CSE 331
Software Design & Implementation

Kevin Zatloukal

Fall 2017

Java GUIs

(Based on slides by Mike Ernst, Dan Grossman, David Notkin, Hal Perkins, Zach Tatlock)

Reminders

- HW8 due today

- Section tomorrow on Android for HW9
  - install Android Studio beforehand if you plan to use Android

Review

- Event-driven program is one whose main loop waits for an event and then processes it (over and over until quit time)
  - this sort of loop is called an event loop

- Examples of event-driven programs: servers & GUIs

- Technicalities (IRL not necessarily for HW9):
  - OSes only let you wait for certain types of events at once
  - work around it by having another thread list for other types
    - (but be careful about what work is done on which thread)

- GUIs differ in support for resizing
  - Android / iPhone and bootstrap (HTML) support fixed sizes

Java AWT / Swing

References on Java AWT / Swing

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

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

- there are other decent Java books out there too

What not to do…

- Don’t try to learn the whole library: there’s way **too much**
- Don’t memorize – look things up as you need them
  - expect to look things up as you switch from Android to iPhone to HTML to ...
- Don’t miss the main ideas & fundamental concepts
- Don’t get bogged down implementing eye candy for HW9
  - (unless you finish everything else)

A very short history (1)

Java’s standard libraries have supported GUIs from the beginning

Original Java GUI: AWT (Abstract Window Toolkit)

- mapped Java UI to host system UI widgets
- limited set of user interface elements (widgets)
  - lowest common denominator

Advantage: looks native

Disadvantage: “write once, debug everywhere”

A very short history (2)

Swing: new*er* GUI library, introduced with Java 2 (1998)

- Android Studio, IntelliJ built using Swing

Basic idea: underlying system provides only a blank window

- Swing draws all UI components directly
- doesn't use underlying system widgets
- (built on top of parts of AWT)

Advantage: **should** work the same on all platforms

- less testing work in general (but be skeptical of that claim)

Disadvantage: doesn't look like a native GUI for that OS

A very short history (3)

SWT: improved version of AWT approach (2004?)

- tries to expose all the functionality of native GUIs
- Eclipse is built using SWT
- not part of the standard Java library

Two choices:

1. Use Swing to make a GUI that looks / works consistently
2. Use SWT to make a native-looking GUI on each platform

Option 1 is less work.

Option 2 usually makes users happier.

We'll cover Swing since it's standard Java...

A very short history (4)

Android: platform for writing phone/tablet apps with Java

- not part of the standard Java library
- open source project from Google

Conceptually similar to AWT/Swing

- but Android devices should look and behave similarly

Unfortunately cannot reuse AWT/Swing code

Main topics to learn

Using AWT/Swing components (a.k.a. widgets):

- different types of components [today]
- how to lay them out in a window [Friday]
- how to handle widget events [last time]

Writing your own components [Friday]:

- how to draw your own UI
- how to handle lower level events

GUI terminology

window: A first-class citizen of the graphical desktop

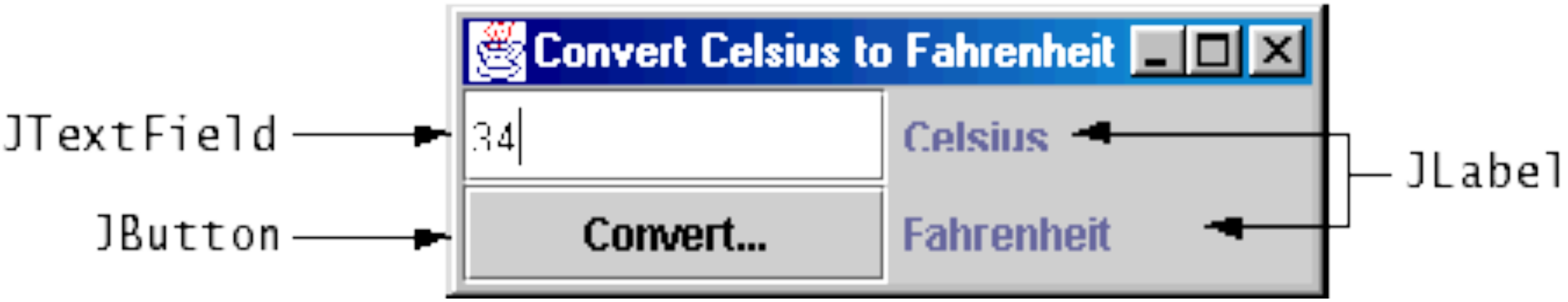
- also called a *top-level container*
- Examples: *frame* (window), dialog box

