Introducing Objects and Object-Oriented Programming

Prior to the development of modern computers, the word computing mostly implied numerical calculation. Today, computing encompasses all forms of electronic manipulation of objects: numbers, text, sound, music, photographs, movies, genes, etc., and even ideas! Computing can be used as an extension of your mind. It is in this form that a computer becomes an enormously powerful device. You might be wondering; how do computers manipulate ideas? Well, not quite in the sense you might have imagined. If you think about objects like numbers, text, sound, music, and even genes and how we are able to process them or do computing with them, then the notion of representing and processing of ideas is not so farfetched. In fact, creative coding is all about the computational representation and manipulation of ideas. The key enabler here is the ability to represent the thing that you are interested in inside a computer and then manipulate it. Most modern programming languages provide facilities for specifying and modeling any object. Further, there is also a need to keep programs organized into modules corresponding to their logical components. You have already seen some rudimentary forms of modularization facilitated by functions. Functions help modularize as well as parameterize specific tasks. Subcomponents that make up a program can be further organized into modules that can serve not only to keep the overall program organized but also to provide the ability to design and create reusable components that could be used in several programs. In this chapter, we will introduce you to a program design methodology called object-oriented programming or OOP. OOP facilitates the design and modeling of objects in an organized and coherent manner.

To get you motivated, consider the popular mobile game app Angry Birds designed by the Finnish video game developer Rovio Entertainment Inc. (www.rovio.com). The game app, originally launched in December 2009, surpassed 1 billion downloads in May 2012. If you have been hiding under a desolate rock the last few years, here is a quick capsule summary of the game: the green pigs have stolen the bird’s eggs and are hiding inside buildings and other structures. The birds are angry, have projectile destructive powers and you, the player, can help the birds dish out revenge on the pigs by launching them from a catapult.

Angry Birds is a physics-based game. Objects in the game obey simple laws of physics. The genius behind the success of the game lies in a unique combination of artistic design, a simple back story, and requires a combination of logic, skill, and brute force to win the most points.

Imagine if you were part of the team at Rovio that created this app. You would have at least a dozen or more programmers in the team working on designing and coding the app. As mentioned previously, several organizational techniques are employed in keeping a project like this from disintegrating into total chaos. When Niklas Hed, one of the creators of Angry Birds, was young, one of the programs he wrote was to create a bouncing ball (see Figure 6-1 for an idea of what this entails). We are going to use the bouncing ball as an example to help you learn the basics of OOP design and how to implement it in Processing.
The key insight underlying object-oriented design is to think about programming problems the same way you might model problems in the real world. It is a way to identify, extract, and abstract the most important aspects of things in the problem domain. For example, in the Angry Birds game, the birds do not have any feet or feathers to flutter, mostly fly as projectiles, bounce around when hit, and so on. Each problem domain has a set of objects that typically interact with each other. The green pigs are green, sometimes wear helmets, and mostly sit in one place, until hit by another object (a piece of a wall or a bird), and explode or die when the impact of a hit is strong enough. The “castles” or structures themselves are objects. The basic building block in object-oriented design, the most important conceptual basis, is an object. Objects possess properties or attributes and are capable of some functionality or behavior. Birds in the game, for example, have properties like their color, size, position, velocity, type of bird, and so on. Functionality of a bird includes a launch operation, the flying motion, the screeching, and so on. This is shown below:

The bouncing ball has attributes specifying its size, current position, and color, with behaviors like move, bounce, and display. See Figure 6-1.

<table>
<thead>
<tr>
<th>Angry Bird</th>
<th>Ball</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Properties:</strong></td>
<td><strong>Properties:</strong></td>
</tr>
<tr>
<td>type</td>
<td>size</td>
</tr>
<tr>
<td>color</td>
<td>position</td>
</tr>
<tr>
<td>size</td>
<td>color</td>
</tr>
<tr>
<td>position</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><strong>Behaviors:</strong></td>
<td><strong>Behaviors:</strong></td>
</tr>
<tr>
<td>launch</td>
<td>move</td>
</tr>
<tr>
<td>fly</td>
<td>bounce</td>
</tr>
<tr>
<td>screech</td>
<td>display</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 6-1: Properties and behaviors of Angry Bird and ball objects
Try This: Think of other kinds of objects you may have come across in solving problems earlier. For example, the dataset of stock prices from Chapter 5 or some other object like a starship, a bank account, or a birthday party. Can you identify the objects, their attributes, and their behavior? Next, think about an entire problem domain, the kinds of objects it might have, and their relevant properties and behaviors.

