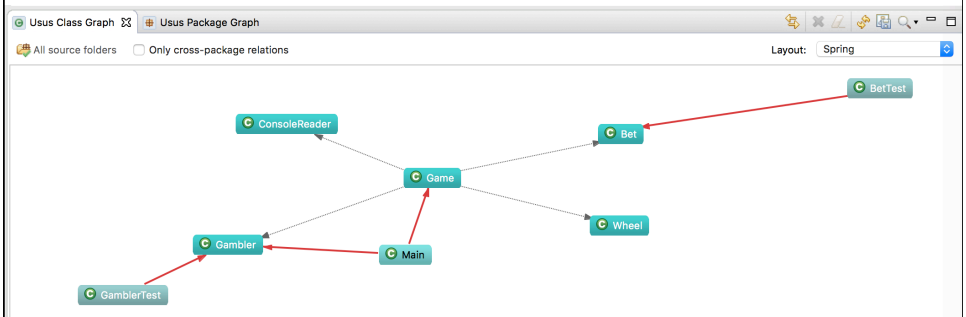
Objectives

- Programming Paradigms
- Introduction to GUIs in Java
  - Event handling

One More Plugin: Project Usus

- https://github.com/usus/usus-plugins/wiki
- Says has metrics capabilities, but they weren't working for me.
- Makes nice visual representations of a project's classes:



PROGRAMMING PARADIGMS

Programming Paradigms

- Our focus has been Object-oriented and Procedural paradigms
- Other paradigms
  - Event-driven
    - GUIs, Web applications
  - Distributed
    - Web applications, Grid, Cloud
  - Concurrent
  - Parallel
  - Aspect-oriented

Blurred lines between paradigms, Not completely independent

GUIS IN JAVA

Java GUI Libraries: AWT & Swing

- AWT: Abstract Windowing Toolkit
  - Original GUI toolkit
  - Relies on operating system to render GUIs
    - Benefit: Match look and feel of platform
  - Classes in `java.awt.*`
- Swing: added to Java2
  - Classes in `javax.swing.*`
  - Extends AWT
  - Provides *Java* look and feel for applications
    - But can plug in other look & feels

Swing & AWT

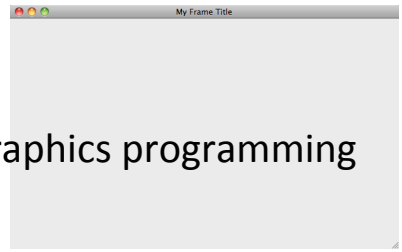
- Swing does not completely replace AWT
- Using the Swing graphics programming model
  - Improves performance
  - Allows more efficient development of GUIs
- We will use Swing mostly
  - Leverage AWT

Swing: Made up of Components

- Top-level components
  - ~Hold GUI elements
  - Examples: `JFrame`, `JWindow`, `JDialog`, `JApplet`
- GUI Elements
  - ~Things user interacts with
  - Examples: `JButton`, `JLabel`, `JMenuBar`

JFRAMES AND PARENT CLASSES

Frames



- **Frame**: Most basic unit of graphics programming
- Example of a *container*
  - A ***container*** contains other UI components
- A top-level window
  - Not contained within another window
- Swing's `JFrame` class implements a frame

Example Frame

```
public class Game extends JFrame implements
    KeyListener {

    public static void main(String[] args) {
        Game session = new Game();
        session.init();
    }

    public void init() {
        // Top-left corner is (0,0)
        // width/height: XBOUND, YBOUND
        setBounds(0, 0, XBOUND, YBOUND);
        // Shows the window
        setVisible(true);
        …
    }
}
```

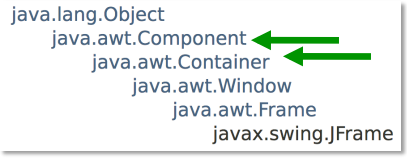

Frame Inheritance

- Class hierarchy

```
java.lang.Object
    java.awt.Component
        java.awt.Container
            java.awt.Window
                java.awt.Frame
                    javax.swing.JFrame
```

- `JFrame` is derived from `java.awt.Frame`
  - `Frame` class is derived from `Container` class
    - Container: anything that can contain UI components
  - Lots of methods available from the hierarchy

