CSE 331
Software Design and Implementation

Lecture 18
Java Graphics and GUIs

Leah Perlmutter / Summer 2018

Announcements

Announcements

- Quiz 6 due Thursday 8/2
- Homework 7 due Thursday 8/2
- Regression testing for HW7 ☺
  - **MAKE SURE HW5 and HW6 TESTS PASS!!!!!**
    - **staff tests AND your own tests**
  - Regression testing helps ensure that new changes (HW7) do not (re)introduce bugs in previous work (HW5/6)
  - You will be graded on HW5/6 tests passing in HW7
    - includes staff tests and your own implementation/spec tests

Introduction

The plan

Today: introduction to Java graphics and Swing/AWT libraries

Next Lecture: event-driven programming and user interaction

None of this is comprehensive – only an overview and guide to what you should expect to be out there

- Some standard terminology and perspective

Credits: material taken from many places; including slides and materials by Ernst, Hotan, Mercer, Notkin, Perkins, Stepp; Reges; Sun/Oracle docs & tutorial; Horstmann; Wikipedia; others, folklore, …

References

Very useful start: Sun/Oracle Java tutorials

- http://docs.oracle.com/javase/tutorial/uiswing/index.html

Mike Hoton's slides/sample code from CSE 331 Sp12 (lectures 23, 24 with more extensive widget examples)

- http://courses.cs.washington.edu/courses/cse331/12sp/lectures/lect23-GUI.pdf
- http://courses.cs.washington.edu/courses/cse331/12sp/lectures/lect24-Graphics.pdf
- http://courses.cs.washington.edu/courses/cse331/12sp/lectures/lect23-GUI-code.zip
- http://courses.cs.washington.edu/courses/cse331/12sp/lectures/lect24-Graphics-code.zip

Book that covers this (and much more): *Core Java* vol. I by Horstmann & Cornell

- There are other decent Java books out there too

Why study GUIs?

- Graphical user interfaces are pretty common (duh ☺)
  - And it's fun!
- Example of using inheritance to organize large class libraries
- Work with a large API – and learn how (not) to deal with all of it
- Many core design patterns show up: callbacks, listeners, event-driven programs, decorators, façade

Dos and Don’ts

- Don’t try to learn the whole library: There’s way too much
- Don’t memorize – look things up as you need them

- Do be aware of rabbit holes.
  - **rabbit hole**: A time-consuming tangent or detour, often from which it is difficult to extricate oneself. (Wiktionary)
- Do explore and be creative!
- Do have fun!

Main topics to learn

Organization of the AWT/Swing library

- Names of essential widgets/components

Graphics and drawing

- Repaint callbacks, layout managers, etc.

Handling user events

Building GUI applications

- MVC, user events, updates, …

A very short history (1)

Java's standard libraries have supported GUIs from the beginning

Original Java GUI: AWT (Abstract Window Toolkit)

- Limited set of user interface elements (widgets)
- Mapped Java UI to host system UI widgets
- Lowest common denominator
- "Write once, debug everywhere"

A very short history (2)

less old

Swing: ~~Newer~~ GUI library, introduced with Java 2 (1998)

Basic idea: underlying system provides only a blank window

- Swing draws all UI components directly
- Doesn't use underlying system widgets

Not a total replacement for AWT: Swing is implemented on top of core AWT classes and both still coexist

