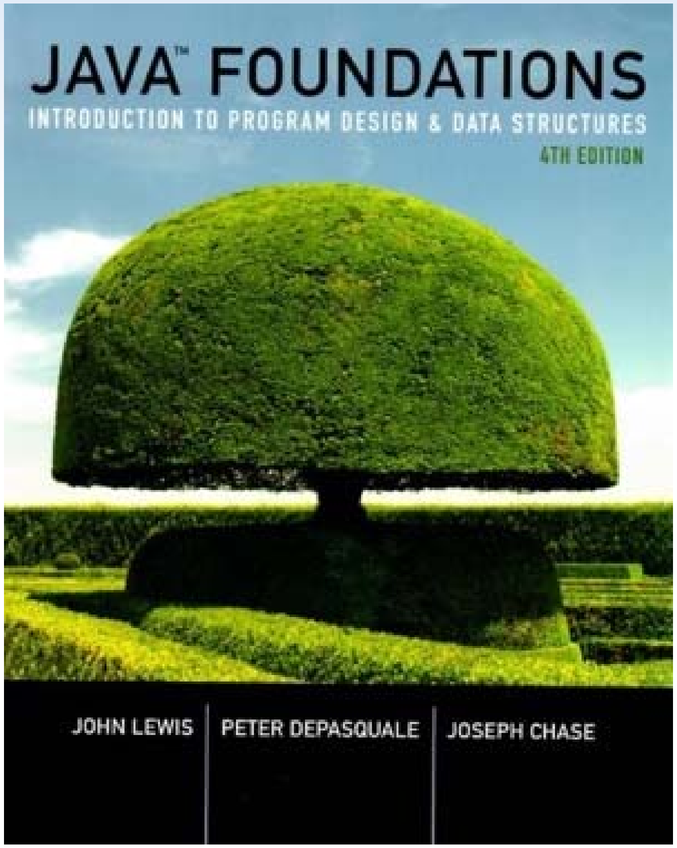


Chapter 17

Recursion

Lewis, J. DePasquale, P. & Chase, J. (2017). PowerPoint lecture slides for java foundations: Introduction to program design and data structures. Pearson.

Chapter Scope

- The concept of recursion
- Recursive methods
- Infinite recursion
- When to use (and not use) recursion
- Using recursion to solve problems
  - Solving a maze
  - Towers of Hanoi

Recursion

- *Recursion* is a programming technique in which a method can call itself to fulfill its purpose
- 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

Recursive Definitions

LIST:	**number**	**comma**	**LIST**
	24	,	88, 40, 37
	number	**comma**	**LIST**
	88	,	40, 37
	number	**comma**	**LIST**
	40	,	37
	number		
	37		

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 is called the *base case*

Recursion in Math

- 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 handle both the base case and the recursive case
- 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 instance of itself)

Recursive Programming

- Consider the problem of computing the sum of all the integers 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

Recursive Programming

- The sum of the integers between 1 and N:

