Chapter 4

Classes

In this chapter you will learn how to design classes in order to write more interesting and understandable programs. So far when we have written a program, it has been by defining a single class that extends `WindowController`. The program's behavior results from defining methods with fixed names like `begin`, `onMousePress`, `onMouseClick`, etc. Now we will be designing very different classes that can generate objects that can work together in programs.

All of the geometric objects that we have been using so far have been generated from classes that have been provided for you. The classes include `Location`, `FilledRect`, `FramedRect`, `FilledOval`, `Line`, etc. The first classes we will be designing will provide methods and constructors similar to those used in the geometric classes. The Java code that we will write to support these will be similar to that which we have already written in earlier programs. We will be declaring constants and instance variables and defining methods in order to provide the desired behavior of objects.

The focus of this chapter is on how to avoid complexity by using classes. Classes allow programmers to design programs as the interaction of objects, each of which is capable of certain behaviors. While each of these behaviors may be relatively simple, the interactions of the objects allow relatively complicated tasks to be handled simply. In particular, each object will take responsibility for a collection of actions that contribute to the program. Because these behaviors can be treated as simple operations, assembling the final program should be relatively straightforward. As a result, the design of classes is one of the most important aspects of object-oriented programming.

4.1 An example without classes

While the focus in this chapter will be on designing and writing classes to generate objects, we will begin with a simple example in which we do not create a new class. The reason for starting this way is so that we can contrast this approach with one where we define a new class to handle some of the more complex aspects of the program. Our eventual goal is to convince you that the creation of new classes makes the programming process easier, and the possibility of reusing these classes can save work in later projects.

Recall the program `WhatADrag` from Figure 3.5. This program allowed the user to drag a rectangle around the screen. Only minor changes would be required to replace the rectangle by an oval. The name of the instance variable `box` would likely be replaced by a name like `circle` and its type would be changed from `FilledRect` to `FilledOval`. Then the assignment statement in the `begin` method would call the construction for `FilledOval` rather than `FilledRect`. However, aside from these minor changes of names, the structure of the program remains essentially the same.

Figure 4.1: Funny face generated by `FaceDrag`

These minor changes work fine as long as we use a geometric object supported by our libraries, but suppose we wanted to drag around a triangle, which is not supported by our library. Now we would have to work harder as we would have to determine how to build the triangle from three straight lines, and we would have to figure out how to determine whether a point is contained in a triangle.

We could have been lucky and had the library support a triangle, but then we might have been missing classes to support a pentagon, hexagon, or some other polygon that we needed for some particular problem. Our point here is that we will often not be lucky enough to have exactly the classes we need to be supplied by existing libraries. In those cases we will have to figure out a way of handling this ourselves.

In this chapter we will illustrate the use and implementation of classes in a more fanciful example of a funny face as shown in Figure 4.1. The program in Figure 4.2 displays a funny face and allows the user to drag it around the screen, very much like program `WhatADrag`. Because the funny face has several geometric components, the main differences with `WhatADrag` are that there are several objects that need to be moved with each drag of the mouse. The `head`, `leftEye`, `rightEye`, and `mouth` are declared as instance variables and are created in the `begin` method. As in `WhatADrag`, `lastPoint` will be an instance variable that keeps track of where the mouse was after the last mouse press or drag.

When we move the face in `onMouseDrag`, all four pieces must be moved. This works fine, but imagine how much longer the method would be if the face had many more pieces. Worse yet, suppose we wanted to have two or more faces on the canvas at a time. It would rapidly become hard to keep track of the various pieces without forgetting one of them.

Wouldn't it be easier if we could construct a funny face with a single construction, just as we did a rectangle? It would be especially easy if this funny face had methods like `move` and `contains` that we could use without having to specify all of the details to manipulate the face.

For the moment, suppose our wish was granted and that we were provided with such a class. Assume class `FunnyFace` has a construction that does all of the initialization performed in the `begin` method of `FaceDrag`. Furthermore assume it has methods `contains` and `move` that determine if a `Location` is inside the funny face and that move the funny face, respectively. We can use this class exactly as we did the filled rectangle in the program `WhatADrag` in Figure 3.5 of the last chapter.

The code for class `RevFaceDrag` is given in Figure 4.3. It is nearly identical to that of the original `WhatADrag`, and is considerably simpler than that for `FaceDrag`, because we don't have to worry about handling all of the different pieces of the funny face. We can treat the face as a single object.

```
public class FaceDrag extends WindowController {
    ...   // Constant declarations omitted
    private FramedOval head;              // head to be dragged
    private FramedOval mouth;             // mouth
    private FramedOval leftEye, rightEye; // eyes
    private Location lastPoint;    // point where mouse was last seen

    // whether the face has been grabbed by the mouse
    private boolean faceGrabbed = false;

    // make the face
    public void begin() {
        head = new FramedOval(FACE_LEFT, FACE_TOP,
                    FACE_WIDTH, FACE_HEIGHT, canvas);
        mouth = new FramedOval(MOUTH_LEFT, MOUTH_TOP,
                    MOUTH_WIDTH, MOUTH_HEIGHT, canvas);
        leftEye = new FramedOval(FACE_LEFT+EYE_OFFSET-EYE_RADIUS/2,
                    FACE_TOP+EYE_OFFSET, EYE_RADIUS, EYE_RADIUS, canvas);
        rightEye = new FramedOval(FACE_LEFT+FACE_WIDTH-EYE_OFFSET-EYE_RADIUS/2,
                    FACE_TOP+EYE_OFFSET, EYE_RADIUS, EYE_RADIUS, canvas);
    }

    // Save point where mouse pressed and whether point was in face
    public void onMousePress(Location point) {
        lastPoint = point;
        faceGrabbed = head.contains(point);
    }

    // if mouse is in face, then drag the face
    public void onMouseDrag(Location point) {
        if ( faceGrabbed ) {
            head.move( point.getX() - lastPoint.getX(),
                       point.getY() - lastPoint.getY() );
            leftEye.move( point.getX() - lastPoint.getX(),
                       point.getY() - lastPoint.getY() );
            rightEye.move( point.getX() - lastPoint.getX(),
                       point.getY() - lastPoint.getY() );
            mouth.move( point.getX() - lastPoint.getX(),
                       point.getY() - lastPoint.getY() );
            lastPoint = point;
        }
    }
}
```

Figure 4.2: Code for dragging a funny face.

```
public class RevFaceDrag extends WindowController {
    ...   // Constant declarations omitted

    private FunnyFace happy;         // FunnyFace to be dragged

    private Location lastPoint;   // point where mouse was last seen

    // whether happy has been grabbed by the mouse
    private boolean happyGrabbed = false;

    // make the FunnyFace
    public void begin() {
        happy = new FunnyFace(FACE_LEFT,FACE_TOP, canvas);
    }

    // Save starting point and whether point was in happy
    public void onMousePress( Location point ) {
        lastPoint = point;
        happyGrabbed = happy.contains( point );
    }

    // if mouse is in happy, then drag happy
    public void onMouseDrag( Location point ) {
        if ( happyGrabbed ) {
            happy.move( point.getX() - lastPoint.getX(),
                        point.getY() - lastPoint.getY() );
            lastPoint = point;
        }
    }
}
```

Figure 4.3: Revised code for dragging a FunnyFace.

The only differences from `WhatADrag` are that the variable `box` of class `FilledRect` has been replaced uniformly by the variable `happy` of class `FunnyFace`, variable `boxGrabbed` has been renamed as `happyGrabbed`, and the `begin` method now creates and names a `FunnyFace` rather than a `FilledRect`. Otherwise they are identical.

