Chapter 12
Recursion

Java Software Solutions
Foundations of Program Design
9th Edition

John Lewis
William Loftus
There are only two ways to live your life. One is as though nothing is a miracle. The other is as though everything is a miracle.
A. Einstein
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
  – recursion in graphics
  – fractals
Outline

Recursive Thinking
Recursive Programming
Traversing a Maze
The Towers of Hanoi
Tiled Images
Fractals
Recursive Thinking

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

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

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

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

• Consider a list of numbers:

   24, 88, 40, 37

• A list can be defined as follows:

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

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

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

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

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

• The following two **axioms** define the natural numbers
  – 0 is a natural number
  – For every natural number \( n \), \( S(n) \) – the successor of \( n \) - is a natural number

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

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

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

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

• For instance:
  
  \[
  \text{A List is a: number comma List}
  \]

• Such a definition would cause **infinite recursion**

• This problem is similar to an infinite loop
Quiz

What is printing the program described in this flowchart?
public int itFactorial (int n) {
    int m = 1, f = 1;
    while (m < n) {
        ++m;
        f = f * m;
    }
    return f;
}
Recursive Definition: Factorial

- N!, for any positive integer N, is defined to be the product of all integers between 1 and N inclusive
  \[ N! = N \times (N-1) \times (N-2) \times \cdots \times 2 \times 1 \]

- This definition can be expressed recursively as:
  \[
  
  \begin{align*}
  1! &= 1 \\
  N! &= N \times (N-1)! \\
  f(1) &= 1 \\
  f(n) &= n \times f(n-1)
  \end{align*}
  \]

- A factorial is defined in terms of another factorial

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

5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!

1
2
6
24
120

Copyright © 2012 Pearson Education, Inc.
Quick Check

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

Write a recursive definition of $f(n) = e^n$, where $n \geq 0$.

\[
\begin{align*}
e^0 &= 1 & \quad & f(0) = 1 \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \quad & \quad & \\
\quad & \ quad
Quick Check

Write a recursive definition of
f(n) = 5 * n
where n > 0.
Write a recursive definition of
\[ f(n) = 5 \times n \]
where \( n > 0 \).

\[
\begin{align*}
5 \times 1 & = 5 \\
5 \times n & = 5 + (5 \times (n-1))
\end{align*}
\]

\[
\begin{align*}
f(1) & = 5 \\
f(n) & = 5 + f(n-1)
\end{align*}
\]
Quick Check

Write a recursive definition of $f(n) = \frac{(n+1)n}{2}$, where $n > 0$. 
Quick Check

Write a recursive definition of \( f(n) = \frac{(n + 1)n}{2} \), where \( n > 0 \).

\[
\begin{align*}
  f(1) &= 1 \\
  f(n+1) &= \frac{(n + 2)(n + 1)}{2} = \frac{(n + 1)n}{2} + \frac{2(n+1)}{2} \\
  &= f(n) + (n + 1)
\end{align*}
\]
Outline

Recursive Thinking
Recursive Programming
Traversing a Maze
The Towers of Hanoi
Tiled Images
Fractals
A (non recursive) Method

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

    // -----------------------------------------------------------------------
    // Gets the value of each slider, then updates the labels and
    // the color panel.
    // -----------------------------------------------------------------------

    public void stateChanged (ChangeEvent event) {
        red = rSlider.getValue();
        green = gSlider.getValue();
        blue = bSlider.getValue();

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

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

• **A recursive method** is a method that **invokes** itself

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

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

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

- Consider the problem of computing the sum of all the numbers between 1 and any positive integer N.
- This problem can be recursively defined as:

\[
\sum_{i=1}^{N} i = N + \sum_{i=1}^{N-1} i = N + (N - 1) + \sum_{i=1}^{N-2} i = \ldots = N + (N - 1) + (N - 2) + \sum_{i=1}^{N-3} i = \ldots = N + (N - 1) + (N - 2) + \ldots + 2 + 1
\]
Sum of 1 to N

- The summation could be implemented recursively as follows:

