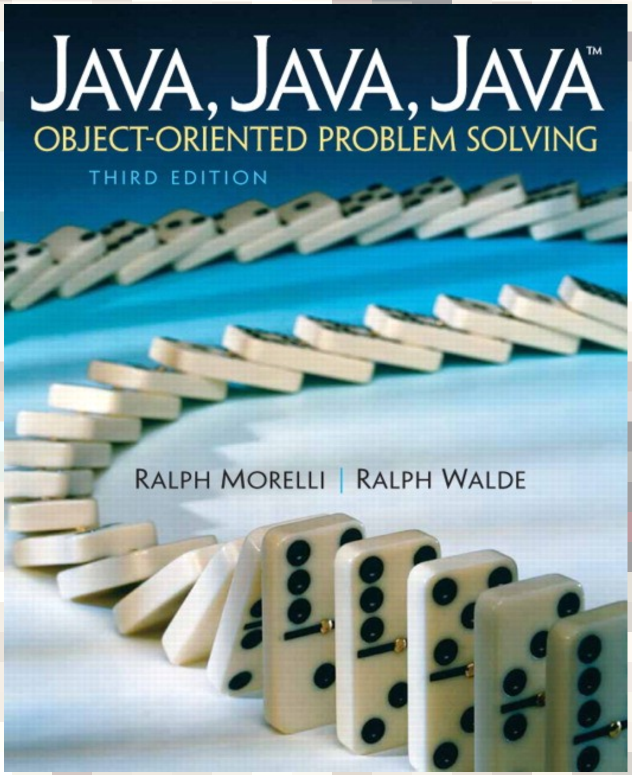


presentation slides for

JAVA, JAVA, JAVA

Object-Oriented Problem Solving

Third Edition

Ralph Morelli | Ralph Walde
Trinity College
Hartford, CT

published by Prentice Hall

Java, Java, Java

Object Oriented Problem Solving

Lecture 11: Recursive Problem Solving (Chapter 12)

Objectives

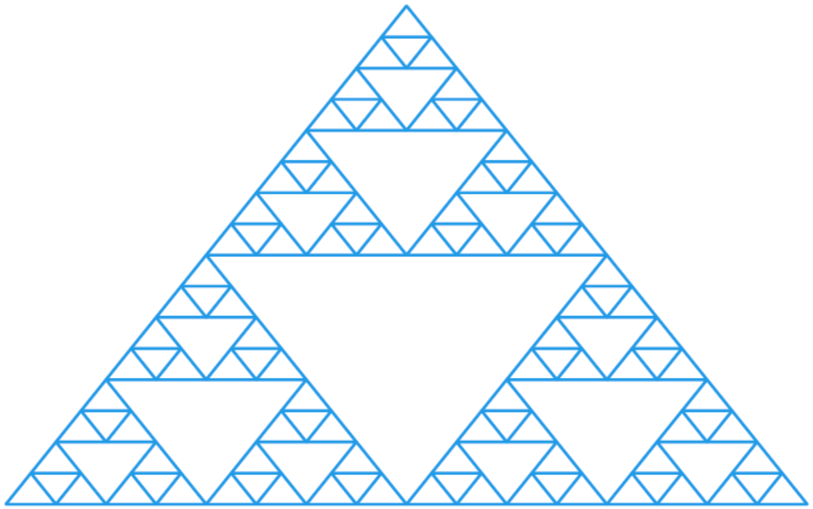
- Understand the concept of recursion.
- Know how to use recursive programming techniques.
- Have a better appreciation of recursion as a problem solving technique.

Outline

- Introduction
- Recursive Definition
- Recursive String Methods
- Recursive Array Processing
- Drawing (Recursive) Fractals
- Object-Oriented Design: Tail Recursion
- Object-Oriented Design: Recursion or Iteration?

Introduction

- The Sierpinski gasket (or Sierpinski triangle) is a *fractal* pattern, a structure that exhibits *self-similarity* at every scale.



- Chapter goal: recursion as a problem-solving technique *and* as alternative to loops.
- Divide-and-Conquer Approach: Divide the gasket into smaller, self-similar patterns.

Recursion as Repetition

- *Recursive method:* a method that calls itself.
- *Iterative method:* a method that uses a loop.
- hello(N) prints "Hello" *N* times.

```
public void hello(int N)  {
    for (int k = 0; k < N; k++)
        System.out.println("Hello");
} // hello()
```

```
public void hello(int N)  {
    if (N > 0) {
        System.out.println("Hello");
        hello(N - 1);
    }
} // hello()
```

Iterative, repeats the loop *N* times.

Recursive, calls itself *N* times.

Trace of Recursive hello()

- Trace of hello(5) :

```
public void hello(int N)  {
    if (N > 0) {
        System.out.println("Hello");
        hello(N - 1);
    }
} // hello()
```

Stop when *N = 0.*

Decrement *N* on each recursive call.

```
hello(5)
    Print "Hello"
    hello(4)
        Print "Hello"
        hello(3)
            Print "Hello"
            hello(2)
                Print "Hello"
                hello(1)
                    Print "Hello"
                    hello(0)
```

Recursion versus Iteration

- Recursion is an alternative to iteration.
- Languages like LISP and PROLOG use recursion instead of loops.
- Loops use less *computational overhead* than method calls.
- **Effective Design: Efficiency.** Iterative algorithms are generally more efficient than recursive algorithms that do the same thing. However, recursive algorithms are more intuitive, clean, simpler, short code, and easy to understand.

