Objectives

- Good enough design
- Introduction to GUIs in Java

Reflection on Assignment 10

- How did you make design decisions?
- Were there any particularly difficult design decisions?
  - What were the tradeoffs?
- Did anybody consider making a FileData or File class?

Assignment 10 Lessons

- Code should be soft
  - Eclipse makes code easier to change
    - The Refactor menu is a great resource
- Keep asking yourself
  - Is this understandable?
    - Will other people know what this code means?
      - Maintaining code and bug fixes are done much more than writing new code
  - How is this code most likely to change?
  - Does this code have a funny smell?
    - Literals, long methods, large classes, …

Refactoring Summary

- Write code and then *rewrite* code
  - Eye toward extensibility, flexibility, maintainability, and readability
  - Maintain correctness
- Reading/understanding other people's code can be difficult
  - Make your code readable, understandable
- Probably impossible to design/write "correctly" the first time
  - A lot harder to get the logic right, make sure you're not creating bugs, know/check the right answer…
  - Could cause yourself headaches coding this way first

Good-Enough Design Discussion

Perfect Design

- ✓ Follows all design principles
  - OCP, Single Responsibility, no code smells, …
- – May not be possible
  - Infinite refactoring, development
- – Code never released

Good-enough Design

- – Not everyone agrees on design
- – Maintenance requires changes to a few places
- ✓ Code gets released to customers

Similar tradeoffs in testing

PROGRAMMING PARADIGMS

Programming Paradigms

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

Blurred lines between paradigms, Not completely independent

GUIS IN JAVA

AWT & Swing

- AWT: Abstract Windowing Toolkit
  - Original GUI toolkit
  - Relies on operating system to render GUIs
    - 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
  - `JFrame, JWindow, JDialog, JApplet`
- GUI Elements
  - `JButton, JLabel, JMenuBar, …`

JFRAMES AND PARENT CLASSES

Frames

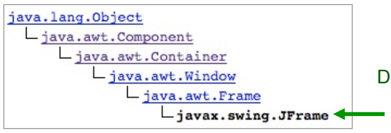
- **Frame**: Most basic unit of graphics programming
- Example of a *container*
  - A ***container*** contains other UI components
- A window that is not contained within another window
  - i.e., a top-level window
- `JFrame` Swing class implements a frame

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

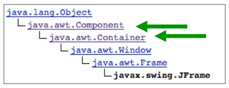
- `JFrame` is derived from `java.awt.Frame`
  - `Frame` class is derived from `Container` class
    - Container: anything that can contain UI components
- Class hierarchy



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

Yikes!
Don't get lost!

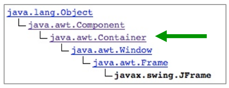
Components & Containers



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

- `Component`
  - *Abstract* class
  - Everything you *see* is a component
    - Superclass of `Container`
  - Many methods
    - Some deprecated: be careful
- `Container`
  - *Concrete* implementation of Component
  - Base class of many classes

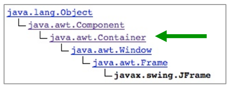
Container Methods



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

- `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()`

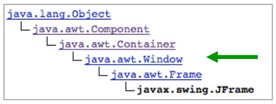
Container Methods



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

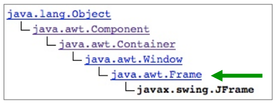
- `remove(Component c)`
- `getSize()`
  - Returns size of frame
- `getLocation()`
  - Returns current location of frame, relative to enclosing container
- `getLocationOnScreen()`
  - Returns current location of frame, using absolute screen coordinates

Window Methods



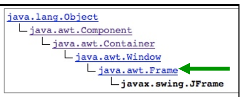
- Top-level window
- No borders
- No Menu Bar
- `dispose()`
  - Closes window and reclaims resources associated with it
- `toBack()`
  - Sends window to back, may lose focus/activation
- `toFront()`
  - Bring to front, make this the focused window

Frame’s Methods



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

Frame Methods



- `getExtendedState()`
- `setExtendedState(int state)`
- States (defined constants):
  - `NORMAL`
  - `ICONIFIED`
  - `MAXIMIZED_HORIZ`
  - `MAXIMIZED_VERT`
  - `MAXIMIZED_BOTH`

Screen Resolution

- Since screens have various resolutions, how do you determine how big to make a frame?
  - Determine the screen resolution
  - Obtain system-information, such as screen resolution, using a `Toolkit` object
    - `Toolkit`’s `getScreenSize()`
      - Returns screen resolution as a `Dimension` object
  - `Toolkit, Dimension`: part of `java.awt` package

