

Week 10 - Friday

COMP 2000

Last time

- What did we talk about last time?
- Client-server model
- Socket communication

Questions?

Project 3

Review

JOptionPane

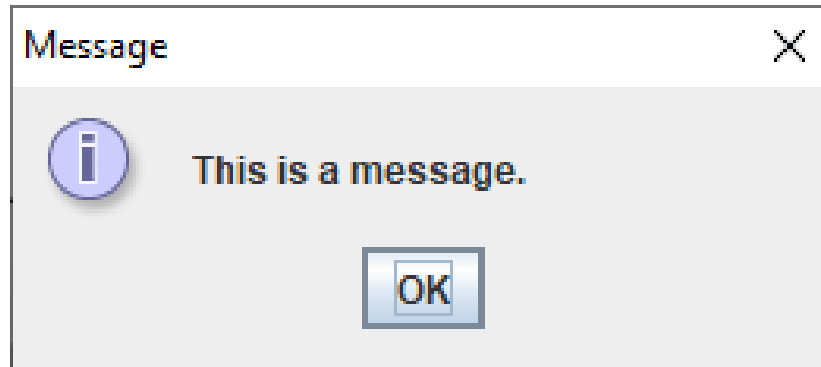
JOptionPane

- **JOptionPane** class provides static methods for:
 - Displaying a message
 - Asking a question
- Although it is possible to create a **JOptionPane** object, you almost never do
- Just call the static methods
 - Which means typing a lot of **JOptionPane**.

showMessageDialog() example

- To display "This is a message." you could call the following:

```
JOptionPane.showMessageDialog(null,  
    "This is a message.");
```



Adding a title

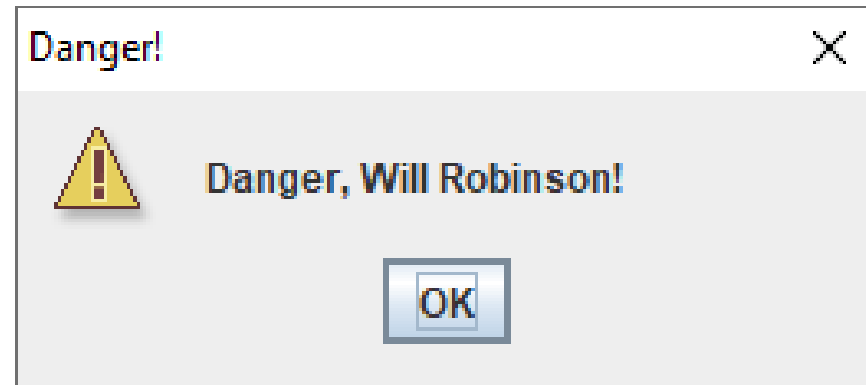
- Most **JOptionPane** methods have many overloads
- If you want to put a title on the window, you can pass it in as the third parameter
- But this overloaded method also requires an **int** parameter that says what kind of message you want
- To add the title "**Window Title**", you might call the following method:



```
JOptionPane.showMessageDialog(null,  
    "This is a message.", "Window Title",  
    JOptionPane.PLAIN_MESSAGE);
```

Different icons

- You can choose an icon associated with one of the following constants:
 - `ERROR_MESSAGE`
 - `INFORMATION_MESSAGE`
 - `WARNING_MESSAGE`
 - `QUESTION_MESSAGE`
 - `PLAIN_MESSAGE`

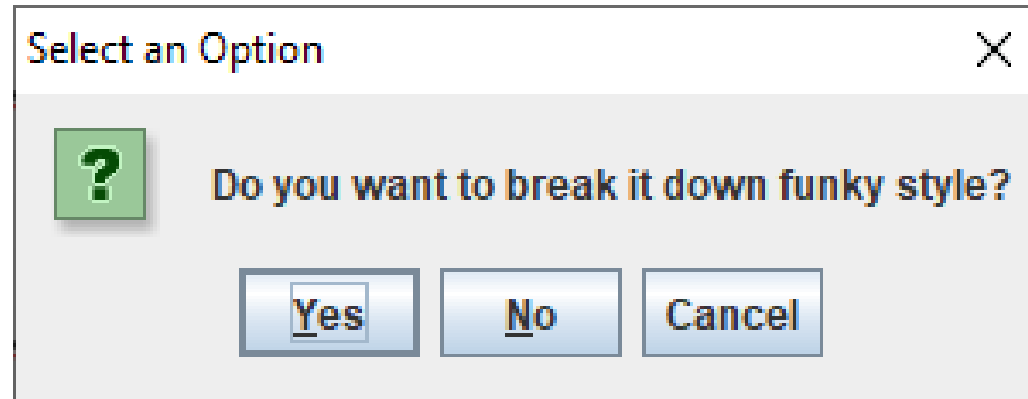


```
JOptionPane.showMessageDialog(null,  
    "Danger, Will Robinson!", "Danger!",  
    JOptionPane.WARNING_MESSAGE);
```

showConfirmDialog() example

```
int answer = JOptionPane.showConfirmDialog(null,  
    "Do you want to break it down funky style?");  
if(answer == JOptionPane.YES_OPTION)  
    JOptionPane.showMessageDialog(null, "Dope!");  
else  
    JOptionPane.showMessageDialog(null, "Weak!");
```

- Hitting the X in the corner is the same as Cancel

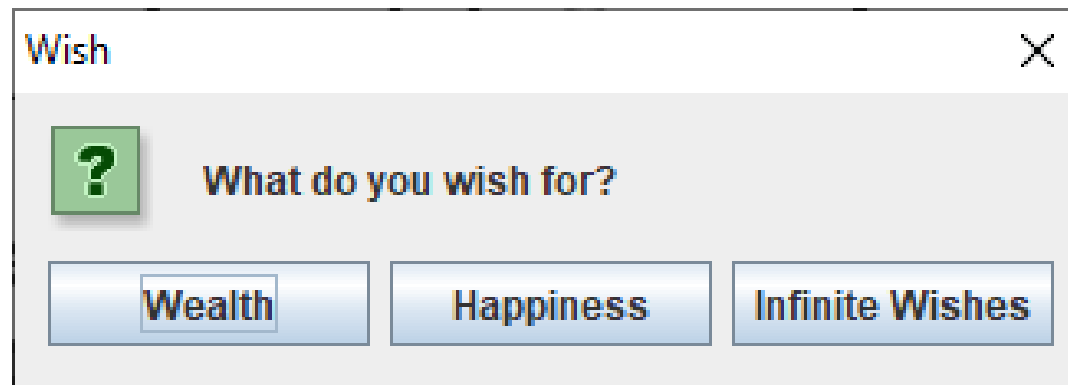


showOptionDialog() example

- Here's an example showing a dialog that allows a user to choose between Wealth, Happiness, and Infinite Wishes

```
String[] options = {"Wealth", "Happiness", "Infinite Wishes"};  
int answer = JOptionPane.showOptionDialog(null,  
    "What do you wish for?", "Wish", JOptionPane.DEFAULT_OPTION,  
    JOptionPane.QUESTION_MESSAGE, null, options, null);
```

