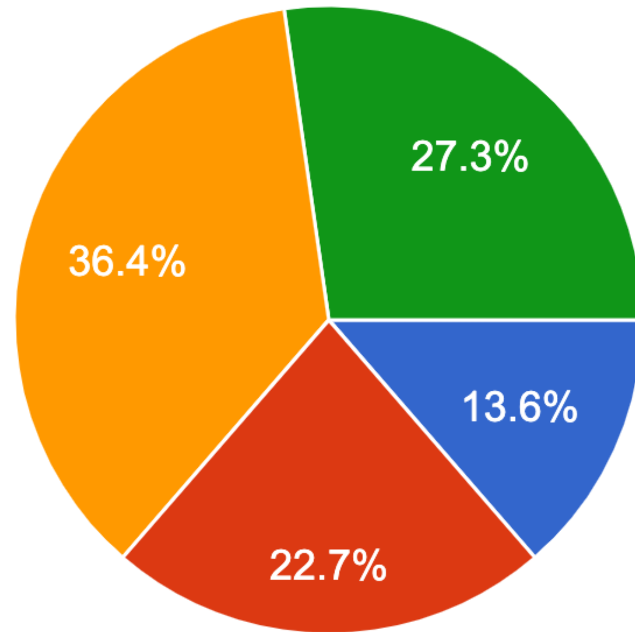
GEOG 178/258
Week 7:

Polygons, GUIs, draw**

mike johnson

What would be the most useful section?

22 responses



- Standard Slides and Live coding until we run out of time (I guess well get through 2)
- No slides and live coding until we run out of time (I guess well get though 3)
- Slides and talking through examples until we run out of time (well get through 3)
- No slides and talking through all examples (well get through 4)

Polygons:

In the above assertions, interior, closure and exterior have the standard topological definitions. The combination of (a) and (c) makes a Polygon a regular closed Point set. Polygons are simple geometric objects. Figure 11 shows some examples of Polygons.