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

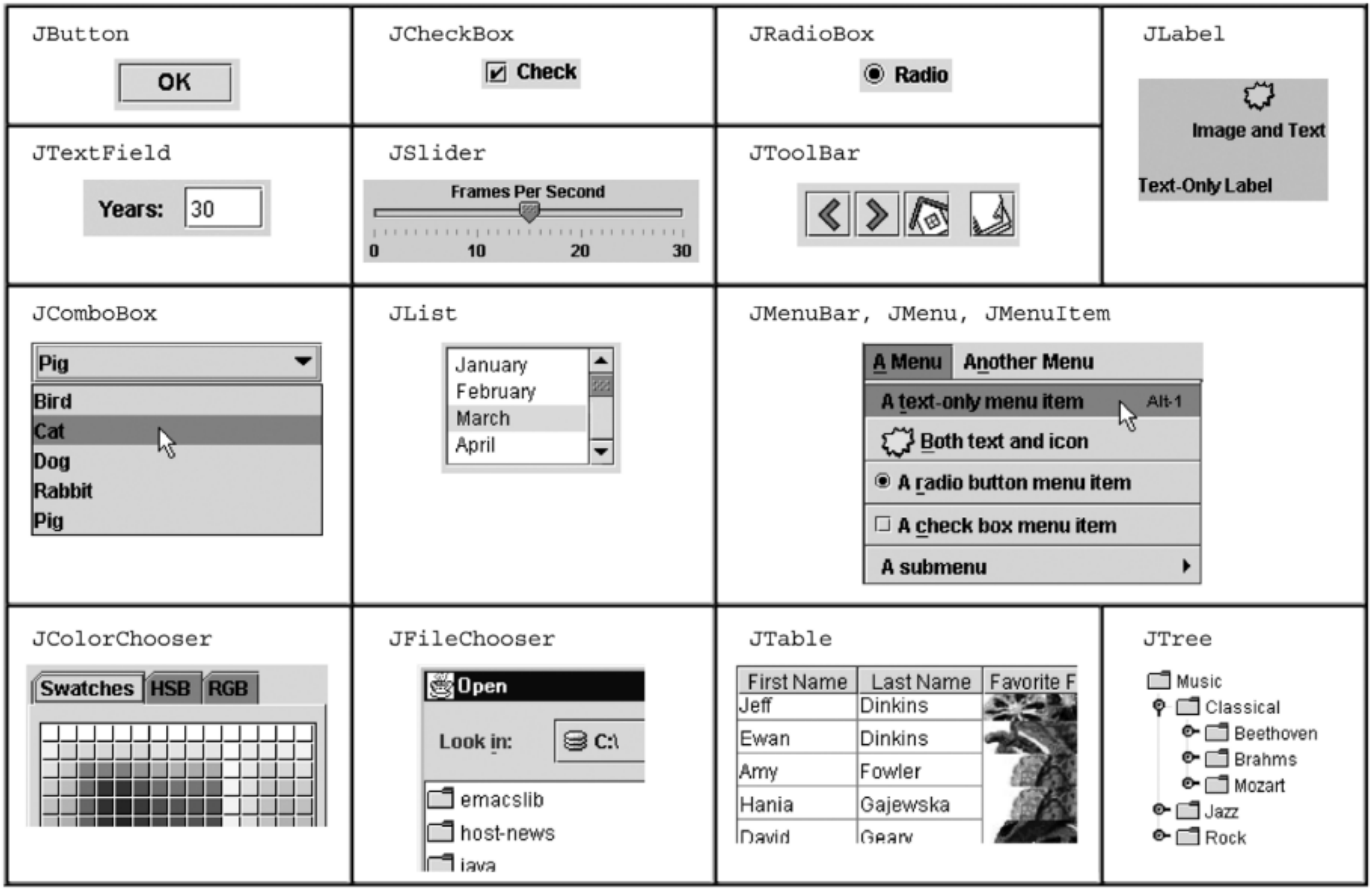
- called *controls* in many other languages
- Examples: button, text box, label

container: A component that hosts (holds) components

- Examples: frame, *panel*, box



Some components…



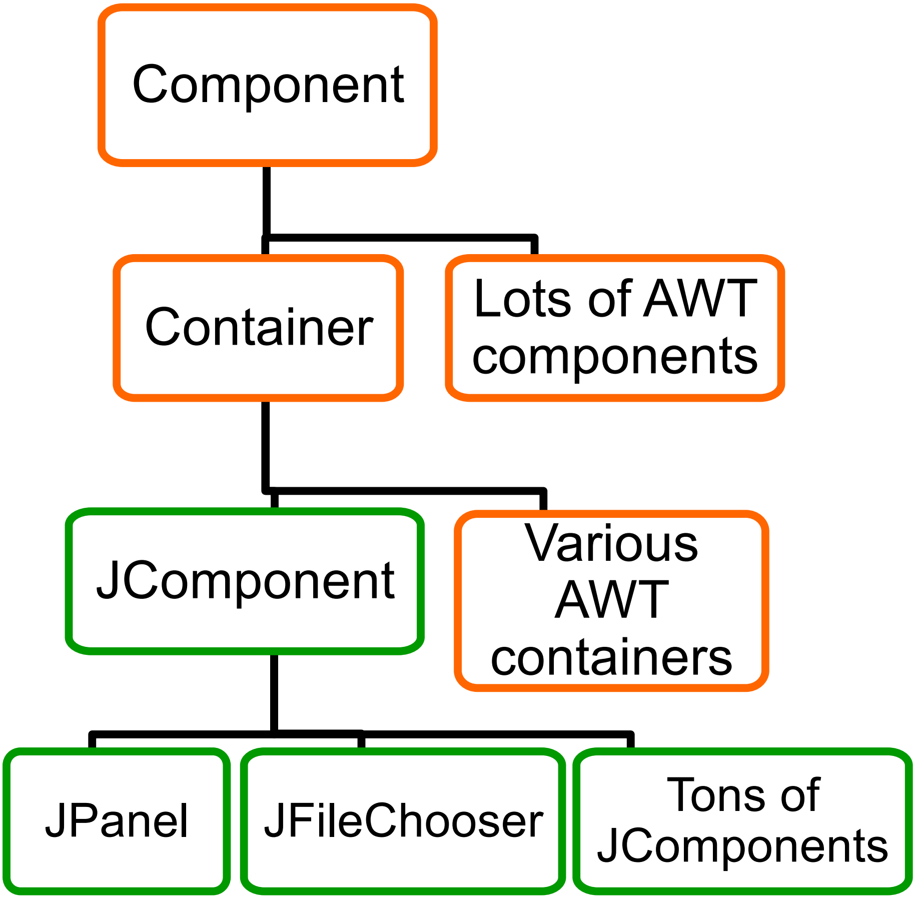
Years: 30

Look in: C:\

First Name	Last Name	Favorite F
Jeff	Dinkins	
Ewan	Dinkins	
Amy	Fowler	
Hania	Gajewska	
David	Geary	

