Week 6 - Monday

COMP 2000

Last time

- What did we talk about last time?
- **`JFrame`**
- Widgets
  - **`JButton`**
  - **`JLabel`**
  - **`JTextField`**
  - **`JTextArea`**
- Started layout

Questions?

Project 2

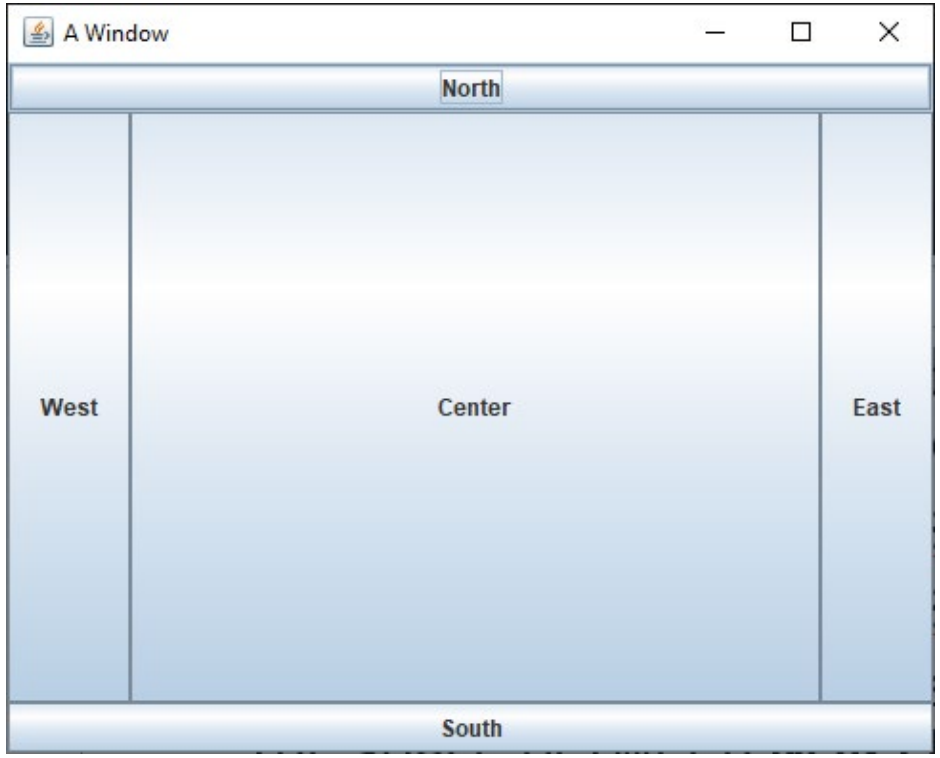
Layout Managers

Layout managers

- When you add a widget to a **`JFrame`** (or to a **`JPanel`**), its layout manager determines how it will be arranged
- There are lots of layout managers, but it's worth mentioning four:
  - **`BorderLayout`**
  - **`GridLayout`**
  - **`FlowLayout`**
  - **`BoxLayout`**
- Note that we won't talk about **`BoxLayout`**, but you should look it up if you get serious about Swing GUIs
- **`BoxLayout`** makes it easy to arrange widgets in a horizontal or vertical line, with different amount of spacing between widgets

