A Cannon Game

In this chapter we will examine an implementation of a classic "shooting cannon" game. In this simple game there is a cannon on the left portion of the user’s window, and a target on the right portion, as shown in Figure 6.1. The user can control the angle of elevation for the cannon and fire a cannonball. The objective of the game is, of course, to hit the target.

As with all the case studies, our objective is not so much the cannon application itself, which is only moderately interesting, but the use of a simple program to illustrate a number of different features of the Java language. In particular, in this chapter we will examine the features of the Java library that simplify the creation of a graphical user interface (GUI). We will develop two variations on this game:

- The first version is the simplest. The user enters the angle of the cannon from the command argument line, the cannon fires once, and the program halts.

Figure 6.1  A window of the Cannon Game.
In the second version, we improve user interaction, by providing both a scroll bar with which the angle of the cannon can be changed, and a button to fire the cannon. By manipulating these, multiple attempts to hit the target can be made during one execution of the program.

**The Simple Cannon Game**

The principal class for our cannon application is shown in Figure 6.2. The main method is similar to those described in the previous chapters. The universe for our application is termed CannonGame. An instance of this class is created, then passed the message show. The message show causes the window in which the application is played to be displayed. The method run then repeatedly moves the ball and redraws the window.

In our first version of the Cannon Game, the angle for the cannon is read from the command-line argument. The first string in the command-line argument list is assumed to be an integer value, representing the angle (in degrees) for the cannon to be set prior to launch of the cannonball. This value is converted by a method from the Java library class Integer, and passed as argument to the constructor for the cannon game. There, the method intValue() is used to convert an integer into an int. We will return to a discussion of the relationship between these two data types in Section 6.1.4.

The cannon game is declared to be a type of JFrame, which you will recall from the preceding chapter is how Java declares a new type of window. The class description defines two public constant values (declared as static final, see Section 5.1) that describe the height and width of the window that represents the cannon application. In addition, three new data fields are declared. These are the cannon ball (an instance of the class Ball we described in the previous chapter), a message that will be displayed in the middle of the playing window, and the cannon. The latter is an instance of class Cannon, which will be described shortly.

The constructor for the class CannonGame resizes the window frame to the declared bounds, and sets the window title. The argument value is converted from the type Integer to the built-in type int using the method intValue. The string representing the message is set to a value that will, when printed, indicate the current angle of the cannon. Finally, the cannonball is created using the angle to determine the initial direction of movement.

There are two other methods defined in this class. The method moveCannonBall will move the cannon ball slightly and see if the target has been hit, updating the message string appropriately if so. The method run cycles over a loop that moves the cannon ball, redraws the window, then sleeps while the slower graphics operations are being performed.
import java.awt.*;
import javax.swing.JFrame;

public class CannonGame extends JFrame {
    public static void main (String [ ] args) {
        CannonGame world = new CannonGame (Integer.valueOf(args[0]));
        world.show(); while (true) world.run();
    }

    public CannonGame (Integer theta) {
        setSize (FrameWidth, FrameHeight); setTitle ("Cannon Game");
        cannon.setAngle(theta.intValue());
        message = "Angle = " + theta;
        aBall = cannon.fire();
    }

    public static final int FrameWidth = 600;
    public static final int FrameHeight = 400;
    private Cannon cannon = new Cannon(new Point(20, FrameHeight-10));
    private CannonTarget target =
        new CannonTarget(new Point(FrameWidth-100, FrameHeight-12));
    private Ball aBall = null;
    private String message;

    public void paint (Graphics g) { ... }

    private void moveCannonBall () {
        aBall.move();
        if (aBall.location().y > FrameHeight) {
            if (target.hitTarget(aBall.location().x))
                message = "You Hit It!";
            else message = "Missed";
            aBall = null;
        }
    }

    private void run () {
        if (aBall != null) moveCannonBall();
        repaint();
        try { Thread.sleep(50); } catch (Exception e) { System.exit(0); }
    }
}

Figure 6.2 Description of the principal class for the Cannon Game.
The paint method for the cannon game simply paints the window, the cannon, the cannon ball (if there is one), the target, and the message in the middle of the screen:

```java
public void paint (Graphics g) {
    super.paint(g);
    cannon.paint(g);
    target.paint(g);
    if (aBall != null)
        aBall.paint(g);
    g.drawString(message, FrameWidth/2, FrameHeight/2);
}
```

The call on `super.paint(g)` is necessary to first erase the screen. Without it each image is drawn on top of the previous, leaving the cannon ball looking like a large ink stain running across the page.

