CS 2113
Software Engineering

Java 5 - GUIs

Import the code to intelliJ
https://github.com/cs2113f18/template-j-5.git

Professor Tim Wood - The George Washington University

Last Time…

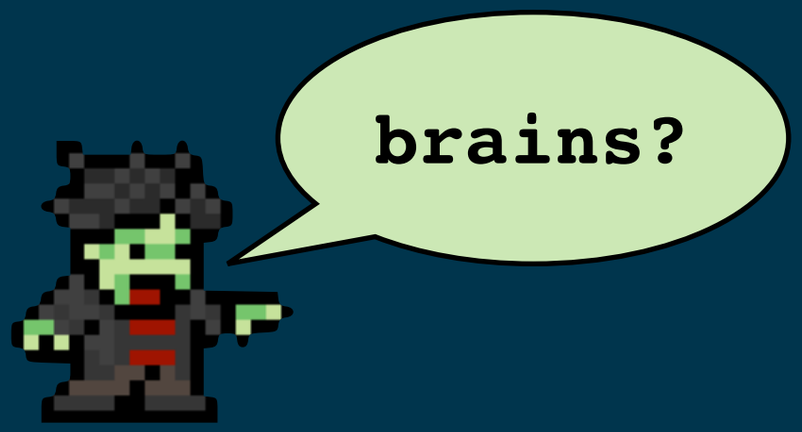
- Class Hierarchies
- Abstract Classes
- Interfaces

This Time

- GUIs in Java
  - AWT vs Swing
  - Swing Basics
- Event Handling
  - Inner and Anonymous classes

Project 2...

Zombie Infestation Simulator



Zombie Sim Structure

- ZombieSim
  - main()
  - instantiates city
  - loop: update city and draw
- City
  - private Walls[][]
  - update
  - draw
  - populate()
  - what else to add???

```
Tips/Best Practices:
 - Think carefully about class
structure and the data and
functions in each one
 - Think carefully about the "is
a" versus "has a" relationship
when designing your classes
 - It is better to have a class
interact with another using an
API (functions) instead of
directly accessing data
 - Use classes to encapsulate
both data and functions. A City
class should be responsible for
everything to do with the city
and a Cat class would be
responsible for everything to do
with cats, etc.
```

Labs

- Starting next week:

- “Over-achievers”
  - Should not go to lab
- “Middlers”
  - Must go to lab
- “Strugglers”
  - Must go to lab and start assignments early

- You will get an email with your group (based on grade in course)

Quiz: bit.ly/petsQuiz2

Click Fork, then Edit

Write a program to:

- Store two types of pets---cats and dogs
  - When you create a pet, the constructor takes a name. Cats also take a number of lives remaining.
  - All pets have a printName() function that prints the name
  - All pets have a makeNoise() function
    - Cats: "NAME says meow" and dogs: "NAME says woof"
- Your main method should:
  - Create a single array with two dogs named Fido and Spot, and three cats named Fluffy, Mowzer, and Pig
  - Print the names of all the pets
  - Call the makeNoise function on the first dog and second cat

- Use good OOP practices!

What is a GUI library?

- A way to:
  - Open windows
  - Display **widgets** on screen
  - Process **events**

