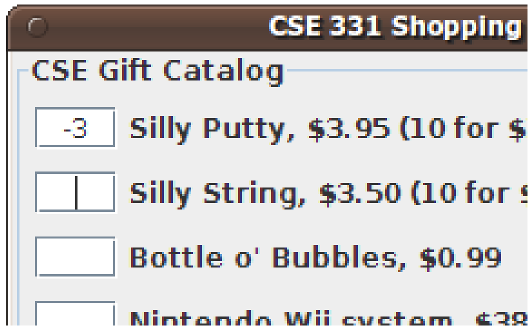
Java Graphics & GUIs (and Swing/AWT libraries)

CSE 331
Software Design & Implementation

Slides contain contributions from: M. Ernst, M. Hotan, R. Mercer, D. Notkin, H. Perkins, S. Regis, M. Stepp; Oracle docs & tutorial, Horstmann, Wikipedia, …

Why study GUIs?

- Learn about *event-driven programming* techniques
- Practice learning and using a large, complex API
- A chance to see how it is designed and learn from it:
  - design patterns: model-view separation, callbacks, listeners, inheritance vs. delegation
  - refactoring vs. reimplementing an ailing API
- Because GUIs are neat!



-3 Silly Putty, $3.95 (10 for $
Silly String, $3.50 (10 for
Bottle o' Bubbles, $0.99
Nintendo Wii system, $38

- *Caution*: There is way more here than you can memorize.
  - Part of learning a large API is "letting go."
  - First, learn the fundamental concepts and general ideas.
  - Then, look things up as you need them
  - Don’t get bogged down implementing eye candy

References

Today: Java graphics and Swing/AWT class libraries

Only an introduction! Also see

- Sun/Oracle Java tutorials
  http://docs.oracle.com/javase/tutorial/uiswing/index.html
- Extra slides, on class website
- *Core Java* vol. I by Horstmann & Cornell
- If you have another favorite, use it

Next lecture:

Event-driven programming and user interaction

Outline

Organization of the Swing/AWT library

Graphics and drawing

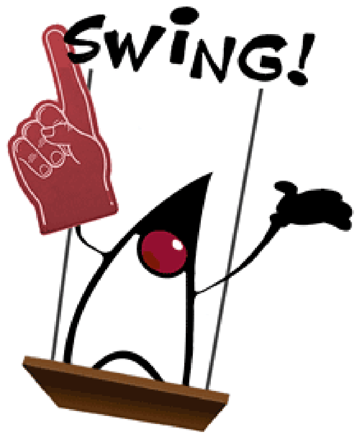
- Repaint callbacks, layout managers, etc.

Handling user events

Building GUI applications

- MVC, user events, updates, &c

Java GUI libraries



- **Swing**: the main Java GUI library
  - *Benefits:* Features; cross-platform compatibility; OO design
  - Paints GUI controls itself pixel-by-pixel
    - Does not delegate to OS's window system

- **Abstract Windowing Toolkit** (**AWT**): Sun's initial GUI library
  - Maps Java code to each operating system's real GUI system
  - *Problems:* Limited to lowest common denominator (limited set of UI widgets); clunky to use.

- Advice: Use Swing. You occasionally have to use AWT (Swing is built on top of AWT). Beware: it's easy to get them mixed up.

GUI terminology

window: A first-class citizen of the graphical desktop

Also called a *top-level container*

Examples: frame, dialog box, applet

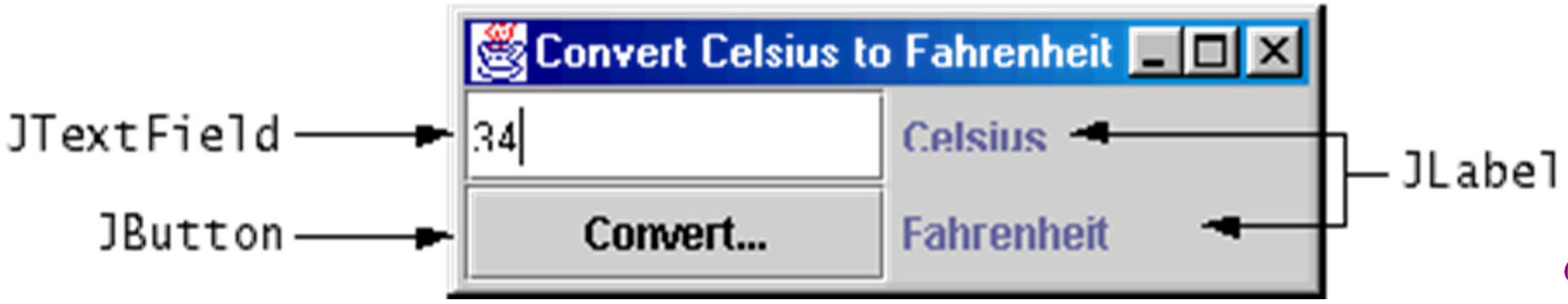
component: A GUI widget that resides in a window

