concepts of recursion(giả thích khái niệm đệ quy đơn gian)
- Examine recursive methods and unravel their processing steps(xem xét phương thức đệ quy và làm sáng tỏ từng bước xử lí)
- Define infinite(vô hạn) recursion and discuss(thỏa luận) ways to avoid it
- Explain when recursion should and should not be used
- Demonstrate(c/m giải thích) the use of recursion to solve problems



Recursive Thinking

- *Recursion* is a programming technique in which a method can call itself to solve a problem
- A *recursive definition* is one which uses the word or concept being defined in the definition itself
- In some 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:

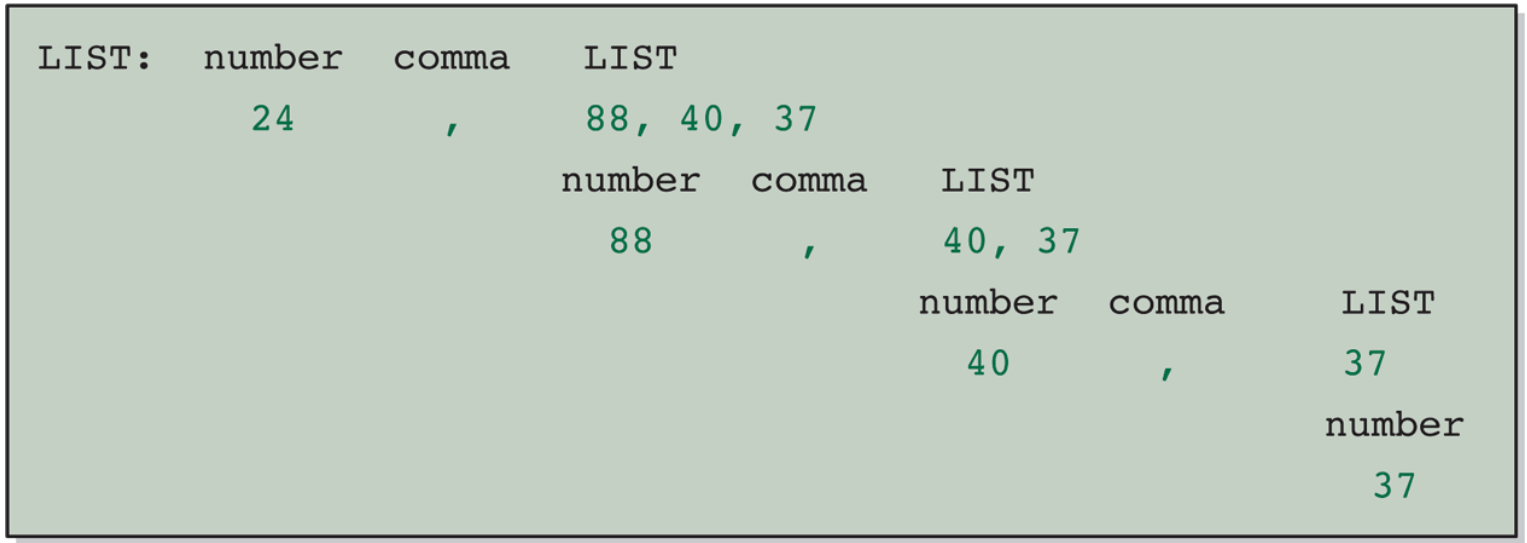
```
24, 88, 40, 37
```

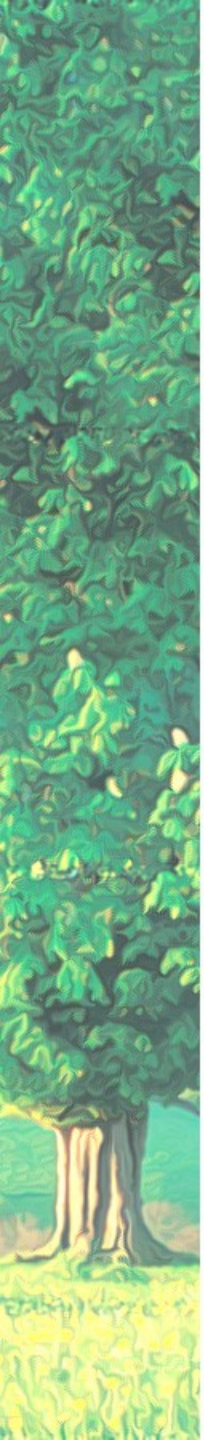
- Such a list can be defined recursively:

A LIST is a:	**number**
or a:	**number comma LIST**

- That is, a LIST can be a number, or a number followed by a comma followed by a LIST
- The concept of a LIST is used to define itself

