HW 2: JavaDraw

Due Midnight ending Wed July 23rd, 2003

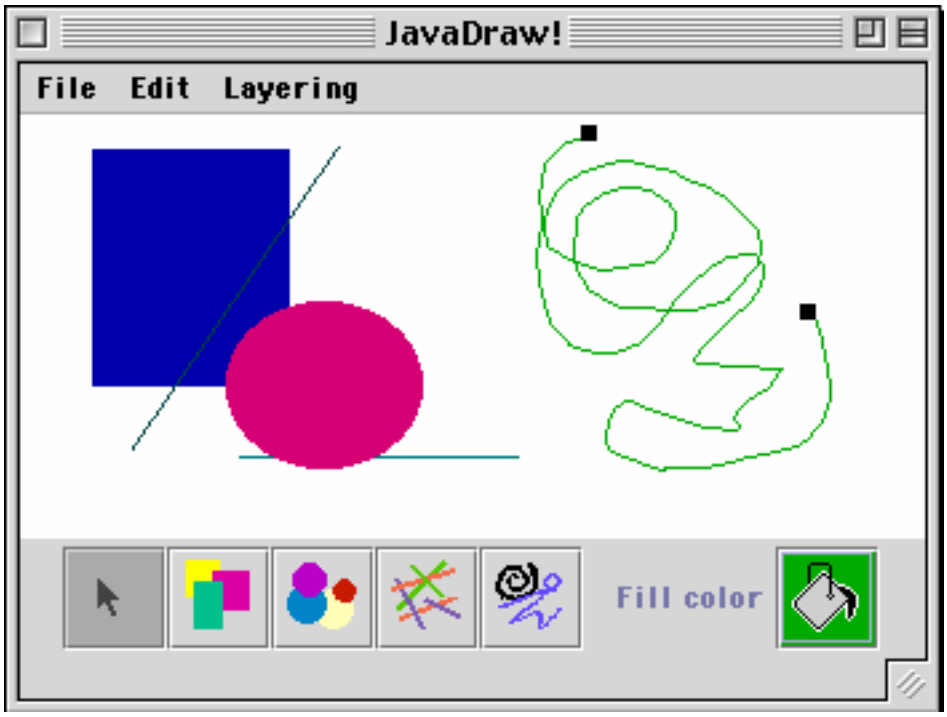
(Thanks to Julie Zelenski for creating this assignment.) Your next assignment will be to construct a simple drawing program that allows the user to draw shapes in various colors, cut and paste shapes, and save and load drawings to disk. In this handout, we will stick to discussing the functional requirements of the assignment. Design hints and development suggestions are given in a separate handout. Be sure to thoroughly read both handouts so you understand both the functional specification and our design expectations. As always, if you find something unclear, please ask! We will have the usual extra office hours in the days before the due date – see the course page.

This assignment will combine two of major OOP themes: using OOP GUI components and the OOP inheritance problem of factoring several related classes to make a little hierarchy. For the Swing part of the assignment, we have seen many of the relevant classes in lecture, but for some classes, you will need to hunt through the Sun docs. The starting project gives you a good headstart on the mundane aspects of setting up the user interface so you don't have to spend too much time on these tedious aspects.

Overall, this assignment is more involved than your last one. Developing a good object-oriented design is an incremental process that often takes several iterations until you settle on a quality solution. Don't handicap yourself by trying to throw the program together at the last minute!

JavaDraw basics

JavaDraw is a simple draw program. In its final form, it has just one window that contains a drawing canvas and a tool bar that supports the creation of rectangles, ovals, lines, and scribbles in various colors. A selection tool allows the user to select, move and resize shapes. There are menu commands to cut, copy, and paste shapes, rearrange layering, and save and load shapes from files.



What you start with

The above description and screenshot may seem daunting—however, don't panic, you won't have to start from scratch. With graphical programs, even more so than with others, it is particularly effective to learn by taking an existing program and extending it as a means of gentle introduction to a complex set of classes. If you compile and run our starting program, you'll find that it can already draw rectangles, select and deselect a rectangle, and move or resize the selected rectangle.