Recursion as a Problem Solving

- **Divide-and-conquer**: Divide the problem into smaller problems.
- **Self-similarity**: Each subproblem is like the original problem.
- Factorial Example:

$$n! = n * (n-1) * (n-2) * \ldots * 1$$

```
4! = 4 * 3 * 2 * 1          = 4 * 3! = 24
3! = 3 * 2 * 1              = 3 * 2! = 6
2! = 2 * 1                  = 2 * 1! = 2
1!                                  = 1 * 0! = 1
0!                                  = 1
```

Recursion:
*4! = 4 * 3!*

Recursive Definition

- Recursive definition of *factorial(n)*:

  n! = 1 if n = 0 // Boundary (or base) case

  n! = n * (n-1)! if n > 0 // Recursive case

  Divide-and-Conquer: *n!* is defined in terms of a smaller, self-similar problem, *(n-1)!*

- **Effective Design: Recursive Definition.** The *base case* serves as the bound for the algorithm. The *recursive case* defines the n^{th} case in terms of the $(n-1)^{th}$ case.

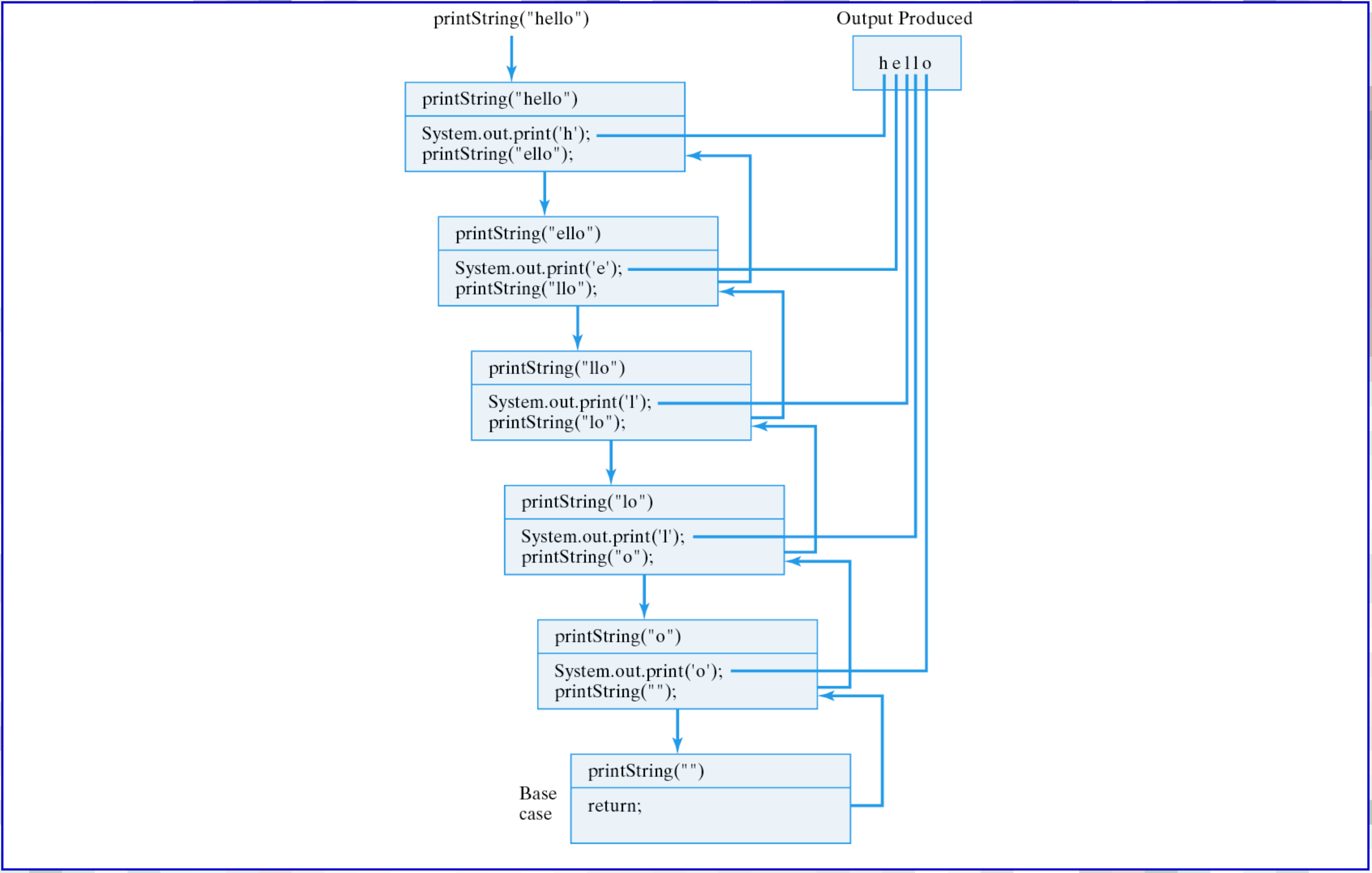
Recursive printString(s)

- *Head-and-tail recursion:* print the *head*, and *recursively* print the *tail* of the string.
- *Head of String s:* s.charAt(0)
- *Tail of String s:* s.substring(1)
- For *Hello*, print *H* and recursively print *ello*.

```
public void printString(String s) {
     if (s.length() == 0)
         return;                          // Base case: do nothing
     else {
         System.out.print(s.charAt(0));   // Recursive case: print head
         printString(s.substring(1));     // Print tail of the string
     }
} // printString()
```

Recursion parameter.