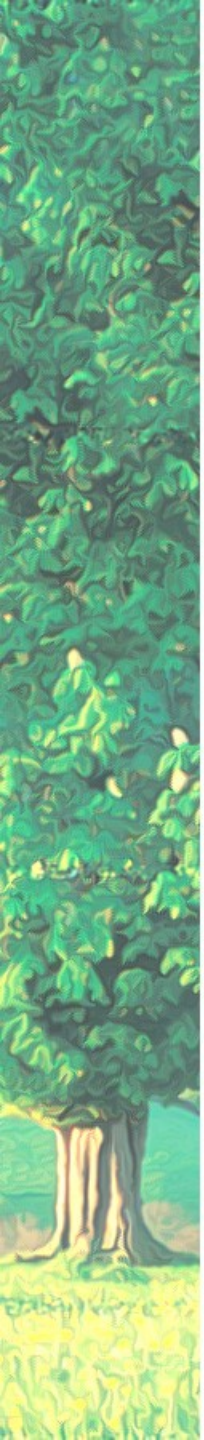
FIGURE 10.1 Tracing the recursive definition of a list





Infinite Recursion

- All recursive definitions must have a non-recursive part
- If they don't, there is no way to terminate the recursive path
- A definition without a non-recursive part causes infinite recursion
- This problem is similar to an infinite loop -- with the definition itself causing the infinite "looping"
- The non-recursive part often is called the *base case*

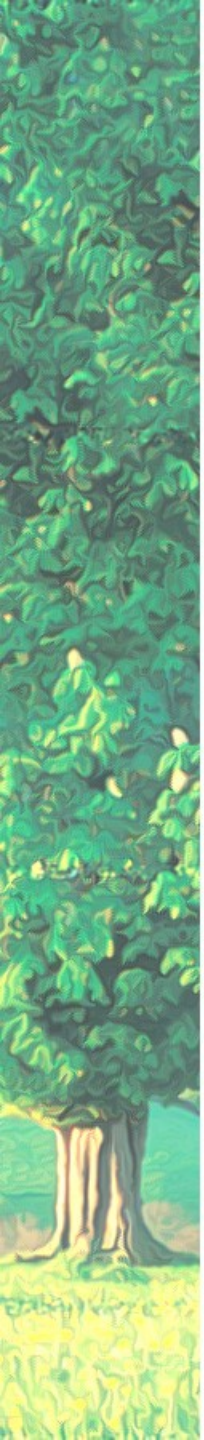


Recursive Definitions

- Mathematical formulas are often expressed recursively
- 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:

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

- A factorial is defined in terms of another factorial until the base case of 1! is reached

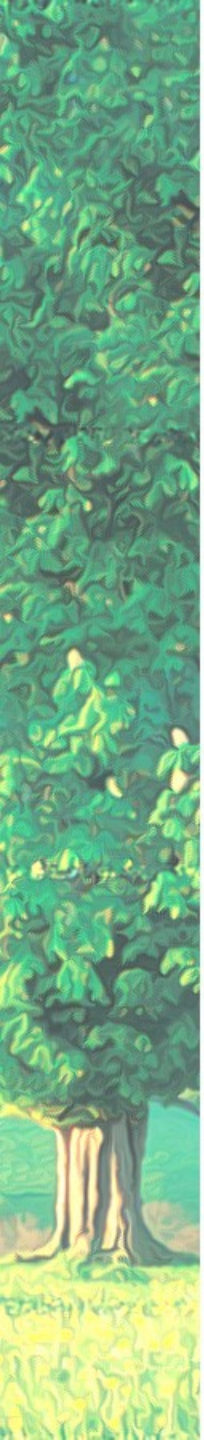


Recursive Programming

- A method in Java can invoke itself; if set up that way, it is called a *recursive method*
- The code of a recursive method must be structured to handle both the base case and the recursive case

stopper recursion

- Each call sets up a new execution environment, with new parameters and new local variables
- As always, when the method completes, control returns to the method that invoked it (which may be another version of itself)



Recursive Programming

- Consider the problem of computing the sum of all the numbers between 1 and N, inclusive
- If N is 5, the sum is
- 1 + 2 + 3 + 4 + 5
- This problem can be expressed recursively as:

The sum of 1 to N is N plus the sum of 1 to N-1

FIGURE 10.2 The sum of the numbers 1 through N, defined recursively

$$\sum_{i=1}^{N} i = N + \sum_{i=1}^{N-1} i = N + N-1 + \sum_{i=1}^{N-2} i$$

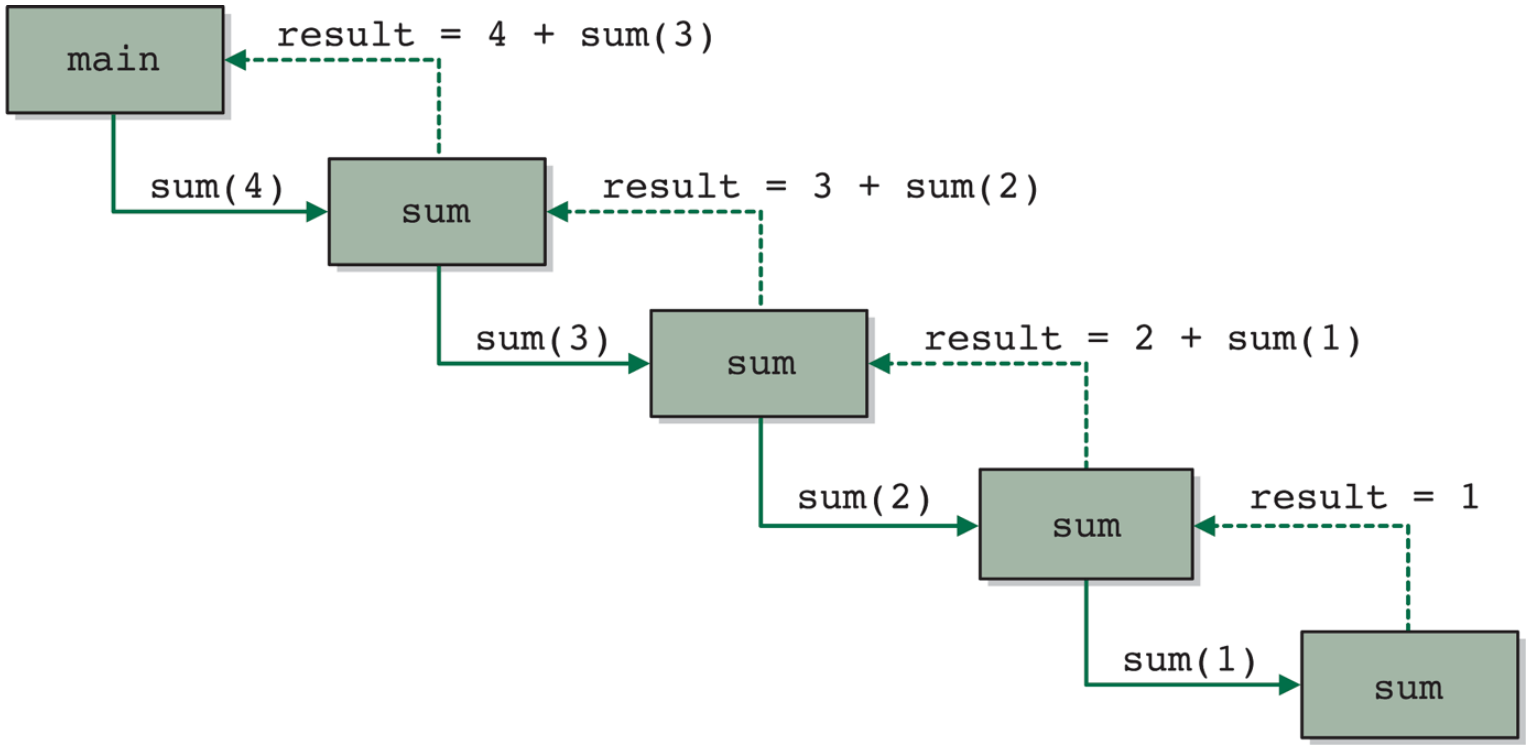
$$= N + N-1 + N-2 + \sum_{i=1}^{N-3} i$$

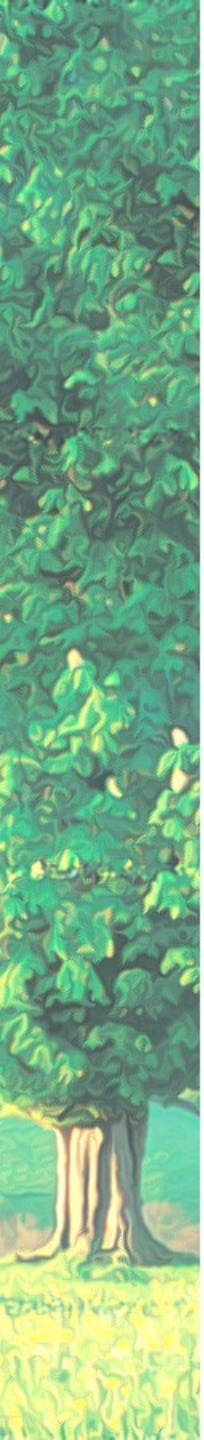
$$= N + N-1 + N-2 + \ldots + 2 + 1$$

Recursive Programming

```
public int sum (int num)
{
    int result;
    if (num == 1)
        result = 1;
    else
        result = num + sum(num-1);
    return result;
}
```

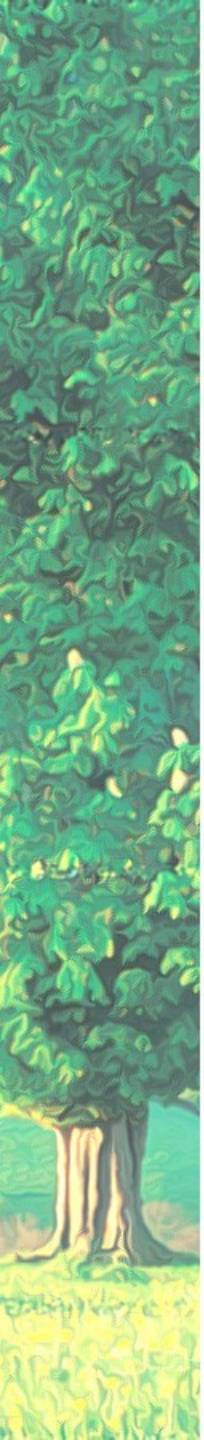
FIGURE 10.3
Recursive calls to the sum method