Your job will be to first read our code to observe the basic usage of the GUI classes and then move on to experimenting with additions, gaining confidence and deeper understanding as you go. You may begin with a bit of a "monkey see, monkey do" approach but your skill grows into a more sophisticated mastery over the course of the assignment. You avoid that frustrating initial struggle to just get anything working and you have a good design in place to guide you. Since you start with a working program, if you make small modifications as you go, you can continue to keep a working program each step along the way, which will simplify the testing and debugging work.

In our starting project, there is a JavaDraw main class that constructs the user interface. It creates the canvas window and its internals and installs the menus and their handlers. There is a Toolbar class that configures the radio group of buttons with images for the tools and the current color button that brings up the color chooser dialog. We also give you a starting implementation of the DrawingCanvas and Rectangle classes that handles simple rectangles.

What you need to do

The users want more and your mission is to start trying to satisfy their demanding requests! There are five features that you are going to add to the program:

1. *New shapes.* The most major extension that you will be making is to support three new shapes. The program as given only supports rectangles, you need to extend it to draw ovals, straight lines, and "scribbles" (a squiggle of connected lines that trace a path).
2. *Layering.* As shapes are created, they are added on top any previous shapes, overlapping and covering up those shapes below. In our starting implementation, there is no way to re-arrange the ordering after a shape has been created. The Layering menu contains unimplemented "Bring to front" and "Send to back" commands. These commands need to be implemented to move the selected shape on top or beneath all other shapes.
3. *Colors.* The existing implementation assumes all shapes are filled in dark gray. You will extend the program to create each new shape in the color currently selected on the toolbar. You will also update the toolbar to reflect the current color whenever a shape is selected and allow the user to change the color of the selected shape.
4. *Edit operations.* The Edit menu has unimplemented cut, copy, paste, and delete commands. You will implement operations to store a copy of a shape on the clipboard, paste the clipboard contents, and delete shapes. The clipboard contents will be handled via object cloning.
5. *File operations.* The File menu contains clear, load, and save commands that also require your implementation. The clear command just removes all shapes from the canvas. The save command allows you to store a drawing to a disk file and the load command loads a previous saved drawing. The file operations will be done using object serialization.

(We've covered the material for 1-3 already, so you can start with those. We'll be done with the material for 5 and 6 in one more lecture, so you can wait on those.)

A little more detail

In general, the program is expected to behave in the common and obvious ways that basically all drawing applications do (MacDraw, PowerPoint, Diagram, and so on). Probably all of you are familiar with at least one of these applications and thus you will have an idea of what the goal is. You can also examine the behavior of our starting program to see how create, select, move, and resize behave. The requirement summary at the end of this handout goes into painstaking detail about the required behaviors that can help if you aren't sure of the nuances.

Here is a quick summary of some of the basic behaviors to get you started: The tool buttons allow the user to select the current tool mode in effect. The tool mode determines what happens when the user clicks and drags across the canvas. If the toolbar has a shape tool selected, clicking and dragging on the canvas draws out a new instance of that shape. If the toolbar has the arrow tool selected, mouse actions on the canvas will select, move, and resize shapes. Clicking on a shape selects it and a selected shape is drawn with resize knobs. Clicking and dragging on a shape moves it. Clicking and dragging on a resize knob resizes the shape. All shapes are filled, the color is set at creation and can later be changed with the toolbar's color button. The layering commands move shapes above or below others. Cut, copy, and paste allow you to store and retrieve duplicate copies of a shape on the clipboard. The load and save commands allow you to save a drawing to disk and reload it later. The assignment does not require support for advanced features like multiple shape selection, undo, printing, framed shapes, multiple document windows, scrolling, and so on.

