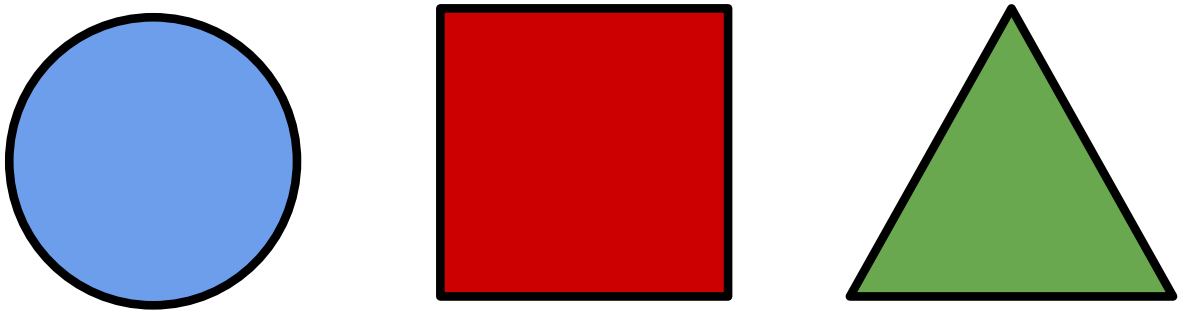
Sketchy

Help Session



Reminders

Code Incrementally!! - steps here

Read the handout - it has a *lot* of resources!

Reminder: Fill out form!

Please fill out the Intended Final Project Form if you haven't already!

- This form is ***not binding!***
- You can complete any final project regardless of what you indicate on the form

Overview

- Drawing lines
- Shapes
  - Creating, storing, and manipulating
- Enums
- Command Pattern
  - Adding, undoing and redoing commands
- Files
  - Opening and closing
  - Saving and loading
- Extra Credit!

Read the Handout

- The handout is long!
- Make sure to read it thoroughly before writing any code.
  - Check out the section on incremental coding if you need help getting started!
- Refer back to specific sections as you code … you'll find there are ***a lot*** of helpful hints

Drawing (1/4)

- You'll need to model a line that the user can draw
  - `javafx.scene.shape.Polyline` represents connected line segments between points - it's a curved line made up of many small straight line segments
  - similar to the `Polygon` class, which also takes in a list of coordinates, but not connected

Drawing (2/4)

- You'll want to make a class `CurvedLine` which stores a `Polyline` instance
- Notes on `Polyline`s:
  - Store an array of doubles representing points
  - `{1,2,3,4} -> one line segment from (1,2) to (3,4)`
  - `getPoints()` returns the current array of points
- Now, adding user input: the `Pane` that represents your canvas should have a handler that fires when the user interacts with it through the mouse

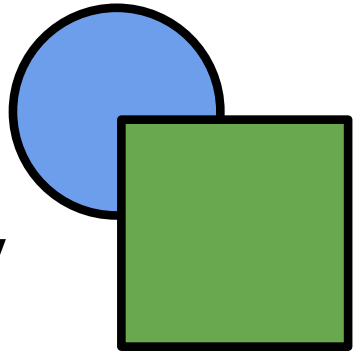
Drawing (3/4): User Input

- Your handler should look something like this:
  - *When the mouse is pressed:*
    - Instantiate a new `CurvedLine` and store a reference to it so you can add points when the mouse is dragged
  - *When the mouse is dragged:*
    - Add a point to the `Polyline` by adding the x and y location of the current mouse point to the list of points of the `Polyline` contained in the `CurvedLine`

Drawing (4/4)

- You'll need to store all of your `CurvedLine`s ~somehow~ so that you can save them to a file and add them back to the canvas when loading
  - More on saving lines later!
- Unlike shapes, you DO NOT need to be able to rotate/resize/translate/select lines that you have drawn
  - though this may be done for extra credit

Making Shapes (1/2)



- You'll need to model shapes in some generic way

- Reasons to do this:
  - You'll need to keep a list of shapes that you can iterate through so you can properly select the current shape
  - Also, you'll need to be able to save any type of shape to a file
  - The commands that users perform on shapes will apply to ***all*** shapes

Making Shapes (2/2)

- How can we generically model shapes?
  - Polymorphism!!
  - Use an `interface`!
    - All of your shape classes will implement a `SketchyShape` interface
  - Why an interface and not an abstract superclass?
    - JavaFX `Shape`s all define their locations differently, which makes manipulation different for each type of shape
      - It will be much easier for the shape classes to define the manipulation methods that are specific to the JavaFX `Shape` that they contain
    - For this project, some repetitive code in your ***shape*** classes is okay!
    - Using an abstract class is certainly possible, but it leads to a somewhat awkward/complicated design that we will not explore in class

Storing Shapes (1/2)

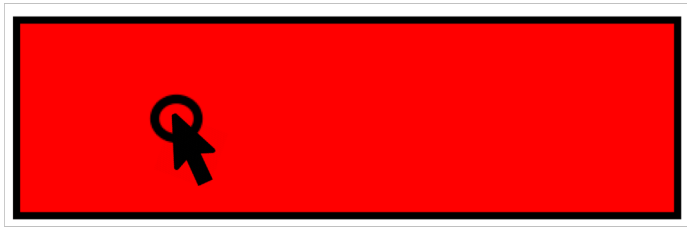
- In previous projects, like DoodleJump, you needed to use a data structure to store a possibly infinite number of shapes
- We will need to do this in Sketchy as well, since you will need to keep track of *all* shapes that you make

Storing Shapes (2/2)

- Does the data structure need to be ordered / sorted?
  - Yes! It must reflect the layering of shapes in the canvas `Pane`
    - How exactly? Stay tuned...
- Think about which data structure would best fit our needs:
  - Unknown/unlimited number of objects
  - Need to be able to iterate through every shape
  - We've used a data structure like this before...

