Week 6 - Friday

# COMP 2000

# Last time

- What did we talk about last time?
- Calculator example
- Audio

# Questions?

# Project 2

# Menus

# Menus

- In addition to widgets scattered across the surface of a **`JFrame`**, users are accustomed to **menus**
- Menus, of course, drop down to display a list of options
- The good news is that we can add action listeners to these menu items just like we can with **`JButton`** objects
- All we have to do is learn how to create menus

# JMenu

- In Swing, the menu itself (not the choices in the menu) is a **JMenu**
- You can create a **JMenu** with the name of your choice much like any other Swing widget with text

```
JMenu fileMenu = new JMenu("File");
```

- Like other widgets, creating the menu doesn't display it
- We'll have to add it to the appropriate container
- Common menus are:
  - File
  - Edit
  - View
  - Help

# JMenuItem

- The choices that you add to a **JMenu** are objects of type **JMenuItem**
- They function much like a **JButton** in that you can add an action listener to them

```
JMenuItem exitItem = new JMenuItem("Exit");
exitItem.addActionListener(e ->
frame.dispose());
```

- Once you create a **JMenuItem**, you can add it to a **JMenu**

```
fileMenu.add(exitItem);
```

- You can even add a **JMenu** to another **JMenu** for nested menus

# JMenuBar

- Where do the **JMenu** objects live?
- A **JMenuBar**, of course
- First, you create a **JMenuBar**
- Then, you add your **JMenu** objects to it (in order)
- Then, you set it as the frame's menu bar

```
JMenuBar menuBar = new JMenuBar();
menuBar.add(fileMenu);  // add one or more menus
frame.setJMenuBar(menuBar);
```

- Make sure you don't *add* the menu bar to the frame by mistake

# Extras

- We're just scratching the surface
- It's possible to add accelerators (keyboard shortcuts) to menu items
- There's an **`addSeparator()`** method on the **`JMenu`** object that can create separators between groups of menu items
- Menu items can have icons
- You can even put check boxes and radio buttons in menus
- If you're interested, read a tutorial or the API

# Recursion

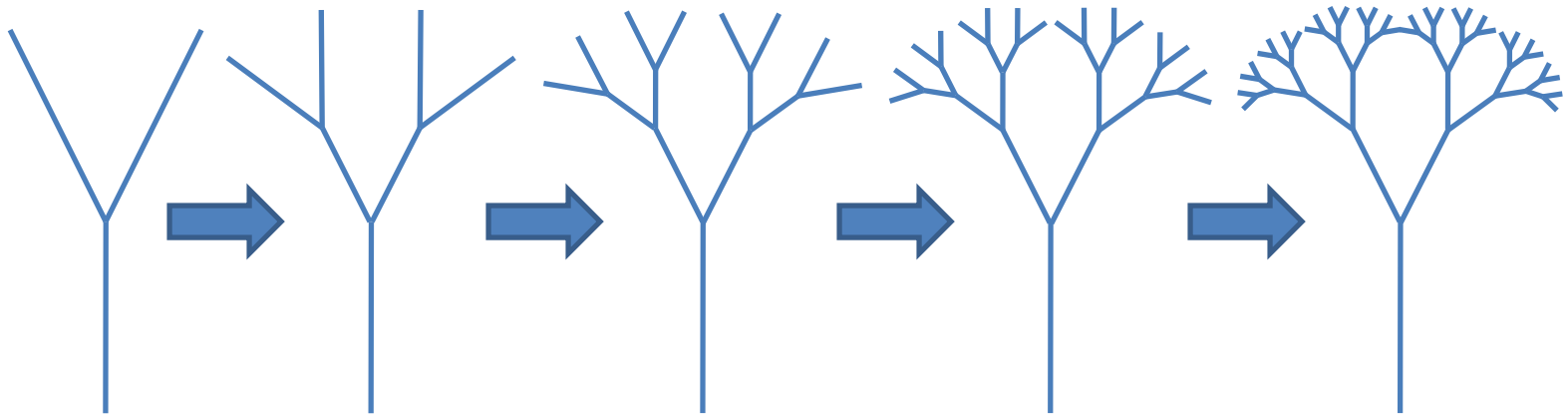
To understand recursion, you must first understand recursion.