Tracing printString("hello")



Effective Design

- **Recursive Progress.** Each recursive call must make progress toward the bound, or base case.
- **Recursion and Parameters.** Recursive methods use a *recursion parameter* to control progress toward its bound.
- **Head/Tail Algorithm.** Divide a sequential structure into its *head* and *tail*. Process the head, and recurse on the tail.

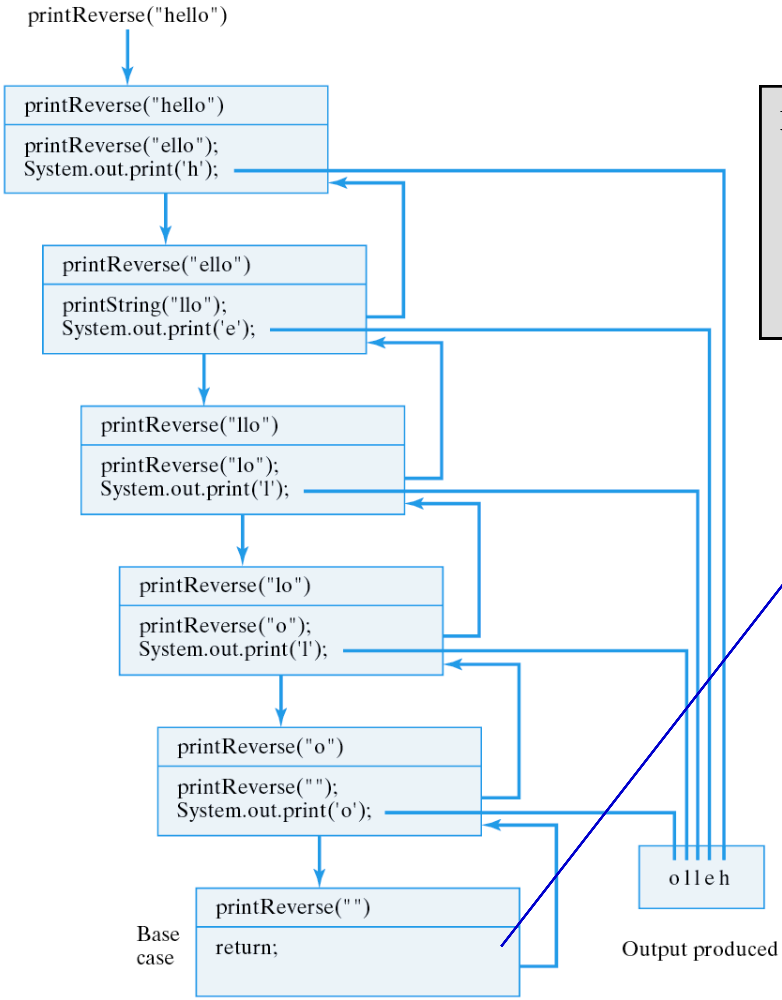
Printing the String Backwards

- Reverse Head/Tail: Recursively process the tail, *then* process the head:

```
public void printReverse(String s) {
     if (s.length() > 0) {                         // Recursive case:
          printReverse(s.substring(1));            // Print tail of the string
          System.out.print(s.charAt(0));           // Then print the first char
     }
} // printReverse()
```

- For *Hello*, recursively printing *ello* then printing *H* will give *olleH*

Tracing printReverse("hello")




```
public void printReverse(String s) {
     if (s.length() > 0) {
          printReverse(s.substring(1));
          System.out.print(s.charAt(0));
     }
} // printReverse()
```

Last-In-First-Out (LIFO). Procedure call follows a LIFO protocol.

Counting Characters in a String

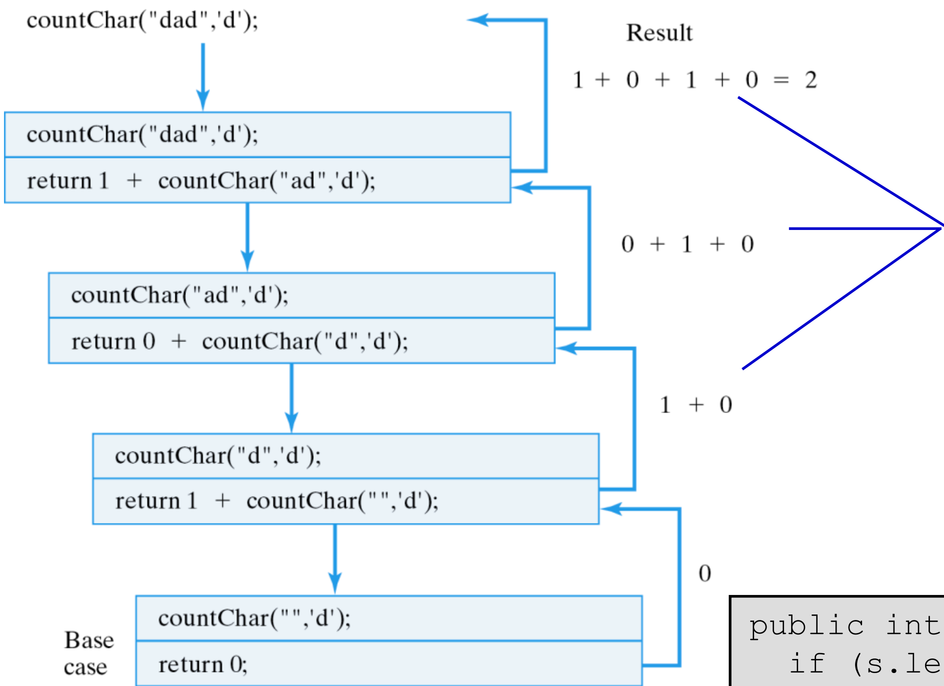
- Count the number of ***ch***'s in a string *s*.
- Algorithm: Count the head of *s* then recursively count the *ch*'s in the tail of *s*.