In the next section we will discuss the design and implementation of classes, using the `FunnyFace` class as a first example. Later in the chapter we will reuse the `FunnyFace` class in yet another program, illustrating the advantages of designing classes to represent objects that may be used in many different situations.

4.2 Writing classes: `FunnyFace`

A class is a template that can be used to create new objects, called *instances* of the class. In writing a class we must specify all of the features, both instance variables and methods, of an object that will be available when it is created.

So far we have written a number of classes that handle mouse events. These class definitions have included constant and instance variable declarations and method definitions. All classes contain features like this, as well as *constructors*, which determine what actions are to be performed in constructions.

We begin the study of classes by implementing the class `FunnyFace`, which was introduced in the last section. It should generate funny faces that can be dragged around on the canvas. In order for this to be possible, we would like each funny face to have methods like those of the geometric objects we have already been using, like `FramedRect`. The class `RevFaceDrag` has code that sends messages `move` and `contains` to a funny face, so we will certainly need to include those. Normally we would add more methods, but these will be sufficient to demonstrate the structure of a class for now.

4.2.1 Instance variables

First we consider what instance variables might be needed to write the code for these methods. Luckily the code in the original `FaceDrag` class in Figure 4.2 gives a clear indication of what is needed. For example, the `onMouseDrag` method of that class included the following code:

```
if ( faceGrabbed ) {
    head.move( point.getX() - lastPoint.getX(),
               point.getY() - lastPoint.getY() );
    leftEye.move( point.getX() - lastPoint.getX(),
               point.getY() - lastPoint.getY() );
    rightEye.move( point.getX() - lastPoint.getX(),
               point.getY() - lastPoint.getY() );
    mouth.move( point.getX() - lastPoint.getX(),
               point.getY() - lastPoint.getY() );
    lastPoint = point;
}
```

This suggests that having instance variables with names like `head`, `leftEye`, `rightEye`, and `mouth` will be necessary in order to represent a `FunnyFace` object. These will all be declared as instance variables of `FunnyFace`, just as they originally were in `FaceDrag`.

```
private FramedOval head;              // head to be dragged
private FramedOval mouth;             // mouth
private FramedOval leftEye, rightEye; // eyes
```

4.2.2 Methods and parameters

Now that we have determined the instance variables, let's write the `move` method for `FunnyFace`.

```
// Move funny face by (dx,dy)
public void move(double dx, double dy) {
    head.move(dx,dy);
    leftEye.move(dx,dy);
    rightEye.move(dx,dy);
    mouth.move(dx,dy);
}
```

This is pretty straightforward. To move a funny face by `dx` in the horizontal direction and `dy` vertically, we simply tell each of the four instance variables to move by that much.

The only things really new in this method definition are the formal parameter declarations. The methods we have written so far have had at most one formal parameter. For example, `onMouseDrag`, `onMouseClick`, etc. have had only a single parameter of type `Location`. The method `move` takes two parameters to determine the distance to move in the horizontal and vertical directions. When there is more than one formal parameter in a method we separate the declarations by commas.

We use this method in the same way we used the methods for geometric objects. In `RevFaceDrag`, the `onMouseDrag` method includes the code

```
happy.move( point.getX() - lastPoint.getX(),
            point.getY() - lastPoint.getY() );
```

where `happy` is a variable of class `FunnyFace`. This method invocation has two (somewhat complex) actual parameters, `point.getX() - lastPoint.getX()` and `point.getY() - lastPoint.getY()`, which are separated by a comma.

The idea is that when `happy.move(...)` is executed, the formal parameters `dx` and `dy` are associated with the values of the actual parameters. Thus when the body of `move` in `FunnyFace` is executed as a result of this call, the values of `dx` and `dy` represent the differences in the x and y coordinates of `point` and `lastPoint`.

The actual parameters are matched up with the formal parameters in the same order they are written. Thus the first actual parameter, `point.getX() - lastPoint.getX()`, is associated with the first formal parameter, `dx`. Similarly the second actual parameter is associated with `dy`.

This is the first time we have seen exactly how actual and formal parameters match up. When we wrote event-handling methods like `onMouseDrag`, we used the formal parameter inside the method body, but never saw how the method was called. That was taken care of by the system when the mouse was actually dragged by the user. On the other hand, we used actual parameters with methods like `move`, `moveTo`, and `contains` with the geometric objects, but we didn't see how they corresponded with formal parameters.

Now we have finally seen both sides of parameters and how they match up. Formal parameters are declared in the method heading and are used in the method bodies. Actual parameters are used in sending messages to objects (or equivalently, invoking methods of objects). When the

method begins execution, the values of actual parameters (which may be given as quite complex expressions) are associated with the formal parameters.

With this understanding of methods and parameters, we are now ready to write the `contains` method, an accessor method. Accessor methods are different from mutator methods because they return a value rather than performing an action. This will lead to two differences from mutator methods. We must write a statement in the body of the method indicating the value to be returned and we must indicate in the method heading the type of that value.

To determine if a location is inside of a funny face, we need only determine whether it is inside the head (we don't really care whether it is inside an eye, for example, as long as it is inside the head). We write it as follows:

```
// Determine whether pt is inside funny face
public boolean contains(Location pt) {
    return head.contains(pt);
}
```

Because `contains` is to return a boolean value, we must write the type `boolean` before the method name. In all mutator method definitions we have written, including, for example, `onMousePress` and `onMouseDrag`, that slot has been filled with the type `void`, which is used in Java to indicate that the method performs an action but does not return a value.

A type different from `void` written before the method name in a method declaration indicates that the method returns a value of that type. For example, if `FunnyFace` had a method returning the x coordinate of the left edge of the face, its declaration might look like

```
public double getLeftEdge()
```

From the declaration we can see that this is a method that has no formal parameters, but returns a value of type `double`. Similarly, a declaration

```
public FramedOval getFace()
```

would indicate that `getFace` has no formal parameters and returns a value of type `FramedOval`.

Let's go back to the body of `contains`. The body of a method that returns a value may include several lines of code, but the last statement executed must be of the form `return` ***expn***, where ***expn*** represents an expression whose type is the return type of the method. When the method is invoked, the method body is executed and the value of ***expn*** is returned as the value of the method invocation. The method `contains` happens to include only a single line, a `return` statement that returns the value of `head.contains(pt)`.

The `contains` method of `FunnyFace` is used in the `onMousePress` method of `RevFaceDrag` in an assignment statement:

```
happyGrabbed = happy.contains( point );
```

Let's trace the execution of this assignment statement to make sure that we understand how the method invocation works. When the expression `happy.contains( point )` is evaluated, the actual parameter `point` is associated with the formal parameter `pt` of the method `contains`. Then the expression `head.contains(pt)` from the `return` statement is evaluated. This will return true exactly when `pt` is inside of the `FramedOval` associated with `head`. That boolean value is returned as the value of `happy.contains(point)`. After it returns, that boolean value is assigned to `happyGrabbed`.

Exercise 4.2.1 *When the computer executes* `happy.move(point.getX(),0)`*, what values are associated with the formal parameters* `dx` *and* `dy` *of method* `move` *of class* `FunnyFace`*.*

4.2.3 Constructors

Now that we've discussed the instance variables and the two methods, `move` and `contains`, we're almost done with the definition of class `FunnyFace`. If we look back at our first attempt at this program, `FunnyFaceDrag`, and compare that with `RevFaceDrag` and the pieces of `FunnyFace` we've just discussed, we find that there is only one thing missing.

