"To a man with a hammer, everything looks like a nail"
-Mark Twain
Underneath the Hood.
The Program Stack

When you invoke a method in your code what happens when that method is completed?

```java
public static void start() {
    int x = 5;
    int y = -5;
    Point pt = new Point(x, y);
    pt.scale(2);
    String s = pt.toString();
}
```
public class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public void scale(int v) {
        x *= v;
        y *= v;
    }

    public int getX() {return x;}
    public int getY() {return y;}

    public String toString() {
        return "x: " + getX() + ", y: " + getY();
    }
}

CS314 Recursion
The Program Stack

- When your program is executed on a processor the commands are converted into another set of instructions and assigned memory locations.
  - normally a great deal of expansion takes place

```java
public static void start() {
    int x = 5; // 50
    int y = -5; // 51
    Point pt = new Point(x, y); // 52
    pt.scale(2); // 53
    String s = pt.toString(); // 54
}
```
A CPU works via a fetch command / execute command loop and a program counter.

Instructions stored in memory (Instructions are data!)

```java
50    int x = 5;
51    int y = -5;
52    Point pt = new Point(x, y);
53    pt.scale(2);
54    String s = pt.toString();
```

What if the first instruction of the scale method is stored at memory location 103?
More on the Program Stack

50 int x = 5;
51 int y = -5;
52 Point pt = new Point(x, y);
53 pt.scale(2);
54 String s = pt.toString();

- Instruction 53 is really saying *jump to instruction 103 with pt as the implicit parameter and 2 as the explicit parameter*

- When scale is done what happens?
  A. Program ends  
  B. goes to instruction 54  
  C. Goes back to whatever method called it
Activation Records and the Program Stack

- When a method is invoked all the relevant information about the current method (variables, values of variables, next line of code to be executed) is placed in an activation record.

- The activation record is pushed onto the program stack.

- A stack is a data structure with a single access point, the top.
The Program Stack

- Data may either be added (*pushed*) or removed (*popped*) from a stack but it is always from the top.
  - A stack of dishes
  - which dish do we have easy access to?
Using Recursion
A Problem

- Write a method that determines how much space is take up by the files in a directory
- A directory can contain files and directories
- How many directories does our code have to examine?
- How would you add up the space taken up by the files in a single directory
  - Hint: don't worry about any sub directories at first
Attendance Question 2

- How many levels of directories have to be visited?
  A. 0
  B. Unknown
  C. Infinite
  D. 1
  E. 8
Sample Directory Structure

- scottm
  - cs307
    - m1.txt
    - m2.txt
  - hw
    - a1.htm
    - a2.htm
    - a3.htm
    - a4.htm
  - AP
    - A.pdf
    - AB.pdf
Java File Class

- **File**(String pathname) Creates a new File instance by converting the given pathname.
- boolean **isDirectory()** Tests whether the file denoted by this abstract pathname is a directory.
- **File[] listFiles()** Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.
Code for `getDirectorySpace()`

```java
// assert dir is a directory and dir != null
public static long spaceUsed(File dir) {
    assert dir != null && dir.isDirectory();
    long spaceUsed = 0;
    File[] subFilesAndDirs = dir.listFiles();
    if(subFilesAndDirs != null)
        for(File sub : subFilesAndDirs)
            if(sub != null)
                if(!sub.isDirectory()) // sub is a plain old file
                    spaceUsed += sub.length();
                else // else sub is a directory
                    spaceUsed += spaceUsed(sub);
    return spaceUsed;
}
```
Attendance Question 3

- Is it possible to write a non recursive method to do this?

A. Yes
B. No
Iterative `getDirectorySpace()`

```java
public int getDirectorySpace(Directory d) {
    ArrayList dirs = new ArrayList();
    File[] fileList;
    Directory[] dirList;
    dirs.add(d);
    Directory temp;
    int total = 0;
    while( ! dirs.isEmpty() ) {
        temp = (Directory)dirs.remove(0);
        fileList = temp.getFiles();
        for(int i = 0; i < fileList.length; i++)
            total += fileList[i].getSize();
        dirList = temp.getSubdirectories();
        for(int i =0; i < dirList.length; i++)
            dirs.add( dirList[i] );
    }
    return total;
}
```
Wisdom for Writing Recursive Methods
The 3 plus 1 rules of Recursion

1. Know when to stop
2. Decide how to take one step
3. Break the journey down into that step and a smaller journey
4. Have faith

From *Common Lisp: A Gentle Introduction to Symbolic Computation* by David Touretzky
Writing Recursive Methods

- Rules of Recursion
  1. Base Case: Always have at least one case that can be solved without using recursion
  2. Make Progress: Any recursive call must progress toward a base case.
  3. "You gotta believe." Always assume that the recursive call works. (Of course you will have to design it and test it to see if it works or prove that it always works.)