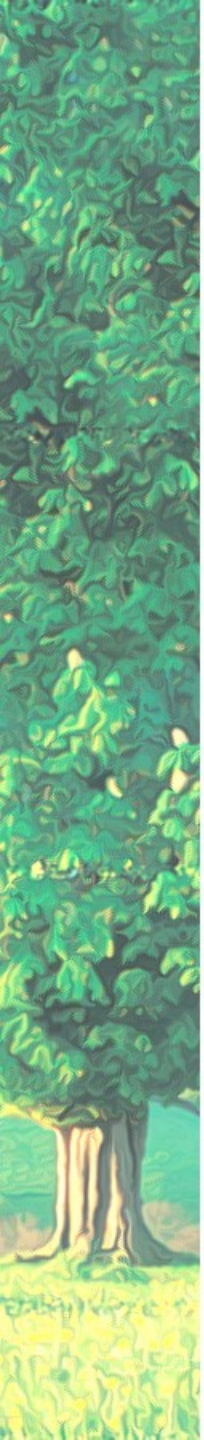
Recursion vs. Iteration

- Just because we can use recursion to solve a problem, doesn't mean we should
- For instance, we usually would not use recursion to solve the sum of 1 to N
- The iterative version is easier to understand (in fact there is a formula that is superior to both recursion and iteration in this case)
- You must be able to determine when recursion is the correct technique to use



Recursion vs. Iteration

- Every recursive solution has a corresponding iterative solution
- For example, the sum of the numbers between 1 and N can be calculated with a loop
- Recursion has the overhead of multiple method invocations
- However, for some problems recursive solutions are often more simple and elegant than iterative solutions