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

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

    return result;
}
```
public int sum (int num) {
    int result;

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

    return result;
}
Recursive Programming

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

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

• However, for some problems, recursion provides an elegant solution, often cleaner than an iterative version (e.g. Fibonacci)

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

• Write a recursive method that computes the factorial of a **non-negative** int number n:
  factorial(0)=1, factorial(n) = n * factorial(n-1)
Factorial

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

factorial(4)

factorial(3)

factorial(2)

factorial(1)

factorial(0)

return 1

return 1*1 = 1

return 2*1 = 2

return 3*2 = 6

return 4*6 = 24
Indirect Recursion

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

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

• For example, method $m_1$ could invoke $m_2$, which invokes $m_3$, which in turn invokes $m_1$ again.

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

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

L is the left propagation and R is the right propagation (they are two functions that map a non negative integer into a non negative integer):

• $L(n) = L(R(n-1))$
• $R(n) = R(L(n-1))$
• $L(1) = R(1) = k$

• For which values of $k>0$, the $L(n)$ and $R(n)$ computations terminate for all $n>0$?
Quiz

L is the left propagation and R is the right propagation:
• \( L(n) = L(R(n-1)) \)
• \( R(n) = R(L(n-1)) \)
• \( L(1) = R(1) = k \)

• For which values of \( k > 0 \), the \( L(n) \) and \( R(n) \) computations terminate for all \( n > 0 \)?
• **Only \( k = 1 \)** (and \( L \) and \( R \) are constant functions = 1)
• In fact, if \( k = 2 \) then, for instance if \( n = 2 \):
  • \( L(2) = L(R(1)) = L(2) \) infinite loop
• If \( k = 3 \) then, for instance if \( n = 2 \):
  • \( L(2) = L(R(1)) = L(3) = L(R(2)) = L(R(L(1))) = L(R(3)) = L(R(L(2))) \) infinite loop
• If \( k = 4 \) …
Exercise

• Implement a class Chap12 that contains the right and left (static) methods described before and test that there is a infinite loop if k>1

• Add a System.out.println("left") (or "righ") as first statement of the left and right methods; to trace the execution.
public class Chap12 {

    private final static int K = 1;

    public static int right(int n) {
        System.out.println("right");
        if (n == 1)
            return K;
        return right(left(n-1));
    }

    public static int left(int n) {
        System.out.println("left");
        if (n == 1)
            return K;
        return left(right(n-1));
    }

    public static void main(String[] args) {
        System.out.println(right(2));
    }

}
Quiz

• What does the following recursive function return? Try it when the parameter s is your name.

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

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

The reverse of the input string.
Mathematical Induction

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

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

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

• Base case:
  \[ 1 = (1+1)1/2 \text{ TRUE} \]

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

• Prove that:
  \[ 1 + 2 + 3 + 4 + \ldots + N = S_n = (N+1)N/2 \]

• \[(1 + 2 + 3 + 4 + \ldots + N) + (1 + 2 + 3 + 4 + \ldots + N) = 2S_n\]
• \[(1 + 2 + 3 + 4 + \ldots + N) + (N + (N-1) + \ldots + 1) = 2S_n\]
• \[(1 + N) + (2 + N-1) + (3 + N-2) + \ldots + (N + 1) = 2S_n\]
• \[N(N+1) = 2S_n\]
• \[S_n = (N+1)N/2\]
Quiz

• Consider the fibonacci sequence:
  – $f(0) = 0, f(1) = 1, f(n) = f(n-1) + f(n-2)$
  – $f(0)=0, f(1)=1, f(2) = f(1)+f(0) = 1, f(3) = f(2)+f(1) = 2, f(4) = 3, f(5) = 5, f(6) = 8, f(7) = 13, ..$

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

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

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

• Base case $n=1$
  – $f(3) = 2$ TRUE

• Induction step
  – If $f(3n)$ is even we must prove that $f(3(n+1))$ is even
  – $f(3(n+1)) = f(3n+2) + f(3n+1) = f(3n+1) + f(3n) + f(3n+1) = 2f(3n+1) + f(3n)$ THIS is EVEN
Recursion can be inefficient

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

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

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

How many calls to `fibonacci` for computing `fibonacci(5)`? 15 calls
Iterative version of Fibonacci