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

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

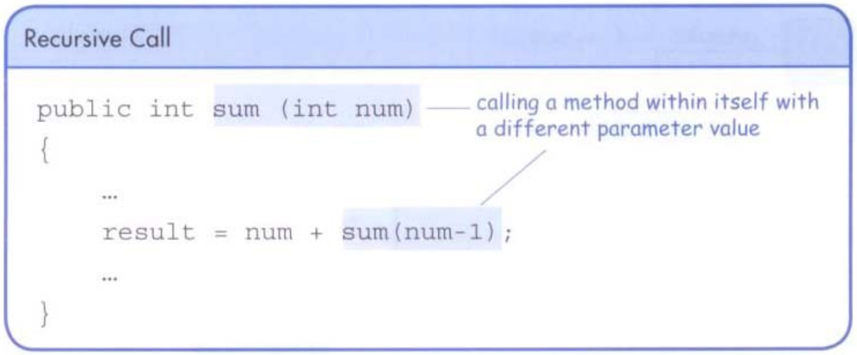
$$\vdots$$

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

Recursive Programming

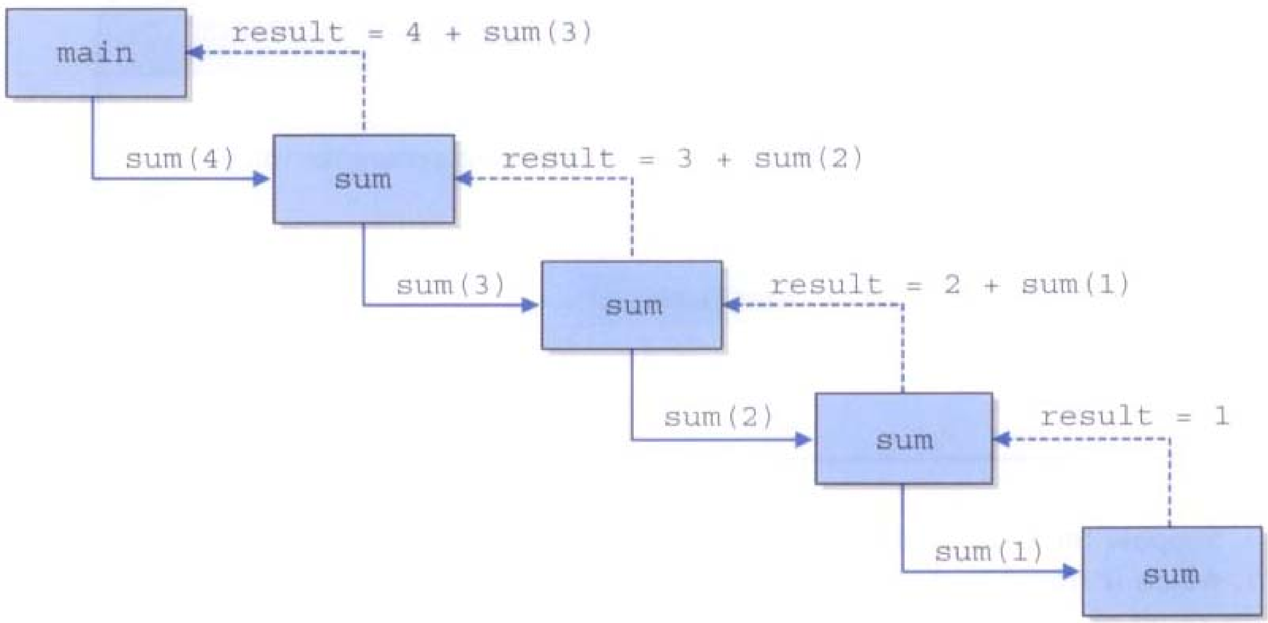
- A recursive method that computes the sum of 1 to N:

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



Recursive Programming

- Tracing the recursive calls of 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 computes it without a loop at all)
- You must be able to determine when recursion is the correct technique to use