Screen Resolution

- `Dimension` object has a width and height, in pixels
  - public instance fields

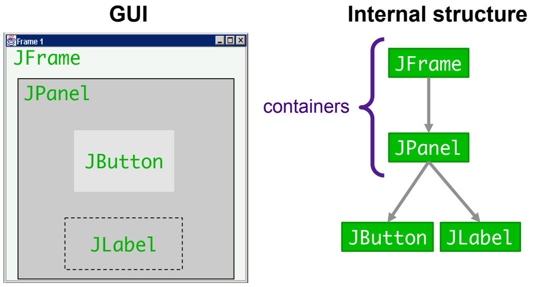
```
Toolkit kit = Toolkit.getDefaultToolKit();
Dimension screenSize = kit.getScreenSize();
int screenWidth  = screenSize.width;
int screenHeight = screenSize.height;
```

Example

What will this Frame look like?

```
class CenteredFrame extends JFrame {

      public CenteredFrame() {
        Toolkit kit = Toolkit.getDefaultToolkit();
        Dimension screenSize = kit.getScreenSize();
        int screenHeight = screenSize.height;
        int screenWidth  = screenSize.width;

        setSize(screenWidth / 2, screenHeight / 2);
        setLocation(screenWidth / 4, screenHeight / 4);

        setTitle(“My Centered 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

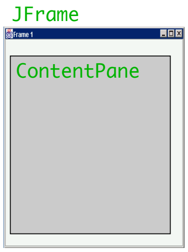
order
important

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

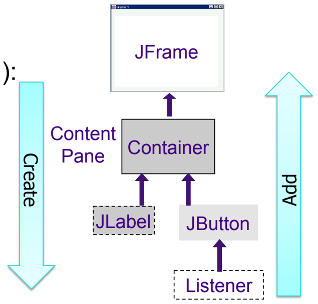
JFrame

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



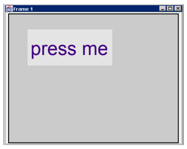
Building a GUI