Embedded recursive call

```
/**
  * Pre:  s is a non-null String, ch is any character
  * Post: countChar() == the number of occurrences of ch in s
  */
public int countChar(String s, char ch) {
    if (s.length() == 0)                            // Base case: empty string
        return 0;
    else if (s.charAt(0) == ch)                     // Recursive case 1
        return 1 + countChar(s.substring(1), ch);   // Head equals ch
    else                                            // Recursive case 2
        return 0 + countChar(s.substring(1), ch);   // Head is not the ch
} // countChar()
```

Return is done *after* the recursive call.

Tracing countChar("dad", 'd')



Embedded recursive call: the results are accumulated.

```
public int countChar(String s, char ch) {
  if (s.length() == 0)
      return 0;
  else if (s.charAt(0) == ch)
      return 1 + countChar(s.substring(1), ch);
  else
      return 0 + countChar(s.substring(1), ch);
} // countChar()
```

Converting a String

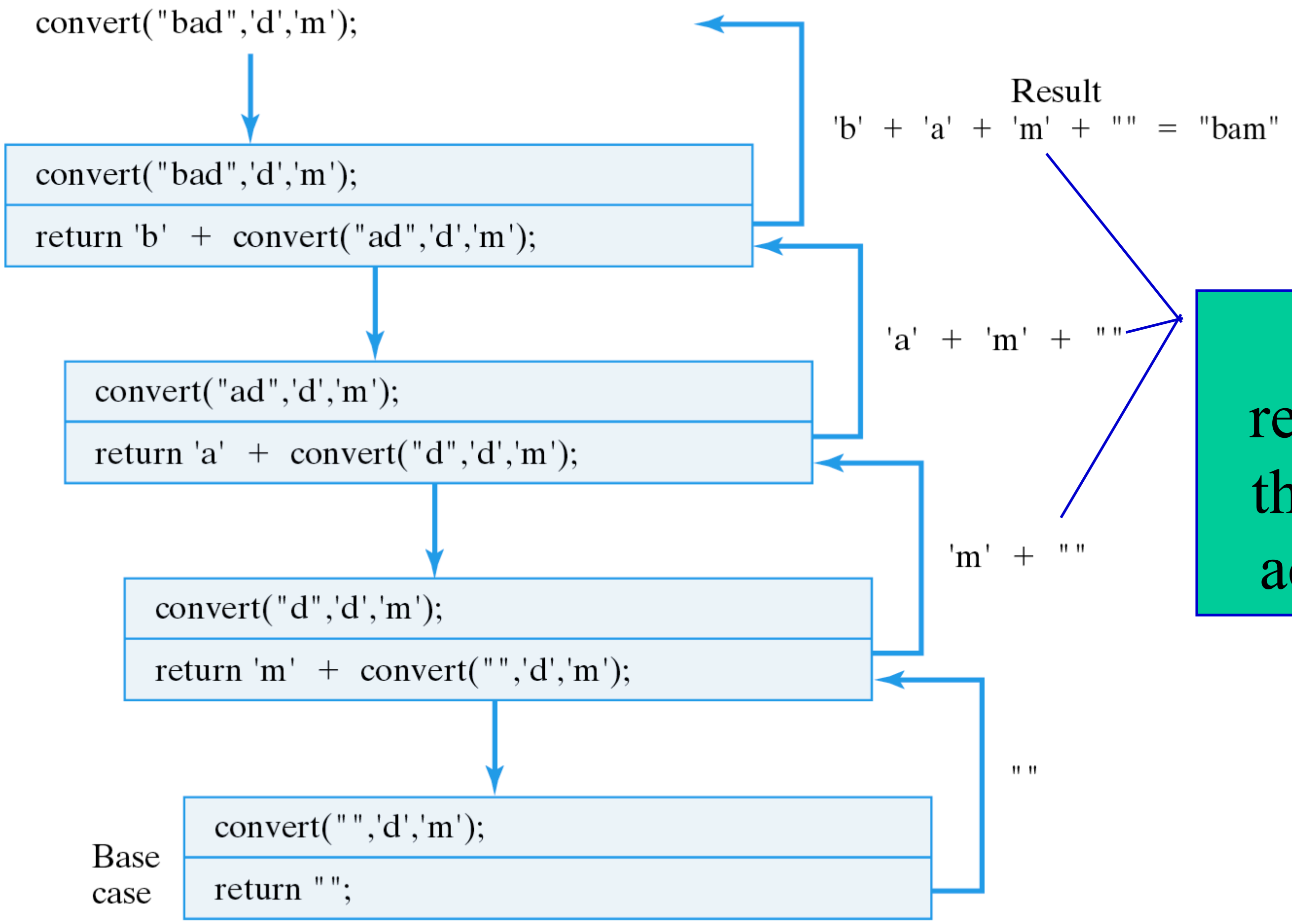
- Replace every occurrence of *ch1* with *ch2* in the string *str.*
- Algorithm: If necessary, replace the head then recursively convert the tail.