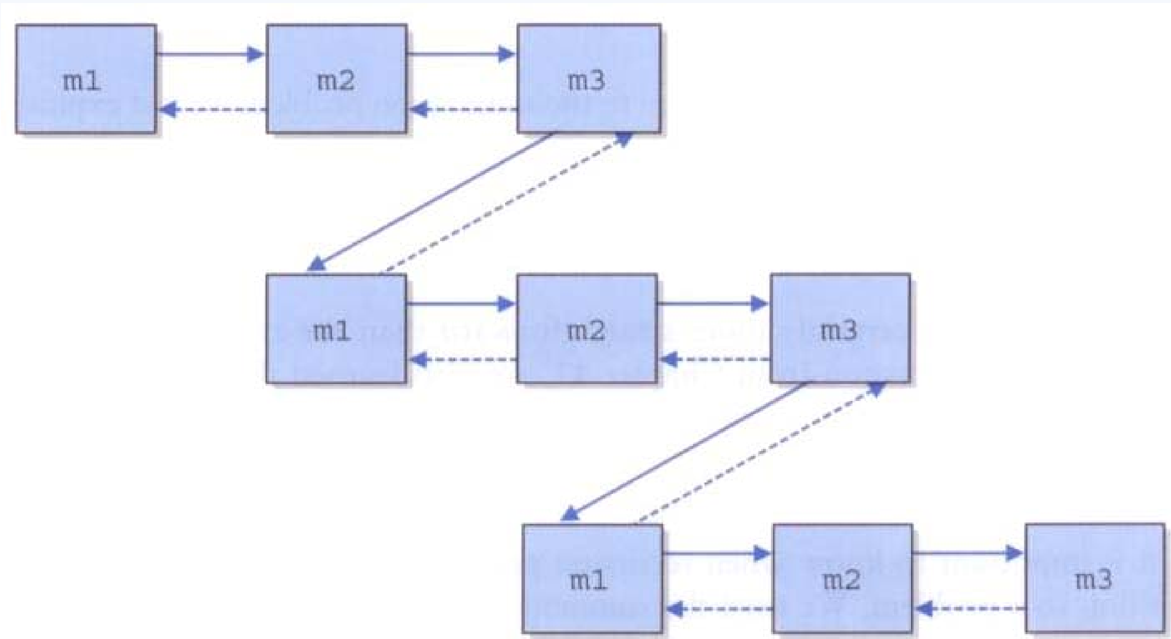
Recursion vs. Iteration

- Every recursive solution has a corresponding iterative solution
- A recursive solution may simply be less efficient
- Furthermore, recursion has the overhead of multiple method invocations
- However, for some problems recursive solutions are often more simple and elegant to express

Direct vs. 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

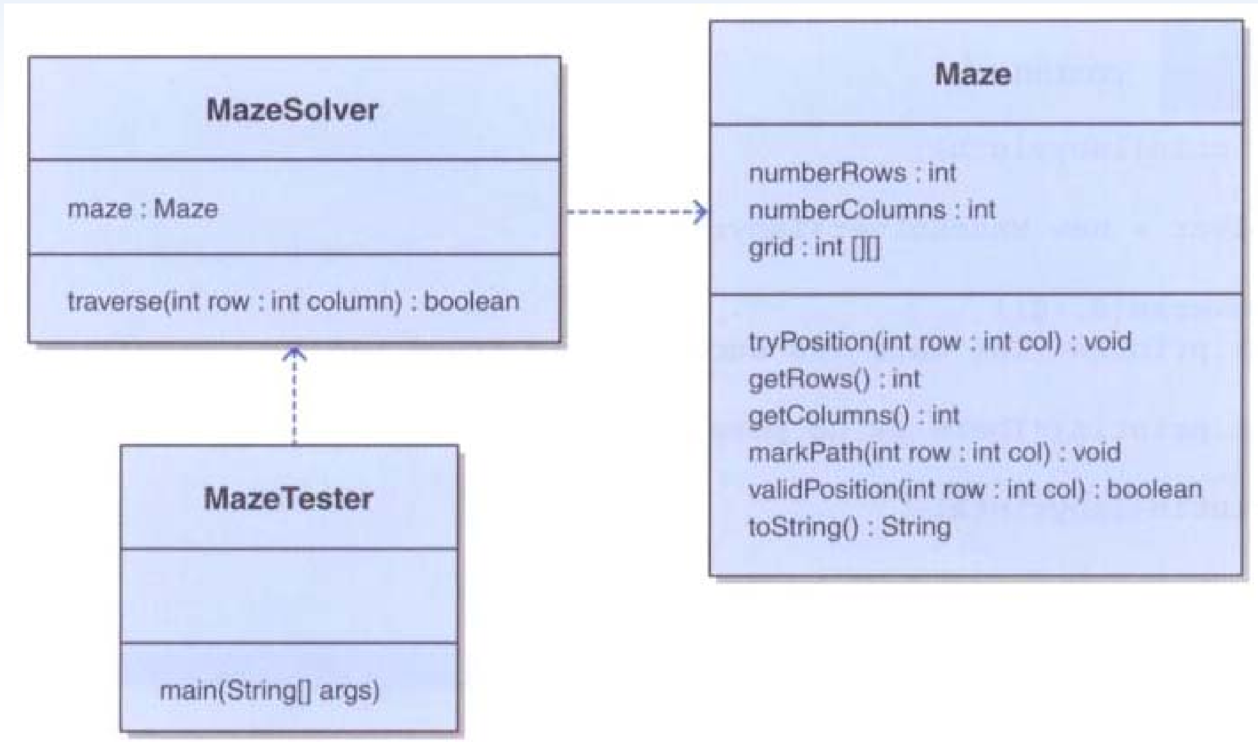
Direct vs. Indirect Recursion



Maze Traversal

- We've seen a maze solved using a stack
- The same approach can also be done using recursion
- The run-time stack tracking method execution performs the same function
- As before, we mark a location as "visited" and try to continue along the path
- The base cases are:
  - a blocked path
  - finding a solution

