Ball Worlds

In the intuitive description of object-oriented programming presented in Chapter 1, an object-oriented program was described as a universe of interacting agents. However, in our first example Java program, in Chapter 4, we did not actually create any new objects, but only used the static method named main in the program class.

Our second program is slightly more complex in structure, although hardly more complicated in functionality. It places a graphical window on the user’s screen, draws a ball that bounces around the window for a few moments, and then halts. A screen shot of this application in execution is shown as Figure 5.1.

Our second example program, or paradigm, is constructed out of two classes. The first of these appears in Figure 5.2. Again, we have added line numbers for the purposes of reference; however, these are not part of the actual program. The reader should compare this program to the example program described in the previous chapter, noting both the similarities and differences. Like the previous program, this program imports (on line 1) information from the Java library. Like the earlier program, execution will begin in the method named main (lines 4–10), which is declared as static, void, and public. Like all main programs, this method must take as argument an array of string values, which are, in this case, being ignored. The main procedure creates the window for the application, instructs this window to display itself (the method show), then moves the ball 1000 times (the loop surrounding the invocation of the method run) before halting the application.

---

1 In order to draw more attention to the Java code itself, the programs presented in this text have purposely been written using very few comments. In practice, comments would usually be used to describe each method in a class.
This program also incorporates a number of new features. These are summarized by the following list, and will be the subject of more detailed discussion in subsequent sections.

- The class defines a number of private internal variable data fields, some of which are constant, some of which are initialized but not constant. These data fields will be described in detail in Section 5.1.

- The main() method creates an instance of the class BallWorld. This object is initialized by means of a constructor. A constructor is a method that automatically ties together the actions of object creation and object initialization. Constructors will be introduced in Section 5.2.

- The class is declared as an extension of an existing Java class named JFrame. This technique is called inheritance, and is the principal means in object-oriented languages for constructing new software abstractions that are variations on existing data types. Inheritance will be introduced in Section 5.3 and will be more extensively studied beginning in Chapter 8.

- The output displayed in a window by this program is created using some of the graphics primitives provided by the Java run-time library. These graphics operators are explained in Section 5.4.
Ball Worlds

```java
import java.awt.*;
import javax.swing.JFrame;

public class BallWorld extends JFrame {
    // 3
    public static void main(String[] args) {
        // 4
        BallWorld world = new BallWorld(Color.red);
        world.show();
        // 5
        for (int i = 0; i < 1000; i++)
            world.run();
        System.exit(0); // halt program
    }

    public static final int FrameWidth = 600;  // 11
    public static final int FrameHeight = 400; // 12
    private Ball aBall = new Ball(new Point(50, 50), 20);  // 13

    private BallWorld (Color ballColor) {
        // 14
        setSize (FrameWidth, FrameHeight);
        setTitle("Ball World");  // 15
        // application specific initialization
        aBall.setColor (ballColor);
        aBall.setMotion (3.0, 6.0);  // 16
    }

    public void paint (Graphics g)  // 17
    { super.paint(g); aBall.paint(g); }  // 18

    public void run () {  // 19
        aBall.move();  // 20
        Point pos = aBall.location();  // 21
        if ((pos.x < aBall.radius()) ||
            (pos.x > FrameWidth - aBall.radius()))
            aBall.reflectVert();  // 22
        if ((pos.y < aBall.radius()) ||
            (pos.y > FrameHeight - aBall.radius()))
            aBall.reflectHorz();  // 23
        repaint();  // 24
        try {
            Thread.sleep(50);  // 25
        } catch (InterruptedException e) { System.exit(0); }  // 26
    }
}
```

Figure 5.2 Class description for Ball World.
DATA FIELDS

We have seen in the previous chapter (Section 4.4) how data fields can be declared within a class and how they can be initialized. The example program here includes features we have not seen in our previous programs in the three data fields declared on lines 11–13:

```java
public static final int FrameWidth = 600;  // 11
public static final int FrameHeight = 400; // 12
private Ball aBall = new Ball (new Point(50, 50), 20); // 13
```

Recall that the keyword public means that the variables being declared can be accessed (that is, used directly) anywhere in a Java program, while those that are declared as private can be used only within the bounds of the class description in which the declaration appears. Recall also that the keyword static means that there is one instance of the data field, shared by all instances of the class. The modifier keyword final means that this is the last time when an object is changed. It is here applied to a variable declaration; we will in later chapters see how the modifier can also be applied to a method definition.