`contains()`: Selecting a Shape (1/4)

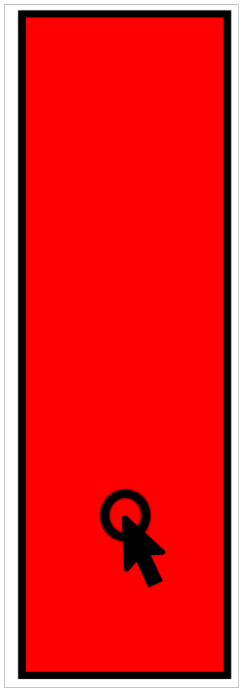
- To allow the user to select a shape, you'll have to check to see if the point they clicked on is contained within the shape

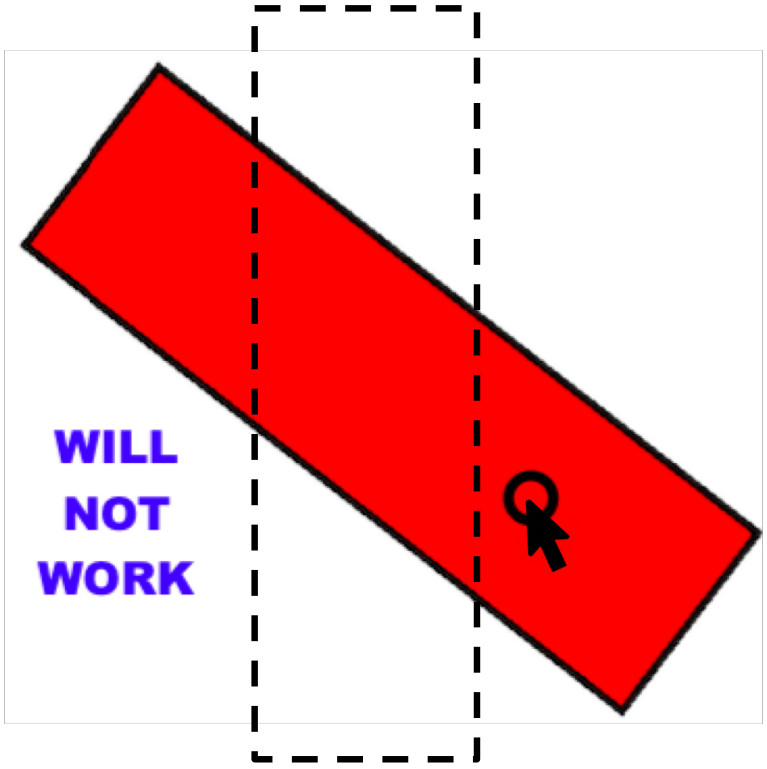


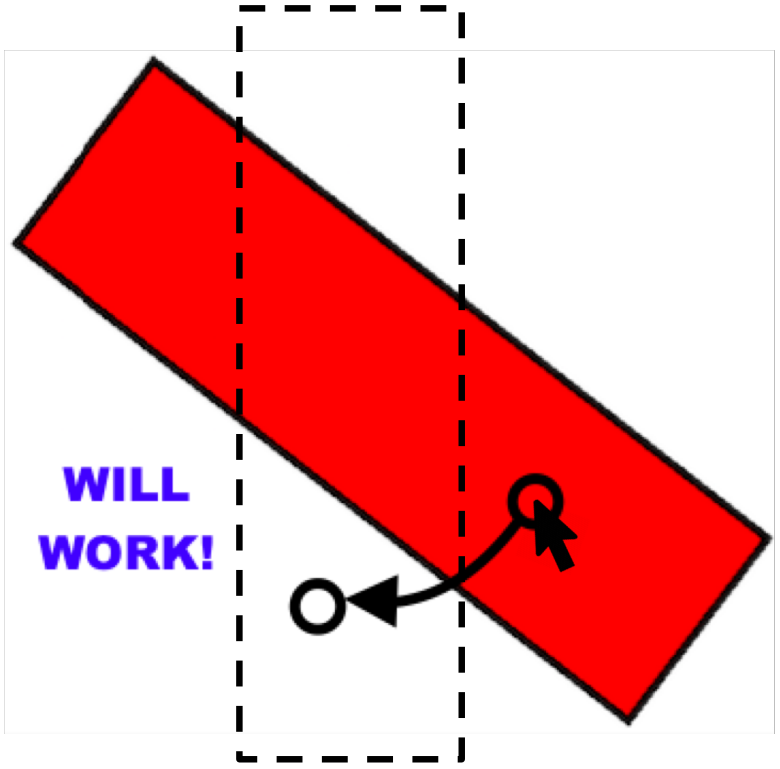
`contains()`: Selecting a Shape (2/4)

- Use `Node`'s `contains()` method
- However, this method only checks the shape's *original* bounds
  - Does ***NOT*** taking rotation into account!
- To resolve: When you are checking for whether a rotated shape contains the point (by mouse input), you first rotate the point!
  - but by how many degrees?