- Widgets:
  - Buttons, images, Menu bars, tabs, popups, etc

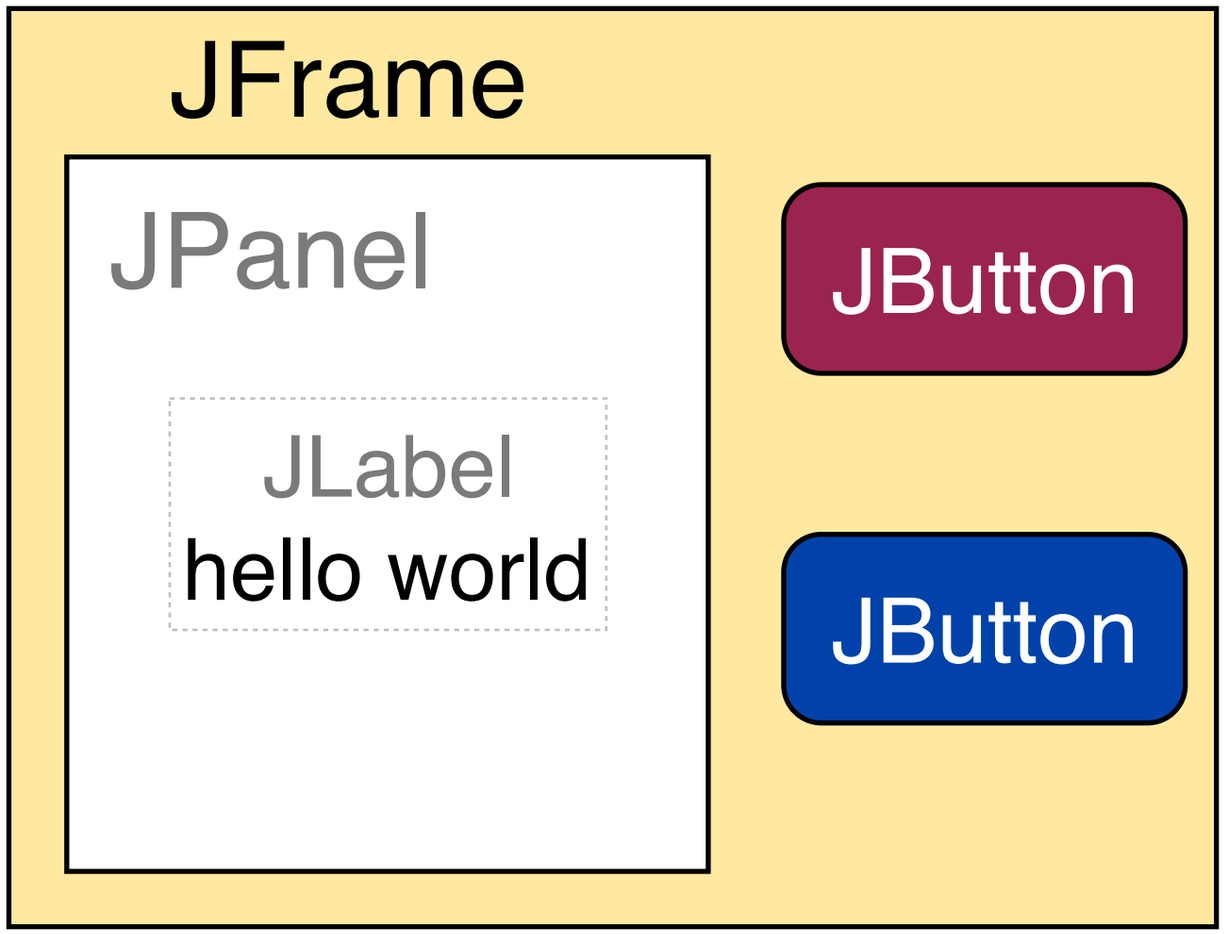
- Events:
  - Mouse clicks, keyboard interactions, windows being moved/resized/minimized/closed, etc

GUIs in Java

- **A**bstract **W**indow **T**oolkit (AWT)
  - Java library to interact with the OS's **native** graphical interface tools
- Swing
  - Interface library relying (almost) purely on Java
- JavaFX
  - Newer redesign of Swing
- **We will use Swing**

GUIs are made up of:

- Containers
  - Holds other widgets
- Components
  - A widget to interact with or display something



- Common examples:
  - Frame: basic window
  - Panel: an area to group other objects or draw images/art
  - TextField/TextArea: allows text input
  - Simple widgets: Checkbox, List Button, Label, Scrollbar and Scrollpane.
- Swing widget classes all start with "J"

Our First Window

- Is this code enough?

```
import javax.swing.*;

public class TestSwing1 {
  public static void main (String[] argv)
  {
    JFrame f = new JFrame ();
  }
}
```

Our First Window

- Is this code enough?

```
import javax.swing.*;

public class TestSwing1 {
  public static void main (String[] argv)
  {
    JFrame f = new JFrame ();
  }
}
```