```c
int itFibonacci(int n) {
    int result = 0, prec = 1;
    for (int i = 1; i <= n; i++) {
        result += prec;  // f(i) = f(i-1) + f(i-2)
        prec = result - prec;  // f(i-1) = f(i) - f(i-2)
    }
    return result;
}
```

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

Recursive Thinking
Recursive Programming
Traversing a Maze
The Towers of Hanoi
Tiled Images
Fractals
Maze Traversal

- We can use recursion to find a path through a maze
- From each location, we can search in each direction
- The recursive calls keep track of the path through the maze
- The base case is an invalid move or reaching the final destination

- See MazeSearch.java
- See Maze.java
public class MazeSearch
{
//-----------------------------------------------------------------
//  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);
}
}
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);
    }
}
public class Maze
{
    private final int TRIED = 3;
    private final int PATH = 7;

    private int[][] grid = {
        {1,1,1,0,1,0,0,0,1,1,1,1},
        {1,0,1,1,1,0,1,1,1,1,0,1},
        {0,0,0,1,0,1,0,1,0,1,0,0},
        {1,1,1,0,1,1,1,0,1,0,1,1},
        {1,0,1,0,0,0,0,1,1,1,0,0},
        {1,0,1,1,1,1,1,1,0,1,1,1},
        {1,0,0,0,0,0,0,0,0,0,0,0},
        {1,1,1,1,1,1,1,1,1,1,1,1}
    };

    continued
public boolean traverse (int row, int column) {
    boolean done = false;

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

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

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

    return done;
}    continued
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

// 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;
    }
}
Quiz

• Trace the calls to traverse() for the maze row[0]={1, 1}, row[1]={0, 1}
return true

traverse(0,0)

return false

traverse(1,0)

return true

traverse(0,1)

return true

traverse(1,1)
Quiz

- More elaborated trace the calls to `traverse()` and `valid()` for the maze row[0]=\{1, 1\}, row[1]=\{0, 1\}
Outline

Recursive Thinking
Recursive Programming
Traversing a Maze
The Towers of Hanoi
Tiled Images
Fractals
Towers of Hanoi

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

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

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

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

Original Configuration

Move 1

Move 2

Move 3

Target
Towers of Hanoi

Move 4

Move 5

Move 6

Move 7 (done)
Recursive Solution

• To solve a N-tower

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

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

- See SolveTowers.java
- See TowersOfHanoi.java
public class SolveTowers
{
    public static void main (String[] args)
    {
        TowersOfHanoi towers = new TowersOfHanoi (4);
        towers.solve();
    }
}
public class SolveTowers {
    // Demonstrates recursion.

    public static void main(String[] args) {
        TowersOfHanoi towers = new TowersOfHanoi(4);
        towers.solve();
    }
}

Output
Move one disk from 1 to 2
Move one disk from 1 to 3
Move one disk from 2 to 3
Move one disk from 1 to 2
Move one disk from 3 to 1
Move one disk from 3 to 2
Move one disk from 1 to 2
Move one disk from 1 to 3
Move one disk from 2 to 3
Move one disk from 2 to 1
Move one disk from 3 to 1
Move one disk from 2 to 3
Move one disk from 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
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
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);
    }
}

private void moveOneDisk (int start, int end) {
    System.out.println ("Move one disk from " + start + " to " + end);
}
Hanoi Tower Solution

Hanoi Tower execution time (seconds)

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

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