`contains()`: Selecting a Shape (3/4)



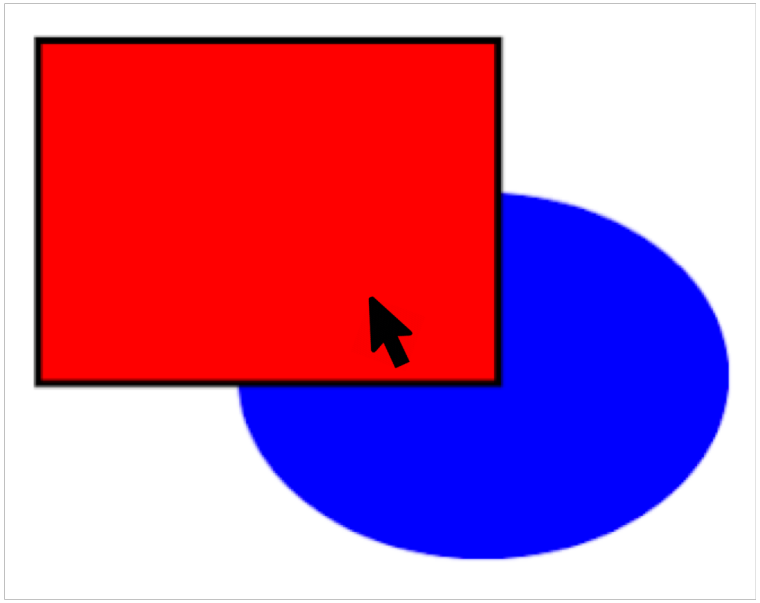




`contains()`: Selecting a Shape (4/4)

- Lastly, you'll want to ensure that if two shapes are overlapping, the shape that's *on top* is the one that's actually selected

- Hint: Use your shapes list!



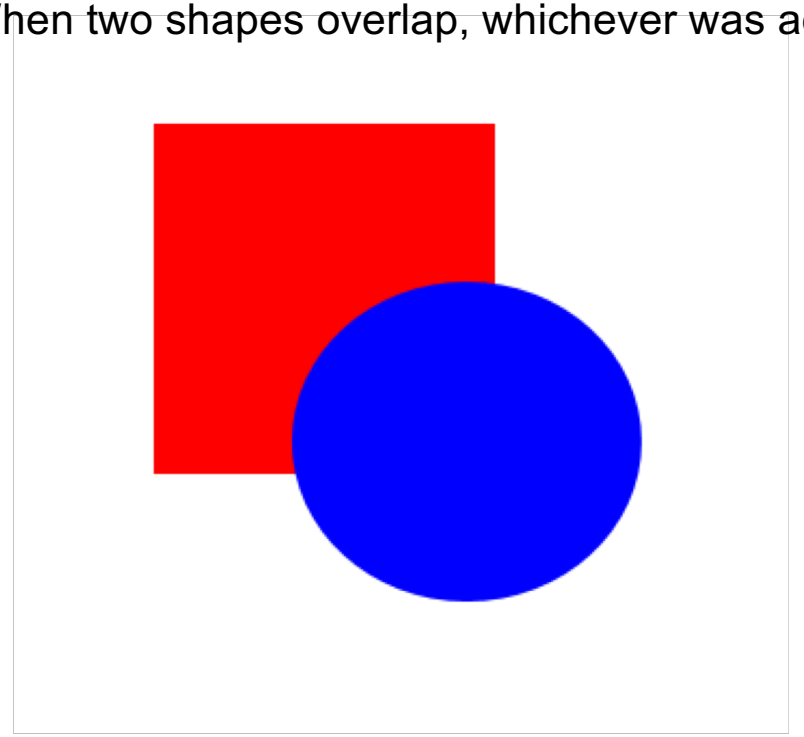
Shape Manipulation

- The handout goes into great detail with regard to shape manipulation
  - By getting information about the changes in the mouse point's location, we can move, resize, and rotate the shape
  - We've provided most of the math/pseudocode necessary to understand the implementation of these methods