The `begin` method of `FunnyFaceDrag` initialized the instance variables `head`, `leftEye`, `rightEye`, and `mouth`. The `begin` method of `RevFaceDrag`, by contrast, included only the single line:

```
happy = new FunnyFace(FACE_LEFT,FACE_TOP, canvas);
```

Obviously, the construction `new FunnyFace(FACE_LEFT,FACE_TOP, canvas)` must somehow create a new value from class `FunnyFace`, in the process initializing the instance variables of the class. We have used constructions very much like this for geometric objects, but we have not yet seen what the code behind the construction looks like.

We provide instructions for the construction by writing a *constructor* definition for `FunnyFace`. Constructor definitions look very much like method definitions except that there is no return type specified, and the name of the constructor must be the same as the name of the class. The constructor for class `FunnyFace` will have three formal parameters: `left`, `top`, and `canvas`, representing the x coordinate of the left edge of the face, the y-coordinate of the top of the face, and the canvas the face will be drawn on. The body of the constructor for `FunnyFace`, shown below, uses `top` and `left` (along with several constants declared in the class) to calculate the locations of each of the components of the funny face, and constructs those components on the `canvas`.

```
// Create pieces of funny face
public FunnyFace( double left, double top, DrawingCanvas canvas) {
    head = new FramedOval( left, top, FACE_WIDTH, FACE_HEIGHT, canvas );
    mouth = new FramedOval( left+(FACE_WIDTH-MOUTH_WIDTH)/2,
                        top + 2*FACE_HEIGHT/3,
                        MOUTH_WIDTH, MOUTH_HEIGHT, canvas );
    leftEye = new FramedOval( left+EYE_OFFSET-EYE_RADIUS/2, top+EYE_OFFSET,
                        EYE_RADIUS, EYE_RADIUS, canvas );
    rightEye = new FramedOval( left+FACE_WIDTH-EYE_OFFSET-EYE_RADIUS/2,
                        top+EYE_OFFSET, EYE_RADIUS, EYE_RADIUS, canvas);
}
```

The parameter `canvas` must be passed to the `FunnyFace` constructor because only classes that extend `WindowController` have automatic access to the `canvas`. If other objects need to use the `canvas`, then it must be passed to them as a parameter.

The constructor body for `FunnyFace` is very similar to the code in the `begin` method of `FaceDrag` in Figure 4.2. In general, constructors for regular classes play a role similar to that of the `begin` method in classes that extend `WindowController`. In this case, the code of the constructor creates the four components of the funny face and stores them in the appropriate instance variables.

The expressions used as parameters in the constructions for the components (*e.g.*, `mouth`) in the `FunnyFace` constructor are more complex than they were in the `begin` method of `FaceDrag`, because the locations of the components must be computed relative to the parameters `left` and `top`.

When a programmer wishes to create a new `FunnyFace`, as in the `begin` method of class `RevFaceDrag` in Figure 4.3, she writes a construction statement like the following:

```
new FunnyFace( leftSide, topSide, canvas );
```

As with methods, the three actual parameters of the construction are associated with the formal parameters of the constructor in the order they are written. Thus the actual parameter `leftSide` is associated with the formal parameter `left`, `topSide` is associated with `top`, and the actual parameter `canvas`, from the `RevFaceDrag` class is associated with the formal parameter `canvas` of the `FunnyFace` constructor.

The last association might seem strange as `canvas` is being associated with `canvas`. The reason that it is necessary to explicitly make this association is that the `canvas` from `RevFunnyFace` has nothing to do with the formal parameter `canvas` of the constructor. The fact that they have the same name is essentially an "accident".[1] The only reason the two are associated when the constructor is called is that each is the third in the list of parameters for the constructor – one as an actual parameter, and the other as a formal parameter.

If the constructor is called several times with different values for `left` and `top`, it will result in several funny faces being constructed in different locations on the canvas. Thus if we write

```
happyLeft = new FunnyFace( 40, 100, canvas );
happyRight = new FunnyFace( 90, 100, canvas );
```

the program will place two funny faces, `happyLeft` and `happyRight`, next to each other on `canvas`.

It is important to remember that each instance of a class has its own distinct collection of instance variables. Thus `happyLeft` will have instance variables `face`, `eyeLeft`, `eyeRight`, and `mouth`. The object named by `happyRight` will have its own distinct copies of each of these variables. Thus the oval representing the face of `happyLeft` will be 50 pixels to the left of the oval representing the face of `happyRight`.

4.2.4 Putting it all together

The complete definition of class `FunnyFace` can be found in Figure 4.4. As usual, the first line of the class definition specifies the name of the class, in this case, `FunnyFace`. This class does *not* extend `WindowController` as it has nothing to do with reacting to mouse actions on the canvas. The definitions of the features of `FunnyFace` are enclosed in braces that begin after the class name. The features include constant definitions, instance variable declarations, the constructor definition, and the method definitions.

Class definitions in Java are written as follows:

```
public class Name {

    constant definitions

    variable declarations

    constructor

    methods
}
```

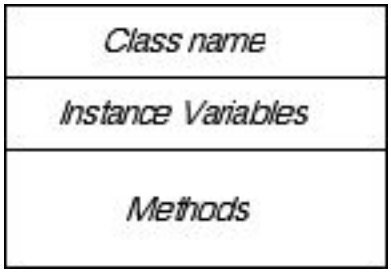
[1] We chose the same name for the two different variables on purpose as we wished to suggest that they should be associated – it is just that the computer doesn't know to make that association unless we tell it to by passing the actual parameter `canvas` to be associated with the formal parameter `canvas`.

```
public class FunnyFace {

    private static final int FACE_HEIGHT = 60,  // dimensions of the face
                             FACE_WIDTH = 60,

                             EYE_OFFSET = 20, // eye location and size
                             EYE_RADIUS = 8,

                             MOUTH_WIDTH = FACE_WIDTH/2, // dimensions of the
mouth
                             MOUTH_HEIGHT = 10;

    private FramedOval head,     // oval for head
                       leftEye,  // ovals for eyes
                       rightEye,
                       mouth;    // oval for mouth

    // Create pieces of funny face
    public FunnyFace( double left, double top, DrawingCanvas canvas) {
        head = new FramedOval( left, top, FACE_WIDTH, FACE_HEIGHT, canvas );
        mouth = new FramedOval( left+(FACE_WIDTH-MOUTH_WIDTH)/2,
                            top + 2*FACE_HEIGHT/3,
                            MOUTH_WIDTH, MOUTH_HEIGHT, canvas );
        leftEye = new FramedOval( left+EYE_OFFSET-EYE_RADIUS/2, top+EYE_OFFSET,
                            EYE_RADIUS, EYE_RADIUS, canvas );
        rightEye = new FramedOval( left+FACE_WIDTH-EYE_OFFSET-EYE_RADIUS/2,
                            top+EYE_OFFSET, EYE_RADIUS, EYE_RADIUS, canvas);
    }

    // Move funny face by (dx,dy)
    public void move(double dx, double dy) {
        head.move(dx,dy);
        leftEye.move(dx,dy);
        rightEye.move(dx,dy);
        mouth.move(dx,dy);
    }

    // Determine whether pt is inside funny face
    public boolean contains(Location pt) {
        return head.contains(pt);
    }
}
```

Figure 4.4: FunnyFace class

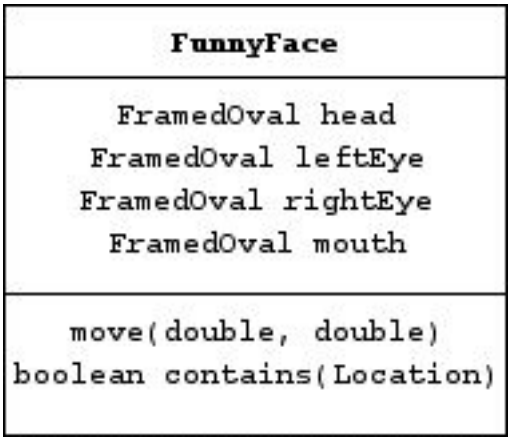
While Java does not insist that the order be exactly that specified above, we will find that ordering convenient for all of the examples in this text.