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` (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 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`**, …
  - usually correspond to OS windows
  - 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, …

`JFrame` – top-level window

- Graphical window on the screen
- Holds 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
  - **`setVisible(boolean`** *b***`)`** : make window visible or not

Example

`SimpleFrameMain.java`

`JFrame` – top-level window

- Graphical window on the screen
- Holds 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
  - **`setVisible(boolean`** *b***`)`** : make window visible or not
  - **`add(Component`** *c***`)`** : add component to window

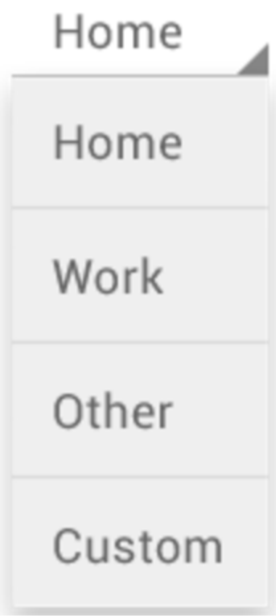
Android

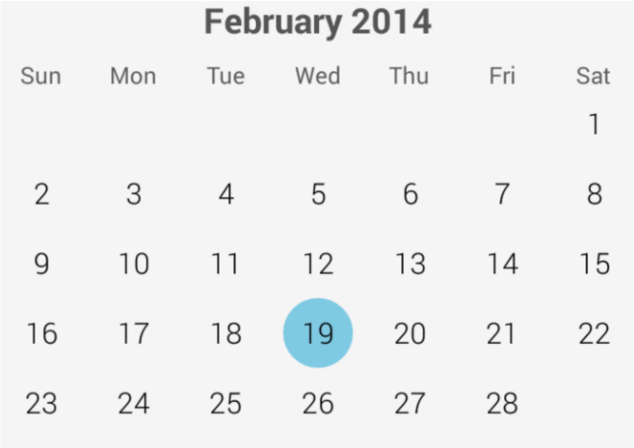
Components

Many of the same ones

But some new ones

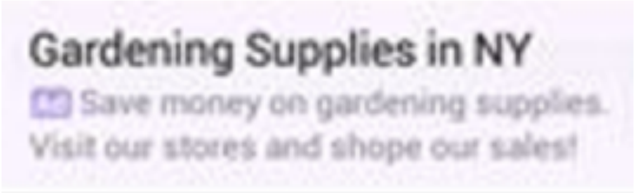
- spinner
- seek bar
- rating bar
- calendar view
- ad view
- ...





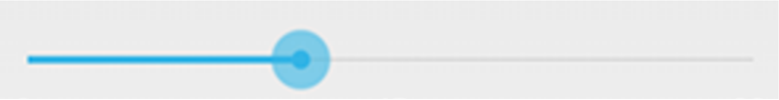
February 2014

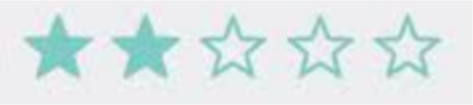
Sun	Mon	Tue	Wed	Thu	Fri	Sat
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	



Gardening Supplies in NY

Ad Save money on gardening supplies. Visit our stores and shope our sales!





Containers

Components are subclasses of View
Containers are subclasses of ViewGroup

Commonly used containers:

- grid
- list view
- linear layout (horizontal or vertical)
- positions children relative to others (e.g., above, to right, centered)

(Ideally, you would skip this and layout at fixed positions.)

First two can be easily used to display data

  - (see HW9)

Activities

Android uses a model similar to a web browser:

- each page is called an "activity"
- back button takes you back to the previous activity

Each app creates one or more activities

- main activity is (normally) started when the app starts
- startActivity(this, OtherActivity.class) starts another activity

Activity is notified when it is in use

- onCreate called to create the UI
- onStop called when it is no longer visible
- onDestroy called when it is destroyed

Back to Swing

Example

`SimpleButtonDemo.java`

Where is the event loop?

GUIs are event-driven programs, so where is the event loop?

- It is created automatically by Swing
  - presumably when we call `frame.setVisible(true)`

- The main method actually returns…

- Swing creates another thread to run the GUI event loop
  - this is called the UI thread
  - the Java VM does not quit the program until *all threads* exit

Example

`SimpleButtonDemo2.java`

`JPanel` – a general-purpose container

- Commonly used to hold a collection of button, labels, etc.
  - (also has another use you will learn about in section)
- 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`**)
  - Can’t find a method/field? Check the superclasses.

A particularly useful method:

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

Example

`SimpleButtonDemo3.java`

Layout in AWT/ Swing

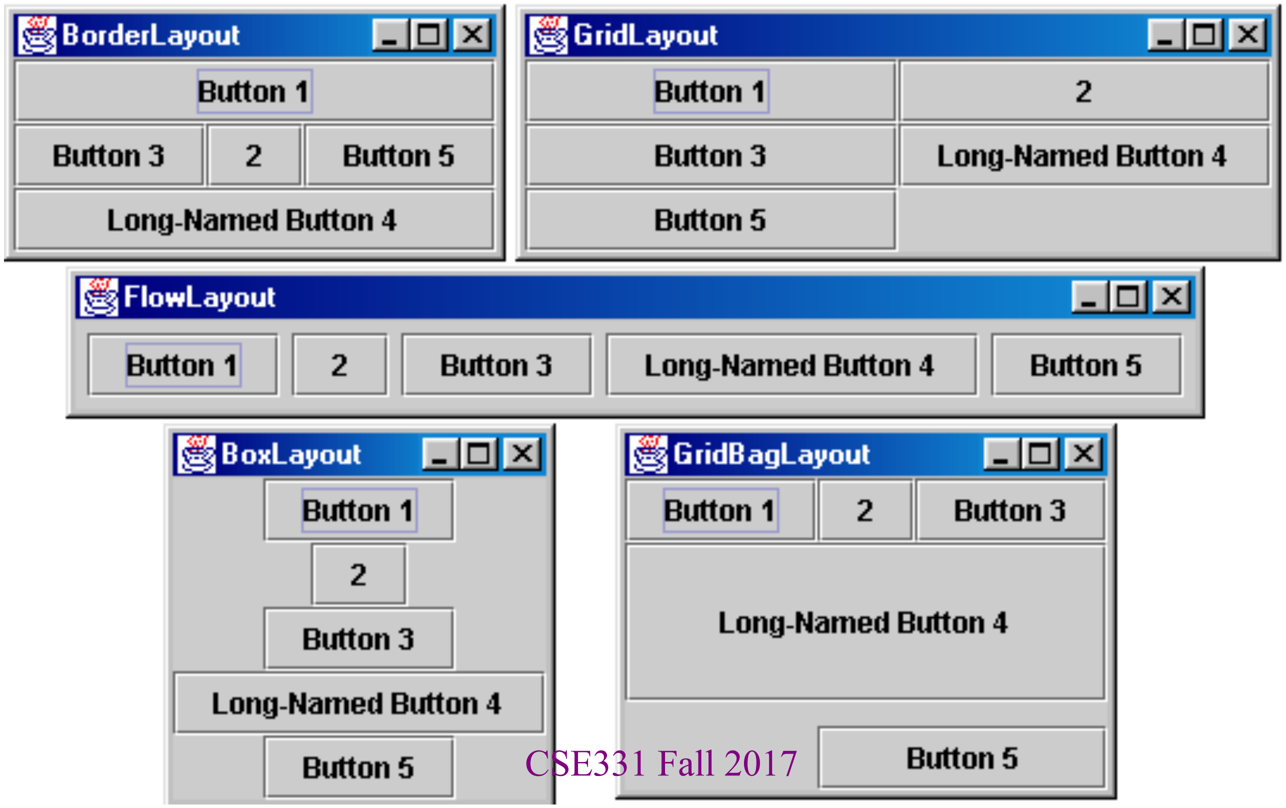
Example

`SimpleFieldDemo.java`

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. None are perfect.)

Layout managers

You can change the layout manager on any **`JComponent c`**

- **`c.setLayout(new GridLayout())`**

`FlowLayout` and **`BorderLayout`** are likely good enough for now…

(There are similar issues creating UI in HTML…)

Example

`SimpleFieldDemo2.java`

Example

`SimpleFieldDemo3.java`

`pack()`

Instead of having the components lay out within the window size, you can instead size the window to fit the components:

```
    frame.pack();
```

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

- (recursively as needed)

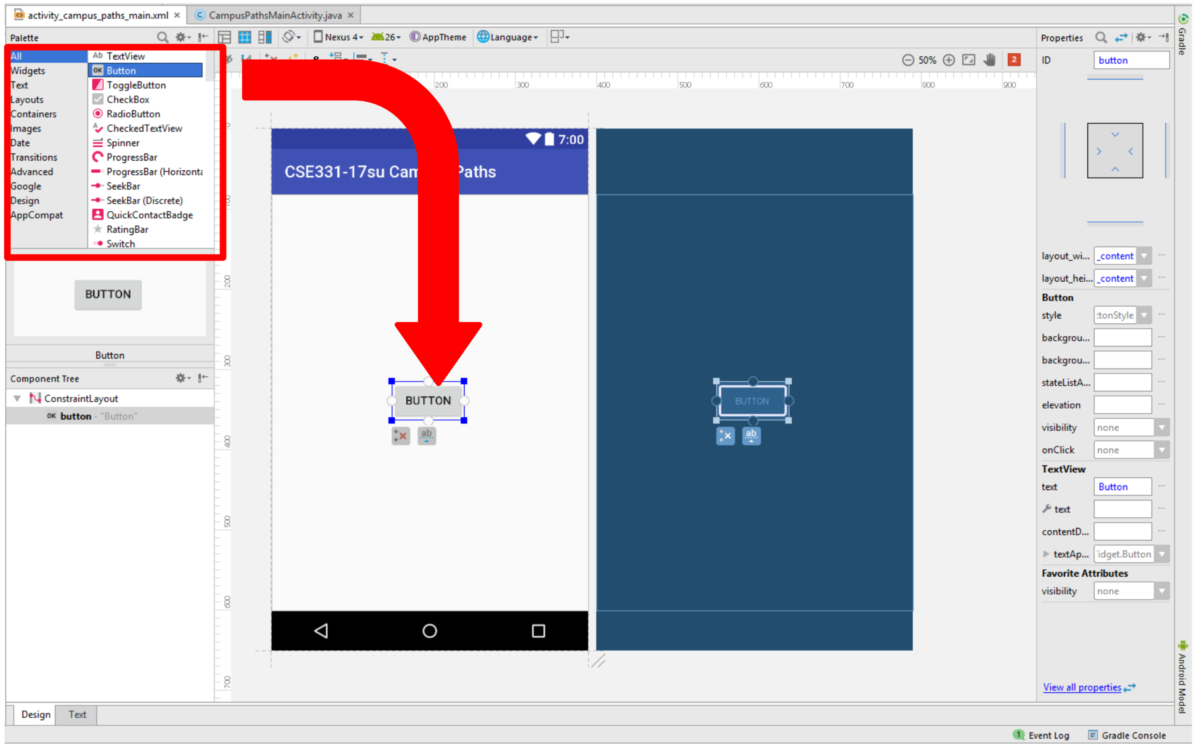
Example

`SimpleFieldDemo4.java`

Android Layout

Activity xml file specifies components layout

Add components from list of all possible components via drag-and-drop mechanics in a graphical user interface and scale them to the desired size



activity_campus_paths_main.xml × | CampusPathsMainActivity.java ×

Properties

ID: button

layout_wi...: _content

layout_hei...: _content

Button

style: :tonStyle

visibility: none

onClick: none

TextView

text: Button

textAp...: idget.Button

Favorite Attributes

visibility: none

Graphics and Drawing

Graphics and drawing (Swing)

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 (Android)

What if we want to actually draw something?

