Pinball Game Construction Kit

In this chapter we will expand on the techniques introduced in the Cannon Game of Chapter 6, creating an entirely different type of interactive application. Along the way, we will use the development of this program to discuss standard data structures, event handling, inheritance, exceptions, and interfaces.

The application we will develop, the Pinball Construction Kit, simulates a pinball game. Users can fire a ball from the small square in the right corner of the bottom of the screen. (See Figure 7.1.) Balls rise, then encounter a variety of different types of targets as they fall back to the ground. The user scores points that depend upon the type and number of targets encountered as the ball descends. The objective of the game is to score as many points as possible.

As we did with the Cannon World application in Chapter 6, we will develop this program in a sequence of stages, each stage emphasizing a small number of new programming concepts. The intent is simply to introduce these concepts in the context of a relatively simple and easy to understand program. Many of the major ideas introduced here will be discussed in more detail in later chapters.

**FIRST VERSION OF GAME**

Our first version (Figure 7.2) is in many ways the same as the Cannon World application from Chapter 6, but with a number of new features. Because we will later be creating objects that will need to communicate with the window object, the variable **world** is declared as a public static value, rather than as a local variable to the main method. Balls are “fired” from a square box labelled with a red disk that appears at the bottom right corner of the window. A notable difference between this version and the Cannon Game application of Chapter 6 is the mechanism for placing a new ball into motion. Whereas in the earlier version firing was tied to a button, in this version we will trap mouse
activities in a more general routine. One more notable change is that the pinball game allows several balls to be in the air at once (the Cannon Game fired only a single ball).

The class PinBallFire, shown in Figure 7.3, is the fire button for the application. It knows how to draw itself. It also can test a point to see if it is in the region of the fire button, and return a new ball.

The class PinBall (Figure 7.4) extends from the class class CannonBall described in Chapter 6 (Section 6.1.3). Differences are that here the initial direction is slightly to the left of vertical, and includes a small random number perturbation so as to be less predictable. Also, a PinBall can return an instance of the standard class Rectangle that represents the bounding rectangle for the ball. We will use the latter feature to help detect when a ball has hit a target.

7.1.1 Collection Classes
Unlike the Cannon Game described in the previous chapter, the pinball game allows several balls to be moving at one time. Every time the user clicks on the "fire" button, a new ball is placed in motion, even if earlier balls have not yet ceased moving.
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;
import javax.swing.JFrame;

public class PinBallGame extends JFrame {

    public static void main (String [ ] args) {
        PinBallGame world = new PinBallGame();
        world.show();
        world.run();
    }

    public PinBallGame () {
        setTitle("Pin Ball Construction Kit");
        setSize(FrameWidth, FrameHeight);
        addWindowListener(new CloseQuit());
        balls = new Vector();
        addMouseListener(new MouseKeeper());
    }

    public static final int FrameWidth = 400;
    public static final int FrameHeight = 400;
    private Vector balls;
    private PinBallFire fireButton =
        new PinBallFire(new Point(FrameWidth-40, FrameHeight-40));

    private void run () { ... }

    private class MouseKeeper extends MouseAdapter {
    }

    public void paint (Graphics g) {
        super.paint(g); // clear window
        fireButton.paint(g); // draw target
        for (int i = 0; i < balls.size(); i++) { // draw balls
            Ball aBall = (Ball) balls.elementAt(i);
            aBall.paint(g);
        }
    }

    Figure 7.2 First version of PinBallGame class.
class PinBallFire {
    public PinBallFire (Point where) { location = where; }

    private Point location;

    public void paint(Graphics g) {
        g.setColor(Color.white);
        g.fillRect(location.x, location.y, 30, 30);
        g.setColor(Color.red);
        g.fillOval(location.x, location.y, 30, 30);
    }

    public boolean includes (int x, int y)
    { return (x > location.x) && (y > location.y); }

    public Ball fire (int x, int y)
    { return new PinBall(new Point(x, y)); }
}

Figure 7.3 The class PinBallFire.

class PinBall extends CannonBall {
    public PinBall (Point loc)
    { super(loc, 8, -2 + Math.random(), -15); }

    public Rectangle box() {
        int r = radius();
        return new Rectangle(location().x-r, location().y-r, 2*r, 2*r);
    }
}

Figure 7.4 The class PinBall.