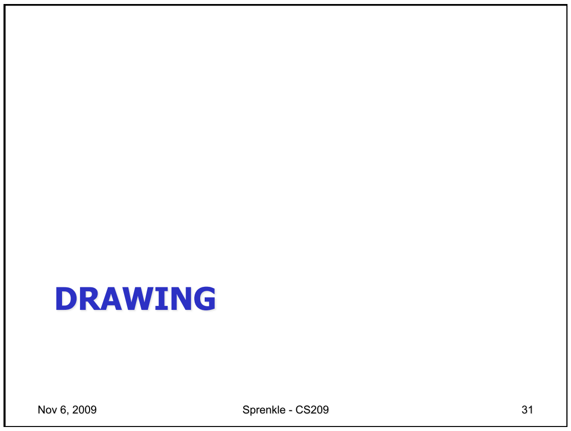
1. Create (top down):
   - Frame
   - Container
   - Components
   - Listeners
2. Add (bottom up):
   - Listeners into components
   - Components into panel
   - Panel into frame



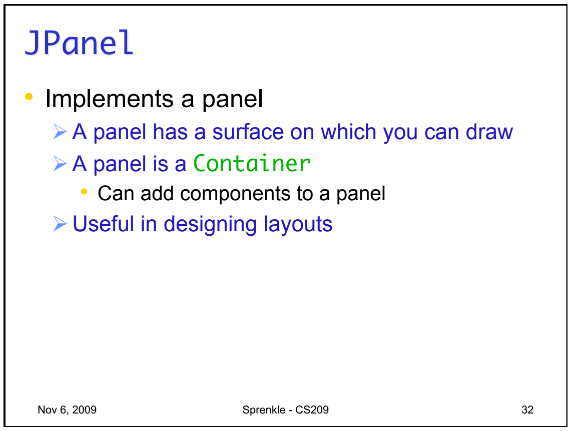
Example Code

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

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

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



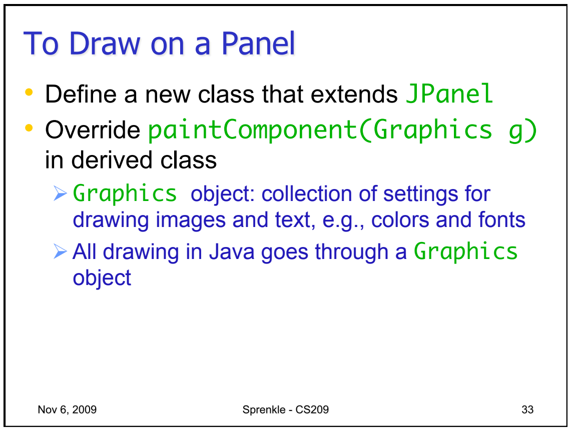


DRAWING



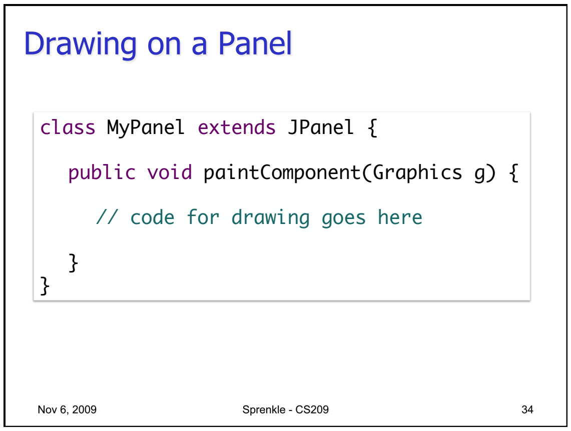
JPanel

- Implements a panel
  - A panel has a surface on which you can draw
  - A panel is a Container
    - Can add components to a panel
  - Useful in designing layouts



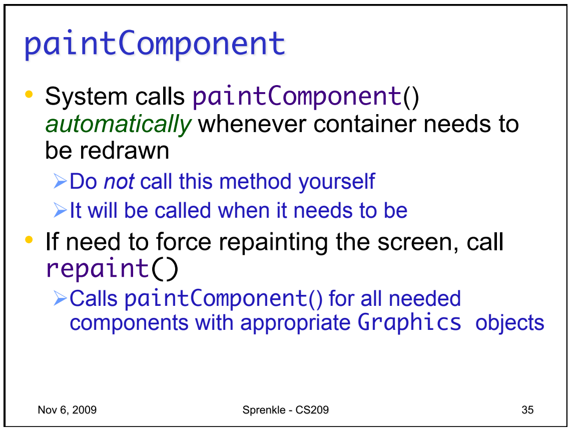
To Draw on a Panel

- Define a new class that extends JPanel
- Override paintComponent(Graphics g) in derived class
  - Graphics object: collection of settings for drawing images and text, e.g., colors and fonts
  - All drawing in Java goes through a Graphics object



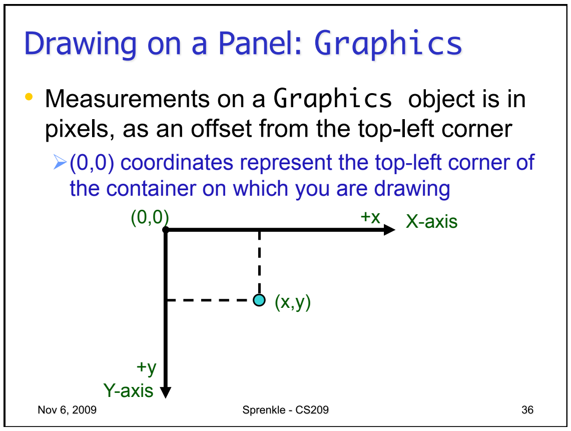
Drawing on a Panel