- A map, an image, a path, …?

Answer: Override method **`onDraw`**

- Components like **`ImageView`** typically inherit an empty **`onDraw`** and can override it to draw things
- Other components typically have attributes you edit in the design interface or an xml file that allow you to edit the text that appears (i.e. the text on a Button)

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

**“Drawing” in Android is synonymous to “Painting” in Swing

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`**):

  ```
  Image pic = ImageIO.read(new File(...));
  ```

- 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`** (extends **`Graphics`**)

- adds an object interface: **`draw(Shape s)`**
- adds other new capabilities (e.g., **`AffineTransform`**)
- see the documentation for details

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

So who calls **`paintComponent`**? And when??

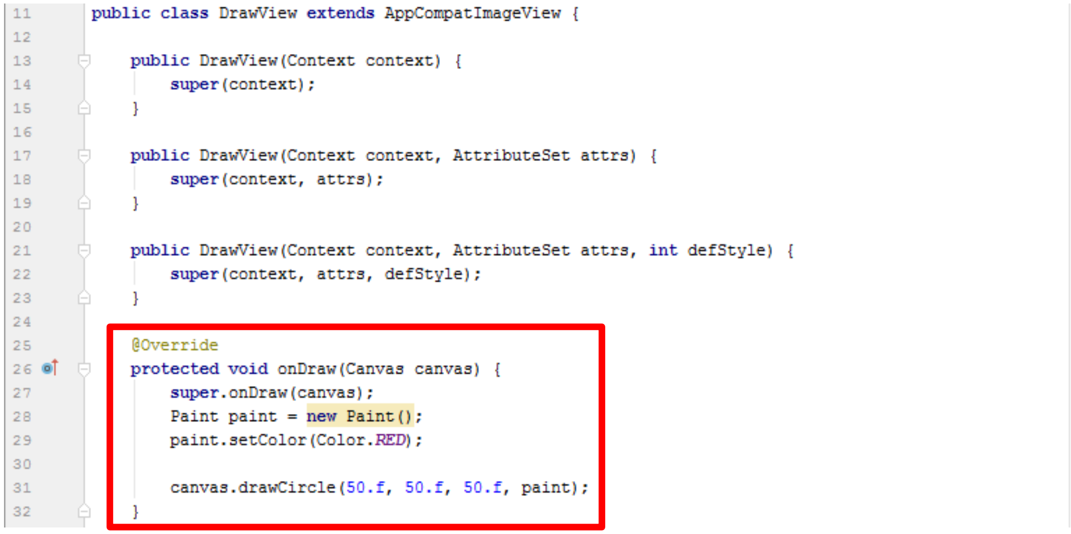
- Answer: the window manager calls **`paintComponent`** *whenever it wants!!!* (a callback!)
  - When the window is first made visible, and whenever after that some or all of it needs to be *repainted*
- Corollary: **`paintComponent`** must ***always*** be ready to repaint regardless of what else is 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()`** 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)
  - Window manager may combine several quick **`repaint()`** requests and call **`paintComponent()`** only once

Android – Graphics and drawing

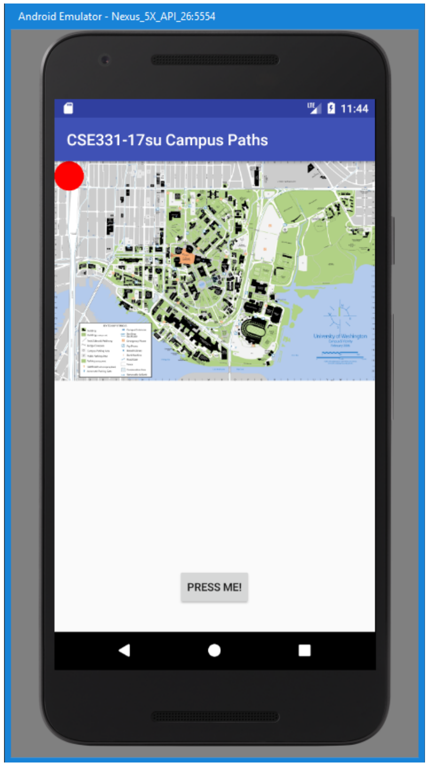
Extend **`AppCompatImageView`** class and override **`onDraw`** method

Like **`paintComponent`** in Swing, we *<u>don’t</u>* call **`onDraw`** in Android
Instead, use **`invalidate()`** to request the app to be redrawn

`Canvas` parameter in **`onDraw`** like **`Graphics`** parameter from **`paintComponent`** in Swing



