Revised: 12/17/2019

Recursion

Michael C. Hackett
Computer Science Department

Community College of Philadelphia

Lecture Topics

- Recursion
  - Recursive Methods
  - Depth of Recursion
  - Designing Recursive Algorithms
  - Iterative vs. Recursive Factorial
  - Iterative vs. Recursive Fibonacci Series

- Working with Directories
  - Recursive Traversal of a Directory

Colors/Fonts

- Local Variable Names – Brown
- Primitive data types – Fuchsia
- Literals – Blue
- Keywords – Orange
- Object names – Green
- Operators/Punctuation – Black
- Field Names – Lt Blue
- Method Names – Purple
- Parameter Names – Gold
- Comments – Gray
- Package Names – Pink

Source Code – **`Consolas`**

Output – `Courier New`

Recursive Methods

- A recursive method is a method that calls itself.
  - **Recursion** is the process of solving a problem by solving smaller instances of the same problem.

```java
public void message() {
    System.out.println("This is a recursive method.");
    message();
}
```

- Calling this particular function would print *This is a recursive method.* indefinitely… Not particularly helpful.

Recursive Methods

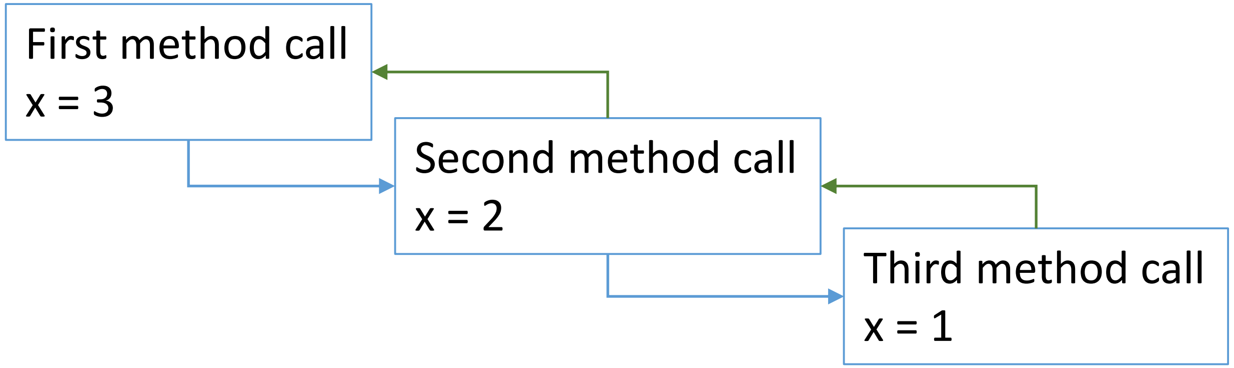
- Without any logic controlling how many times it repeats, it will repeat indefinitely.

```
public void message(int x) {
    System.out.println("This is a recursive method.");
    if(x > 1) {
        message(x - 1);
    }
}
```

Recursive Methods

- x = 3

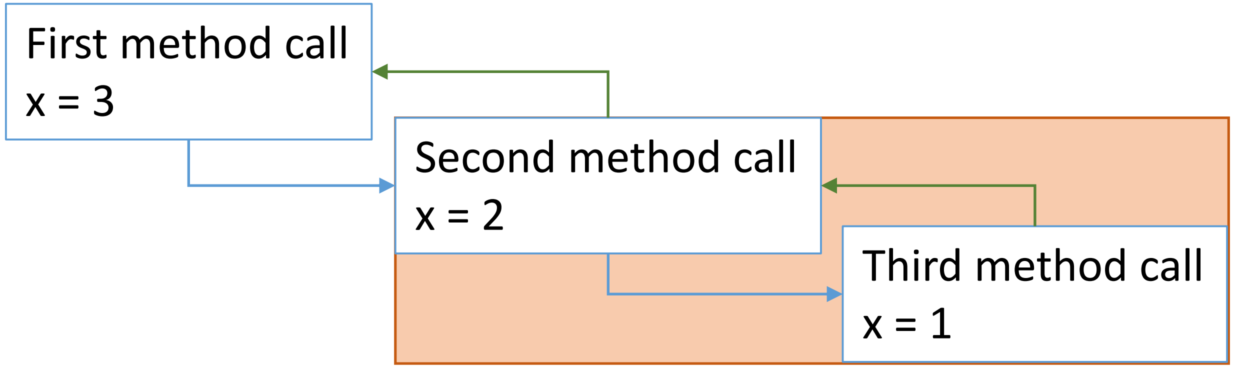
```java
public void message(int x) {
    System.out.println("This is a recursive method.");
    if(x > 1) {
        message(x - 1);
    }
}
```