It is useful to have a graphical way of presenting important information about a class. We will use a picture like the following to represent classes:



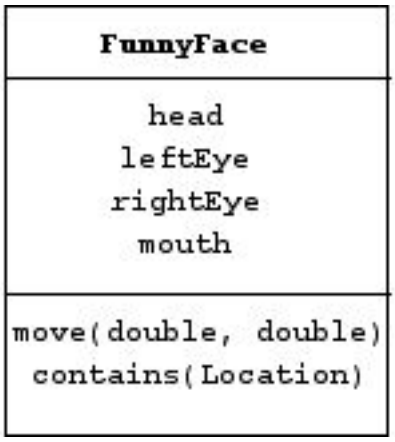
Notice that all of the important features of the class except for the constructors are represented in this diagram.

For example, we can represent the `FunnyFace` class as:



We include the names and types for instance variables. We also include the names of the methods and the types of their parameters. We write the return types for accessor methods only. If the return type is void, then it is omitted in the diagram. We do not bother to record the names of the formal parameters because the names of formal parameters are not important in invoking a method. The caller need only make sure that there are the appropriate number of actual parameters and that they have the necessary types.

Occasionally we will omit some of the type information when writing a class diagram. For example, we provide another diagram for `FunnyFace` below where we omit the type information for instance variables, and the return types for methods.



We will use this more compact diagram only where the omitted type information is unimportant or where it is obvious from context.

Now that we have completed the `FunnyFace` class, let's compare that class and `RevFaceDrag` with the original `FunnyFaceDrag`. The class `FunnyFaceDrag` had 5 instance variables – the four components of the face and the variable `lastPoint` to keep track of the last location of the mouse. The four components of the face were moved to the `FunnyFace` class, while `lastPoint` was kept with the mouse-handling code of `RevFaceDrag` because it was more relevant to the mouse movements than to the funny face.

All of this effort to create a `FunnyFace` class may seem like a lot of work for little benefit. After all, the code of class `FaceDrag` is shorter than the combination of the code in classes `FunnyFace` and `RevFaceDrag`. However, there are several advantages to creating and using the class `FunnyFace`.

One benefit is that having created the `FunnyFace` class, we can now use this class in any program now or in the future, rather than having to rewrite the functionality. In particular, this class may be used with a variety of other classes to create more complex programs. (We illustrate this later in the chapter by using `FunnyFace` in a different program.)

Another benefit is that because this program supports the same kind of methods as the geometric objects in the ObjectDraw library, we may use it just like these other objects. In the example above, the difference between a program allowing the user to drag around a rectangle and one allowing the user to drag around a funny face is minimal. We can thus reuse with minimal changes other code that we have already determined works with other kinds of objects. In a later chapter we'll introduce a feature of Java (the `interface`) that will make this reuse of code even easier.

4.3 Adding methods to `FunnyFace`

Just as we were able to define methods for `FunnyFace` that allowed it to be used very much like geometric objects from the library, there will likely be other programs written for geometric objects in which we would like to substitute a `FunnyFace` for a filled rectangle. (OK, funny faces are not in such great demand, but real examples, like triangles, would likely be useful in these ways!) Thus, we would normally want to add other methods like those supported by the other geometric objects to the `FunnyFace` class, even though we did not need them for dragging funny faces. Such methods would likely include `hide`, `show`, `setColor`, and `getColor`, among others.

The code for `hide` is similar to that of `move` in that it simply forwards `hide` messages to each of its components.

```
// Hide the funny face
```

```
public void hide() {
    face.hide();
    leftEye.hide();
    rightEye.hide();
    mouth.hide();
}
```

The code for `show`, `setColor` and `getColor` should be easy for you to write, as they are similar to `move`, `hide`, and `contains`.

Exercise 4.3.1

1. *Write the methods* `show` *and* `setColor` *for class* `FunnyFace`*.*
2. *Write the code for* `getColor`*. It is an accessor method and should be similar to* `contains`*. Hint: Is there any reason to get the colors of* all *of the components?*

Because `FunnyFace` has a `move` method, an obvious addition would be a `moveTo` method that would have a `Location` as its formal parameter. Writing the method `moveTo` will be a bit trickier than `move` because we definitely do *not* want to move all of the components of the funny face to the same place. If we did that, both of the eyes and the mouth would end up on top of each other in the upper left hand corner of the face. We could instead calculate where we want all of the pieces to be after the `moveTo` and send a `moveTo` method with different parameters to each of the components of the funny face. However, it would be pretty painful to have to calculate all of those locations. (Look at how horrible all the calculations were in the constructor!)

There is an alternative that is both clever and simple – an ideal combination. What we will do is figure out *how far* to `move` the funny face rather than *where* to move it. Once we calculate the horizontal and vertical distance the funny face is to move, we can pass those as actual parameters to the funny face's `move` method and take advantage of the fact that we have already written that method:

```
public void moveTo(Location pt) {
    this.move(pt.getX() - head.getX(), pt.getY() - head.getY());
}
```

In the code for `moveTo` we calculate the difference between the current location of the head and the location to which it should move. We calculate the distance in the x and y directions separately. These x and y differences are then used as the actual parameters for an invocation of the method `move`.

One difference between the message in the first line of the `moveTo` method and others we have seen so far is that we want to execute the move method of the funny face itself, rather than telling the objects associated with the face's instance variables to execute their methods. We indicate that this is our intention by sending the message to `this`. Java provides the keyword `this` for programmers to use in method bodies as a name for the object that is currently executing the method. Thus when we write `this.move(...)` we are telling Java that we want to execute the `move` method for `this` object, the one currently executing the `moveTo` method.

Let's think about this a bit more concretely. The `RevFaceDrag` class has a variable of type `FunnyFace` named `happy`. Suppose we add an `onMouseClick` method to `RevFaceDrag` that causes `happy` to be moved to wherever the user clicked:

```
// Move happy to where clicked
public void onMouseClick(Location point) {
    happy.moveTo( point );
}
```

What happens when the statement `happy.moveTo( point )` is executed? The code for method `moveTo` of the `FunnyFace` class is just the single instruction

```
this.move( pt.getX() - head.getX(), pt.getY() - head.getY() );
```

When that instruction is executed, `this` will stand for `happy`, as it is the object executing the `moveTo` method. Thus the `move` message is sent to `happy`, with the same effect as if we had written `happy.move(...)`. The formal parameters `dx` and `dy` of the method `move` will be set to the values of `pt.getX() - head.getX()` and `pt.getY() - head.getY()`, respectively.

Because it is fairly common to have one method of an object invoke another method of the same object, Java allows the programmer to abbreviate the message send by omitting the "`this.`" prefix. That is, we could instead write the body of `moveTo` as

```
public void moveTo(Location pt) {
    move(pt.getX() - head.getX(), pt.getY() - head.getY());
}
```

At least initially, we will include the target `this` in the message send in order to remind the reader that messages are sent to objects – they do not stand alone.

We will see later that we can also pass `this` as a parameter of a message so that the method can send messages back to the sender. This is not unlike sending someone your e-mail address so that they can correspond with you.

From now on we will assume that the definition of `FunnyFace` includes at least the method `moveTo` defined above. As suggested earlier that will allow us to use `FunnyFace` more effectively in other programs.