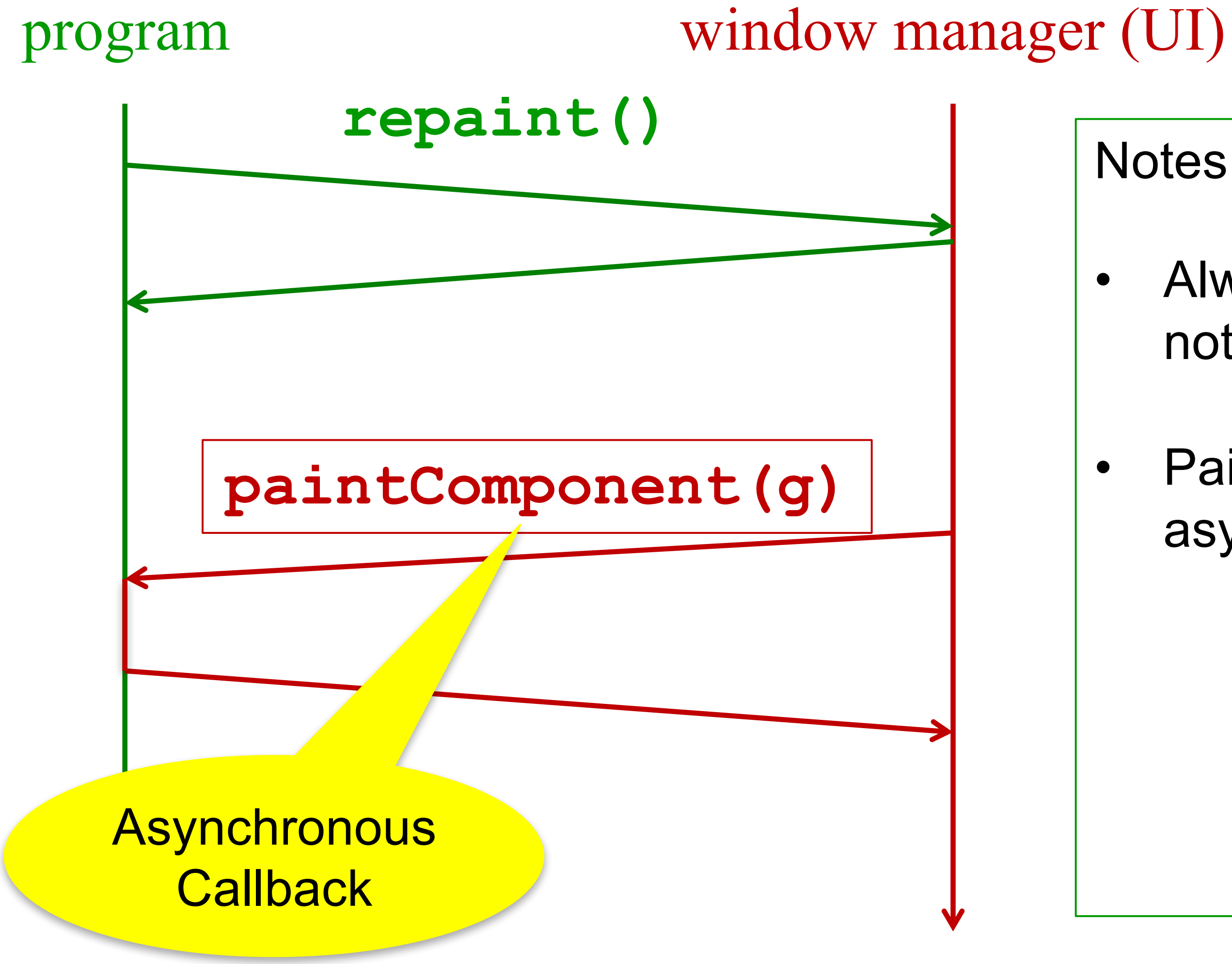
```java
public class DrawView extends AppCompatImageView {

    public DrawView(Context context) {
        super(context);
    }

    public DrawView(Context context, AttributeSet attrs) {
        super(context, attrs);
    }

    public DrawView(Context context, AttributeSet attrs, int defStyle) {
        super(context, attrs, defStyle);
    }

    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        Paint paint = new Paint();
        paint.setColor(Color.RED);

        canvas.drawCircle(50.f, 50.f, 50.f, paint);
    }
```



Example

`FaceMain.java`

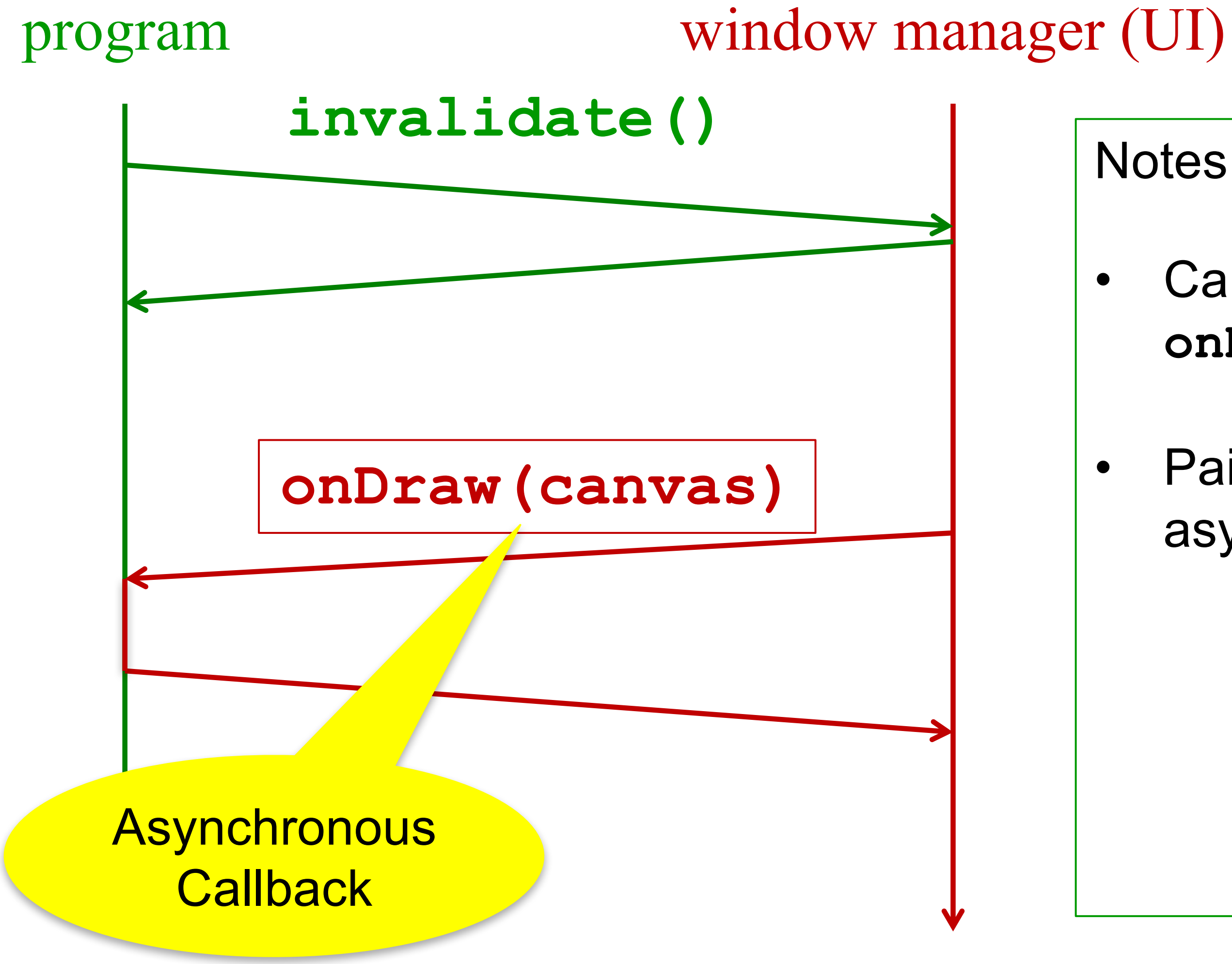
How repainting happens (Swing)