- Note that many parameters can be **null**: parent, icon, default option

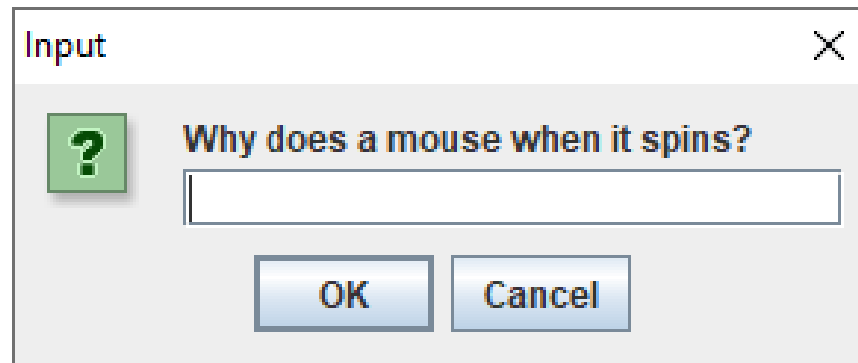


showInputDialog() example

- This input dialog asks a pressing question

```
String answer =  
    JOptionPane.showInputDialog(null,  
        "Why does a mouse when it spins?");
```

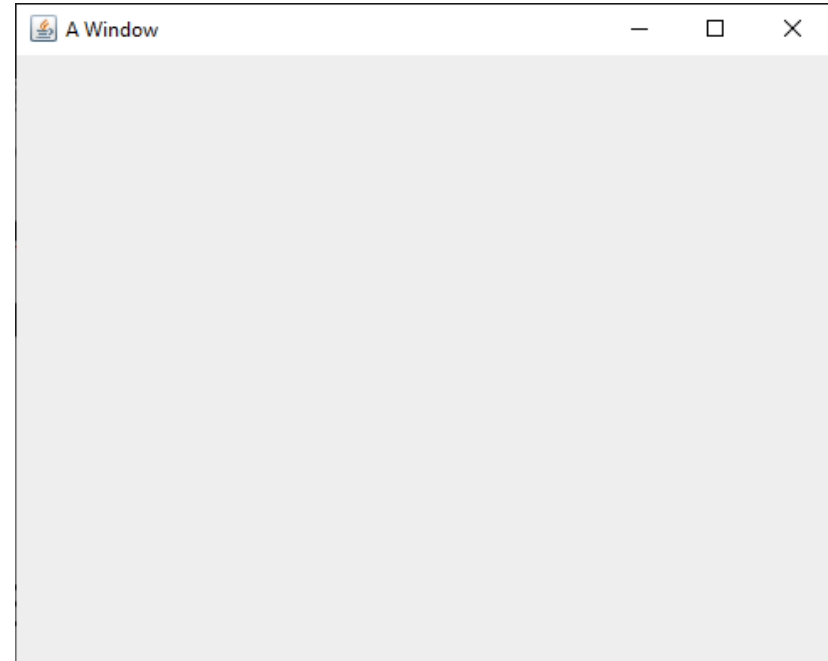
- As with other methods, there are overloaded versions that allow for titles, icons, and other options



JFrame

Creating a JFrame

- To create a **JFrame**, we will usually call its constructor that takes a **String**, giving it a title
- Then, we have to make it visible so that we can see it



```
JFrame frame = new JFrame("A Window");  
frame.setVisible(true);
```

setDefaultCloseOperation()

- Next, you'll notice that closing the window doesn't end the program
 - The little red square on the Eclipse Console is still clickable, meaning that the program is running
- By default, closing the window by clicking its X only hides the window
- By calling the **setDefaultCloseOperation()**, we can make it so that the default operation is dispose (getting rid of the window)

```
JFrame frame = new JFrame("A Window");  
frame.setSize(500, 400);  
frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);  
frame.setVisible(true);
```

- Many books suggest passing in **JFrame.EXIT_ON_CLOSE**, but you **should not!**
- Doing so will kill the rest of your program like **System.exit()**

Recap

- To use a **JFrame** you must:
 - Create a **JFrame** object
 - Set its size (either directly or by putting widgets on it and then calling **pack()**)
 - Set its default close operation to dispose
 - Make it visible
- Now that we've got a window, we can put widgets on it!

Widgets

- **Widget** is a generic term for a wide range of GUI controls
 - Buttons
 - Labels (allowing us to put text or images on a GUI)
 - Text fields
 - Text areas (like text fields but larger)
 - Menus
 - Checkboxes
 - Radio buttons
 - Lists
 - Combo boxes
 - Sliders

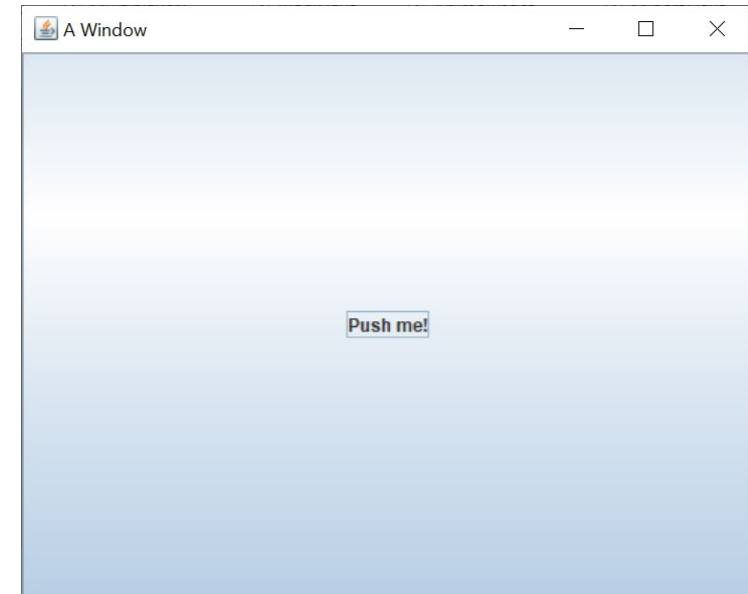
JButton

- A button you can click on is provided by the **JButton** class
- A **JButton** is usually created with text or an image
 - You'll need to make **JButtons** with images for Project 2
- Just creating the **JButton** doesn't do anything
- You have to add it to a **JFrame** (or other container) to see it
- Right now, we're just creating the buttons
- Next week, we'll learn how to add actions to them

```
JButton button = new JButton("Push me!");
```

Adding a JButton to a JFrame

- Once you've created a **JButton**, you can add it to a **JFrame** by calling the **add()** method on the **JFrame**
- All GUI containers have an **add()** method that allows us to add a widget to it



```
JFrame frame = new JFrame("A Window");
frame.setSize(500, 400);
frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
JButton button = new JButton("Push me!");
frame.add(button);
frame.setVisible(true);
```

Displaying an icon on a JButton

- You can also make a **JButton** with an image instead of text
- To do so, you create an **ImageIcon** and pass that to the constructor of the **JButton**
- You'll need the path to an image



```
JButton bowieButton = new JButton(new ImageIcon("bowie.jpg"));  
frame.add(bowieButton, BorderLayout.CENTER);
```