Input and output

Again, you will need to use a bit of file I/O, yet we still haven't gotten to talking about exceptions which are usually heavily in the stream classes. As we did for hw1, we will provide you with wrapper classes to help smooth this over. The `SimpleObjectReader` class can open a file for reading and has methods to read objects from it one by one and rehydrate them. It expects the objects were serialized to the file using a `SimpleObjectWriter`. This class opens a file for writing and can write serializable objects to it. The provided source files are documented (in javadoc format) with more details on the methods and usage for each class.

Requirements summary

Like we did for HW2, a summary list we constructed to help you manage the details:

General

- Your source files should be easily readable on UNIX— i.e. end-of-line characters should be proper, lines shouldn't wrap in obnoxious places, etc. This is of particular importance to those of you moving your files to Solaris from elsewhere.
- The submitted project should include all necessary source files and images. We should be able to issue the command `javac *.java` and all files should compile cleanly (i.e. with no warnings).
- Your main class should be named JavaDraw and should require no command-line arguments. After compiling, we should be able to run your program with the command `java JavaDraw`.
- The program should run as a Java application, not an applet.

Selection tool

When the arrow tool is the currently selected tool mode on the toolbar:

- A click causes the topmost shape under the mouse to become the currently selected shape. Any previously selected shape is deselected. The currently selected shape is drawn with resize knobs on its corners. Clicking on a shape that is already selected makes no change.
- Clicking on the canvas but not directly on a shape deselects any previously selected shape, leaving the selection empty.
- Clicking and dragging on a visible knob of the selected shape resizes the shape, allowing it to shrink and grow. Resizing affects some shapes a little differently (see descriptions below).
- Clicking on a shape (but not within a resize knob) and dragging causes the shape to move with the mouse. The shape stays the same size but is translated around the canvas.
- There is immediate visual feedback when selecting or deselecting a shape as well as continual updates during moving and resizing.

Rectangles

- When the rectangle is the currently selected tool mode on the toolbar, clicking and dragging on the canvas creates a new rectangle that resizes with the user's drag. The user can drag out the new shape in any direction (i.e. not just down and to the right). The new rectangle appears on top of any previous shapes and is filled with the toolbar's currently selected color.
- Moving a rectangle translates the coordinates of its bounding box in the obvious way.

- Resizing a rectangle moves the selected knob to a new location while the opposite diagonal knob remains "anchored" in place. A rectangle can be resized smaller or larger in either or both dimensions.

Ovals

- When the oval tool is the currently selected button on the toolbar, clicking and dragging on the canvas creates a new oval that resizes with the user's drag. The new oval appears on top of any previous shapes and is filled with the toolbar's currently selected color.
- An oval can be thought of as transcribed within a rectangular bounding box. The behavior of that bounding box is quite similar to the standard rectangle shape.
- An oval has four resizing knobs, one on each corner of the bounding box.
- For a click to select an oval, it must be within the oval's filled region. A click within the bounding box but outside the filled area (i.e. in one of the corners) does not select the oval.
- Moving an oval translates the coordinates in the obvious way.
- Resizing an oval moves the selected knob to a new location while the opposite diagonal knob remains "anchored" in place. This changes the oval's bounding box, which changes the dimensions of the oval. An oval can be resized smaller or larger in either or both dimensions.

Lines

- When the line tool is the currently selected button on the toolbar, clicking and dragging on the canvas creates a new straight line that resizes with the user's drag. The new line appears on top of any previous shapes and is drawn in the toolbar' s currently selected color.
- A line can be thought of as a diagonal line across a rectangular bounding box. The behavior of that bounding box is quite similar to the standard rectangle shape.
- A line has two resizing knobs, one on each end of the line segment.
- For a click to select a line, it must be fairly close to the line segment. A click within the bounding box that is not near the line (i.e. in an opposite corner) does not select the line.
- Moving a line translates its coordinates in the obvious manner.
- Resizing a line moves the selected knob to a new location while the other knob remains "anchored" in place. This changes the line's bounding box, which changes the length and angle of the line. A line can be resized smaller or larger in either or both dimensions.