Notes:

- Always call **`repaint`** not **`paintComponent`**
- Painting occurs asynchronously

How redrawing happens (Android)



Notes:

- Call **`invalidate`** not **`onDraw`**
- Painting occurs asynchronously

Crucial rules for painting (Swing)

- 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

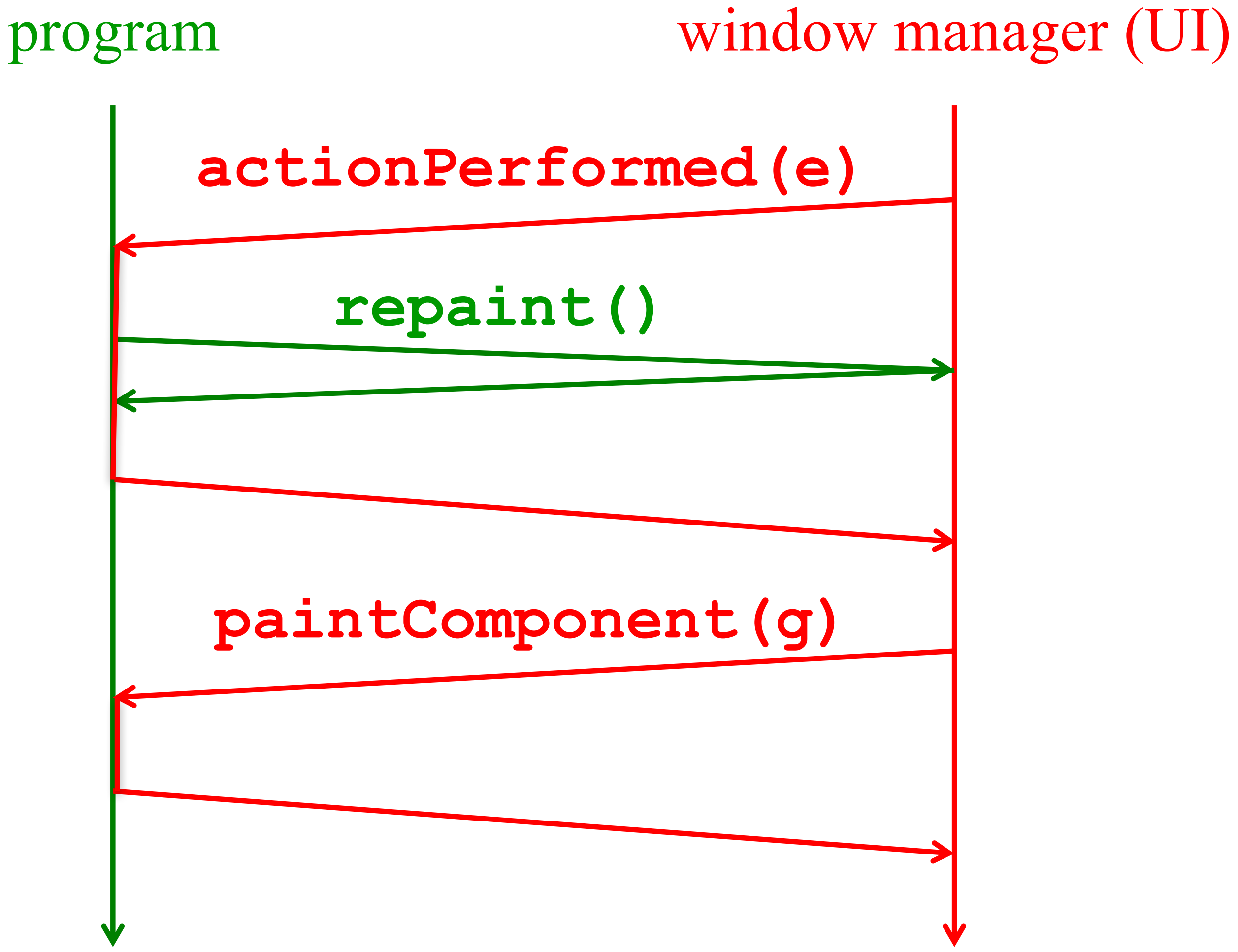
Fine print: Once you are a certified™ wizard, you may find reasons to do things differently, but that requires deeper understanding of the GUI library's structure and specification