To manage this, we need a data structure that can hold a collection of values. The one collection data structure we have seen up to now is the array (see Chapter 5). However, the array is limited by the fact that when we allocate a new array object we must state the number of elements the array will hold. In the present case, we cannot make any such estimate since we do not know how many times the user will hit the "fire" button.

Fortunately, the Java library provides a number of other data structures we can employ. One of the simplest is a Vector. A vector is, like an array, an indexed data structure; meaning that each element has a position in the collection, and elements are accessed by requesting the value at a given position. However,
7.1 First Version of Game

unlike an array, a vector can dynamically grow as new values are inserted into the collection.

To use a vector, the programmer must first import the vector class definition from the standard library:

```
import java.util.Vector;
```

The vector is declared by simply providing a name:

```
private Vector balls;
```

Note, in particular, that unlike an array, it is not necessary to state the type of values that a Vector will hold. For technical reasons having to do with their internal structure, a vector is restricted to holding only objects. Thus, for example, one cannot create a vector of integer values (ints) but one can create a vector of instances of class Integer. This is one reason for the existence of "wrapper" classes, such as Integer and Float.

Just as an array in Java separates the declaration of the array name and the allocation of space for the array, the space for a Vector must be similarly created and assigned. In our example program this occurs in the constructor for the class PinBallGame. Note that no fixed limit is set for the space:

```
balls = new Vector();
```

Although we have not seen it yet, a new element will be inserted into the vector by the method `addElement`:

```
balls.addElement (newBall);
```

The number of values stored in a Vector can be determined by invoking the method `size`.

```
for (int i = 0; i < balls.size(); i++)
```

Finally, values are accessed in a vector using the method `elementAt`. Like an array, the set of legal index values ranges from zero to one less than the number of elements in the collection. The compiler only knows that the accessed element is a value of type object; it must be cast to the appropriate type before it can be used. Here we cast the value into the type PinBall. A run-time check is performed to ensure that the conversion is actually valid:

```
PinBall aBall = (PinBall) balls.elementAt (i);
```

The reader should note how the `paint` method in Figure 7.2 cycles over the collection of `balls`, asking each to paint itself.
7.1.2 Mouse Listeners

As noted in Chapter 6, the Java event model is based around the concept of listeners; objects that wait and "listen" for an event to take place, and then respond appropriately. The earlier examples showed how to create a listener by defining a class that implements the corresponding interface for the event in question.

Mouse events are treated in a similar fashion; however, there are five different mouse-related events that could potentially be of interest. Thus, the interface for a MouseListener has the following structure:

```java
public interface MouseListener {
    public void mouseClicked (MouseEvent e);
    public void mouseEntered (MouseEvent e);
    public void mouseExited (MouseEvent e);
    public void mousePressed (MouseEvent e);
    public void mouseReleased (MouseEvent e);
}
```

Often a programmer is interested in only one or two of these five events. However, to implement an interface the Java language insists that the programmer provide a definition for all operations. To simplify such cases, the Java library provides a simple class named MouseAdapter. The class MouseAdapter implements the MouseListener interface but uses an empty method for each method. That is, a MouseAdapter does nothing in response to any mouse event. However, the programmer can write a new class that inherits from MouseAdapter, and overrides (or redefines) the methods of interest.

That is what we do in the example program. An inner class defines a MouseListener by extending MouseAdapter. An instance of this class is created, and passed as an argument to the method addMouseListener, which is inherited from class JFrame.

```java
addMouseListener(new MouseKeeper());
```

The class MouseKeeper inherits all five methods defined by MouseAdapter and redefines only one. The other four messages will be handled by the methods inherited from the parent class, which will do nothing.

```java
private class MouseKeeper extends MouseAdapter {
    public void mousePressed (MouseEvent e) {
        int x = e.getX(); // get coordinates where
        int y = e.getY(); // mouse was pressed
        // only handle mouse event in the fire region
        if (fireButton.includes(x, y))
            balls.addElement(fireButton.fire(x, y));
    }
```
7.2 Adding Targets: Inheritance and Interfaces

The argument passed to each method in the MouseListener interface is a value of type MouseEvent. The mouse event encodes certain information relating to the type of event that occurred. In our case, the most important information is the position (or coordinate) of the mouse at the moment the button was pressed. This information can be derived from the mouse event object using the methods getX and getY. With these values, a new ball is created, added to the list of balls, and placed into motion.