Also called *controls* in many other languages

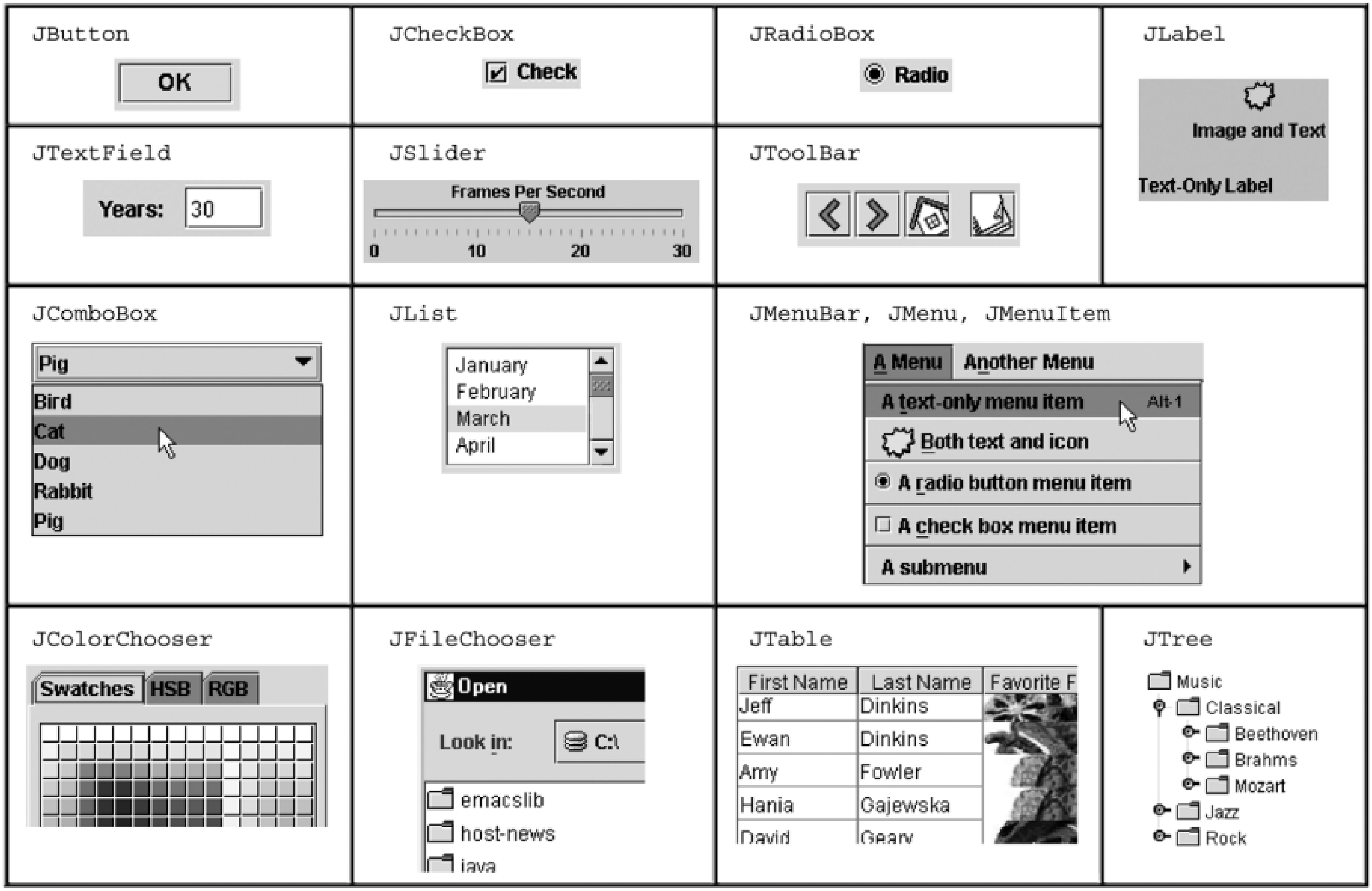
Examples: button, text box, label

container: A component that hosts (holds) components

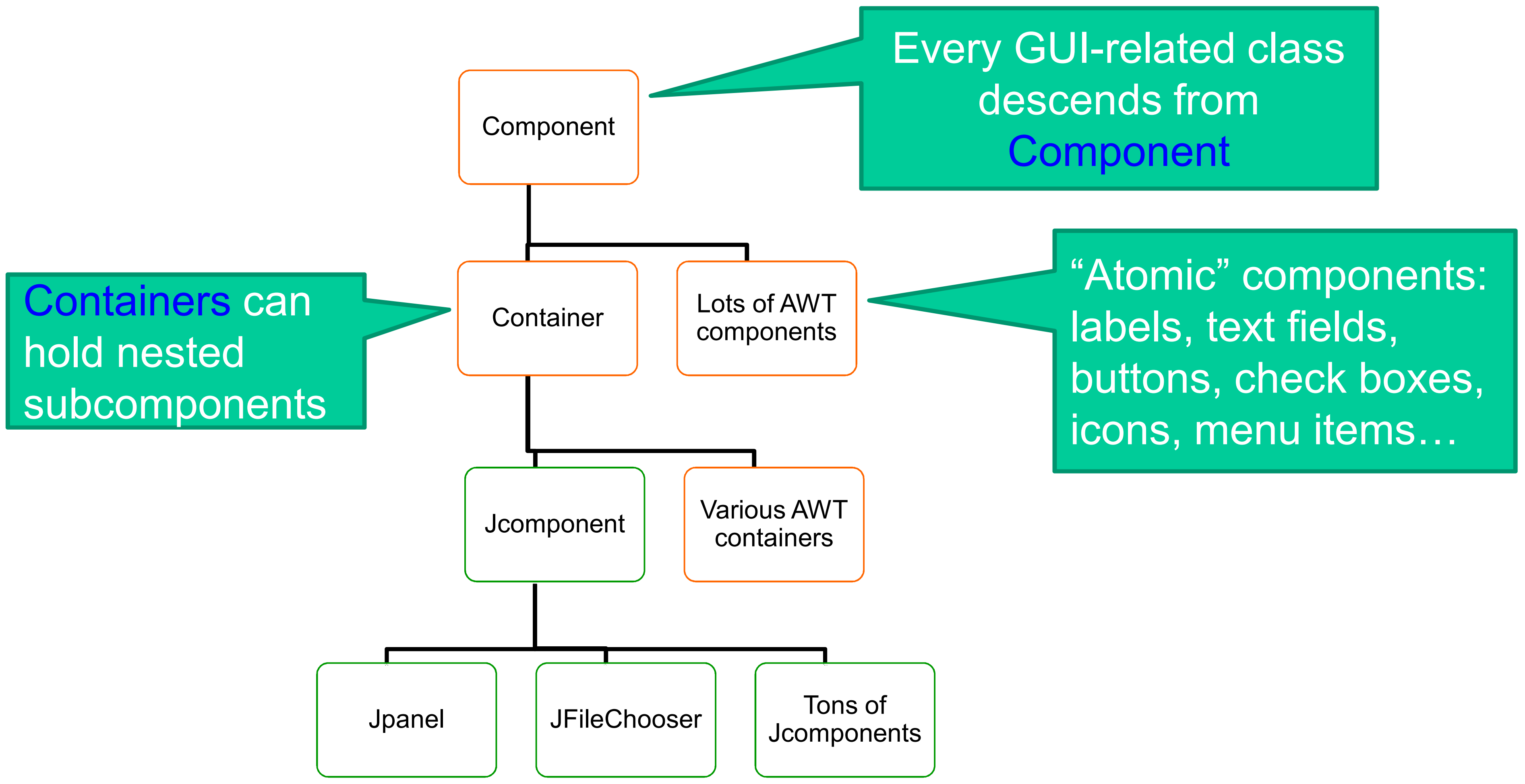
Examples: panel, box



Components


JButton
OK
JCheckBox
Check
JRadioBox
Radio
JLabel
Image and Text
Text-Only Label
JTextField
Years: 30
JSlider
Frames Per Second
0 10 20 30
JToolBar
JComboBox
Pig
Bird
Cat
Dog
Rabbit
Pig
JList
January
February
March
April
JMenuBar, JMenu, JMenuItem
A Menu
Another Menu
A text-only menu item Alt-1
Both text and icon
A radio button menu item
A check box menu item
A submenu
JColorChooser
Swatches
HSB
RGB
JFileChooser
Open
Look in: C:\
emacslib
host-news
java
JTable
First Name
Last Name
Favorite F
Jeff Dinkins
Ewan Dinkins
Amy Fowler
Hania Gajewska
David Geary
JTree
Music
Classical
Beethoven
Brahms
Mozart
Jazz
Rock