JLabel

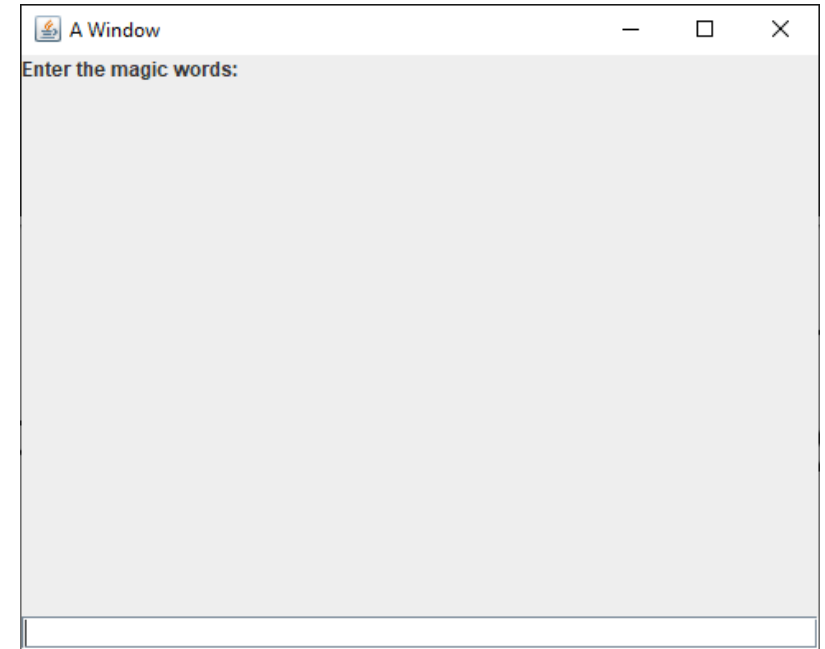
- A **JLabel** is like a button you can't click
- Its constructors work just like the **JButton** ones
- It allows you to display text or an image



```
JLabel nameLabel = new JLabel("David Bowie");  
JLabel bowieLabel = new JLabel(new ImageIcon("bowie.jpg"));  
frame.add(nameLabel, BorderLayout.NORTH);  
frame.add(bowieLabel, BorderLayout.CENTER);
```

JTextField

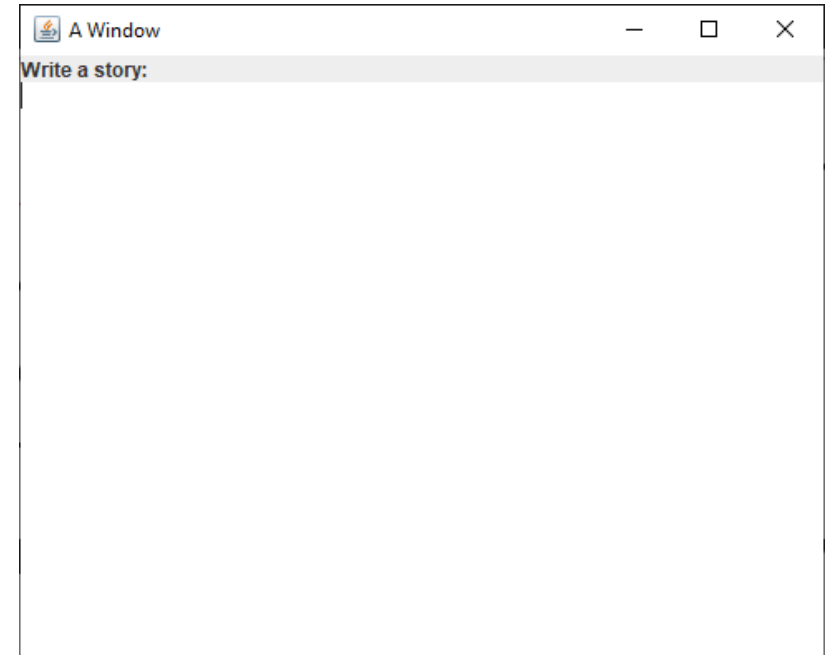
- A **JTextField** allows a user to enter a (short) amount of text
- Usually, you'll need a **JLabel** to tell the person what they should enter
- The example is ugly because the **JLabel** and the **JTextField** don't fill the 500 x 400 **JFrame**



```
JLabel messageLabel = new JLabel("Enter the magic words:");  
JTextField magicField = new JTextField();  
frame.add(messageLabel, BorderLayout.NORTH);  
frame.add(magicField, BorderLayout.SOUTH);
```

JTextArea

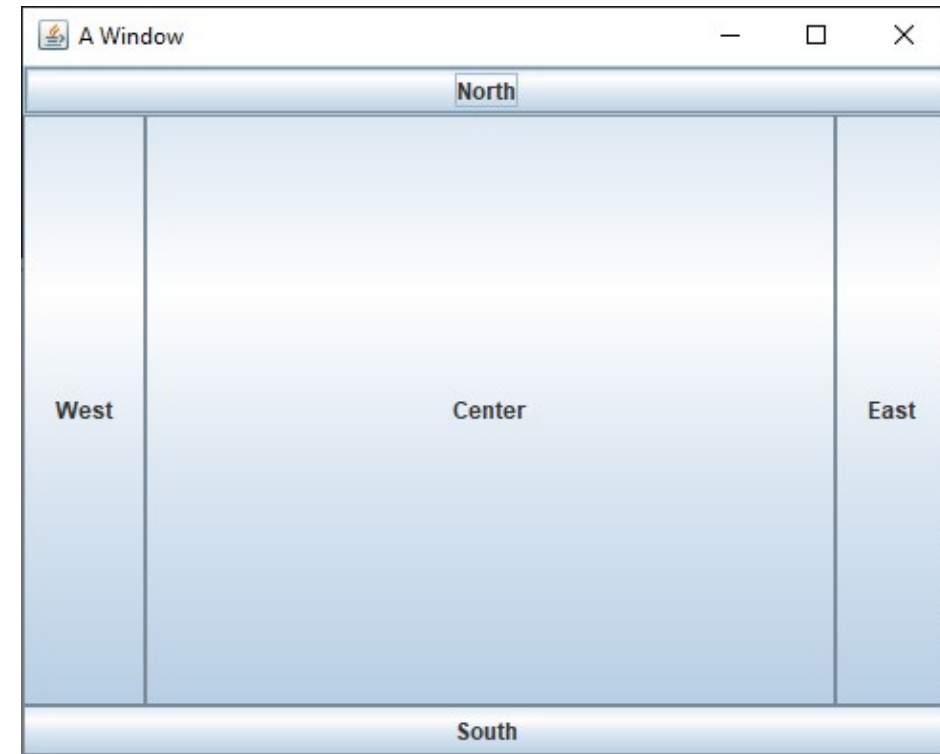
- A **JTextField** is for entering small pieces of information
 - Name
 - Address
 - Telephone number
- For larger texts, we can use a **JTextArea**



```
JLabel storyLabel = new JLabel("Write a story:");
JTextArea storyArea = new JTextArea();
frame.add(storyLabel, BorderLayout.NORTH);
frame.add(storyArea, BorderLayout.CENTER);
```


BorderLayout

- **BorderLayout** is the default layout for **JFrame**
- When you add widgets, you can specify the location as one of five regions:
 - **BorderLayout.NORTH** stretches the width of the container on the top
 - **BorderLayout.SOUTH** stretches the width of the container on the bottom
 - **BorderLayout.EAST** sits on the right of the container, stretching to fill all the space between **NORTH** and **SOUTH**
 - **BorderLayout.WEST** sits on the left of the container, stretching to fill all the space between **NORTH** and **SOUTH**
 - **BorderLayout.CENTER** sits in the middle of the container and stretches to fill all available space
- If you don't specify where you're adding a widget, it adds to **CENTER**
- If you add more than one widget to a region, the new one **replaces** the old
- Unused regions disappear



GridLayout

- **GridLayout** allows you to create a grid with a specific number of rows and columns
- All the cells in the grid are the same size
- As you add widgets, they fill each row



```
frame.setLayout(new GridLayout(4, 5));  
for(int row = 0; row < 4; ++row)  
    for(int column = 0; column < 5; ++column)  
        frame.add(new JButton("" + (row * 5 + column + 1)));
```

Action Listeners

Making buttons do things

- We have added **JButtons** to **JFrames**, but those buttons don't do anything
- When clicked, a **JButton** fires an event
- We need to add an action listener to do something when that event happens
- A CLI program runs through loops, calls methods, and makes decisions until it runs out of stuff to do
- GUIs usually have this **event-based** programming model
- They sit there, waiting for events to cause methods to get called