7.1.3 Running the Application

Once the initial window has been displayed, the heart of the application is found in the method run and the other methods it calls. The run method repeatedly calls moveBalls to move the balls, then repaints the window, sleeping for a short period in order to permit the window to be redrawn.

```java
private void run () {
    while (true) {
        moveBalls();
        repaint();
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) { System.exit(0); }
    }
}
```

The moveBalls routine cycles over the list of balls, moving each one:

```java
private moveBalls () {
    for (int i = 0; i < balls.size(); i++) {
        Ball theBall = (Ball) balls.elementAt(i);  
        if (theBall.location().y < FrameHeight) 
            theBall.move();
    }
}
```

**Adding Targets: Inheritance and Interfaces**

To provide realism and interest to our pinball game, we need to add targets for the ball to encounter on its way down the playing surface. As in real pinball games, we will want to include a variety of different types of targets. Some targets simply add values to the score, some move the ball in a new direction,
and some swallow the ball, removing it from play. In order to simplify the program, we will want to maintain all the different types of targets in a single data structure, a vector.

7.2.1 The Pinball Target Interface

Because we want to process targets uniformly, for example in a loop that asks whether a ball has hit any target, we need all the targets to have a uniform interface. However, the various different types of targets will be represented internally by different data structures. Thus, we do not want to use inheritance, such as we have been doing with the different forms of Ball up to this point. Inheritance is a mechanism for sharing structure; a PinBall, for example, is simply a type of Ball that has the same structure and behavior as a Ball, adding a few new features, but maintaining all the characteristics of the original. There is little in the way of common structure between a Peg (a target that when hit by a ball scores a number of points and moves the ball in a new direction) and a Wall (a target that when struck simply reflects the motion of the ball).

What is needed in this case is the ability to state that the two concepts (Peg and Wall, in this case), share the same behavior, although they have nothing in common in structure. As we saw in our earlier case study, in Java this is accomplished by describing the common behavior as an interface, and declaring that both objects implement the same interface. An interface for our pinball targets can be described as follows:

```java
interface PinBallTarget {
    public boolean intersects (PinBall aBall);
    public void moveTo (int x, int y);
    public void paint (Graphics g);
    public void hitBy (PinBall aBall);
}
```

The interface in this case is declaring that there are four characteristics of interest in a pinball target. Each target can tell if it has been hit by a ball; that is, if it intersects the region occupied by a ball. Each target can be moved to a specific point on the playing surface, each can paint itself, and each provides some response when it has been hit by a given ball. However, the means by which each of these behaviors will be achieved is left unspecified, and different targets are free to implement the interface in different fashions.

An examination of a few different targets will help illustrate the point. Our first type of target will be a Spring. When hit by a falling ball, a Spring rebounds the ball back into the playing area, moving it upward where we hope it will encounter further targets. A spring is represented graphically by a small horizontal box, and a series of zigzag lines, as in Figure 7.5.

The class description for Spring is shown in Figure 7.6. Note how the class Spring must explicitly state that it implements the PinBallTarget interface, and
must provide a specific meaning for each of the four elements of that interface. In this case, we will state that a spring intersects a ball if the rectangle surrounding the ball intersects with the rectangle representing the spring platform. When the spring is hit by the ball, it reverses the vertical direction of movement for the ball.

One slight element of interest has been added to the drawing of the Spring object. We have provided two different graphical representations for the Spring object, selected by an integer variable named state. Normally, a spring will be held in state 1. When struck, the value of state is changed to 2, and the next time the spring is redrawn it will present an alternative image, one in which the spring has been elongated. Drawing this second image changes the state back to state 1, and a subsequent redraw will display the original. The effect is a simple form of animation, where a moving spring will appear to stretch momentarily, then return to a ready state.

A second type of target is a Wall. A Wall (Figure 7.7) is a narrow rectangular region. A ball intersects a wall if their regions overlap, and if so the ball is simply reflected back, in effect bouncing off the wall. The bounce is obtained by reversing either the horizontal or vertical component of the direction of motion, depending upon which wall has been hit.