Component & container classes


Component
Container
Lots of AWT components
Jcomponent
Various AWT containers
Jpanel
JFileChooser
Tons of Jcomponents
Every GUI-related class descends from Component
Containers can hold nested subcomponents
“Atomic” components: labels, text fields, buttons, check boxes, icons, menu items…

Swing/AWT inheritance hierarchy

```
Component   (AWT)
   Window
      Frame
         JFrame   (Swing)
         JDialog

   Container
      Jcomponent  (Swing)
         JButton        JColorChooser    JFileChooser
         JComboBox      JLabel           JList
         JMenuBar       JOptionPane      JPanel
         JPopupMenu     JProgressBar     JScrollbar
         JScrollPane    JSlider          JSpinner
         JSplitPane     JTabbedPane      JTable
         JToolbar       JTree            JTextArea
         JTextField     ...
```

Component fields (actually properties)

Each has a **`get`** (or **`is`**) accessor and a **`set`** modifier.
Examples: **`getColor, setFont, isVisible`**, …

name	description
background	background color behind component
border	border line around component
enabled	whether it can be interacted with
focusable	whether key text can be typed on it
font	font used for text in component
foreground	foreground color of component
height, width	component's current size in pixels
visible	whether component can be seen
tooltip text	text shown when hovering mouse
size, minimum / maximum / preferred size	various sizes, size limits, or desired sizes that the component may take

Types of containers

- Top-level containers: JFrame, JDialog, …
  - Often correspond to OS windows
  - Can be used by themselves, but usually as a host for other components
  - Live at top of UI hierarchy, not nested in anything else
- Mid-level containers: panels, scroll panes, tool bars
  - Sometimes contain other containers, sometimes not
  - JPanel is a general-purpose component for drawing or hosting other UI elements (buttons, etc.)
- Specialized containers: menus, list boxes, …
- Technically, all J-components are containers

JFrame – top-level window

Graphical window on the screen

Typically holds (hosts) other components

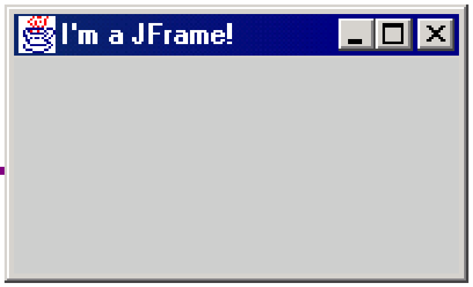
Common methods:

- **`JFrame(String`** *title***`)`** – constructor, title optional
- **`setSize(int`** *width***`, int`** *height***`)`** – set size
- **`add(Component`** *c***`)`** – add component to window
- **`setVisible(boolean`** *v***`)`** – make window visible or not. Don’t forget this!

Example

SimpleFrameMain.java

More JFrame



- `public void setDefaultCloseOperation(int op)`
  Makes the frame perform the given action when it closes.
  - Common value passed: `JFrame.EXIT_ON_CLOSE`
  - If not set, the program will never exit even if the frame is closed.

- `public void setSize(int width, int height)`
  Gives the frame a fixed size in pixels.

- `public void pack()`
  Resizes the frame to fit the components inside it snugly.

JPanel – a general-purpose container

Commonly used as a place for graphics, or to hold a collection of button, labels, etc.

Needs to be added to a window or other container

```
frame.add(new Jpanel(…))
```

`JPanels` can be nested to any depth

Many methods/fields in common with **`JFrame`** (since both inherit from Component)

- Advice: can't find a method/field? Check the superclass(es)

