**Homework Assignment #1**

**Java GUI Programming with Anonymous Inner Classes  
The Shape Calculator**

Link: https://www.clear.rice.edu/comp201/08-spring/labs/lab10/

In this lab we will create Graphical User Interface ("GUI") "Shape Calculator" program which will instantiate any shape we desire with the desired, appropriate dimensions, draw it on the screen and calculate the area of that shape. Hopefully through this process we will learn a number of techniques and concepts:

* How to create a simple window frame and poplulate it with panels, buttons, labels and text fields.
* Using abstract factory objects to create abstract product objects.
* Using anonymous inner classes to define concrete instantiations, especially in the context of factories.
* Using the closure properties of anonymous inner classes to directly access instance and other variables.
* Using anonymous inner classes as event listeners in GUIs.
* Using graphics calls to paint a simple shape onto a component.
* Numerous demonstrations of polymorphic behavior, especially those that cannot be replicated with conditional statements.
* Using a incremental process to slowly build a complex software system by gradually adding components and capabilities.

This lab, in all its parts, comprises the homework assignment for this week. You are to start the work during lab and continue it on your own to be handed in next week.

**Demo**

To see where we are going, download and run the demo (executable jar file--select "Open" when downloading): [ShapeCalc.jar](https://www.clear.rice.edu/comp201/08-spring/labs/lab10/ShapeCalc.jar)*Note that it is an Honor Code violation to attempt to decompile the demo.*

There are 3 shape factories available in the demo: shapecalc.shapes.SquareFactory, shapecalc.shapes.CircleFactory and shapecalc.shapes.NullFactory

**Important directory organization instructions:**

Since this lab consists of several parts, where the code evolves from one part to another, you need to make several git branches:

1. Create an overall lab directory: Already been created for you, its on your github
2. Create 4 branches called roll-part1, roll-part2, roll-**part3**, and roll-**part4** respectively.

Each branch should contain the **entire** code that completes that section of the lab.

**Background Information**

For background information on the various components that go into a Java GUI, please see the lecture notes ([lecture 1](https://www.clear.rice.edu/comp201/08-spring/lectures/lec24), [lecture 2](https://www.clear.rice.edu/comp201/08-spring/lectures/lec25)).

This lab makes extensive use of classes in Java API, whose documentation can be found here: <http://java.sun.com/j2se/1.5.0/docs/api/index.html>

A nice resource is the Sun's tutorials on making GUIs with Swing: <http://java.sun.com/docs/books/tutorial/uiswing/TOC.html>

**So, let's get started! Remember to do the following parts*in order*:**

Warning! As you progress through the lab, the directions will deliberately get less and less detailed, as you are expected to figure more and more of the process out on your own!

**The completion of this lab will constitute your HW08!**

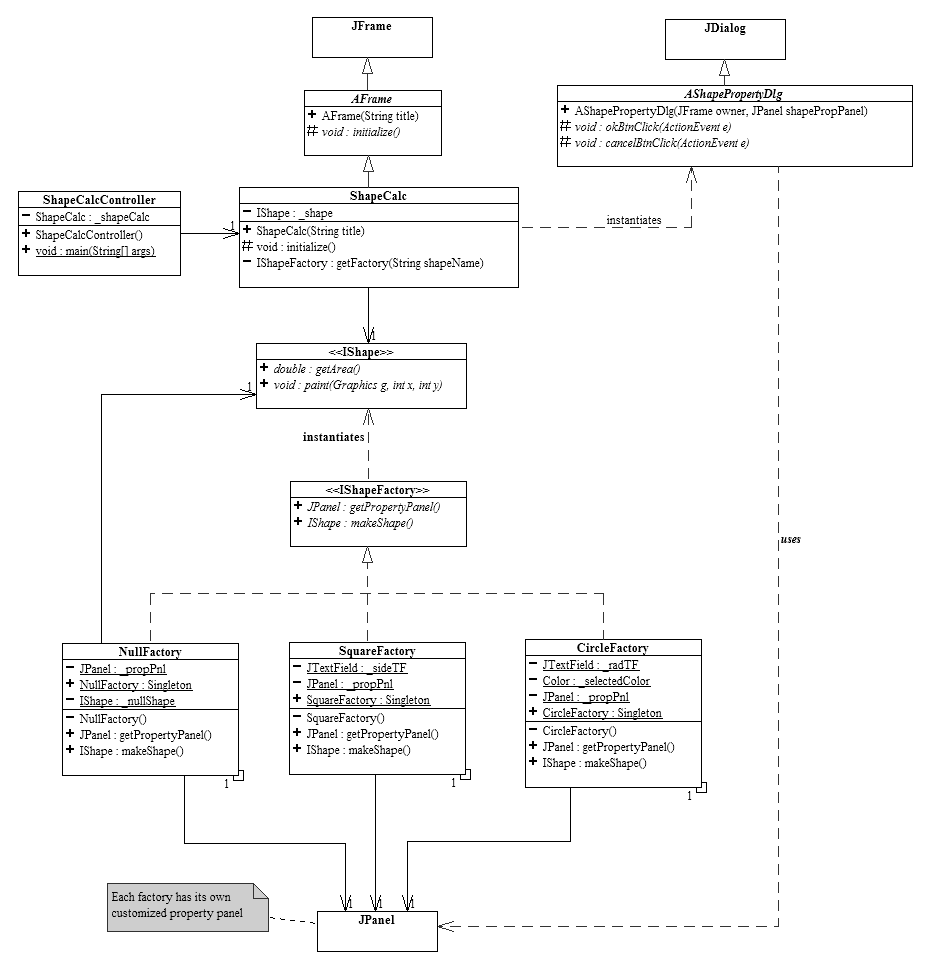
[Part 1: Making a simple window frame](https://www.clear.rice.edu/comp201/08-spring/labs/lab10/part1/index.shtml)