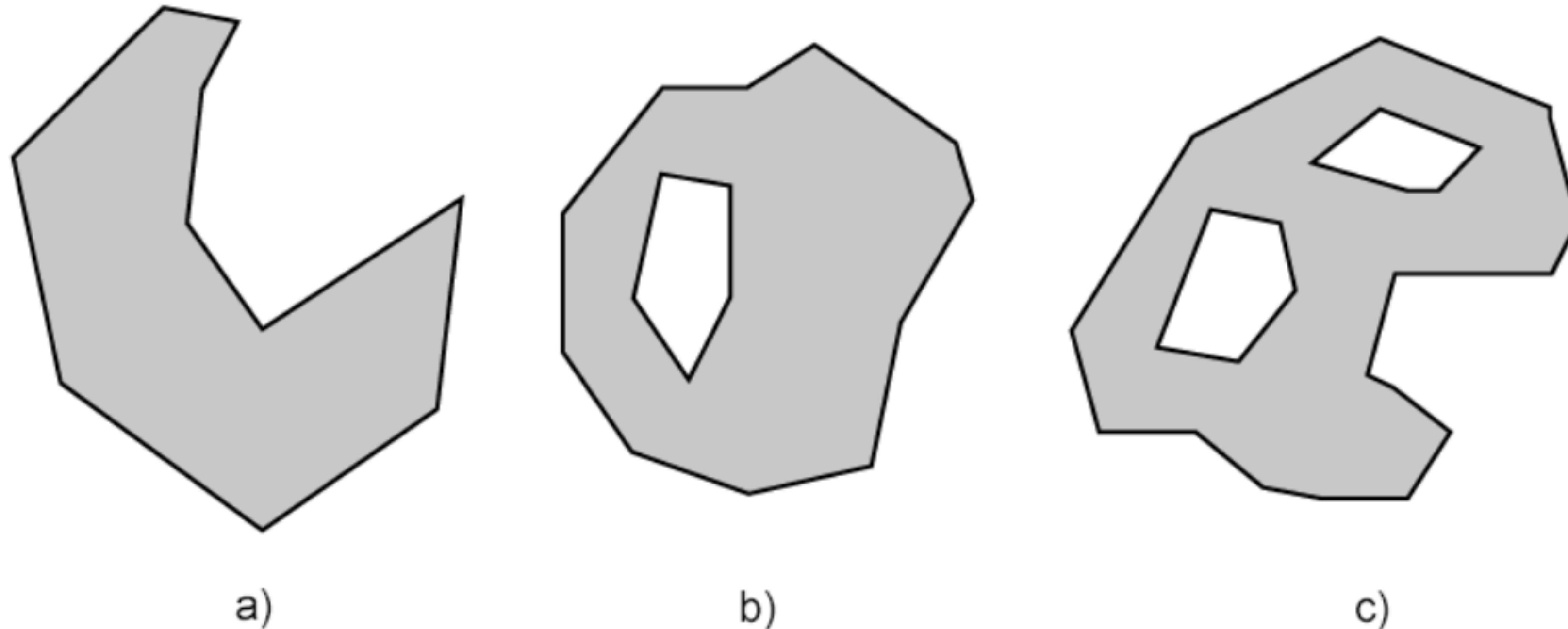


Figure 11: Examples of Polygons with 1 (a), 2 (b) and 3 (c) Rings, respectively

6.1.11.1 Description

A Polygon is a planar Surface defined by 1 exterior boundary and 0 or more interior boundaries. Each interior boundary defines a hole in the Polygon. A Triangle is a polygon with 3 distinct, non-collinear vertices and no interior boundary.

The exterior boundary LinearRing defines the “top” of the surface which is the side of the surface from which the exterior boundary appears to traverse the boundary in a counter clockwise direction. The interior LinearRings will have the opposite orientation, and appear as clockwise when viewed from the “top”,

The assertions for Polygons (the rules that define valid Polygons) are as follows:

a) Polygons are topologically closed;

b) The boundary of a Polygon consists of a set of LinearRings that make up its exterior and interior boundaries;

c) No two Rings in the boundary cross and the Rings in the boundary of a Polygon may intersect at a Point but only as a tangent, e.g.

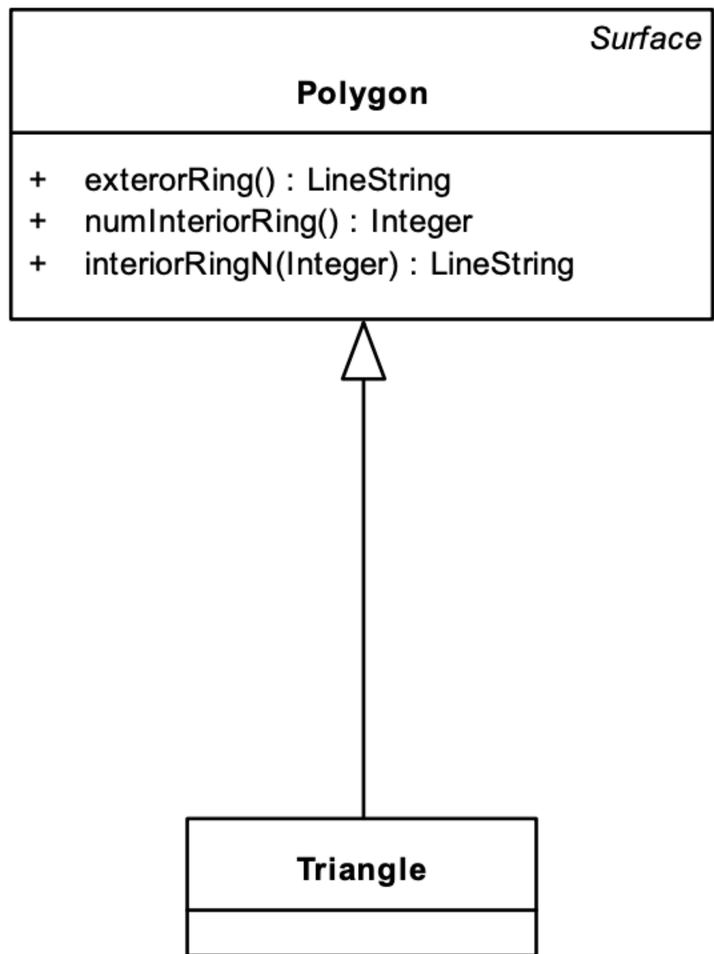


Figure 13: Polygon

*Surface Interface
* Inherits from Triangle

6.1.11.2 Methods

— **ExteriorRing** ( ): LineString — Returns the exterior ring of *this* Polygon.

— **NumInteriorRing** ( ): Integer — Returns the number of interior rings in *this* Polygon.