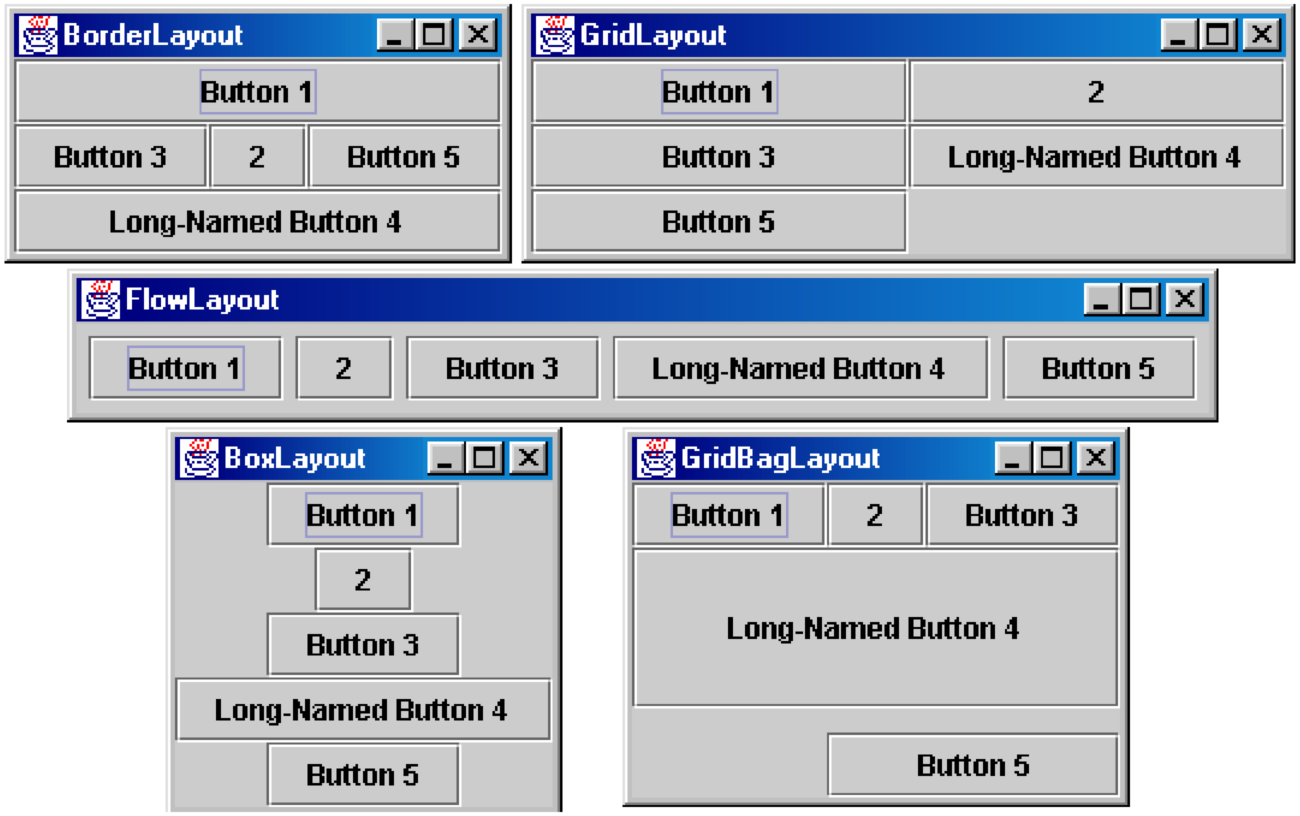
Some new methods. Particularly useful:

`setPreferredSize(Dimension` *d***`)`**

Containers and layout

What if we add several components to a container?
How are they positioned relative to each other?

Answer: each container has a layout manger.



Layout managers

Kinds:

- **`FlowLayout`** (left to right, top to bottom) – default for **`JPanel`**
- **`BorderLayout`** (“center”, “north”, “south”, “east”, “west”) – default for **`JFrame`**
- **`GridLayout`** (regular 2-D grid)
- others... (some are incredibly complex)

The first two should be good enough for now….

Place components in a *container*; add the container to a frame.

- **container**: An object that stores components and governs their positions, sizes, and resizing behavior.

pack()

Once all the components are added to their containers, do this to make the window visible

```
pack();
setVisible(true);
```

`pack()` figures out the sizes of all components and calls the layout manager to set locations in the container (recursively as needed)

If your window doesn’t look right, you may have forgotten **`pack()`**

Example

SimpleLayoutMain.java

Sizing and positioning

How does the programmer specify where each component appears, how big each component should be, and what the component should do if the window is resized / moved / maximized / etc.?

- **Absolute positioning** (C++, C#, others):
  Programmer specifies exact pixel coordinates of every component.
  - "Put this button at (x=15, y=75) and make it 70x31 px in size."
- **Layout managers** (Java):
  Objects that decide where to position each component based on some general rules or criteria.
  - "Put these four buttons into a 2x2 grid and put these text boxes in a horizontal flow in the south part of the frame."