Data fields that are declared static and final behave as constants, because they exist in only one place and cannot change value. The identifier of such a data field is sometimes called a symbolic name. Because they cannot be modified, there is less reason to encapsulate a static final variable by declaring it private. Such values are often made public, as shown here. The particular symbolic values being defined in this program represent the height and width of the window in which the application will eventually produce its output. Symbolic constants are useful in programs for a number of different reasons:

- By being defined in only one place, they make it easy to change subsequently, should circumstances require. For example, changing the height and or width of the window merely requires editing the file to change the values being used to initialize these symbolic constants, rather than hunting down all locations in the code where the quantities are used.
- When subsequently used elsewhere in the program, the symbolic name helps document the purpose of the constant values.

The third data field is declared as an instance of class Ball, which is the second class used in the creation of our example program. A Ball is an abstraction that represents a bouncing ball. It is represented by a colored circular disk that can move around the display surface. The class Ball will be described in Section 5.5.
5.2 Constructors

Constructors

As noted at the beginning of the chapter, one of the major topics of this chapter is the creation of new objects. This occurs in two places in the program shown in Figure 5.2. The first is in the main program, which creates an instance of the class BallWorld.

```
BallWorld world = new BallWorld (Color.red); // 5
```

The `new` operator is always used to create a new object. In this case, it is being used to create an instance of BallWorld, which (the next section shows) is the name given to the window in which the program will display its output. The `new` operator is followed by a class name, indicating the type of object being created. A parenthesized list then gives any arguments needed in the initialization of the object.

Object creation and object initialization are intimately tied in concept, and it is important that a programming language also bring these concepts together. Without support from the programming language, two types of errors can easily occur:

- An object is created, but it is used before it is initialized.
- An object is created and is initialized several times before it is used.

The language Java uses a concept called a constructor to guarantee that objects are placed into a proper initial state the moment they are created. A constructor bears a strong resemblance to a method; however, the name of the constructor matches the name of the class in which it appears, the constructor does not specify a return type, and the user will never (indeed, can never) execute a constructor except as part of creating a new object. Like a method a constructor can have arguments, and the body of the constructor consists of a sequence of statements. In our example program the body of the constructor is found on lines 14–21:

```
private BallWorld (Color ballColor) {
    // general application initialization
    setSize (FrameWidth, FrameHeight); // 15
    setTitle("Ball World"); // 16
    // application specific initialization
    aBall.setColor (ballColor); // 18
    aBall.setMotion (3.0, 6.0); // 19
} // 20
```

When an object is created (via the new operator), the first method invoked using the newly created object is the constructor method. The arguments passed to the constructor are the arguments supplied in the `new` expression.
In this particular case, the argument represents a color. The class Color is part of the Java run-time library. The value red is simply a constant (a value declared both as static and final) in the class description of Color.

```java
BallWorld world = new BallWorld (Color.red); // 5
```

The corresponding parameter value in the constructor method is named ballColor (see line 14). The constructor method must ensure that the instance of the class BallWorld is properly initialized. As noted earlier, the BallWorld represents the window in which the output will be displayed. The first two statements in the constructor set some of the attributes for this window; namely, the size and the title.

The declaration of aBall also uses two occurrences of the new operator to create and initialize a new ball. One is creating an instance of the class Point, which is a Java library abstraction for representing a two-dimensional coordinate. The arguments to the constructor will set the x and y values of the point. The second new operation will create an instance of the class Ball. Not only will memory for this object be created by the new statement, but also the arguments will be matched by a corresponding constructor in the class Ball, which will then be invoked to initialize the newly created ball:

```java
public class Ball { // a generic round colored object that moves

    public Ball (Point loc, int r) { // ball center and radius
        ...
    }
}
```

The complete class description for Ball is shown in Figure 5.3. Not all aspects of a Ball are set by the constructor. The final two statements in the constructor for BallWorld set the color of the ball, and set the direction of motion for the ball. These attributes will be discussed in more detail in Section 5.5.

### 5.2.1 Constructing the Application

It is perhaps helpful at this point to say a word about the role and positioning of the `main` method. As you learned in Chapter 4, this method is invoked when execution of the program begins. This is possible because it is declared as static and therefore exists before any objects of its class exist. Consequently, an object of class BallWorld need not already exist in order to invoke the `main()` method.

