Chapter 4 Graphics and Inheritance

“One is not likely to achieve understanding from the explanation of another.”

Takuan Soho

Introduction
This chapter will give you exposure to and practice with writing classes in Java. It will also illustrate how to do simple graphics and introduce inheritance, a powerful feature of object oriented programming. Like the last chapter, it will not present all the details of the constructs used; that will be delayed until the next chapter. For now, try to become familiar with the process of thinking a problem through, coming up with an elegant design for a solution, then implementing and testing it -- those are the important lessons that will carry over into other programming languages and possibly even other areas. To learn to program, you must practice, reading about it is not good enough (as the Takuan quote implies).

A Description of the Task
Your task in this chapter will be to draw two eyes on the screen. For simplicity you need only draw the iris and pupil. Make the iris the exact same color as yours. The distance between the two eyes and the size of the pupil relative to the iris should be adjustable by the user.

Creating a Prototype
In the previous chapter you were introduced to the techniques of: a) Building a prototype then gradually adding functionality, and b) Sketching the GUI, then creating and testing it before writing any other code. Do that now. First, create a new project in Netbeans. Add a GUI Frame called EyeFrame. Add and connect however many Buttons you will need. Compile and run your project, testing to make sure everything works so far (i.e. that all the Buttons invoke the correct actionPerformed() methods). Once you do that, you will be in a position to try out the various Graphics commands as you work through the chapter.

Object Oriented Design -- Choosing Classes to Implement
When designing a program to solve some problem an early decision you must make is what classes you will write. Since your task is to draw two eyes on the screen, one natural candidate for a class would be Eye. Since an Eye consists of an iris and a pupil, Iris and Pupil are also candidate classes. How many classes make sense in a particular context is less than perfectly defined. For now, let’s assume you will need an Eye class, and put off the decision on Iris and Pupil until you know a little more.

Before designing the Eye class, there are several facts about Graphics and Color in Java that you need to know. Often in designing a class you must do some experiments, play with the elements involved, and learn about the related classes Java provides, before you know enough to make informed decisions. These next several sections will illustrate that process; then we will return to the Eye class; and once the Eye class is done, so is our task!
The Graphics Class
Java provides the Graphics class (in java.awt) to draw on the screen. A Graphics object provides a context in which Java graphics operations can be performed. In other words, to draw on the screen you must first have a Graphics context in which to do so. You will learn how to draw on the screen from a Frame by including a public void paint(Graphics) method.

Components and public void paint(java.awt.Graphics)
When you write a public void paint(Graphics) method in your EyeFrame, it overrides the default paint() method in Frame. That sentence requires a bit of explanation. Look at the heading of the EyeFrame class definition (in the EyeFrame.java file). The first line of code is:

```java
public class EyeFrame extends javax.swing.JFrame
```

Thus, the class is named EyeFrame, it is public and it is a subclass of javax.swing.JFrame (in other words, it extends javax.swing.JFrame). When you extend a class, instances of the subclass inherit all the functionality of the superclass. To add functionality, you simply add methods. To change the behavior of a method in the superclass, you write a subclass method with the same signature that does something different. Because EyeFrame extends Component (actually it extends Window, which extends Panel, which extends Container, which extends Component, but never mind right now), it automatically inherits a paint() method (which does very little). If you want to paint your Frame differently (by drawing a circle or whatever on the screen) you write a public void paint(java.awt.Graphics) method in your subclass, and then that is executed instead of the method in the superclass. You will see examples of adding and modifying functionality in the FilledCircle class, below.

Basics of Graphics in Java
To start using graphics in Java, you must understand the Graphics coordinate system, and a few simple methods.

The Coordinate System
From the perspective of a Java graphics context, the drawable area is a rectangle of dots numbered from left to right and top to bottom. The dots are called picture elements or pixels. The pixel numbered (0,0) is in the upper left corner.

A Few Graphics Methods
The only method you need to draw an eye is drawOval(), but that will be easier to understand after you know how drawRect() works. The drawRect() method has four parameters, all ints. The first two specify the upper left corner of the rectangle; the third and fourth, its width and height. In each pair, the first is horizontal, the second is vertical. Thus drawRect(x,y,width,ht) will draw a rectangle whose upper left corner is at (x,y), whose width is width, and whose height is ht.
The `drawOval()` method is similar. The four parameters are identical, specifying a rectangle, exactly as in `drawRect()`; the oval is inscribed in the specified rectangle.