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



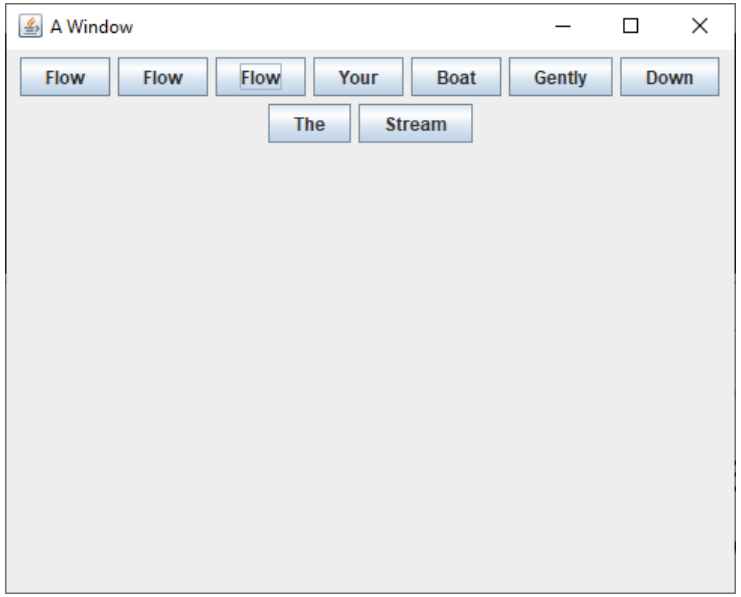
GridLayout

- **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)));
```

FlowLayout

- **FlowLayout** is the default layout manager for **JPanel**
- Widgets are arranged in centered rows in **FlowLayout**
- If you keep adding widgets to a **FlowLayout**, they'll fill the current row until there's no more room
- Then, they'll flow onto the next row
- It's ugly but easy to use



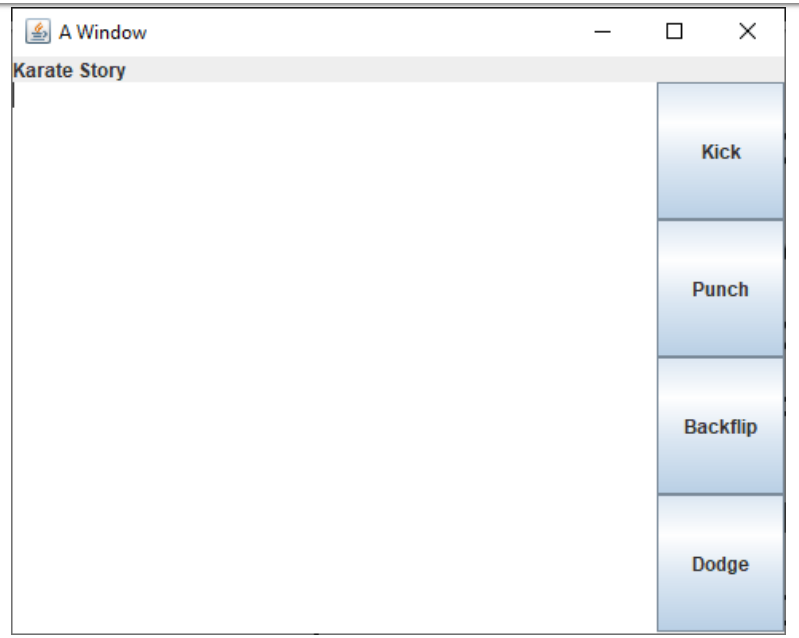

```
frame.setLayout(new FlowLayout());
frame.add(new JButton("Flow"));
frame.add(new JButton("Flow"));
frame.add(new JButton("Flow"));
frame.add(new JButton("Your"));
frame.add(new JButton("Boat"));
frame.add(new JButton("Gently"));
frame.add(new JButton("Down"));
frame.add(new JButton("The"));
frame.add(new JButton("Stream"));
```

JPanel

- A **JPanel** is an invisible container that:
  - Acts like a widget in that you can add it to a **JFrame** (or another **JPanel**)
  - Can hold other widgets
  - Can have its layout customized with a layout manager
- What if you have a **BorderLayout** and you want the **EAST** region to contain widgets arranged with a **GridLayout**?
- Easy: you create a **JPanel**, set its layout manager to **GridLayout**, add it to the **EAST** region, then add widgets to the **JPanel**

Complicated layouts

- For complicated layouts
  - Sketch out what you want it to look like
  - Use **`BorderLayout`**s to give components a spatial relationship
  - Nest **`JPanel`**s inside of **`JPanel`**s (inside of **`JPanel`**s…) if you need to
  - Use **`GridLayout`**s whenever you want to have a grid
  - Be patient: it's hard to get it right the first time




```
JPanel buttonPanel = new JPanel(new GridLayout(4,1));
buttonPanel.add(new JButton("Kick"));
buttonPanel.add(new JButton("Punch"));
buttonPanel.add(new JButton("Backflip"));
buttonPanel.add(new JButton("Dodge"));
frame.add(buttonPanel, BorderLayout.EAST);
frame.add(new JLabel("Karate Story"), BorderLayout.NORTH);
frame.add(new JTextArea(), BorderLayout.CENTER);
```

Calculator example

- Make the GUI for a calculator
- Title the **`JFrame`** "Calculator"
- Text field at the top giving the current value
- Grid of 16 buttons (**0-9**, **.**, **+**, **-**, *****, **/**, **Enter**)

Action Listeners

Making buttons do things

- We have added **`JButton`**s to **`JFrame`**s, 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!";
    }
};
```

Anonymous inner classes continued

```
System.out.println("Minding my business..."); // normal code
NoiseMaker maker = new NoiseMaker() {           // create a class
     public String makeNoise() {
          return "Yowza!";
     }
};
```

- What the hell is that?
- Aren't we instantiating an interface, which is impossible?
- No, this makes a **new, unnamed** class that implements an interface or extends a parent class, on the fly, at the same moment you're allocating it

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

Things you might do in an action listener

- Call arbitrary methods
- **`setText()`** sets the text on many widgets
- **`getText()`** gets the text from widgets so you can do something with it
- Both **`setText()`** and **`getText()`** apply to:
  - **`JButton`**
  - **`JLabel`**
  - **`JTextField`**
  - **`JTextArea`**
- **`setIcon()`** sets the icon on many widgets
  - **`JButton`**
  - **`JLabel`**
- **`setEnabled()`** can be used to enable and disable buttons

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);
});
```

Weird rules

- Using lambdas looks cleaner, but the same anonymous inner classes are being created
- When you write code in the method of an anonymous inner class
  - You can refer to member variables and methods in the anonymous inner class (if any)
  - You can refer to member variables and methods in the surrounding object (even private ones)
  - You can generally read the values of local variables, but you **cannot** change them
- Don't worry too much about all this
- Just write your action listeners and come see me if you have problems

Make the calculator work

- Using this information about action listeners, we should be able to make calculator GUI we created functional
- Buttons **0-9** and **.** should add the appropriate symbol to the display **`JTextField`**
- Buttons **+**, **-**, *, and **/**
  - Should parse what's in the **`JTextField`** into a **`double`** and store it in a member variable
  - Store the appropriate operation in a member variable (maybe as a **`char`**?)
  - Should clear the **`JTextField`**
- Button **=**
  - Should parse what's in the **`JTextField`** into a double
  - Perform the operation that was stored earlier with this value and the value stored earlier
  - Should put the result back in the **`JTextField`**

Upcoming

Next time…

- Finish calculator example
- Mouse listeners
- Playing sounds
- Menus

Reminders

- Keep reading Chapter 15
- Keep working on Project 2