If statement is false

Depth of Recursion

- The number of times a method calls itself is the *depth* of recursion.



- In the previous example, where x = 3, the depth of recursion is 2, since the message method calls itself a total of two times.
  - The first method call doesn't count towards the depth.

Solving Problems Using Recursion

- If an overall task can be broken down into successive, smaller, identical tasks, then the task can be solved using recursion.
  - This is why recursive algorithms are similar to loops (iterative algorithms).

- Any problem that can be solved using an iterative algorithm can be solved using a recursive algorithm and vice versa.
  - For this reason, recursion is never absolutely necessary.

Efficiency Issues Using Recursion

- Recursive algorithms are generally less efficient than iterative algorithms.
    - Calling methods cause several operations:
        - Has to allocate memory and store address locations of where to return control when the function has ended.
        - This is often referred to as *overhead*.

- Some repetitive problems are more easily solved using a recursive algorithm than with an iterative algorithm.
    - The iterative algorithm may have faster execution time, but for complex problems a recursive algorithm may be faster to design.

Designing a Recursive Algorithm

- How a recursive method should work:
  - If the problem can be solved now, without the need for recursion, then the method solves it and ends.
  - If the problem cannot be solved now, the method reduces the task to a smaller but similar task and calls itself to solve the smaller task.

- To begin, first find at least one case where the problem can be solved without using recursion.
  - This is the *base case*.

Designing a Recursive Algorithm

- Next, determine a way to solve the smaller task.
  - This is the recursive case.

- The recursive case should reduce the problem to a smaller version of the original problem.
  - Eventually, the problem will be reduced enough to reach the base case… the case that requires no recursion.

Designing a Recursive Algorithm

```java
public void message(int x) {
    System.out.println("This is a recursive method.");
    if(x > 1) {
        message(x - 1);
    }
}
```

- What is the problem we are trying to solve?
- How is this problem broken down into smaller tasks?
- What is the base case?
- What is the recursive case?

Designing a Recursive Algorithm

- What is the problem we are trying to solve?
  - Printing the text *“This is a recursive method.”* x-number of times.
- How is this problem broken down into smaller tasks?
  - Printing each line individually.
- What is the base case?
  - x = 1   …or, print *“This is a recursive function.”* only one time.
- What is the recursive case?
  - x > 1

Solving a Factorial

- In mathematics, the notation *n!* represents the factorial of some number, *n*.
- The factorial of a non-negative number is defined by the following rules:
  - If n = 0   n! = 1
  - If n > 0   n! = n * n-1 * n-2 * … * 1
  - Examples:
    - 0! = 1
    - 1! = 1
    - 3! = 3 * 2 * 1 = 6
    - 6! = 6 * 5 * 4 * 3 * 2 * 1 = 720

Solving a Factorial

- An iterative algorithm to solve a factorial.

```
public int factorial(int n) {
    int result = 1;
    for(int i = 1; i <= n; i++){
        result *= i;
    }
    return result;
}
```

Using Recursion to Solve a Factorial

- Let's rewrite n! so we think of it more as a method call than a mathematical expression:
  - If n = 0   factorial(n) = 1
  - If n > 0   factorial(n) = n * n-1 * n-2 * … * 1

- The overall problem we are trying to solve is the factorial of a non-negative integer.
  - How can this problem be broken down into smaller, identical tasks?
  - What is the base case?/In what situation can the factorial be solved without recursion?
  - What is the recursive case?