The `drawLine()` method also has four parameters; the first two specify the coordinates of one end of the line, the second two, the other. See Listing 4.1 for an illustration. See Figure 4.1 for its result.

**Listing 4.1 A paint() method**
```java
1    public void paint(java.awt.Graphics g) {
2        g.drawRect(25,25,100,100);
3        g.drawOval(25,25,100,100);
4        g.drawLine(0,0,350,200);
5        g.drawString("g.drawRect(25,25,100,100);", 20, 150);
6        g.drawString("g.drawOval(25,25,100,100);", 20, 165);
7        g.drawString("g.drawLine(0,0,350,200);", 20, 180);
8    }
```

*Line 2:* Draws a square with sides 100 pixels long, upper left corner at (25,25)
*Line 3:* Draws a circle centered in it, i.e. centered at (75,75), not (25,25).
*Line 4:* Draws a line from one corner of the graphics context to the other.

Figure 4.1a shows the result of the `paint()` method in Listing 4.1, if it is part of an `Applet`. The coordinates of the corners of the graphics context and the square are indicated.
Figure 4.1b (from a Frame): the result of the paint() method in Listing 4.1, if it is part of a Frame. Note that (0,0) is behind the title bar! Everything else (except the resize thingee in the lower right) is the same.

Notice that if we were hoping for a circle centered at (25,25) we did not get what we wanted. We will have to take this into account in writing the graphical display method for the Circle class.

The Circle Class -- Design and Implementation
Circles are used to represent many things in GUIs. In a simulation circles might represent molecules; in a game, balls; on a map, populations in cities, or incidence of infectious disease.

As you have just seen, you can draw a circle in a graphics context by using
\[ g\text{.drawOval}(int, int, int, int) \]. But, if you were writing a program that displayed many circles, or if the circles moved around the screen (like the balls in a billiards game), or changed sizes (like graphics representing levels of infection), you wouldn’t want to keep finding and changing the appropriate drawOval() code. If you tried, it would take a lot of careful attention to avoid changing the wrong one. A better solution is to hide the information about a particular circle, namely where and how big it is, inside an object. Then when you have multiple circles, you can just deal with them as Circles and let the details of where they are right now and how to draw them be handled by the Circle class. Additionally, if you need to change how they are drawn, there is only one piece of drawing code to change instead of numerous copies of it. Thus, you can avoid complexity and bugs at the same time. What a deal!

Circle Class Design
As with any class, in designing the Circle class, you must decide: 1) what state information it will contain, and 2) what actions it will support, including how it will be displayed and tested.
State -- What Information Completely Describes the State of a Circle?
To completely describe a circle you must specify its center and its radius; that’s it. So we will need three variables; one for its radius and two for its position (x, and y). For simplicity, and since the screen is made of discrete pixels, these can all be whole numbers, ints. For now. When we use Circle to display molecules later, it will turn out to be crucial for their positions to be able to be intermediate between pixels.

Action -- What Must a Circle Do?
We will need to be able change the size of a circles on the screen and where they are located (Why? Look back at the task we are writing Circle for if it’s slipped your mind). Thus, we need accessors for all our variables, so that we can discover and/or change the position or size of a Circle. We will also need methods to display a Circle, both for debugging and to display it when the program is running.

Converting the Design to Java Code
The next step, after designing a class is to convert that design to Java.

Creating the Circle Class
Create a Circle class (by New/File/Java Classes/Java Class, and name it Circle!).

Variables
Perhaps you already know how to declare the three variables? If not, here’s how:

```java
int x;
int y;
int radius;
```

Accessors
These are just like the accessors for Account (See Listing 4.2 or class notes) except the names of the variables are different. Look back at that example and try to write the accessors for x, before looking at them in the Listing below.

Listing 4.2 Accessors for the Circle Class

```java
1    public int getX() {return x;
2    public int getY() {return y;
3    public int getRadius() {return radius;
4
5    public void setX(int nuX) {x = nuX;
6    public void setY(int nuY) {y = nuY;
7    public void setRadius(int nuRadius) {radius = nuRadius;
```

The toString() Method
Java has a special method that is used almost exclusively for debugging. It is called toString(), and its signature is: public String toString(). As the name implies (once you are familiar with colloquial Java-speak) it converts an object to a String. While you are testing your classes (or debugging in general) you sometimes need to know what information is in an object, whether it contains the information you expect. If you have an object called anyObject, you can always find out what’s inside by:
System.out.println(anyObject);

You don’t need to type `toString()` because in the context of a `System.out.println`, Java automatically adds it for you (although you may type it if you want).