- Nope!
- Also need to:
  - Give the window a size and make itself visible

Open a Window

- Get the code for today from the class site

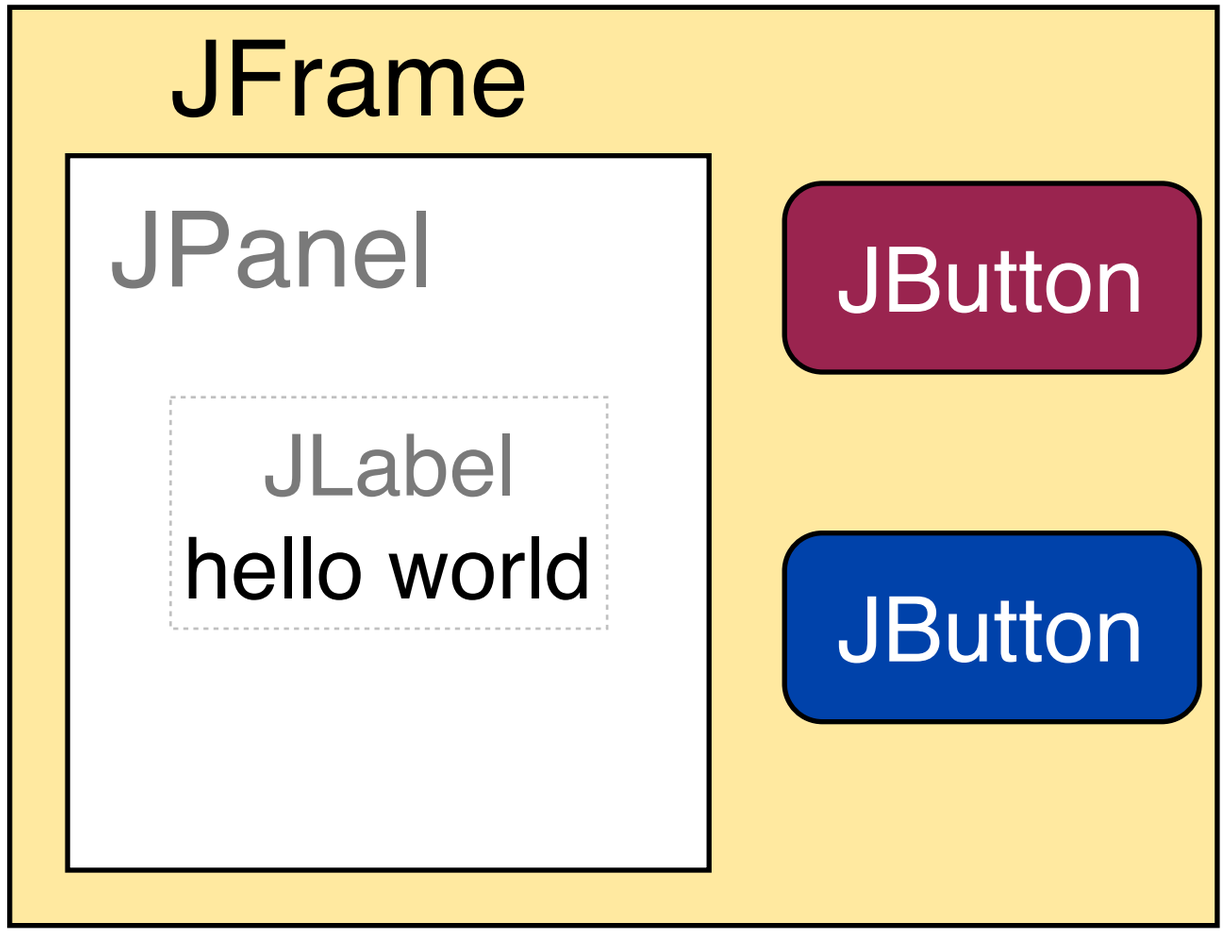
- Look at the **guis.HelloSwing.java** file
  - What happens when you run it?
  - What happens when you try to close the window?

- Can you figure out how to set the title of the window to "Hello World"?

```
More fun:
 - have the window appear at a specific location
 - open five windows instead of one
```

Content Pane

- A JFrame has a **ContentPane** to hold widgets
- Can use this to:
  - Make low level drawing calls (strings, circles, lines, etc)
  - Add components like buttons, sliders, and other containers



Draw me a picture

- Draw a pretty picture
  - Edit the **guis.PrettyPicture.java** file

- **drawRect**(int topleftx, int toplefty, int width, int height):
  - The first two integers specify the topleft corner.
  - The next two are the desired width and height of the rectangle.
- **drawOval**(int topleftx, int toplefty, int width, int height):
  - The first two integers specify the topleft corner.
  - The next two are the desired width and height of the enclosing rectangle.
- Also have filledRect and filledOval equivalents
- **drawLine**(int x1, int y1, int x2, int y2):
  - Unfortunately, the line thickness is fixed at one pixel.
  - To draw thicker lines, you have to "pack" one-pixel lines together yourself.