The advantage of declaring these two different structures as both implementing an interface is that an interface name can be used as a type. That is, we can declare a variable as holding a value of type PinBallTarget. In much the same way that a variable declared as maintaining a value of type Ball could, in fact, be holding a cannonball, a pinball, or any other value derived from a class that extends the original class Ball, a variable declared as maintaining a PinBallTarget could, in fact, be holding either a Spring, a Wall, or any of the other varieties of target to be described subsequently. (This is one aspect of polymorphism, a topic we will return to in more detail in Chapter 12.) We will shortly make use of this property, by storing all the targets in our game in a single data structure, and testing the motion of the ball against the location of each target in turn.

Consider now a third type of target, a Hole. A Hole (Figure 7.8) consumes any ball it comes in contact with, removing the ball from the playing surface. A Hole is represented by a circular colored image, just like a ball. A Hole has a location on the playing surface, just like a Ball. In fact, because a hole is structurally
class Spring implements PinBallTarget {
    public Spring (int x, int y)
    { pad = new Rectangle(x, y, 30, 3); }

    private Rectangle pad;
    private int state = 1;

    public void moveTo (int x, int y)
    { pad.setLocation(x, y); }

    public void paint (Graphics g) {
        int x = pad.x; int y = pad.y;
        g.setColor(Color.black);
        if (state == 1) { // draw compressed spring
            g.fillRect(x, y, pad.width, pad.height);
            g.drawLine(x, y+3, x+30, y+6);
            g.drawLine(x+30, y+6, x, y+7);
            g.drawLine(x, y+7, x+30, y+9);
            g.drawLine(x+30, y+9, x, y+11);
        }
        else { // draw extended spring
            g.fillRect(x, y-8, pad.width, pad.height);
            g.drawLine(x, y+6, x+30, y-1);
            g.drawLine(x+30, y-1, x, y+3);
            g.drawLine(x, y+3, x+30, y+7);
            g.drawLine(x+30, y+7, x, y+11);
            state = 1;
        }
    }

    public boolean intersects (Ball aBall)
    { return pad.intersects(aBall.box()); }

    public void hitBy (Ball aBall) {
        aBall.reflectHorz();
        state = 2;
    }
}

Figure 7.6 Definition for class Spring.
7.2 Adding Targets: Inheritance and Interfaces

```
class Wall implements PinBallTarget {
    public Wall (int x, int y, int width, int height) {
        location = new Rectangle(x, y, width, height);
    }

    public Rectangle location;

    public void moveTo (int x, int y) {
        location.setLocation(x, y);
    }

    public void paint (Graphics g) {
        g.setColor(Color.black);
        g.fillRect(location.x, location.y,
                    location.width, location.height);
    }

    public boolean intersects (Ball aBall) {
        return location.intersects(aBall.box());
    }

    public void hitBy (Ball aBall) {
        Point ballPos = aBall.location;
        if ((ballPos.y < location.y) ||
            (ballPos.y > (location.y + location.height)))
            aBall.reflectVert();
        else
            aBall.reflectHorz();
    }
}
```

Figure 7.7 Definition of class Wall.

similar to a Ball, we can use inheritance to simplify the implementation of the Hole abstraction.¹

This illustrates the important difference between the use of inheritance and the use of interfaces. The mechanism of inheritance should be used when two (or more) concepts have a structural relationship. Note that with objects, a structural relationship almost always implies at least some behavioral relationship. In contrast, the interface mechanism should be used with two (or more) concepts having a behavioral relationship but no structural relationship. We will explore these ideas in more detail in Chapter 8.

¹ Whether the structural similarity of Ball and Hole is sufficient grounds for the use of inheritance is a debatable point we will return to in Chapter 8.
class Hole extends Ball implements PinBallTarget {

    public Hole (int x, int y) {
        super(new Point(x, y), 12);
        setColor(Color.black);
    }

    public boolean intersects (Ball aBall) {
        int dx = aBall.location().x - location().x;
        int dy = aBall.location().y - location().y;
        int r = 2 * radius();
        return (-r < dx) && (dx < r) && (-r < dy) && (dy < r);
    }

    public void hitBy (Ball aBall) {
        // move ball totally off frame
        aBall.moveTo(0, PinBallGame.FrameHeight + 30);
        // stop motion of ball
        aBall.setMotion(0, 0);
    }
}

Figure 7.8 Definition of class Hole.