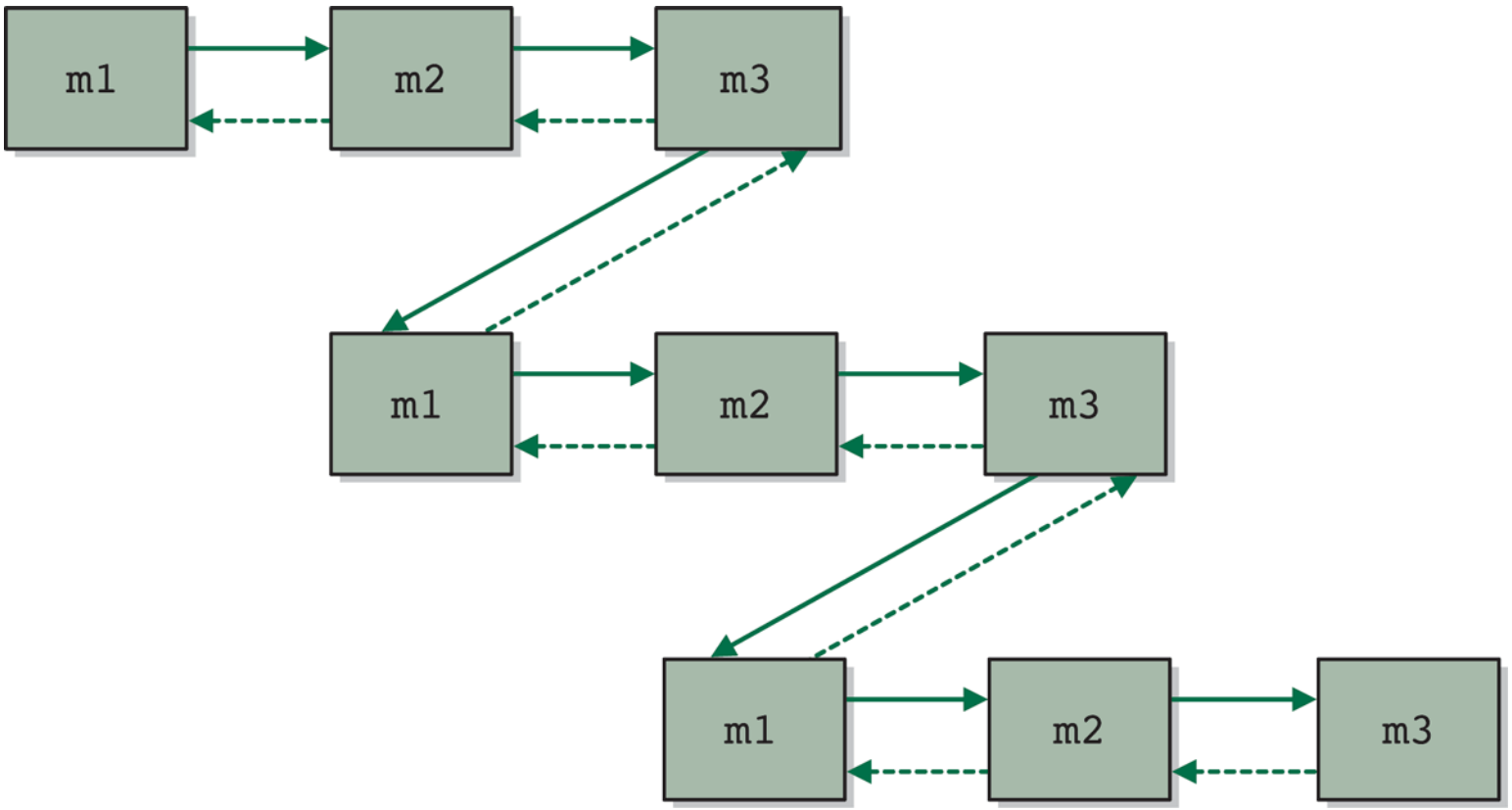


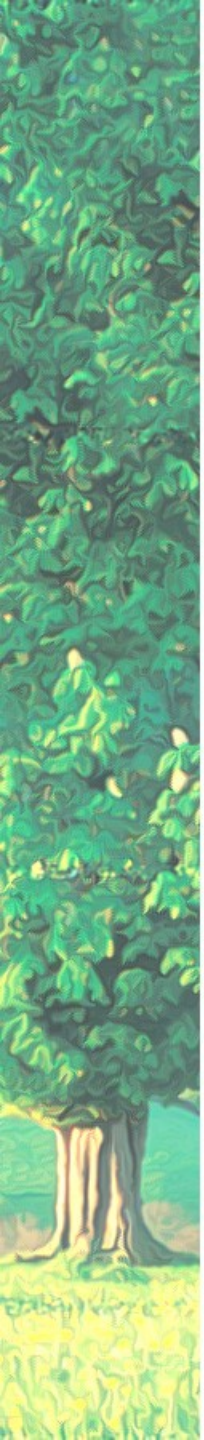
Indirect Recursion

- 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 invokes m1 again
- This is called *indirect recursion*
- It is often more difficult to trace and debug

hvis en metode kalder en anden metode, som gør det samme

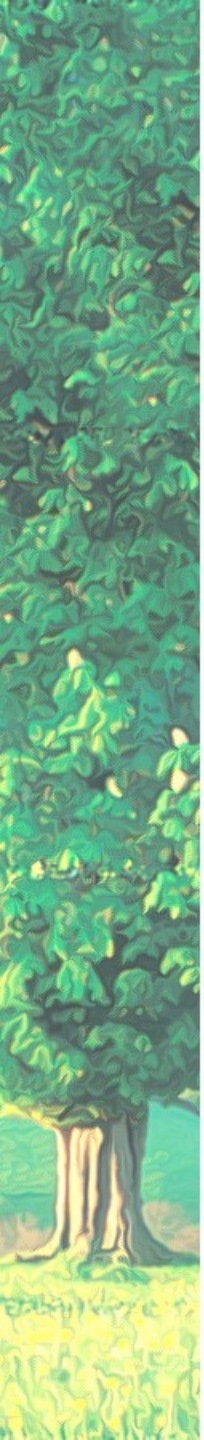
FIGURE 10.4 Indirect recursion





Maze Traversal

- Let's use recursion to find a path through a maze
- A path can be found through a maze from location x if a path can be found from any of the locations neighboring x
- We can mark each location we encounter as "visited" and then attempt to find a path from that location's unvisited neighbors



Maze Traversal

- Recursion will be used to keep track of the path through the maze using the run-time stack
- The base cases are
  - a prohibited (blocked) move, or
  - arrival at the final destination

Listing 10.1

Listing 10.1

```
//********************************************************************
//  MazeSearch.java       Authors: Lewis/Chase
//
//  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);
   }
}
```

Listing 10.2

Listing 10.2

```
//********************************************************************