Using Recursion to Solve a Factorial

- The base case is n = 0.
  - If n = 0   factorial(n) = 1

- The recursive case is when n > 0.
  - If n > 0   factorial(n) = n * n-1 * n-2 * … * 1

    <u>can be rewritten as:</u>

    factorial(n) = n * factorial(n - 1)

Using Recursion to Solve a Factorial

- Recursive case:
  - If n > 0   factorial(n) = n * factorial(n - 1)

- Example:

n = 4

factorial(4) = 4 * factorial(4 - 1)

factorial(3) = 3 * factorial(3 - 1)

factorial(2) = 2 * factorial(2 - 1)

factorial(1) = 1 * factorial(1 - 1)

factorial(0) = 1

4 * 6 = 24

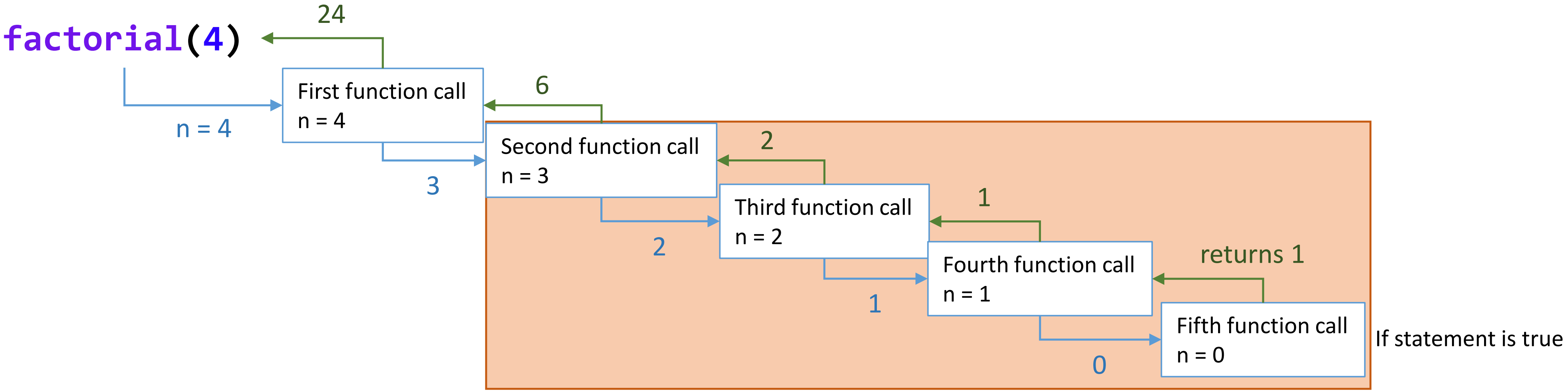
3 * 2 = 6

2 * 1 = 2

1 * 1 = 1

Using Recursion to Solve a Factorial

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



Using Recursion to Solve a Fibonacci Series

- A Fibonacci Series is a series of numbers where each number is the sum of the two previous numbers.
  - Series may begin with 0 and 1, or 1 and 1.

- First ten numbers in the series:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Using Recursion to Solve a Fibonacci Series

- The Fibonacci Series is defined by the following rules:
  - Where n is the n^{th} number in the series.
    - n = 0 fib(n) = n
    - n = 1 fib(n) = n
    - n > 1 fib(n) = fib(n-1) + fib(n-2)
  - Examples:
    - fib(2) = fib(1) + fib(0) = 1 + 0 = 1
    - fib(3) = fib(2) + fib(1) = 1 + 1 = 2
    - fib(4) = fib(3) + fib(2) = 2 + 1 = 3
    - fib(5) = fib(4) + fib(3) = 3 + 2 = 5