[Part 2: Instantiating a fixed shape and displaying its area on a label.](https://www.clear.rice.edu/comp201/08-spring/labs/lab10/part2/index.shtml)

[Part 3: Painting a shape onto a panel and instantiating an arbitrary shape.](https://www.clear.rice.edu/comp201/08-spring/labs/lab10/part3/index.shtml)

[Part 4. Adding a dynamically generated dialog box to set a shape's properties](https://www.clear.rice.edu/comp201/08-spring/labs/lab10/part4/index.shtml)

For reference, here is the UML diagram of the complete system that we will be building:



## Making a Simple Window Frame

In the immortal words of [Elmer Fudd](http://looneytunes.warnerbros.com/stars_of_the_show/elmer_fudd/elmer_story.html), "befwar we can make wabbit fwicasee, we must fwirst catch a wabbit!" (or if you'd rather, [genuine Elmer Fudd quotes](http://www.barbneal.com/elmer.asp)). So, before we make an application that uses a GUI, we must first figure out how to make a GUI.

**Put all the following classes in a package called shapeCalc.**

## The Controller

The main function of a Java program serves only to start the object system up and is not really a part of the overall functioning of the system. It's job is simply to create the first object.

So, what we will do here is to create a special class called ShapeCalcController whose sole purpose is to start the system up.

1. Go ahead now and create ShapeCalcController with
   * a no-parameter constructor
     + Just so that you know that it is running properly, put some "stub" code in the constructor such as a System.out.println that says that the constructor was run. We'll remove this stub code later.
   * a main method, which always has the following signature: public static void main(String[] args)
     + The body of the main method should simply instantiate ShapeCalcController. There is no need to assign the newly created object to variable, all you are doing is bringing it into existence. It will do the rest.
2. Compile and run your program by right-clicking the ShapeCalcController class in DrJava and selecting "Run Document's Main Method".

Later in this course the controller class will become a much more important piece of a "[Model-View-Controller](http://www.exciton.cs.rice.edu/JavaResources/DesignPatterns/MVC.htm)" design pattern. But for now, we will simply use it to instantiate our main window frame.

## The Frame

A "frame" is the basic window object in Java. But as supplied by Sun, a frame has no buttons, or text fields or labels or other GUI components to make it useful.. We will make a "custom" frame by subclassing Sun's frame, JFrame.

1. Copy the code for [AFrame from the lecture notes](https://www.clear.rice.edu/comp201/08-spring/lectures/lec24/AFrame.html), placing the code in your shapeCalc package. Be sure to change the package name inside the code to shapeCalc!
2. Create a class called ShapeCalc that extends AFrame. This means that ShapeCalc will automatically be able to do all the cool things a JFrame can do, such as resize, minimize, maximize, move around the screen, etc. This also means that your frame will end the program when it is closed, as it inherits this behavior from AFrame.
   * In ShapeCalc, define a 1-parameter constructor that takes a String for its title. Since AFrame's constructor takes the title, all that ShapeCalc needs to do is to call its superclass constructor, handing it the title, i.e. super(title);
   * In ShapeCalc, define a concrete implementation of the abstract *intitialize()* method inherited from AFrame. Leave the body of this method blank for now--we will add code for various components later.
3. Modify the constructor of ShapeCalcController such that it instantiates an instance of ShapeCalc (with a title string), assigning it to a variable, which can be either a local variable or a field of ShapeCalcController.
   * Under what circumstances would it make a difference to use a local variable vs. a field?
4. Simply instantiating a frame doesn't show it on the screen however.
   * After instantiating the ShapeCalc, call its validate() method (no parameters). This will ensure that all the various components in the frame are properly positioned and sized. Of course, we don't have any components in our frame yet, but "real soon now..."
   * To actually get our frame to show on the screen, we need to set its visibility to true. We do this by calling its setVisible() method and passing true as the input parameter.
5. Run your ShapeCalcController again and see that a small but beautiful frame does pop up! You can remove the println stub code at this point, now that you have other verification that the program is running properly.
6. To get your frame to have a reasonable size and to pop up in the middle of the screen, write the following in the frame's intitialize()method, in this order:
   * setSize(400, 400); // This will set the size of the frame to 400x400 pixels.
   * setLocationRelativeTo(null); // This will center the frame on the screen. This must follow the setSize method because otherwise the width and height are still 0 and the frame won't center properly.