//  Maze.java        Authors: Lewis/Chase
//
//  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} };
```

Listing 10.2 (cont.)

```
//-----------------------------------------------------------------
//  Attempts to recursively traverse the maze. Inserts special
//  characters indicating locations that have been tried and that
//  eventually become part of the solution.
//-----------------------------------------------------------------
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
      else
      {
   done = traverse (row+1, column);      // down
         if (!done)
            done = traverse (row, column+1);  // right
```

Listing 10.2 (cont.)

Listing 10.2 continued

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

Listing 10.2 (cont.)

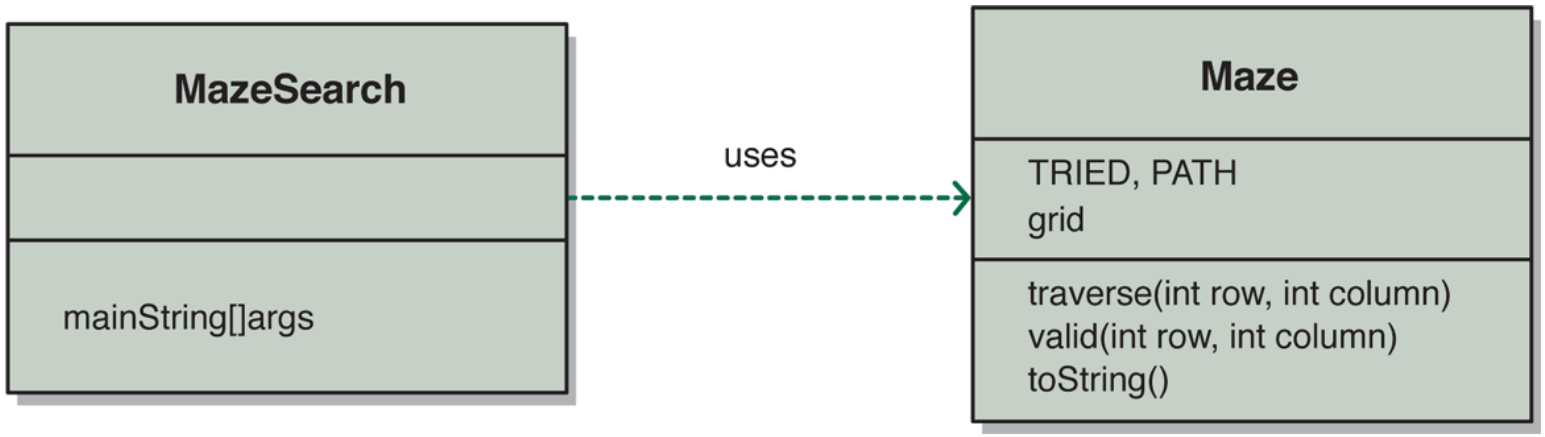
```
      // check if cell is in the bounds of the matrix
      if (row >= 0 && row < grid.length &&
          column >= 0 && column < grid[row].length)

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

      return result;
   }

   //-----------------------------------------------------------------
   //  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] + "";
```