### 6.1.1 The Target

The target is a simple object that has only two responsibilities. It must draw itself (as a rounded rectangle) and it must determine if a horizontal value is within the bounds of the target:

```java
public class CannonTarget {
    public CannonTarget (Point loc) { location = loc; }

    private Point location;
    private static final int width = 50;

    public boolean hitTarget (int x) {
        // see if target has been hit
        return (x > location.x) && (x < location.x + width);
    }

    public void paint (Graphics g) {
        g.setColor (Color.red);
        g.fillRoundRect(location.x, location.y, width, 10, 6, 6);
    }
}
```

### 6.1.2 The Cannon

The most complicated graphics occurs in the class `Cannon`, which manipulates the angle of the cannon and the firing of cannon balls. This class is shown in Figure 6.3.
6.1 The Simple Cannon Game

```java
import java.awt.*;

class Cannon {
    public Cannon (Point location) { loc = location; setAngle(45); }

    private final Point loc;
    private int angle;
    private double radianAngle;

    public void setAngle (int a) {
        angle = a;
        radianAngle = angle * Math.PI / 180.0;
    }

    public Ball fire () {
        Ball cannonBall = new CannonBall(new Point(loc), 10,
        15 * Math.cos(radianAngle),
        -12 * Math.sin(radianAngle));
        return cannonBall;
    }

    public void paint (Graphics g) {
        int barrelLength = 40;
        int barrelWidth = 8;
        int wheelRadius = 12;
        // draw wheel
        g.drawOval (loc.x - wheelRadius, loc.y - wheelRadius,
        2 * wheelRadius, 2 * wheelRadius);
        // draw barrel
        int lv = (int) (barrelLength * Math.sin(radianAngle));
        int lh = (int) (barrelLength * Math.cos(radianAngle));
        int sh = (int) (barrelWidth * Math.sin(radianAngle));
        int sv = (int) (barrelWidth * Math.cos(radianAngle));
        g.drawLine(loc.x-sh, loc.y-sv, loc.x+sh, loc.y+sv);
        g.drawLine(loc.x-sh, loc.y-sv, loc.x+lh-sh, loc.y-lv-sv);
        g.drawLine(loc.x+sh, loc.y+sv, loc.x+lh+sh, loc.y+lv+sv);
        g.drawLine(loc.x+lh-sh, loc.y-lv-sv, loc.x+lh+sh, loc.y-lv+sv);
    }
}
```

**Figure 6.3** The cannon used in the Cannon Game application.
Rendering pictures is complicated by the fact that Java, like almost all windowing systems, uses a coordinate system that is "upside down." As shown in Figure 6.4, the upper left corner of a window is the 0,0 coordinate, with \( x \) values increasing as they move right, and \( y \) values increasing as they move down. However, most people prefer to think in a coordinate system where the 0,0 location is the bottom left corner, and \( y \) values increase as they move up as depicted in Figure 6.5.

This upside down geometry should be kept in mind when reading the graphics code in Figure 6.3. To draw the cannon we first draw the circle that represents the wheel. A small amount of trigonometry is used to determine the lines that will be used to draw the barrel of the cannon. To move upwards a value is subtracted from the current \( y \) location, while to move to the right a value is added to the \( x \) location.
The data fields in the cannon game have all been declared private. The location (here representing the center of the wheel) is set by the constructor. The angle is set by the method setAngle. Since the methods Math.sin and Math.cos use radians instead of degrees, a conversion is performed. The method fire creates a new cannonball that is headed in the appropriate direction.

6.1.3 Balls That Respond to Gravity

The class Ball was introduced in Chapter 5. A Ball, you will recall, possessed a radius, a location, a color, and a direction. The latter was represented by a pair of values, representing the extent of motion in the x coordinate and the extent of motion in the y coordinate.