ActionListener interface

- What can listen for a **JButton** to click?
- Any object that implements **ActionListener**
- **ActionListener** is an interface like any other with a single abstract method in it:

```
void actionPerformed(ActionEvent e) ;
```

- We need to write a class with such a method
- We will rarely need to worry about the **ActionEvent** object
- But it does have a **getSource ()** method that will give us the **Object** (often a **JButton**) that fired the event

Anonymous inner classes

- Now, we get to something tricky
- It's possible to create a class on the fly, right in the middle of other code
- Consider the following interface:

```
public interface NoiseMaker {  
    String makeNoise();  
}
```

- We can create, in the middle of other code, a class that implements **NoiseMaker**, like this:

```
NoiseMaker maker = new NoiseMaker() {  
    public String makeNoise() {  
        return "Yowza!";  
    }  
};
```

Adding an action listener

- The reason we brought up anonymous inner classes is that we can use this syntax to make an **ActionListener** object right when we need it, for a button

```
JButton button = new JButton("Push me!");  
button.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        button.setText("Ouch!"); // arbitrary code  
    }  
}); // ugly: parenthesis for end of method call
```

- It's ugly, but it works

Java 8 style

- Before Java 8, we only had two choices:
 - Make a whole class that implements **ActionListener** and might have to do different actions based on which button fired the event
 - Make a separate anonymous inner class for every single button, each doing the action for that button
- Java 8 adds something called lambdas which actually make anonymous inner classes too, but the syntax is much nicer
- Java 8 style:

```
JButton button = new JButton("Push me!");  
button.addActionListener(e -> button.setText("Ouch!"));
```


More on Java 8 style

- An interface with only a single method in it (like **ActionListener**) is called a **functional interface**
- Java 8 lets us instantiate functional interface by filling out the method:
(Type1 arg1, Type2 arg2, ...) -> { /* method body */ }
- But if it's possible for the compiler to infer the argument types, they don't have to be written
- If you only have a single argument, you don't need parentheses
- And if you only have a single line in your method body, you don't need braces
- Multi-line example:

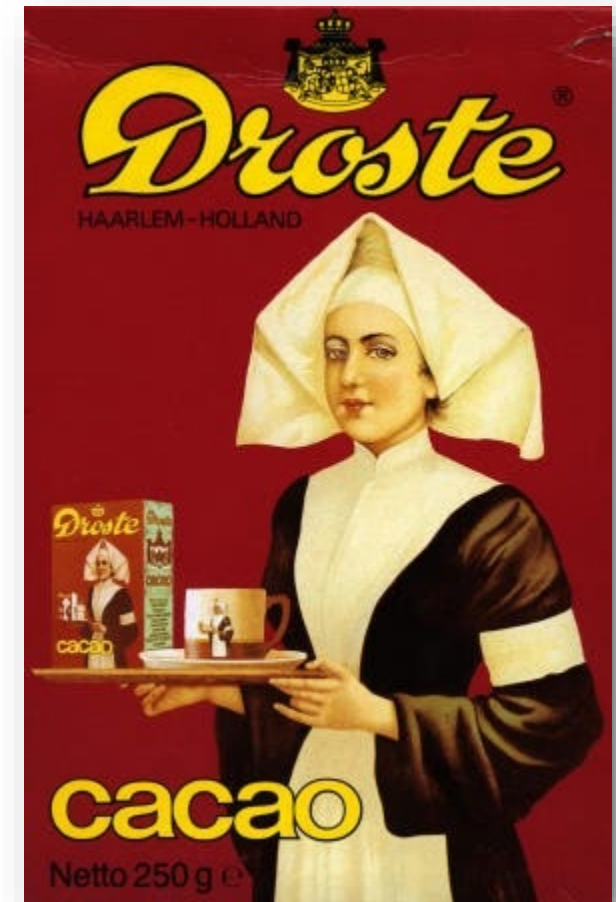
```
JButton button = new JButton("Push me!");  
button.addActionListener(e -> {  
    button.setText("Ouch!");  
    button.setEnabled(false);  
});
```

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



Useful Recursion

Two parts:

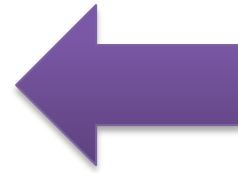
- Base case(s)
 - Tells recursion when to stop
 - For factorial, $n = 1$ or $n = 0$ are examples of base cases
- Recursive case(s)
 - Allows recursion to progress
 - "Leap of faith"
 - For factorial, $n > 1$ is the recursive case