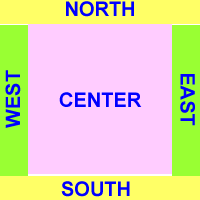
## 

## Adding Components to the GUI

Ok. So you're bored with your plain little frame that you managed to pop up. (*Old wheezing voice:*) "Well, back when I was your age--little whippersnapper--it took hunderds of lines of code to just to get a plain little window to pop up. You just don't know how easy you've got it these days with that Java toolkit stuff and all! In my day, GUI programming built character--a little pain and suffering is good for the soul!"

Old reminiscing aside, let's add some more graphical components to our frame.

**All of the following code is to be added to ShapeCalc.initialize(), except for import statements:**

1. First, let's put 2 panels and a label onto the frame.
   1. Create 3 local variables, ctrlPnl, displayPnl, areaLbl of types JPanel, JPanel and JLabel respectively.
      * You will need to import javax.swing to do this.
      * Initialize areaLbl by calling the constructor of JLabel with a the string "Area:" so that by default, it will show this text.
   2. A frame is capable of having many "panes" which can be layered one on top of the other like overhead tranparencies. We will only deal with one pane in this course, the "content pane" which is the main, default pane.
      * Create a temporary variable of type Container called cp and set it equal to getContentPane(). You will need to import the java.awt package to do this.
      * A frame, by default, automatically lays out the components added to it (to its content pane, actually). Java containers (GUI components that can contain other GUI components -- does this sound familiar?) can be set to have any sort of automatic layout mechanism that is desired. Be default, a frame has a "Border Layout", which means that the components are arranged along the edges of the frame or in its center:  
        
      * To add the ctrlPnl to the top of the frame's edge, we write: cp.add(ctrlPnl, BorderLayout.NORTH); -- See how the border layout positions are implemented as static, final fields of the BorderLayout class?
      * Add the displayPnl to the center and and areaLbl to the south edge of the frame.
   3. Run your program. You will not see the panels at the north and center but you should be able to see the label at the south edge of the frame.
2. Now let's add a text field and a button to the north panel (ctrlPnl):
   1. JPanels are Containers and by default have a FlowLayout, which means that any components that are added simply go in one after another, "flowing" from left to right and wrapping to the next line if necessary.
   2. Declare a local variable of type JTextField called typeTF. If you pass the JTextfield's constructor a String, it will start with that String in it by default. Hand it the string "Square" for now.
   3. Declare a local variable of type JButton called newBtn. Hand the JButton's constructor the String "New..." which will appear on the face of the button.
   4. Using the ctrlPnl's add method, add typeTF and newBtn to the panel. No other input parameters are necessary since each component will simply appear (form left to right) in the order in which it was addd.
   5. Run your program to verify that the text field and the button now appear at the top edge of your frame.
3. Now let's see if we can get some GUI action! We need to connect an event "listener" to the button so that when the user clicks on the button, something happens. Anonymous inner classes will come into their own here.
   1. Import java.awt.event, as we will need some classes from that package.
   2. Call newBtn's addActionListener method, handing it a new instance of an anonymous inner class derived from ActionListener.
      * ActionListener is an interface and must implement the method public void actionPerformed(ActionEvent e). When the button is clicked, it is this method that will be called by the button.
      * For now, just put a System.out.println in the body of actionPerformed to verify that the listener actually gets called when the button is clicked. This is j ust stub code that you will delete later.
      * Run your program and click the button. Does does the correct message get printed out every time you click the button?
   3. Now let's try something more sophisticated. This is still stub code, but it is fancier stub code. Let's try to transfer the text from the text field to the label when the button is clicked.
      * A JTextField has a String getText() method that will return the text that is currently showing on it.
      * A JLabel has a void setText(String s) method that sets the text of the label to the supplied string.
      * Using the above information, replace your System.out.println in actionPerformed with a single line of code that will transfer the text in typeTF to areaLbl when the newBtn is clicked.
        + You will need to declare both typeTF and areaLbl as final. Why?
      * Run your program and verify that clicking the button will transfer whatever text you type in to the textfield to the label.

Congratulations! You've built your first Java GUI program!

## Instantiating a Fixed Shape and Displaying Its Area on a Label

***Before* starting this section, copy all the code from part 1 into your part2 subdirectory and make any modifcations to the code in part2. Do NOT modify your part1 code!**

In this section we will start the process of creating a factory that creates shapes. In the spirit of [eXtreme Programming](http://www.extremeprogramming.org/) ("XP"), we will continue to build our system slowly, adding in functionality bit by bit.

But before we start making code to process shapes, we must first define what it means to have a shape and to have a factory that produces them:

## Interfaces for Shapes and Factories for Shapes

The interfaces to the key *abstract* elements of our system define the semantics, the meaning, of those objects in our system. In geek-speak, we are effectively creating the "Application Programming Interface" or "API" to our system. In a nutshell, we are defining the manner in which the invariant parts of the code communicate with the variant parts of the code. In this case, the invariant part will be the GUI portion (technically, including the abstract interfaces), and the variant part will be the concrete shapes and the concrete factories that create them.

So, what is a shape?

