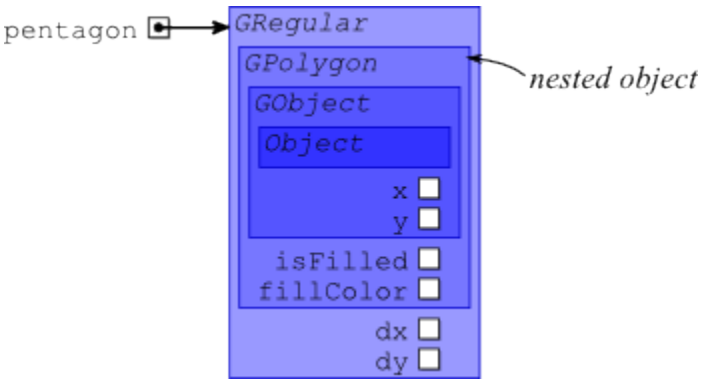
DEFINING SUBCLASSES IN JAVA

15.1. Nested objects

When we've created one class *A* that is a subclass of *B*, any *A* object will then contain within it a *B* object, so that any instance variables defined in *B* will be remembered as part of the *A* object. To be more concrete, suppose we define a class `GRegular` that extends `GPolygon` (which itself extends `GObject`, which extends `GObject`), and then our program creates an instance of `GRegular`.

```
GRegular pentagon = new GRegular(5, 100, 100, 20);
```

How does this appear in memory? We should imagine this new object as having within it a nested object from its superclass. That's not a commonly recognized phrase, but we need a term for the concept in order to discuss it.



As our diagram indicates, we imagine a `GRegular` with its instance variables `dx` and `dy` inside it. But also inside it is its nested `GPolygon` object, with any instance variables defined by the `GPolygon` class; these variables include `isFilled` and `fillColor`, as well as several others not in the diagram. And the nested `GPolygon` object has nested within it a `GObject`object, which contains instance variables `x` and `y`, as well as others not diagrammed. And within it is nested an `Object` object.

This object nesting has an important implication: All those instance variables in `GPolygon` will have to be given their initial values somehow. And the way `GPolygon` was designed to be initialized is through its constructor. As a result, whenever the computer is asked to construct an instance of `GRegular`, it will actually go through three steps.

1. First it allocates enough memory to store all the instance variables required by `GRegular` as well as those required by `GPolygon`, `GObject`, and `GObject`.

2. Then it initializes the nested objects: First it executes an `Object` constructor, then a `GObject` constructor, then a `GPolygon` constructor, each initializing its respective nested object.
3. Finally it enters the `GRegular` constructor to initialize the instance variables specific to `GRegular` objects.

The order here is important: The computer needs to perform the superclass's constructor before it performs the subclass's constructor, because the subclass constructor may very well invoke some methods from the superclass to further configure it. (This indeed happened with `GRegular`: It invoked `addVertex` several times.) For the superclass methods to work, the nested `GPolygon` object must already be initialized. Thus, the `GPolygon` constructor must be completed before we enter the `GRegular` constructor.

15.2. Subclass constructors

Now what happens when the superclass's constructor requires parameters? That issues didn't arise with `GRegular`, because the constructor for its superclass `GPolygon` didn't require any parameters. However, it is an issue that would arise with creating a subclass of a class such as `GOval`.

The Java compiler can't simply divine what parameters to give to the superclass's constructor. So it will require us to insert a special line at the top of our definition of the constructor in the subclass: the `super` line. It's easiest to see this using an example, which we see in Figure 15.1.

Figure 15.1: The `Ball` class, extending `GOval`.

```
3  public class Ball extends GOval {
4      private double dx;
5      private double dy;
6
7      public Ball(double centerX, double centerY, double radius) {
8          super(centerX - radius, centerY - radius, 2 * radius, 2 * radius);
9          double angle = Math.random() * 2 * Math.PI;
10         dx = Math.cos(angle);
11         dy = Math.sin(angle);
12     }
36
37     public void step(double winWidth, double winHeight) {
38         this.move(dx, dy);
39         double x = this.getX();
40         double y = this.getY();
```

```
41          double diam = this.getWidth();
42          if(x < 0 || x + diam >= winWidth)  dx = -dx;
43          if(y < 0 || y + diam >= winHeight) dy = -dy;
44      }
45  }
```

Because `Ball` extends a class (`GOval`) whose constructor requires parameters, our `Ball` constructor is required to have a **`super`** line as the first statement of its constructor. We see this in line 8. This line has the word **`super`** followed by a set of parentheses and then a semicolon. Inside the parentheses is a list of all the parameters that should be used for constructing the `GOval` nested object within the `Ball` object being constructed.