# What is recursion?

- Defining something in terms of itself
- To be useful, the definition must be based on progressively simpler definitions of the thing being defined

# Bottom Up

- It is possible to define something recursively from the bottom up
- We start with a simple pattern and repeat the pattern, using a copy of the pattern for each part of the starting pattern



# Top Down

Explicitly:

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

Recursively:

- $n! = (n)(n-1)!$
- $1! = 1$

- $6! = 6 \cdot 5!$
  - $5! = 5 \cdot 4!$
    - $4! = 4 \cdot 3!$
      - $3! = 3 \cdot 2!$
        - $2! = 2 \cdot 1!$
          - $1! = 1$
- $6! = 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 720$

# Examples in Acronyms

- PHP
  - PHP: Hypertext Processor
    - (PHP: Hypertext Processor): Hypertext Processor
      - …
- XINU
  - XINU Is Not Unix
    - (XINU Is Not Unix) Is Not Unix
      - …

# Useful Recursion

Two parts:

- Base case(s)
  - Tells recursion when to stop
  - For factorial, $n = 1$ or $n = 0$ are examples of base cases
- Recursive case(s)
  - Allows recursion to progress
  - "Leap of faith"
  - For factorial, $n > 1$ is the recursive case

# Solving Problems with Recursion

# Approach for Problems

- Top down approach
- Don't try to solve the whole problem
- Deal with the next step in the problem
- Then make the "leap of faith"
- Assume that you can solve any smaller part of the problem

# Walking to the Door

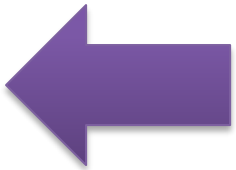
- Problem: You want to walk to the door
- Base case (if you reach the door):
  - You're done!
- Recursive case (if you aren't there yet):
  - Take a step toward the door

# Implementing Factorial

- Base case ($n \leq 1$):
  - $1! = 0! = 1$
- Recursive case ($n > 1$):
  - $n! = n(n-1)!$

# Code for Factorial

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



Base Case

Recursive Case

# Recursive style

- When we do recursion, we want to pass all the data in through our method arguments
- We want to get all of our results back through return statements
- Think of each recursive method call as a frozen moment in time
- Thus, we usually **don't** want to assign variables
- Instead, variables change *as they pass* to the next method call

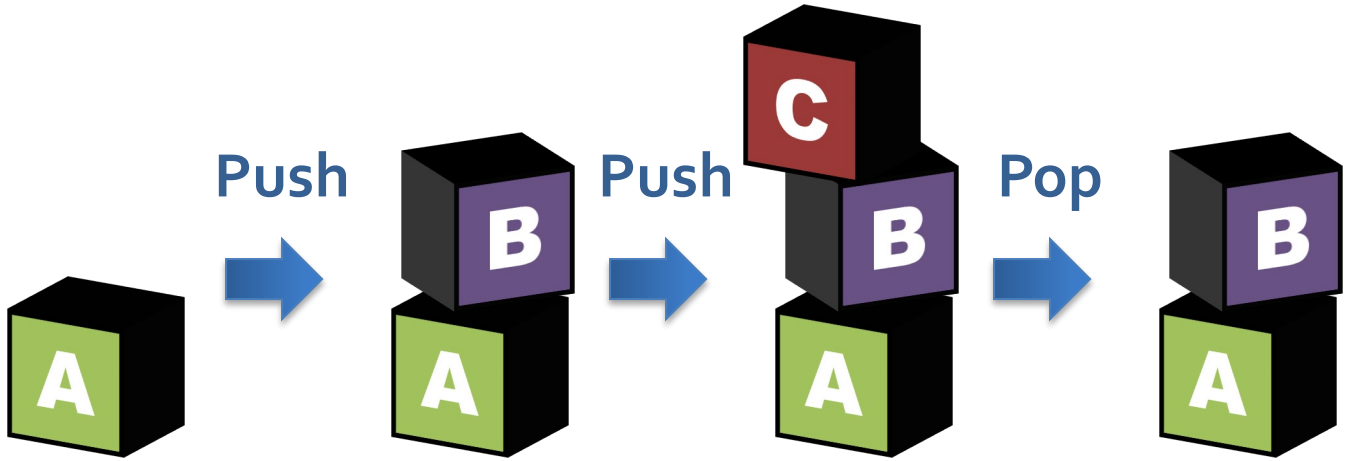
# How Does Recursion Work Inside The Computer?