— **InteriorRingN** (N: Integer): LineString — Returns the Nth interior ring for *this* Polygon as a LineString.

```
WKBTriangle {
  byte              byteOrder;
  static uint32     wkbType = 17;
  uint32            numRings;
  LinearRing        rings[numRings]}

WKBTriangleZ {
  byte              byteOrder;
  static uint32     wkbType = 10 17;
  uint32            numRings;
  LinearRingZ       rings[numRings]}

WKBTriangleM {
  byte              byteOrder;
  static uint32     wkbType = 20 17;
  uint32            numRings;
  LinearRingM       rings[numRings]}

WKBTriangleZM {
  byte              byteOrder;
  static uint32     wkbType = 30 17;
  uint32            numRings;
  LinearRingZM      rings[numRings]}
```

If we use the **extends** keyword

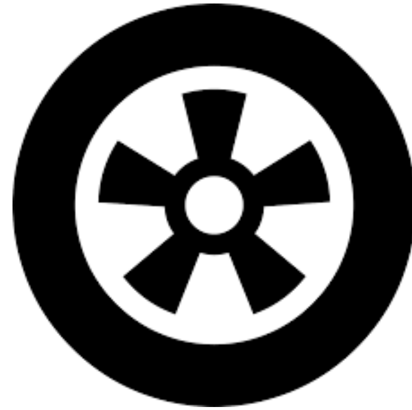
We are **inheriting (like genes)** the classes and methods from a parent class

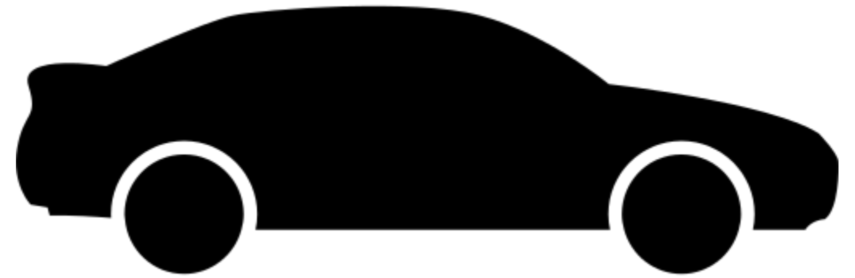
If we use the **implements** keyword

We are **defining a contract that must be meet**

Classes and Objects Define Pieces of Code that we can use

When java compiles we can build instances of classes:





Main methods tell Java how to compile elements into something that runs, prints, executes, ect

Up until now we have been piping all of our output – as text - to the console using System.out.print*

instead, we want to direct our output to a new graphic window using Java swing components

Here is a nice picture

It is static

It is an arrangement of parts including:

A) A racoon
B) Text

This is a button

It is part of the picture WRT to what we want to see

To function as a button:

It must also listen to the picture (clicks)

What happens when we click the button must also be define!

Actions are a common entity in Java GUIs. Therefore to ensure consistency. listeners are added by implementing
The **actionListener** interface

A picture without a home can not be displayed.

If we want to display a picture on the computer we need a frame to hold it

Just like a real frame,

We need to ”pack” our picture in the frame,

And

Mount it on the wall (make visible)

If a change is instigated in the picture,

We don’t need to take down the frame, unpack the elements, and re-draw...

We simply need to **repaint**!