You can write any `toString()` method that you choose, so long as the signature matches. Listing 4.3 shows a `toString()` for the `Circle` class that is a bit verbose. Why it is written this particular way will become obvious in the next chapter.

Listing 4.3 `toString()` for the `Circle` Class

```java
public String toString() {
    String returnMe = "I am a Circle: ";
    returnMe += "x=" + getX();
    returnMe += "y=" + getY();
    returnMe += "radius=" + getRadius();
    return returnMe;
}
```

Line 2: Declare the String variable to return; set it to "I am a Circle: ">
Line 3: Paste a tab (\t) onto it, followed by "x=" and the value of x.
Line 6: Return that whole String as the value of `toString()`

Testing Your Code

That’s enough code to test. Create a `Circle` and check that all the methods work. To do that, after you instantiate the `Circle`, display it, then change all the variables and display it again. Since `toString()` uses `getX()`, `getY()` and `getRadius()`, by doing that you have used all the methods. See Listing 4.4 for how the code might look.

Listing 4.4 Testing the First Prototype `Circle` Class

```java
public static void main(String[] args) {
    Circle aCircle = new Circle();
    System.out.println("before " + aCircle);
    aCircle.setX(123);
    aCircle.setY(17);
    aCircle.setRadius(34);
    System.out.println("after " + aCircle);
}
```

Line 2: Instantiate a `Circle` called `aCircle`
Line 3: Display it.
Lines 4-6: Set all the variables
Type and run this test program (if you made mistakes, debug your typing). Then, once you know the `Circle` class can keep track of and change its variables correctly you are ready to add the graphical display.

Displaying a `Circle` Graphically

To display a `Circle` graphically requires a method that draws the right sized circle in the correct location. It seems this should only take one line of code. Something like:

```java
g.drawOval(x, y, width, ht);
```
But, what values should we use for $x$, $y$, width and $ht$? And where does the graphics context, $g$, come from?

**public void paint(java.awt.Graphics)**

A Circle knows its location and size (the location of the center is in its $x$ and $y$ variables and its size is in its radius variable). There is no way a Circle should know anything about graphics contexts, so that is best provided from the outside, by whatever method asks the Circle to display itself. This is what parameters are used for, to pass information to a method. The Frame was displayed graphically by a method called paint(); to keep down the cognitive overhead, we will use the same name, paint(), for the method that displays a Circle graphically. So, perhaps all we need is: $g$.drawOval($x,y,radius,radius$); as in Listing 4.5.

**Listing 4.5 A First Try at a paint() Method for Circle**

```java
1    public void paint(java.awt.Graphics g) {
2        g.drawOval(x,y,radius,radius);
3    }
```

**Testing the paint() Method**

Add the paint() method from Listing 4.5 to your Circle class and test it by modifying your EyeFrame to create a Circle, set its $x$, $y$, and radius variables to 100, and display it graphically in paint() (see Listing 4.6).

**Listing 4.6 Creating and Displaying a Circle Graphically**

```java
1    public class EyeFrame extends java.awt.Frame {
2        Circle aCircle = new Circle();
3
4        /** Initializes the new EyeFrame */
5        public EyeFrame() {
6            initComponents();
7            setVisible(true);
8            setSize(350,200);
9            setTitle("An EyeFrame!");
10
11            aCircle.setX(100);
12            aCircle.setY(100);
13            aCircle.setRadius(100);
14        }
15
16        public void paint(java.awt.Graphics g) {
17            aCircle.paint(g);
18        }
19    }
20
21    Lines 16-18: Set the variables in aCircle to 100.
22    The code in Listing 4.5 draws the circle that would fit inside a square of size radius, whose upper left corner is at $(x,y)$. That has two problems: 1) it is centered at $(x+radius/2, y+radius/2)$, and, 2) its diameter is the radius of the Circle (To understand this, draw yourself a picture labeled with coordinates.)
**Problem Solving Technique: Draw a picture.**

By drawing a picture, you can engage your visual-spatial processing system. Although people tend to take it for granted, the ordinary ability to walk through a crowd involves a feat of information processing. Your visual-spatial processor is more powerful than any computer on the planet, but so long as you are stuck in linguistic space it is idle. Drawing a picture can activate it. And when you look back at the picture you will remember what you were thinking.

Not interested in drawing right now? Then, it’s time to put this down and do something else. You can’t, can’t, cannot, program without understanding. It’s hopeless. So, either take the time, spend the effort to understand it, or, do something else! No sense wasting your time. That second problem is very easy to fix, simply pass radius*2 instead of radius for the width and height to drawOval() (see Listing 4.7).

**Listing 4.7 A Second Try at a paint() Method for Circle**