A CannonBall is built using inheritance as an extension of class Ball. This means that a CannonBall has all the properties of a Ball and also includes new properties or alters existing properties. In this case, a CannonBall changes the move method, simulating the effect of gravity by reducing the change in the vertical direction by a small amount each update cycle.

```java
import java.awt.Point;

public class CannonBall extends Ball {
    public CannonBall (Point loc, int r, double dx, double dy) {
        super(loc, r);
        setMotion (dx, dy);
    }

    public double GravityEffect = 0.3;

    public void move () {
        changeInY += GravityEffect;
        super.move(); // update the ball position
    }
}
```

We have already encountered the pseudovariable super, shown here in both methods, but have not yet provided an explanation of this construct. When a method overrides (that is, replaces) a similarly named method in the parent class, a technique must be provided to indicate that the method should invoke the method inherited from the parent class. The method cannot simply be named, as the name in the child class and the parent class are the same. Thus, for example, if one was to try to invoke the move method in class Ball by simply executing the method move, the result would be an infinite series of recursive method calls; which is not the outcome we wish.

The pseudovariable super is used to represent the receiver but is viewed as an instance of the parent class, not the current class. Thus, by invoking
super.move(), the method move is asking that the move method from the parent
class be executed, and not the version overridden by the class CannonBall.
Similarly, the call on super.paint() in the paint method is instructing that the
paint method inherited from JFrame be executed.

In a similar fashion, the use of the name super as a method in a constructor
is used to indicate that the constructor for the parent class should be invoked,
using the arguments shown. Note how the class CannonBall is invoked with a
slightly different set of arguments than were used for the class Ball.

6.1.4 Integers and ints
As noted in Section 4.3 of Chapter 4, integer values are not actually objects, in
the technical sense of the world. Thus, integers do not have classes, there are no
methods associated with integers, and so on. For each of the nonclass primitive
types, the Java library defines a “wrapper” class that can be used as an object.
For integers this wrapper class is named Integer. The class Integer provides a
number of useful operations. The one we utilize in our example program is the
ability to take a String (the command-line argument), and convert it into
a numeric value. To do this, a string is passed as argument to the message
valueOf:

```java
CannonGame world = new CannonGame (Integer.valueOf(args[0]));
```

The result produced by the message valueOf is a new instance of class Integer,
initialized to the indicated value. The conversion from Integer to int is performed
using the message intValue:

```java
cannon.setAngle(theta.intValue());
```

## Adding User Interaction

The user interaction in our first application was very primitive. The user could
select one angle, run the program, and see the result. In our second variation,
we improve user interaction by providing the ability to dynamically set the
angle of the cannon, and fire several times during one execution.

In the language of graphical user interfaces, buttons are used to signal
actions, scroll bars are used to set a variable quantity, checkboxes or radio
buttons are used to select alternatives, and text boxes are used to enter textual
information. In our revised game we will incorporate two of these features,
placing a button on the top of the screen, and a scroll bar to the right of the
playing area, as shown in Figure 6.6.

The program for the revised game, now named CannonWorld, is shown in
Figure 6.7. One feature to note is that we must now import the Java libraries
java.awt.event* and java.swing.event.*, in order to include the definitions of the
6.2 Adding User Interaction

![Figure 6.6](image)

A screen of the revised game, renamed CannonWorld.

Event-handling routines for the Java system. This is in addition to the `java.awt.*` and `javax.swing.*` libraries we have been including from the start. Although in code length the amount of change from the first version of our program is relatively small, a number of important features distinguish this program. We will explore these in the following sections.

6.2.1 Listeners

User interaction in Java is based on a model termed *event-driven* execution. An event-driven program will wait for an event, and then respond to it. In Java, the technique used to wait for events is termed a *listener*. A listener is simply an object whose sole responsibility is to wait for a specific event. When the event occurs, the listener wakes up, and performs whatever action is necessary.

Our first listener is a useful tool we will employ in many subsequent programs. You may have noticed that it was difficult to halt our first version of the program. Halting a running program in inherently platform-specific. On Unix systems you can type Control-C to force a halt. On other platforms you may or may not have this ability. We can get around this problem and provide a platform independent way to halt our program by defining a listener and attaching it to the close-box that is part of any window.