Embedded recursive call

```
public static String convert(String str, char ch1, char ch2) {
    if (str.length() == 0)                           // Base case: empty string
        return str;
    else if (str.charAt(0) == ch1)                   // Recursive 1: ch1 at head
        return ch2 + convert(str.substring(1), ch1, ch2);   // Replace it
    else                                             // Recursive 2: ch1 not at head
        return str.charAt(0) + convert(str.substring(1), ch1, ch2);
} // convert()
```

Return is done *after* the recursive call.

Tracing convert("bad",'d','m')



Embedded recursive call: the results are accumulated.

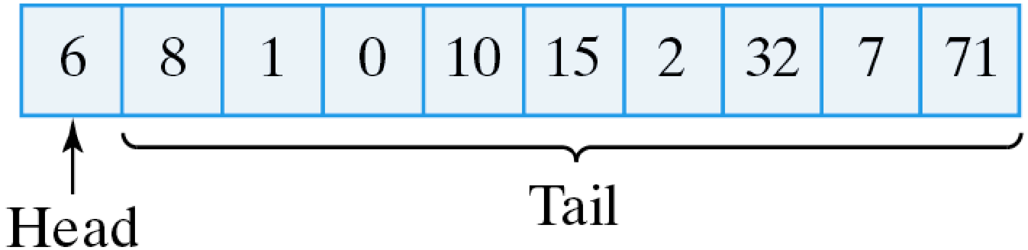
Example: Tossing *N* Coins

- Write a recursive method that prints all possible 2^N outcomes when *N* coins are tossed.

```
/**
 * printOutcomes(str, N) prints out all possible outcomes
 * beginning with str when N more coins are tossed.
 * Pre: str is a string of Hs and Ts.
 * Pre: N is a positive integer.
 * Post: none
 */
public static void printOutcomes(String str,int N){
      if (N == 1){ // The base case
          System.out.println(str + "H");
          System.out.println(str + "T");
      } else {  // The recursive case
          printOutcomes(str + "H", N - 1);
          printOutcomes(str + "T", N - 1);
      } //else
} // printOutcomes()
```

Recursive Array Processing

- Arrays also have a recursive structure.



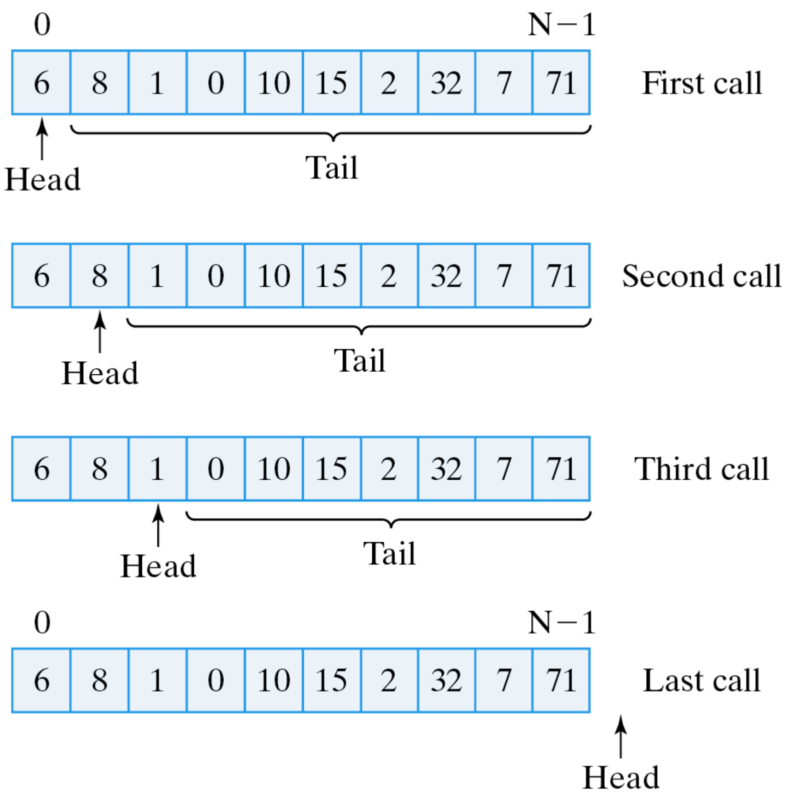
- Recursive search: check the head *then* recursively search the tail.
- Recursive sort: put the smallest number at the head *then* recursively sort the tail.

Recursive Sequential Search

- Pseudo-code:

```
If the array is empty, return -1                  // Failure
If the head matches the key, return its index     // Success
If the head doesn't match the key,                // Recursive
    return the result of searching the tail of the array
```

- Use a parameter to represent the location of the *head*. The *tail* starts at *head+1*.



The Recursive search() Method

Recursion parameter:
0 … arr.length

```
/**
  * search(arr, head, key) --- Recursively search arr for key
  *   starting at head
  * Pre:  arr != null and 0 <= head <= arr.length
  * Post: if arr[k] == key for some k,  0 <= k < arr.length, return k
  *       else return -1
  */
private int search(int arr[], int head, int key)  {
    if (head == arr.length)         // Base case: empty list - failure
        return -1;
    else if (arr[head] == key)   // Base case: key found --- success
        return head;
    else                                     // Recursive case: search the tail
        return search(arr, head + 1, key);
}
```

Information Hiding

Searcher
+ search(in arr[] : int, in key : int) : int
− search(in arr[] : int, in head : int, in key : int) : int

Same function name with different arguments.