Use Swing, but deal with AWT when you must

Java Swing

GUI terminology

window: A first-class citizen of the graphical desktop

- Also called a *top-level container*
- Examples: *frame*, dialog box, applet

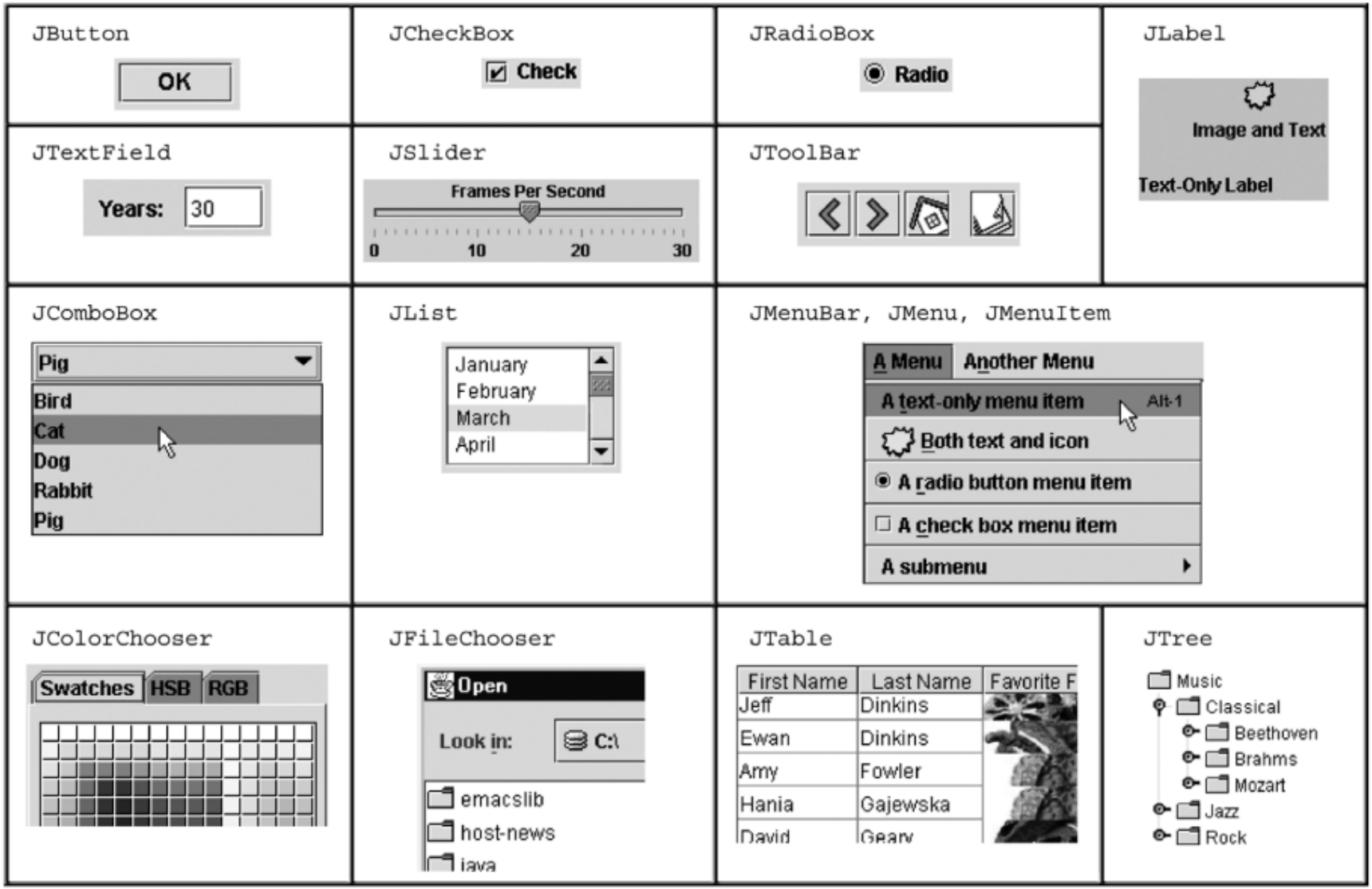
component: A GUI *widget* that resides in a window