Components & Containers



- Component
  - ***Abstract*** class
  - Everything you *see* is a component
    - All nonmenu-related AWT components
  - Many methods
    - Some deprecated: be careful
- Container
  - *Concrete* implementation of Component
  - Base class of many classes

Container Methods

- add(Component c)
- setSize
  - Sets size of frame in *pixels*
- setLocation
  - Sets location of frame
    - Coordinates of top-left corner
- setBounds
  - Sets both size and location of frame
    - Provides information needed for setSize and setLocation

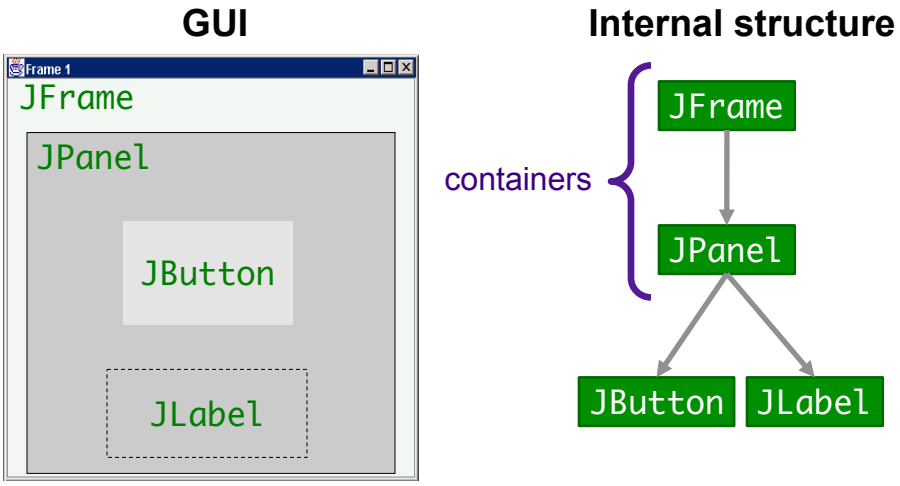
Window Methods

- Top-level window
- No borders
- No Menu Bar
- `dispose()`
  - Closes window and reclaims resources associated with it
- `toBack()`
  - Sends window to back, may lose focus/activation
- `toFront()`
  - Bring to front, make this the focused window

Frame's Methods

- Top-level window *with title and borders*
- `setTitle(String title)`
  - Sets title of frame (displayed in title bar)
- `setResizable(boolean resizable)`
  - Can the user resize the frame?

Anatomy of an Application GUI



Implementing a GUI Component

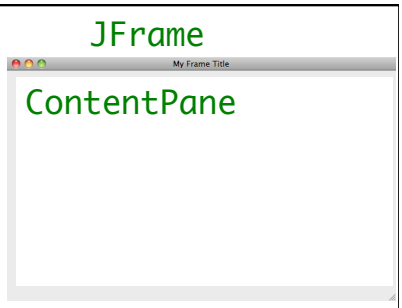
1. Create it
2. Configure it
3. Add children (if container)
4. Add to parent (if not JFrame)
5. Listen to it

Implementing a GUI Component

1. Create it
```
JButton b = new JButton();
```
2. Configure it
```
b.setText("press me");
b.setForeground(Color.blue);
```
3. Add it to parent
```
panel.add(b);
```
4. Listen to it
   - Events: Listeners

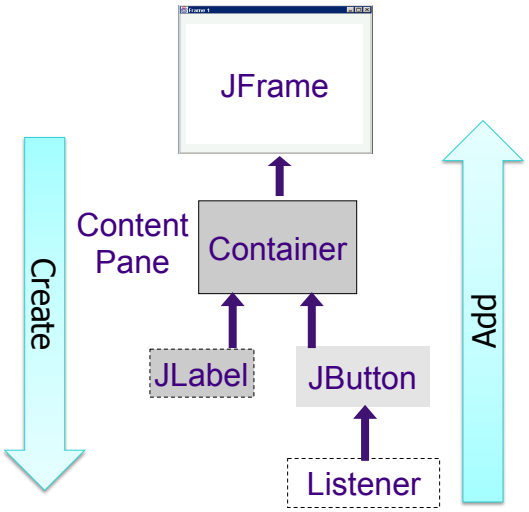
JFrame

- Contains ContentPane
  - A Container object that holds components you add, placing them in the frame
  - The part of the frame that holds UI components