Scribbles

- When the scribble tool is the currently selected button on the toolbar, clicking and dragging on the canvas traces out a path on the canvas following the mouse movement. The new scribble appears on top of any previous shapes and is drawn in the toolbar's currently selected color.
- A scribble can be thought of as a sequence of points connected by line segments. The minimum and maximum coordinates along the path in both dimensions define the scribble's rectangular bounding box.
- A scribble has two resizing knobs, one on each end of the scribble (i.e. the first and last points on the path)

- For a click to select a scribble, it must be reasonably close to a segment of the path. A click within the bounding box that is not near a path segment does not select the scribble.
- Moving a scribble translates its coordinates in the obvious manner.
- Resizing a scribble works differently than other shapes. Clicking and dragging one of the resize knobs doesn't scale the entire scribble, but instead allows you to extend the path in that direction with additional points. The end of the scribble path follows the mouse drag and lengthens the path, potentially enlarging its bounding box. There is no way to shrink or shorten a scribble path.

Layering

- The "Bring to Front" command moves the selected shape to the top so that it is drawn above all other shapes and will be the first to receive mouse clicks. The ordering of all other shapes remains the same. This command has no effect when there is no selected shape.
- The "Send to Back" command moves the selected shape to the bottom so that it is drawn below all other shapes and will be the last to receive mouse clicks. The ordering of all other shapes remains the same. This command has no effect when there is no selected shape.

Colors

- Whenever a new shape is selected, the color button on the toolbar updates to show the color of the selected shape.
- Clicking on the color button brings up the color chooser dialog that allows the user to select a new color. After the user chooses a color and dismisses the dialog, the currently selected shape (if any) changes to the new color.

Edit operations

- The clipboard operations must be implemented by object cloning. Each shape must be capable of cloning itself to store onto the clipboard. We are not interfacing with the system clipboard which has its own complications. Our “clipboard” is simply a copy of the most recently cut or copied shape.
- The "Copy" command stores a copy of the selected shape onto the clipboard, replacing any previous clipboard contents. There is no change to the shape or the canvas. The copy command has no effect when there is no selected shape.
- The "Cut" command copies the selected shape onto the clipboard and deletes the shape from the canvas. After a cut, there will be no selected shape. The cut command has no effect when there is no selected shape.
- The "Paste" command adds the shape on the clipboard to the canvas. The newly pasted shape becomes the selected shape. The pasted shape should be a duplicate of the last shape that was cut or copied (e.g. same shape type, location, size, color). The same clipboard contents can be pasted multiple times. It is a nice touch (but not required) if a subsequent paste is offset slightly from the previous so the identical shapes don’t pile on top of one another. The paste command has no effect if the clipboard is empty (i.e. no cut or copy has been performed).

- The "Delete" command deletes the selected shape from the canvas. It does not change the clipboard contents. After a delete, there will be no selected shape. This command has no effect when there is no selected shape.

File operations

- The "Clear All" command deletes all shapes, returning to a blank canvas. There will be no selected shape.
- The file operations must be implemented by object serialization. Each shape must be able to read and write itself to a stream. We provide simple classes that wrap the object streams to simplify the I/O and handle exceptions for you.
- The "Load" command brings up a file chooser dialog that allows the user to select a file to load. The program should open the file, read the shapes, and replace the current contents of the canvas with the shapes from the file. There should be no selected shape after a successful load. If the file cannot be read or doesn't contain any valid shapes, the current contents of the canvas are left unchanged.
- The "Save" command brings up a file chooser dialog that allows the user to select a filename for saving. All of the shapes in the canvas should be saved to the file. The contents of the canvas are not changed in any way. If the file cannot be written, no error is reported.
- The "Quit" command exits the program. It does not have any safety checks to ask the user to save changes before quitting or any such niceties; it just immediately exits. Closing the window also exits (The code for this case is already in the starter file.)

A few random details

The starting project uses Swing components and thus will require an environment with Swing GUI support – see the course tools page for help on running Swing on the various platform. Running a Swing app on our leland Solaris machines sometimes may spew messages about missing fonts. These are harmless and can be ignored. Depending on your window manager on leland, some windows may show up with the slightly wrong appearance. This is an interaction bug between the Swing implementation and the window manager, so it's nothing to do with your code.