JFrame as container

A `JFrame` is a container. Containers have these methods:

- `public void` **`add`**`(Component comp)`
  `public void` **`add`**`(Component comp, Object info)`
  Adds a component to the container, possibly giving extra information about where to place it.

- `public void` **`remove`**`(Component comp)`

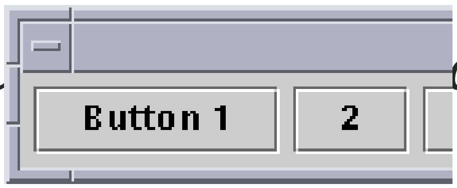
- `public void` **`setLayout`**`(LayoutManager mgr)`
  Uses the given layout manager to position components.

- `public void` **`validate`**`()`
  Refreshes the layout (if it changes after the container is onscreen).

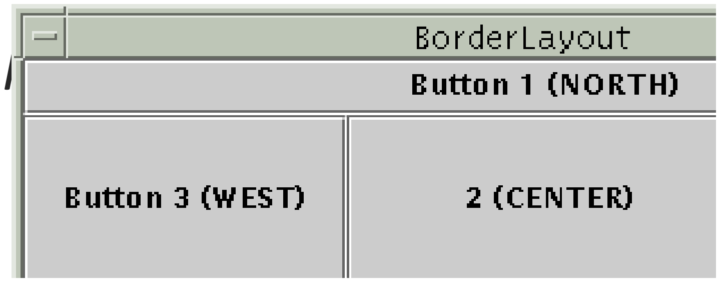
Preferred sizes

- Swing component objects each have a certain size they would "like" to be: Just large enough to fit their contents (text, icons, etc.).
  - This is called the *preferred size* of the component.

  - Some types of layout managers (e.g. `FlowLayout`) choose to size the components inside them to the preferred size.
  - Others (e.g. `BorderLayout`, `GridLayout`) disregard the preferred size and use some other scheme to size the components.

Bu 

d size:

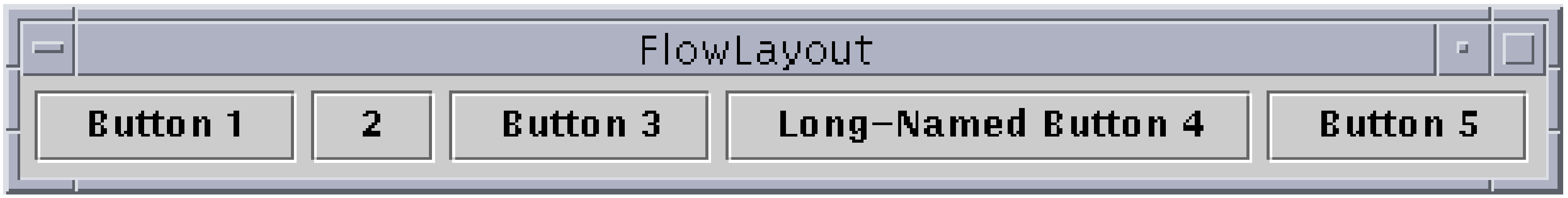
I 

FlowLayout

```
public FlowLayout()
```

- treats container as a left-to-right, top-to-bottom "paragraph".
  - Components are given preferred size, horizontally and vertically.
  - Components are positioned in the order added.
  - If too long, components wrap around to the next line.

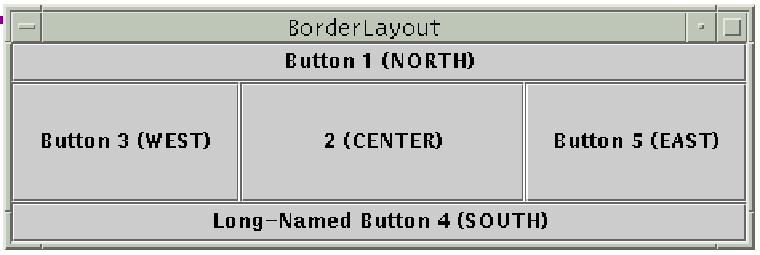
```
myFrame.setLayout(new FlowLayout());
myFrame.add(new JButton("Button 1"));
```



  - The default layout for containers other than `JFrame` (seen later).