Exercise 4.3.2 *Normally we would make the* ***`FunnyFace`*** *class even more flexible by also including parameters in the constructor to specify the width and height of the funny face being created. We left those off in this version of* ***`FunnyFace`*** *in order to keep the code simpler. Rewrite the constructor for* ***`FunnyFace`*** *to take two extra parameters* ***`width`*** *and* ***`height`*** *of type* ***`double`*** *to represent the width and height of the funny face being constructed.*

4.4 Another example: Implementing a Timer class

In this section we practice what we have learned in this chapter by designing another program that uses classes. There are two main points that we would like to make with this example. The first is that we can define classes like the provided `Location` that are useful in programs, but do not result in anything being displayed on the canvas. The second main point is to show how we can reuse previously defined classes to solve new problems.

The application we are interested in is a very simple game in which the user tries to click on an object soon after it moves to a new position. Because success in the game depends on clicking soon after the object has moved, we will create a `Timer` class that provides facilities to calculate elapsed time. We will reuse the `FunnyFace` class to create the object to be chased. Unlike the `FunnyFace` class, the `Timer` does not result in anything being drawn on the screen.

In order to calculate elapsed time, the `Timer` class uses the method `System.currentTimeMillis()`. This is a method from Java's `System` class that returns the number of milliseconds (thousandths of a second) since January 1, 1970. Because we are interested in the length of time intervals, we will obtain starting and ending times and then subtract them.

The `Timer` class in Figure 4.5 is very simple. The class has a single instance variable `startTime` that is initialized in the constructor with the value of `System.currentTimeMillis()`. It can be reinitialized in the `reset` method by reevaluating `System.currentTimeMillis()`. The methods `elapsedMilliseconds` and `elapsedSeconds` calculate the elapsed time between `startTime` and the time when the methods are called. As the time values used are all in units of milliseconds, the number of seconds of elapsed time must be calculated by computing the number of milliseconds and then dividing by 1000. The type of the value returned by both of these methods is `double`; neither takes any parameters.

The class `Chase` is an extension of `WindowController`. Using class `FunnyFace`, it creates a funny face in the upper part of the canvas, and constructs a `Text` message, `infoText`, that tells the user to click to start playing the game. When the user clicks for the first time, the funny face moves to a new position on the canvas. If the user clicks on it within `TIME_LIMIT` seconds, the user wins. If the user misses the funny face with the click then a message appears indicating that the user missed, and the funny face moves to a new randomly selected location. If the user clicks on the funny face, but too slowly, then the message "Too slow" appears and the funny face is moved. Finally, if the user does click on the funny face in time, then a message is displayed and the user is informed to click to start the game again.

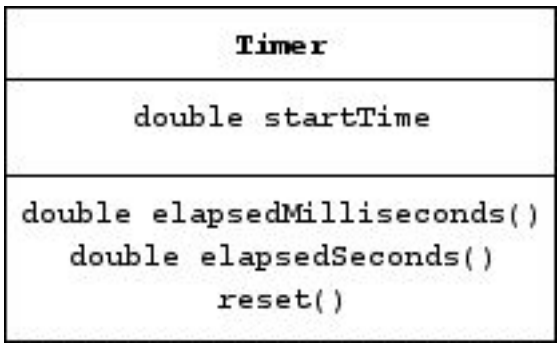
The code for class `Chase` is given in Figures 4.6 and 4.7. Instance variables are declared for the funny face, timer, and text display. There are also two instance variables representing random number generators to obtain new x and y coordinates of the funny face (which is moved randomly on the canvas). Finally there is a boolean instance variable, `playing`, that keeps track of whether or not the game has actually started.

The `begin` method initializes `stopWatch` with the current time, and creates random number generators to be used to select new x and y coordinates when the funny face is moved. The new `FunnyFace`, which is constructed a little above the center of the screen, is associated with instance variable `funFace`. The `Text` object `infoText` is created just below the face with a startup message telling the user how to start the game. Finally `playing` is initialized to be false. This instance variable keeps track of whether the game is in progress and the system is waiting for the user to click on the funny face, or if the system is simply waiting for the user to click to start the game.

The `onMouseClick` method of `Chaser` determines how the program reacts to a click. The conditional statement provides cases for when the game has not yet started (`playing` is false), when the click does not occur in `funFace`, when the user succeeds (the click does occur in `funFace` and within the appropriate time interval), and when the elapsed time is too long. The code in each of those cases is straightforward. Notice that we use the method `moveTo` of `FunnyFace`. Because that method takes a `Location` as a parameter, the actual parameter involves the construction of a new `Location` whose components are randomly generated with the random number generators.

Exercise 4.4.1 *Why is `playing` given values in the first and third branches of the conditional (**if**) statement, but not in the second and fourth?*

We will make several improvements to this program in the following sections as we introduce new features of Java that are useful in writing classes.




```
//  Class allowing calculations of timing between events.

public class Timer {

    private double startTime;   // time when Timer started or reset

    // Create timer, initializing startTime with current time
    public Timer() {
        startTime = System.currentTimeMillis();
    }

    // Return number of milliseconds since last reset
    public double elapsedMilliseconds() {
        return System.currentTimeMillis() - startTime;
    }

    // Return number of seconds since last reset
    public double elapsedSeconds() {
        return this.elapsedMilliseconds() / 1000;
    }

    // Reset startTime
    public void reset() {
        startTime = System.currentTimeMillis();
    }
}
```

Figure 4.5: `Timer` class and diagram

```
public class Chase extends WindowController
{
    private static final int WINDOW_HT = 400,   // Window dimensions
                             WINDOW_WIDTH = 400;

    private static final double TIME_LIMIT = 1.5;  // time available to click

    private FunnyFace funFace;          // funny face to be chased

    private Timer stopWatch;            // Timer to see if click fast enough

    private Text infoText;              // Text item to be displayed during game

    private RandomIntGenerator randXGen,    // generators to get new x and y
coords
                               randYGen;

    private boolean playing;            // Is player playing or waiting to start

    // Set up timer and funny face for game
    public void begin() {
        stopWatch = new Timer();
        randXGen = new RandomIntGenerator(0, WINDOW_WIDTH);
        randYGen = new RandomIntGenerator(0, WINDOW_HT);
        funFace = new FunnyFace(WINDOW_WIDTH/2, WINDOW_HT/3, canvas);
        infoText = new Text("Click to start chasing the FunnyFace.",
                            WINDOW_WIDTH/3, WINDOW_HT/2, canvas);
        playing = false;
    }
```

Figure 4.6: `Chase` class: instance variables and `begin` method

```
    // Determine if user won and move funny face if necessary
    public void onMouseClick(Location pt) {
        if (!playing) {                          // Start playing
            playing = true;
            infoText.setText("Click quickly on the FunnyFace to win!");
            funFace.moveTo(new Location(randXGen.nextValue(),
randYGen.nextValue()));
            stopWatch.reset();
        }
        else if (!funFace.contains(pt)) {        // missed the funny face
            infoText.setText("You missed!");
            funFace.moveTo(new Location(randXGen.nextValue(),
randYGen.nextValue()));
            stopWatch.reset();
        }
        else if (stopWatch.elapsedSeconds() <= TIME_LIMIT) {  // got it in time
            playing = false;
            infoText.setText("You got the FunnyFace in time.  Click to start
over.");
        }
        else {                                   // User was too slow
            infoText.setText("Too slow!");
            funFace.moveTo(new Location(randXGen.nextValue(),
randYGen.nextValue()));
            stopWatch.reset();
        }
    }
}
```

Figure 4.7: `Chase` class: `onMouseClick` method.

4.5 Access control: public versus private

Earlier we wrote that all instance variables must be declared as `private` and methods should be declared as `public`. In this section we explain what those words actually mean. The short explanation is that any feature declared as public is accessible anywhere in a program, while a feature declared as private is accessible only inside of the class in which it is declared.