Because it exists, in a sense, outside of all objects of the class, the `main` procedure can be used to create an instance of the class. This accounts for what might seem the strange phenomenon of `main` being a method in the BallWorld class, but at the same time being used to create a BallWorld object.
5.2 Constructors

import java.awt.*;

public class Ball {
    public Ball (Point lc, int x) { loc = lc; rad = r; }

    protected Point loc; // position in window
    protected int rad; // radius
    protected double changeInX = 0.0;
    protected double changeInY = 0.0;
    protected Color color = Color.blue; // color of ball

    public void setColor (Color newColor) { color = newColor; }

    public void setMotion (double dx, double dy)
    { changeInX = dx; changeInY = dy; }

    public int radius () { return rad; }

    public Point location () { return loc; }

    public void reflectVert () { changeInX = - changeInX; }

    public void reflectHorz () { changeInY = - changeInY; }

    public void moveTo (int x, int y) { loc.move(x, y); }

    public void move ()
    { loc.translate((int) changeInX, (int) changeInY); }

    public void paint (Graphics g) {
        g.setColor (color);
        g.fillOval (loc.x-rad, loc.y-rad, 2*rad, 2*rad);
    }
}

Figure 5.3 Implementation of the class Ball.

Some developers believe that it is less confusing to place the main method for a program into a class of its own, and to use this additional class merely to create an instance of the application class (in this case BallWorld), and then send it an appropriate message to begin the application. The code for such a class might look as follows:
public class BallWorldProgram {

    public static void main(String[] args) {
        BallWorld world = new BallWorld(Color.red);
        world.show();
        for (int i = 0; i < 1000; i++)
            world.run();
        System.exit(0);
    }
}

A disadvantage of this approach is that it creates two classes which, on many platforms, must reside in two separate files. Since this is in many ways just as cumbersome as the approach we have taken, and in most cases results in a longer program, we have adopted the style of incorporating the main method into the application class itself.

INHERITANCE

The most important feature of this program is the use of inheritance (sometimes also called extension). As noted earlier, the ball world is a rectangular window in which the action of the program (the bouncing ball) is displayed. The code needed to display and manipulate a window in a modern graphical user interface is exceedingly complex, in part because of the fact that the user can indicate actions such as moving, resizing, or iconifying the window. As a consequence, recent languages attempt to provide a means of reusing existing code so that the programmer need only be concerned with those features of the application that distinguish the program from other window applications.

The programming language Java uses the class JFrame to represent a generic window. By saying that the class BallWorld extends the class JFrame, we indicate that our new class, BallWorld, is a type of frame, but a more specialized type with a single purpose. The class JFrame defines code to perform actions such as resizing the window, arranging for the window to be displayed on the workstation screen, and so on. By extending the class JFrame, our new class inherits this functionality, which means the abilities are made available to the new class, and do not need to be rewritten anew.

public class BallWorld extends JFrame {
    // 3

    By executing the example program, the reader can verify that the window exhibits the functionality we expect of graphical windows—the ability to move,

    2Users without access to the newer Swing library will use the base class Frame instead of JFrame. Other minor differences between the program that uses the original AWT library and the Swing version will be described in Appendix xx.
5.3 Inheritance

resize, and iconify, even though the program does not explicitly define any code to support these behaviors. (The reader might also note some expected behaviors that are not provided. For example, the handling of menu items and the close or quit box. In subsequent chapters we will describe how these features can be provided.)

We can observe the use of inheritance in the variety of methods that are invoked in our example program, but are not defined by the class BallWorld. These methods are instead inherited from the parent class JFrame, or from the classes that JFrame in turn inherits from. Two examples are the methods setSize and setTitle invoked in the BallWorld constructor. These methods set the dimensions (in pixels) and title value for the window, respectively.

```java
private BallWorld (Color ballColor) {
    // general application initialization
    setSize (FrameWidth, FrameHeight); // 16
    setTitle("Ball World"); // 17
    ...
} // 21
```

Another example is the method show, which is invoked in the static method main after the instance of BallWorld has been created. The show method arranges for the window to appear on the display, and then for the surface of the window to be drawn.

```java
public static void main (String [ ] args) { // 4
    BallWorld world = new BallWorld (Color.red); // 5
    world.show(); // 6
    for (int i = 0; i < 1000; i++) // 7
        world.run(); // 8
    System.exit(0); // halt program // 9
} // 10
```