BorderLayout

`public BorderLayout()`



- Divides container into five regions:
  - `NORTH` and `SOUTH` regions expand to fill region horizontally, and use the component's preferred size vertically.
  - `WEST` and `EAST` regions expand to fill region vertically, and use the component's preferred size horizontally.
  - `CENTER` uses all space not occupied by others.

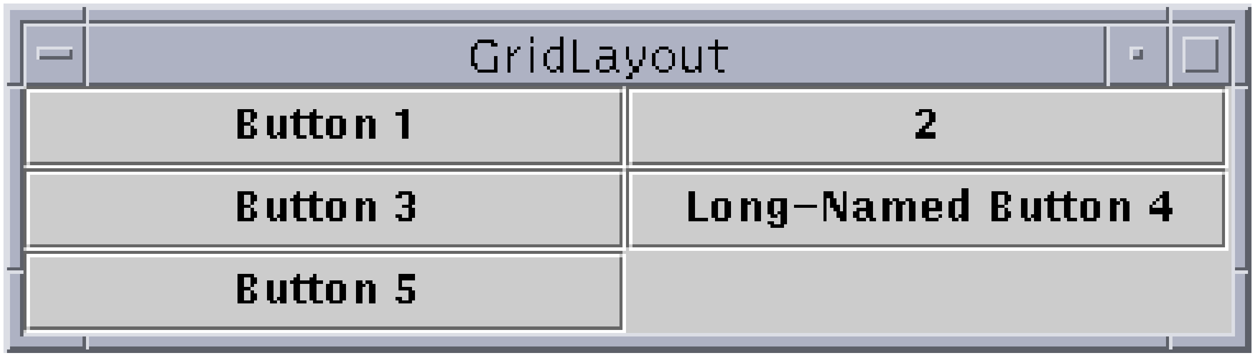
```
myFrame.setLayout(new BorderLayout());
myFrame.add(new JButton("Button 1"), BorderLayout.NORTH);
```

- This is the default layout for a `JFrame`.

GridLayout

```
public GridLayout(int rows, int columns)
```

- Treats container as a grid of equally-sized rows and columns.
- Components are given equal horizontal / vertical size, disregarding preferred size.
- Can specify 0 rows or columns to indicate expansion in that direction as needed.



Graphics and drawing

So far so good – and very boring…

What if we want to actually draw something? A map, an image, a path, …?

Answer: Override method **`paintComponent`**

- Method in **`JComponent`** that draws the component
- In **`JLabel`**'s case, it draws the label text

Example

SimplePaintMain.java

Graphics methods

Many methods to draw various lines, shapes, etc., …

Can also draw images (pictures, etc.). Load the image file into an **`Image`** object and use **`g.drawImage(`**…**`)`** :

- In the program (***not*** in **`paintComponent`**):

```
Image pic =
     Toolkit.getDefaultToolkit()
            .getImage(image path);
```

- Then in **`paintComponent`**:

```
g.drawImage(pic, ...);
```

Graphics vs Graphics2D

Class **`Graphics`** was part of the original Java AWT

  Has a procedural interface: **`g.drawRect(…)`**, **`g.fillOval(…)`**

Swing introduced **`Graphics2D`**

  Added a object interface – create instances of **`Shape`** like **`Line2D`**, **`Rectangle2D`**, etc., and add these to the **`Graphics2D`** object

Parameter to paintComponent is always **`Graphics2D`**. Can always cast it to that class. **`Graphics2D`** supports both sets of graphics methods.