Clicking the mouse in the close box for a window will generate a window event. To trap this condition, we need a listener specialized for window events. The Java method `windowClosing` defined as part of the standard library class `WindowAdapter` defines the behavior we want. By using inheritance to subclass

---

1 It is possible to rewrite this program to use the older AWT library, and avoid the newer Swing components. See appendix xx.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class CannonWorld extends JFrame {
    public static void main (String [ ] args) {
        CannonWorld world = new CannonWorld();
        world.show(); while (true) world.run();
    }

    public CannonWorld () {
        ...
        addWindowListener(new CloseQuit());
        // add graphical objects
        JButton fire = new JButton("fire");
        fire.addActionListener(new FireButtonListener());
        getContentPane().add("North", fire);
        scrollbar.addAdjustmentListener(new ScrollBarListener());
        getContentPane().add("East", scrollbar);
    }

    private JScrollBar scrollbar =
        new JScrollBar(JScrollBar.VERTICAL, 45, 5, 0, 90);

    public void paint (Graphics g) { ... }
    private void moveCannonBall () { ... }
    private void run () { ... }

    private class FireButtonListener implements ActionListener {
        public void actionPerformed (ActionEvent evt) {
            aBall = cannon.fire();
        }
    }

    private class ScrollBarListener implements AdjustmentListener {
        public void adjustmentValueChanged (AdjustmentEvent e) {
            int angle = scrollbar.getValue();
            cannon.setAngle(angle);
            message = "Angle: " + angle;
            repaint();
        }
    }
}

Figure 6.7 Revised program renamed CannonWorld with button and scroll bar.
from WindowAdapter we can attach whatever actions we want to this method. In
the present program, we simply want to halt the application. Thus, we create
the following simple class:

```java
import java.awt.event.*;

public class CloseQuit extends WindowAdapter {
    public void windowClosing (WindowEvent e) {
        System.exit(0);
    }
}
```

The argument to the method windowClosing contains information about the
mouse-click event, for example the location of the mouse-click action. In the
class CloseQuit we simply ignore this additional information.

Having defined the class CloseQuit, we next must register the listener with
the window. This is accomplished by the method addWindowListener:

```java
addWindowListener(new CloseQuit());
```

Now to stop the program the user can simply click the close box, and the
application will immediately halt.

### 6.2.2 Inner Classes

One of the more notable features of the program in Figure 6.7 is the declaration
of two new classes within the application class itself. A class declared in such
a fashion is known as an inner class. Modifiers used in declaring an inner class
match the meanings we have previously described for data fields and methods;
thus, an inner class that is declared as private (such as shown here) can only be
used within the outer class in which it is defined.

Inner classes are allowed to access their surrounding environment. That is,
methods in an inner class can use data fields declared in the surrounding outer
class, as well as invoke methods from the surrounding context. We see this in
the method defined in class ScrollBarListener, which is allowed access to both
the data fields cannon and message.

### 6.2.3 Interfaces

Both the inner classes created in this example use the keyword implements
in their header. The implementation of an interface is yet another program-
structuring mechanism, one that can be understood by comparison to the
technique of inheritance we have been using previously.

An interface in Java is a description of behavior. It is written in a fashion
that appears similar to a class definition; however, there are no implementations
(method bodies) associated with methods, and the only data fields permitted must be declared static. An interface for ActionListener, used in our sample program, might be written as follows:

```java
interface ActionListener {
    public void actionPerformed (ActionEvent);
}
```

The Java library, particularly in those sections that relate to the handling of events (such as pressing buttons or moving scroll bars), makes extensive use of interfaces. When a class is declared as implementing an interface, it is a guarantee (one that is checked by the compiler) that the class must provide a certain behavior. In this case, an assertion that a class is an ActionListener means that the class must provide a method named actionPerformed.

The reader should consider carefully the difference between the implementation of an interface and the extension of a class by means of inheritance. Using inheritance, methods and data fields that are declared in the parent class may be used in the child class. Thus, the child class inherits the structure of the parent, as well as being able to mimic the parent behavior. Using an interface, on the other hand, there is no implementation of the methods in the "parent interface" at all. Instead, the parent simply defines the names of the methods, which must then be implemented in the child class. In this manner a child class inherits a specification of methods from an interface, but no structure, no data fields, and no member functions.