Note how a Hole uses both inheritance and an interface. The hole inherits much of its behavior from the class Ball, including the methods paint and moveTo. The Hole declares that it implements the PinBallTarget interface, and to do so must provide a method to see if the hole has intersected with a ball, and the actions to be performed when such an event occurs. In the case of a hole, the ball is moved clear off the playing surface, and motion of the ball is halted.

A class that inherits from an existing class that implements an interface must of necessity also implement the interface. We will use this property in defining the next two types of targets in our pinball game. A ScorePad is, like a hole, represented by a circular region. When struck by a ball, the score pad has no effect on the ball (the ball simply moves over it); however, the score pad adds a certain amount to the player’s score. The particular amount to add is defined as part of the state for the score object.

Note how the ScorePad class (Figure 7.9) inherits the intersects behavior from class Hole and the moveTo behavior from class Ball, but overrides the paint and hitBy methods that would otherwise be inherited from class Hole. The first now draws a colored circle with the scoring amount in the middle, while the latter adds the given value to the player score.
class ScorePad extends Hole {
    public ScorePad (int x, int y, int v) {
        super(x, y);
        value = v;
        setColor(Color.red);
    }

    protected int value;

    public void hitBy (BallаБall) {
        PinBallGame.world.addScore(value);
    }

    public void paint (Graphics g) {
        g.setColor(color);
        int r = radius();
        g.drawOval(location().x-r, location().y-r, 2*r, 2*r);
        String s = "" + value;
        g.drawString(s, location().x-7, location().y+1);
    }
}

Figure 7.9 Definition of the class ScorePad.

public class PinBallGame extends JFrame {

    public PinBallGame () {
        :
        getContentPane().add("North", scoreLabel);
    }

    private int score = 0;
    private Label scoreLabel = new Label("Score = 0");

    public void addScore (int v) {
        score = score + v;
        scoreLabel.setText("Score = " + score);
    }

    :
}

Figure 7.10 Adding labels to our pinball game.
Because a ScorePad inherits from class Hole, which implements the PinBallTarget interface, the class ScorePad is also said to implement the interface. This means, for example, that a ScorePad could be assigned to a variable that was declared to be a PinBallTarget.

### 7.2.2 Adding a Label to Our Pinball Game

In the revised version of our program we will add a new graphical element, a textual label in a banner across the top of the window. This is accomplished by declaring a new Label, and adding it in the “North” part of the window (Figure 7.10). As the user scores new points, the text of the label is updated. Note that a ScorePad refers back to the application object through the variable world.

A Peg is similar to a ScorePad, but it sticks up above the playing surface. Thus, when a Peg is struck, it deflects the ball off in a new direction, depending upon the angle of the ball and the point at which it encounters the peg. (The algorithm used in Figure 7.11 is not exactly correct as far as actual physics is concerned, but it does have the advantage of being easy to compute.) The ball is then updated until it no longer intersects with the peg, thereby avoiding having the method executed multiple times for a single encounter. We have once again added a simple animation to the class Peg, so that the first time a peg is redrawn after it has been struck, the circle surrounding the peg will appear to enlarge and then return to a normal size.

To create the second version of our game (Figure 7.12), we simply create a Vector of targets, along with the vector of balls. We initialize the targets in the constructor for the game, including placing walls on the sides and top of the playing area, to reflect wayward balls. The user fires balls as before, which then proceed to interact with the various targets as the balls move down the playing surface. Each time a ball moves, a loop is executed to determine if the new location of the ball has struck a target. If so, the target is informed, and the location of the ball potentially updated. Finally, the entire screen is repainted, which involves repainting both the targets and the collection of balls.

### Pinball Game Construction Kit: Mouse Events Reconsidered

Although the second version of the pinball game is certainly more interesting than the first, it is still limited by the fact that the layout of the various targets is determined by the original programmer. To create a different layout, the program must be changed and then recompiled and executed. In our final version, we will show how this limitation can be overcome, by providing a pallet of target elements from which the user can select, dynamically constructing the pinball game while the program is executing.
In appearance, our revised game will move the playing area slightly to the right, placing a sequence of potential target components along a strip in the far left. (See Figure 7.13.) The user can click the mouse down in one of these alternatives, then slide the mouse (still down) over into the playing area. When the user releases the mouse, the selected target element will be installed into the new location.