- Called *controls* in many other languages
- Examples: button, text box, label

container: A component that hosts (holds) components

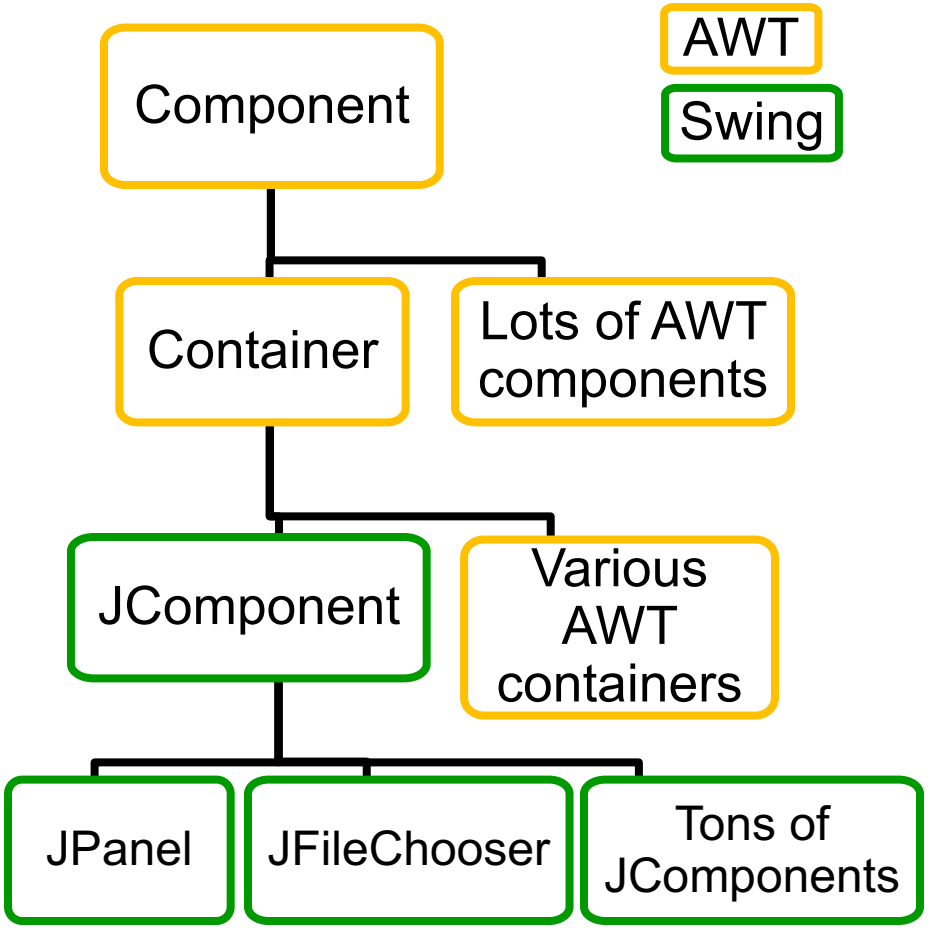
- Examples: frame, applet, *panel*, box

Some components…



Component and container classes

- Every GUI-related class descends from Component, which contains dozens of basic methods and fields
  - Examples: **`getBounds`**, **`isVisible`**, **`setForeground`**, …
- "Atomic" components: labels, text fields, buttons, check boxes, icons, menu items…
- Many components are containers – things like panels (**`JPanel`**) that can hold nested subcomponents



Swing/AWT inheritance hierarchy

```
Component
    Container
        Window
            Dialog
                JDialog
            Frame
                JFrame
        JComponent
            JButton
            JComboBox
            JMenuBar
            JPopupMenu
            JScrollPane
            JSplitPane
            JToolbar
            JTextField
            JColorChooser
            JLabel
            JOptionPane
            JProgressBar
            JSlider
            JTabbedPane
            JTree
            ...
            JFileChooser
            JList
            JPanel
            JScrollbar
            JSpinner
            JTable
            JTextArea
```

KEY:
AWT (black)
Swing (blue)

Component properties

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

name	type	description
background	**`Color`**	background color behind component
border	**`Border`**	border line around component
enabled	**`boolean`**	whether it can be interacted with
focusable	**`boolean`**	whether key text can be typed on it
font	**`Font`**	font used for text in component
foreground	**`Color`**	foreground color of component
height, width	**`int`**	component's current size in pixels
visible	**`boolean`**	whether component can be seen
tooltip text	**`String`**	text shown when hovering mouse
size, minimum / maximum / preferred size	**`Dimension`**	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
  - Usually 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 **`JComponent`**s are containers

`JFrame` – top-level window

- Graphical window on the screen
- Typically holds (hosts) other components
- Common methods:
  - **`JFrame(String`** *title*`)`: constructor, title optional
  - **`setDefaultCloseOperation(int`** *what*`)`
    - What to do on window close
    - **`JFrame.EXIT_ON_CLOSE`** terminates application
  - **`setSize(int`** *width*`, int`** *height*`)`: set size
  - **`add(Component`** *c*`)`: add component to window
  - **`setVisible(boolean`** *b*`)`: make window visible or not