Trace the recursive calls.

A) 11
B) 8
C) 5
D) 3
E) 24
public int mystery(int x, int y) {
    if (x == y) return 0;
    else return mystery(x - 1, y) + 1;
}

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

C) 5

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

- `mistery(8,3)`
  - return 5
- `mistery(7,3)`
  - return 4
- `mistery(6,3)`
  - return 3
- `mistery(5,3)`
  - return 2
- `mistery(4,3)`
  - return 1
- `mistery(3,3)`
  - return 0
Quiz

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

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

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

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

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

What does the following method compute? Assume the method is called initially with a string a, a char b and i = 0

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

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

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

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

Recursive Thinking
Recursive Programming
Traversing a Maze
The Towers of Hanoi
Tiled Images
Fractals
Tiled Images

• 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 TiledImages.java
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.effect.ColorAdjust;
import javafx.scene.effect.SepiaTone;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

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

public class TiledImages extends Application {

    private final static int MIN = 20;

    private Image image;
    private ColorAdjust monochrome;
    private SepiaTone sephia;
    private Group root;

    continue
// Sets up the display of a series of tiled images.
public void start(Stage primaryStage)
{
    image = new Image("girl.jpg");

    monochrome = new ColorAdjust(0, -1, 0, 0); //hue, saturation,
  //brightness, contrast
    sepia = new SepiaTone();

    root = new Group();
    addPictures(300);

    Scene scene = new Scene(root, 600, 600, Color.WHITE);

    primaryStage.setTitle("Tiled Images");
    primaryStage.setScene(scene);
    primaryStage.show();
}
private void addPictures(double size) {
    ImageView colorView = new ImageView(image);
    colorView.setFitWidth(size);
    colorView.setFitHeight(size);
    colorView.setX(size); // bottom right sector
    colorView.setY(size);

    ImageView monochromeView = new ImageView(image);
    monochromeView.setEffect(monochrome);
    monochromeView.setFitWidth(size);
    monochromeView.setFitHeight(size);
    monochromeView.setX(0); // bottom left sector
    monochromeView.setY(size);

    ImageView sepiaView = new ImageView(image);
    sepiaView.setEffect(sepia);
    sepiaView.setFitWidth(size);
    sepiaView.setFitHeight(size);
    sepiaView.setX(size); // top right sector
    sepiaView.setY(0);
}
continue

    root.getChildren().addAll(sepiaView, colorView, monochromeView);

    if (size > MIN)
        addPictures(size / 2);
    
}
continue

root.getChildren().addAll(sepiaView, colorView, monochromeView);

if (size > MIN) addPictures(size / 2);
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.text.Text;
import javafx.stage.Stage;

//************************************************************************
//                KochSnowflake.java        Author: Lewis/Loftus
//
//    Demonstrates the use of recursion to draw a fractal.
//************************************************************************

public class KochSnowflake extends Application {
    
    private final static int MIN_ORDER = 1;
    private final static int MAX_ORDER = 6;

    private int order;
    private Button up, down;
    private Text orderText;
    private KochPane fractalPane;

    continue
Displays two buttons that control the order of the fractal shown in the pane below the buttons.