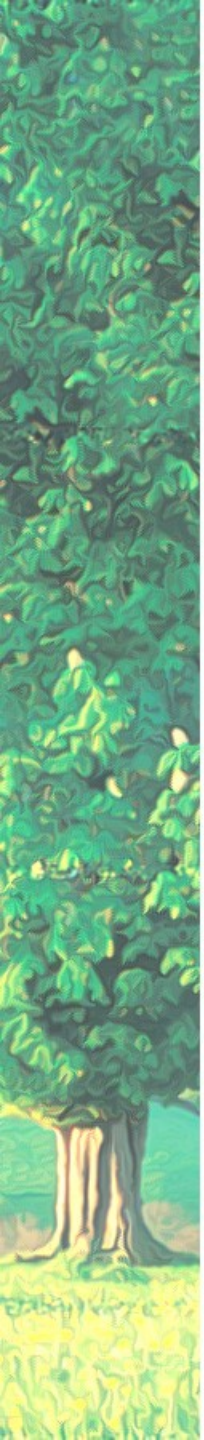
Listing 10.2 (cont.)

Listing 10.2 continued

```
            result += "\n";
        }

        return result;
    }

}
```

FIGURE 10.5 UML description of the Maze and MazeSearch classes



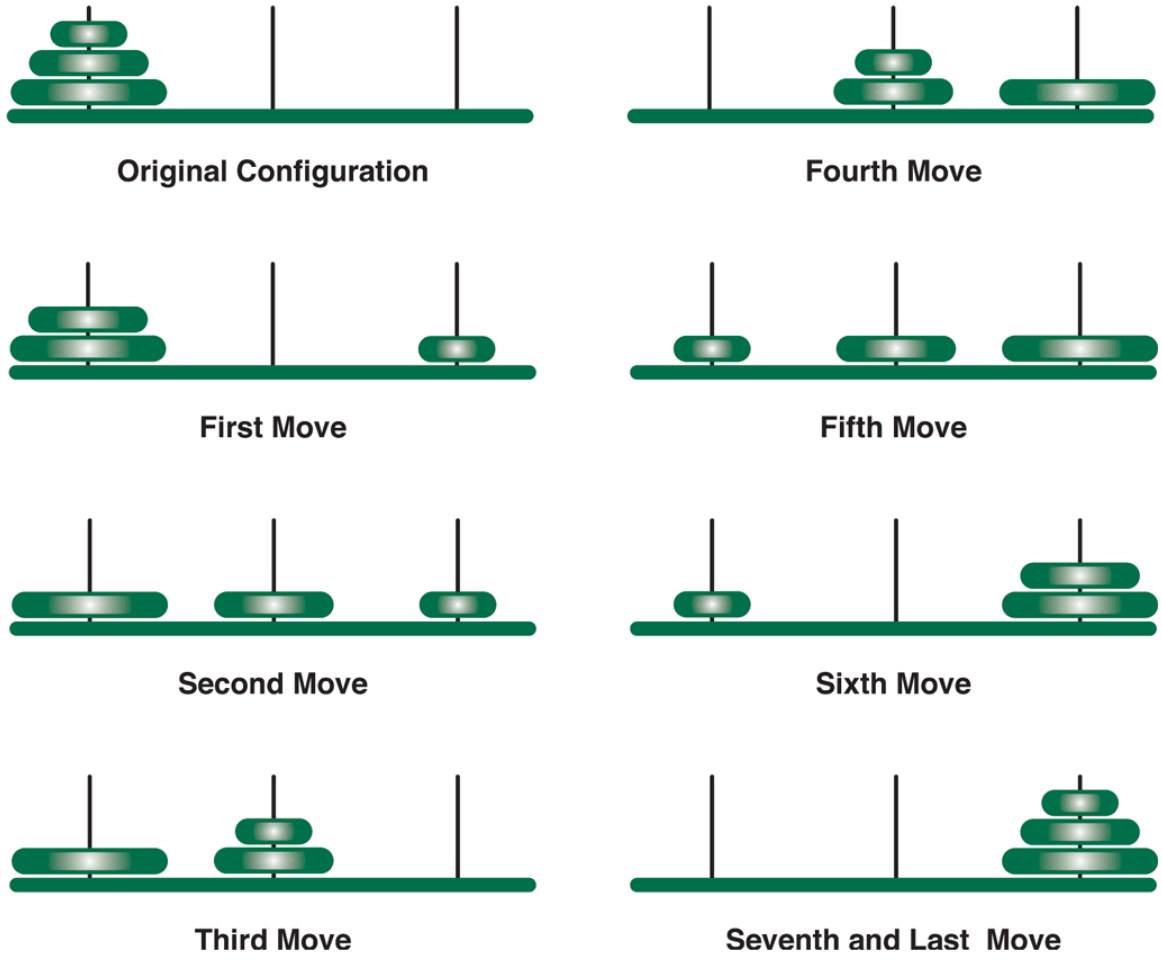


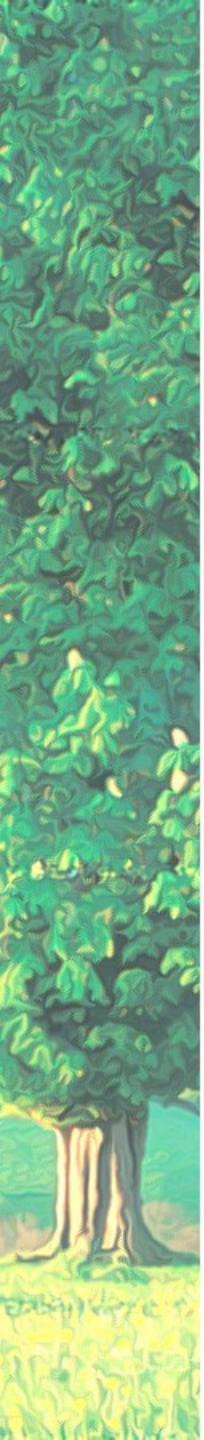
The Towers of Hanoi

- 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 and increasingly smaller disks on top
- The goal is to move all of the disks from one peg to another following these rules:
  - Only one disk can be moved at a time
  - A disk cannot be placed on top of a smaller disk
  - All disks must be on some peg (except for the one in transit)

FIGURE 10.6
The Towers of Hanoi puzzle

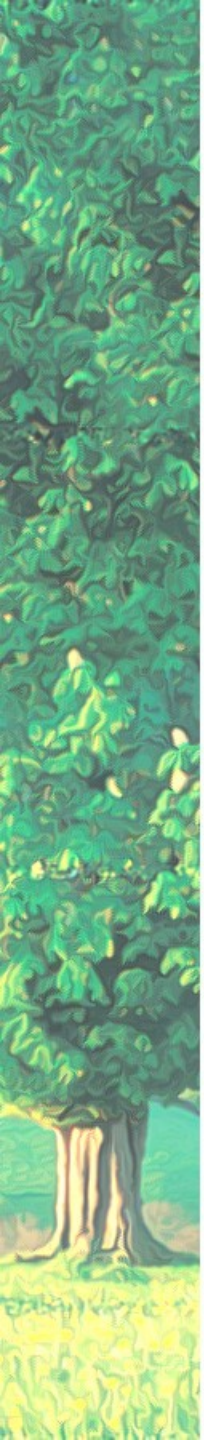
FIGURE 10.7 A solution to the three-disk Towers of Hanoi puzzle





Towers of Hanoi

- To move a stack of N disks from the original peg to the destination peg:
  - Move the topmost N-1 disks from the original peg to the extra peg
  - Move the largest disk from the original peg to the destination peg
  - Move the N-1 disks from the extra peg to the destination peg
- The base case occurs when a "stack" contains only one disk



Towers of Hanoi

- Note that the number of moves increases exponentially as the number of disks increases
- The recursive solution is simple and elegant to express (and program)
- An iterative solution to this problem is much more complex

Listing 10.3

Listing 10.3

```
//********************************************************************
//  SolveTowers.java        Authors: Lewis/Chase
//
//  Demonstrates recursion.
//********************************************************************

public class SolveTowers
{
   //-----------------------------------------------------------------
   //  Creates a TowersOfHanoi puzzle and solves it.
   //-----------------------------------------------------------------
   public static void main (String[] args)
```

Listing 10.3 (cont.)

Listing 10.3 continued

```
    {
        TowersOfHanoi towers = new TowersOfHanoi (4);

        towers.solve();
    }
}
```

Listing 10.4

Listing 10.4

```
//********************************************************************
//  TowersOfHanoi.java        Authors: Lewis/Chase
//
//  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 ()
   {
```

Listing 10.4 (cont.)