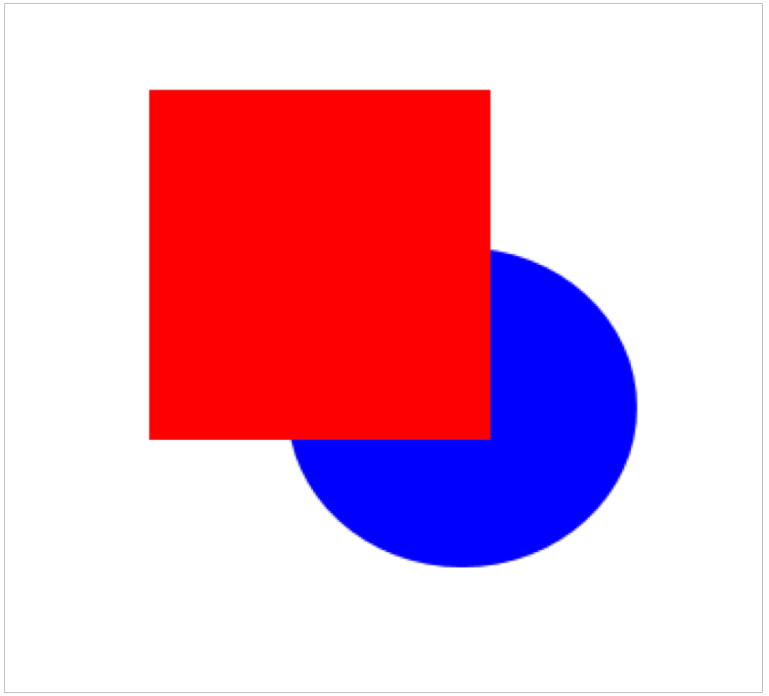
- You'll need to handle
  - Translation of the shape from one point to another point
  - Rotating the shape around its center of rotation
  - Scaling the shape around a fixed central point

Raising and Lowering Shapes (1/4)

When two shapes overlap, whichever was added to the Pane's children list LAST is rendered on top!



`getChildren()` : {rectangle, ellipse}



`getChildren()` : {ellipse, rectangle}

Raising and Lowering Shapes (2/4)

- Must be able to move a shape one layer forwards or one layer backwards
- How do we change the visual layering of shapes on our `Pane`?
  - Move it forwards or backwards by 1 index in the `Pane`'s list of children!
  - More hints for this in the handout!

Raising and Lowering Shapes (3/4)

- Graphical ordering of shapes should be reflected by the logical ordering of shapes in your data structure(s)
  - Make sure that you change the position of a shape in your list of shapes (and any other lists) if it is raised or lowered
    - If a shape is raised or lowered, what should its position in the pane's list of children change to?
- A correctly ordered list is necessary for selecting the correct shape with the mouse
  - The order of shapes should be changed in all relevant lists
- It is also important for saving/loading shapes with the correct layering
  - More on this in the saving/loading section!

Raising and Lowering Shapes (4/4)

- We need to keep track of and change a shape's index in the list of children, the shapes list, and possibly another list....
  - what's a good way to model this?
- A `Layer` class!
  - keeps track of several indices using instance variables, with setters and getters for each one:
    - The index of the shape in the pane's list of children
    - Its index in the shapes list and any other lists you have
- Useful method to write: `moveShapeToLayer(Shape, Layer)`
  - changes a shape's indices in all relevant lists to those stored in the layer object - can be used for both raising and lowering!

Reminders

Code Incrementally!! - steps here

Read the handout - it has a *lot* of resources!

Enums - SelectOption (1/2)

- An `Enum` can be used to keep track of what option (radio button) the user currently has selected (e.g. Draw Rectangle, Draw Line, etc.)
- We could use an `int` variable: `_option = 1` when the user is in “Select Shape” mode, `_option = 2` for “Draw Line”, and so on
- An `Enum` is very similar to this, but all of the options will have names instead of numbers - more readable!
- You create an enum in its own file, like a class or interface

Enum - SelectOption (2/2)

- In its own file, an example of an enum is:

```
public enum Direction { NORTH, SOUTH, EAST, WEST; }
```

- Using this enum:

```
Direction currDirection = Direction.SOUTH;//how to set an enum value

switch(currDirection) { //a switch statement comparing the enum value

        case NORTH:

                /*code elided*/

}
```

- Similar to using 0, 1, 2, and 3 to represent different directions; more readable

The Command Pattern

- The *command pattern* is one of the central design concepts in Sketchy. You will be using this pattern to implement the unlimited undo/redo feature!
- Some topics to cover:
  - Commands and Sketchy
  - Modeling Commands in Java
  - The Document History
  - Quick Stack Review
  - Example: Adding a Command
  - Example: Undo-ing a Command
  - Example: Redo-ing a Command

Commands and Sketchy

- What constitutes a command?
  - A command corresponds to an action that is performed by the user
  - You should have a class for each command
- What are the commands in Sketchy?