After displaying the window, the program will execute the method run one thousand times before halting execution. A graphical application will not halt simply because the end of the main method is encountered, but must explicitly call the system method System.exit to terminate execution. The ++ operator is used to increment an integer variable. The statement

```java
i++
```

is equivalent to

```java
i = i + 1
```

We will describe the effect of the run method in the next section.
THE JAVA GRAPHICS MODEL

Graphics in Java is provided as part of the Abstract Windowing Toolkit (AWT). The Java AWT is an example of a software framework. The idea of a framework is to provide the structure of a program but no application-specific details. The overall control, the flow of execution, is provided by the framework and therefore does not need to be rewritten for each new program. Thus, the programmer does not “see” the majority of the program code.

This is illustrated by the actions that occur subsequent to the program issuing the show method that is inherited from the class JFrame. The window in which the action will take place is created, and the image of the window must be rendered (drawn on the screen). To do so, the show method invokes a method named paint, passing as argument a graphics object.

The programmer defines the appearance of the window by providing an implementation of the method paint. The graphics object passed as argument provides the ability to draw a host of items, such as lines and polygons as well as text. In our example program the paint method simply instructs the ball to paint itself. Before doing this, the method invokes the paint method in the parent class JFrame. The method in the parent class erases any previous contents of the window, so that we only concern ourselves with drawing the new image. The variable super is used to distinguish the parent method from the child method, as both have the same name, paint. (As an experiment, you can try removing the call to super, and notice that our ball movement then becomes a continuous blur).

```java
public void paint (Graphics g) // 22
{ super.paint(g); aBall.paint(g); } // 23
```

In later examples we will investigate more of the abilities of the graphics objects provided by the Java library. (Including how to avoid the annoying flicker that may on some platforms be observed as the animation progresses.)

The loop in the run method instructs the ball to move slightly, checks to see if the ball has struck the edge of the playing surface, reflects the ball if necessary, and finally requests that the window be repainted. Note that the method to do this is called repaint; an application does not call the paint method directly, but only indirectly by requesting a repaint.

```java
public void run () { // 24
    aBall.move(); // 25
    Point pos = aBall.location(); // 26
    if ((pos.x < aBall.radius()) || // 27
        (pos.x > FrameWidth - aBall.radius())) // 28
        aBall.reflectVert(); // 29
    if ((pos.y < aBall.radius()) || // 30
        (pos.y > FrameHeight - aBall.radius())) // 31
```
5.5 The Class Ball

```java
aball.reflectHoriz(); // 32
repaint(); // 33
try {
    Thread.sleep(50); // 35
} catch (InterruptedException e) { System.exit(0); } // 36
}```

Graphical operations, such as repainting the window, are relatively slow in comparison to Java execution. Therefore it is necessary to halt the program for a short period of time so that the graphics system can catch up. This is accomplished by using the command `Thread.sleep()`. The command `sleep` will suspend the program for a small period of time, measured in milliseconds. While suspended, the window will be repainted. Because this method can raise an exception, it must be surrounded by a `try` block that will catch the exception and deal with it. In this case, an exception might indicate the program was interrupted while sleeping. To handle this we will terminate the program.

**The Class Ball**

We will use a ball, that is, a round colored object that moves, in a number of our subsequent example programs. It is therefore useful to define the behavior of a Ball in a general fashion so that it can be used in a variety of ways. The description of class Ball is placed in its own file (Ball.java) and is linked together with the BallWorld class to create the executable program.

A Ball (Figure 5.3) maintains five data fields. The location of the ball is represented by a Point, a general purpose class provided in the Java run-time library. The radius of the ball is an integer amount. Two floating point values are used to represent the horizontal and vertical components of the direction of motion for the ball. Finally, the color of the ball is represented by an instance of class Color, a Java library class we have previously encountered.

These five data fields are declared as protected. This allows classes within the same package, as well as any subsequent child classes we might create to have access to the data fields, without exposing the data to modification by other objects. It is good practice to declare data fields protected, rather than private, even if you do not anticipate extending the class to make new classes.

The constructor for the class Ball records the location by creating a new instance of class Point. The constructor also provides default values for color (blue) and motion. As we have seen in our example program, these can be redefined by invoking the method setColor and setMotion.

A number of methods are used to access some of the attributes of a ball. Attributes that can be obtained in this fashion include the radius, and the position of the ball. Methods that allow access to a data field in a class are termed accessor methods. The use of accessor methods is strongly encouraged.
in preference to making the data fields themselves public, as an accessor
method permits the value to be read but not modified. This ensures that any
modification to a data field will be mediated by the proper method, such as
through the methods setMotion or moveTo.

The methods reflectVert and reflectHorz reflect the motion of a ball in a vertical
or a horizontal direction, respectively. We have seen already how these are used
to alter the direction of the ball.

The method move makes use of an operation provided by the class Point. A
point can have its position changed (translated) by giving an amount the point
should move in the two-dimensional surface.

Finally, the method paint uses two operations that are provided by the class
Graphics in the Java library. These are the methods to set the current color for
rendering graphics (setColor) and to display a painted oval at a given location
on the window (fillOval).

## Multiple Objects of the Same Class

Every instance of a class maintains its own internal data fields. We can illustrate
this by making variations on our sample program. The simplest change is to
modify the main routine to create two independent windows. Each window will
have a different ball, each window can be independently moved or resized.