The effect is produced by overriding both the mousePressed and the mouseRelesed methods inherited from the mouse adapter (Figure 7.14). The two methods communicate with each other by means of a variable named element. The mousePressed method creates a potential target, determined by the coordinates of the point at which the mouse goes down. Note that we have not eliminated

class Peg extends ScorePad {

    public Peg (int x, int y, int v)
    { super(x, y, v); }

    private int state = 1;

    public void paint (Graphics g) {
        super.paint(g);
        int r = radius();
        if (state == 2) { // draw expanded circle
            g.drawOval(location().x-(r+3), location().y-(r+3),
                        2*(r+3), 2*(r+3));
            state = 1;
        } else
            g.drawOval(location().x-(r+2), location().y-(r+2),
                        2*(r+2), 2*(r+2));
    }

    public void hitBy (Ball aBall) {
        super.hitBy(aBall); // update the score
        aBall.reflectVert();
        aBall.reflectHorz();
        while (intersects(aBall)) // move out of range
            aBall.move();
        state = 2; // next draw will expand circle
    }
}

Figure 7.11  Definition of the class Peg.
public class PinBallGame extends JFrame {

    private Vector targets;

    public PinBallGame () {
        
        // create the targets
        targets = new Vector();
        targets.addElement(new Wall(30, 50, 2, 380));
        targets.addElement(new Wall(30, 50, 360, 2));
        targets.addElement(new Wall(390, 50, 2, 380));
        targets.addElement(new Hole(100, 100));
        targets.addElement(new ScorePad(150, 220, 100));
        targets.addElement(new Peg(300, 140, 200));
        targets.addElement(new Spring(120, 350));
    }

    public void moveBalls () {
        for (int i = 0; i < balls.size(); i++) {
            PinBall theBall = (PinBall) balls.elementAt(i);
            if (theBall.location().y < FrameHeight) {
                theBall.move();
                // see if we ran into anything
                for (int j = 0; j < targets.size(); j++) {
                    PinballTarget target =
                        (PinballTarget) targets.elementAt(j);
                    if (target.intersects(theBall)) target.hitBy(theBall);
                }
            }
        }
    }

    public void paint (Graphics g) {
        
        for (int j = 0; j < targets.size(); j++) { // draw targets
            PinballTarget target = (PinballTarget) targets.elementAt(j);
            target.paint(g);
        }
    }

    Figure 7.12 Addition of targets to the class PinBallGame.
the original use of the mousePressed method, simply added a new condition. The MouseReleased method checks the location of the release, and if it is in the playing area and if a target item was previously selected (both conditions must be true), then it adds a new target to the game.

Other changes needed to provide our final version of the Pinball Construction Kit simply involve repositioning the left wall, and drawing the images of the selection pallet.

**Chapter Summary**

In this chapter the development of an example program has once again served as a vehicle to introduce a number of features of the Java programming language. Introduced in this chapter were the following features:

- The use of collection classes, in particular the collection class Vector. We will discuss collection classes in more detail in Chapter 19.
private class MouseKeeper extends MouseAdapter {

    private PinBallTarget element;

    public void mousePressed (MouseEvent e) {
        element = null;
        int x = e.getX();
        int y = e.getY();
        if (firebutton.includes(x, y))
            balls.addElement(fireButton.fire(x, y));
        if (x < 40) { // each target occupies a 40 by 40 pixel box
            switch (y / 40) {
                case 2: element = new Hole(0, 0); break;
                case 3: element = new Peg(0, 0, 100); break;
                case 4: element = new Peg(0, 0, 200); break;
                case 5: element = new ScorePad(0, 0, 100); break;
                case 6: element = new ScorePad(0, 0, 200); break;
                case 7: element = new Spring(0, 0); break;
                case 8: element = new Wall(0, 0, 2, 15); break;
            }
        }
    }

    public void mouseReleased (MouseEvent e) {
        int x = e.getX(); // only perform release action if mouse is
        int y = e.getY(); // released on playing surface
        if ((element != null) && (x > 50)) {
            element.moveTo(x, y);
            targets.addElement(element);
            repaint();
        }
    }
}