Grading

Functionality will be tested by verifying that your draw program can perform all the required operations on all four types of shapes, redraw correctly as shapes are moved, resized, change color, etc., can cut/copy/paste, load and save files, and so on. Design will be weighted somewhat more on this assignment than the last, since that is where much of your effort will be going. Design will consider how well you succeeded at constructing a sensible hierarchy, factoring the common data and code, and designing objects that provide for good abstraction and encapsulation.

Getting started

As usual, there are starter files available off the course page. There's a lot of basic stuff done for you, so you'll need to spend a little time familiarizing yourself with the provided code.

No matter where you do your development work, when done, it is your responsibility to ensure your project will compile and run properly on the leland workstations. All assignments will be electronically submitted there and that is where we will compile and test your project. We recommend moving your code over a day or two in advance to give yourself plenty of time to test and resolve any problems instead of trying to deal with everything in a last-minute panic.

Design Ideas

We're going to continue the trend of separating the assignment handout into two parts. The first discusses all of the functional requirements while this second part offers ideas and strategies for design. Read the assignment handout first to get familiar with the scope of the problem and then use this one to get an idea of how you might approach it. This handout describes the code in the starting implementation and suggest strategies for evolving it into the final goal and what milestones to shoot for along the way. We also have some notes about the sticky parts you might encounter and discussion about good use of inheritance. Hope you find our suggestions worthwhile and helpful, even if you eventually decide choose different paths of your own for completing the assignment.

A roadmap to the starting implementation

There are six classes provided in the starting project. Each of the source files is reasonably well-commented, but let's take a look at each to get an overview:

JavaDraw.java

This is the main class for the application. It only has two methods, the static main() that kicks the whole thing off and a menu creation method. The given code instantiates the drawing window and creates and install the drawing canvas, toolbar, and menus. All of the code in this class works correctly as is and should require no changes.

Setting up a user interface in Java tends to involve tedious calls to create components, configure their parameters, and make lots of tweaks to lay them out into visually pleasing ways. Scan the code we give to see what I mean. Our code creates menus and menu items, sets keyboard accelerators and registers event listeners to pass along the menu commands to the drawing canvas. There are many anonymous inner classes used as listeners, this is a very common technique and one that you will want to become familiar with.

Toolbar.java

The Toolbar sets up and manages the group of tool buttons and color selection button installed along the bottom of the drawing window. It has a handful of public methods that allow a client to set and get the current tool and color and register a listener to be notified on color changes. All of the code works correctly as is and should require no changes.

The different tool buttons form a group where at most one button can be selected at a time— choosing a different tool always deselects the previous choice. The background of the color button shows the currently selected color. Clicking the button brings up the color chooser which allows the user to select a different color. Whenever the current color on the toolbar changes, the toolbar notifies any objects registered as change listeners by sending them a change event. Although you shouldn't need to edit the code in the toolbar, it is worth going over it to see the various details that need to be handled to set up this part of the user interface. The drawing canvas will need to interact with the toolbar object, using the methods to set and get the settings as well as registering a change listener.

SimpleObjectReader.java and **SimpleObjectWriter.java**

Like the two I/O classes we gave you for HW1, these classes are simple wrappers around standard java.io stream classes with the added feature that they handle the exceptions for you. The writer class has only three methods: a static method to open a new file for writing, a method to write an object and a method to close the file. The reader class has a parallel set of three methods for reading. The objects that are being read and written must

implement the Serializable interface. All of the code works correctly as is and should require no changes.

Rect.java

The Rect class defines a simple rectangular shape object. It tracks its bounding box, selected state, and the canvas it is being drawn in. It has methods to select, move, and resize itself. It can properly draw itself and updates the canvas whenever the state or bounds of the rectangle change. The given code works properly, but you will need to extend and change the class to support additional features.