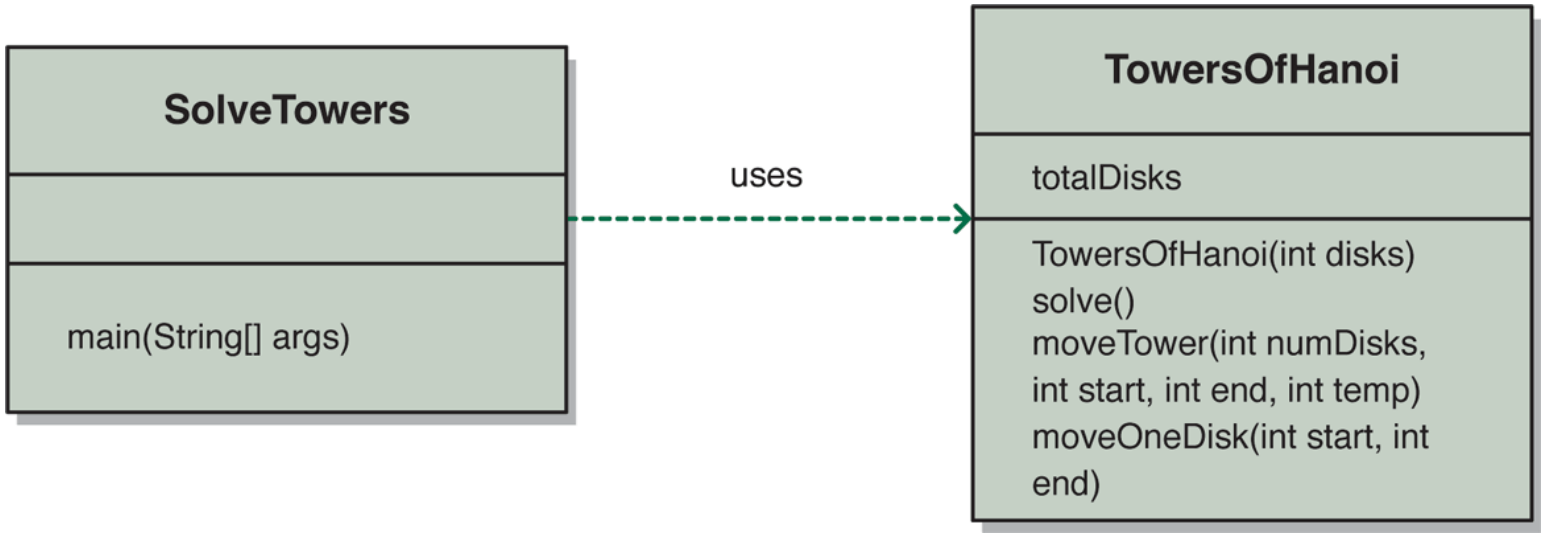
Listing 10.4 continued

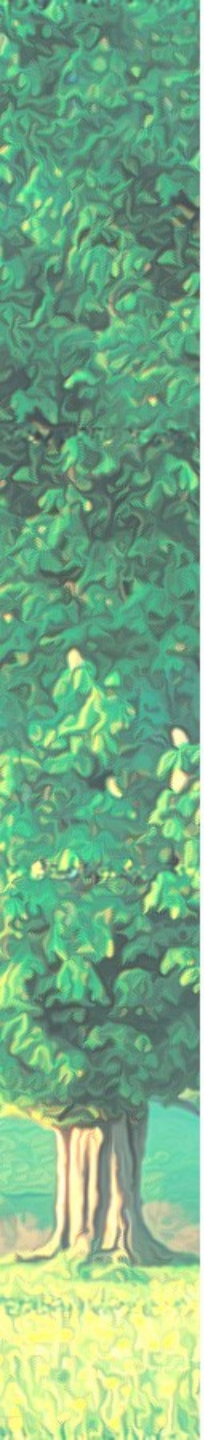
```
      moveTower (totalDisks, 1, 3, 2);
   }

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

Listing 10.4 (cont.)

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

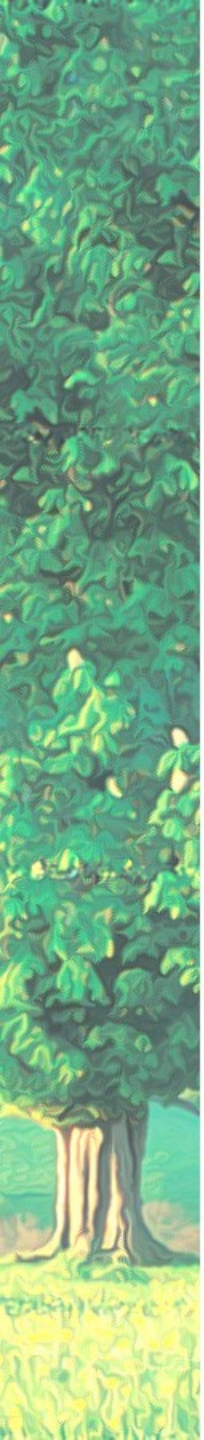
FIGURE 10.8 UML description of the SolveTowers and TowersofHanoi classes





Analyzing Recursive Algorithms

- When analyzing a loop, we determine the order of the loop body and multiply it by the number of times the loop is executed
- Recursive analysis is similar
- We determine the order of the method body and multiply it by the *order of the recursion* (the number of times the recursive definition is followed)



Analyzing Recursive Algorithms

- For the Towers of Hanoi, the size of the problem is the number of disks and the operation of interest is moving one disk
- Except for the base case, each recursive call results in calling itself twice more
- To solve a problem of N disks, we make 2^N-1 disk moves
- Therefore the algorithm is $O(2^n)$, which is called exponential complexity