Initialize the recursion parameter.

```
/**
  *  search(arr,key) -- searches arr for key.
  * Pre:  arr != null and 0 <= head <= arr.length
  * Post: if arr[k] == key for some k,  0 <= k < arr.length, return k
  *       else return -1
  */
public int search(int arr[], int key) {
     return search(arr, 0, key);       // Call search to do the work
}
```

- Design Issue: The user shouldn't have to call
  search(arr, 0, 25)
- Information Hiding: search(arr, 25) hides search(arr,0,25) from the user.

Effective Design

- **Information Hiding.** Unnecessary implementation details -- recursion versus iteration --should be hidden. Users of a class or method should see only what they need to know.

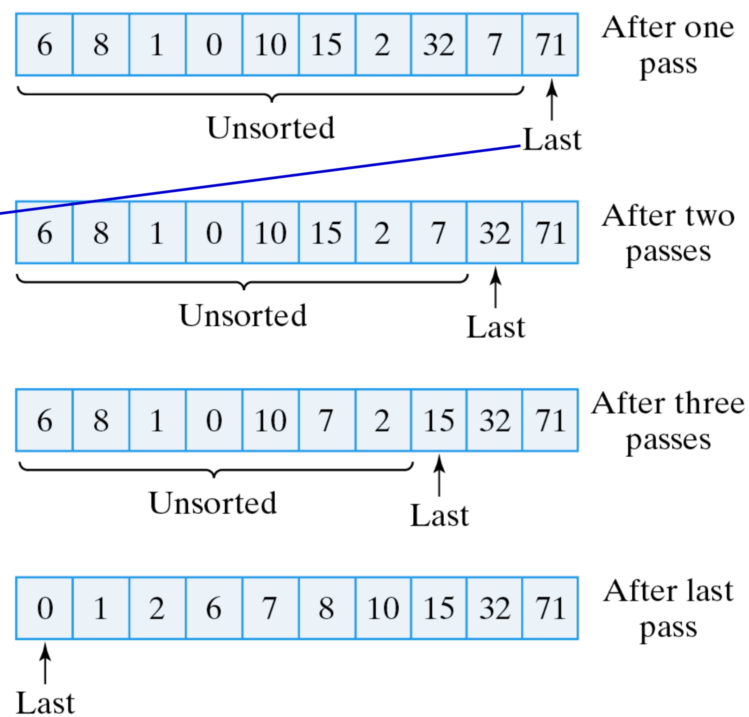
Search for 0 to 20 in the array "*numbers*"

```
public static void main(String args[]) {
    int numbers[] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18};
    Searcher searcher = new Searcher();
    for (int k = 0; k <= 20; k++) {
        int result = searcher.search(numbers, k);
        if (result != -1)
            System.out.println(k + " found at " + result);
        else
            System.out.println(k + " is not in the array ");
    } // for
} // main()
```

Recursive Selection Sort

```
private void selectionSort(int arr[], int last) {
    if (last > 0) {
        int maxLoc = findMax (arr, last);  // Find the largest element
        swap(arr, last, maxLoc);           // Put it in the last location
        selectionSort(arr, last - 1);        // Move down the array
    }
} // selectionSort()
```

Recursion parameter, *last.*



The swap() and findLast() Methods

Use a temp variable.

```
/** swap(arr0, el1, el2) swaps el1 and el2 in the arrary, arr  */
private void swap(int arr[], int el1, int el2)  {
    int temp = arr[el1];  //   Assign the first element to temp
    arr[el1] = arr[el2];   //   Overwrite first with second
    arr[el2] = temp;      //   Overwrite second with temp (i.e., first)
} // swap()
```

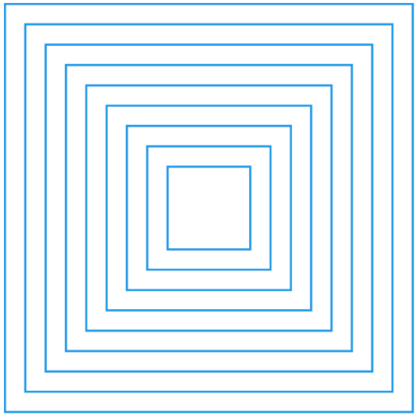
```
/** findMax(arr, N) returns the index of the largest
  *  value between arr[0] and arr[N], N >= 0.
  *  Pre: 0 <= N <= arr.length -1
  *  Post: arr[findMax()] >= arr[k] for any k between 0 and N.
  */
private int findMax(int arr[], int N) {
    int maxSoFar = 0;
    for (int k = 0; k <= N; k++)
        if (arr[k] > arr[maxSoFar])
            maxSoFar = k;
    return maxSoFar;
} // findMax()
```

Traverse the array and keep track of the index of the largest.