Crucial rules for drawing (Android)

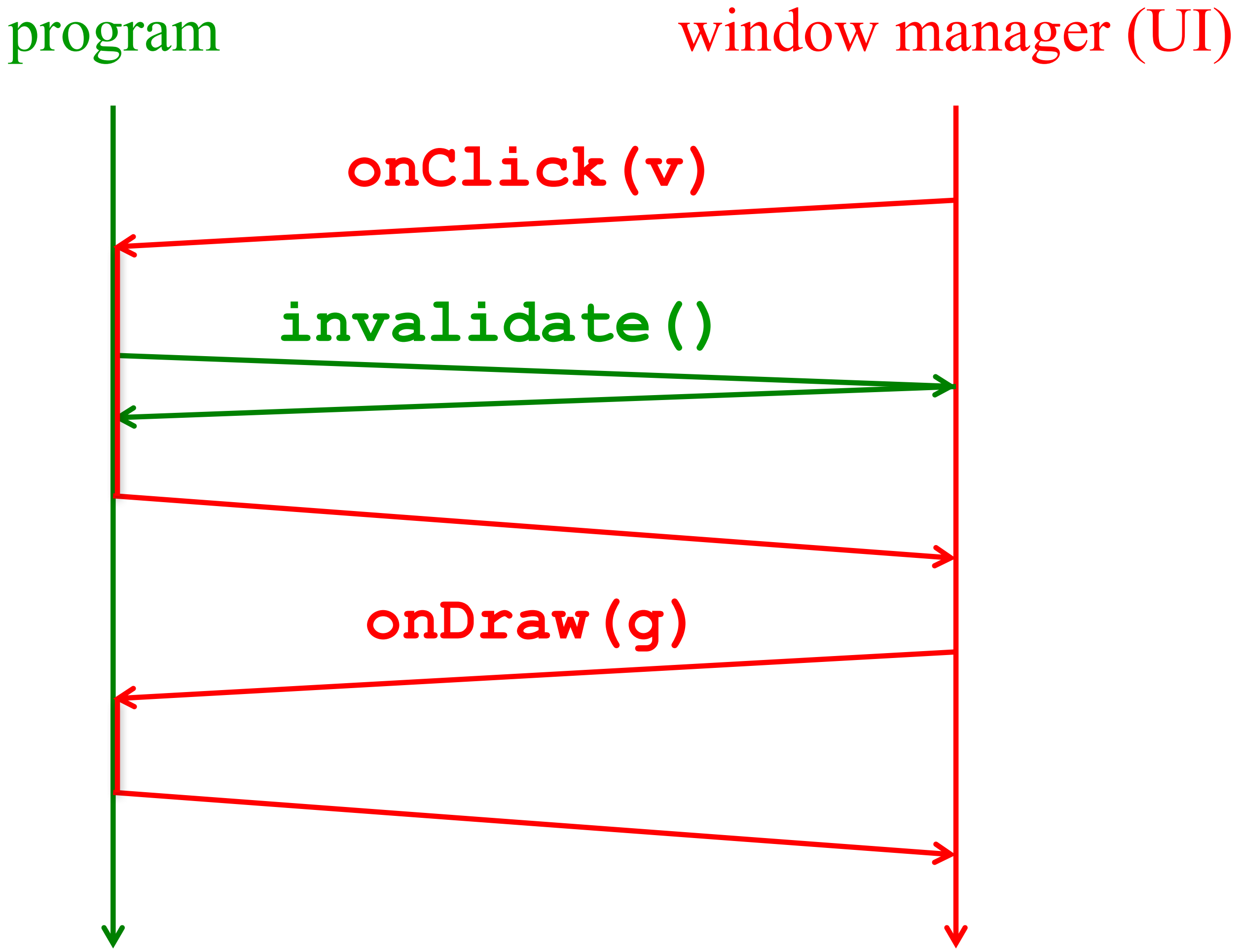
- Always override **`onDraw(canvas)`** if you want to draw on a component
- Always call **`super.onDraw(canvas)`** first
- ***NEVER, EVER, EVER*** call **`onDraw`** yourself
- Always paint the entire picture, from scratch
- Use **`onDraw`**'s **`Canvas`** parameter to do all the drawing. ***ONLY*** use it for that. Don't copy it, try to replace it, or mess with it. It is quick to anger. (You can reuse **`Paint`** objects though!)
- **DON'T** create new **`Canvas`** 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

Event handling and repainting (Swing)



Event handling and repainting (Android)



What’s next – and not

You’re on your own to explore all the wonderful widgets in Swing/AWT and Android.

- Have fun!!
- (But don’t sink huge amounts of time into eye candy)
- If you’re unsure what components to include, start reading the Android/Swing or Android API to see what’s available!

Larger example – bouncing balls

A hand-crafted MVC application. Origin is somewhere back in the CSE142/3 mists. Illustrates how some swing GUI components can be put to use.

Disclaimers:

- Not the very best design (maybe not even particularly good)
- Unlikely to be directly appropriate for your project
- Use it for ideas and inspiration, and feel free to steal small bits if they *really* fit

Enjoy!

IRL: threading issues

- ballSim is multithreaded
  - one thread runs the simulation
    - updates the model periodically with new ball positions
  - one thread displays the UI
- easier to **just use one thread**
  - can use **`javax.swing.Timer`** to be called periodically
  - just make sure the work is done quickly (e.g., <100ms)
  - (ballSim is not really thread-safe as written)
- if you use multiple threads: do not call UI methods from the other (non-UI) threads
  - one exception: **`repaint`** is (supposedly) thread safe
  - use **`javax.swing.SwingUtilities.invokeLater`** to schedule work to run on the UI thread