Create a shape	Fill a shape
Draw a line	Delete a shape
Move a shape	Raise a shape
Resize a shape	Lower a shape
Rotate a shape	

- Everything that the user can do should be undoable and redoable except for saving and loading

Modeling Commands in Java (1/2)

- The Command pattern lends itself to *polymorphism*
- We can model a generic command as an interface or an abstract superclass
  - It will **declare** (but not define) the common properties and capabilities of commands, in particular, the *redo* and *undo* capabilities
  - The actual implementation of these capabilities will be left to the implementing/extending classes (i.e. the concrete commands)

Modeling Commands in Java (2/2)

- Specific commands are represented by **classes** which implement/inherit from the generic `Command`
  - Will know about the properties specific to a particular command
- Example: the move shape command
  - Will know about additional properties specific to move shape action:
    - The target shape
    - The before and after locations of the target
    - redo() - will set the location of the target to after
    - undo() - will set the location of the target to before

What to do with these Commands? (1/2)

- They allow us to construct the document history
  - This is a “log” of all actions that have been performed since the document was created/opened
  - Similar to a web browser’s history
- The undo/redo feature allows the user to move backward/forward through the history

What to do with these Commands? (2/2)

- The history has two major components:
  - The commands that were performed before the point where the user is now
    - We'll call these the *undo* commands
  - The commands that were performed *after* the point where the user is now
    - We'll call these the *redo* commands
- So we have two distinct collections of commands. What kind of structure should we use to organize them?
  - We want for the most recently (last) performed command to be the first one that is undone (*LIFO* order)

Quick Stack Review (1/2)

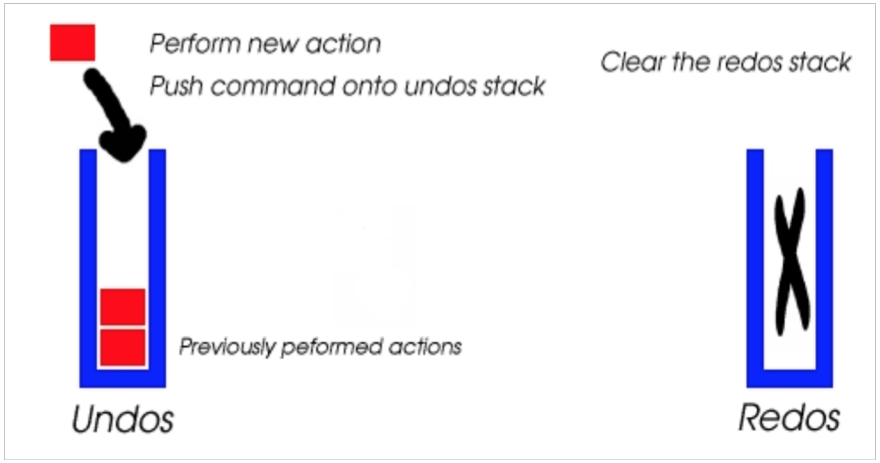
- How does a stack work?
  - “LIFO” (Last-in, first-out) data structure
- Analogy: The Ratty plate dispenser is a real-life stack
- Why use a Stack?
  - The stack is a fairly simple data structure, so why not use something more sophisticated (e.g. an `ArrayList`)?
  - Answer: The stack closely models how actions are done/undone in our program. Nothing more complex is necessary!
  - You can use `java.util.Stack` for this!

Quick Stack Review (2/2)

- The Command Stack
  - We need to ensure that only `Command`s are pushed to and popped from our stack
  - How? Generics!
  - ex. `Stack<String> stringStack = new Stack<String>();`
- Consult the Stacks lecture for more!

Example: Adding a Command

- What steps should we take when an undo-able action is performed by the user?
  - Instantiate a command representing this action
  - Push this action onto the undo stack
  - Clear the redo stack, since the user can no longer redo (think about the web browser analogy)

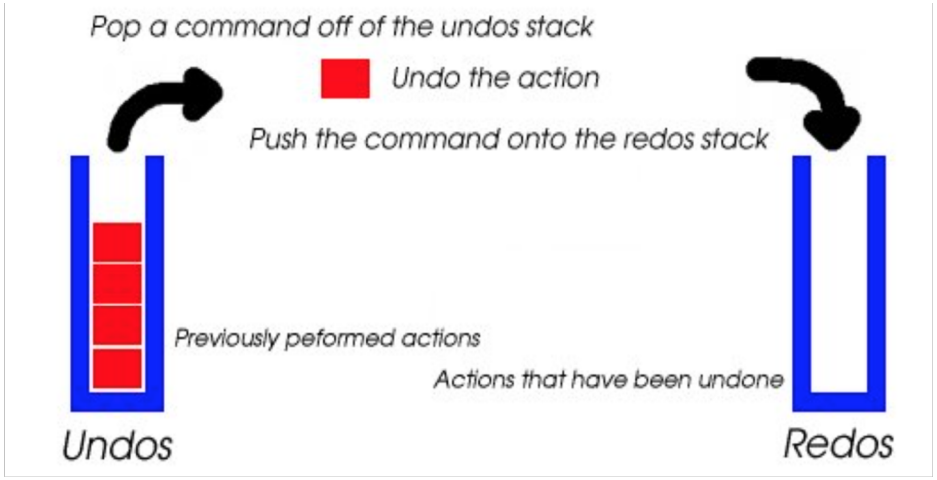


Note: some actions are trickier to model than others

- A simple one: Creating a shape
  - Instantaneous: Triggered by a single mouse click or button press
- A harder one: Moving a shape
  - Triggered by the mouse dragging - the action occurs over an arbitrary period of time