Most of the given code will end up consolidated in a common base class from which all your shape variations will inherit. Rearchitecting the code to support inheritance will require some careful planning and design to maximize code sharing. We recommend carefully thinking through your strategy before rearranging any of the code.

DrawingCanvas.java

The DrawingCanvas is a subclass of JPanel that provides a drawing surface for creating and modifying rectangles. It keeps an ArrayList of Rects, each of which is responsible for tracking its own size and position. The canvas paints by iterating over its rectangle objects and telling each draw itself. The canvas stores a selected rectangle. Mouse events on the canvas are handled by an inner class that listens for mouse events. The inner class determines which rect is under the mouse and then messages it to move or resize as the mouse is dragged. The code in the canvas class works properly, but you will need to extend and change the code to support additional features as required. The DrawingCanvas needs to eventually support all four shape types, as well as the additional operations for using the clipboard, layering, colors, and saving and loading files.

Getting your bearings

The first thing you should do is just compile and run the provided code and play with it. Drag out new rectangles, select and deselect them, move and resize them, and get a general feel for how the program works. Try out fringe cases and see if you can find the program's weak spots. Ask for your money back if you expose a bug we failed to find.

Next, take a look at the given code, in particular, focusing on the DrawingCanvas and Rect classes since those are the two that you are going to extensively modify. Map out for yourself what responsibility each object has and how it interacts with the other objects. Look carefully at the starting design and evaluate how well it meets the goals of a good obejct-oriented program. Do objects encapsulate their state properly? Are the methods well-designed to maintain object consistency? Do objects take responsibility for their own behavior? Are the relationships between the various classes clear and easy to understand?

Now, study how the graphical aspects are managed. How do the canvas and shapes cooperate on drawing? How does the shape track the state to be drawn? How does the canvas refresh when changes have been made? You will want to have a good understanding of the basic framework so that you will be able to make changes without breaking any of the given behavior.

Although the assignment handout lists the new shapes as the first task in the list of requirements, it is actually one of the harder jobs you have to do and one that requires the most skill. We recommend keeping the program as just a rectangle program for a while and adding in some of

the easier features and then as you gain more confidence, tackle the more complex task of introducing the new shape subclasses later on.

Layering

The straightforward layering commands make a good starting point. The menu items have already been created and set up to send the canvas the appropriate front/back methods, your job is just to fill in the missing implementation. How does the canvas record and control the layering of shapes? Study the canvas class to learn what it required for a shape to be moved "above" or "beneath" all others. What will be the appropriate way to redraw the affected part of the canvas? Be sure to test all the various fringe cases such as when there is only one shape to reorder or no selected shape or you to move the topmost shape to the top and so on. Also, see the ArrayList docs. Testing hint: it's hard to determine the layering when all shapes are the same color, so you may want to temporarily have each rectangle be created with a randomly chosen color so that you can more easily see the layering relationship.

Colors

Adding color support has a bit more to it than layering, but not too much, so it makes a good second task. Start by making new shapes take on the current toolbar color. How do shapes currently decide what color they are drawn in? How can you change that? When a new shape is being created, how can the canvas determine what color to use? Once this is working, set things up so the toolbar updates to the color of the currently selected shape. Finally, hook up a listener so that changing the current color on the toolbar changes the color of the selected shape.

Clipboard

There is a Clipboard class in the java.awt package, but unfortunately, its design is a bit cumbersome and its features are overkill for what we need. We recommend ignoring the system clipboard and creating your own clipboard class that can handle the simple needs of your program. Your clipboard only needs to support storing a copy of the selected shape and later retrieving it, nothing more fancy than that. The cut, copy, and paste menu items are already set up to send the appropriate messages to the canvas, you will need to fill in their implementation to work with your clipboard. You will need to make a copy of the selected shape to store on the clipboard because storing just a reference will lead to sharing between the original and duplicate shape (i.e. moving one moves the other or changing one's color changes the other which is not what you want!) The Cloneable interface and the clone method will be the tools you use to make a shape capable of producing a duplicate of itself. Be careful when implementing the clone method— it needs to make a full deep copy so that there are no unintended shared references between the two shapes. Be sure to test all your clipboard operations thoroughly in all their variations to shake out any lurking bugs.