```java
public void start(Stage primaryStage) {
    Image upImage = new Image("up.png");
    up = new Button();
    up.setGraphic(new ImageView(upImage));
    up.setOnAction(this::processUpButtonPress);

    Image downImage = new Image("down.png");
    down = new Button();
    down.setGraphic(new ImageView(downImage));
    down.setOnAction(this::processDownButtonPress);
    down.setDisable(true);

    order = 1;
    orderText = new Text("Order: 1");

    HBox toolbar = new HBox();
    toolbar.setStyle("-fx-background-color: darksalmon");
    toolbar.setAlignment(Pos.CENTER);
    toolbar.setPrefHeight(50);
    toolbar.setSpacing(40);
    toolbar.getChildren().addAll(up, orderText, down);
}
```
fractalPane = new KochPane();

VBox root = new VBox();
root.setStyle("-fx-background-color: white");
root.getChildren().addAll(toolbar, fractalPane);

Scene scene = new Scene(root, 400, 450);
primaryStage.setTitle("Koch Snowflake");
primaryStage.setScene(scene);
primaryStage.show();
}

//--------------------------------------------------------------------
// Increments the fractal order when the up button is pressed.
// Disables the up button if the maximum order is reached.
//--------------------------------------------------------------------
public void processUpButtonPress(ActionEvent event)
{
    order++;
    orderText.setText("Order: " + order);
    fractalPane.makeFractal(order);

    down.setDisable(false);
    if (order == MAX_ORDER)
        up.setDisable(true);
}
public void processDownButtonPress(ActionEvent event)
{
    order--;
    orderText.setText("Order: "+ order);
    fractalPane.makeFractal(order);

    up.setDisable(false);
    if (order == MIN_ORDER)
        down.setDisable(true);
}
```java
public void processDownButtonPress(ActionEvent event) {
    order --;
    orderText.setText("Order: " + order);
    fractalPane.makeFractal(order);
    up.setDisable(false);
    if (order == MIN_ORDER)
        down.setDisable(true);
}
```
Koch Snowflakes

\[ \langle x_5, y_5 \rangle \rightarrow \langle x_1, y_1 \rangle \]

Becomes

\[ \langle x_5, y_5 \rangle \rightarrow \langle x_4, y_4 \rangle \rightarrow \langle x_3, y_3 \rangle \rightarrow \langle x_2, y_2 \rangle \rightarrow \langle x_1, y_1 \rangle \]

\[ = \sqrt{3/6} \cdot |P_5 - P_1| \]
import javafx.scene.layout.Pane;
import javafx.scene.shape.Line;

//************************************************************************
// KochPane.java       Author: Lewis/Loftus
//
// Represents the pane in which the Koch Snowflake fractal is presented.
//************************************************************************

public class KochPane extends Pane
{
    public final static double SQ = Math.sqrt(3) / 6;

    //--------------------------------------------------------------------
    // Makes an initial fractal of order 1 (a triangle) when the pane
    // is first created.
    //--------------------------------------------------------------------
    public KochPane()
    {
        makeFractal(1);
    }
}

continue
public void makeFractal(int order)
{
    getChildren().clear();
    addLine(order, 200, 20, 60, 300);
    addLine(order, 60, 300, 340, 300);
    addLine(order, 340, 300, 200, 20);
}

public void addLine(int order, double x1, double y1, double x5, double y5)
{
    double deltaX, deltaY, x2, y2, x3, y3, x4, y4;
continue

    if (order == 1)
    {
        getChildren().add(new Line(x1, y1, x5, y5));
    }
    else
    {
        deltaX = x5 - x1;  // distance between the end points
        deltaY = y5 - y1;

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

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

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

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

• Chapter 12 has focused on:
  – thinking in a recursive manner
  – programming in a recursive manner
  – the correct use of recursion
  – recursion examples
  – recursion in graphics
  – fractals
Exercise

• Write a recursive definition of a valid Java identifier (see Chapter 1). Imagine that a Java Letter is either a letter of the English alphabet or $ or _

Identifier

Java Letter → Java Letter → Java Digit → Java Letter →
Exercise

• Write a recursive definition of a valid Java identifier (see Chapter 1). Imagine that a Java Letter is either a letter of the English alphabet or $ or _

A Java-Identifier is:

1. a: Letter
2. or a: Java-Identifier followed by a Letter
3. or a: Java-Identifier followed by a digit
Exercise

• Write a recursive definition of \( f(n) = n \times k \) (integer multiplication), where \( n > 0 \).

• Define the multiplication process in terms of integer addition.
  – For example, \( 4 \times 7 \) is equal to 7 added to itself 4 times.
Exercise

• Write a recursive definition of \( f(n) = n \times k \) (integer multiplication), where \( n > 0 \).
• Define the multiplication process in terms of integer addition.
  – For example, \( 4 \times 7 \) is equal to 7 added to itself 4 times.

• \( 1 \times k = k \), if \( n=1 \)
• \( n \times k = k + (n-1) \times k \) for \( n > 1 \)
Exercise

• Modify the method that calculates the sum of the integers between 1 and N shown in this lecture. Have the new version match the following recursive definition of the sum of the integers between n and m (n<=m):
  – Sum(n, m) = n, if n=m
  – Sum(n, m) = Sum(n, (n+m)/2) + Sum((n+m)/2 +1, m), if n<m

• Trace your solution using for N = 7 (n=1, m=7).
public int sum(int n1, int n2) {
    int result;
    if (n2 - n1 == 0) {
        result = n1;
    } else {
        int mid = (n1 + n2) / 2;
        result = sum(n1, mid) + sum(mid + 1, n2);
    }
    return result;
}
Trace