Example

SimpleFrameMain.java

`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(…))
  ```

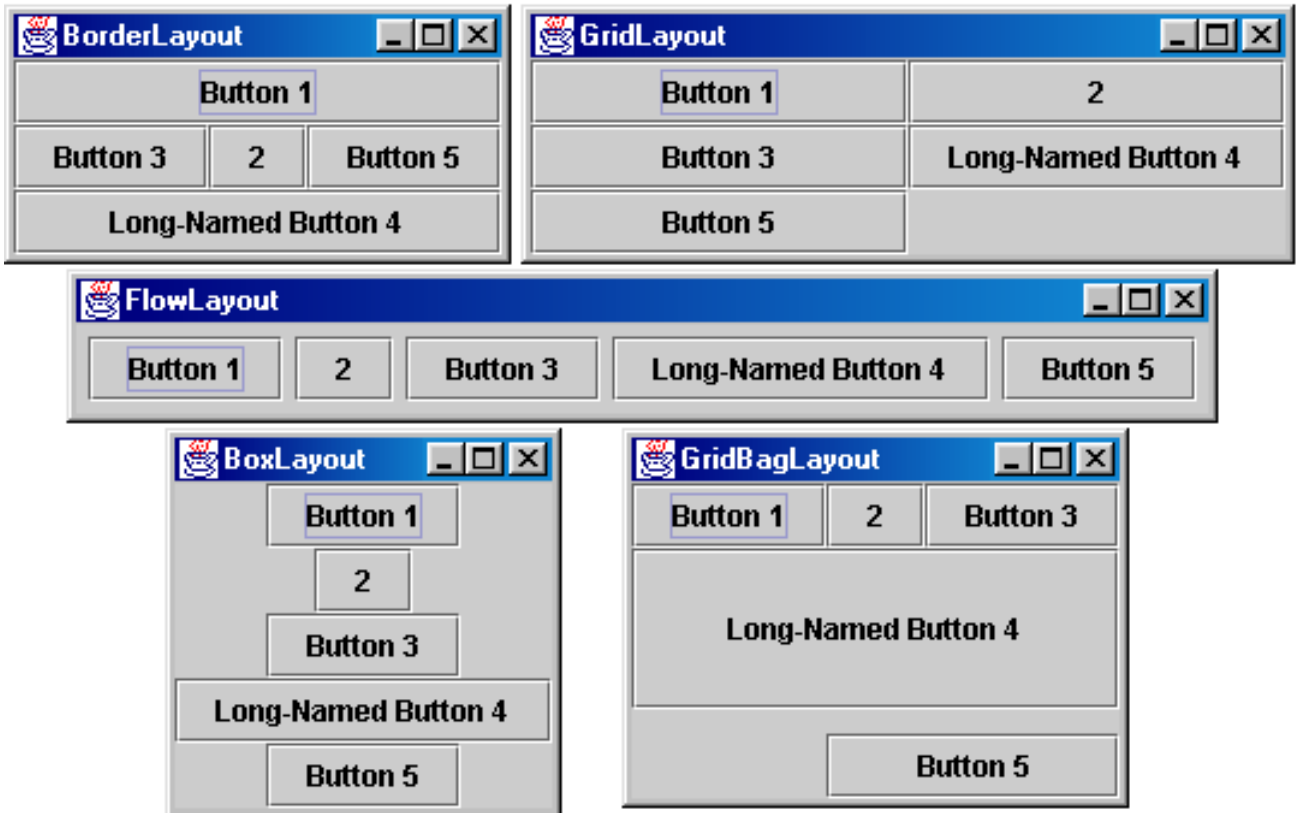
- **`JPanel`**s can be nested to any depth
- Many methods/fields in common with **`JFrame`** (since both inherit from **`Component`**)
  - Advice: can't find a method/field? Check the superclasses

A particularly useful method:

- **`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 [changeable], top to bottom)
  - Default for **`JPanel`**
  - Each row centered horizontally [changeable]
- **`BorderLayout`** (“center”, “north”, “south”, “east”, “west”)
  - Default for **`JFrame`**
  - No more than one component in each of 5 regions
  - (Of course, component can itself be a container)
- **`GridLayout`** (regular 2-D grid)
- Others... (some are incredibly complex)

`FlowLayout` and **`BorderLayout`** should be good enough for now…

`pack()`

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