Maze Traversal



```java
import java.util.*;
import java.io.*;

/**
 * MazeTester uses recursion to determine if a maze can be traversed.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class MazeTester
{
    /**
     * Creates a new maze, prints its original form, attempts to
     * solve it, and prints out its final form.
     */
    public static void main(String[] args) throws FileNotFoundException
    {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter the name of the file containing the maze: ");
        String filename = scan.nextLine();

        Maze labyrinth = new Maze(filename);

        System.out.println(labyrinth);

        MazeSolver solver = new MazeSolver(labyrinth);
```

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

        System.out.println(labyrinth);
    }
}
```

```java
import java.util.*;
import java.io.*;

/**
 * Maze represents a maze of characters. The goal is to get from the
 * top left corner to the bottom right, following a path of 1's. Arbitrary
 * constants are used to represent locations in the maze that have been TRIED
 * and that are part of the solution PATH.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class Maze
{
    private static final int TRIED = 2;
    private static final int PATH = 3;

    private int numberRows, numberColumns;
    private int[][] grid;
```

```java
/**
 * Constructor for the Maze class. Loads a maze from the given file.
 * Throws a FileNotFoundException if the given file is not found.
 *
 * @param filename the name of the file to load
 * @throws FileNotFoundException if the given file is not found
 */
public Maze(String filename) throws FileNotFoundException
{
    Scanner scan = new Scanner(new File(filename));
    numberRows = scan.nextInt();
    numberColumns = scan.nextInt();

    grid = new int[numberRows][numberColumns];
    for (int i = 0; i < numberRows; i++)
        for (int j = 0; j < numberColumns; j++)
            grid[i][j] = scan.nextInt();
}

/**
 * Marks the specified position in the maze as TRIED
 *
 * @param row the index of the row to try
 * @param col the index of the column to try
 */
public void tryPosition(int row, int col)
{
    grid[row][col] = TRIED;
}
```

```java
/**
 * Return the number of rows in this maze
 *
 * @return the number of rows in this maze
 */
public int getRows()
{
    return grid.length;
}

/**
 * Return the number of columns in this maze
 *
 * @return the number of columns in this maze
 */
public int getColumns()
{
    return grid[0].length;
}

/**
 * Marks a given position in the maze as part of the PATH
 *
 * @param row the index of the row to mark as part of the PATH
 * @param col the index of the column to mark as part of the PATH
 */
public void markPath(int row, int col)
{
    grid[row][col] = PATH;
}
```

```java
/**
 * Determines if a specific location is valid. A valid location
 * is one that is on the grid, is not blocked, and has not been TRIED.
 *
 * @param row the row to be checked
 * @param column the column to be checked
 * @return true if the location is valid
 */
public boolean validPosition(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;
}
```

```java
    /**
     * Returns the maze as a string.
     *
     * @return a string representation of the maze
     */
    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;
    }
}
```

```java
/**
 * MazeSolver attempts to recursively traverse a Maze. The goal is to get from the
 * given starting position to the bottom right, following a path of 1's. Arbitrary
 * constants are used to represent locations in the maze that have been TRIED
 * and that are part of the solution PATH.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class MazeSolver
{
    private Maze maze;

    /**
     * Constructor for the MazeSolver class.
     */
    public MazeSolver(Maze maze)
    {
        this.maze = maze;
    }
```

```java
/**
 * Attempts to recursively traverse the maze. Inserts special
 * characters indicating locations that have been TRIED and that
 * eventually become part of the solution PATH.
 *
 * @param row row index of current location
 * @param column column index of current location
 * @return true if the maze has been solved
 */
public boolean traverse(int row, int column)
{
    boolean done = false;

    if (maze.validPosition(row, column))
    {
        maze.tryPosition(row, column);   // mark this cell as tried