Another way to say "hello"

- It doesn't always make sense to use drawString()
  - Low level function
  - What if we want to change the text dynamically?
  - Does not feel very "object oriented"

- Can also use the **JLabel** component

```
Container cPane = f.getContentPane();
JLabel helloLabel = new JLabel("Hello!");
cPane.add(helloLabel);
```

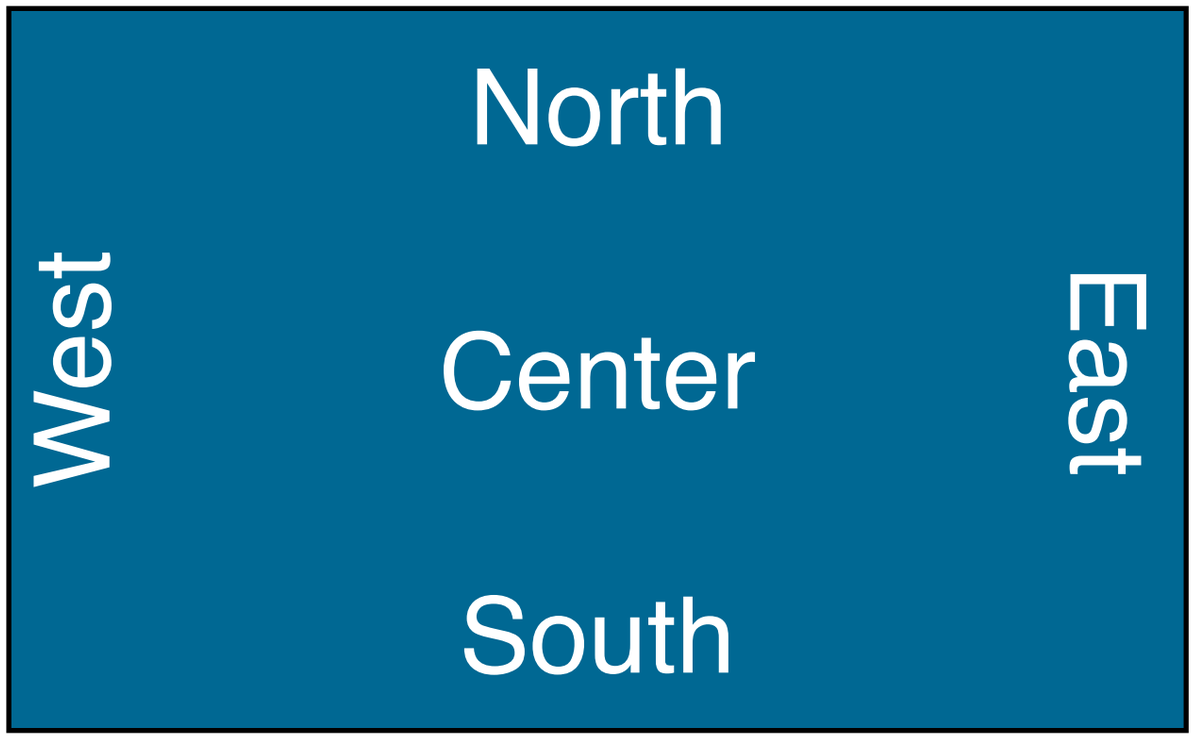
- Gives us an object to store a message
- Add it to a panel/frame and it will be drawn automagically!

JLabel

- Try out **guis.HelloSwing2.java**
- At a low level, how do you think JLabel works?
- Where does the label appear?
- What happens if you create another JLabel and add it to the frame as well?

