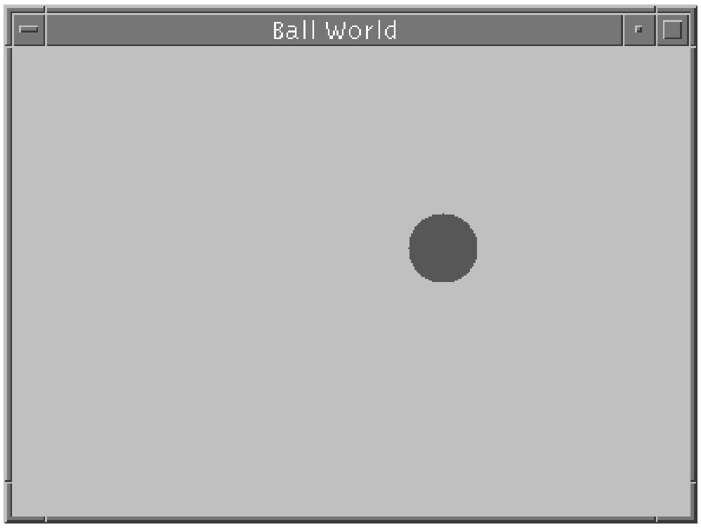
# Chapter 5

# Ball Worlds

In our intuitive description of object-oriented programming presented in Chapter 1, we described an object-oriented program 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 procedure 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.




Our second example program, or paradigm, is constructed out of two classes. The first of these appears in Figure 5.1. Again, we have added line numbers for the purposes of reference, however these are not part of the actual program.[1] 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 procedure named main, (lines 3 through 6), which is declared as static, void and public. Like all main programs, this procedure must take as argument an array of string values, which are, in this case, being ignored.

However, 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 fields, some of which are constant, some of which are initialized but not constant, and some of which are not initialized. These data fields will be described in detail in Section 5.1.
- The main program creates an instance of the class BallWorld. This object is initialized by means of a *constructor*. A constructor is a function 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 Frame. 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.

## 5.1 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 four data fields declared on lines 7 to 10:

```
public static final int FrameWidth = 600;                     // 7
public static final int FrameHeight = 400;                    // 8
private Ball aBall;                                           // 9
```

[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 function in a class.

```
import java.awt.*;

public class BallWorld extends Frame {

    public static void main (String [ ] args) {
        BallWorld world = new BallWorld (Color.red);
        world.show ();
    }

    public static final int FrameWidth = 600;
    public static final int FrameHeight = 400;
    private Ball aBall;
    private int counter = 0;

    private BallWorld (Color ballColor) {// constructor for new window
            // resize our frame, initialize title
        setSize (FrameWidth, FrameHeight);
        setTitle ("Ball World");

            // initialize aBall data field
        aBall = new Ball (10, 15, 5);
        aBall.setColor (ballColor);
        aBall.setMotion (3.0, 6.0);
    }

    public void paint (Graphics g) {
            // first, draw the ball
        aBall.paint (g);
            // then move it slightly
        aBall.move();
        if ((aBall.x() < 0) || (aBall.x() > FrameWidth))
            aBall.setMotion (-aBall.xMotion(), aBall.yMotion());
        if ((aBall.y() < 0) || (aBall.y() > FrameHeight))
            aBall.setMotion (aBall.xMotion(), -aBall.yMotion());
            // finally, redraw the frame
        counter = counter + 1;
        if (counter < 2000) repaint();
        else System.exit(0);
    }
}
```

Figure 5.1: Class Description for Ball World

```
private int counter = 0;                                    // 10
```

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 generally 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 function definition. When used with a variable declaration, the declaration must also include an initialization, as shown here.

Variables that are static and final are used to create symbolic names for constants, as they are variables that are guaranteed to exist only in one place, and not change values. Because they cannot be modified, there is less reason to encapsulate a static final variable by declaring it private. Thus, 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 subsequently change, 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 counter data field is an integer value, initialized to zero:


```
private int counter = 0;                                    // 10
```


Because the field is declared private we know it can be used only within the bounds of the class definition. Because it was not declared static, we know that each instance of the class will hold its own different value. Because it was not declared as final we know that the value being assigned is simply the initial value the variable will hold, but that it subsequently could be reassigned. We will see how this variable is used when we discuss the graphical aspects of the current program.

```
private Ball aBall;                                         // 9
```

The final 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 circle that can move around the display surface. The class Ball will be described in Section 5.5. How this field is initialized is described in the next section.