```java
public void paint(java.awt.Graphics g) {
    g.drawOval(x,y,radius*2,radius*2);
}
```

The size is right, but the circle is still in the wrong place; the center is at (x+radius,y+radius), instead of (x,y). How could you fix this? The simplest way is just to subtract the radius from x and y in the parameters you send to drawOval(), as seen in Listing 4.8.

**Listing 4.8 A Correct paint() Method for Circle**

```java
public void paint(java.awt.Graphics g) {
    g.drawOval(x-radius,y-radius,radius*2,radius*2);
}
```

**Line 2:** Draws a circle whose radius is radius, centered at (x, y). In the context of a particular Circle, x, y, and radius are variables specifying the state of that Circle. Modify your code. Execute it to verify that the circle is displayed at the appropriate place.

**More than One Circle**

Now that you have a working Circle class, you can create and display as many Circles as you want. Add another Circle, as shown in Listing 4.9. Execute it to make sure it is working properly; the circles should be concentric. Add another, intersecting Circle to make sure you understand the procedure (don’t forget to add a line in paint() to display the third one!). Notice that you can add and display as many Circles as you want without ever looking back at the Circle class code. This is a huge advantage of programming with objects; once a class is written, you can forget the details inside it.

You are almost ready to design and implement the Eye class, as soon as you know a bit about color in Java.

**Listing 4.9 Adding a Second Concentric Circle, by Modifying Listing 4.6**

```java
public class EyeFrame extends java.awt.Frame {
    Circle aCircle = new Circle();
    Circle bCircle = new Circle();
}```
/** Initializes the new EyeFrame */
public EyeFrame() {
    initComponents();
    setVisible(true);
    setSize(350,200);
    setTitle("An EyeFrame!");
    aCircle.setX(100);
    aCircle.setY(100);
    aCircle.setRadius(100);
    bCircle.setX(100);
    bCircle.setY(100);
    bCircle.setRadius(50);
}
public void paint(java.awt.Graphics g) {
    aCircle.paint(g);
    bCircle.paint(g);
}
Changes to Listing 4.6 are in bold. If you wanted a non-concentric Circle, change the parameters in setX() and setY() for bCircle.

The Color Class

Setting the Color of the Graphics Context
A graphics context has a number of state variables, including the current color. The default color is black. You can change it with the accessor setColor(Color); i.e. you set the color of a Graphics object by sending it a setColor() message with a Color as the parameter. Just as you set the balance of an Account by sending it the setBalance() message with an int as the parameter, or the radius of a Circle using setRadius().

Built in Colors
The Color class has about a dozen colors predefined. To set the color to red, you would say:
g.setColor(java.awt.Color.RED); Add that line between lines 18 and 19 in Listing 4.9 and execute the Frame. Notice that only the second circle is red; if you move the setColor() before line 18, then both Circles will be red.

Creating Your Own Colors
There are millions of colors possible in Java. You can create any of them by saying:

    java.awt.Color myColor = new java.awt.Color(red, green, blue);

The three int parameters to the Color constructor set the intensities of red, green, and blue. All three must have values in the range, 0 to 255.
Exactly how many colors are available?
You can calculate this from the fact that there are three parameters, each of which can take on 256 different values. It’s very much like the analysis in “Problem Solving Principle #1, in Chapter 1.