Java Layout Managers

- Swing (and AWT) use Layout Managers to control where components are placed
  - You (generally) do not have precise control over placement
  - Simplifies automated GUI creation
  - Makes hand designed GUIs trickier
- Default layout manager: BorderLayout



JLabel take two

- You can specify (approximately) where to add a component with:

```
cPane.add(helloLabel, BorderLayout.WEST);
// or .EAST, NORTH, SOUTH, CENTER
```

- Add a second JLabel so that it does NOT replace the first one

```
More fun:
 - go back to PrettyPicture.java and make it add
 three PrettyPanels to the window next to each
 other
```

More Layouts

- Commonly used layouts managers:
  - **BorderLayout**: tries to place components either in one of five locations: North, South, East, West or Center (default).
  - **FlowLayout**: places components left to right and row-by-row.
  - **CardLayout**: displays only one component at a time, like a rolodex.
  - **GridLayout**: places components in a grid.
  - **GridBagLayout**: uses a grid-like approach that allows for different row and column sizes.
- Change a container's layout with:

```
Container cPane = f.getContentPane();
cPane.setLayout(new FlowLayout());
```

Events and Listeners

- Clicking a button, tapping a key, or moving the mouse causes events

- What happens if a tree falls in a forest and nobody is there to hear it?
  - Same idea with buttons

- How do you think events should work code-wise?

How should this work?

- Any class that is a MouseListener should implement the following classes:

```
public void mouseClicked(MouseEvent m);

public void mouseEntered(MouseEvent m);

public void mouseExited(MouseEvent m);

public void mousePressed(MouseEvent m);

public void mouseReleased(MouseEvent m);
```

- What support does Java provide to ensure our class will definitely handle these methods?

Event Interfaces

- Java uses Interfaces!
  - Interfaces define a contract - you must implement the methods
- Then we can:
  - Define a new class that implements the appropriate interface
  - Tell a JFrame that our class can definitely support events related to dragging the mouse around
  - Tell a JButton that our class can definitely support events related to clicking it

Mouse/Keyboard Events

- Interfaces for basic mouse and keyboard events

- **MouseListener**

```
public void mouseClicked(MouseEvent m);

public void mouseEntered(MouseEvent m);

public void mouseExited(MouseEvent m);

public void mousePressed(MouseEvent m);

public void mouseReleased(MouseEvent m);
```

- **KeyListener**

```
public void keyTyped(java.awt.event.KeyEvent arg0);

public void keyPressed(java.awt.event.KeyEvent arg0);

public void keyReleased(java.awt.event.KeyEvent arg0);
```

Button Events

- **actionListener**
  - Must implement one function:

```
public void actionPerformed (ActionEvent a)
```

- ActionEvent object:

a.getActionCommand( )
Returns the command string associated with this action.

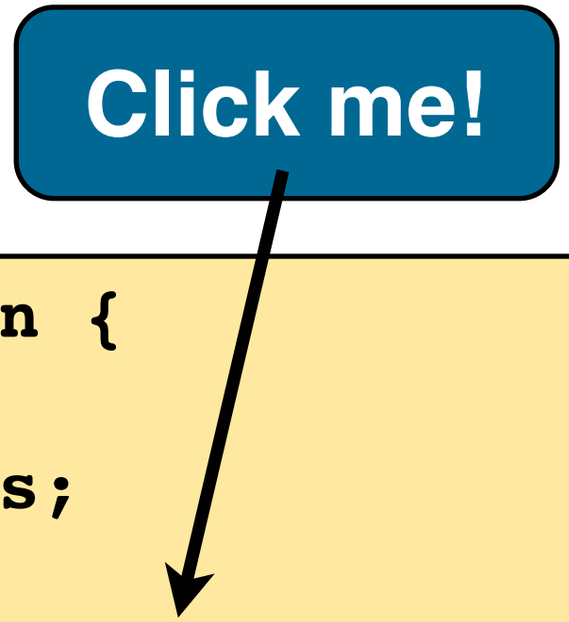
the button's label

a.getWhen( )
Returns the timestamp of when this event occurred.

a.getSource( )
Returns the object on which the Event initially occurred.

the button itself

Inside a Button




```
public class JButton extends AbstractButton {

private ArrayList<ActionListener> listeners;

protected void fireActionPerformed(ActionEvent event) {
  for(ActionListener al: listeners) {
     al.actionPerformed(event);
  }

protected addActionListener(ActionListener L) {
  listeners.add(L);
}
```

Button Events

- Something must implement ActionListener
- One option: have the JFrame do it

```
class NewFrame extends JFrame implements ActionListener {
  public NewFrame (int width, int height)
  {
   // ...
   button.addActionListener(this);
   // ...
  }
  public void actionPerformed (ActionEvent a)
  {
    System.out.println ("ActionPerformed!");
  }
```

Why **this**?