Despite the fact that methods of an interface have no implementation, an interface can be used as a type name in an argument declared in a method header. The matching parameter value must then be an instance of a class that implements the interface. In our example program, the addActionListener method in class JButton expects an argument that implements the ActionListener interface, and the addAdjustmentListener method in class JScrollBar expects an argument that implements the AdjustmentListener interface.

### 6.2.4 The Java Event Model

As we noted earlier, modern graphical user interfaces are structured around the concept of events. An event is an action, such as the user clicking the mouse on a button, selecting a menu item, pressing a key, or inserting a disk into a drive. The program responds to an event by performing certain actions. Thus, such interfaces are said to be event-driven. For each type of event, there is an associated listener. When the event occurs, the listener goes into action and performs its assigned behavior.

Our sample program uses several listeners. One listener, an instance of CloseQuit, waits for the user to click in the close box. Another is waiting for the fire button to be pressed. The class JButton is one of many graphical elements provided by the Java Swing library. The string argument passed to
the constructor for the class is the text that will appear on the face of the button. A listener for a button must implement the interface ActionListener. The listener in our sample program is declared as an instance of the inner class FireButtonListener, and is attached to a newly created button by the following code:

```java
    JButton fire = new JButton("Fire");
    fire.addActionListener(new FireButtonListener());
```

When the button is pressed, the message actionPerformed will be passed to the listener. In this case, this message will be handled by the one method in the class body:

```java
    private class FireButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            aBall = cannon.fire();
        }
    }
```

The argument of type ActionEvent which is passed to the method actionPerformed describes details of the event with more precision. However, the value is ignored, since in this case there is only one event a button can perform that is of any interest, namely the event that occurs when it is pressed. In this case we simply fire the cannon, which will return a new Ball that is then put into motion.

The scroll bar that is used to control the elevation of the cannon is created in a similar fashion. The constructor for class ScrollBar takes as argument an indication whether the scroll bar is horizontal or vertical, an initial value, and the range of accepted values. The scroll bar itself is created as part of the declaration of the variable named scrollbar, while the listener is added as part of the constructor for the game:

```java
    scrollbar.addAdjustmentListener(new ScrollBarListener());
```

The listener must implement the AdjustmentListener interface. When the scroll bar is moved, the listener is given the message adjustmentValueChanged. Note that, unlike the situation involving the button, the scroll bar must be made available to the listener, so that the current value of the scroll bar can be determined (using the message getValue). It is for this reason that the scroll bar is saved in a data field, but the button need not be. Once the value of the scroll bar has been determined, the angle of the cannon can be changed, and the window scheduled for repainting.

```java
    private class ScrollBarListener implements AdjustmentListener {
        public void adjustmentValueChanged (AdjustmentEvent e) {
```
int angle = scrollbar.getValue();
cannon.setAngle(angle);
message = "Angle: " + angle;
repaint();
}

6.2.5 Window Layout

Part of every Java program is a layout manager. The layout manager controls the placement of graphical items in a Java program. By using sophisticated layout managers the programmer can have a great deal of control over the appearance of a Java window. For simple programs, however, we can use the default layout manager, BorderLayout, which permits values to be placed on the four sides of the screen. These four portions of the screen are identified as North (the top), East (the right), West (the left), and South (the bottom).

The current layout manager is accessed indirectly by first getting hold of the window pane, and then using add to insert an object into the pane.\footnote{The method getContentPane is specific to the swing graphics library. Users of the older AWT library should see Appendix xx.} The constructor for our application places the button on the top of the window, and the scroll bar on the right hand side.

public CannonWorld () {
    :
    :
    // add graphical objects
    JButton fire = new JButton("fire");
    fire.addActionListener(new FireButtonListener());
    getContentPane().add("North", fire);
    scrollbar.addAdjustmentListener(new ScrollBarListener());
    getContentPane().add("East", scrollbar);
}

In later chapters we will explore a variety of other layout managers.

CHAPTER SUMMARY

This chapter has once again made use of a relatively simple application as a vehicle to introduce a number of new concepts. The following list summarizes some of the ideas introduced in this chapter:
6.3 Chapter Summary

- The class `Integer`, which is a wrapper class that can hold an integer value. Instances of this class are objects, unlike normal integer values, which are not objects. The class `Integer` provides some useful functionality, such as the ability to parse a string value that holds the textual representation of an integer quantity.