RGB Color Model
Java has two color models, but the simpler is the RGB model. RGB stands for red-green-blue. The color of each pixel is determined by the amount of illumination in those three colors. Any combination of values for red, green and blue is legal. To get pure red, you say

```
new java.awt.Color(255, 0, 0);
```

thus passing 255, 0, and 0, as the parameters to `new java.awt.Color(); 255 for red (i.e. all the way on), 0 for green and blue (i.e. all the way off). Purple is a mixture of red and blue. So, for bright purple you would pass (255,0,255); for dark purple, perhaps (50,0,50).

The Difference Between Pigment and Light
The RGB values set the intensity of light emitted in each color. You are probably more used to mixing pigments than light. Light and pigment are not identical. If you mix blue paint with red paint, you get purple. If you then add yellow you get muddy brown (or possibly even black). When you mix red light with blue light you also get purple; but if you then add green, you get white! A combination of all colors of light yields white light. Think of a prism. It breaks white light into its constituents. If there is no pigment, white paper remains white; if there is no light, everything is black. So, `new java.awt.Color(255,255,255)` is white, `new java.awt.Color(0,0,0)` is black.

The Eye Class: Design and Implementation
As always, before writing a class, you should decide what it will do and how it will do it; this is referred to as design.

Designing an Eye Class
For your purposes here, an `Eye` is two concentric `Circle`, the larger (the iris) filled with the color of your eyes (some shade of brown, blue, or green), the smaller (the pupil) filled with black. If you wanted two unfilled black circles, the `Eye` class could have two `Circle` variables and you’d be almost done already.

For the user interface, you must allow the user to move at least one eye horizontally, and adjust the size of the pupils. The simplest way to handle this is with two `Buttons` to move one eye right and left; and two more to grow and shrink the pupils (if your interface uses some other scheme, that’s fine). Thus, you will need methods to change the size and location of an Eye. Fortunately these will be very easy. Assume an `Eye` had two `Circle` variables. When the user wanted to shrink the pupils, a `shrink()` message would be sent to the `Eye`. The `shrink()` method could then reduce the size of the pupil `Circle` using `getRadius()` and `setRadius()` (i.e. `iris.setRadius(iris.getRadius()-3)`). Similarly, the `moveRight()` method could adjust the locations of both `Circle` using `getX()` and `setX()`.
The Circle class does almost what you need already. Two things need to be modified. Instead of drawing a circle in black, you want it to fill the same circle in a particular color. One way to accomplish this would be to modify the Circle class. You could change drawOval() to fillOval() in paint(), add a Color variable and use it to set the color of the graphics context before you fill the circle. But, then if you wanted to be able to draw unfilled circles you’d need to re-modify the Circle class. Instead, we will extend the Circle class. That way, you won’t have to change the Circle class and code reuse can be illustrated by subclassing.

**Inheritance:** class FilledCircle extends Circle

As mentioned above, when you extend a class, the subclass inherits the data and methods of the superclass. Thus FilledCircle can use x, y, and radius without redeclaring them. Plus, you can add additional methods to add functionality, and override existing methods to change functionality.

**Design**

FilledCircle is very much like Circle; it only needs one additional variable and to override one method.

**Variables**

FilledCircle inherits x, y, and radius from Circle. It needs one additional variable, to keep track of its color.

**Methods**

There must be some way to set the color of a FilledCircle, so it will need a setColor() accessor. The paint() method must be modified to fill the circle in that color instead of just drawing the outline of the circle.

**Implementation**

Implementation is straightforward: create the class, add the variable, and add the method.

**Create a FilledCircle Class**

You do remember how, right? If not, look back [here](#).

**Add a Color Variable**

Here’s how to declare a variable of type Color called myColor:

```java
java.awt.Color myColor = new java.awt.Color(100,0,100);
```

This line auto initializes myColor to a medium purple. This default color will help in debugging: anytime you see it, you will know that you forgot to set the color for this FilledCircle.

Some people don’t like to type, or look at, "java.awt." over and over. If you would prefer to just type:

```java
Color myColor = new Color(100,0,100);
```
See Listing 4.11 for a technique to allow this.

Add the Accessor to Set the Color of the FilledCircle

To be able to change the color of a FilledCircle, there must be an accessor. The standard name is `setMyColor()` and its form is identical with the other accessors you’ve seen; see Listing 4.10. Before long accessors like this will be second nature. For now, realize that there is one parameter of type `Color`, named `c` (line 7, `java.awt.Color c`); and whatever value is passed through that parameter is stored in the instance variable named `myColor` (line 8, `myColor = c;`).