        if (row == maze.getRows()-1 && column == maze.getColumns()-1)
            done = true;  // the maze is solved
        else
        {
            done = traverse(row+1, column);      // down
            if (!done)
                done = traverse(row, column+1);  // right
```

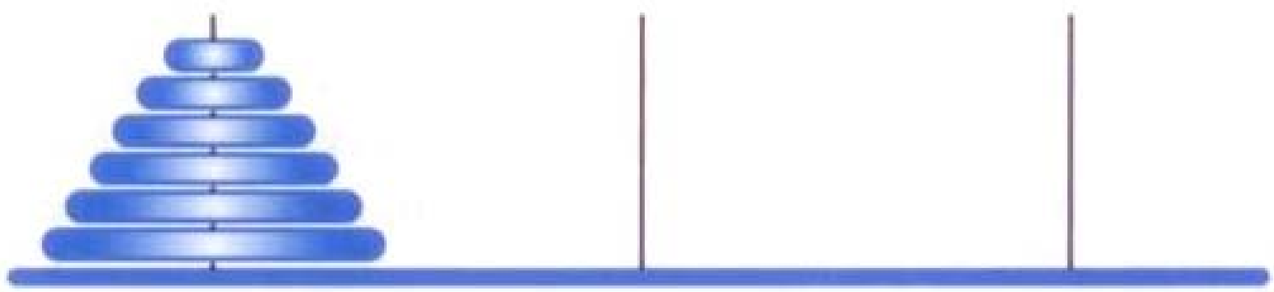
```java
                if (!done)
                    done = traverse(row-1, column);  // up
                if (!done)
                    done = traverse(row, column-1);  // left
            }

            if (done)  // this location is part of the final path
                maze.markPath(row, column);
        }

        return done;
    }
}
```

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)

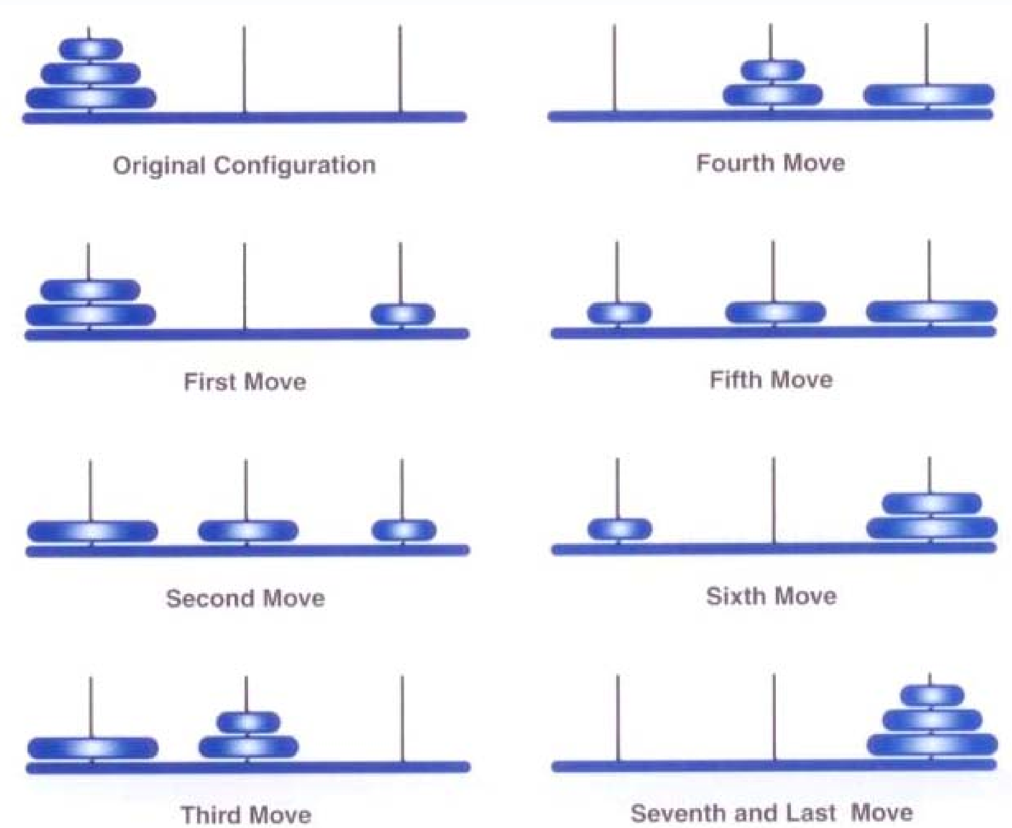
Towers of Hanoi

- The initial state of the Towers of Hanoi puzzle:



Towers of Hanoi

- A solution to the three-disk Towers of Hanoi puzzle:



Lewis, J. DePasquale, P. & Chase, J. (2017). PowerPoint lecture slides for java foundations: Introduction to program design and data structures. Pearson.

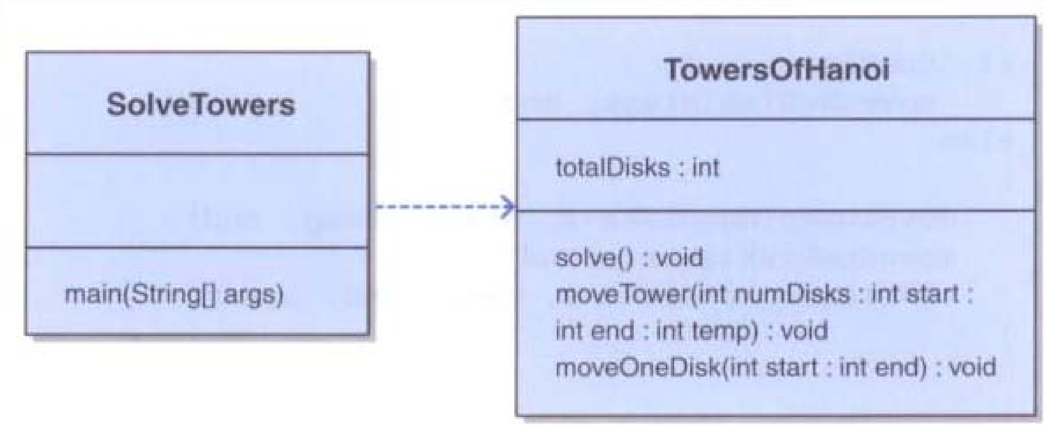
Towers of Hanoi

- A solution to ToH can be expressed recursively
- To move 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 peg contains only one disk

Towers of Hanoi

- The number of moves increases exponentially as the number of disks increases
- The recursive solution is simple and elegant to express and program, but is very inefficient
- However, an iterative solution to this problem is much more complex to define and program

Towers of Hanoi



Lewis, J. DePasquale, P. & Chase, J. (2017). PowerPoint lecture slides for java foundations: Introduction to program design and data structures. Pearson.