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

Code for Factorial

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



Base Case



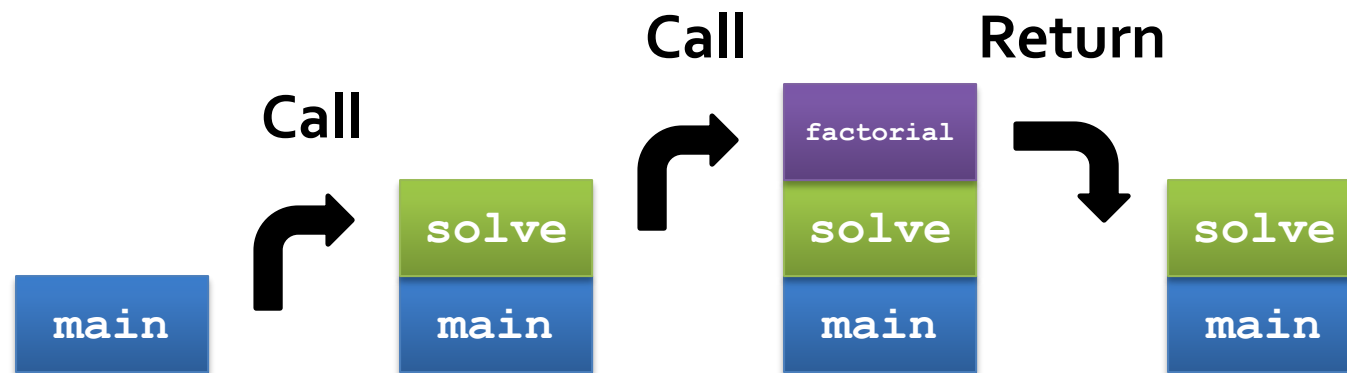
Recursive
Case

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

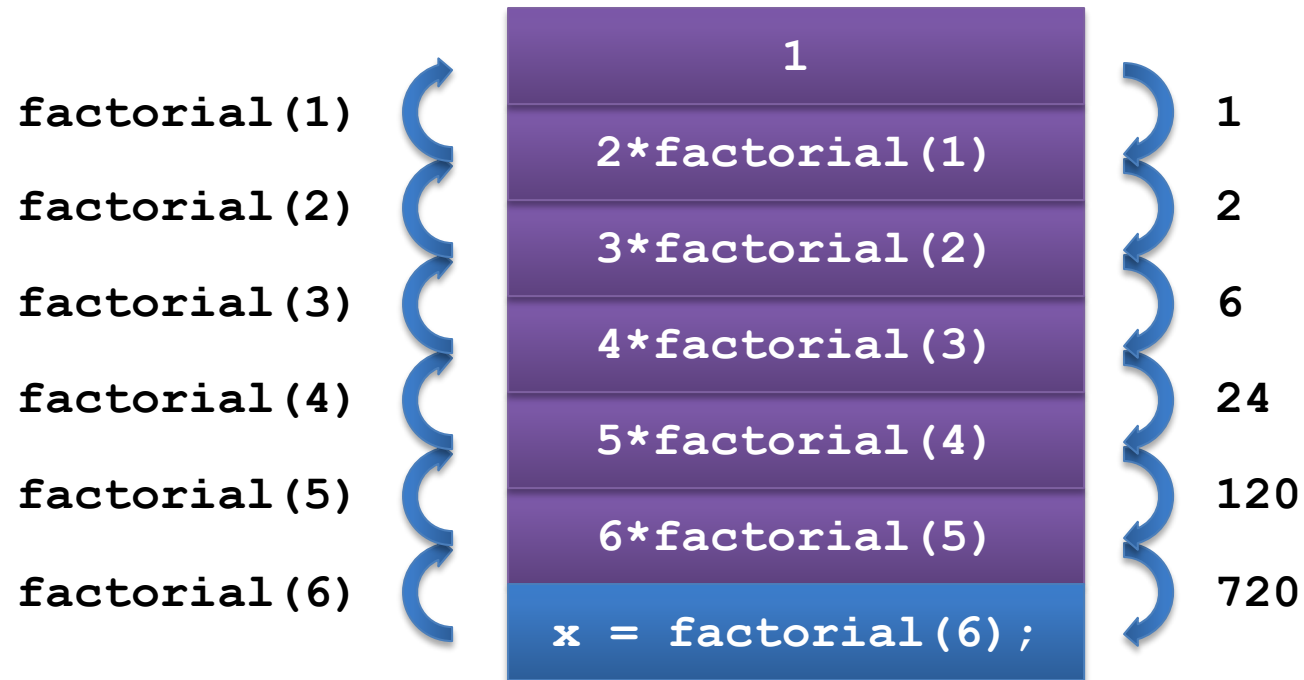
Call stack

- 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! :

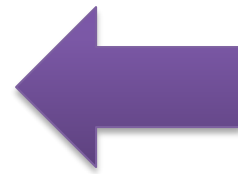


Exponentiation

- Similarly, exponentiation is repeated multiplication
- Thus, $x^y = x \cdot x \cdot x \dots \cdot x$
(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

Extra information

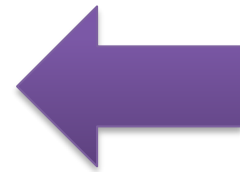
- Recursion sometimes requires similar information that can be passed along to each recursive call
- This information could be an index into a **String** or an array
- In graph or tree algorithms, it might be the parent node you visited previously
- There are recursive methods with 10 or more parameters
- There's nothing wrong with that, provided that you actually *need* them all

Summing an array

- What if we want to sum the values in an array called *array*?
- We need some extra information: current index
- Base case (*index* = *length*):
 - Sum(from *index* onward):
 - o (Nothing left to sum)
- Recursive case (*index* < *length*):
 - Sum(from *index* onward):
 $array[index] + \text{Sum}(\text{from } index + 1 \text{ onward})$

Code for summing an array

```
public static double sum(double array[], int index) {  
    if( index == array.length )  
        return 0.0;  
    else  
        return array[index] + sum(array, index + 1);  
}
```



Base Case



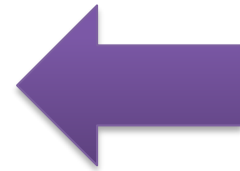
Recursive
Case

Reversing a String

- What if we want to reverse the contents of a string called *s*?
- We need some extra information: current index
- Base case (*index* = *length*):
 - Reverse(from *index* onward):
"" (Nothing left to reverse)
- Recursive case (*index* < *length*):
 - Reverse(from *index* onward):
 $s[\text{length} - \text{index} - 1] + \text{Reverse}(\text{from } \text{index} + 1 \text{ onward})$

Code for reversing a String

```
public static String reverse(String s, int index) {  
    if( index == s.length() )  
        return "";  
    else  
        return s.charAt(s.length() - index - 1) +  
            reverse(s, index + 1);  
}
```



Base Case



Recursive
Case

Using the stack to go in reverse

- All stacks (including the call stack) are first-in last-out (FILO) structures
- In situations where we want to deal with things in backwards order, we can use this natural reversing tendency
- For example, if we want to print out a **String** in reverse, we can recurse through each character and print them as the recursion returns
- Doesn't make sense yet?

Printing a String in reverse

- What if we want to print the contents of a string called *s* in reverse?
- We need some extra information: current index
- Base case (*index* = *length*):
 - ReversePrint(from *index* onward):
Print nothing
- Recursive case (*index* < *length*):
 - ReversePrint(from *index* onward):
ReversePrint(from *index* + 1 onward)
Then print *s* [*index*]

Code for printing a String in reverse

```
public static void reversePrint(String s, int index)
{
```

 (Empty)
Base Case

```
    if( index < s.length() ) {
        reversePrint(s, index + 1);
        System.out.print(s.charAt(index));
    }
```


Recursive
Case

```
}
```

Reversing a String (the remix)

- We can even use this approach to reverse a string in a different manner than we did before
- Base case (*index* = *length*):
 - Backwards(from *index* onward):
"" (Nothing left to reverse)
- Recursive case (*index* < *length*):
 - Backwards(from *index* onward):
Backwards(from *index* + 1 onward) + *s[index]*

Remixed code for reversing a String

```
public static String backwards(String s, int index) {  
    if( index == s.length() )  
        return "";  
    else  
        return backwards(s, index + 1) + s.charAt(index);  
}
```

 Base Case


Recursive
Case

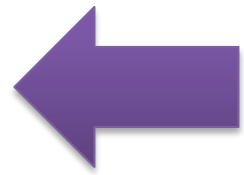
Merge Sort


Merge Sort algorithm (recursive)

- Beautiful divide and conquer algorithm
- Base case: List has size 1
 - You're done!
- Recursive case: List has size greater than 1
 - Divide your list in half
 - Recursively merge sort each half
 - Merge the two halves back together in sorted order

Merge Sort code

```
public static void mergeSort(int [] array) {  
  
    if(array.length > 1) {  
        int[] a = new int[array.length/2];  
        int[] b = new int[array.length - a.length];  
        for(int i = 0; i < a.length; ++i) //copy first half  
            a[i] = array[i];  
        for(int i = 0; i < b.length; ++i) //copy second half  
            b[i] = array[i + a.length];  
        mergeSort(a); //sort first half  
        mergeSort(b); //sort second half  
        merge(a, b, array);  
    }  
}
```

 (Empty)
Base Case

 Recursive
Case

Merging (the hard part)

- The code to merge two sorted subarrays into a third array trips up a lot of people
- Use three indexes, one for each array
- Always copy the smaller value from the two subarrays
- The tricky part is that you might no longer have anything left to copy from a subarray
- At that point, you must copy from the other subarray
- In other words, always check the validity of an index before using it

Merge code

```
public static void merge(int[] a, int[] b, int[] array) {  
    int aIndex = 0;  
    int bIndex = 0;  
    for(int i = 0; i < array.length; ++i) {  
        if(aIndex >= a.length)  
            array[i] = b[bIndex++];  
        else if(bIndex >= b.length)  
            array[i] = a[aIndex++];  
        else if(a[aIndex] <= b[bIndex])  
            array[i] = a[aIndex++];  
        else  
            array[i] = b[bIndex++];  
    }  
}
```