# All this math is great, but…

- How does it actually work inside a computer?
- Is there a problem with calling a method inside the same method?
- How does the computer keep track of which method is which?

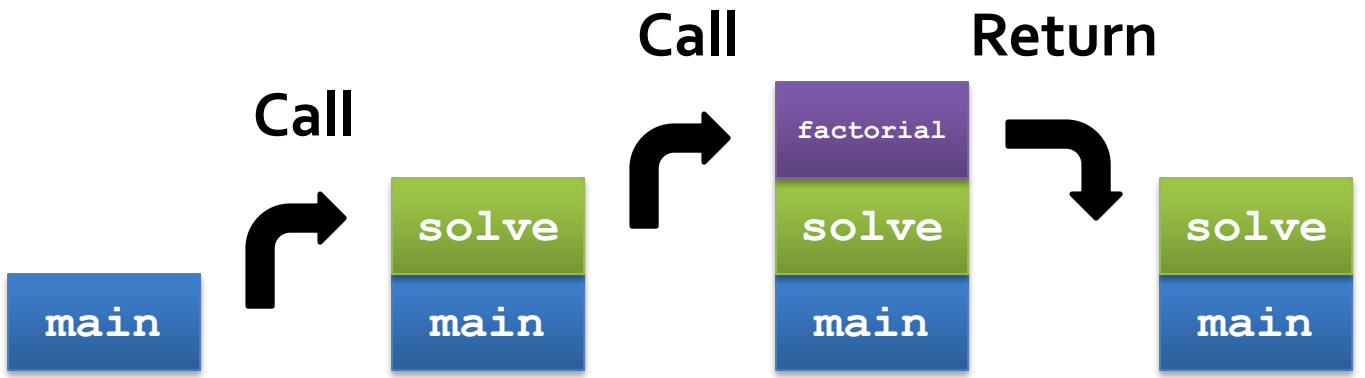
# Stacks

- A **stack** is a first-in last-out (FILO) data structure used to store and retrieve items in a particular order
- Just like a stack of blocks:



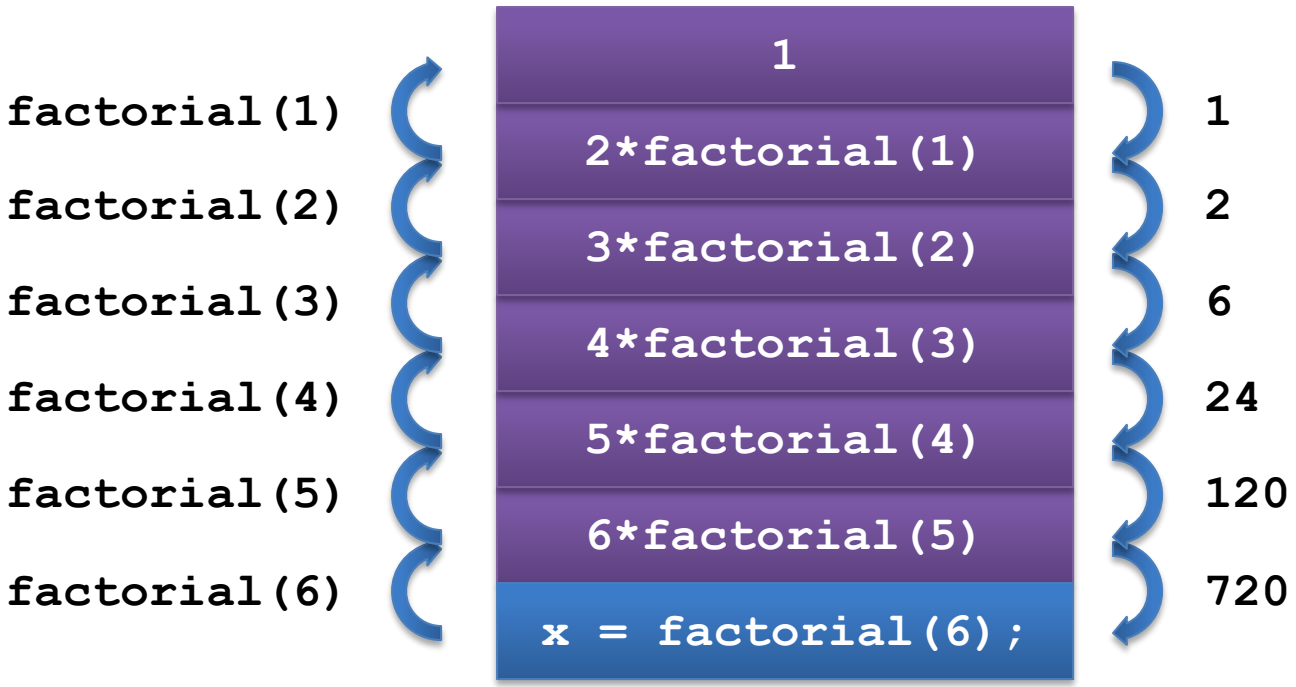
# Call stack

- In the same way, the local variables for each method are stored on the stack
- When a method is called, a copy of that method is **pushed** onto the stack
- When a method returns, that copy of the method **pops** off the stack



# Example with Factorial

- Each copy of factorial has a value of ***n*** stored as a local variable
- For 6! :



# More Examples

# Recursion and loops have the same power

- You can use recursion anytime you would use loops
- With one exception:
  - If you make too many method calls, you will run out of stack space, and your program will crash
  - On these machines, it's probably between 10,000 and 30,000 method calls
- Some languages like Lisp don't even have loops!
- Everything is done recursively
- Some problems are naturally modeled recursively

# Multiplication

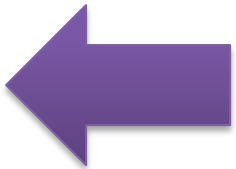
- Although it's not efficient to do so, we can think of multiplication as repeated addition
- Thus, $x \cdot y = x + x + x \ldots + x$ ($y$ times)
- Base case (***y*** = 0):
  - $x \cdot 0 = 0$
- Recursive case (***y*** > 0):
  - $x \cdot y = x + \big(x \cdot (y - 1)\big)$

# Code for multiplication

```
public static int multiply( int x, int y ){

  if( y == 0 )
     return 0;
  else
     return x + multiply( x, y - 1 );

}
```



Base Case

Recursive Case

# Exponentiation

- Similarly, exponentiation is repeated multiplication
- Thus, $x^y = x \cdot x \cdot x \ldots \cdot x$ ($y$ times)
- Base case (***y*** = 0):
  - $x^0 = 1$
- Recursive case (***y*** > 0):
  - $x^y = x \cdot x^{y-1}$

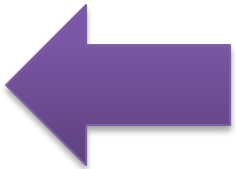
- There is a more efficient way to do this, but you'll have to take COMP 2100 to talk about it

# Code for exponentiation

```
public static double power( double x, int y ){

  if( y == 0 )
     return 1.0;
  else
     return x * power( x, y - 1 );

}
```



Base Case

Recursive Case

# Upcoming

# Next time…

- More recursion

# Reminders

- Keep reading Chapter 19
- Keep working on Project 2
- **InSocial Risk Advisors are looking for a consultant**
  - They need help linking together some services with Zapier
  - Should be a small amount of work, but it might open up other opportunities
  - If interested, send a resume to Jim Waterwash
  - Get his contact information from me