```
frame.pack();
frame.setVisible(true);
```

`pack()` figures out the sizes of all components and calls the container's 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

Graphics and Drawing in Swing

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`**

- Components like **`JLabel`** provide a suitable **`paintComponent`** that (in **`JLabel`**'s case) draws the label text
- Other components like **`JPanel`** typically inherit an empty **`paintComponent`** and can override it to draw things

Note: As we'll see, *we* override **`paintComponent`** but *we don't* call it

Graphics and drawing

How does custom stuff get drawn to the screen?

Your Code

you define a callback!

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

public class SimpleLayoutMain {

  /** Create an empty, labeled panel and
  public static void main(String[] args)
```

```
@Override protected void
paintComponent(Graphics g)
{...}
```

```
    frame.add(label,BorderLayout.SOUTH);
    frame.pack(); // what happens if we omit this?
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
```

```
frame.setVisible(true);
```

```
    JPanel panel = new JPanel();
    panel.setPreferredSize(new Dimension(300,200));

    JLabel label = new JLabel("Smile!!");
    label.setHorizontalAlignment(SwingConstants.CENTER);
    label.setBorder(new StrokeBorder(new BasicStroke(2.0f)));

    frame.add(panel,BorderLayout.CENTER);
    frame.add(label,BorderLayout.SOUTH);
    frame.pack(); // what happens if we omit this?
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

AWT and Swing magic takes over

your callback gets called!

Window Manager

magic

...

add paint event to the event queue

...

paintComponent()

...

more magic

Runs in parallel with your other code

Example

SimplePaintMain.java

Graphics methods

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

Can also draw images (pictures, etc.):

- In the program (***not*** in **`paintComponent`**):
  - Use AWT's "Toolkit" to load an image:

```
Image pic =
    Toolkit.getDefaultToolkit()
        .getImage(file-name (with 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 an object interface – create instances of **`Shape`** like **`Line2D`**, **`Rectangle2D`**, etc., and add these to the **`Graphics2D`** object

Actual parameter to **`paintComponent`** is always a **`Graphics2D`**

- Can always cast this parameter from **`Graphics`** to **`Graphics2D`**
- **`Graphics2D`** supports both sets of graphics methods
- Use whichever you like for CSE 331

Graphics and drawing

How does custom stuff get drawn to the screen?

Your Code

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

public class SimpleLayoutMain {

  /** Create an empty, labeled panel and
  public static void main(String[] args)
```

you define a callback!

```
@Override protected void
paintComponent(Graphics g)
{...}
```

```
    frame.add(label,BorderLayout.SOUTH);
    frame.pack(); // what happens if we omit this?
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
```

```
frame.setVisible(true);
```

```
    JPanel panel = new JPanel();
    panel.setPreferredSize(new Dimension(300,200));

    JLabel label = new JLabel("Smile!!");
    label.setHorizontalAlignment(SwingConstants.CENTER);
    label.setBorder(new StrokeBorder(new BasicStroke(2.0f)));

    frame.add(panel,BorderLayout.CENTER);
    frame.add(label,BorderLayout.SOUTH);
    frame.pack(); // what happens if we omit this?
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

AWT and Swing magic takes over

Window Manager

This part is required before paintComponent will work as expected.