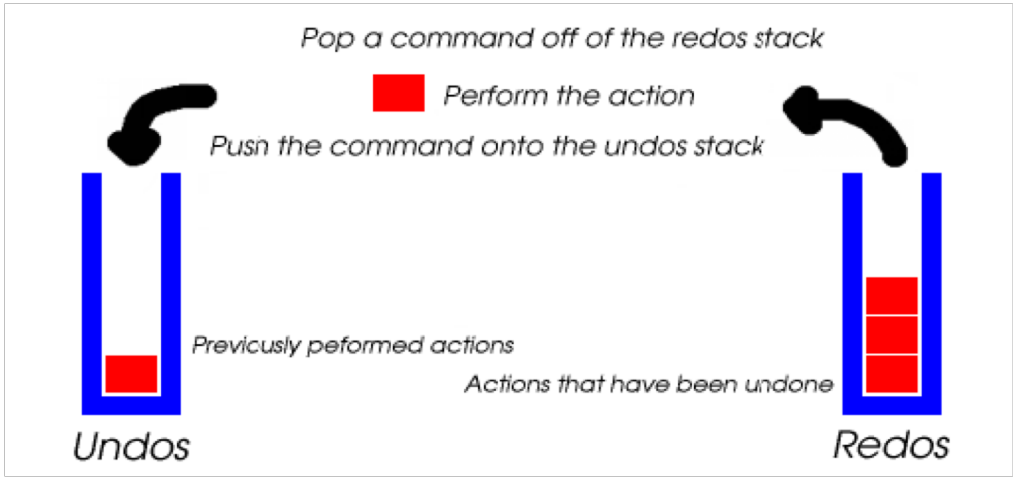
Example: Undo-ing a Command

- What happens when the user calls undo?
  - First, check that the undo stack isn't empty (or, you can have the undo button disabled whenever the undo stack is empty -- even better!)
  - Pop a command from the undo stack
  - Call the `undo()` method on this command
  - Push the command onto the redo stack



Example: Redo-ing a Command

- What about when redo is performed? How is it different from undo?
- It is the reverse of undo:
  - First, make sure that the redo stack isn't empty
  - Pop a command off the redo stack
  - Call the `redo()` method on this command
  - Push the command onto the undo stack



How to save and load from Sketchy

How do we transform the shapes and lines from our screen into a text file? How do we transform them back?

- How do we go from an internal data format (visual shapes/lines) to an external file format?
- How do we go from an external file format to the internal data format?
- *Generically*: how do we go between a text format and a data format?
- This is a problem we see a lot in computer science
  - What concept can we use to address this?
  - Time for a brief discussion of....

Parsing

- Parsing is one of the most common concepts in computer science -- it's used all over the place!
  - Parsing is the process of taking human-readable text or strings and turning them into data or instructions that a computer can understand and execute

- If you take more computer science classes you will do more parsing:
  - `cs123`: Sceneview
  - `cs33/167`: Shell
  - `cs32`: Maps/Traffic

Internal Data vs. External Files

- In Sketchy, you will need to move between your **internal data format** (shapes/lines) to an **external format** (text) and vice versa
- Before you can think about moving between these two formats, you need to make sure that both formats are properly defined
  - **Internal format:** Shapes/lines shown on screen and stored in a data structure
  - **External format:** Text file listing shapes/lines and their various attributes

Suggested file format for Sketchy (1/4)

- Your file format will need to contain all of the information about the attributes of your your shape objects and line objects
- What attributes do shapes have?
  - Type of shape (e.g. `Rectangle`, `Ellipse`)
  - Location
  - Dimensions
  - Rotation
  - Color
- What attributes do curved lines have?
  - List of coordinates
  - Color
- How will they be stored in your text file? Don't worry, we will explain a bit of how this works in the next slide!

Suggested file format for Sketchy (2/4)

- Remember, these are just our suggestions! Feel free to define your format however you want!
- Need to define a generic format for the attributes specified earlier:

```
<shape> <x-coord> <y-coord> <width> <height> <degrees> <red> <green> <blue>
```