A recursive solution solves a small part of the problem and leaves the rest of the problem in the same form as the original
the classic first recursion problem / example

5! = 5 * 4 * 3 * 2 * 1 = 120

```c
int res = 1;
for(int i = 2; i <= n; i++)
    res *= i;
```
Factorial Recursively

- Mathematical Definition of Factorial
  \[ 0! = 1 \]
  \[ N! = N \times (N - 1)! \]

  The definition is recursive.

  ```java
  // pre n >= 0
  public int fact(int n) {
    if (n == 0)
      return 1;
    else
      return n * fact(n - 1);
  }
  ```
Tracing Fact With the Program Stack

System.out.println( fact(4) );
Calling fact with 4

System.out.println( fact(4) );

n \[4\] in method fact

partial result = n \times fact(n-1)
Calling fact with 3

\[
\begin{array}{|c|}
\hline
n & 3 \\
\hline
\end{array}
\]

in method fact

partial result = n * fact(n-1)

\[
\begin{array}{|c|}
\hline
n & 4 \\
\hline
\end{array}
\]

in method fact

partial result = n * fact(n-1)

System.out.println( fact(4) );
### Calling fact with 2

<table>
<thead>
<tr>
<th>n</th>
<th>2</th>
<th>in method fact</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>partial result = n * fact(n-1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>3</th>
<th>in method fact</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>partial result = n * fact(n-1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>4</th>
<th>in method fact</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>partial result = n * fact(n-1)</td>
</tr>
</tbody>
</table>

System.out.println( fact(4) );
### Calling fact with 1

<table>
<thead>
<tr>
<th>n</th>
<th>1</th>
<th>in method fact</th>
<th>partial result = n * fact(n-1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>2</td>
<td>in method fact</td>
<td>partial result = n * fact(n-1)</td>
</tr>
<tr>
<td>n</td>
<td>3</td>
<td>in method fact</td>
<td>partial result = n * fact(n-1)</td>
</tr>
<tr>
<td>n</td>
<td>4</td>
<td>in method fact</td>
<td>partial result = n * fact(n-1)</td>
</tr>
</tbody>
</table>

System.out.println( fact(4) );
Calling fact with 0 and returning 1

<table>
<thead>
<tr>
<th>n</th>
<th>0</th>
<th>in method fact</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>returning 1 to whatever method called me</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>1</th>
<th>in method fact</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>partial result = n * fact(n-1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>2</th>
<th>in method fact</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>partial result = n * fact(n-1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>3</th>
<th>in method fact</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>partial result = n * fact(n-1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>4</th>
<th>in method fact</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>partial result = n * fact(n-1)</td>
</tr>
</tbody>
</table>

System.out.println( fact(4) );
Returning 1 from fact(1)

```
System.out.println( fact(4) );
```

```
partial result = n * 1,
return 1 to whatever method called me
```

```
partial result = n * fact(n-1)
```

```
partial result = n * fact(n-1)
```

```
partial result = n * fact(n-1)
```

```
partial result = n * fact(n-1)
```

```
partial result = n * fact(n-1)
```

```
System.out.println( fact(4) );
```
Returning 2 from fact(2)

- n = 2 in method fact
  
  partial result = 2 * 1,
  return 2 to whatever method called me

- n = 3 in method fact
  
  partial result = n * fact(n-1)

- n = 4 in method fact
  
  partial result = n * fact(n-1)

System.out.println( fact(4) );
Returning 6 from fact(3)

\[
\begin{align*}
\text{partial result} &= 3 \times 2, \\
\text{return 6 to whatever method called me}
\end{align*}
\]
Returning 24 from fact(4)

\[ n = 4 \] in method fact

partial result = 4 * 6,
return 24 to whatever method called me

```
System.out.println( fact(4) );
```
Calling System.out.println

System.out.println(24);

??
top
Evaluating Recursive Methods
Evaluating Recursive Methods

- you must be able to evaluate recursive methods

```java
public static int mystery (int n){
    if( n == 0 )
        return 1;
    else
        return 3 * mystery(n-1);
}
```

// what is returned by mystery(5)
Evaluating Recursive Methods

- Draw the program stack!

\[ m(5) = 3 \times m(4) \]
\[ m(4) = 3 \times m(3) \]
\[ m(3) = 3 \times m(2) \]
\[ m(2) = 3 \times m(1) \]
\[ m(1) = 3 \times m(0) \]
\[ m(0) = 1 \]

\[ \rightarrow 3^5 = 243 \]

- with practice you can see the result
What is returned by \texttt{mystery(-3)}?