Using Recursion to Solve a Fibonacci Series

- The base case is n = 0 or n = 1.
  - If n = 0: fib(n) = n
  - If n = 1: fib(n) = n

- The recursive case is when n > 1.
  - If n > 1: fib(n) = fib(n-1) + fib(n-2)

Solving a Fibonacci Series

- An iterative algorithm to solve the Fibonacci Series.

```
public int fib(int n) {
    int n1 = 0;
    int n2 = 1;
    for(int i = 0; i < n; i++){
        int temp = n1;
        n1 = n2;
        n2 = temp + n2;
    }
    return n1;
}
```

Using Recursion to Solve a Fibonacci Series

- A recursive algorithm to solve the Fibonacci Series.

```java
public int fib(int n) {
    if (n == 0 || n == 1) {
        return n;
    }
    else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

Directories

- A ***directory*** (or folder) contains files and other directories (or subdirectories.)
- The File object can be a file or a directory.
- Both examples below are valid.

```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");
File myDirectory = new File("C:\\path\\to\\my");
```

Determining if a File object is a directory

- To determine if a File object is a directory, use the isDirectory method.
  - You'll only need to use this if you're not sure whether or not a File object is a directory.

```
File myDirectory = new File("C:\\path\\to\\my");
if(myDirectory.isDirectory()) {
    //Do Stuff
}
```

- The isDirectory method returns true if the File Object is a directory or false if the File object is a file.
- Alternatively, you can use the File object's isFile method, which returns true if the File object is a file or false if the File object is a directory.

Directory Permissions and Names/Paths

- All of the File object methods seen earlier in the lecture can be used for directories.
  - exists method
  - canRead method
  - canWrite method
  - getName method
  - getParent method
  - getPath method
  - getAbsolutePath method

Getting a list of files and subdirectories

- The listFiles method returns an array of File objects.
  - The File objects in the array are the files *and subdirectories* contained in the directory.

```
File myDirectory = new File("C:\\path\\to\\my");
if(myDirectory.isDirectory()) {
  File[] contents = myDirectory.listFiles();
}
```

Getting a list of files and subdirectories (names/Strings)

- The list method returns an array of Strings.
  - These Strings are the **names** of the files and subdirectories contained in the directory.

