
CSE 331
Software Design & Implementation

Hal Perkins
Winter 2013
Java Graphics & GUIs

The plan

Today: introduction to Java graphics and Swing/AWT class libraries

Then: 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.

Credits: material ~~stolen~~ adapted from many places; including slides and materials by Ernst, Hotan, Mercer, Notkin, Perkins, Stepp; Regis, 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)

A good book that covers this (and much more): *Core Java* vol. I by Horstmann & Cornell (but if you've got another favorite, go for it)

Why study GUIs?

It's how the world works!!

Classic example of using inheritance to organize large class libraries

Work with a ~~large~~ huge API – and learn how (not) to deal with all of it

Many core design patterns show up here: callbacks, listeners, event-driven programs

It's cool!! It's fun!!!!

What not to do...

There's way too much to know all of it

Don't memorize – look things up as you need them

Focus on the main ideas, fundamental concepts

Don't get bogged down implementing eye candy

The (more detailed) plan

Organization of the AWT/Swing library

Graphics and drawing

Repaint callbacks, layout managers, etc.

Handling user events

Building GUI applications

MVC, user events, updates, &c

A very short history (1)

Graphical user interfaces have existed in Java since 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)

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

Basic idea: underlying system only provides 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

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

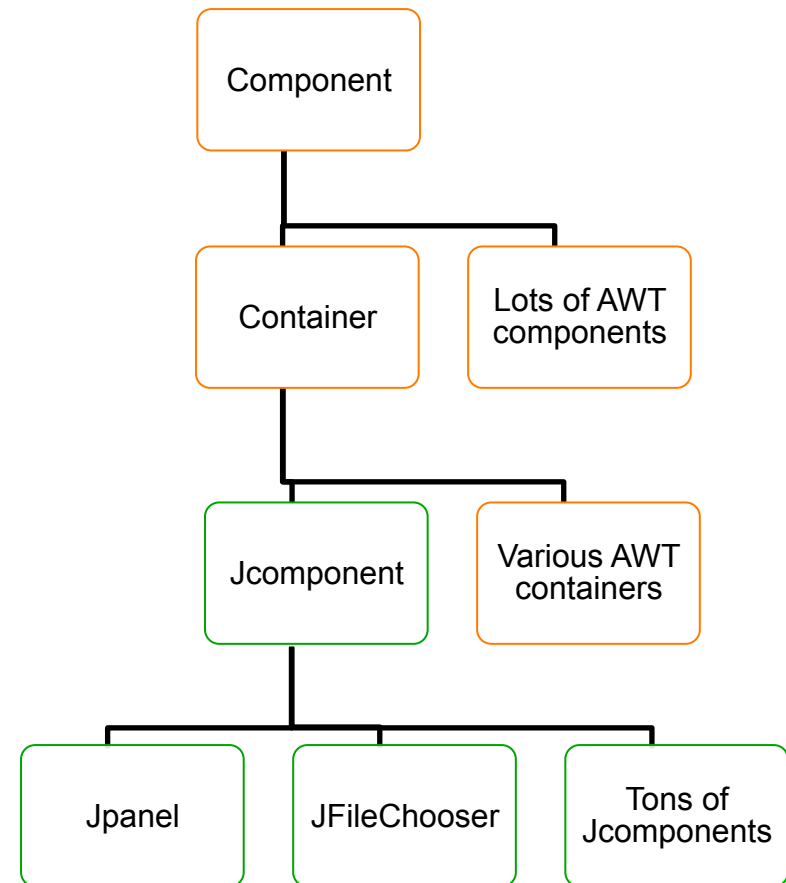
container: A component that hosts (holds) components

examples: panel, box



Component & container classes

- Every GUI-related class descends from **Component**, which contains dozens of basic methods and fields
- “Atomic” components: labels, text fields, buttons, check boxes, icons, menu items...
- Many components are **Containers** – things like panels 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. Ex: **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
 - 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

setDefaultCloseOperation (**int** *what*) – what to do on window close. **JFrame.EXIT_ON_CLOSE** terminates application when window closed.

setSize (**int** *width*, **int** *height*) – set size

add (**Component** *c*) – add component to window

setVisible (**boolean** *v*) – 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 (...))
```

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)
```


Layout managers

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

Answer: each container has a **layout manger**. 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....

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

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 typically inherit an empty `paintComponent` and can override it to draw things

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)

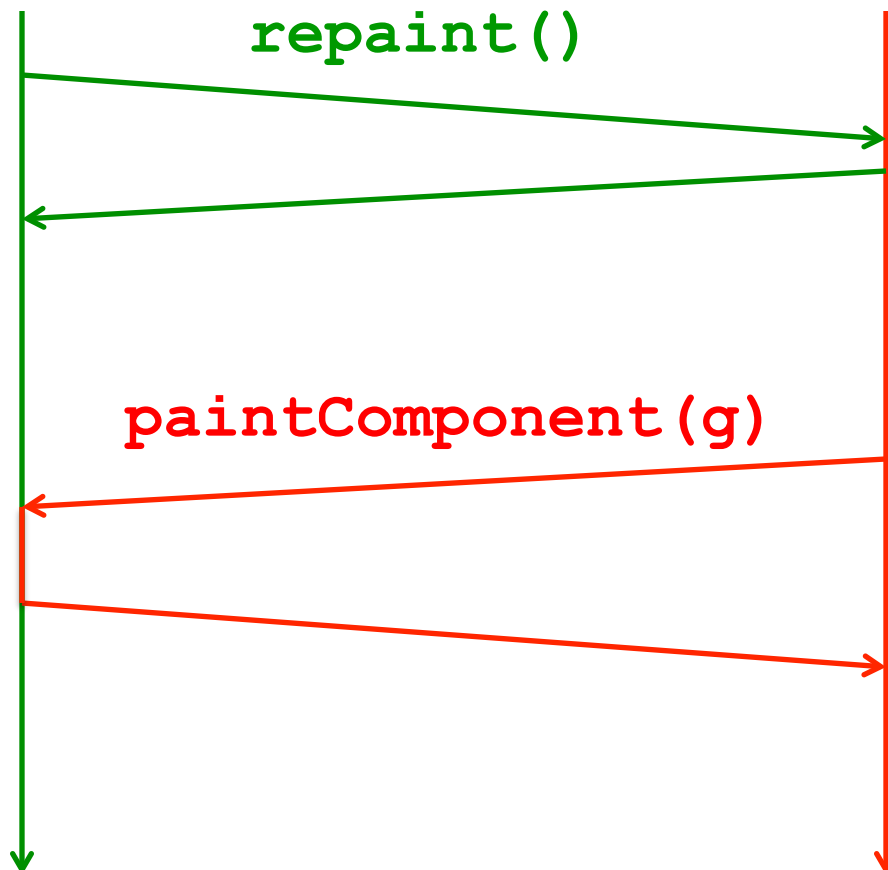
Example

FaceMain.java

How repainting happens

program

window manager (UI)



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, 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 you aren't there yet.

What's next – and not

Major topic next time is how to handle user interactions

We already know the core idea – it's a big-time application of 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)