Override `paint()`

The heading of `paint()` is `public void paint(java.awt.Graphics g)`. Thus, the parameter is of type `Graphics` and is named `g` locally (i.e. in the `paint()` method). The body of the method must first set the `Graphics` color to the color of this particular `FilledCircle`, then draw the filled circle. See Listing 4.10.

Listing 4.10 The FilledCircle Class

```java
public class FilledCircle extends Circle {
    java.awt.Color myColor = new java.awt.Color(100,0,100);

    /** Creates a new instance of FilledCircle */
    public FilledCircle() {}

    public void setColor(java.awt.Color c) {
        myColor = c;
    }

    public void paint(java.awt.Graphics g) {
        g.setColor(myColor);
        g.fillOval(x-radius, y-radius, radius*2, radius*2);
    }
}
```

*Line 2: Declare a Color named `myColor` and initialize it to medium purple*

*Lines 7-9: Accessor to set the color of a FilledCircle*

*Lines 11-14: Paint the FilledCircle by setting the graphics color, then `fillOval()`*

See Listing 4.11 for a way to avoid typing `java.awt.` over and over.

Listing 4.11 Simplified FilledCircle Class Using `import`

```java
import java.awt.*;
public class FilledCircle extends Circle {
    Color myColor = new Color(100,0,100);

    /** Creates a new instance of FilledCircle */
    public FilledCircle() {}

    public void setColor(Color c) {
        myColor = c;
    }

    public void paint(Graphics g) {
        g.setColor(myColor);
        g.fillOval(x-radius, y-radius, radius*2, radius*2);
    }
}
```

12
This import statement allows you to skip typing `java.awt.` before `Color` and `Graphics`. Compare to Listing 4.10.

**Testing FilledCircle**

Modify your existing `Frame` to test `FilledCircle`. It will be enough to simply change the `Circles` to `FilledCircles` and set the color of the smaller one to black. What will it display if it is working correctly? See Listing 4.12 for the necessary changes.

Listing 4.12 Test code for `FilledCircle`; Changes from Listing 4.9 Are in **Bold**

```java
public class EyeProtoFrame2 extends java.awt.Frame {
    FilledCircle aCircle = new FilledCircle();
    FilledCircle bCircle = new FilledCircle();

    /** Initializes the new EyeFrame */
    public EyeFrame() {
        initComponents();
        setVisible(true);
        setSize(350, 200);
        setTitle("An EyeFrame!");
        aCircle.setX(100);
        aCircle.setY(100);
        aCircle.setRadius(100);
        bCircle.setX(100);
        bCircle.setY(100);
        bCircle.setRadius(50);
        bCircle.setColor(java.awt.Color.black);
    }

    public void paint(java.awt.Graphics g) {
        aCircle.paint(g);
        bCircle.paint(g);
    }
}
```

**Lines 2-3:** Declare, instantiate, and store `FilledCircles` instead of `Circles`.
**Line 23:** Set the smaller’s color to black so it won’t be purple!

**The Eye Class**

Having built and tested a GUI Frame and a `FilledCircle` class, most of the work of building the `Eye` class is finished. Create an `Eye` class and add the following variables and methods.

**Variables**

An `Eye` has an iris and a pupil; these are both `FilledCircles`. Thus:

```java
    FilledCircle iris = new FilledCircle();
    FilledCircle pupil = new FilledCircle();
```

**Methods**

Because an `Eye` is composed of two `FilledCircles`, most `Eye` methods will simply send the appropriate messages to those `FilledCircles`. 
**MoveLeft and MoveRight**

To move an Eye left you must move both of its FilledCircles left, so the `moveLeft()` method would simply set `x` in each to a slightly smaller number; see Listing 4.13.

Listing 4.13 moveLeft() for Eye

```java
1    public void moveLeft() {
2        iris.setX(iris.getX()-2);
3        pupil.setX(iris.getX());
4    }
```

Line 2: Set the `x` coordinate to 2 less than it was.

Line 7: Set the pupil `x` variable to the same value.

The `moveRight()` method would be similar, except increasing `x` for each.

**ShrinkPupil and GrowPupil**

To shrink the pupil you can simply reduce the value of the `radius` variable of the pupil FilledCircle; see Listing 4.14.