```
File myDirectory = new File("C:\\path\\to\\my");
if(myDirectory.isDirectory()) {
  String[] names = myDirectory.list();
}
```

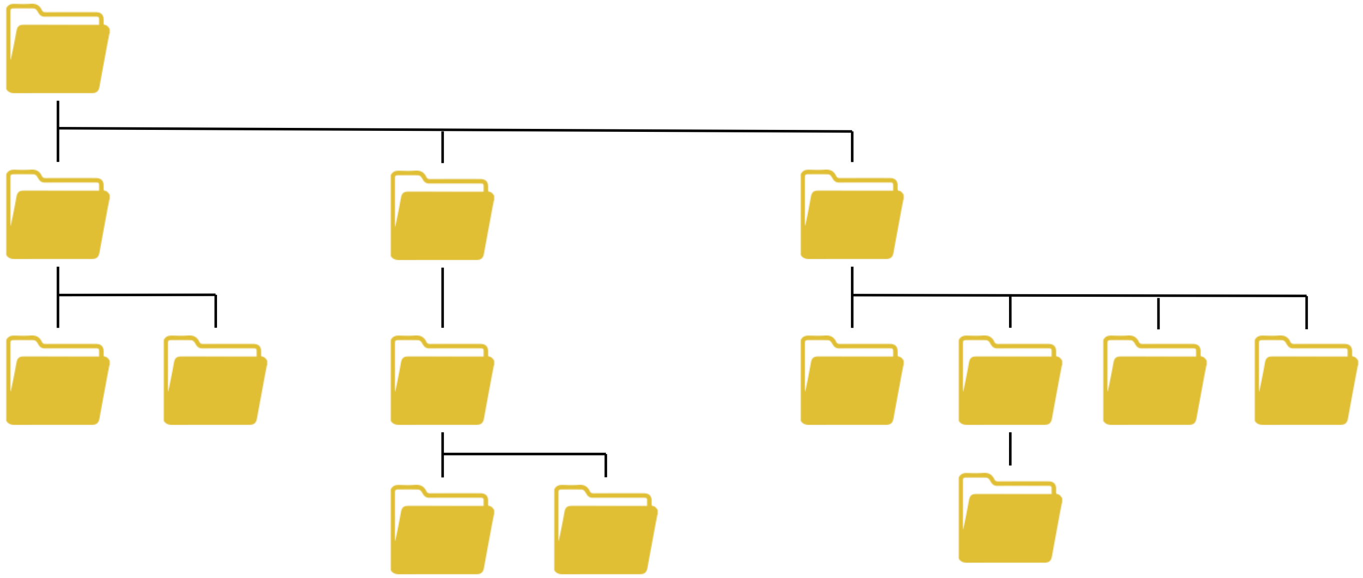
Traversing Directories

- Many applications need to traverse a directory structure when searching for (or searching through) files.

- One way that a directory structure can be traversed is by using a depth-first search (or “DFS”).
  - A breadth-first search (or “BFS”) can also be used, but requires the use of data structures (specifically, a queue) that we do not cover in this course.
  - These traversals are used in many other contexts in computing, aside from traversing directories.

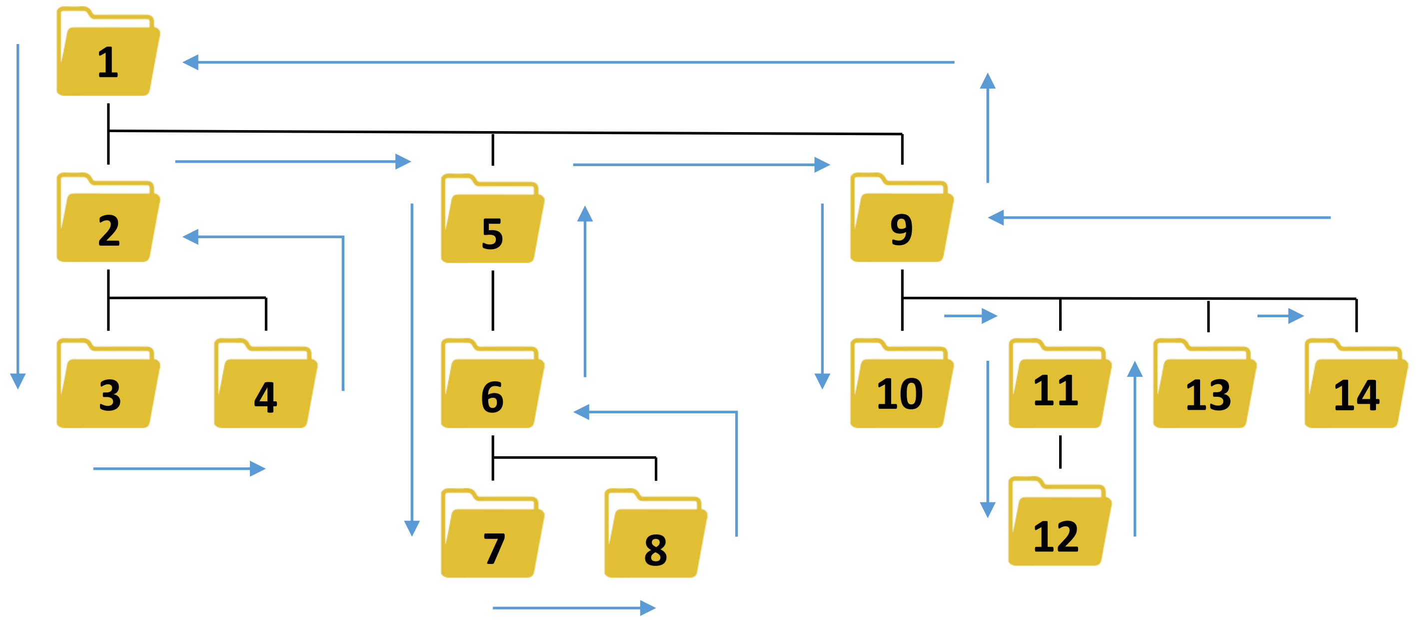
Depth-First Search

- The depth-first search will be demonstrated first.
- Imagine a program that was looking for a file named MissingFile.txt in the directory structure illustrated below:



Depth-First Search

- The arrows show the path a depth-first search would take when searching for the file.