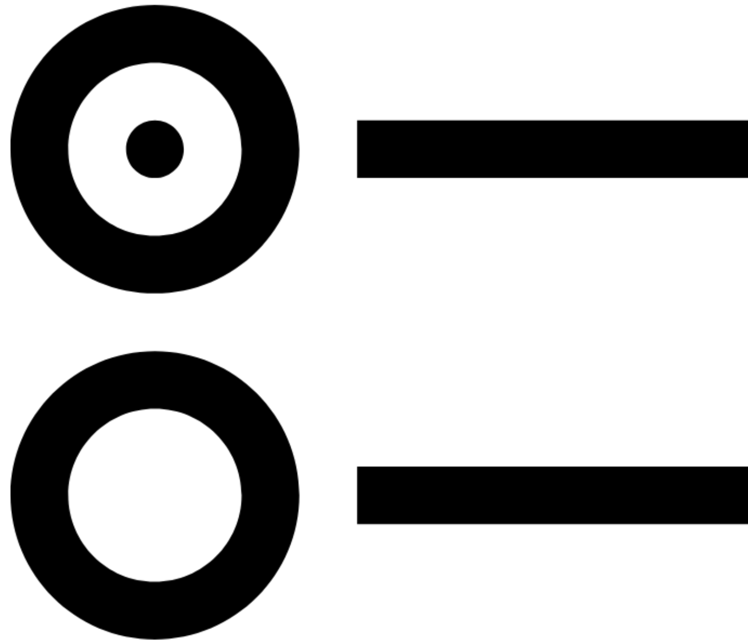
So lets think about the whole system

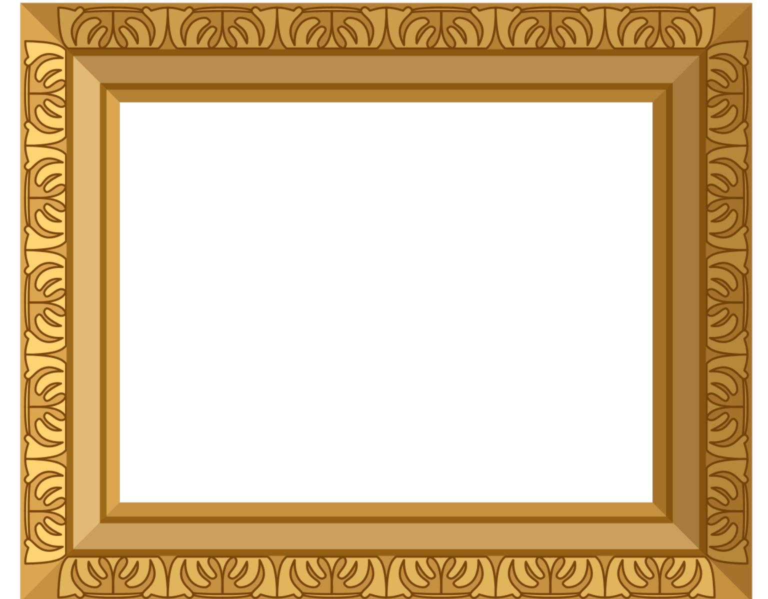
- We need a panel (class with member variables!)
- We need to define how the panel is painted
- We need to add buttons (or any Jcomponent) to the panel with appropriate listeners
- We need to define what the buttons do by syncing Boolean conditions (more on that in examples)
- We need to load an instance of our panel into a frame within a main method.

Buttons
Listeners

Panel
paintComponent

Frame
Packed panels,
visibility option





Buttons are both **part** of panels AND **listen** to panels
Panels are **painted** (or **repainted**) and go in frames
Frames pack panel objects, have visibility, and need to be able to close

Now how do we paint a Panel?

We rely on the draw* methods of the Swing

Lets look at one here:

fillRect

```
public abstract void fillRect(int x,
            int y,
            int width,
            int height)
```

Fills the specified rectangle. The left and right edges of the rectangle are at `x` and `x + width - 1`. The top and bottom edges are at `y` and `y + height - 1`. The resulting rectangle covers an area `width` pixels wide by `height` pixels tall. The rectangle is filled using the graphics context's current color.

Parameters:

`x` - the *x* coordinate of the rectangle to be filled.

`y` - the *y* coordinate of the rectangle to be filled.

`width` - the width of the rectangle to be filled.

`height` - the height of the rectangle to be filled.

See Also:

`clearRect(int, int, int, int)`, `drawRect(int, int, int, int)`

Unfortunately this is pretty obnoxious compared to our geometry objects, so lets look at a trick to make our lives easier through examples!!

Assignment

- Display an open 'polygon' on your panel and implement a button that will perform a **buffer snap** if the first and last points are within a given **threshold** (to correct overshoots and undershoots)
- Display **geometries** on your panel and implement a button that generates (and displays) a **bounding box** around **each** individual geometry (GEOG 178/258) and **all** geometries. (GEOG 258 only)
- Develop your code in a way that **hides** the specific Java graphics details (i.e., don't use Poind2D, etc for your model classes).
- Read chapter 16, 9, and chapter 15 (only the part on events)
- Explain in 2-3 sentences what null is.
- Upload a zip file [LN1W5.zip] with the *.java files to GauchoSpace.
- Assignments and **executable** programs are due the day before lecture at **5pm PST** of each week.

Example 2

Example 3

Example 1

Remember we saw this in our points!!