If a method or instance variable of an object is declared to be public, then it will be available to other objects to be used. Because `FunnyFace` declared the method `move` to be `public`, the `onMouseDrag` method of `RevFaceDrag` could invoke `move` on `happy`. Similarly, `happy.contains( point )` may be written in the method `onMousePress` because `contains` is a public method of `FunnyFace`. If `contains` had been declared to be private, then that expression would not have been legal in Java.

Similarly, if `ear` were to be defined as a new `public` instance variable of `FunnyFace`, then a programmer could write `happy.ear` in a method of `RevFaceDrag` in order to get access to the current value of `happy`'s instance variable, `ear`. However, because the instance variable `head` of `FunnyFace` was declared to be `private`, it may not be used outside of the class definition of `FunnyFace`. In particular, writing `happy.head` anywhere inside of `RevFaceDrag` would result in an error message. In summary, features declared to be `private` are accessible in the constructor and methods of the class in which it is defined, but not elsewhere.

Let us now reexamine the `onMouseClick` method of class `Chase` to see how a `private` method may be of help. You may have noticed that there are two lines in the `onMouseClick` method that are repeated in three different cases of the conditional statement. They are

```
funFace.moveTo(new Location(randXGen.nextValue(), randYGen.nextValue()));
stopWatch.reset();
```

If they had been repeated in all four cases in the conditional, we could have just moved them to occur immediately after the conditional. However, this code need not be executed in the case that the user clicked on the funny face in time, so moving it to the end is not a good strategy.

When we have repeated code like this it is worth trying to figure out what these statements represent, and building a method that represents that abstraction. This particular repeated set of statements is resetting the game so that it can be played again. Because it is conceptually a single task, we can name this block of code and make it into a method, `resetGame`. Java code including this method and the revised code for `onMouseClick` that uses the new method is given in Figure 4.8. We have also moved the declaration of the local variable `newLocation` to the new method since it is no longer needed in `onMouseClick`. Notice how using this new method makes the `onMouseClick` method somewhat easier to understand, because the reader need not be concerned about the details of reseting the game.

We have declared `resetGame` to be a `private` method because we do not need or want any other object to invoke it. It is only to be used inside the class. Private methods, like private instance variables, are only available inside the methods of the class in which they are declared.

In this text we will always declare instance variables to be `private`.[2] It is generally considered bad programming practice to allow other objects access to an object's instance variables by declaring them to be `public`. While it is easier to explain these reasons after you have gained more programming experience, the general principle is that an object should be responsible for maintaining its own state. It can provide access to portions of the state by providing accessor methods to

[2] In more advanced programming it is sometimes useful to declare certain instance variables to be `protected` rather than `private`. These situations will not appear in this text, so we do not discuss the meaning of `protected`.

```
    // Move funFace to new randomly selected location and reset timer
    private void resetGame() {
        funFace.moveTo(new Location(randXGen.nextValue(), randYGen.nextValue()));
        stopWatch.reset();
    }

    // Determine if user won and move funny face if necessary
    public void onMouseClick(Location pt) {
        if (!playing) {                          // start playing
            playing = true;
            infoText.setText("Click quickly on the FunnyFace to win!");
            resetGame();
        }
        else if (!funFace.contains(pt)) {        // missed the funny face
            infoText.setText("You missed!");
            resetGame();
        }
        else if (stopWatch.elapsedSeconds() <= TIME_LIMIT) {  // got it!
            playing = false;
            infoText.setText("You got the FunnyFace in time.  Click to start
over.");
        }
        else {                                   // too slow
            infoText.setText("Too slow!");
            resetGame();
        }
    }
}
```

Figure 4.8: `Chase` class: revised `onMouseClick` method using private method `resetGame`.

```
// Create pieces of funny face
public FunnyFace( double left, double top, DrawingCanvas canvas) {
    head = new FramedOval( left, top, FACE_WIDTH, FACE_HEIGHT, canvas );
    mouth = new FramedOval( left+(FACE_WIDTH-MOUTH_WIDTH)/2,
                        top + 2*FACE_HEIGHT/3,
                        MOUTH_WIDTH, MOUTH_HEIGHT, canvas );
    mouthCover = new FilledOval(left+( FACE_WIDTH-MOUTH_WIDTH)/2,
                        top + 2*FACE_HEIGHT/3-EYE_RADIUS/2,
                        MOUTH_WIDTH, MOUTH_HEIGHT, canvas );
    mouthCover.setColor(Color.white);
    leftEye = new FramedOval( left+EYE_OFFSET-EYE_RADIUS/2, top+EYE_OFFSET,
                        EYE_RADIUS, EYE_RADIUS, canvas );
    rightEye = new FramedOval( left+FACE_WIDTH-EYE_OFFSET-EYE_RADIUS/2,
                        top+EYE_OFFSET, EYE_RADIUS, EYE_RADIUS, canvas);
}
```

Figure 4.9: `FunnyFace` constructor with a smile.

return those values and can allow changes by providing mutator methods to update those values. However, by not allowing unfettered access to instance variables, we can prevent other objects from making the instance variables hold inconsistent data. We can also later change our mind about how we will represent the state of an object by changing the collection of instance variables, yet not have to worry about the impact of this on code using objects from the class.

Be careful to always include the access qualifier of `private` with all instance variable declarations, and either `public` or `private` with all method declarations. If you leave these access qualifiers off, Java will not complain, but will instead assign them "default" access, which is typically not what is desired.

4.6 Local variables

So far we have seen how to give names to instance variables and formal parameters. There is another kind of variable that can be named and used in Java classes. These are called local variables. In this section we introduce local variables and compare and contrast their use with that of instance variables and parameters.

In order to better motivate the use of local variables, we make the funny face a bit more elaborate. We modify the picture so that rather than having an oval for the mouth, we have a smile. We can obtain this by painting a white oval over the top part of the mouth, leaving only the bottom part of the mouth showing. We introduce a new instance variable, `mouthCover`, with type `FilledOval`, for this purpose. The revised constructor code is given in Figure 4.9.

The code for this constructor involves a number of complex expressions that calculate the x and y coordinates of the upper-left corners of the components of the funny face. Moreover, several of these expressions are calculated more than once. Thus the code involves both difficult to understand expressions and redundant calculations. It would be easier to understand this code if we could precompute some of these values and give them understandable names to be used in creating the components of the face.

Figure 4.10 contains the code for a revised version of the constructor. The effect of the construc-

```
// Create pieces of funny face
public FunnyFace( double left, double top, DrawingCanvas canvas) {
    double mouthLeft, mouthTop, eyeTop; // coordinates of mouth and eyes

    mouthLeft = left+(FACE_WIDTH-MOUTH_WIDTH)/2;
    mouthTop = top + 2*FACE_HEIGHT/3;
    eyeTop = top+EYE_OFFSET;

    head = new FramedOval( left, top, FACE_WIDTH, FACE_HEIGHT, canvas );
    mouth = new FramedOval( mouthLeft, mouthTop,
                            MOUTH_WIDTH, MOUTH_HEIGHT, canvas );
    mouthCover = new FilledOval(mouthLeft, mouthTop-EYE_RADIUS/2,
                            MOUTH_WIDTH, MOUTH_HEIGHT, canvas );
    mouthCover.setColor(Color.white);
    leftEye = new FramedOval( left+EYE_OFFSET-EYE_RADIUS/2, eyeTop,
                            EYE_RADIUS, EYE_RADIUS, canvas );
    rightEye = new FramedOval( left+FACE_WIDTH-EYE_OFFSET-EYE_RADIUS/2,
                            eyeTop, EYE_RADIUS, EYE_RADIUS, canvas);
}
```

Figure 4.10: A revised version of the smiling `FunnyFace` constructor using local variables.

tor is exactly the same as the original, but we have pulled out three expressions from the original and given them names: `mouthLeft`, `mouthTop`, and `eyeTop`. As one would hope, the names suggest what they stand for, the coordinates of the left and top of the mouth, and the top of the eyes.