Listing 4.14 shrinkPupil() for Eye

```java
1    public void shrinkPupil() {
2        pupil.setRadius(pupil.getRadius() - 2);
3    }
```

Line 2: Set the radius to 2 less than it was.

The `growPupil()` method is nearly identical. After you add these methods to the Eye class, go back to the `actionPerfomed()` method for the shrink and grow Buttons, and modify them to send those messages. There are two things you must make sure of in doing this:

1. There must be an Eye variable declared before you can send the message to it. Every message has the form `someObject.someMessage();` -- see “The Message Statement” in Chapter 5.

2. To change what is displayed, you must invoke `paint(Graphics)` and to do that you must send the `repaint()` message. The details of this will be explained in a later chapter. For now, just use the code in Listing 4.15.

Listing 4.15 actionPerformed() for growButton

```java
1    private void growButtonActionPerformed(java.awt.event.ActionEvent evt) {
2        rightEye.growPupil();
3        repaint();
4    }
```

Line 2: Send the `rightEye` the `growPupil()` message.

Line 3: Send `repaint()` to the Frame so you can see the new pupil size -- don’t forget this!!

**Composition and public void paint()**

To display an Eye you must display both FilledCircles, first the iris, then the pupil (since if you do it in the other order, the pupil will be invisible). Listing 4.16 shows the simplicity composition gives.
Listing 4.16 paint() for Eye
1    public void paint(java.awt.Graphics g) {
2        iris.paint(g);
3        pupil.paint(g);
4    }

That’s all there is to it.

That’s all the methods we need (Or is it? Check the design to see if we did everything we planned to. Look back at Listing 4.12, which tested the FilledCircle class; did it send any messages besides paint() to the FilledCircles?); so it’s time to test.

Testing
Modify your Frame to create and display one Eye, as in Listing 4.17.

Listing 4.17 Test Code for Eye
1    public class EyeFrame extends java.awt.Frame {
2        Eye rightEye = new Eye();
3
4        /** Initializes the new EyeFrame */
5        public EyeFrame() {
6            initComponents();
7            setVisible(true);
8            setSize(500,500);
9            setTitle("An EyeFrame!");
10        }
11
12        public void paint(java.awt.Graphics g) {
13            rightEye.paint(g);
14        }

Note that unlike Listing 4.12 there is nothing after the setTitle(). Run it. Once you find and eliminate all the typing errors, you should notice that there’s no sign of the Eye. Why not?

Debugging
There are many possible reasons. Maybe it’s never being sent paint(). Maybe it is painted in white. Maybe it’s being drawn off the screen. Maybe it is so small you can’t see it. Maybe something else is being drawn on top of it. The job of the programmer, at this juncture, is to determine the cause of the problem and fix it. Assuming it is one of the reasons listed above, how could you go about determining which it is? The answer is, use the scientific method. Design and carry out experiments to verify or eliminate each of those hypothetical bugs. Until you determine what is causing the problem, it will be difficult to fix.

You might start by making the Frame window bigger; maximize it and see if the Eye appears. Or, you might push the “grow pupil” Button; do it several times. This assumes that you have modified the event handling code for that Button so that it sends the growPupil() method to the Eye. If you haven’t added that code yet, do so now.
In the author’s Frame, after he pushed the grow Button several times, he was surprised to see a quarter of a purplish circle expanding from the upper left corner. Having seen this effect before, he immediately realized that the reason he didn’t see anything at first was that the radius, x, and y, were all zero. Do you know why? The default initial value of instance variables is zero (see the paragraph titled, “Putting it all Together -- Finally!”), in Chapter 3.

If you compare Listing 4.12 (the Frame to test FilledCircle) and Listing 4.17 (to test Eye), you will notice that init() in the former sets x, y, and radius for both FilledCircles and sets the color of the smaller to black; in the latter it does not. Somehow we must specify the location of the Eye and make its pupil black.

There are a number of ways we might set the initial size and location of an Eye. For now, simply add setX(), setY() and setRadius() methods to Eye, and send these messages to the Eyes in init(). To setX() for an Eye, all you need to do is send setX() to both the iris and the pupil. For setRadius(), send setRadius() to both, but make the radius of the pupil smaller.