A shape could be many, many things with many different types of behaviors. So in the spirit of KISS (for this class: "Keep It Simple Smartypants!"), we will simple say that a shape is intelligent and can calculate its area and paint itself onto a graphics object at a given (x,y) coordinate.

package shapecalc;

import java.awt.\*; // needed for Graphics class

public interface IShape {

public abstract double getArea();

public abstract void paint(Graphics g, int x, int y);

}

If we need additional functionality later, we will add it then.

So a shape factory is thus something that makes *IShapes*:

package shapecalc;

public interface IShapeFactory {

public abstract IShape makeShape();

}

Once again, we will add additional functionality later as needed.

Let's turn the argument around for a minute. If an IShapeFactory makes an IShape, then what is an IShape?

Consider this object:

new IShape() {

private double b1 = 3.14;

private double b2 = 2.72;

private double h = 6.022;

public double getArea() {

return (b1+b2)\*h/2.0;

}

public void paint(Graphics g, int x, int y) {

// some pretty image

}

}

Is this an IShape? Does it act as an IShape should? If something walks like a duck, quack like a duck and swims like a duck, is it a duck?

(FYI, the above shape acts like a trapezoid.)

An IShape is an object that can calculate its area and paint itself onto a graphics object in a manner as defined by the factory that made it. A square does "square" things because it was made by a square factory. Bottom line here: it doesn't matter what we call any given shape, just that it has the abstract behaviors of a shape. A shape factory is an object that guarantees that any shape it creates will be properly behaving shape.

## Making a Shape

Let's slowly integrate IShape and IShapeFactory into our GUI framework:

1. In a new shapecalc.shapes package, create a concrete implementation of an IShapeFactory called SquareFactory.
2. Implement SquareFactory's makeShape() method using an anonymous inner class to create the new instance of IShape.
   * The anonymous inner class should have a private double field representing the length of the side of the square, which is initialized, for now, to some fixed value.
   * The getArea() method should return the area based on the length of the side of the square.
   * For now, simply leave the body of paint() blank (a "no-op").
   * Make SquareFactory into a singleton.
3. In ShapeCalc, remove the stub code from the button listener and replace it new stub code:
   * Call to SquareFactory's singleton to make a shape.
   * Take that shape and set the areaLbl's text to "Area: "+ the shape's area.
4. Run and test your program to see if your button can now make a viable shape object.

So far so good!

## Painting a Shape onto a Panel and Instantiating an Arbitrary Shape

***Before* starting this section, copy all the code from part 2 into your part3 subdirectory and make any modifcations to the code in part3. Do NOT modify your part2 code!**

So far so good. We've got a GUI application and we've been able to successfully use a factory to instantiate a shape. Now can we get that shape to paint itself onto the screen?

## Painting a Shape

As defined by IShape, all shapes have the ability to paint themselves onto a Graphics object. Well, what does that mean anyway? Let's discuss painting in Java for a bit:

Whenever a graphical component in Java wishes to display itself on the screen it must "paint" itself onto the screen. There are lots of issues and technicalities surrounding the proper way to paint an image onto the screen--consult any of a number of volumunous graphics textbooks on the subject. In Java, however, there are few guiding and thankfully, simplifying viewpoints taken:

* Painting is a service provided to the the systems (i.e. the Java virtual machine) so that when the *system* so desires, an object can display itself onto the screen. An object does NOT determine for itself when it should paint itself onto the screen.
* An object can *request* that the screen be repainted, but cannot determine if or when that request is fulfilled. All graphical components (subclasses of java.awt.Component) have a repaint()method, which is a request to the JVM to paint that component *whenever the JVM can get around to it.*
* Any object does not either know about the conditions in the rest of the system nor does it care about where on the screen it is located. When asked to paint by the JVM, an object is handed a java.awt.Graphics object, which provides an environment just for that particular painting process. The Graphics object provides a localized coordinate system to be used, releasing the object being painted from knowing the specifics of its graphical environment. Graphics provides information useful for the painting process, such as the current pen color and provides services such as drawing a straight line, box, filled circle, etc. **The Graphics object is NOT instantiated anywhere in the code under development--it is created by the JVM expressly to paint a particular component.**

In order to get our shape to paint itself onto the screen there are a number of steps to go through.