Because we now have names for these expressions, the names can be used in the constructions for `mouth`, `mouthCover`, `leftEye`, and `rightEye`. Moreover the calculations for these values are only made once, with the values available for reuse in multiple places in the code.

New identifiers that are declared in constructor or method bodies are known as *local variables*. Local variables may only be used inside the methods or constructors in which they are defined, and their declaration must occur before their use. Thus `mouthLeft` is used inside the body of the constructor `FunnyFace`, but is not accessible inside any of the methods of class `FunnyFace`. This is quite different from the instance variables of the class, which are accessible inside any of the constructors or methods of the class.

Because local variables are accessible only inside the method or constructor in which they are defined, Java does not require or even allow them to be given an access qualifier of `public` or `private`. Instead, local variables are specified by just giving their types and names. Later we will see examples involving the use of local variables inside methods, but their usage is basically the same as for constructors.

Local variables, like instance variables, may be initialized in their declarations. Thus we may replace the first four lines of the body of the constructor `FunnyFace` in Figure 4.10 by

```
double mouthLeft = left + (FACE_WIDTH-MOUTH_WIDTH)/2;
double mouthTop = top + 2*FACE_HEIGHT/3;
double eyeTop = top + EYE_OFFSET;
```

We can modify the `onMouseClick` method of class `Chase` one more time to show the use of a local variable there. Suppose we wish to display for the user the elapsed time between the time

```
    // Determine if user won and move funny face if necessary
    public void onMouseClick(Location pt) {
        double elapsedTime = stopWatch.elapsedSeconds();
        if (!playing) {                          // start playing
            playing = true;
            infoText.setText("Click quickly on the FunnyFace to win!");
            resetGame();
        }
        else if (!funFace.contains(pt)) {      // missed the funny face
            infoText.setText("You missed!");
            resetGame();
        }
        else if (elapsedTime <= TIME_LIMIT) {  // got it!
            playing = false;
            infoText.setText("You got the FunnyFace in "+elapsedTime+
                             " seconds. Click to start over.");
        }
        else {                                   // too slow
            infoText.setText("Too slow!");
            resetGame();
        }
    }
}
```

Figure 4.11: Chase class: revised `onMouseClick` method using local variable.

the funny face moved and when the user clicked on it. In that case it might be helpful to calculate the elapsed time at the beginning of the method, using that saved time wherever the elapsed time is needed. In the code in Figure 4.11 it is calculated and saved in local variable `elapsedTime` and used twice in the third case of the conditional. It is used to test if the user was fast enough, and also in printing a message in `infoText`.

4.7 Using instance variables, parameters, and local variables

We have now introduced three different kinds of identifiers (names) that represent values. They are instance variables, local variables, and formal parameters. We take the rest of this section to compare and contrast their use.

Instance variables are visible inside all constructors and methods of the class in which they are declared. They should be declared as `private`. Local variables and parameters are defined and visible only in the constructors and methods in which they are declared. They must *not* be declared as `private` or `public`.

If instance variables are not initialized in their declarations, they are automatically provided with *default* values – `null` in the case of objects (see the discussion of `null` below), and 0 in the case of numeric instance variables. Instance variables are typically initialized in the constructor of the class, as was the case in class `FunnyFace`. Their initial values may depend on the values of the formal parameters of the constructor. Instance variables retain their values between method invocations

and can be updated inside of methods of the class. Thus they are used to hold information that must be retained to be used in later invocations of methods.

The special value `null` is a member of every class type. If you send a message to the value `null` at run-time it will generate an error message in your program. It is most often used as an initial value of instance variables, as indicated above. The programmer can test to see if the value of an instance variable is different from `null` in order to determine if the variable has been assigned a meaningful value.

Formal parameters become associated with the actual parameters during a method invocation. While a formal parameter may be assigned a new value inside the method or constructor in which it appears, the assignment has no impact on the actual parameter, so assigning to formal parameters is unusual. A formal parameter always loses its value when the execution of the method or constructor in which it is contained finishes, so parameters are typically used to transmit information needed into a method body or constructor body. If that information is needed after the method terminates, then it should be associated with an instance variable inside the method body.

Sending a message to a formal parameter may result in changes to the state of the corresponding actual parameter. Look at the following example:

```
public void makeYellow(FilledRect aRect) {
    aRect.setColor(Color.yellow);
}
```

The result of executing the method invocation `someObj.makeYellow(myRect)` will be to change the color of `myRect` to yellow. Notice that we have not replaced `myRect` with a different object; we have instructed the filled rectangle associated with `myRect` to change its state by sending it the `setColor` message.

By contrast suppose we (foolishly) write the following method:

```
public void makeYellowRect(FilledRect aRect) {
    aRect = new FilledRect(50, 30, 20, 40, canvas);
    aRect.setColor(Color.yellow);
}
```

You might expect the result of executing the method invocation `someObj.makeYellowRect(myRect)` would result in `myRect` holding a new 20 by 40 pixel yellow rectangle, but you would be wrong. The assignment in the method body causes the name `aRect` to be associated with a new filled rectangle, but this has no impact on the actual parameter `myRect`. Thus `myRect` still refers to exactly the same rectangle as before the method invocation (and it stays the same color as before the invocation). Exactly the same effect would be obtained if `makeYellowRect` took no parameters and used `aRect` as a local variable. Either way the new yellow rectangle appears on the canvas, but the version above may confuse the reader into thinking that this will have an impact on the value of the actual parameter to the method.

Local variables may be initialized in their declarations or by assignment statements inside the method or constructor in which they are declared. They are *never* provided with default values. The values of local variables are lost when the method or constructor in which they are declared finishes executing. Hence local variables must be reinitialized each time the method or constructor containing it is executed. Local variables are thus used to store temporary values used in a local computation.

Here are general guidelines to determine when each of these three kinds of identifiers should be used.

1. If a value is to be retained after a constructor has been called or is needed by the object after the execution of one of its methods, then it should be stored in an instance variable.

2. If a value must be obtained from the context where a constructor or method is invoked, then it should be passed as a parameter.

3. If a value is only needed temporarily during the execution of a constructor or method, can be initialized locally, and need not be retained for later method calls, then it should be declared as a local variable.

Novices often use instance variables where it makes more sense to be using local variables. Why does it matter? Local variables are to be preferred to instance variables because they make it easier to read and understand your classes. When a programmer is looking at a class definition, the instance variables provide information on what state information is retained between invocations of methods. If superfluous variables are declared as instance variables it makes it harder for the reader to understand the state of objects from the class.

Programmers also find it easiest to understand variables if they are declared close to where they are used. This makes it easier for a reader to find the definitions and hence read the comments explaining what the identifier is standing for. Because instance variables are clustered at the head of a class, local variables are likely to be much closer to their uses.

Objects with excess instance variables also take up more space in memory than they need to, a problem that may be important for large programs, but for now we are most concerned about the readability issues.

For each instance variable declaration in a class, consider why it is there and whether it really needs to be retained for use by later methods. Parameters provide the primary mechanism for passing data between different objects. If a method or constructor needs information that is not held in its own instance variables, it typically needs to obtain that data via a parameter. If the value is obtained from elsewhere, but needs to be retained for use in later method executions, then it can be passed in as a parameter and saved to an instance variable. However, there is no reason to save parameter values as instance variables if they are not needed later. Just use the parameters directly in the body of the method into which they are passed.