Drawing (Recursive) Fractals

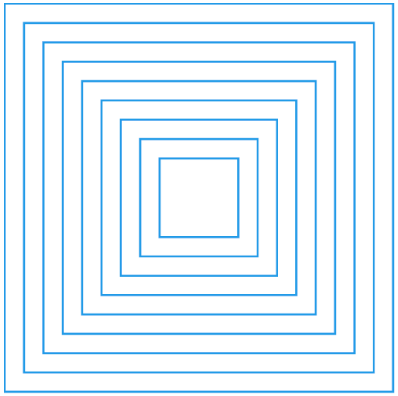
- A *fractal* is a geometric shape that exhibits a recursive structure in which each of its parts is a smaller version of the whole.
- Examples: trees, shells, coast lines.
- Nested Box Algorithm:

```
Draw a square.
If more divisions are desired
    draw a smaller nested box within the square.
```

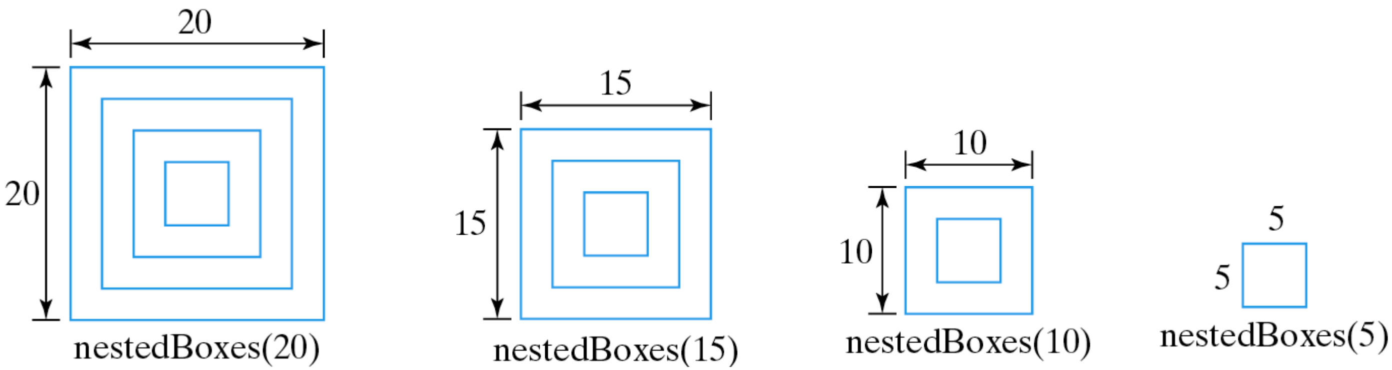


Drawing a Nested Pattern

- Draw the nested box pattern:
  - Base case: if side < 5
    do nothing
  - Recursive case: if side >= 5
    draw a square
    draw a smaller pattern inside the square



- For *nestedBoxes(20):*

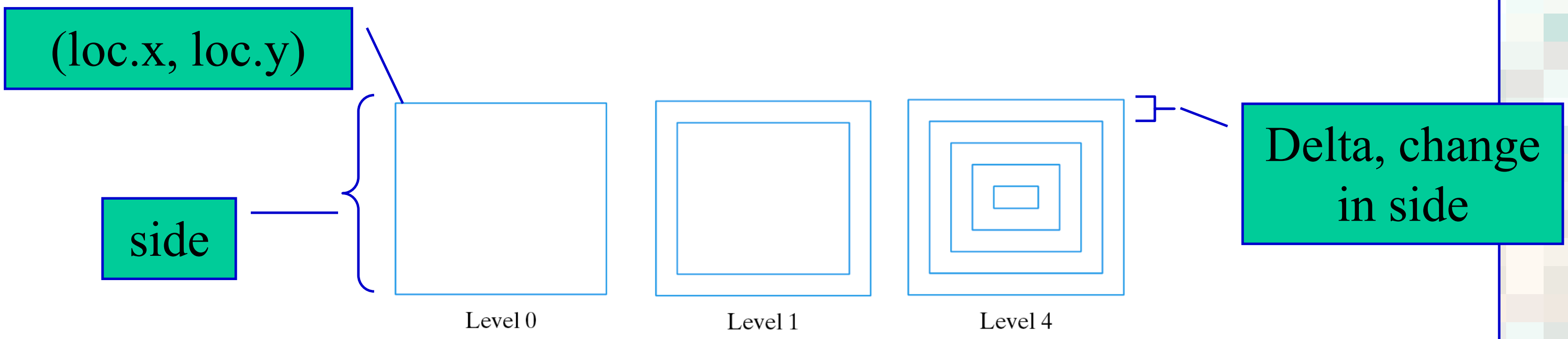


The Recursive drawBoxes() Bethod.

- Design: The *level* recursion parameter.

```
private void drawBoxes(Graphics g, int level, Point loc, int side, int delta) {
     g.drawRect(loc.x, loc.y, side, side );
     if (level > 0) {
         Point newLoc = new Point(loc.x + delta, loc.y + delta);
         drawBoxes(g, level - 1, newLoc, side - 2 * delta, delta);
     }
} // drawBoxes()
```

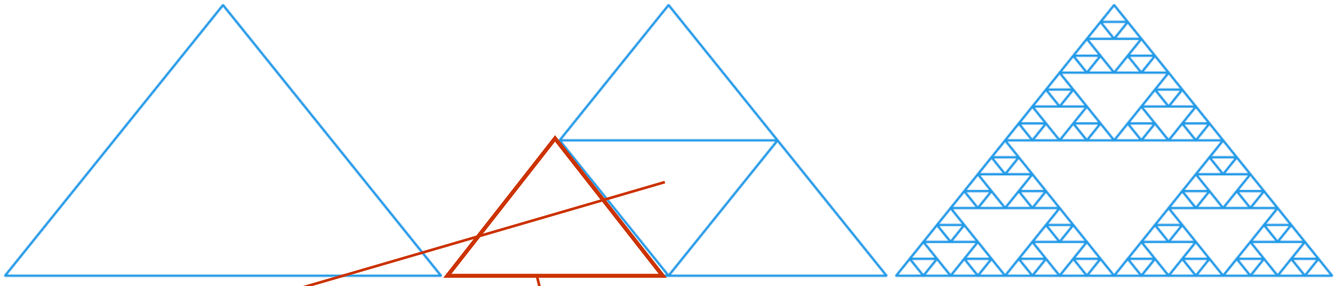
Need Graphics for drawing.