1. Make the shape that the factory creates accessible from anywhere inside of ShareCalc.
   1. This is easily enough accomplished by using a private field of type IShape.
   2. The call to the shape factory's makeShape() method simply sets this variable's value.
   3. You will need to modify your code that displays the shape's area accordingly.
2. The shape is to paint itself onto the displayPnl, so the painting service provided by that panel must be overriden.
   1. We must therefore subclass JPanel to make our custom painting panel. But since we only need one instance of this special panel, we should use...you guessed it, an anonymous inner class!
   2. Change the initialization of displayPnl to be an anonymous inner class derived from JPanel, where the following painting service method is overriden: public void paintComponent(Graphics g)
   3. In the body of paintComponent
      1. First make a call to the superclass's paintComponent method, i.e. super.paintComponent(g);. This ensures that whatever is normally done in the process of painting, e.g. clearing the area to be painted, gets done.
      2. Then call the shape's paint method, handing it the Graphics object upon which it will paint. The shape wil thus paint onto the panel after the panel is done doing its normal painting processes.
         * You can hand the paint's x and y inputs any value you'd like, but if you want the shape to paint in the middle of the panel, use the getWidth() and getHeight() methods of a Component (such as the displayPnl) to pass the coordinates of the center of the panel, with respect to its upper left corner, which coincides with the origin of the Graphicsobject.
3. At this point, try running your program.
   1. The first thing you should get is a null pointer exception because your shape variable hasn't been initialized but yet the system is trying to paint it on the screen.
      1. In designing an object-oriented system, we give individual objects behaviors such that the correct overall system behavior is a natural outgrowth of the interactions between those objects. The problem with the null value is that it not an object, and in such, has no well defined behaviors or intelligence of its own. Rather than using a conditional statement to attempt to catch and eal with any possible situation that might encounter a null value, as is done in procedural programming, in our OO system we want to define a "null object" that has the proper behaviors of and represents the null value. This is a description of the [Null Object Design Pattern](http://www.exciton.cs.rice.edu/JavaResources/DesignPatterns/NullPattern.htm). In mathematics, "zero" and the "empty set" are examples of null objects.
      2. You need a "null shape"--an object that is a shape but that represents no shape. Such an object should have zero area and not paint anything onto the screen.
      3. Since all shapes are made by factories, we thus need a null shape factory.
      4. Using your SquareFactory as a template, write a singleton NullFactory that returns a shape whose area is always 0.0 and whose paint method does nothing.
      5. Initialize your shape field with the result of calling makeShape() on your NullFactory.
      6. Your null pointer errors should go away. Hooray!
   2. Since your shape's paint method is a no-op, you should see no change it the behavior of your program. Try putting a println statement in the shape's paint method to convince yourself that it really is being called whenever the frame is repainted, for instance, when you resize it. (Note: moving the frame on the screen does not always force a repaint because the operating system simply copies the pixels from one place to another.)
4. Now it's time to add some actual painting functionality to the shape.
   1. A Graphics object provides a method called drawRect that draws the outline of a rectangle on the screen. The signature of this method is   
      public void drawRect(int x, int y, int width, int height)  
      where x & y are the coordinates of the upper left hand corner of the rectangle, where x is the left-to-right position in pixels and y is the up-to-down positon in pixels. width and height are the width and height of the rectangle in pixels, respectively.
   2. If you set x and y to be the paint method's input parameters x & y, the rectangle will draw such that its upper left corner is at that point.
   3. The width and height are obviously the length of the side of the square. You will need to cast the double to an int however, which will round the double *down* to the nearest integer value ("truncate").
   4. Add the call to drawRect into the paint method of your square object and run your program. A square should appear. Yeah!
5. Well, almost. You probably noticed that the square only appears if you force the frame to repaint by resizing it or some other means. The square didn't appear automatically like you probably wanted it to.
   1. You need to have your program *request* that the frame (or display panel) be repainted after the new shape is made.
   2. Technically, only the display panel needs to be repainted, but if you ask the frame to repaint, the display panel will also repaint because the frame, when painting, will ask all the components it contains to also paint.
   3. So, simply add a call to repaint() right after the shape is made.
   4. Re-run your program. Ahh...that's better!
6. Did you notice that your square did not draw exactly at the center of the panel? Fix the math in your code such that the square will always draw exactly centered in the display panel.

## Adding Dynamic Class Loading

Notice however, that other than where the SquareFactory is hard coded into the system, none of the code written so far depends on what sort of factory was being used. So we should not write into it some limiting set of possible factories. Luckily, Java has the ability to load a class in at run time and make it part of the currently running system. All one has to do is to supply the full classname, which includes the package name, to Java's "Reflection" system. Be sure to import java.lang.reflect.\* in your code.  Consider the following method, which you can add to ShapeCalc:

/\*\*

\* Look for the Singleton field in the given class name.

\*

\* @param shapeName a String that is the full classname of the desired class

\* @return an IShapeFactory

\*/

private IShapeFactory getFactory(String shapeName) {

// There may be errors encountered so try the following code

try {

// Find the specified class and find the Singleton field in it.

Field f = Class.forName(shapeName).getField("Singleton");

// Return the static value of the Singleton field

return (IShapeFactory)f.get(null);

}

// Do the following code only if an error was "caught"

catch (Exception e) {

System.err.println(e); // Print the error that occurred

return null; // return value when error occurred

}

}

**Exception handling**:

Sometimes in your program you can expect that errors may occur that are completely out of your control as a programmer. These sorts of errors can be catastrophic for a program. But often times, if one knew that the error occured the "exception" as it's called, can be handled in a graceful manner. In the above code, for instance, it is possible that the user requests a class that doesn't exist. The Class.forName(shapeName) method would thus fail. Luckily, Java has a very nice mechanism, call the try-catch block" that handles such exceptions very well. The syntax for the try-catch block is:/p>

try {

// code that might fail ("throw an exception")

}

catch(Exception e) {

// code to process the exception

}

Simplistically, the way a try-catch block works is that the code between the curly braces following the try keyword is executed. If no errors occur, then the execution continues onward, skipping the code in the catch statement. If an error occurs in the try section of code ("throws an exception" in Java-ese), then execution in the try section immediately stops and the catch clause is executed. I like to think of the catch clause as a little function that the JVM calls when an error occurs, handing that function and Exception object containing information about the exception.