  Use whichever you like for CSE 331

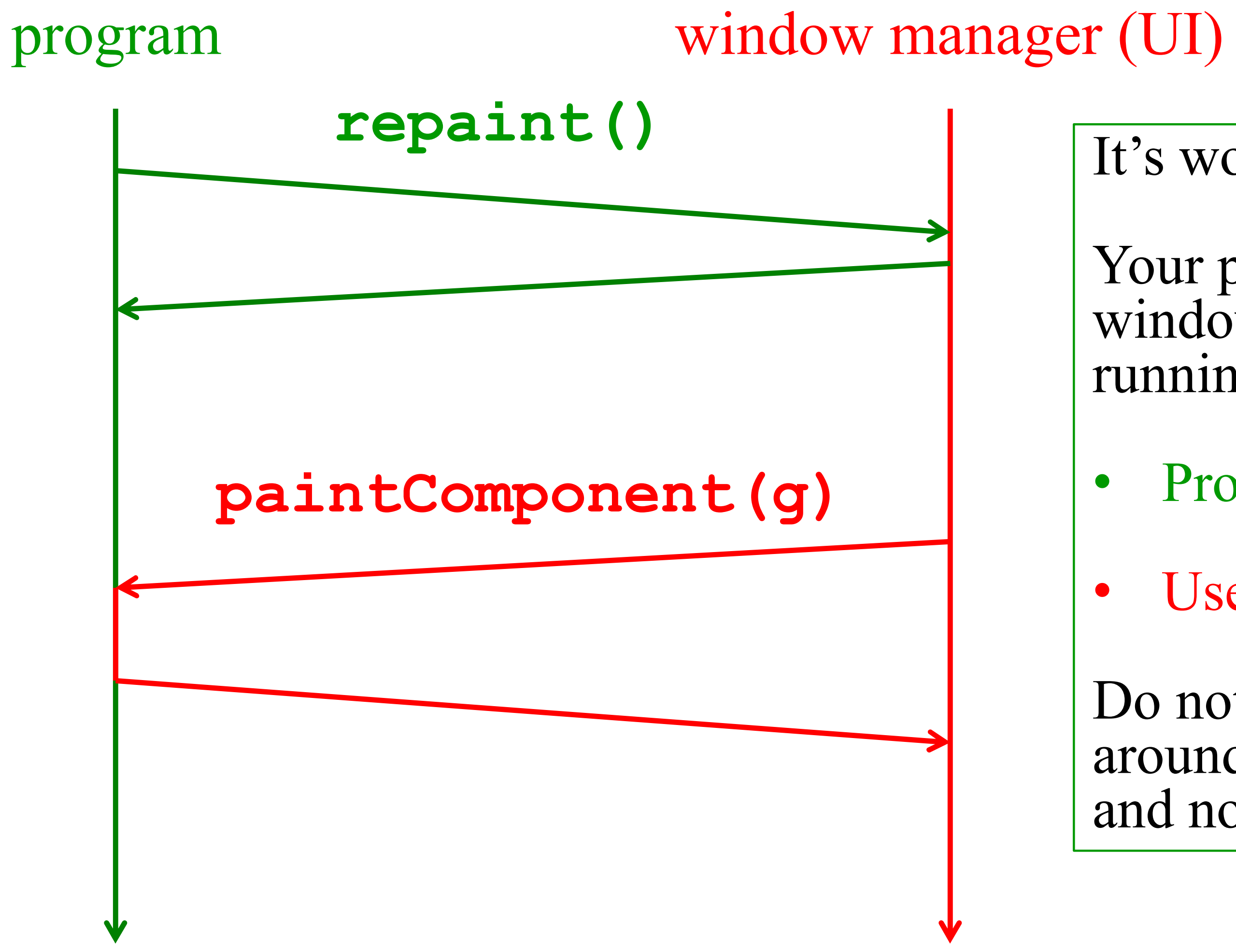
So who calls paintComponent? And when??

- Answer: the window manager calls **`paintComponent`** *whenever it wants!!!*
  - When the window is first made visible, and whenever after that it is needed
- Corollary: **`paintComponent`** must **always** be ready to repaint – regardless of what else is going on
  - You have no control over when or how often – must store enough information to repaint on demand
- If you want to redraw a window, call **`repaint()`** from the program (*not* from **`paintComponent`**)
  - Tells the window manager to schedule repainting
  - Window manager will call **`paintComponent`** when it decides to redraw (soon, but maybe not right away)

Example

FaceMain.java

How repainting happens



It's worse than it looks!

Your program and the window manager are running concurrently:

- Program thread
- User Interface thread

Do not attempt to mess around – follow the rules and nobody gets hurt!

Rules for painting – Obey!

- Always override **`paintComponent(g)`** if you want to draw on a component
- Always call **`super.paintComponent(g)`** first
- ***NEVER*** call **`paintComponent`** yourself. That means **ABSOLUTELY POSITIVELY NEVER!!!**
- Always paint the entire picture, from scratch
- Use **`paintComponent`**'s **`Graphics`** parameter to do all the drawing. ***ONLY*** use it for that. Don't copy it, try to replace it, permanently side-effect it, etc. It is quick to anger.
- **DON'T** create new **`Graphics`** or **`Graphics2D`** objects

- Fine print: Once you are a certified™ wizard, you may find reasons to do things differently, but you aren't there yet.

What’s next – and not

Major topic next time is how to handle user interactions

Key idea: the observer pattern

Beyond that you’re on your own to explore all the wonderful widgets in Swing/AWT. Have fun!!!

(But don’t sink huge amounts of time into eye candy)