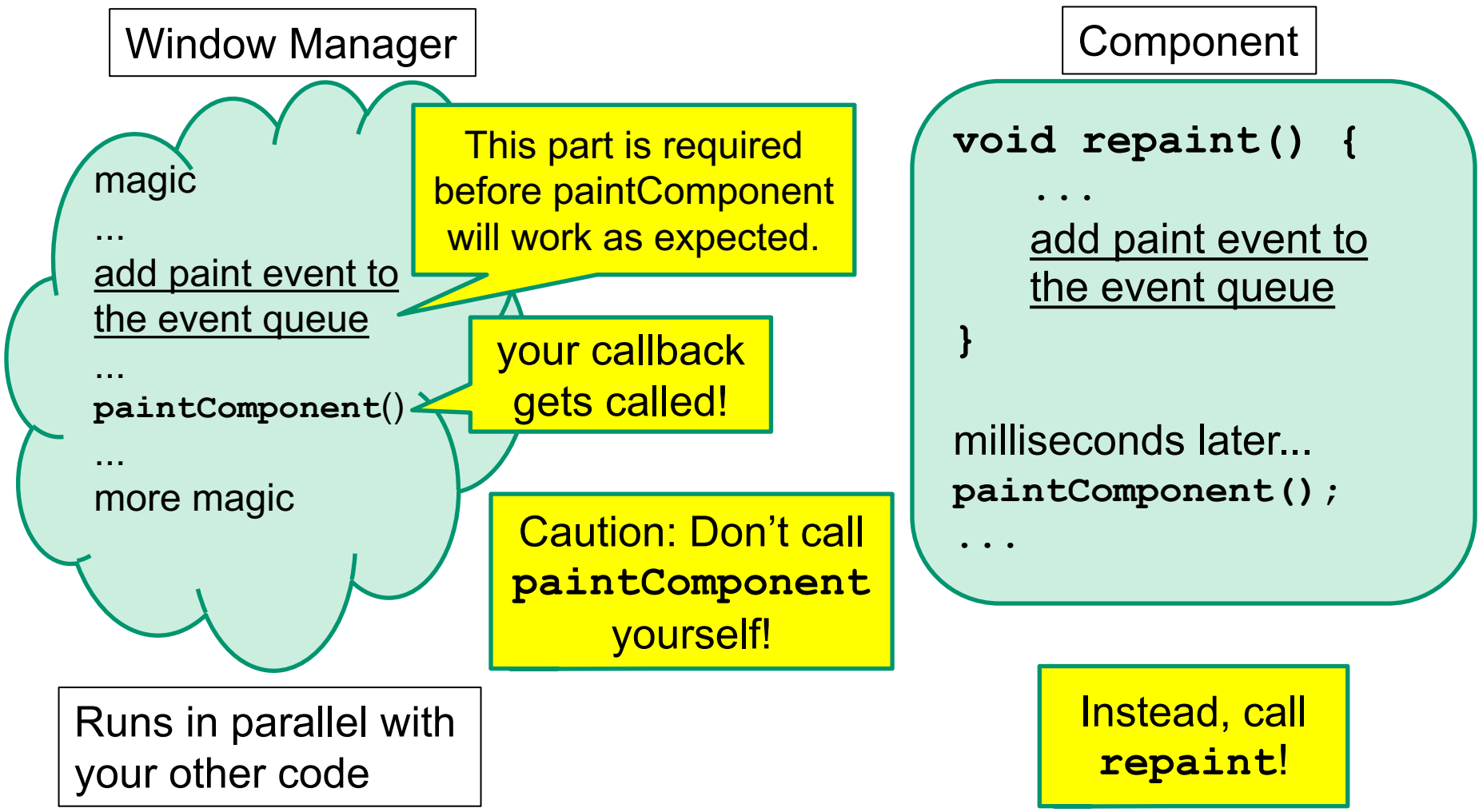
your callback gets called!

magic
...
add paint event to the event queue
...
`paintComponent`()
...
more magic

Caution: Don't call **`paintComponent`** yourself!

Runs in parallel with your other code

Graphics and drawing



So who calls `paintComponent`? And when??