A maxim of object programming is for classes to know the minimum. It makes sense for the Frame to control the location of the Eye, and possibly the size. Nevertheless, every Eye will have a black pupil, so the right place to set the color of the pupil is in Eye, not Frame.

You may have noticed this code (written by Netbeans) in Eye.java.

```java
/** Creates a new instance of Eye */
public Eye() {

}
```

This looks like a method without a return type, with the same name as the class. It is called the default constructor, and is invoked when you say new Eye(). If there is any initialization code for instances of a class (i.e. anything that needs to be done once, right when an instance is created), it goes in the default constructor. So that is where the code to set the color of the pupil to black goes. See Listing 4.18 for the code you should add.

Listing 4.18 Additional Code for Eye

```java
1 /** Creates a new instance of Eye */
2 public Eye() {
3    pupil.setColor(java.awt.Color.BLACK);
4 }
5
6 public void setRadius(int r) {
7    iris.setRadius(r);
8    pupil.setRadius(r/2);
9 }
10
11 public void setX(int x) {
12    iris.setX(x);
13    pupil.setX(x);
14 }
```

**Line 3:** Sets the color of the pupil to black (so it won’t be purple).
Lines 6-9: To set the radius of the Eye, set the radius of the iris to the parameter, set the radius of the pupil to half that.

Lines 11-14: To set x for the Eye, setX() for both its iris and pupil to the parameter.

Add that code, then run your program again. If you’ve made no mistakes, it will display an Eye with a black pupil. Chances are you have made one or more mistakes. If so, figure out what’s gone wrong. Don’t panic! Just pick up the balls and keep practicing. Try out the buttons. Do they work? Did you write code for each one?

Assembling a Working Eyes Program

Now that you have a working Eye class and a Frame with buttons to adjust it, accomplishing the task of displaying two of Eyes is fairly trivial. Probably you already know what needs to be done. There are four things, all in the EyeFrame class.

1. Declare the second Eye (at the top)
2. Set the size and position of the second Eye (in EyeFrame())
3. Display the second Eye (in paint())
4. Resize both pupils (in actionPerformed() for the shrink and grow Buttons).

These should all be simple since the code is already there for rightEye.

Make those changes, and test your code. The only thing remaining now is to make the Eye color match yours. You could experiment with changing the RGB parameters on line 3 of Listing 4.11, but that means you’d have to recompile each time. A more efficient (and fun) technique is to use NetBean’s Color Editor (all you need do is: In the Form Editor, select a Button, then in Properties click the ... button to the right of background, click RGB and slide the sliders).

Conclusion

This chapter developed a program to display two eyes the color of the programmer’s in a Frame. It did so by designing and implementing a Circle class, extending that to a FilledCircle, and finally building an Eye class that was composed of two FilledCircles. It thus illustrated both mechanisms for code reuse: inheritance and composition. It also illustrated the use of simple Java Graphics and Color plus walked through the process of developing a program incrementally.

A novice programmer would have spent roughly 3-6 hours to work through this chapter; there are so many details that needed to be correct. There is no substitute for spending the time to learn to program. Like juggling, you simply can not learn to do it by reading about it or watching someone else do it. Is the investment of time and energy to gain this skill a good one? Consider what you might do with that time otherwise. If the time would have been spent watching TV or playing video games... odds are you can finish that sentence.

The next chapters will review the material glossed over here in a more detailed fashion. If you choose to continue, see you in the next chapter!
Review Questions
4.1 Why are prototypes useful to build first?
4.2 Why is design important?
4.3 What are the first two things to do in design?
4.4 What is a graphics context?
4.5 What message do you send to an Frame to cause paint() to happen? Does it have parameters?
4.6 What are parameters for?
4.7 What are the two techniques of class reuse?
4.8 What does pixel mean?
4.9 How many colors (exactly) are possible in Java?
4.10 What are the parameters for drawRect()? fillRect()? drawOval()? fillOval()? drawLine()? setColor()? 
4.11 How do you change the size of the Frame (so it stays changed!)?
4.12 What are accessors for?
4.13 How do you fix a bug you can’t find?

Programming Exercises
4.14 Write the paint() method for Circle.
4.15 Write the accessors for Circle.
4.16 Write the Circle class.
4.17 Write the FilledCircle class.
4.18 Create a Target class that is displayed as alternating red and white bands of color. Hint; draw the biggest fillOval first and work in.