```java
public static void main (String [] args) {
    // create first window with red ball
    BallWorld world = new BallWorld (Color.red);
    world.show();
    // now create a second window with yellow ball
    BallWorld world2 = new BallWorld (Color.yellow);
    world2.show();
    // now put them both in motion
    for (int i = 0; i < 1000; i++) {
        world.run();
        world2.run();
    }
    System.exit(0);
}
```

The reader should try making this change, and observe the result. (On some
platforms the two windows will be placed one on top of the other, and so it will
be necessary to move the topmost window in order to see the second window
underneath). Note how one window is bouncing a red ball and the second is
bouncing a yellow ball. This indicates that each instance of class BallWorld
must be maintaining its own Ball value, given that a ball cannot be both red
and yellow at the same time.
class MultiBallWorld extends JFrame {
    
    private static final int BallCollectionSize = 10;
    private Ball[] balls = new Ball[BallCollectionSize];

    private MultiBallWorld(Color ballColor) {
        // general application initialization
        setSize(FrameWidth, FrameHeight);
        setTitle("Ball World");
        // application specific initialization
        for (int i = 0; i < BallCollectionSize; i++) {
            balls[i] = new Ball(new Point(10, 15), 5);
            balls[i].setColor(ballColor);
            balls[i].setMotion(3.0+i, 6.0-i);
        }
    }

    public void paint(Graphics g) {
        super.paint(g);
        for (int i = 0; i < BallCollectionSize; i++)
            balls[i].paint(g);
    }

    public void run() {
        for (int i = 0; i < BallCollectionSize; i++) {
            balls[i].move();
            Point pos = balls[i].location();
            if ((pos.x < balls[i].radius()) ||
                (pos.x > FrameWidth - balls[i].radius()))
                balls[i].reflectVert();
            if ((pos.y < balls[i].radius()) ||
                (pos.y > FrameHeight - balls[i].radius()))
                balls[i].reflectHorz();
        }
        repaint();
    }
}

Figure 5.4 Class description for Multiple Ball World.
A second variation illustrates even more dramatically the independence of different objects, even when they derive from the same class. The class MultiBallWorld (Figure 5.4) is similar to our initial program except that it creates a collection of balls rather than just a single ball. Only the lines that have changed are included, and those that are elided are the same as the earlier program. The new program declares an array of Balls, rather than just a single ball. Note the syntax used to declare an array. As noted in the previous chapter, arrays in Java are different from arrays in most other languages. Even though the array is declared, space is still not set aside for the array elements. Instead, the array itself must be created (again with a new command):

```java
private Ball[] balls = new Ball[BallCollectionSize];
```

Note how the size of the array is specified by a symbolic constant, defined earlier in the program. Even then, however, the array elements cannot be accessed. Instead, each array element must be individually created, once more using a new operation:

```java
for (int i = 0; i < BallCollectionSize; i++) {
    balls[i] = new Ball(new Point(10, 15), 5);
    balls[i].setColor(ballColor);
    balls[i].setMotion(3.0+i, 6.0-i);
}
```

Each ball is created, then initialized with the given color, and set in motion. We have used the loop index variable `i` to change the direction of motion slightly, so that each ball will initially move in a different direction.

When the program is executed, ten different balls will be created. Each ball will maintain its own location and direction. Each ball will move, independently of all other balls. To paint the window each of the ten balls is asked to paint itself.

## Chapter Summary

The two major themes introduced in this chapter have been the creation and initialization of new objects using a combination of the operator `new` and a constructor, and the definition of new classes using inheritance to extend an existing class. Topics discussed in this chapter include the following:

- Data fields that are declared `final` cannot be subsequently redefined. A `static` and `final` value is the technique normally used to create a symbolic constant.
- New objects are always created using the operator `new`. 
Study Questions

- When a new object is created, the *constructor* for the class of the object is automatically invoked as part of the creation process. The constructor should guarantee the object is properly initialized.

- A constructor is a method that has the same name as the class in which it is defined.

- Any arguments used by the constructor must appear in the new statement that creates the corresponding object.

- Classes can be defined using *inheritance*. Such classes extend the functionality of an existing class. Any public or protected data fields or methods defined in the parent class become part of the new class.

- The class `JFrame` can be used to create simple Java windows. This class can be extended to define application-specific windows.

- The framework provided by the Java AWT displays a frame (a window) when the frame object is given the *message* `show`.

- To create the image shown in the window the programmer executes the inherited message `repaint`. The framework eventually responds by issuing the message `paint`. The programmer can redefine the `paint` method to produce application-specific pictures.

- The `paint` method is given as argument an instance of the library class `Graphics`. This object can be used to create a variety of graphical images.

- The class `Point` (used in our class `Ball`) is a library class that represents a two-dimensional point on the window surface. The class provides a large amount of useful functionality.

**CROSS REFERENCES**

We will use the `Ball` class in case studies in Chapters 6–8 and 20. The topic of inheritance is simple to explain but has many subtle points that can easily trap the unwary. We will examine inheritance in detail in Chapters 8 through 11. The AWT and Swing will be examined in more detail in Chapter 13.

**STUDY QUESTIONS**

1. How would you change the color of the ball in our example application to yellow?

2. How would you change the size of the application window to 500 by 300 pixels?
3. What does the modifier keyword final mean when applied in a data field declaration?

4. Why do symbolic constants make it easier to read and maintain programs?

5. What two actions are tied together by the concept of a constructor?

6. What types of error does the use of constructors prevent?

7. What does it mean to say that a new class inherits from an existing class?

8. What methods inherited from class JFrame are used in our example application?

9. What methods provided by our example program are invoked by the code inherited from class JFrame?

10. What abstraction does the Java library class Point represent?

11. What are some reasons that data fields should be declared as private or protected, and access provided only through public methods?

**EXERCISES**

1. The method Math.random returns a random floating-point value between 0 and 1.0. This value can be multiplied by a constant to yield a value from a larger range. If needed, assigning the result to an integer yields a random integer value. For example, the following will generate a random integer between 0 and 10:

   ```java
   int i = (int) (10 * Math.random());
   ```

   Using this method, modify the example program shown in Figure 5.2 so that the ball will initially move in a random direction.

2. Modify the MultiBallWorld class so that the colors of the various balls created are selected randomly from the values red, blue and yellow. (*Hint: Call Math.random() and test the resulting value for various ranges, selecting red if the value is in one range, blue if it is in another, and so on.*)

3. Modify the MultiBallWorld class so that it will produce balls of different radiiuses, as well as different colors.

4. Rather than testing whether or not a ball has hit the wall in our main program, we could have used inheritance to provide a specialized form of Ball. Create a class BoundedBall that inherits from class Ball. The constructor for this class should provide the height and width of the window, which should subsequently be maintained as data fields in the class. Rewrite the
move method so that if the ball moves outside the bounds, it automatically reflects its direction. Finally, rewrite the BallWorld class to use an instance of BoundedBall, rather than an ordinary Ball, and eliminate the bounds test in the main program.

5. Our Ball abstraction is not as simple as it could have been. Separate the Ball class into two separate classes. The first, the new class Ball, knows only a location, its size, and how to paint itself. The second class, MovableBall, extends the class Ball and adds all behavior related to motion, such as the data fields changeN and changeN, the methods setMotion, reflectVert, reflectHorz and move, and so on. Rewrite the MultiBallWorld to use instances of class MovableBall.

6. Modify the run method in the multi-ball world so that after it has advanced a ball a small distance it will check to see if it has run into any other balls. (You can do this by seeing if its center is less than 2 radius distance from the center of any other ball). If so, alter (reflect) the directions of both balls.