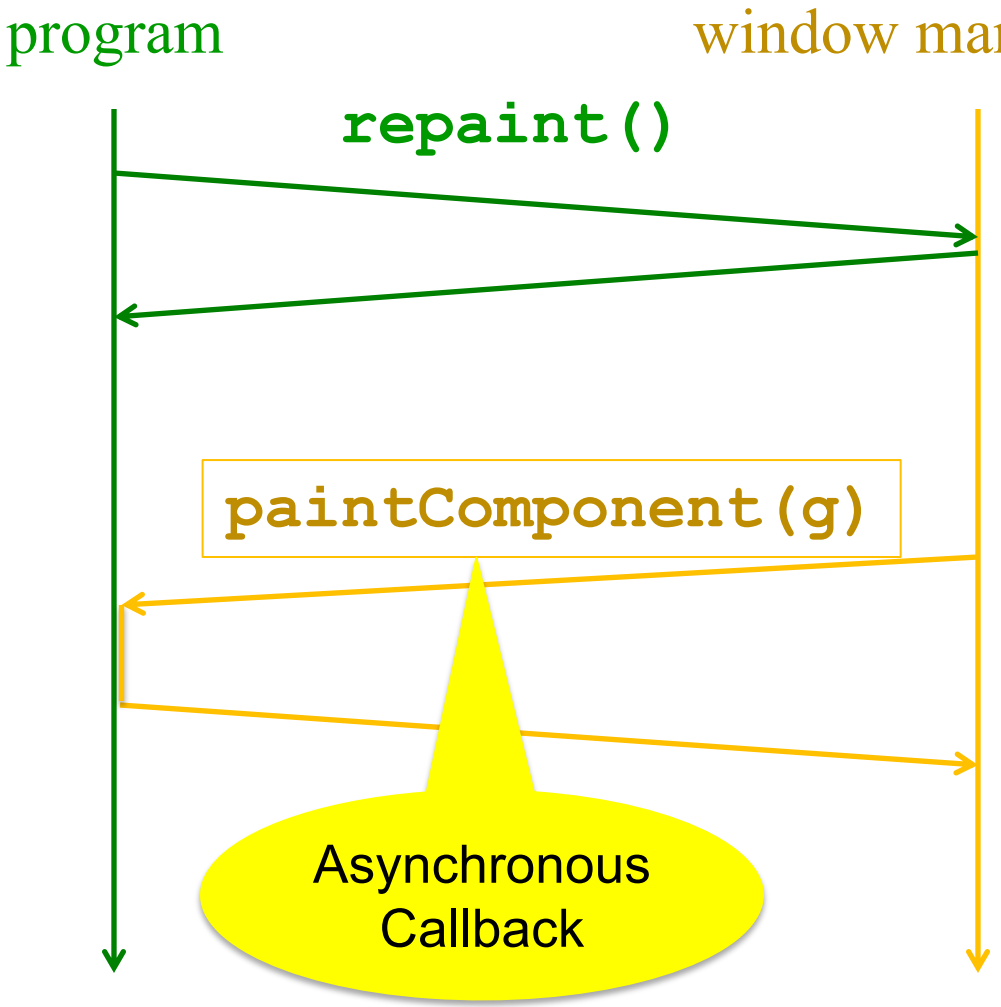
- Answer: the window manager calls **`paintComponent`** *whenever it wants!!!* (a callback!)
  - When the window is first made visible, and whenever after that some or all of it needs to be *repainted*
- Corollary: **`paintComponent`** must ***always*** be ready to repaint regardless of what else is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

Example

Face.java

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!

Crucial rules for painting

- Always override **`paintComponent(g)`** if you want to draw on a component
- Always call **`super.paintComponent(g)`** first
- ***NEVER, EVER, EVER*** call **`paintComponent`** yourself
- 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, or mess with it. 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 that requires deeper understanding of the GUI library's structure and specification

What's next

Major topic for next lecture is how to handle user interactions

- We already know the core idea: it's a big-time use of the observer pattern

Beyond that you're on your own to explore all the wonderful widgets in Swing/AWT.

- Have fun!!
- (But beware of time sinks)

Announcements

Announcements

- Quiz 6 due Thursday 8/2
- Homework 7 due Thursday 8/2
- Regression testing for HW7 ☺
  - **MAKE SURE HW5 and HW6 TESTS PASS!!!!!**
    - **staff tests AND your own tests**
  - Regression testing helps ensure that new changes (HW7) do not (re)introduce bugs in previous work (HW5/6)
  - You will be graded on HW5/6 tests passing in HW7
    - includes staff tests and your own implementation/spec tests