In this case, the `Ball` constructor takes three parameters specifying the *x*- and *y*-coordinates of the ball's center and the ball's radius. However, `GOval` expects the coordinates of the oval's top left corner and the oval's width and height. Thus, in the **`super`** line we specify how to compute these values based on the parameters provided to the `Ball`.

Java will insist that the **`super`** line always be the very fisrt line in the constructor. This is occasionally irritating because you want to perform some calculations to determine the parameters for the **`super`** line. But Java will insist that you somehow must fit all the parameter calculations within the **`super`** line.

15.3. Overriding methods

Sometimes there are methods in the superclass that don't work as we would prefer that they work in the subclass. We might, for instance, argue that the `Ball` class's `getX` method should actually return the *x*-coordinate of the ball's center rather than its left side. Changing the `getX` method's behavior is simple enough: As we define the subclass, we simply define a new method with the same name and same parameters. This new definition replaces the inherited definition of the method.

But how should we define its body so that it returns the center's *x*-coordinate? Somehow we have to retrieve that number. It comes as a parameter into the constructor, so one possibility is to define an instance variable and have the constructor stash the parameter in that instance variable, which can then be returned within the `getX` method. This approach, though, is stylistically poor: It uses an extra instance variable, which is something that we should avoid when possible. And in this case, the information is sitting right there in the nested `GObject` object.

But how can we get at that `GObject` instance variable? The variable is **`private`**, so we can't get at it directly. Normally we'd retrieve it using the `getX` method, but that won't work here, because we're in the process of redefining it. Any attempt to invoke `getX` method will simply re-invoke the method

we're defining, getting the computer stuck continually invoking the same method until it runs out of memory.

What we want is some way to specify that we actually want to use the `getX` method in the superclass. As it happens, Java provides a technique for this: We can again use the`super` keyword, writing `super.getX()`. We can use this technique here.

```
public void getX() {
    return super.getX() + this.getWidth() / 2;
}
```

This technique is available everywhere within the `Ball` class. Our `step` method, for instance, will need to be modified since it's currently written based on `getX` returning the *x*-coordinate of the ball's left side; one way to change it is to write `super.getX()` in place of `this.getX()`.

However, this technique is not available outside the `Ball` class. Once we override `getX`, the `GOval` class's `getX` method is completely unaccessible for any `Ball` objects.

15.4. Dynamic dispatch

If you have variable `boing` referencing a `Ball` object and you wanted to use `GOval`'s `getX` method to retrieve the *x*-coordinate of the ball's left side, one way you might consider doing it is the following.

```
GOval oval = boing;
this.println("left side is " + oval.getX()); // wrong!
```

This technique doesn't work. The Java language is defined so that when you invoke an instance method, the computer checks the object's actual type, and it invokes the method that applies to the type. Though the term isn't all that important to our purposes, this process of determining the actual type is called *dynamic dispatch*. In this case, `oval` — though it is a `GOval` variable — actually references a `Ball` object, so the computer enters the `getX` method defined in the `Ball` class. It would only enter `GOval`'s `getX` method if`GOval` didn't override the method.

This behavior isn't particularly intuitive, but Java is defined this way because the behavior is often quite useful. As an example, consider the below fragment to remove all shapes in the window containing the point (`x`, `y`).

```
for(int i = 0; i < this.getElementCount(); i++) {
    GObject o = this.getElement(i);
    if(o.contains(x, y)) this.remove(o);
}
```

Here we are using `GObject`'s `contains` method. But this method actually behaves differently for different shapes: Testing whether a point lies within an oval is different from testing whether it lies within a polygon. The `contains` method works differently for the different classes because, in fact, subclasses of `GObject` like `GOval` and `GPolygon` override the `contains` method defined in `GObject`. (As it's defined in `GObject`, it checks whether the point lies in the bounding box of the object. That's good for rectangles, so `GRect` doesn't need to override the method; but it's not good for much else.