## 5.2 Constructors

As we noted at the beginning of the chapter, one of the major topics we address in this chapter is the creation of new objects. This occurs in two places in the program shown in Figure 5.1. The first is in the main program, which creates an instance of the class BallWorld.

```
BallWorld world = new BallWorld (Color.red);              // 4
```

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 (we will see in the next section) 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 initialized multiple 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 function, however the name of the function matches the name of the class in which it appears, the function does not specify a return type, and the user will never directly execute the constructor. However, like a function, a constructor can have arguments, and the body of the constructor consists of a sequence of statements. In our example program the constructor occurs in lines 11 to 19:

```
private BallWorld (Color ballColor) {// constructor for new window 11
        // resize our frame, initialize title                      12
    setSize (FrameWidth, FrameHeight);                           // 13
    setTitle ("Ball World");                                     // 14

        // initialize aBall data field                             15
    aBall = new Ball (10, 15, 5);                                // 16
    aBall.setColor (ballColor);                                  // 17
    aBall.setMotion (3.0, 6.0);                                  // 18
}                                                                // 19
```

When an object is created (via the new operator), the first function invoked using the newly created object is the constructor function. 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.

```
        BallWorld world = new BallWorld (Color.red);              // 4
```

The corresponding parameter value in the constructor function is named ballColor (see line 11). The constructor function must ensure that the instance of the class BallWorld is properly initialized. As we 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.

Line 17 of the constructor again uses the new operator to create and initialize a new object. In this case the object is an instance of the class Ball. Not only will memory for this object be created by the new statement, but the arguments will be matched by a corresponding constructor in the class Ball, which will then be invoked to initialize the newly created ball:

```
public class Ball {     // a generic round colored object that moves
    ...
    public Ball (int x, int y, int r) { // ball with given center and radius
        ...
    }
}
```

The complete class description for Ball will be shown in Figure 5.2 (page 84). 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.

## 5.3 Inheritance

The most important feature of this program is the use of *inheritance* (sometimes also called *extension*). As we 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; due in part to 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 Frame to represent a generic window. By saying that the class BallWorld extends the class Frame, we indicate that our new class, the ball world, is a type of frame, but a more specialized type with a single purpose. The class Frame 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 Frame, 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 Frame {                          // 2
```

By executing the example program the reader can verify that the window exhibits the functionality we expect of graphical windows; the ability to move, 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 a later chapter 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 functions are instead inherited from the *parent class* Frame. Two examples are the functions setSize and setTitle invoked in the BallWorld constructor. These functions set the dimensions (in pixels) and title value for the window, respectively.

```
    private BallWorld (Color ballColor) {// constructor for new window 11
            // resize our frame, initialize title                   12
        setSize (FrameWidth, FrameHeight);                      // 13
        setTitle ("Ball World");                                // 14
            ...
    }                                                           // 19
```

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

```
    public static void main (String [ ] args) {                 // 3
        BallWorld world = new BallWorld (Color.red);            // 4
        world.show ();                                          // 5
    }                                                           // 6
```

## 5.4 The Java Graphics Model

Graphics in Java is provided as part of the AWT, or the *Abstract Windowing Toolkit*. 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 Frame. The window in which the action will take place is created, and the image of the window must be rendered. To do so, the show method invokes a function named paint, passing as argument a *graphics object*.

The programmer defines the appearance of the window by providing an implementation of the function 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 we use the paint procedure for two purposes. The only image in the window itself is the bouncing ball. The image of the ball is produced by invoking the paint method in the class Ball (see Figure 5.2). The second purpose of the paint method is to provide a simple means of updating the location for the ball. The ball is moved slightly, checking to see if the resulting new location is outside the bounds of the window. If so, the direction of the ball is reversed. Finally, invoking the repaint method (also inherited from Frame) indicates to the framework that the window should be redrawn, and the cycle continues.[2] A counter is used to prevent the program from running indefinitely, invoking the function System.exit after a certain number of iterations. (Later programs will use other techniques to halt the program).

```
public void paint (Graphics g) {
        // first, draw the ball
    aBall.paint (g);
        // then move it slightly
    aBall.move();
    if ((aBall.x() < 0) || (aBall.x() > FrameWidth))
        aBall.setMotion (-aBall.xMotion(), aBall.yMotion());
    if ((aBall.y() < 0) || (aBall.y() > FrameHeight))
        aBall.setMotion (aBall.xMotion(), -aBall.yMotion());
        // finally, redraw the frame
    counter = counter + 1;
    if (counter < 2000) repaint();
    else System.exit(0);
```

[2]Some readers might object that the control of the animation has little to do with the rendering of the image on the window, and thus does not belong in the paint routine. While there is merit to this argument, this is also the simplest way to make simple animations. In later chapters we will present more robust ways to control animations.