Try-catch blocks have more capabilities and issues than are outlined here and this topic will be covered in more detail in lecture.

**Using Dynamic Class Loading**

To use the dynamic class loading, we will use the text field and allow the user to type in any factory's class name they desire and have it loaded into the system at run time.

1. In the newBtn's ActionListener's actionPerformed method, create a local variable of type IShapeFactory and initialize it to the return value from the above getFactory method where the input String is the text from the typeTF textfield.
2. For convenience's sake, change the initialization of the text in typeTF to be "shapecalc.shapes.SquareFactory"
3. You should be able to dynamically load either the SquareFactory or the NullFactory and the system should display them properly.
4. Try this: type an incorrect class name into the text field and then click the "New..." button.
   1. Don't you just love the smell of null pointers in the morning?
   2. In the getFactory method code above, what should be the value returned when an error occurs? Remember that fundamentally, the method must return a factory. Adjust your code accordingly. **No conditional statements (e.g. if) allowed!!**

NNote: Ballworld used this sort of dynamic class loading to load the strategies from the disk at run time.

## A Shape That's a Bit More Fun

Let's make another shape factory that involves more fun things to do with graphics.

Call your new factory CircleFactory (in the shapecalc.shapes package) and let it create IShapes that behave like circles. Let's have our circles be filled circles of a particular solid color.

I will simple lay out some tools for you to use and you can put them together yourself (see the[Java 1.5.0 API documentation](http://java.sun.com/j2se/1.5.0/docs/api/index.html)for more information):

* TThe java.awt.Color class represents a color in Java GUIs. It has a number of predefined static fields for common colors: Color.RED, Color.BLUE, Color.GREEN, etc.
* The Graphics class as the following useful methods:
  + getColor() which returns the current pen color (the color it paints with).
  + setColor(Color c) which sets the current pen color.
  + fillOval(int x, int y, int width, int height) paints a filled oval with the current pen color where the size of the oval is determined by an imaginary "bounding box" that exactly surrounds the oval and which has an upper left corner at the given (x, y) coordinate and the given width and height.
* Math.PI is the value of pi.

**Requirements:**:

* Before you set the graphics pen color, save the original color in a local variable and then after you are done painting, restore the pen color to its original color. This keeps your program from changing the pen color on another graphics process that is assuming that the pen color is stable.
* Use a fixed value for the circle's radius for now.
* Contrary to first instincts, do not put a Color field inside the IShape anonymous inner class. Instead, put the field in the factory itself and call it \_selectedColor. The reasons for doing this are 1) we will use this field in the next part of the lab and 2) it will cause an interesting problem that I would like you to ponder.
* Adjust your mathematics so that the circle paints in the middle of the display panel.

Test your program to prove that you can indeed load you new class and run it properly *without modifying or even recompiling any existing code*! This is a feat that cannot be duplicated with conditional statements.

## Adding a Dynamically Generated Dialog Box to Set a Shape's Properties

***Before* starting this section, copy all the code from part 3 into your part4 subdirectory and make any modifcations to the code in part4. Do NOT modify your part3 code!**

We've gotten a lot to work so far. One last thing to do...

Our system is currently capable of dynamically loading a factory class and using it to create an arbitrary shape. There's a catch however. The parameters necessary to specify how to make any given shape varies from shape to shape. For example, to create a square requires a single value for the length of its side, but to create a triangle requires two values, one for the length of its base and the other for the length of its height. The null shape requires no initialization parameters, but a circle requires a Color object, which is not a numeric value at all!

One of the most important precepts in object-oriented programming is "*delegate to the object that knows best"*. So who knows what parameters a square needs? There's only one choice here, as a square shape is defined by the factory that made it. The SquareFactory must be the one that determines what parameters are needed to instantiate a square shape. Likewise, the CircleFactory is the only object that knows what parameters are needed to construct a circle shape.

The number and type of parameters is clearly a variant here, but then what is the invariant?

As we've mentioned before, sometimes the obvious is the most difficult thing to see: the invariant is that we always *ask* for the parameters to construct the shape.

Thus what we need is a mechanism that enables us to ask for the construction parameters but without know what those parameters are (the invariant process) plus a way to get the values for the necessary specific, concrete parameters but without knowing how to ask for them (the variant process).

## Dialog Boxes

All modern windowed GUIs have a special kind of window called a dialog box that pops up with some sort of question that we must answer before continuing, e.g. "Do you agree to the following license agreement that indentures your first born child? Yes? or No?". These "modal" windows are unlike regular windows (frames) that come up and have a "life" of their own, instead they are tied to a parent window and prevent ("block") the parent from further processing until modal window has been closed.

Dialog boxes are all about asking questions of the user, so they suit our needs quite well here. In the end,what we need is a dialog box which has a panel supplied by the shape factory that has al the necessary labels, buttons and textfields needed to ask the user what parameter values are desired to make the shape instance.