```java
/**
 * SolveTowers uses recursion to solve the Towers of Hanoi puzzle.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class SolveTowers
{
    /**
     * Creates a TowersOfHanoi puzzle and solves it.
     */
    public static void main(String[] args)
    {
        TowersOfHanoi towers = new TowersOfHanoi(4);
        towers.solve();
    }
}
```

```java
/**
 * TowersOfHanoi represents the classic Towers of Hanoi puzzle.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class TowersOfHanoi
{
    private int totalDisks;

    /**
     * Sets up the puzzle with the specified number of disks.
     *
     * @param disks the 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);
    }
```

```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.
 *
 * @param numDisks  the number of disks to move
 * @param start     the starting tower
 * @param end       the ending tower
 * @param temp      the temporary tower
 */
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);
    }
}
```

```java
    /**
     * Prints instructions to move one disk from the specified start
     * tower to the specified end tower.
     *
     * @param start  the starting tower
     * @param end    the ending tower
     */
    private void moveOneDisk(int start, int end)
    {
        System.out.println("Move one disk from " + start + " to " + end);
    }
}
```

Analyzing Recursive Algorithms

- To determine the order of a loop, we determined the order of the body of the loop multiplied by the number of loop executions
- Similarly, to determine the order of a recursive method, we determine the the order of the body of the method multiplied by the number of times the recursive method is called
- In our recursive solution to compute the sum of integers from 1 to N, the method is invoked N times and the method itself is O(1)
- So the order of the overall solution is O(n)

Analyzing Recursive Algorithms

- For the Towers of Hanoi puzzle, the step of moving one disk is O(1)
- But each call results in calling itself twice more, so for N > 1, the growth function is

$$f(n) = 2^n - 1$$

- This is *exponential efficiency*: $O(2^n)$
- As the number of disks increases, the number of required moves increases exponentially