The Sierpinski Gasket

- Algorithm:

Base case: draw a triangle.
Recursive case: if level > 0,
    draw three smaller gaskets within the triangle.



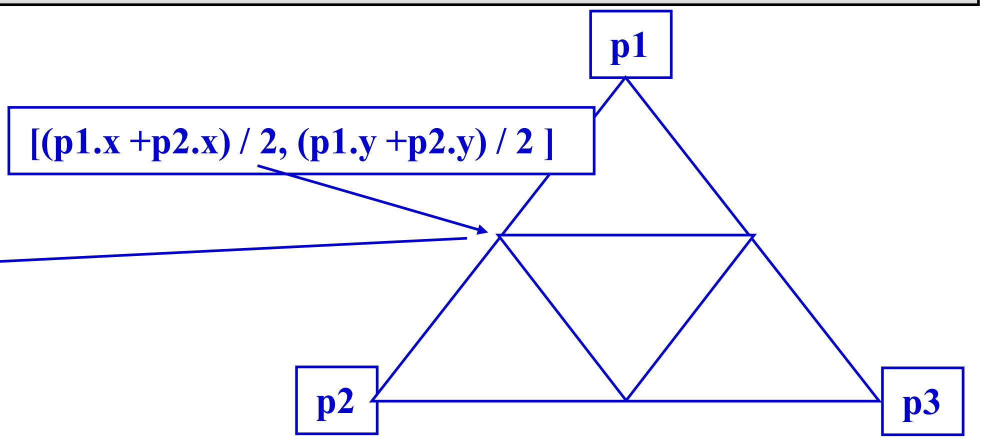
Draw a smaller triangle inside using the 3 midpoints of three sides of original triangle.

Smaller triangles at one vertex and midpoints of two sides.

The drawGasket() Method

Recursion parameter

```
private void drawGasket(Graphics g, int lev, Point p1, Point p2, Point p3) {
    g.drawLine(p1.x, p1.y, p2.x, p2.y);         // Base case: Draw a triangle
    g.drawLine(p2.x, p2.y, p3.x, p3.y);
    g.drawLine(p3.x, p3.y, p1.x, p1.y);
    if (lev > 0) {                 // If more divisions desired, draw 3 smaller gaskets
        Point midP1P2 = new Point( (p1.x + p2.x) / 2, (p1.y + p2.y) / 2 );
        Point midP1P3 = new Point( (p1.x + p3.x) / 2, (p1.y + p3.y) / 2 );
        Point midP2P3 = new Point( (p2.x + p3.x) / 2, (p2.y + p3.y) / 2 );
        drawGasket(g, lev - 1, p1, midP1P2, midP1P3);
        drawGasket(g, lev - 1, p2, midP1P2, midP2P3);
        drawGasket(g, lev - 1, p3, midP1P3, midP2P3);
    }
} // drawGasket()
```



Midpoint

Object-Oriented Design: Tail Recursion

- A *tail recursive* method is one such that its recursive calls are its last actions.

```
// print N lines of Hello
public void printHello(int N) {
    if (N > 0) {
        System.out.println("Hello");
        printHello(N - 1);
    }
} // printHello()
```

Obviously tail recursive.

```
public void printHello(int N) {
    if (N > 1) {
        System.out.println("Hello");
        printHello(N - 1);    // This will be the last executed statement
    } else
        System.out.println("Hello");
} // printHello()
```

Also tail recursive. Its last action is a recursive call.

Convert Tail Recursive to Iterative

- Tail recursive algorithms are relatively simple to convert into iterative algorithms.

```
public void printHello(int N) {
    if (N > 0) {
        System.out.println("Hello");
        printHello(N - 1);
    }
} // printHello()
```

Recursion parameter becomes loop counter

```
public void printHelloIterative(int N)
{
    for (int k = N; k > 0; k--)
        System.out.println("Hello");
}
```

- Non-tail-recursive algorithms are difficult to convert to iterative.

Design: Recursion or Iteration?

- Whatever can be done recursively can be done iteratively, and vice versa.
- Iterative algorithms use less memory.
- Iterative algorithms have less computational overhead.
- For many problems recursive algorithms are easier to design (e.g., Sierpinski Gasket).
- **Effective Design:** Unless efficiency is an issue, choose the approach that easier to understand, develop, and maintain.

Technical Terms

- base case
- computational overhead
- head-and-tail algorithm
- iterative method
- last-in-first-out (LIFO)
- method call stack
- recursion parameter
- recursive case
- recursive definition
- recursive method
- self-similarity
- tail recursive

Summary Of Important Points

- A *recursive definition* defines the *n*th case of a concept in terms of the (n-1)st case plus a *limiting condition*.
- A *recursive method* is one that calls itself. It is usually defined in terms of a *base case* and a *recursive case*
- A *recursion parameter* is generally used to to control the recursion.
- Any algorithm that can be done iteratively can also be done recursively, and vice versa.

Summary Of Important Points (cont)

- The *base case* defines a limit, and each recursive call should make progress toward the limit.
- Recursive methods use more memory and computation than iterative methods.
- A recursive method is *tail recursive* if and only if each of its recursive calls is the last action executed by the method.