Classes: Object Factories

In a typical program, or problem domain, there are several objects of the same kind: several Angry Birds, many bouncing balls, several bank accounts, etc. In OOP, objects are defined using classes. A class essentially defines the objects and serves as a factory for creating or instantiating specific objects. My car, for example, is a specific instantiation of the 2012 VW Passat models made by the Volkswagen Company. It is unique, in that it has a unique VIN or Vehicle Identification Number, but it is similar in most other respects to all the other Passats manufactured in the 2012 edition. They all have several attributes (like color, number of doors, etc.), and functionalities or behaviors whose manifestations might distinguish one from another and yet they all belong to the same class.

The preceding descriptions of Angry Bird and Ball are considered class definitions. Given a class, say Ball, one can create (or instantiate) several objects from it. For example, three balls labeled b1, b2, and b3. Each of the balls has the same internal structure as defined by the class and yet they each are distinct objects distinguished by the different values of their properties. Ball b1 might be small in size, green in color as opposed to b2, which might be large and blue. Before we discuss behaviors further, let’s make what we have learned more concrete in terms of programming a given object-oriented design in Processing.

Object-Oriented Programming in Processing

Remember, the point of object-oriented design is to organize potentially chaotic large programs into manageable, logical components. After you have outlined the classes or objects required for your program, you can use the object-oriented programming features of Processing to do the coding. Let's begin by defining the Ball class outlined previously. We will start small, so as to grasp the fundamentals of OOP design, as well as their implementation in Processing. As we proceed, we will add more sophistication to our programs by way of extending and enhancing the design. Here is how you can define the Ball class:

class ball {  // Define the ball class
    // Attributes
    float x, y;  // the x and y location
    color ballColor;  // its color
    float radius;  // its radius in pixels
}  // class ball

This code defines a class called ball. Attributes of the class are defined as variables (they are also sometimes called fields). Thus, x and y specify the x- and y- coordinates of the ball’s position, ballColor, and radius will be used to store the color and size, respectively. The class syntax is very simple:
class ClassName {
    // Attributes
    ...
    // Behaviors
    ...
}

To define any class, you have to specify the name of the class in the structure as shown previously. Attributes or properties are defined as regular variables inside the body of the class.

Defining a class is essentially equivalent to defining a new type name in Processing. That is, just like the type names int, float, etc. you have extended processing’s type names by an additional type, ball. Now, you can define variables of type ball:

ball b1, b2, b3;

As we discussed in Chapter 5, ball is considered as an aggregate type. It is an aggregate of all the properties and behavior defined in the class. Also, from what you learned earlier, the variables b1, b2, and b3 are reference variables and after the above definition, their values will be null. Next, we learn how to create an instance of the ball object. For that, the class definition has to include a constructor. A constructor is a set of instructions that will be executed each time you create a new object, or instance of a class. The name of the constructor, in Processing, is the same as the name of the class (see below). In addition, there are two things needed here. First, you need to create a new instance of a ball object. This is done, similar to arrays, using the new command:

b1 = new ball();

The constructor is defined inside the class definition as shown here:

```java
class ball {  // Define the ball class
    // Attributes
    float x, y;  // the x and y location
    color ballColor;  // its color
    float radius;  // its radius in pixels
    // Constructor
    ball() {
        // The ball has an initial random location in the sketch window
        x = random(width);
        y = random(height);
        // All balls are red
        ballColor = color(255, 0, 0);
        // All balls are of radius 25 pixels
        radius = 25;
    }  // ball()
}  // class ball
```

The constructor is invoked whenever a new object is created. It serves the purpose of creating an actual object residing in the computer’s memory. The memory allocated contains space for storing all the object’s attributes and their values. Additionally, when an object is constructed or instantiated, the constructor is responsible for setting the initial values of the properties of the object being created. In the preceding code, the x- and y- locations (x and y) are set to a random
location in the sketch window; the color (ballColor) is set to red, and the radius to 25 pixels. In the following set of commands:

```java
b1 = new ball();
b2 = new ball();
b3 = new ball();
```