```
        sum(1,7)
       /    \
  sum(1,4)  sum(5,7)
 /     |       |
sum(1,2)  sum(3,4)  sum(5,6)
|     |     |     |
sum(1,1) sum(2,2) sum(3,3) sum(4,4)
|     |     |     |
sum(1,1) sum(2,2) sum(3,3) sum(4,4)
```

Exercise

• Write (another) recursive method to reverse a string.

• Use the following String methods
  – charAt(int n) : char
  – substring(int beginIndex, int endIndex) : String

• Implement a procedure that concatenates the last character of the input string with the (recursive) reverse of the string composed by the first N-1 characters of the string (N is the length of the string).
public String reverse(String text) {
    String result = text;
    if (text.length() > 1)
        result = text.charAt(text.length() - 1) +
                 reverse(text.substring(0, text.length() - 1));
    return result;
}
Tower of Hanoi

• Produce a chart showing the **number** of moves required to solve the Towers of Hanoi puzzle using the following number of disks: 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10

• Write a recursive definition of the formula giving Moves(n), the number of moves required to solve the Hanoi tower of n disks
## Solution

<table>
<thead>
<tr>
<th>Disks</th>
<th>Moves</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
</tr>
<tr>
<td>6</td>
<td>63</td>
</tr>
<tr>
<td>7</td>
<td>127</td>
</tr>
<tr>
<td>8</td>
<td>255</td>
</tr>
<tr>
<td>9</td>
<td>511</td>
</tr>
<tr>
<td>10</td>
<td>1023</td>
</tr>
</tbody>
</table>

\[
\text{Moves}(1) = 1 \\
\text{Moves}(n) = \text{Moves}(n-1) + 1 + \text{Moves}(n-1) \\
\text{Moves}(n) = 2 \times \text{Moves}(n-1) + 1
\]
Kock Snowflake

• How many line segments are used to construct a Koch snowflake of order N? Produce a chart showing the number of line segments that make up a Koch snowflake for orders 1 through 7.

• Give the general formula $\text{Segments}(n) = ?$
## Solution

<table>
<thead>
<tr>
<th>Order</th>
<th>Segments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4*3 = 12</td>
</tr>
<tr>
<td>3</td>
<td>4<em>4</em>3 = 48</td>
</tr>
<tr>
<td>4</td>
<td>4<em>4</em>4*3 = 192</td>
</tr>
<tr>
<td>5</td>
<td>4<em>4</em>4<em>4</em>3 = 768</td>
</tr>
<tr>
<td>6</td>
<td>4<em>4</em>4<em>4</em>4*3 = 3072</td>
</tr>
<tr>
<td>7</td>
<td>4<em>4</em>4<em>4</em>4<em>4</em>3 = 12288</td>
</tr>
</tbody>
</table>

Segments(1) = 3  
Segments(n) = Segments (n-1) * 4  

Segments(n) = 3*4^{n-1}
Exercise

• Give the value of mistery2(3):

```java
public static String mistery2(int n) {
    if (n <= 0)
        return "";
    return mistery2(n - 3) + n + mistery2(n - 2) + n;
}
```
misery2(3)

misery2(0)

misery2(1)

misery2(-2)

misery2(-1)

return ""

return ""+3+"11"+3 = "3113"
Exercise

• Criticize the following recursive function:

```java
public static String mistery3(int n) {
    String s = mistery3(n - 3) + n + mistery3(n - 2) + n;
    if (n <= 0)
        return ""
    return s;
}
```