A. 0
B. 1
C. Infinite loop
D. Syntax error
E. Runtime error due to stack overflow
Evaluating Recursive Methods

- What about multiple recursive calls?

```java
public static int bar(int n) {
    if (n <= 0)
        return 2;
    else
        return 3 + bar(n-1) + bar(n-2);
}
```

- Draw the program stack and REMEMBER your work
Evaluating Recursive Methods

- What is returned by \( \text{bar}(5) \)?

\[
\begin{align*}
\text{b}(5) &= 3 + \text{b}(4) + \text{b}(3) \\
\text{b}(4) &= 3 + \text{b}(3) + \text{b}(2) \\
\text{b}(3) &= 3 + \text{b}(2) + \text{b}(1) \\
\text{b}(2) &= 3 + \text{b}(1) + \text{b}(0) \\
\text{b}(1) &= 3 + \text{b}(0) + \text{b}(-1) \\
\text{b}(0) &= 2 \\
\text{b}(-1) &= 2
\end{align*}
\]
Evaluating Recursive Methods

- What is returned by \( \text{bar}(5) \)?

\[
\begin{align*}
\text{b}(5) &= 3 + \text{b}(4) + \text{b}(3) \\
\text{b}(4) &= 3 + \text{b}(3) + \text{b}(2) \\
\text{b}(3) &= 3 + \text{b}(2) + \text{b}(1) \\
\text{b}(2) &= 3 + \text{b}(1) + \text{b}(0) \quad //\text{substitute in results} \\
\text{b}(1) &= 3 + 2 + 2 = 7 \\
\text{b}(0) &= 2 \\
\text{b}(-1) &= 2
\end{align*}
\]
Evaluating Recursive Methods

- What is returned by \( \text{bar}(5) \) ?

\[
\begin{align*}
\text{b}(5) &= 3 + \text{b}(4) + \text{b}(3) \\
\text{b}(4) &= 3 + \text{b}(3) + \text{b}(2) \\
\text{b}(3) &= 3 + \text{b}(2) + \text{b}(1) \\
\text{b}(2) &= 3 + 7 + 2 = 12 \\
\text{b}(1) &= 7 \\
\text{b}(0) &= 2 \\
\text{b}(-1) &= 2
\end{align*}
\]
Evaluating Recursive Methods

- What is returned by $\text{bar}(5)$?

\[ b(5) = 3 + b(4) + b(3) \]
\[ b(4) = 3 + b(3) + b(2) \]
\[ b(3) = 3 + 12 + 7 = 22 \]
\[ b(2) = 12 \]
\[ b(1) = 7 \]
\[ b(0) = 2 \]
\[ b(-1) = 2 \]
Evaluating Recursive Methods

- What is returned by \texttt{bar}(5) ?

\[
b(5) = 3 + b(4) + b(3)
\]

\[
b(4) = 3 + 22 + 12 = 37
\]

\[
b(3) = 22
\]

\[
b(2) = 12
\]

\[
b(1) = 7
\]

\[
b(0) = 2
\]

\[
b(-1) = 2
\]
Evaluating Recursive Methods

What is returned by $\text{bar}(5)$?

\[
\begin{align*}
\text{b}(5) &= 3 + 37 + 22 = 62 \\
\text{b}(4) &= 37 \\
\text{b}(3) &= 22 \\
\text{b}(2) &= 12 \\
\text{b}(1) &= 7 \\
\text{b}(0) &= 2 \\
\text{b}(-1) &= 2
\end{align*}
\]
Recursion Practice

- Write a method `raiseToPower(int base, int power)`
- //pre: power >= 0

- Simple recursion (also called tail recursion)
Finding the Maximum in an Array

- public int max(int[] values) {
- Helper method or create smaller arrays each time
Attendance Question 5

- When writing recursive methods what should be done first?

A. Determine recursive case
B. Determine recursive step
C. Make recursive call
D. Determine base case(s)
E. Determine Big O
Your Meta Cognitive State

- Remember we are learning to use a tool.
- It is not a good tool for *all* problems.
  - In fact we will implement several algorithms and methods where an iterative (looping without recursion) solution would work just fine.
- After learning the mechanics and basics of recursion the real skill is knowing what problems or class of problems to apply it to.
Determining the Big O of recursive methods can be tricky.

A *recurrence relation* exits if the function is defined recursively.

The $T(N)$, actual running time, for $N!$ is recursive

$$T(N)_{\text{fact}} = T(N-1)_{\text{fact}} + O(1)$$

This turns out to be $O(N)$

– There are $N$ steps involved
Common Recurrence Relations

- $T(N) = T(N/2) + O(1) \implies O(\log N)$
  - binary search
- $T(N) = T(N-1) + O(1) \implies O(N)$
  - sequential search, factorial
- $T(N) = T(N/2) + T(N/2) + O(1) \implies O(N)$,
  - tree traversal
- $T(N) = T(N-1) + O(N) \implies O(N^2)$
  - selection sort
- $T(N) = T(N/2) + T(N/2) + O(N) \implies O(N \log N)$
  - merge sort
- $T(N) = T(N-1) + T(N-1) + O(1) \implies O(2^N)$
  - Fibonacci