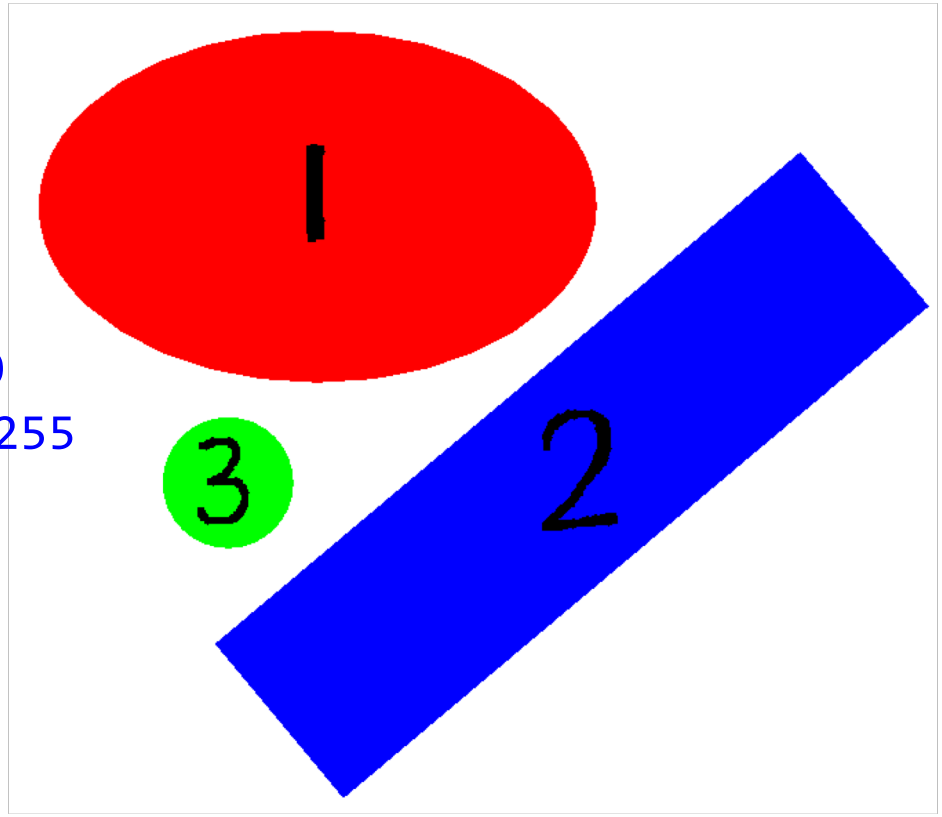
- What would this look like in a file?
- One example:

```
ellipse 100 80 400 250 0 255 0 0
rectangle 210 330 550 140 140 0 0 255
ellipse 190 360 90 90 0 0 255 0
```

- Can you “parse” this by hand? What would the image described by this file look like?

Suggested file format for Sketchy (3/4)

```
1. ellipse 100 80 400 250 0 255 0 0
2. rectangle 210 330 550 140 140 0 0 255
3. ellipse 190 360 90 90 0 0 255 0
```



Suggested file format for Sketchy (4/4)

- Storing lines will require storing all of the points on the line
- Storing a `CurvedLine` might look like this:

```
line <red> <green> <blue> <num-points> <x-coord> <y-coord> <x-coord> <y-coord> … <last x-coord> <last y-coord>
```

- Must store as many x-y pairs as there are in the line
- Saving a single `CurvedLine` may require a lot of space in your text file!

Support.FileIO Object

- Now that a file format has been defined, you can think about moving between internal and external formats
- To ease this transition, we've given you the `cs015.fnl.SketchySupport.FileIO` object! `cs015.fnl.SketchySupport.FileIO` has the following methods:

```
public FileIO();
public static String getFileName(boolean save, Window stage);
public void openRead(String filename);
public void openWrite(String filename);
public String readString();
public void writeString(String stringToWrite);
public int readInt();
public void writeInt(int intToWrite);
public double readDouble();
public void writeDouble(double doubleToWrite);
public void closeRead();
public void closeWrite();
public boolean hasMoreData();
```

Opening and closing files

- When saving a file:

```
io.openWrite(filename);
// code to write to file here
io.closeWrite();
```

When you open a file, what you write will **replace its contents rather than be appended to the end**

- When loading a file:

```
io.openRead(filename);
// code to read from file here
io.closeRead();
```

- Make sure that these methods are the first and last calls when reading from or writing to a file!

But how to get a filename?

- Support code: `FileIO.getFileName(boolean save, Window stage);`
  - This method pops up a save or open dialog (depending on `boolean save`) for the user and returns the filename
  - Call `<Pane>.getScene().getWindow()` on your canvas `Pane` to get its `Window`

- Remember to handle the case where the user hits cancel! (In this case, `getFileName` will return `null`)

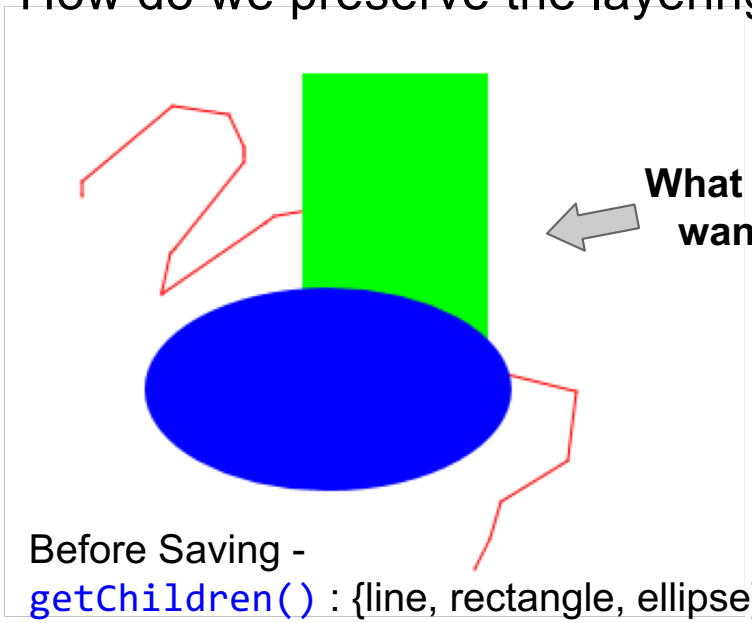
Saving (1/3)