Merging

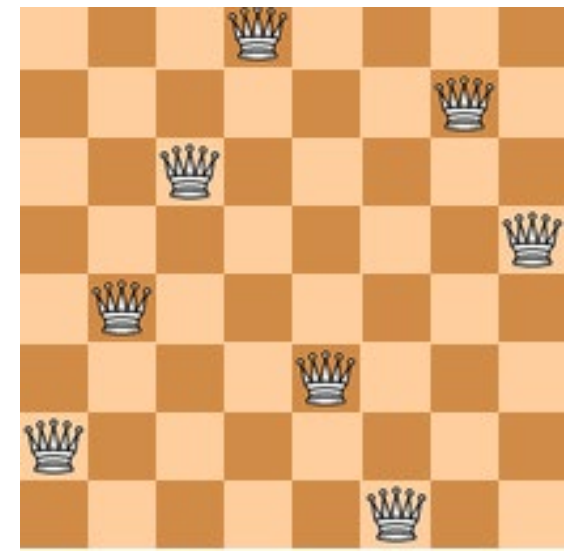
- I prefer the merging given on the previous slide
- A single **for** loop that fills the array makes sense to me
- I'm not a huge fan of using the postincrement operator (**aIndex++**), but this is what it's designed for:
 - Getting a value and then incrementing it, all in a single line of code
 - Otherwise, we'd need braces for the cases
- Note that you can combine the four seemingly repetitive cases into three cases (but not two)
- Another way to do the merge is with three **while** loops, given on the next slide

Merge code (alternative)

```
public static void merge(int[] a, int[] b, int[] array) {  
    int aIndex = 0;  
    int bIndex = 0;  
    int i = 0;  
    while(aIndex < a.length && bIndex < b.length) {  
        if(a[aIndex] <= b[bIndex])  
            array[i] = a[aIndex++];  
        else  
            array[i] = b[bIndex++];  
        ++i;  
    }  
    while(aIndex < a.length) {  
        array[i] = a[aIndex++];  
        ++i;  
    }  
    while(bIndex < b.length) {  
        array[i] = b[bIndex++];  
        ++i;  
    }  
}
```

N-Queens

- Given an $N \times N$ chess board, where $N \geq 4$ it is possible to place N queens on the board so that none of them are able to attack each other in a given move
- Write a method that, given a value of N , will return the total number of ways that the N queens can be placed



Problem solving approach

- We will use recursion to place queens on the board, one row at a time
- To save typing, we will use a loop to place the queen at each different column within the row and then recurse
 - Egad! A loop inside recursion!
 - It happens.
- If we have placed queens on all the rows, we return 1 (a successful placement)
- We sum up all the successful placements that our recursive children make

N-Queens algorithm (recursive)

- Base case: (***row*** = 8)
 - You have placed queens on rows 0-7
 - Return 1 (a successful placement)
- Recursive case: (***row*** < 8)
 - Keep a sum of the successful placements made by placing in future rows, initially 0
 - Try to place a queen on columns 0-7
 - For each successful column placement, recursively try to place queens on the next row and add those successful placements to your sum
 - Return sum

Files

Text files

- First, we're going to talk about **text files**
- All files are sequences of bytes stored in binary, but in text files, those bytes form human-readable text like words and numbers
- Unlike files storing data in binary, working with text files is similar to the command-line I/O we've been doing since before COMP 2000
- Examples of text files:
 - Source code for most programming languages (**.c**, **.java**, **.py** files, etc.)
 - Plain text files (often with a **.txt** extension)
 - Many configuration and log files

Reading

- Reading from a text file is straightforward
- We use **Scanner**, just like reading from the command line
- We just have to create a new **File** object that gives the file path we want to read from

```
Scanner in = new Scanner(new File("input.txt")) ;
```

- This code will read from some file called **input.txt**, as if someone were typing its contents into the command line

Scanner methods

- Recall that we can read correctly formatted text with a **Scanner** using the following methods
 - **nextInt()** Reads an **int** value
 - **nextDouble()** Reads a **double** value
 - **next()** Reads a white-space delimited **String**
 - **nextLine()** Reads a **String** up to the next newline (which can cause problems if there's a newline left over from previous reads)
- These methods are usually what you need to get the job done, but there are also **nextBoolean()**, **nextByte()**, **nextFloat()**, **nextLong()**, and **nextShort()** methods
- Note that all the integer reading methods have a second version that takes a base so that you can read values in bases 2-36

Writing

- Writing to files uses a different sequence of steps
- If you want to write to a text file, you've got to create a **PrintWriter** object, based on a **FileOutputStream** object (which takes the file name as a parameter)

```
PrintWriter out = new PrintWriter(new  
    FileOutputStream ("output.txt")) ;
```

- Once you've got a **PrintWriter**, you can use it just like **System.out**

Exceptions

- When making a **Scanner** from a **File** or making a **PrintWriter** from a **FileOutputStream** can potentially throw a **FileNotFoundException**
- Since it's a checked exception, you need a **try-catch** or a **throws**

Writing example

- This example opens a file called **goodbye.txt**, writes some text, and then closes the file
- Note that we are not showing the **try-catch** or **throws**

```
PrintWriter out = new PrintWriter(new  
    FileOutputStream("goodbye.txt"));  
out.println("So long!");  
out.println("Farewell!");  
out.println("Auf Wiedersehen!");  
out.println("Goodbye!");  
out.close();
```

Shut 'em down!

- You should always close files as soon as you're done reading from them or writing to them
- If you don't close files you're writing to before your program ends, output can be lost
- Keeping files open ties up system resources
- There's a maximum number of files one program can have open at a time
- Since we always want to close files, it's smart to put the closing in a **finally** block

Full example

- This example copies the text from `input.txt` to `output.txt`

```
Scanner in = null;
PrintWriter out = null;
try {
    in = new Scanner(new File("input.txt"));
    out = new PrintWriter(new FileOutputStream("output.txt"));
    while(in.hasNextLine())
        out.println(in.nextLine());
}
catch(FileNotFoundException e) {
    e.printStackTrace();
}
finally {
    if(in != null) in.close();
    if(out != null) out.close();
}
```


Why use binary files?

- Wouldn't it be easier to use all human readable files?
- Binary files can be more efficient
 - In binary, all `int` values are 4 bytes
- In text, they can take up a lot more
- In text, you also need a space or other separator to divide the numbers

| Integer | Bytes in text representation |
|-------------|------------------------------|
| 0 | 1 |
| 92 | 2 |
| 789 | 3 |
| 4551 | 4 |
| 10890999 | 8 |
| 204471262 | 9 |
| -2000000000 | 11 |

Most files are binary files

- Because they have a representation that is more compact (and more similar to how data is stored in your program), most files are binary (non-human-readable) files
- Many media files start with **metadata**
 - Format information
 - Size
- Then, they have the actual data (RGB values, audio samples, frames of video, etc.)
- Binary files include most common file formats: **.jpg**, **.png**, **.mp3**, **.avi**, **.pdf**, **.docx**, **.pptx**, and on and on

Reading binary files

- Reading from binary files uses a completely different set of objects than reading from text files
- We create a **DataInputStream** from a **FileInputStream**
- The **FileInputStream** takes the name of the file path

```
DataInputStream in = new DataInputStream(new  
    FileInputStream("input.dat"));
```

- You can create a **FileInputStream** first on a separate line, but there's no need to do so

Example summing double values

- The following code assumes that a file contains starts with an **int** value giving the number of **double** values that come after it

```
DataInputStream in = null;
try {
    in = new DataInputStream(new FileInputStream("numbers.dat"));
    int length = in.readInt();
    double sum = 0.0;
    for(int i = 0; i < length; ++i)
        sum += in.readDouble();
    System.out.println("Sum: " + sum);
}
catch(IOException e) {
    System.out.println("File problems!");
}
finally {
    try{ in.close(); } catch(Exception e){}
}
```