Even though the behavior is often useful, it can also be problematic. And the case of overriding `getX` and `getY` illustrates this. The program of Figure 15.2 uses the `Ball` class to illustrate several balls bouncing around the window.

Figure 15.2: The `BouncingBalls` program.

```
1  import acm.program.*;
2  import acm.graphics.*;
3
4  public class BouncingBalls extends GraphicsProgram {
5      public void run() {
6          for(int i = 0; i < 10; i++) {
7              double x = Math.random() * (this.getWidth() - 20) + 10;
8              double y = Math.random() * (this.getHeight() - 20) + 10;
9              Ball b = new Ball(x, y, 10);
10             b.setFilled(true);
11             this.add(b);
12         }
13
14         while(true) {
15             this.pause(20);
16             for(int i = 0; i < this.getElementCount(); i++) {
17                 Ball b = (Ball) this.getElement(i);
18                 b.step(this.getWidth(), this.getHeight());
19             }
20         }
21     }
22 }
```

If we were to execute this program, we'd find that the balls don't actually bounce off the walls of the window as we might hope. Instead, they bounce off an invisible line 10 pixels right from the window's left edge, another 10 pixels down from the window's top edge, another 10 pixels right of

the window's right edge, and another 10 pixels down from the edge. It works this way even though the `step` method has been properly modified to account for the change in behavior of `getX` and `getY`.

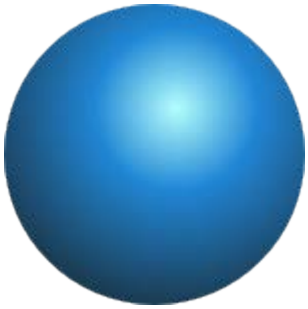
The problem here is rather subtle, and you'd have a hard time figuring it out based on what you've read in this book. But the problem is instructive, too. In fact, the balls are changing their locations exactly as we want. The problem is that the circles aren't being *drawn* exactly as we want. You see, `GOval` has a method named `paint`, which is used for painting the oval onto the screen. We haven't had to worry about this method because `GraphicsProgram` deals with invoking it for us whenever appropriate. It happens that `GOval`'s`paint` method uses `getX` and `getY` to determine where to paint the window. We overrode these methods, so now whenever `GraphicsProgram` enters `GOval`'s `paint` method, that method will invoke `getX`, which will enter the overridden version of `getX` found in the `Ball` class. It of course returns the *x*-coordinate of the ball's center, but the `paint` method treats the returned value as the ball's left edge. As a result, all the balls are drawn 10 pixels down and to the right of where they actually are.

One way of repairing this is to also override the `paint` method. With this change, the above `BouncingBalls` program works correctly.

```
public void paint(Graphics g) {
    int x = (int) super.getX();     int y = (int) super.getY();
    int w = (int) super.getWidth(); int h = (int) super.getHeight();
    if(this.isFilled()) {
        g.setColor(this.getFillColor());
        g.fillOval(x, y, w, h);
    }
    g.setColor(this.getColor());
    g.drawOval(x, y, w, h);
}
```

(We'd probably also want to override `contains`, since it also uses `getX` assuming it returns the ball's left edge. This behavior isn't apparent in the `BouncingBalls` program, because it doesn't use the `contains` method.)

Of course, another way of repairing it is to give up on overriding `getX` and `getY`, instead defining new methods `getCenterX` and `getCenterY`. That approach makes sense here, because `GOval` was written with the intention that `getX` and `getY` would return the left and top edges of the oval.



Sometimes, though, overriding a method makes sense: Suppose we wanted a class `GSphere`, which appears as a circle on the screen but the interior is drawn in a way indicating a three-dimensional sphere. The easiest way of doing this would be to create a subclass of `GOval`, and then we'd override `GOval`'s `paint` method.

Source : http://www.toves.org/books/java/ch15-makesub/index.html