Serialization

In order to save and load drawings from files, you will need to be able to archive the shapes. As is customary in the OO paradigm, each object that needs to be saved should take responsibility for writing itself out and reading itself back in. The Java means to do this is by having the object support the Serializable interface and potentially override the readObject and writeObject methods to customize the process. (We'll get to serialization in lecture shortly.) With serialization, writing out a drawing mostly becomes a matter of having the canvas ask the shapes to write themselves. Loading a drawing means having the shapes read themselves back in. When making an object support serialization, you will need to make careful decisions about exactly

what state needs to be written and read, as well as accounting for any additional actions that need to be taken when re-constructing the drawing from the newly read shapes.

The Shape classes

The canvas needs to support a number of different shapes, each having some features in common with the others, while other features are unique. A common parent class can be used to capture the similar elements, while inheritance and overriding can produce specialized variants.

The starting Rect has methods for creating, selecting, drawing, moving and resizing rectangles. Although that code that is there is specific to rectangles, a lot of it can be made to work in a common fashion for all shapes. Where possible, you want to organize the functionality to be shared. You will also need to add new features like colors, cloning, etc. to all shapes in a way that unifies all the common code.

Designing an object hierarchy

Taking the starting code and re-architecting it into a related set of shape classes is an excellent exercise in designing an inheritance tree. One of your most important jobs is going to be laying out the tree and figuring out what data and methods go with each object. We encourage you to sketch things out on paper and plan your strategy before writing a lot of code since you want to avoid committing to a design that would be difficult to implement. The most important part here is learning how to enable code re-use by factoring shared code upward in the hierarchy.

The different shapes have a lot of behavior in common— the challenge of the assignment is structuring your classes so as to avoid repeating code. You should also be able to add other types of shapes without a lot of modification. This means you will want many small well-named methods. Although may of the methods will be quite short (a line or two) by factoring into methods of their own, you enable subclasses to override the method and easily extend or change details while still inheriting all the common behavior with no effort.

Where code is not quite the same, you do not want to copy and paste methods from class to class, making small modifications. Instead factor up all the common parts and move the different pieces to small helper methods that can be overridden as needed. With all common behaviors factored out to the parent class(es), only the extensions and differences appear in the specialized subclasses. Don't give in to the temptation to just duplicate even a few lines of between similar methods. Committing yourself to making inheritance work for you on the small things will help you reap the larger benefits on inheritance when designing more complex hierarchies later.

The parent class declares the common methods, those that can be reasonably sent to any of the subclass varieties. Whether or not there is a good default implementation will decide whether the method is abstract. Be careful to not push behavior up past where it is appropriate, a parent class should only declare messages that make sense for it and all subclasses. If you find yourself subclassing and wanting to "take away" behavior from the parent rather than extend it, it may be a sign your design needs some work.

If you have done a good job, when you're done it should be easy to add other shape variants with minimal effort. There is a fair amount of latitude in the way you can construct a reasonable set of classes. The most important expectation is that your code uses sensible object-oriented design. Your classes should present a clean abstraction that successfully coordinates with other classes

through a reasonable interface. Classes should take responsibility for their own behaviors (displaying, changing status, etc.) rather than being externally manipulated. Code should not be repeated between methods or classes. And so on...

Ovals

The oval is probably the easiest subclass since it is so similar to the rectangle. The only mildly complex issue for the oval is doing hit-detection to determine if a given point is inside the oval. Rather than force you to consult your old geometry textbook, some pseudocode to guide you:

```
x and y are the point we are testing
centerX and centerY are the center of the oval
height & weight are the oval dimensions

May want to enlarge width/height to allow for a little extra tolerance
around outside edge of oval and avoid divide by zero in following code

double normx = (x - centerX) / (width/2.0);
double normy = (y - centerY) / (height/2.0)
inside = (normx * normx + normy * normy) <= 1.0;
```

Lines

The line has a few little quirks of its own. Although its bounding box behavior is the same as rectangles, it also needs to track the direction of the diagonal across the box. It also is a bit different in the knob handling since a line only has knobs on the two ends. However, with some careful planning in the Shape parent class, you can make sure that the Line only has to make minimal changes to override and tweak the behaviors as necessary.