Simple Button

- In **`guis.buttons.ButtonAction`**
- Implement the **`ActionListener`** interface
- Define an **`actionPerformed()`** function
- Use **`b.addActionListener`** to set the object that will handle the events

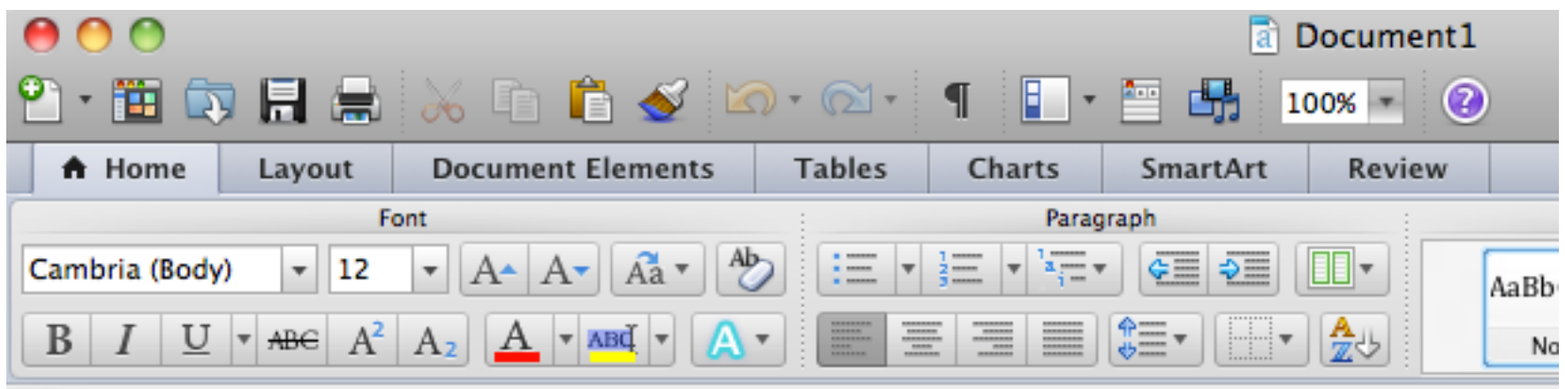
- Print a message when the button is clicked and make it exit the program with System.exit()

Multiple Buttons

- In **`guis.buttons.ButtonTest`**
  - Make the hello and bye buttons change the text of the msg label.
- Two approaches:
  - 1: Create different event handler classes for each Button
  - 2: Look at the event parameter to distinguish the source of an event within a single event handler class
- Let's use both!
  - Already have a **QuitActionListener** class for quit
  - Make **NewFrame2** implement ActionListener for the other buttons
  - How can we differentiate between the two buttons?
  - Inspect the **ActionEvent** parameter you are passed
    - **a.getActionCommand**() returns the clicked button's text label
    - **a.getSource**() returns a reference to the object that started the event (i.e., the JButton instance that was clicked)

…that can get messy

- when we have lots buttons!




```
public void actionPerformed (ActionEvent a)
{
  // Get the button string.
  String s = a.getActionCommand();

  if (s.equalsIgnoreCase ("Bold")) {
    // ...
  }
  else if (s.equalsIgnoreCase ("Italic")) {
    // ...
  }
  else if (s.equalsIgnoreCase ("Right Justify")) {
    // ...
  }
}
```

What else can we do?

- Having one giant event handler function is messy
  - Need to be careful that a change to one button won't break code for another

- Implementing lots of classes is also undesirable
  - What is a key limitation of using separate classes?
  - What can you do in NewFrame2 but not in a separate class?

- We want to use OOP principles!
  - Compartmentalize functionality
  - Reuse code instead of copy/pasting
  - Isolate and protect data

Options…

- We could create custom classes just for handling the events

```
class QuitButtonHandler implements ActionListener {
  public void actionPerformed (ActionEvent a)
  {
    System.out.println ("ActionPerformed!");
  }
}
```

```
class NewFrame extends JFrame {
  public NewFrame ()
  {
   // ...
   quitButton.addActionListener(new QuitButtonHandler);
   randButton.addActionListener(new RandomButtonHandler);
  }
```

Problem???