Three ball objects are created. They will all be red in color, 50 pixels in diameter, and located on three random places in the sketch. To see the three ball objects rendered in the sketch you have to define the display() behavior. Behaviors are defined as functions in the class body following the constructor. For example, the behavior to draw a ball object (display()) is shown here:

```java
class ball {
    // Define the ball class
    // Attributes
    float x, y;       // the x and y location
    color ballColor;  // its color
    float radius;     // its radius in pixels
    // Constructor
    ball() {
        // The ball has an initial random location in the sketch window
        x = random(width);
        y = random(height);
        // All balls are red
        ballColor = color(255, 0, 0);
        // All balls are of radius 25 pixels
        radius = 25;
    } // ball()
    // Behaviors
    void display() {
        // display the ball
        // set color attributes
        noStroke();
        fill(ballColor);
        // draw the ball
        ellipse(x, y, 2*radius, 2*radius);
    } // display()
} // class Ball
```

We now have a minimal, but usable class definition of a ball object. The structure of the complete program shows how to create and display three balls in the sketch window:

```java
// Sketch: 6-1: Creating and displaying three ball objects
ball b1, b2, b3;
void setup() {
    size(400, 400);
    smooth();
    // create the balls
    b1 = new ball();
    b2 = new ball();
    b3 = new ball();
}
```
b3 = new ball();
} // setup()

void draw() {
    background(255);
    // display the balls
    b1.display();
    b2.display();
    b3.display();
} // draw()

class ball {   // Define the ball class

    // Attributes
    float x, y;       // the x and y location
    color ballColor;  // its color
    float radius;     // its radius in pixels

    // Constructor
    ball() {
        // The ball has a random location in the sketch window
        x = random(width);
        y = random(height);
        // All balls are red
        ballColor = color(255, 0, 0);
        // All balls are of radius 25 pixels
        radius = 25;
    } // ball()

    // Behaviors
    void display() {
        // display the ball
        // set color attributes
        noStroke();
        fill(ballColor);
        // draw the ball
        ellipse(x, y, 2*radius, 2*radius);
    } // display()
} // class ball

Try This: Enter the preceding sketch and run it a few times. Try to modify the attributes (x, y, radius, ballColor) in the Ball class constructor and observe the outcomes. Try creating an array of ten Ball objects and drawing them.
In OOP terminology, functions defined inside classes are also called *messages* or *methods*. Thus, the `display()` behavior defined in the ball class is the method for drawing ball objects. The syntax for invoking methods on an object is shown here:

```
object.method(...);
```

The way to read the preceding syntax is to say, “apply *method* to *object*,” or “send the message *method* to *object*.” In your programs, you may have other kinds of objects, like green pigs, for example. How a green pig is drawn will be specified by a `display()` method inside the class definition for green pig objects. Thus, in your program you may have:

```java
ball b1, b2, b3;
greenPig p1, p2, p3;
```

After these objects are instantiated, you might want to draw some of these objects:

```java
b1.display();
p3.display();
```

Which `display()` method will be executed for `b1`? For `p3`? The ones defined in their respective classes. This is one of the advantages of OOP-- you can define several methods by the same name (such as `display()`). The type of object the method is applied to determines which specific `display()` method is executed.

### Customizing Instances

All the ball objects created so far were defined to be of same size and color. What if you wanted to customize each object’s size, color, or location? This is done by specifying parameters to the constructor as shown here:
ball(float px, float py, float r, color c) {
    // set up ball
    // with position (px, py)
    x = px;
    y = py;
    // radius, r pixels
    radius = r;
    // color, c
    ballColor = c;
} // ball()

Now, for example, you can create a completely customized ball object:

ball b1 = new ball(width/2, height/2, 50, color(0));

It assigns to b1 a ball object that is located in the center of the sketch, 100 pixels in diameter, and colored black. Often, it is also the case that you may just want to customize a selection of attributes: Say, the radius and color. Processing allows you to create multiple constructors with different parameters. Thus, in your class definition, you could define all of the following constructors:

// Default Constructor
ball() {
    // When created, the ball has a random location
    x = random(width);
    y = random(height);
    // All balls are red
    ballColor = color(255, 0, 0);
    // All balls are of size 50 pixels
    ballSize = 50;
} // ball()

ball(float px, float py, float r, color c) {
    // set up Ball with position (x, y)
    x = px;
    y = py;
    // size, r pixels
    radius = r;
    // color, c
    ballColor = c;
} // ball()

ball(float r, color c) {
    // set up ball with random position
    x = random(width);
    y = random(height);
    // radius, r pixels
    radius = r;
} // ball()
The number and type of arguments used to define a constructor (or a function, or a method) denote its *signature*. For multiple definitions with the same name, as long as the signature is unique, you can define as many constructors (or functions, or methods) as you like. In the preceding code we have defined three constructors with the signatures shown here:

- `ball()`
- `ball(float, float, float, color)`
- `ball(float, color)`

When you write the following statements:

```plaintext
// create the balls
b1 = new ball();
b2 = new ball(width/2, height/2, 50, color(0));
b3 = new ball(25, color(125, 125, 125));
```

The constructor corresponding to the matching signature, as determined by the number, type, and ordering of the arguments in the invocation, is called. A resulting sketch from the preceding statements is shown in Figure 6-3.

Figure 6-3: Customized ball objects
A Useful Keyword: this

Look carefully at the definition of the second `ball` constructor in the preceding example. Notice that we named its parameters `px` and `py`. What would happen if we named them `x` and `y`? This is shown here:

```java
ball(float x, float y, float r, color c) {
    // set up ball with position (x, y)
    x = x;
    y = y;
    // radius, r pixels
    radius = r;
    // color, c
    ballColor = c;
} // ball()
```

We have only shown one constructor here. Notice that the names of the arguments of `ball()`, `x` and `y`, are the same as the position attributes. Thus, when you see the statements

```java
x = x;
y = y;
```

it is not clear whether the attributes are being set to the values of the parameters. Can you tell which is which? If you remember the rules in Processing, if two or more variables with the same name are accessible in the same scope, the names refer to the variables which were defined most recently, and assignment operations occur form right to left. In the preceding, the `x` and `y` in both statements, on both sides of the `=`-operator refer to the parameters `x` and `y` and not the attributes. Thus, these statements will not have the desired effect of setting the values of attributes `x` and `y` to the values of parameters `x` and `y`. You have already seen one solution to this: name them differently. Another way around it, is to make use of the object qualifier, `this`. `this` is a variable name that refers to the current object when code inside a class is being executed. Using this you can write the preceding constructor as shown here:

```java
class ball {    // Define the ball class
    // Attributes
    float x, y;               // the x and y location
    color ballColor;  // its color
    float radius;        // its size in pixels

    ball(float x, float y, float r, color c) {
        // set up Ball with position (x, y)
        this.x = x;
        this.y = y;
        // radius, r pixels
        this. radius = r;
        // color, c
        this.ballColor = c;
    } // ball()

    ...
} // class ball()
```
Now, which variable (attribute or parameter) is being referred is clear. Another convention that avoids using this that is commonly used in these circumstances is to prefix the parameter name with an underscore (_):

class ball {    // Define the ball class
    // Attributes
    float x, y;               // the x and y location
    color ballColor;  // its color
    float radius;        // its size in pixels
    ...
    ball(float _x, float _y, float r, color c) {
        // set up Ball with position (x, y)
        x = _x;
        y = _y;
        // radius, r pixels
        radius = r;
        // color, c
        ballColor = c;
    } // ball()
    ...
} // class ball()

An underscore (_) is a valid part of a variable name and thus can be used as shown here in these types of situations.

Inside a class, the specification of the this qualifier is optional. In fact, the use of this in setting the radius and color is not necessary because there is no ambiguity. It is a good idea, especially if you intend to use same names to mean different things in different contexts, to explicitly use this as a qualifier.

There are several other uses of the this keyword. For instance, you can use this(...) inside the constructor to invoke a single constructor that initializes objects:

    // Default Constructor
    ball() {
        this(random(width), random(height), color(255, 0, 0), 25);
    } // ball()

    ball(float r, color c) {
        this(random(width), random(height), r, c);
    } // ball()

    ball(float x, float y, float r, color c) {    // The main constructor
        // set up ball with position (x, y)
        this.x = x;
        this.y = y;
        // size, r pixels
        radius = r;
        // color, c
        ballColor = c;
    } // ball()
Not only do you get more succinct code, you also reduce the chance of errors. Now, all initialization is done inside a single constructor. Other constructors just invoke the ‘main’ constructor where all initializations occur.