We will take this step by step again, but with less detailed directions now since you are more versed at building GUIs! Reference your ShapeCalc code while writing this code. Don't forget to import any required packages!

1. Create a class shapecalc.AShapePropertyDlg that extends JDialog.
   * You wll need some import statements.
   * Do not mark the class as abstract just yet. We will make it abstract in a little bit, after we are sure that everything is working up to that point.
2. Create a constructor for AShapePropertyDlg that takes two parameters, a JFrame owner and a JPanel property panel.
   * When we are done, the owner will be ShapeCalc and the property panel will be generated by the shape factory.
   * In the constructor, call the superclass constructor with the following 3 parameters, in this exact order: owner, title String (e.g. "Shape Properties") and the boolean value true, signifying that the dialog should be modal.
3. Add whatever import statements are necessary.
4. Test your code using DrJava's interaction pane.
   * First import javax.swing.\*
   * Then instantiate an AShapePropertyDlg object, handing 2 null values to its constructor.
   * Set the dialog's visibility (setVisible) to true.
   * You should see the dialog box pop up and the interactions pane should not accept any input until the dialog is closed.
5. Now using local variables, create 2 JButtons, one that says "Ok" and one that says "Cancel".
6. Likewise, create a button panel (JPanel) to hold the two above buttons.
7. For kicks, let's try a different layout other than the default FlowLayout in the button panel. Let's use a BoxLayout, which lays out components from the left to the right, or from the top to the bottom, all non-centered.
   * Set the layout of the button panel to a new BoxLayout.
   * The constructor for a BoxLayout takes 2 parameters,
     1. the first is the component it is laying out, here, the button panel.
     2. the second is the direction of the layout, here we'll use the static field value: BoxLayout.X\_AXIS
   * Add the two buttons to the button panel. No additional parameters are needed.
8. Add the button panel to the SOUTH edge of the dialog's content pane(defaults to BorderLayout as all frames do).
9. Add the properties panel to the CENTER of the content pane.
10. Using DrJava's interaction pane again, test your dialog box. Be sure to give the constructor a new JPanel as its second input parameter since it will be added to the content pane.
    * Your dialog frame wll probably pop up too small, so add the call, pack(), at the end of the constructor to force the layout manager to set the sizes of the frame to its minimal size that still shows all the components.
    * Your dialog also probably doesn't pop up in the middle of the screen either. Use the same method call as in ShapeCalc to center the dialog on the screen.
11. Write the following two methods with stub code that prints a message to the console as their bodies:
    * protected void okBtnClick(ActionEvent e)
    * protected void cancelBtnClick(ActionEvent e)
12. Back in the constructor, add ActionListeners to the Ok and Cancel buttons using anonymous inner classes. The actionPerformed methods of these listeners should call their respective "click" methods above.
13. Test your dialog box again to make sure that your buttons work properly. When you click a button, you should see the message from the respective click method appear on the console.
14. Now that you've verified that your dialog box is fully operational, we can make it abstract, as the actual click methods will perform duties specific to our ShapeCalc system.
    * Make the entire class abstract
    * Make both click methods (in step 11 above) abstract and remove their stub-coded method bodies completely.

Now all we have to do is to subclass AShapePropertyDlg and override the two click methods with the specific behavior we need. If we instantiate the dialog with a panel filed with all the necessary buttons, textfields, etc needed to construct a shape, then we will have everything we need!

Notice how the development of this abstract class started at with very concrete behavior that we could easily understand and test. Then we abstracted that behavior to get the final expression of our problem. This is a classic technique to develop highly abstract systems. And as usual, the driving motivation through this process is the separation of the variant from the invariant. Here, AShapePropertyDlgrepresents the invariant of asking the use for property values, without the variant collection of the actual properties needed (the property panel) or the variant process of what to do with those properties (the concrete click methods implemented in any subclasses).

## Factories Making Panels

Since the shape factories are going to be creating property panels to input the varous parameters needd to construct their shapes, we need to add another method to the IShapeFactory interface:   
JPanel getPropertyPanel().

For the moment, simply implement that method in each of the factories by having them return a private static JPanel field that was initialized to a plain JPanel instance. Why is it ok to use a static field here?

## Bringing Up a Dialog Box

Now we need to get ShapeCalc to bring up the dialog box. The properties dialog needs to pop up after the factory is instantiated but before the resultant shape is used. Find this point in your code and insert the following code there. You will find that the new code will literally engulf the code where the factory creates the new shape instance.