  - The numbers show the order in which the directories will be visited.



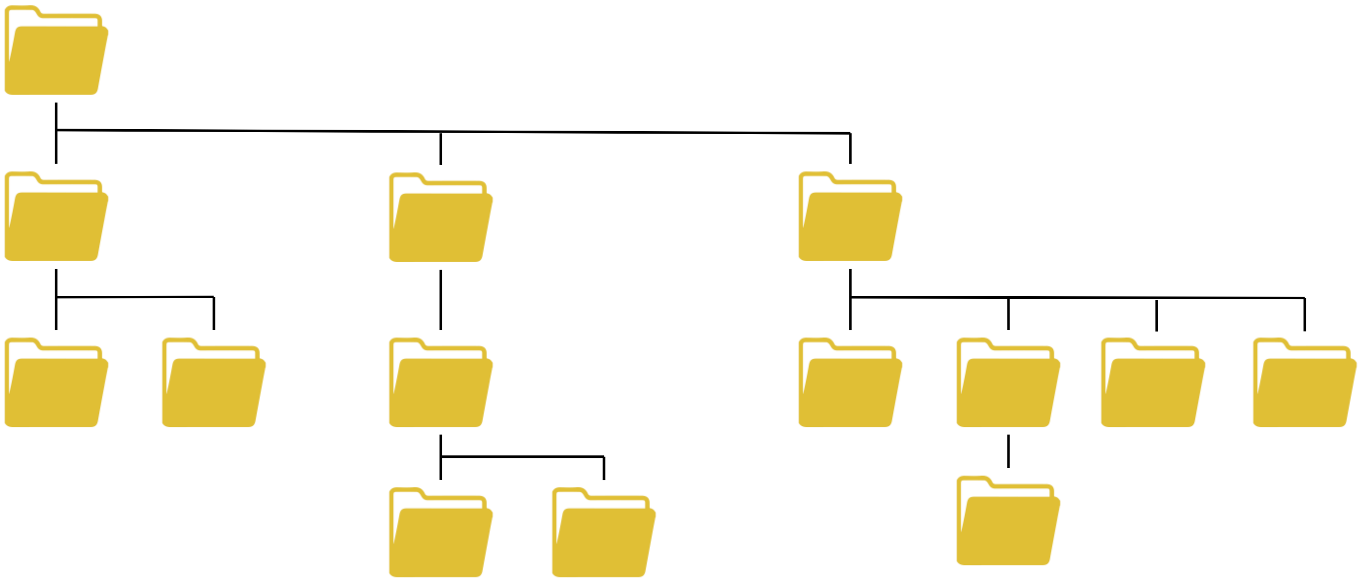
Depth-First Search

- A recursive depth-first algorithm that searches a directory structure.

```java
public File findFile(String name, File directory) {
    File[] contents = directory.listFiles();
    for(File f : contents) {
        if(f.isDirectory()) {
            findFile(name, f);
        }
    }
    File foundFile = null;
    for(File f : contents) {
        if(f.isFile() && f.getName().equals(name)) {
            return f;
        }
    }
    return foundFile;
}
```

Breadth-First Search

- We'll use the same directory structure as before to simply illustrate the process of a BFS.
  - We won't implement a BFS algorithm because it requires the use of data structures that aren't covered until CSCI 211



Breadth-First Search

- The arrows show the path a breadth-first search would take when searching for the file.
  - The numbers show the order in which the directories will be visited.