- We could create custom classes just for handling the events

```
class QuitButtonHandler implements ActionListener {
  public void actionPerformed (ActionEvent a)
  {
    System.out.println ("ActionPerformed!");
  }
}
```

What if the event handler needs access to data from NewFrame?

```
class NewFrame extends JFrame {
  public NewFrame ()
  {
   // ...
   quitButton.addActionListener(new QuitButtonHandler);
   randButton.addActionListener(new RandomButtonHandler);
  }
```

What we really want:

- To isolate the event handler for each object

- To allow the event handlers to access the data of the class they are in

What we really want:

- To isolate the event handler for each object
  - but a single class can only implement the functions in an interface once!

- To allow the event handlers to access the data of the class they are in
  - but if we use separate classes for each event handler we won't be able to do this!

- Oh noes!
  - :(

Inner Classes

- Java to the rescue!
- Use an **Inner Class**

```
class myClass {
  class myInnerClass {
    void someFunc() {
    }
  }

}
```

Inner Classes

- Use an **Inner Class**

```
class myPanel {
   private JLabel myLabel;

   class eHandler1 implements ActionListener {
      myLabel.setText("Handler 1!");
   }

   class eHandler2 implements ActionListener {
   }

   class eHandler3 implements ActionListener {
   }
}
```

What can it do?

- Can an inner class touch its outer's privates?
  - **Yes** it can!
- Can an "outer" class call functions in the inner?
  - **Yes** it can!
- Can an inner class implement an interface?
  - **Yes** it can!
- Can an inner class extend another class?
  - **Yes** it can!
- Can an inner class access local variables in outside functions?
  - **No** it can't!

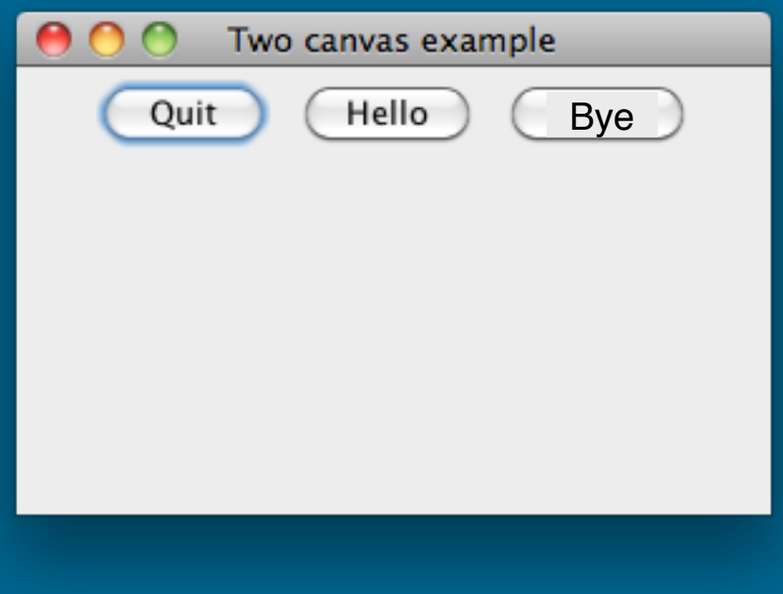
Sample Code

- Check out **guis.inner.InnerTest.java**
- Note:
  - The inner class can have: functions, data, and constructors
  - The inner class can access private data of its outer class
  - The outer class can access private data of the inner class

- **Todo:**
  - **Make the outer class print out the values of X and Y**

Inner Class Event Handlers

- Look at **guis.inner.InnerEvents.java**



- We want to have:
  - **Quit**: quits
  - **Hello**: display "hello"
  - **World**: prints "bye"
  - (in the msg JLabel)
- The quit button currently uses an inner class
- **Your turn:**
  - Add two new inner classes for Hello and Bye
- **Elite Hacker:**
  - Combine your two inner classes into a single inner class

Types of Classes in Java

- A public/private class
  - Must have name equal to file
- A class with no privacy modifier
  - Only usable within that package
- An inner class inside of another class
  - Inner can access the outer and vice versa
- An anonymous inner class
  - Declared as part of a function call

```
quitB.addActionListener (
  new ActionListener() {
    public void actionPerformed (ActionEvent a)
    { System.exit (0); }
  }
);
```

… and a few other types too!

Event Listeners