```
}                                                       // 33
```

Note that the programmer calls the inherited method named repaint, which in turn will eventually result in the paint method being invoked. The programmer does not directly call the paint method for the class.

In later examples we will investigate more of the abilities of the graphics objects provided by the Java library.

## 5.5 The class Ball

We will use a ball, that is, 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 5.2) maintains four data fields. The location of the ball is represented by a Rectangle, a general purpose class provided in the Java run-time library. Two floating point values 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 four data fields are declared as protected. This allows 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 Rectangle. Note that the three integer arguments passed as arguments to the constructor represent the center location of the ball and the radius: a simple calculation is used to convert these to the corner of the rectangle and the extent. 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 functions setColor and setMotion.

A number of functions are used to access some of the attributes of a ball. Attributes that can be obtained in this fashion include the radius, the x and y coordinate of the center of the ball, the horizontal and vertical directions of motion, and the region occupied by the ball. Functions that allow access to a data field in a class are termed *accessor functions*. The use of accessor functions is strongly encouraged in preference to making the data fields themselves public, as an accessor function only permits the value to be *read*, and not modified. This ensures that any modification to a data field will be mediated by the proper function, such as through the function setMotion or moveTo.

Some of the functions use operations provided by the class Rectangle. A rectangle can provide a width (used in function radius), the location of the upper corner (used in functions

```
public class Ball {     // a generic round colored object that moves
    protected Rectangle location;     // position on graphic surface
    protected double dx, dy; // x and y components of motion vector
    protected Color color;     // color of ball

    public Ball (int x, int y, int r) { // ball with given center and radius
        location = new Rectangle(x-r, y-r, 2*r, 2*r);
        dx = 0; dy = 0; // initially no motion
        color = Color.blue;
    }

        // functions that set attributes
    public void setColor (Color newColor) { color = newColor; }

    public void setMotion (double ndx, double ndy) { dx = ndx; dy = ndy; }

        // functions that access attributes of ball
    public int radius () { return location.width / 2; }

    public int x () { return location.x + radius(); }

    public int y () { return location.y + radius(); }

    public double xMotion () { return dx; }

    public double yMotion () { return dy; }

    public Rectangle region () { return location; }

        // functions that change attributes of ball
    public void moveTo (int x, int y) { location.setLocation (x, y); }

    public void move () { location.translate ((int) dx, (int) dy); }

    public void paint (Graphics g) {
        g.setColor (color);
        g.fillOval
            (location.x, location.y, location.width, location.height);
    }
}
```

Figure 5.2: Implementation of the class Ball

x and y), can move to a new position (used in function moveTo), and can transliterate on the 2-dimensional surface (used in the function move).

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

## 5.6 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.

```
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();
}
```

The reader should try making this change, and observe the result. 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, as a ball cannot be both red and yellow at the same time.

A second variation illustrates even more dramatically the independence of different objects, even when they derive from the same class. The class MultiBallWorld (Figure 5.3) is similar to our initial program, only 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 we 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):

```
ballArray = new Ball [ BallArraySize ];
```

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:

```
public class MultiBallWorld extends Frame {

    ...
    private Ball [ ] ballArray;
    private static final int BallArraySize = 10;

    private MultiBallWorld (Color ballColor) {
        ...
            // initialize object data field
        ballArray = new Ball [ BallArraySize ];
        for (int i = 0; i < BallArraySize; i++) {
            ballArray[i] = new Ball(10, 15, 5);
            ballArray[i].setColor (ballColor);
            ballArray[i].setMotion (3.0+i, 6.0-i);
            }
        }

    public void paint (Graphics g) {
        for (int i = 0; i < BallArraySize; i++) {
            ballArray[i].paint (g);
                // then move it slightly
            ballArray[i].move();
            if ((ballArray[i].x() < 0) ||
                    (ballArray[i].x() > FrameWidth))
                ballArray[i].setMotion
                    (-ballArray[i].xMotion(), ballArray[i].yMotion());
            if ((ballArray[i].y() < 0) ||
                    (ballArray[i].y() > FrameHeight))
                ballArray[i].setMotion
                    (ballArray[i].xMotion(), -ballArray[i].yMotion());
        ...
            }
    }
}
```

Figure 5.3: Class description for Multiple Ball World

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

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

When executed, ten different balls will be created. Each ball will maintain its own location and direction. As each ball is asked to paint it will display its value on the window. Each ball will then move, independently of all other balls.

## 5.7 Chapter Summary

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

- Data fields which 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.
- 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 function 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 functions defined in the parent class become part of the new class.
- The class Frame 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 message paint is used. The programmer can define this 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 Rectangle (used in our class Ball) is a library class that represents a rectangular region on the two-dimensional window surface. The class provides a large amount of useful functionality.
- Multiple instances of the same class each maintain their own separate data fields. This was illustrated by creating multiple independent Ball objects, which move independently of each other.

## Cross References

We will use the Ball class in case studies in Chapters 6, 7, 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 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 errors 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 Frame are used in our example application?
9. What methods provided by our example program are invoked by the code inherited from class Frame?
10. What abstraction does the Java library class Rectangle represent?
11. What are some reasons that data fields should be declared as private or protected, and access provided only through member functions?
12. In Java, what are the steps involved in creating an array of objects?

## Exercises

1. The function Math.random returns a random floating point value between 0 and 1.0. Using this function, modify the example program shown in Figure 5.1 so that the ball will initially move in a random direction.

2. Modify the MultiBallWorld 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 so that it will produce balls of different radiuses, 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 function so that if the ball moves outside the bounds, it automatically reverses 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 dx and dy, the functions setMotion and move, and so on. Rewrite the MultiBallWorld to use instances of class MovableBall.