- The use of inheritance in the construction of the class `CannonBall`, so that the majority of the behavior for the ball is inherited from the parent class `Ball`.

- The pseudovariable `super`, which when used as a value inside a method refers to the parent class from which the current class inherits.

- The idea of an `inner class`, which is a class definition nested within another class definition. Inner classes are allowed full access to the data values and methods provided by surrounding classes.

- The idea of an `interface`, which is a means to ensure that classes satisfy certain behavior. An interface defines the names and arguments for member functions but does not provide an implementation. A class that declares itself as implementing an interface must then provide an implementation for these operations. A method can insist that an argument implement certain functionality, by declaring the argument using the interface as a type.

- The Java event model, in which `listener` objects are attached to event producing objects, such as buttons. When an event occurs, the listener is notified and then performs whatever action is appropriate.

- The graphical component classes `JButton` and `JScrollBar`, which simplify the creation of graphical features in a user interface.

- The idea of a window layout manager. In our application program we used the default layout manager, which is an instance of the class `BorderLayout`.

Note that as our application has become more complex, we have moved closer to the idea that an object-oriented programming is a "community" of agents that interact to produce the desired behavior. Instances of following categories of objects are all used in this example program:

- The class `CannonWorld`, which inherits from the class `JFrame` provided by the Java library.

- The class `CannonBall`, built as an extension of the earlier class `Ball` developed in Chapter 5. Note that each instance of `Ball` also includes a `Point` object.

- A `Cannon` object.
The instance of CannonTarget, which also maintains an instance of Point.

- The class Integer, used here for its ability to translate a number in text into a numeric quantity.

- The instance of class CloseQuit that waits for the user to click in the close box.

- The graphical component classes JButton and JScrollBar, and their listener classes FireButtonListener and ScrollBarListener, the latter two constructed as inner classes within our application class.

- Instances of the class ActionEvent and AdjustmentEvent, which are created when an event occurs and carry information to the event listener.

- The layout manager, an instance of class BorderLayout.

CROSS REFERENCES

Wrapper classes, such as Integer and Double, will be explained in more detail in Chapter 19. The distinction between inheritance and implementation, and the uses of each, will be a topic addressed in Chapter 10. Window layouts, layout managers, graphical components, and other features of the AWT will be examined in more detail in Chapter 13.

STUDY QUESTIONS

1. What is the parent class for class CannonGame?
2. In the first cannon game, how is the angle of the cannon determined?
3. What is the difference in behavior between a Ball and a CannonBall?
4. How is the pseudovariable super used in a method? What effect does it have in a message expression?
5. What is the difference between the types Integer and int?
6. What is an inner class?
7. What is an interface?
8. What would an interface for the class CannonBall look like?
9. What does it mean to say that a class implements an interface?
10. What does it mean to say that a program is event-driven?
11. What is an event listener?
12. What is a window layout manager?
Exercises

13. In Figure 6.7, although both the fire button and the scroll bar are graphical components, one is declared as a member data field and the other a local variable. Explain why. Could they both have been declared as member data fields? As local variables?

EXERCISES

1. Change the CannonGame so that the message being displayed provides not only the angle of the cannon, but also the current position of the cannonball. This value should change as the cannonball moves through the air.

2. Modify the class CannonBall so that the ball is colored blue when the ball is moving upward, and colored red when the ball is descending. Will this change have any impact on the rest of the program?

3. Modify the CannonWorld program so that it maintains both a count of the number of balls fired and the number of times the target was hit. Display both of these values in the message area, as well as the angle of the cannon.

4. 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. Create a simple program that draws a window with a colored ball in the center. Place a button at the top of the window. When the user presses the button, the color of the ball will change from red to yellow, or from yellow to green, or from green to red.

6. Create a simple program that draws a window with a colored ball in the center. Place a scroll bar on the right side of the window. As the user moves the scroll bar, move the ball up and down the window.

7. The constructor for class Color can also take three integer arguments, which represent the saturation values for red, blue, and green. The arguments must be integers between 0 and 255. Create a simple program that draws a window with a colored ball in the center. Place scroll bars on three sides. As the user moves the scroll bars, the saturation values change for one of these arguments, and the ball changes color.