Figure 7.14 Capturing both mouse presses and releases

- An expanded discussion of the Java listener event model (started in Chapter 6), focusing on how to create objects that will listen for mouse events.
- Our first example of a statement that could potentially produce an exception, and the way the Java language permits the programmer to specify what actions to take when an exception occurs.
Study Questions

- More on interfaces, contrasting the use of the interface mechanism with
  the use of inheritance.
- One aspect of the important concept of polymorphism. A variable declared
  as an instance of a parent class (such as Ball) can, in fact, be holding a
  value derived from a child class (such as PinBall). Similarly, a variable
  declared as an interface value (such as PinBallTarget) can, in fact, hold
  any object that implements that interface (such as Peg). This property
  allows us to create arrays of different objects, such as an array of pinball
  targets, and process them in a uniform fashion.

CROSS REFERENCES

The distinction between interfaces and inheritances is explored in more detail
in Chapter 8. Collection classes will be investigated in detail in Chapter 19.
Chapter 13 presents a more systematic investigation of the services provided
by the AWT.

STUDY QUESTIONS

1. Why must the variable world be declared static?
2. In what ways is a Vector object similar to an array? In what ways is it
different?
3. What method is used to determine the number of elements held in a
   Vector? What method is used to access the values? What method is used
to insert a new value into the collection?
4. What is the relationship between MouseAdapter and MouseListener? In what
   ways are they different?
5. What is an exception?
6. What action is performed by the method System.exit? Under what circum-
   stances in our program will this method be called?
7. When should two software components be tied together through the use
   of inheritance rather than a common interface?
8. What type of objects can be held by a variable declared using the interface
   PinBallTarget?
9. In what ways does the class Hole modify the behavior inherited from class
   Ball?
10. What is a Label? How is a label attached to a window? What methods
    are used to change the text of a label?
EXERCISES

1. The class Peg inherits from ScorePad, which in turn inherits from Hole, which in turn inherits from Ball. For each of these classes, describe all the methods defined in the class or inherited from parent classes, and for each of the latter indicate in which parent class the method definition occurs.

2. The pinball game as presented allows the user an unlimited number of balls. Change the program to fire only a fixed number of balls, disallowing firing once the supply is exhausted. Change the display at the top of the screen so that it will indicate the number of remaining balls, as well as the score.

3. Add a "reset" button to the bottom of the screen. When pressed, the reset button sets the score back to zero and, if you implemented the suggestion in the previous question, resets the number of balls in play.

4. On some platforms it may be difficult to halt the PinBall application once it has finished. Add a button labeled "Quit" to the bottom (south) part of the application window. When pressed, this button should execute the method System.exit(0).

5. In the final program, the items on the pallet are still stored in the targets vector, so that they will be checked for a hit, even though they can never be hit by a ball. A better solution would have been to create a new vector pallet that will hold these items, redrawing both the pallet and the targets on a repaint, but only if a target in the targets vector is hit by a ball. Modify the program in this fashion.

6. Currently balls do not test to see if they intersect with other balls. We could support this modification by making PinBall implement the PinBallTarget interface, and adding balls to the list of targets as well as the list of balls. Describe what changes would need to be added to modify the program in this fashion.

7. Another change could allow the programmer to reposition items even after they have been placed in the playing area. If a mouse click occurs on the playing surface over a target, select the target and move it to the location given by the associated mouse up. Be careful that you don't end up placing the element in the target vector twice.

8. Create a program that opens a window, listens for mouse clicks, and when a mouse is released will display the distance (in pixel units) between the location the mouse was pressed and the location it was released.

9. Write a program that places a red circle in the middle of the window. The circle should change color to blue when the mouse enters the window, then return to red when the mouse leaves the window. When the mouse
is clicked inside the window, the circle should change color to green and remain green for 1000 milliseconds, before returning to blue. Finally, if the mouse is clicked within the bounds of the circle and released outside the circle, the circle should be moved so as to be centered on the location of the mouse release.

10. Develop a “paddle” target object. When the user clicks the mouse over the paddle, the paddle should move back and forth (perhaps only once). If a paddle encounters a ball, the ball is reflected off the paddle.