Building a GUI

1. Create (top down):
   - Frame
   - Container
   - Components
   - Listeners
2. Add (bottom up):
   - Listeners into components
   - Components into panel
   - Panel into frame



Example Code

```
// create the components
JFrame f = new JFrame("title");
f.setBounds(0, 0, 100, 100);
Container pane = f.getContentPane();
JButton b = new JButton("press me");

// add button to panel
pane.add(b);

// show the frame
f.setVisible(true);
```

More GUI Components

- Choice
  - Drop-down list
- FileDialog
  - Opening and saving files
- List
  - Scrollable
  - Allows multiple selections
- ScrollPane
  - scrollbars
- TextField
  - Single line of text
- TextArea
  - Multiple lines of text

Menus

- MenuBar
  - Thing across top of frame
  - `Frame: setMenuBar(MenuBar mb);`
- Menu
  - The dropdown part
  - A sequence of `MenuItems`
  - `Menu` is a subclass of `MenuItems`, so can have submenus

Practice: Combining Components

- Create a panel with three buttons on it

`ButtonPanel.java`

Placement of Components

- How does the panel know where to place a button?
- How does the panel know where to place the next button?
- How does the panel know where to place *any* component that is added to it?

LAYOUT MANAGERS

Layout Managers

- Java uses ***layout managers*** to place components inside a container
- LayoutManager automatically handles placement of components
  - When a component is added to a container (through add), layout manager decides where to place the component

Default Layout Managers

- `JFrame's` content pane: `BorderLayout`
- `JPanel's`: `FlowLayout`

The Flow Layout Manager

- Default layout manager for a *panel*
- Lines components up *horizontally* until no more room in container
  - Then starts a new row of components
- If user resizes component, layout manager automatically reflows components

The Flow Layout Manager

- Can choose how to arrange components in a row
  - Default: center each row
  - Other options: left or right align
- Change alignment using `setLayout`

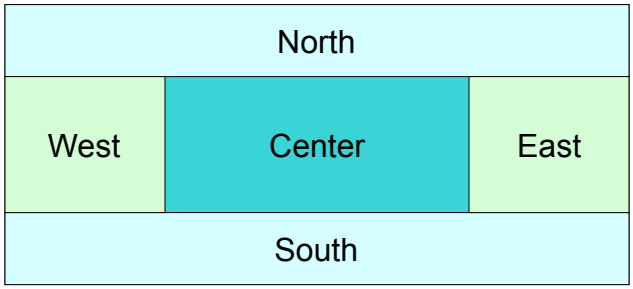
```
setLayout(new FlowLayout( FlowLayout.LEFT ));
```

components aligned to the left

- Another constructor has `hgap` and `vgap` for gaps to put around components

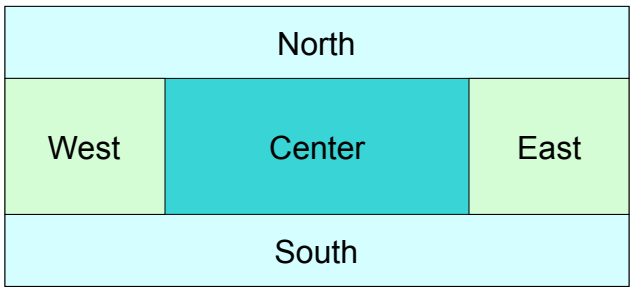
Border Layout Manager

- Default layout manager of the content pane for `JFrame`
- Lets you choose where you want to place each component



with respect to the container

Border Layout Regions



- Edge components are laid out first
- Center occupies remaining space

Border Layout Rules

- Grows all components to fill available space
- If container is resized, edge components are redrawn and center region size recomputed
- To add a component to a container using a border layout
  - Ex: JFrame's content pane

```
Container contentPane = getContentPane();
contentPane.add(button, BorderLayout.SOUTH);
```

Adding Components Using a Border Layout

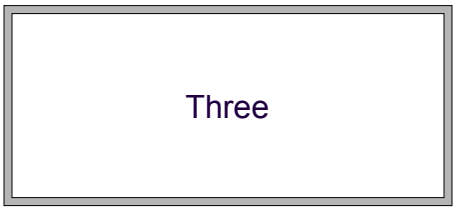
```
Container contentPane = getContentPane();
contentPane.add(button, BorderLayout.SOUTH);
```

- If no region specified, assumes center region

> What happens if we add multiple components, e.g., three buttons, without specifying a region?