The hit-detection for a line can be a bit messy. Here is some psuedocode you might find helpful. This computes the angles of the two points relative to the start and then invokes good old Pythagoras to compute the distance of the point away from the line. Figure that any click within 5 or 6 pixels is "close enough" to be considered a hit on the line.

```
startx & starty is one end of line, endx & endy the other
x and y are the point in question to test

Before doing anything complicated, make sure the point is within the
bounding box of the line  -- enlarge the rect a bit (6 pixels) to allow for
some tolerance, if not inside that box, don't bother with the rest here

width = endx - startx, height = endy - starty;
dx = x - startx, dy = y - starty;

 if (dx != 0 && width != 0) { // if both lines are not vertical
     angleToCorner = Math.atan(height/width);
     angleToPoint = Math.atan((dy/dx);

     distance = Math.sqrt(dx*dx+dy*dy)*
                      Math.sin(Math.abs(angleToCorner-angleToPoint));
 } else       // one or both slopes have zero divisor (are vertical)
     distance = Math.abs(dx);
inside = distance < tolerance (5-6 pixels?)
```

Scribble

The scribble is the most divergent of the shapes, but still has a lot in common with the others and can leverage a lot of the shared behavior. Its geometry cannot be expressed with just a simple bounding box, since it also has to track the sequence of points along the scribble path, so it will

require some extra state and handling to work correctly for the various operations. As usual, try to make use of as much of the common code as you can and only override where absolutely necessary. I think you'll find the Scribble the most interesting of the shapes to implement since it does offer some neat challenges in unifying its behavior with the others.

Hit-testing for a Scribble involves hit-testing over a lot of line segments, so it needs to incorporate the same logic presented above for the Line. Take care to unify the line-point-distance code that will be needed by both Line and Scribble. This is not necessarily through an inheritance relationship, but perhaps a shared a helper static method that both use, but it is just good form to not duplicate code in more than one place.

Incremental testing

This program is complex enough that just starting to add code wherever and hack your way to completion is unlikely to be a fun or successful endeavor. The program we gives you work as is, take is as a goal to always keep your program not far from a working version as you go. Make small changes, test them thoroughly, and then move on to the next step. Adding 300 lines of code that add 4 new features at once without even stopping to compile along the way is a sure means to make debugging an exercise in torture. If you just added the cut command, you can focus on testing it in all uses without worrying about what other unintended side effects are coming from the other things you changed at the same time. As you add each new shape subclass, thoroughly test it before moving on. Does it properly select and deselect? Move? Resize? Change colors? Cut and paste? Only when you're sure it has all the required features working should you move on to the next shape challenge. This strategy will be more effective and satisfying, and as a side bonus it ensures that you're always near a good stopping point. If you run out of time near the end of the project, would you rather submit a program that attempts 100% of the functionality and gets it right 80% of the time or a program that has 80% of the features perfectly implemented?

A few random details

- All the previous class design guidelines still apply (compiles cleanly, no public instance variables, functionality in right place, etc.).
- Although it might be tempting, you should avoid using the runtime type identification (i.e. `instanceof` and `getClass`) to figure out what class an object is so you can make decisions based on its class. You should instead design things so you send the same message and expect the objects to respond in their own way through the miracle of dynamic binding. This is the value of polymorphism— don't figure out what class an object is and construct a switch or if statement to route control yourself!

- You might find it convenient to take a class name expressed as a String or a Class object itself and create an instance of that class, such as creating a shape by name or class. If you're curious how to do this, check out the documentation on java.lang.Class which has methods that allow you to retrieve a class by name and create an instance.