Exercise 4.7.1 *1. Why is `playing` an instance variable of class `Chaser` rather than a local variable?*

2. Why is `elapsedTime` a local variable of method `onMouseClick` rather than an instance variable of `Chaser`?

4.8 Overloaded methods and constructors

In our examples using geometric objects, we have used two different versions of the `moveTo` method and of the constructions for geometric objects. These different versions allowed the programmer to either use two parameters representing the x and y coordinates of where the object was to appear or to use a single `Location` parameter.

Java allows a class to have two methods with the same name, as long as they can be distinguished by the types of their parameters.[3] Methods such as this are said to be *overloaded*.

[3]Of course we have already seen many examples where different classes have methods with the same name. That is never a problem because when we send a message to an object, the system obtains the corresponding method from the class of the receiver.

```
public void moveTo(double x, double y) {
    this.move(x - head.getX(), y - head.getY());
}

public void moveTo(Location pt) {
    this.moveTo(pt.getX(), pt.getY());
}
```

Figure 4.12: Writing overloaded methods in terms of one version

Let's see how we can define overloaded methods by going back to class `FunnyFace`. That class included a method `moveTo` with a parameter of type `Location`. We repeat the method below:

```
public void moveTo(Location pt) {
    this.move(pt.getX() - head.getX(), pt.getY() - head.getY());
}
```

We have seen that the predefined geometric classes have two versions of `moveTo` – one taking the single parameter of type `Location` and the other taking two parameters of type double. Let's add the missing version to `FunnyFace`:

```
public void moveTo(double x, double y) {
    this.move(x - head.getX(), y - head.getY());
}
```

The code for the new version of `moveTo` is very similar to that of the earlier version, replacing `pt.getX()` by parameter `x` and `pt.getY()` by parameter `y`. This is no coincidence, as overloaded methods should all have the same effect. This follows from the general principle that code that looks the same should do the same thing. If overloaded methods do very different things then programmers will be more likely to be confused.

One way of ensuring that these methods do the same things is to have one of them written in terms of the other. Because the second version is somewhat simpler than the first, we will use that as the base version, writing the version with the `Location` parameter in terms of the version with two `doubles`. See the code in Figure 4.12.

The second version of `moveTo` is obtained by invoking the first version of `moveTo` on itself (`this.moveTo`) with parameters `pt.getX()` and `pt.getY()`. This indirect execution of the version of `moveTo` may be a bit slower than the direct call of `this.move` or indeed of directly copying the body of `move` into the `moveTo` method. However it has the great advantage of maintaining consistency if either `move` or one of the `moveTo`'s is modified.

An example of the use of the overloaded `moveTo` would be to simplify the body of private method `resetGame`. Rather than writing the original code:

```
// Move funFace to new randomly selected location and reset timer
private void resetGame() {
    funFace.moveTo(new Location(randXGen.nextValue(), randYGen.nextValue()));
    stopWatch.reset();
}
```

we can write

```
// Move funFace to new randomly selected location and reset timer
private void resetGame() {
    funFace.moveTo(randXGen.nextValue(), randYGen.nextValue());
    stopWatch.reset();
}
```

which eliminates the need for the `Location` construction in the actual parameter of `moveTo`.

We can also overload constructors in Java. We illustrate this with `FunnyFace`. The original constructor for `FunnyFace` (repeated below) takes two `double`'s and a `DrawingCanvas` as parameters.

```
// Create pieces of funny face
public FunnyFace( double left, double top, DrawingCanvas canvas) {
    head = new FramedOval( left, top, FACE_WIDTH, FACE_HEIGHT, canvas );
    mouth = new FramedOval( left+(FACE_WIDTH-MOUTH_WIDTH)/2,
                        top + 2*FACE_HEIGHT/3,
                        MOUTH_WIDTH, MOUTH_HEIGHT, canvas );
    leftEye = new FramedOval( left+EYE_OFFSET-EYE_RADIUS/2, top+EYE_OFFSET,
                        EYE_RADIUS, EYE_RADIUS, canvas );
    rightEye = new FramedOval( left+FACE_WIDTH-EYE_OFFSET-EYE_RADIUS/2,
                        top+EYE_OFFSET, EYE_RADIUS, EYE_RADIUS, canvas);
}
```

It might also be helpful to include a constructor that takes a `Location` rather than the two parameters of type `double`. Recall that the constructor for a class must have the same name as the class, so we have no freedom in naming the constructor – it must be overloaded. If the new version has the header,

```
public FunnyFace( Location upperLeft, DrawingCanvas canvas) {
```

we could simply copy the code from the first version, replacing all occurrences of `left` with `upperLeft.getX()` and all occurrences of `top` with `upperLeft.getY()`. However, as with overloaded methods, we can also write the constructor so that it executes the code of the original constructor, but with the parameters replaced as above and no copying of code. It would be written:

```
public FunnyFace( Location upperLeft, DrawingCanvas canvas) {
   this( upperLeft.getX(), upperLeft.getY(), canvas );
}
```

From our example above with overloaded methods, you might have expected the body of the new `FunnyFace` constructor to begin with `this.FunnyFace(...)` or even `new this.FunnyFace(...)`, but Java has a different syntax for calling existing constructors from a new constructor. One simply writes `this(...)`. That syntax tells Java to look for another constructor for the class (they all have the same name – `FunnyFace` in this case) with formal parameters whose types correspond to those in the parentheses after `this`. Again, the reason for using this notation rather than copying the old method body and editing each line to use the new parameters is that the constructors are more likely to stay consistent if they all execute the same code.

Overloaded methods should be used sparingly in Java. When used, each of the overloaded variants should do the same thing. Overloading should only be used to allow programmers more

flexibility in the choice of parameters. Having very different methods with the same names will only confuse programmers (including yourself!). Because all constructors for a class must have the same name (the name of the class itself), if a class has more than one constructor, it must be overloaded.

A common error is updating one version of an overloaded operation to have new behavior, but forgetting to update one or more of the other versions. We urge you to guard against this by having variants call one of the other versions so that changes to one are necessarily reflected in all. While this may result in slightly slower execution time than making duplicate copies of the code, you are more likely to stay out of trouble this way.

Finally, we note that one of the first things the Java compiler does when compiling your programs is to rename all of the overloaded methods so that they have distinct names. As a result, overloading has no direct impact at runtime. The only reason for using overloaded methods is if using the same name is a significant advantage for programmers.

4.9 Summary

In this section we showed how to design new classes in Java. We can design classes to represent interesting collections of objects with similar behaviors. We illustrated these ideas by building classes `FunnyFace` and `Timer`. The use of these classes made it much simpler to define programs to drag around a funny face on the screen and a game in which the user is to click on the funny face within a small time interval.

We showed how a method could call another method of the same class using the keyword `this`. We also discussed access control for features of Java by using the keywords `public` and `private`. `Private` features will only be available to constructors and methods of the class in which they are defined, while `public` features are available outside of the class.

We introduced local variables and compared their use with instance variables and parameters. Instance variables are to be used when information must be retained between calls to constructors and methods. Parameters are used to provide values to be used within the bodies of methods. Formal parameters become associated with the values of corresponding actual parameters during each method invocation. These associations are not retained after the execution of the method body. Local variables are used for values that need only be saved temporarily during the execution of a constructor or method.

Finally we discussed overloaded methods and constructors, and under what circumstances it makes sense to provide overloaded features. We also illustrated how to keep overloaded features consistent by having variants call a method or constructor with the same name but slightly different parameters.