Many object-oriented programming languages use the name this to refer to the object. Others make explicit use of a self-reference name, which can be this, or self, or something else that you can define. It also comes in handy in situations where an object’s attributes themselves are other objects. We will show this later in the chapter.

**Tabs: Organizing Code**

In the preceding example, we presented the code for the ball class in line with the rest of the program: setup(), followed by draw(), followed by the ball class definition. If your program has several classes, as it will later on, the Processing IDE provides an excellent way to organize them into separate files. In fact, from the very beginning in your coding process, you are encouraged to utilize the Tabs feature of the Processing IDE. Take a careful look at your previous program in the Processing IDE. We show a snapshot of our version in Figure 6-4.

![Figure 6-4: The Sketch_6_1 tab and where to look for new tabs](image)

Notice how, just above the code, on the top left, the name of the sketch (Sketch_6_1) appears in a tab. In the IDE, your code can be organized into tabs, and each tab corresponds to a separate file in the sketch folder. In programs that use object-oriented design, it is a good idea to place each class separately in its own tab/file. Thus, your “main” program: the setup(), draw(), functions that use the classes and form the main part of your program, will be in the main tab. You can place each new class you define under a separate tab. To create a new tab, click on the small dark arrow, drawn in a shaded box, above the code, on top right (shown circled in Figure 6-4). You get a drop-down menu (see Figure 6-5).
Select New Tab and you see a yellow text box where you have to enter the name of the class you are defining in the new tab. Enter the name of the class (ball) as shown in Figure 6-6, exactly as you intend to define it, and press the OK button. A new tab will be created with a fresh code window. Enter your code for the ball class in this new window. This is shown in Figure 6-7. As you can see, the code that defines the class is now separated from the code that uses it. You can define as many tabs as you like in this manner to keep your programs organized. The contents of each tab is stored in a separate file in the sketch folder. In the situation shown in Figure 6-7, the tab with the ball class is stored as a file ball.pde in the sketch folder.

Defining Additional Behaviors: Motion

A number of creative effects can be generated by defining animations or motions of objects in sketches. Here we use the ball example to illustrate the design and implementation of simple motion. The movement of the object is modeled inside the move() method. To do the animation, the move() method can be used in draw() as follows (in the case of our example with three ball objects):

```java
void draw() {
    background(255);
    // display the balls
    b1.move();
    b1.display();
    b2.move();
    b2.display();
    b3.move();
    b3.display();
} // draw()
```

That is, before displaying a ball, its position is updated (that is, moved to its next location). The movement of the ball itself can be modeled using simple approximations of physical behavior utilizing the speed (rate at which a ball travels). Speed can be modeled as an increment of the ball’s position. (that is, the change in the balls x- and y- coordinates). Here is a very simple version:
void move() {
    x = x + dx;
    y = y + dy;
} // move()

You need to define dx and dy as additional attributes for the class ball as follows:

float dx = 1;
float dy = 1;

We have initialized dx and dy in the definition itself. Alternatively, you can just define the variables and later initialize them inside the constructor.

Try This: Add the preceding code to your sketch and observe the behavior. Because the position of the ball changes by 1 in both x- and y- directions, the balls move diagonally across the screen, until eventually they leave the sketch window. We address that next. But first, experiment with different values of dx and dy. The way it is defined above, each ball travels at the same speed. You can easily vary each ball's speed by using random values:

float dx = random(1, 3);
float dy = random(1, 3);

Again, instead of initializing these in the definition, you can choose to initialize these in the constructor(s). To ensure that a ball stays within the bounds of the sketch window, you can additionally define a bounce behavior:

void bounce() {
    if (x > (width-radius)) {   // bounce against the right edge
        x = width-radius;
        dx = -dx;
    }
    if (x < radius) {   // bounce against the left edge
        x = radius;
        dx = -dx;
    }
    if (y > (height-radius)) {   // bounce against the bottom edge
        y = height-radius;
        dy = -dy;
    }
    if (y < radius) {   // bounce against the top edge
        y = radius;
        dy = -dy;
    }
} // bounce()