Error handling

- The reading methods in **`DataInputStream`** can throw:
 - **`EOFException`** if the end of the file was reached but you still try to read something
 - **`IOException`** if the stream was closed (or something else goes wrong)
- Since **`EOFException`** and even **`FileNotFoundException`** are both children of **`IOException`**, it's possible (as we did on the previous slide) to have a single catch block that handles an **`IOException`**

Closing the file

- As with text files, we closed our files in a **finally** block
- You might have noticed that there was a baby **try-catch** block inside of there as well

```
finally {  
    try{ in.close(); } catch(Exception e){}  
}
```

- For whatever reason, closing a **DataInputStream** can throw an **IOException**
- By having a **try-catch** that will catch anything, we deal with the **IOException** as well as catching the **NullPointerException** that happens if we try to close a **null DataInputStream**
- Is that a good idea?
- Eh...it's fine: We're just trying to close the file and not crash our program

Writing binary files

- Writing to binary files is very similar to reading from binary files
- We create a **DataOutputStream** from a **FileOutputStream**
- The **FileOutputStream** takes the name of the file path

```
DataOutputStream out = new FileOutputStream(new  
    FileOutputStream("output.dat"));
```

- The writing methods are similar too

Example writing double values

- The following code assumes that a file starts with an **int** value giving the number of **double** values that come after it

```
DataOutputStream out = null;
try {
    out = new DataOutputStream(new FileOutputStream("numbers.dat"));
    out.writeInt(100);
    for(int i = 0; i < 100; ++i)
        out.writeDouble(Math.random() * 1000);
}
catch(IOException e) {
    System.out.println("File problems!");
}
finally {
    try{ out.close(); } catch(Exception e){}
}
```


Putting the I/O together

- File input and output need to match each other well, especially for binary I/O
- If data values are out of order, you'll get garbage, and it'll be hard to know why
- Once you write the file output code, you can easily copy and paste it to write the input code
 - Change every **out** to **in**
 - Change every **write** to **read** (and move the method arguments to save return values)
- The structures are parallel

Serialization

- Serialization takes a reference to an object and dumps it into a file
- It writes representations to primitive types pretty much the same way that a **DataOutputStream** does
- And if there're objects inside of the object you're serializing, it serializes them too
- **And!** Serialization makes a note of all the objects that are getting serialized, so if it sees an object a second time, it just writes down a serial number for it instead of the whole thing

Serializable interface

- Serialization is one of the closest things to magic you'll see in programming
- You only need to implement the **Serializable** interface on your object
 - And the **Serializable** interface has no methods!
- It's just a way of marking an object as reasonable to try to dump into a file
- Most objects are reasonable to dump into a file!

Example Serializable class

- Here's a class we might want to be able to dump into a file

```
public class Troll implements Serializable {  
    private String name;  
    private int age;  
    private Object hatedThing; // All trolls hate something  
    public Troll(String name, int age, Object hatedThing) {  
        this.name = name;  
        this.age = age;  
        this.hatedThing = hatedThing;  
    }  
    public Object getHatedThing() {  
        return hatedThing;  
    }  
}
```

Writing using serialization

- To write an object marked **Serializable**, you need to create an **ObjectOutputStream**
- You create an **ObjectOutputStream** the same way that you create a **DataOutputStream**, by passing in a **FileOutputStream**
 - At this point, you might be wondering why all these objects take **FileOutputStream** objects and can't take just take a **File** object or even a file name
 - In actuality, you can pass in any **OutputStream** object (of which **FileOutputStream** is a child), like maybe one that sends the data across the network instead of storing it into a file
- An **ObjectOutputStream** object has many methods, but the only one that matters is **writeObject()**
- Pass your object to that method and it'll get written out in its totality, no fuss

Example of writing

- Here's some code that creates a couple of **Troll** objects and then writes them to a file called **trolls.dat**

```
Troll tom = new Troll("Tom", 351, "Bilbo Baggins");
Troll bert = new Troll("Bert", 417, tom);
ObjectOutputStream out = null;
try {
    out = new ObjectOutputStream(new FileOutputStream("trolls.dat"));
    out.writeObject(tom);
    out.writeObject(bert);
}
catch (IOException e) {
    System.out.println("Serialization failed.");
}
finally { try{ out.close(); } catch (Exception e){} }
```

Reading using serialization

- To read objects that have been serialized to a file, you need to create an **ObjectInputStream**
- You create an **ObjectOutputStream** the same way that you create a **DataInputStream**, by passing in a **FileInputStream**
- For each object serialized, you call the **readObject()** method to restore it from the file
- Note that **readObject()** has a return type of **Object**, so you'll need to cast your object if you want to store it in a reference of its own type

Example of reading

- Here's some code that reads in the **Troll** objects we serialized in the previous example

```
Troll tom = null;
Troll bert = null;
ObjectInputStream in = null;
try {
    in = new ObjectInputStream(new FileInputStream("trolls.dat"));
    tom = (Troll)in.readObject();
    bert = (Troll)in.readObject();
}
catch(IOException e) {
    System.out.println("Deserialization failed.");
}
finally { try{ in.close(); } catch(Exception e){} }
```

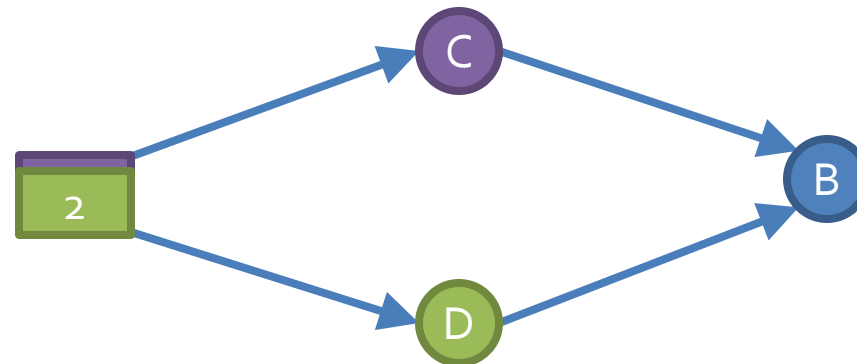

Networking

What is the Internet?

- The network of hardware and software systems that connects many of the world's computers
- Typically, people say the Internet and capitalize the "I" because there is only one
 - Until we meet aliens
 - Or decide to break off from the rest of the world
- The World Wide Web is the part of the Internet that is concerned with webpages
- The Internet also includes:
 - FTP
 - VOIP
 - Bittorrent
 - Multiplayer video games
 - Much, much more...

Packet switched

- The Internet is a packet switched system
- Individual pieces of data (called packets) are sent on the network
 - Each packet knows where it is going
 - A collection of packets going from point **A** to point **B** might not all travel the same route



IP addresses

- Computers on the Internet have addresses, not names
- **Google.com** is actually **[74 . 125 . 67 . 100]**
- **Google.com** is called a **domain**
- The Domain Name System or DNS turns the name into an address

IPv4

- Old-style IP addresses are in this form:
 - **74 . 125 . 67 . 100**
- 4 numbers between 0 and 255, separated by dots
- That's a total of $256^4 = 4,294,967,296$ addresses
- But there are 7 billion people on earth...

IPv6

- IPv6 are the new IP addresses that are beginning to be used by modern hardware
 - 8 groups of 4 hexadecimal digits each
 - **2001:0db8:85a3:0000:0000:8a2e:0370:7334**
 - 1 hexadecimal digit has 16 possibilities
 - How many different addresses is this?
 - $16^{32} = 2^{128} \approx 3.4 \times 10^{38}$ is enough to have 500 trillion addresses for every cell of every person's body on Earth
 - Will it be enough?!