```
class MyPanel extends JPanel {

   public void paintComponent(Graphics g) {

      // code for drawing goes here

   }
}
```



paintComponent

- System calls paintComponent() *automatically* whenever container needs to be redrawn
  - Do *not* call this method yourself
  - It will be called when it needs to be
- If need to force repainting the screen, call repaint()
  - Calls paintComponent() for all needed components with appropriate Graphics objects



Drawing on a Panel: Graphics

- Measurements on a Graphics object is in pixels, as an offset from the top-left corner
  - (0,0) coordinates represent the top-left corner of the container on which you are drawing

Rendering Text

- Displaying text is a special type of drawing, called *rendering text*
- To render text on a panel, call `drawString()`

```
class HelloWorldPanel extends JPanel {
      public static final int MESSAGE_X = 75;
      public static final int MESSAGE_Y = 100;

      public void paintComponent(Graphics g) {
            super.paintComponent(g);

            g.drawString(“Hello World.”,
                  MESSAGE_X, MESSAGE_Y);
      }
}
```

Drawing on a Panel

- Notice we call superclass's (`JPanel`) `paintComponent()` method
- `JPanel` has its own idea on how to draw/ paint the panel
  - Fills in the background color
- To make sure background color gets filled, call superclass's `paintComponent()`
  - Every `JPanel` should color its background

FONTS

Changing the Text Font

- Previous code drew text using default system font
- Can change the font
- Need to determine which fonts are installed on machine running the program

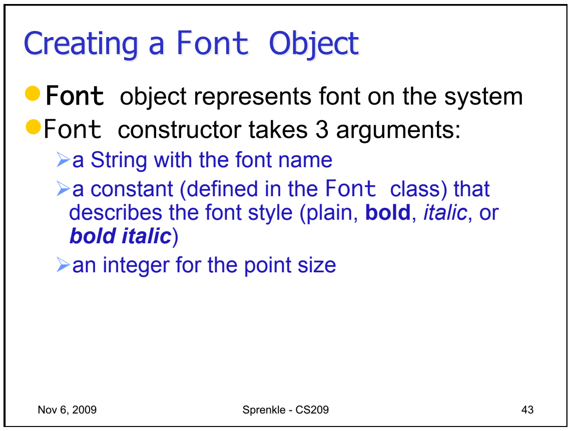
Determining Available Fonts

- `GraphicsEnvironment`
  - Represents the system's graphical environment
  - Call `getAvailableFontFamilyNames()`
    - Returns an array of Strings
    - Each String is the name of a font installed on the system
- Your program can look through fonts to see if font(s) it wants is available on system
- Five fonts are always available, mapped to some font on machine
  - SansSerif, Serif, Monospaced, Dialog, DialogInput

Determining the Available Fonts

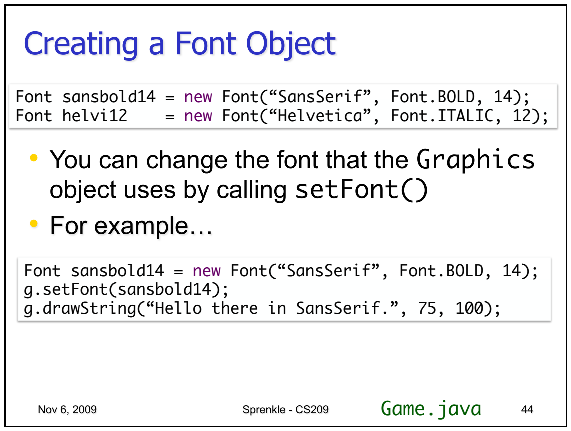
- To list all fonts installed on a particular system:

```
import java.awt.*;

public class ListFonts {

   public static void main(String[] args) {
     String[] fontNames = GraphicsEnvironment
         .getLocalGraphicsEnvironment()
         .getAvailableFontFamilyNames();
     for (int i=0; i < fontNames.length; i++)
           System.out.println(fontNames[i]);
   }
}
```



Creating a Font Object

- `Font` object represents font on the system
- `Font` constructor takes 3 arguments:
  - a String with the font name
  - a constant (defined in the `Font` class) that describes the font style (plain, **bold**, *italic*, or ***bold italic***)
  - an integer for the point size



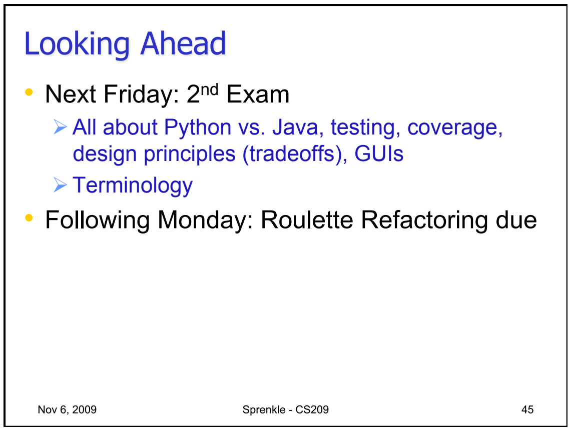
Creating a Font Object

```
Font sansbold14 = new Font(“SansSerif”, Font.BOLD, 14);
Font helvi12    = new Font(“Helvetica”, Font.ITALIC, 12);
```

- You can change the font that the `Graphics` object uses by calling `setFont()`
- For example…

```
Font sansbold14 = new Font(“SansSerif”, Font.BOLD, 14);
g.setFont(sansbold14);
g.drawString(“Hello there in SansSerif.”, 75, 100);
```

`Game.java`



Looking Ahead

- Next Friday: 2nd Exam
  - All about Python vs. Java, testing, coverage, design principles (tradeoffs), GUIs
  - Terminology
- Following Monday: Roulette Refactoring due