Each of the four if-statements tests to see whether the ball has reached an edge of the sketch window. Depending on which edge (left, right, top, or bottom) the displacement (dx or dy) is negated so that the ball reverses direction. Additionally, notice the conditions are checking against the outer edge of the ball and not its center (represented by x, and y). Thus, the condition (x < radius) in the second if-statement is checking to see whether the left edge of the ball is less than the radius of the ball, implying that the ball has displaced beyond the left edge. In that case, we
set the ball’s center back to align it with the left edge \((x = \text{radius})\), and then reverse the \(x\)-direction \((dx = - dx)\). These kinds of adjustments are typical in the world of graphics and animations where the increments, in terms of float values, can lead to the objects crossing the boundaries. By resetting the objects back to the boundary you can avoid situations where the animations go awry. Study this carefully, implement it by adding the \texttt{bounce()}\ method to the class \texttt{ball} and modifying the \texttt{move()}\ method as shown here:

```java
void move() {
    x = x + dx;
    y = y + dy;
    bounce();   // check for a bounce
} // move()
```

This implements a ball continuously bouncing from the “walls” of the sketch. Alternatively, you can model a different behavior where the ball behaves like a bouncing ball using a model of \textit{gravity}. Gravity is simply acceleration or the rate at which the speed changes. In this instance, we will be modeling the balls falling from top to bottom. This is implemented in the following definition of \texttt{move()}:

```java
void move() {
    x = x + dx;
    y = y + speed;
    speed = speed + gravity;
    // check to see if it bounces
    bounce();
} //move()
```

We will need to define four additional attributes for the \texttt{ball} objects:

```java
float speed = 5.0;       // The speed at which the ball is moving
float gravity = 0.1;    // the rate of increase of speed
float dx = 1;                // amount of lateral movement
float dampen = -0.9;// amount of dampening after each bounce
```

Given these definitions, let’s try to understand the model of a ball’s motion. Each time \texttt{move()}\ is called, the ball’s \(x\)-position \((x)\) is updated by \(dx\), its \(y\)-position \((y)\) is updated by \(speed\), and \(speed\) is updated by \(gravity\). We have used some initial values for \(speed\) (5.0), \(gravity\) (0.1), and \(dx\) (1). You can feel free to experiment with these in your sketch (see below). Further, when a ball reaches the bottom of the sketch window it should bounce. The complete sketch, including the new version of the \texttt{bounce()}\ method is shown here:

```java
// Sketch: 6-2: Creating and displaying three ball objects
ball b1, b2, b3;
void setup() {
    size(400, 400);
    smooth();
    // create the balls
    b1 = new ball();
    b2 = new ball();
    b3 = new ball();
} // setup()
```
void draw() {
    background(255);
    // display the balls
    b1.move();
    b1.display();
    b2.move();
    b2.display();
    b3.move();
    b3.display();
} // draw()

// Sketch: 6-2: File ball.pde

class ball {    // Define the ball class

    // Attributes
    float x, y;          // the x and y location
    color ballColor;     // its color
    float radius;        // its radius in pixels

    float speed = 5.0;     // The speed at which the ball is moving
    float gravity = 0.1;   // the rate of increase of speed
    float dx = 1;          // amount of lateral movement
    float dampen = -0.9;   // amount of dampening after each bounce

    // Constructors
    // Default Constructor
    ball() {
        this(random(width), random(height), 25.0, color(255, 0, 0));
    } // Ball()

    ball(float x, float y, float r, color c) {
        // set up ball with position (x, y)
        this.x = x;
        this.y = y;
        // size, r pixels
        radius = r;
        // color, c
        ballColor = c;
    } // ball()

    ball(float r, color c) {
        this(random(width), random(height), r, c);
    } // ball()

    // Behaviors

    void display() {
        // display the ball
        // set color attributes
        noStroke();
        fill(ballColor);
    } // display()
// draw the ball
ellipse(x, y, 2*radius, 2*radius);
} // display()

void move()
{
x = x + dx;
y = y + speed;
speed = speed + gravity;

    // check to see if it bounces
bounce();
} // move()

void bounce()
{
    if (x > (width-radius)) {   // bounce against the right edge
        x = width-radius;
dx = -dx;
    }

    if (x < radius) {   // bounce against the left edge
        x = radius;
dx = -dx;
    }

    if (y > (height-radius)) {   // bounce against the bottom edge
        y = height-radius;
speed = speed * dampen ;
    }

} // bounce()
} // class ball

Take a close look at the above code. In the bounce() method, if the ball has reached the edge of the sketch, we change the direction of the ball’s lateral displacement (dx). Similarly, we check to see whether the ball has reached the bottom of the sketch window, if so, we negate, and decrement the speed. Notice also the use of the dampen variable to reflect a dampening of speed after each bounce. Study this code carefully and make sure you understand it.