1. Use an anonymous inner class, subclass and instantiate a new instance of AShapePropertyDlg, overriding the two click methods (we'll fill in their method bodies in a minute).
   1. We've never made an anonymous inner class of a superclass whose constructor required input parameters. But the syntax is very intuitive. In this case, the constructor for AShapePropertyDlgtakes two parameters, so we simply say:   
      new AShapePropertyDlg(param1, param2) { .... }
   2. The first constructor parameter is the host frame, in this case, the ShapeCalc object itself. But since we are currently inside of the anonymous inner class for the newBtn's ActionListener, we can't say "this" because it would refer to the wrong object. Instead, we must say "ShapeCalc.this" to reference the entire ShapeCalc object.
   3. The second constructor parameter is the property panel that the shape factory is supposed to make, so simply use the result of calling the shape factory's getPropertyPanel method.
   4. The instance of the dialog box doesn't need to be assigned to a variable. It simply needs to be set visible. Since it is modal, the execution of the code will not proceed until the dialog closes. So, without assigning to any sort of variable, instantiate the dialog box and set its visibility to true.
2. Now for the bodies of the click methods
   1. The okBtnClick method runs if the user clicks the Ok button in the dialog. The user thus is saying "Yes, please make this shape!" In this situation we need to
      1. Tell the factory to make the shape. We had this code already, now we know where to put it.
      2. Get rid of the dialog box. A dialog box, as any frame, has a dispose() method that will close the window and destroy the object.
   2. The cancelBtnClick method runs if the user clicks the Cancel button in the dialog. The user is thus saying "No, please don't do anything!". In this siutation we simply need to close the dialog down using the dispose() method.
3. Test your program. You should be getting some nice dialogs coming up now with blank panels. Be sure to verify that the Ok and Cancel buttons work properly. Progress, progress!

Now all we need to do is to get the respective shape factories to produce useful property panels and we're all done!

## Property Panels for Squares

The property panel for a square would contain simply a texfield into which one could input the length of the side and a label describing the textfield.

1. The text field needs to be a field of the factory so that it is usable by the makeShape method. It can be static--why?
2. In the constructor of SquareFactory, add the label and the text field to the property panel.
   1. A BoxLayout might be nice here, but with a BoxLayout.Y\_AXIS orientation.
3. All we need to do now is to initialize the field in the IShape anonymous inner class that represents the length of the side of the square.
   1. Unfortunately, the field is a double and the text field where the user input the desired value holds a String.
   2. All the wrapper classes for the Java primitive types have static methods that can "parse" a String and convert it to the associated primitive type. For instance, in this case we can use thefollowing static method: public static double Double.parseDouble(String s)
   3. Use the above method to initialize the square's field to the value in the text field.
4. One problem is that the user may have typed in an invalid number, such as "3.g7" or perhaps nothing at all. If parseDouble tries to parse such an invalid input, it will throw an exception and stop the program.
   1. We need to "catch" the exception should it occur.
   2. Use the following try-catch syntax to replace the entire method body of makeShape:
   3. try {
   4. [put entire original "return ....;" statement here]
   5. }
   6. catch(Exception parseError) {
   7. [set the text field's text to the string "0.0"]
   8. [call makeShape() again]
   9. }

* 1. This crude exception handling will simply replace any invalid input with "0.0". But it does keep the program from crashing!

If you look simply at the code in ShapeCalc, it does not look like the newly constructed shape is at all connected to the property panel used to configure it. They are created and used in different sections of the code after all. But when you look at the SquareFactory, we see that the property panel and the new IShape instance are indeed connected: **the text field of the property page is in the closure of the the shape object.**Hence the shape object can reference the values of the property panel!

Closures give us a tremendously powerful communications pathway that leaves our systems beautifully decoupled!

## Property Panels for Circles

The property panel for the CircleFactory is very similar to that for the SquareFactory, but it does need to use another label and a button.

The functionality that you want to achieve for the property panel includes the following:

1. The button to select the color should be the currently selected color.
2. The currently selected color should persist between usages. That is, for instance, if red was selected and then any number of non-circle shapes made, then the next time CircleFactory is used, then red should be the default color.
3. Canceling the creation of a new shape should *not* affect the color of the current shape, even if it is a circle.
4. As with SquareFactory, an invalid radius input value will cause a shape of radius=0 to be made.

Here's some hints and tools to use:

* Base your CircleFactory code on your SquareFactory's code. Just extend it to meet your new requirements.
* A JButton object has a method called setBackground that takes a Color object as an input and sets the background color of the button.
* The JColorChooser class has a static method called showDialog which will bring up the standard modal dialog box for choosing colors.
  + The signature for showDialog is   
    public static Color showDialog(Component parent, String title, Color initialColor)
  + The return value of showDialog is the color that was selected by the user. (Note: null is returned if the user cancels, but null usually appears as black).
  + The parent Component is the property panel.
  + The title is the title at the top of the dialog box.
  + The initialColor is the color that should be selected initially. See functionality requirement #2 above.
* You may need to set the color button's background color in more than one place in your code.

Depending on how you implemented things, you may observe some odd behavior when you run the program. Try the following sequence of actions:

1. Create a new circle shape using the CircleFactory. Select any color and size you want. The shape is made correctly.
2. Start to create another circle shape using CircleFactory again. Select a new color but don't make the shape yet.
3. Notice that even though the new circle shape has not been made, the color of the old circle shape has changed to the newly selected color.
4. Even if you cancel the construction of the new shape, the old shape retains the new color.

Why is this happening? Try to explain the phenomenon in terms of closures.

Fix the problem if it occurs in your program so that you can achieve functionality #3 above.

### More stuff to do...

* Make more kinds of shapes!
* Make shapes with more kinds of construction parameters
* Let your imagination run wild--have fun!