- Saving is the process of going from shape objects to a text file
- You will have to go through all the shapes and write the appropriate data for each to an external file
- We recommend the use of “smart” shapes - shapes handle their own saving
- For each shape and line:
  - pass the `FileIO` object to its save method
  - The shape or line will then know how to save or “print” itself to the file using the output methods of `FileIO`
- How would we print one line of our file?

```
io.writeString(“rectangle”);
io.writeDouble(x);
io.writeDouble(y);
// etc.
```

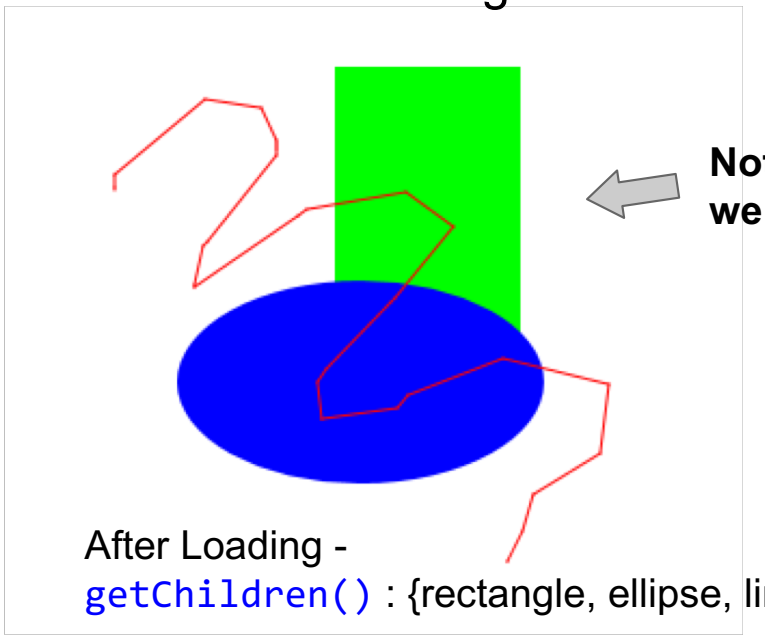
Saving (2/3)

How do we preserve the layering of shapes and lines when saving and loading?



Before Saving -
`getChildren()` : {line, rectangle, ellipse}

VS.



After Loading -
`getChildren()` : {rectangle, ellipse, line}

- Let's say when saving, we wrote all shapes from our shapes list *then* wrote all of the lines to the file
- When loading, the ordering/layering of shapes and lines would **not** be preserved! How do we fix this?

Saving (3/3)

Now that we know how to save a single shape to a file, how do we save all the shapes/lines on our screen?

- Shapes/lines must maintain their layering when saving and loading
- How do we achieve this?
  - `Saveable` Interface
  - Since our canvas' data structure will only hold shapes, and not lines, we will need another data structure that will hold both lines and shapes in their proper layering
- `Saveable` Interface
  - Implemented by shapes and lines
  - Allows us to *polymorphically* store both lines and shapes with their proper layering in a data structure (think generics!)
  - Can iterate through the structure when saving to maintain layering
  - Make sure that you remember to update your `Saveable`s structure when you change layering!

Loading

- Use `FileIO.getFileName(...)` to select a file to load
- Using the file format you defined for saving, parse the strings contained in the file to reproduce the shapes and lines with the proper attributes
  - Since we used our `Saveable`s list to write to the file, the correct layering will be preserved!
- Make sure that you start a new list of shapes and `Saveable`s when loading your file!

Note : This is just one possible design for saving and loading. You are welcome to explore other ways of polymorphically storing shapes and lines!

You too can be a hero, do extra credit!

Sketchy's a lot of fun – especially if you do some extra credit! Here are some great ideas:

- Extra-good UI
- Extra shapes
- Photoshop-like bounding box: instead of resizing shapes from the center, make them resize at the corners by dragging squares (more difficult than it sounds!)
- Be able to control stroke width of lines
- Select, move, rotate, and resize lines
- Be able to control layering of lines
- Animation
- Copy and paste
- Use photos as objects (and be able to interact with them)
- Export as image
- Define layers (like in Photoshop)
- Select multiple objects at once (and do operations on them simultaneously)
- and many, many more!

Tips

- Sketchy is longer than the other projects that you have done so far, so make sure you start early!
- Thoroughly read the handout multiple times and make sure that you understand the requirements to meet **Minimum Functionality**.
- Refer back to specific sections of the handout while you are coding as it contains some very helpful hints
- Sketchy has many different steps, however, some of them are self-contained. If you find yourself stuck on a particular step or are trying to fix a complicated bug for a reasonable amount of time, try to work on something that is not dependent on the functionality of the current step that you are on.
    - For example, you need to get your shapes to appear first in order to work on commands like rotation, translation, resizing, etc. However, if you are stuck on rotating, try to work on undo/redo or saving and loading.
    - However, **don't** accumulate a lot of bugs and wait to fix them at the end of the project.

GOOD LUCK!

~Enjoy Thanksgiving Break!~