OSI 7 layer model

- Not every layer is always used
- Sometimes user errors are referred to as Layer 8 problems

| Layer | Name | Mnemonic | Activity | Example |
|-------|---------------------|-------------|---|---------------------|
| 7 | Application | Away | User-level data | HTTP |
| 6 | Presentation | Pretzels | Data appearance, some encryption | Unicode |
| 5 | Session | Salty | Sessions, sequencing, recovery | TLS |
| 4 | Transport | Throw | Flow control, end-to-end error detection | TCP |
| 3 | Network | Not | Routing, blocking into packets | IP |
| 2 | Data Link | Dare | Data delivery, packets into frames, transmission error recovery | Ethernet |
| 1 | Physical | Programmers | Physical communication, bit transmission | Electrons in copper |

Physical layer

- There is where the rubber meets the road
- The actual protocols for exchanging bits as electronic signals happen at the physical layer
- At this level are things like RJ45 jacks and rules for interpreting voltages sent over copper
 - Or light pulses over fiber

Data link layer

- Ethernet is the most widely used example of the data layer
- Machines at this layer are identified by a 48-bit Media Access Control (MAC) address
- The Address Resolution Protocol (ARP) can be used for one machine to ask another for its MAC address
- Some routers allow a MAC address to be spoofed, but MAC addresses are intended to be unique and unchanging for a particular piece of hardware

Network layer

- The most common network layer protocol is Internet Protocol (IP)
- Each computer connected to the Internet should have a unique IP address
 - IPv4 is 32 bits written as four numbers from 0 – 255, separated by dots
 - IPv6 is 128 bits written as 8 groups of 4 hexadecimal digits
- We can use **tracert** on Windows to see the path of hosts leading to some IP address

Transport layer

- There are two popular possibilities for the transport layer
- Transmission Control Protocol (TCP) provides reliability
 - Sequence numbers for out of order packets
 - Retransmission for packets that never arrive
- User Datagram Protocol (UDP) is simpler
 - Packets can arrive out of order or never show up
 - Many online games use UDP because speed is more important

Session layer

- This layer doesn't necessarily exist in the TCP/IP model
- Transport Layer Security (TLS) uses the session layer
- TLS is the end-to-end encryption that HTTPS uses
- You know you're using TLS if there's a little lock showing on your browser
- Google is pushing for all websites to be HTTPS
- HTTPS is safer, but there's some overhead for the encryption, and websites have to have certificates for their public keys

Presentation layer

- The presentation layer is often optional
- It specifies how the data should appear
- This layer is responsible for character encoding (ASCII, UTF-8, etc.)
- MIME types are sometimes considered presentation layer issues

Application layer

- This is where the data is interpreted and used
- HTTP is an example of an application layer protocol
- A web browser takes the information delivered via HTTP and renders it
- Code you write deals significantly with the application layer

Mnemonics

- Seven layers is a lot to remember
- Mnemonics have been developed to help

| Application | All | All | A | Away |
|--------------|--------|------------|--------------|-------------|
| Presentation | Pros | People | Powered-Down | Pretzels |
| Session | Search | Seem | System | Salty |
| Transport | Top | To | Transmits | Throw |
| Network | Notch | Need | No | Not |
| Data Link | Donut | Data | Data | Dare |
| Physical | Places | Processing | Packets | Programmers |

TCP/IP

- The OSI model is sort of a sham
 - It was invented after the Internet was already in use
 - You don't need all layers
 - Some people think this categorization is not useful
- Most network communication uses TCP/IP
- We can view TCP/IP as four layers:

| Layer | Action | Responsibilities | Protocol |
|-------------|------------------------------|---|-----------------|
| Application | Prepare messages | User interaction | HTTP, FTP, etc. |
| Transport | Convert messages to packets | Sequencing, reliability, error correction | TCP or UDP |
| Internet | Convert packets to datagrams | Flow control, routing | IP |
| Physical | Transmit datagrams as bits | Data communication | |

TCP/IP

- A TCP/IP connection between two hosts (computers) is defined by four things
 - Source IP
 - Source port
 - Destination IP
 - Destination port
- One machine can be connected to many other machines, but the port numbers keep the different connections straight

Common port numbers

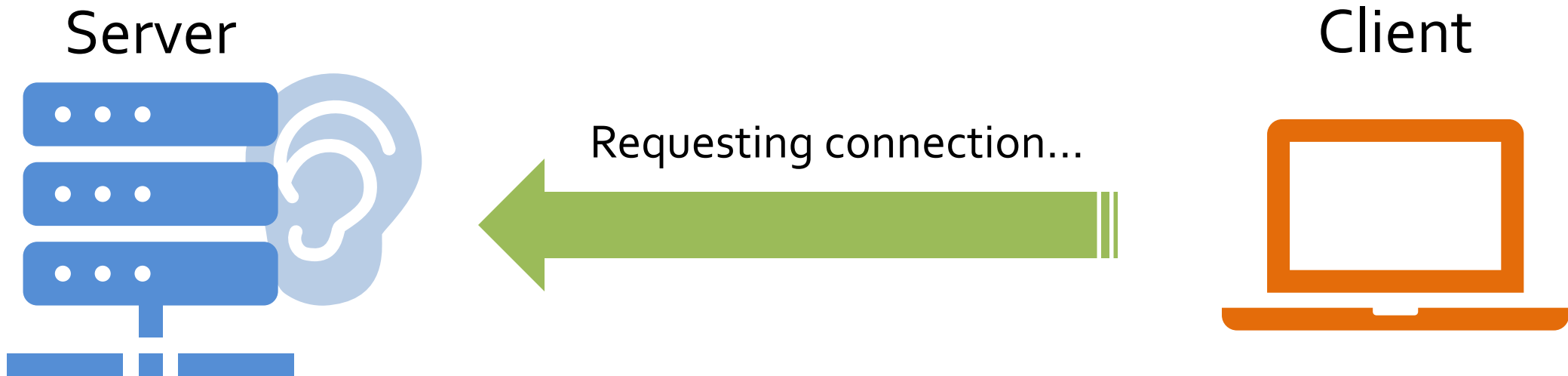
- Certain kinds of network communication are usually done on specific ports
 - 20 and 21: File Transfer Protocol (FTP)
 - 22: Secure Shell (SSH)
 - 23: Telnet
 - 25: Simple Mail Transfer Protocol (SMTP)
 - 53: Domain Name System (DNS) service
 - 80: Hypertext Transfer Protocol (HTTP)
 - 110: Post Office Protocol (POP₃)
 - 443: HTTP Secure (HTTPS)

Clients vs. servers

- Using sockets is usually associated with a client-server model
- A **server** is a process that sits around waiting for a connection
 - When it gets one, it can do sends and receives
- A **client** is a process that connects to a waiting server
 - Then it can do sends and receives
- Clients and servers are processes, not computers
 - You can have many client and server processes on a single machine

Listening server

- The server sits there, waiting for a client to connect
- Until that happens, the **accept()** method will not return
- When it does return, it will return with a socket that can be used for communicating with the client



Loopback IP address

- It's inconvenient to need two different computers to write network code
- For testing purposes, you can often use a single computer as both the server and the client
- To do so, you need to connect to yourself
- What's your IP address?
- Well, it might always be changing
- To make things simpler, there's a loopback IP address that always refers to the computer you're currently on: **127.0.0.1**
- The IPv6 loopback address is **::1** (where **::** is notation that means "fill in with appropriate numbers of zeroes")

Upcoming

Next time...

- Exam 2 on Monday

Reminders

- **Work on Project 3**
 - **Project 3 is now due on April 3**
- Review everything after Exam 1
- Exam 2 is Monday, March 30