  - Recall: border layout grows component to fit specified region

A Border Layout Limitation



- Last button added grows to completely fill center region
- First two buttons were discarded/overwritten by each subsequently added component

Changing Layout Managers

- Any container can use any layout manager
- Use `setLayout` to change layout manager *before adding components*

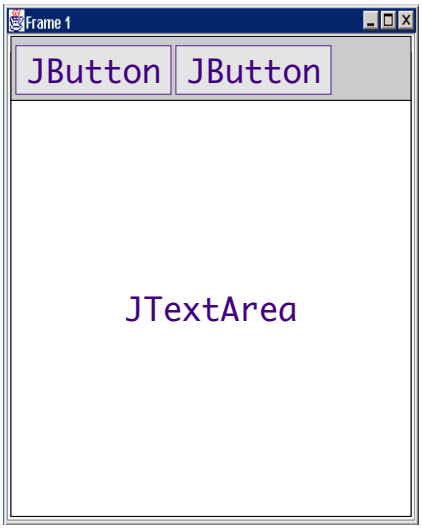
```
// sets layout to a new flow layout manager that
// aligns row components to the left and uses a 20 pixel
// horizontal separation and 20 pixel vertical separation
setLayout(new FlowLayout(FlowLayout.LEFT, 20, 20));

// sets layout to a new border layout manager that
// uses a 45 pixel horizontal separation between
// components (regions) and a 20 pixel vertical separation
setLayout(new BorderLayout(45, 20));
```

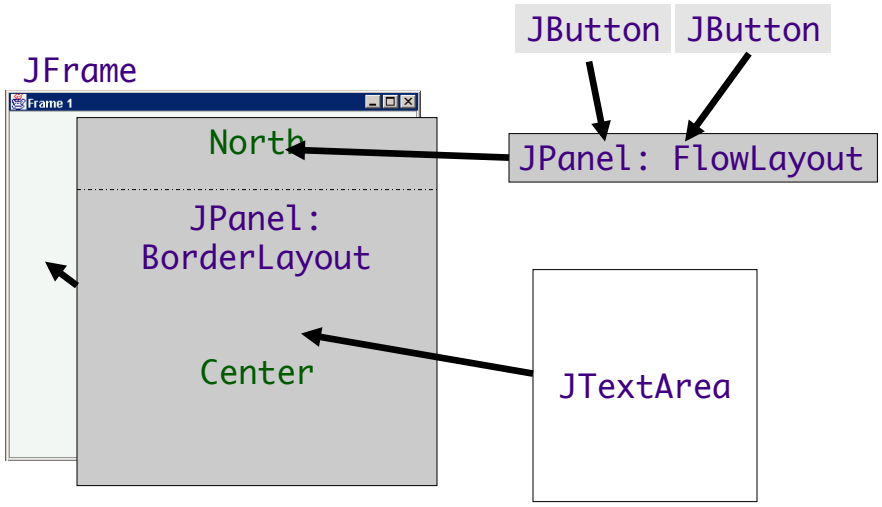

Combining Panels

- **Panels** act as (smaller) containers for UI elements
- Can be arranged inside a larger panel by a layout manager
- Use additional panels to customize look
  - Create a panel
  - Add some buttons to it
  - Add that panel to a region in content pane

Combining Panels



Combining Panels



Using Additional Panels

- Get fairly accurate and precise placement of components
- Use nested panels with

Layout	Use
BorderLayout	Content panes and enclosing panels
Flow Layouts	Panels containing buttons and other UI components

FlexibleLayout.java

Another Layout Manager: Grid

- Divides container into columns and rows of equal size, which collectively occupy the entire container region
- Rows and columns are aligned like a table
  - When container is resized, the "cells" grow and/or shrink
  - Cells always maintain identical sizes
- Example:

```
panel.setLayout(new GridLayout(5, 4)); // 5 rows, 4 cols
```