Try This: Run the preceding sketch (including using a separate tab for the ball class). Next, play with the values of speed, gravity, and dx, as well as the dampening value of speed (dampen). Run the sketch over and over again and carefully observe the motion of each ball. You will need to adjust the values of various attributes to get visually pleasing bouncing behavior. Notice what happens when a ball is no longer bouncing. How can you make it stop? Hint: Think about modeling friction. As defined previously, each ball has the same starting values for speed, gravity, and dx. How can you modify the program so each ball has different starting values of these attributes?

In the preceding example, we called the bounce() method inside the move() method. Alternatively, you could choose to invoke it from draw():

void draw()
{
... 
b1.move();
b1.bounce();
b1.display();

... } // draw()

You will notice that object-oriented design and development allows for incremental enhancements and, at the same time, keeps your programs well organized. Also, at each step, there are many decisions that need to be made: What attributes to add? How and when to initialize them? How are object instances customized? What behaviors are needed? How they are invoked? As you make these decisions you will be confronted with choices in the implementation. It is always possible to achieve the same behavior in a sketch out of many possible implementation choices. Which choices you make should be based on keeping a clean design, ensuring future extendibility, and providing as many opportunities for parameterization as might be sensible.

OOP and Encapsulation in Processing

One of the tenets of object-oriented programming is the idea of encapsulation. There are two key aspects to encapsulation, one of which you have now seen: it is a way of creating new aggregate types and bundling their data (attributes) and functionality (behavior/methods) together into a separate module (a class). This type of modularization is good for keeping programs organized. Additionally, it allows for reuse of useful classes of objects. The second, perhaps equally important aspect, of encapsulation is information hiding. This is a way of restricting access to some parts of an object's definition. For example, in the example in the previous section, the draw() function of the sketch uses the move() and display() methods on specific ball objects. We also saw two different definitions of move(): one to set the ball in linear motion, and one that implements a bouncing behavior. From the perspective of the draw() method (also called a client of the class ball), the details of how the motion is implemented are hidden. This type of procedural abstraction is already available via functions in any programming language. This type of modularization allows you to change the move() method without requiring any changes to the draw() method. This is considered good program design.

While it is straightforward to restrict the details of implementation of a class’s methods, the issue of restricting access to a class’s attributes requires further elaboration. Here is the fundamental question: Is it possible to access an object’s attributes (and their values) outside the class? For instance, what happens when you do the following?

void setup() {
    ...
    b1 = new ball(...);
    b1.x = 42;
    ...
} // setup()

Encapsulation in object-oriented programming languages typically allows you, the programmer, to decide whether or not this is something you want to allow. If not, how do you restrict access to an object’s attributes?

First of all, in Processing, you are allowed to have complete access to an object’s attributes. That is, Processing’s manifestation of object-oriented programming does not give you mechanisms for restricting such access. If it did, you would be able to designate each definition (attribute as well as method) in a class to be public, or private. Once you designate something as private, access to that attribute is no longer permitted outside of the class. In the example, in setup() where b1.x is being modified it will result in a compilation error. The programming language Java, which is an object-oriented language, and which is the underlying implementation substrate for Processing, allows the private or public designation of class definitions. Java allows all definitions to be prefixed with the keyword public or private:
public class Ball {
    private float x, y;
    ...
    public void move() {
        ...
    } // move()
    ...
} // class Ball

However, this Java feature is not available in Processing. As we mentioned previously, due to the way Processing is implemented on top of Java, all definitions in a sketch are considered public. So, it will be up to you to maintain discipline in your programs so as not to access those things which might be considered private in true object-oriented design. Sometimes, though, it is necessary to be able to do so in many situations.