Adding Components to a Grid Layout

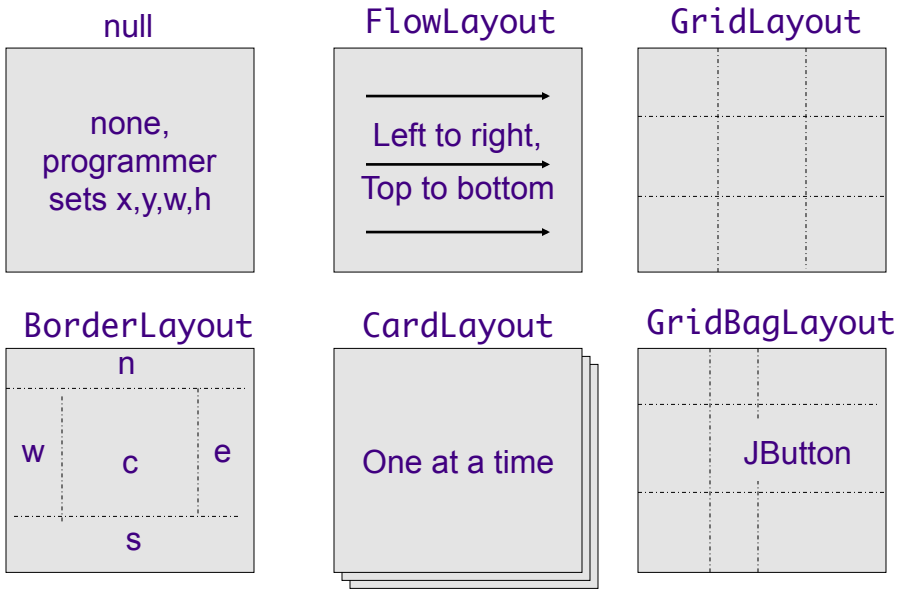
- Components added *sequentially*
- 1st add adds component to 1st row, 1st col
- 2nd add adds component to 1st row, 2nd col

- And so forth until 1st row is filled
- Then 2nd row begins with the 1st column
- Continues until the entire container is filled

Grid Layout Rules

- Components are resized to take up entire cell
- Restrictive but can be useful for some applications
- Example: Create a row of buttons of identical size
  1. Make a panel that has a grid layout with one row
  2. Add a button to each cell
  3. Set horiz/vert separation so buttons are not touching

Layout Manager Heuristics



HANDLING USER INTERACTIONS

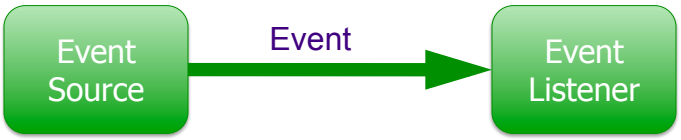
Event-Driven Programming

- User actions (e.g., mouse clicks, key presses), sensor outputs, or messages from other applications determine flow of program

- Application architecture:

```
while ( true ) {
      event = waitForEvent();
      handleEvent(event);
}
```


Event Basics



- An ***event*** is generated from an ***event source*** and is transmitted to an ***event listener***
- Event sources allow event listeners to ***register*** with them
  - Registered listener requests event source send its event to listener when event occurs

Java Event Handling

- All events are objects of event classes
  - Derive from `java.util.EventObject`
- **Event source**
  - Sends out event objects to *all registered* listeners when that event occurs
- **Listener**
  - Implements a listener interface
  - Uses `EventObject` to determine its reaction to the event

Java Event Handling

- Register a listener with an event source:

```
eventSourceObject.addEventListener(
      eventListenerObject);
```

- Example:

```
ActionListener listener = . . .;
JButton button = new Jbutton("Click Me!");
button.addActionListener(listener);
```

  - Whenever an "action event" occurs on `button`, `listener` is notified
    - For buttons, an action event is a button click

Listener Objects

- A listener object must be an instance of a class that implements the appropriate interface
  - For buttons, that's `ActionListener`
- Listener class must implement `actionPerformed(ActionEvent event)`

Listener Objects and Event Handling

- When a user clicks a button, `JButton` object generates an `ActionEvent` object

Which makes `JButton` a *what*?

- `JButton` calls listener object's `actionPerformed` method, passing generated event object
- A single event source can have *multiple listeners* listening for its events
  - Source calls `actionPerformed` on each of its listeners

An Example of Event Handling

- Suppose we want to make a panel that has three buttons on it
  - Each button has a color associated with it
  - When user clicks a button, background color of panel changes to the corresponding color
- We need
  1. A panel with 3 buttons on it
  2. 3 listener objects, one registered to listen for a button's events