M301: Software Systems & their Development

Block 1: Introduction to Java

Unit 2: Basic Constructs in Java

Aims of the unit:

- The aim of this unit is to continue your introduction to Java. In particular, you will learn how to:
- create new objects and send messages to them;
- develop applications using windows;
- use and understand inheritance;
- use Java's event model.

Block 1: Introduction to Java

Unit 2: Basic Constructs in Java

Section 1: Developing A Simple Java Program using JBuilder

This section is intended as an introduction to Section 2. All the Java constructs you meet here will be revisited in Section 2.

Objectives:

On completing this section you should be able to:

- create a simple object using a constructor;
- create and display a window frame on your computer screen;
- paint a message in a window;
- use data fields to store simple values;
- pause the execution of a program using the sleep method from the class `Thread`;
- refer to a superclass using the reserved word `super`;
- build a simple animation using a loop.

IMPORTANT: Read Section 2.1 from Block 1 and do all the practical activities

Block 1: Introduction to Java

Unit 2: Basic Constructs in Java

Section 2: Ball Worlds: Swing Classes, Constructors and Inheritance

Objectives:

On completing this section you should be able to:

- create a small application that uses the Abstract Windowing Toolkit (AWT) and Swing packages to simulate movement in a window based on the Java graphics model;
- develop Java programs with several classes using inheritance;
- define and use object constructors;
- define constants in Java;
- represent a simple Java program as a class diagram;
- construct a Java program by accessing code held in files.

Summary of Chapter 5

An object-oriented program can be described as a universe of interacting agents (or objects).

The program discussed in this chapter places a graphical window on the user's screen, draws a ball that bounces around the window for a few moments, and then halts.

The program consists of two classes. The first one is shown below.

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

public class BallWorld extends JFrame {

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

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

        private BallWorld (Color ballColor) {
                setSize (FrameWidth, FrameHeight);
                setTitle ("Ball World");

                aBall.setColor (ballColor);
                aBall.setMotion (3.0, 6.0);
        }

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

        Public void run() {
                aBall.move();
                Point pos = aBall.location();

                if ((pos.x < aBall.radius()) ||
                (pos.x > FrameWidth – aBall.radius()))
                        aBall.reflectHorz();
                if ((pos.y < aBall.radius()) ||
                (pos.y > FrameHeight – aBall.radius()))
                        aBall.reflectVert();

                repaint();

                try {
                        Thread.sleep(50);
                } catch (InterruptedException e) {System.exit(0);}
        }
}
```

Data Fields

In the above program there are three data fields

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

Recall the following:

public means that the variables being declared can be accessed (i.e. used directly) anywhere in a java program.

private can be used only within the bounds of the class description in which the declaration appears.

static means that there is only one instance of the data field shared by all the instances of the class.

Data fields that are declared static and final behave as *constants*, because they exists in only one place and cannot change value. The identifier of such a data field is sometimes called a *symbolic* name.

Since constants cannot be modified they are often made public.

Symbolic constants are useful in programs for the following reasons:

- By being defined in only one place, they make it easy to change subsequently, should circumstances require.

- When used elsewhere in the program, the symbolic name helps document the purpose of the constant values.

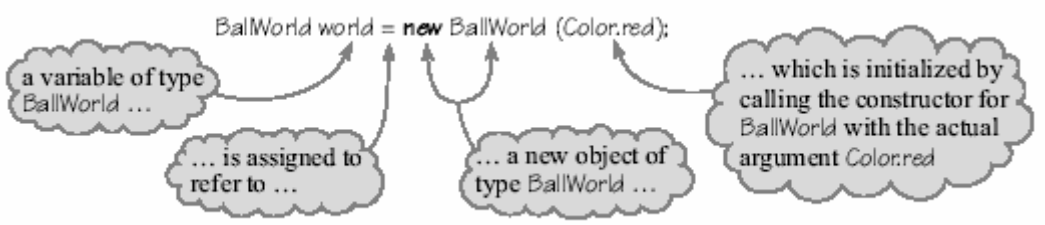
The third data field is an instance of class Ball, which is the second class in our example (for its code refer to Budd, pp. 73). A Ball is an abstraction representing a bouncing ball. It is represented by a colored circular disk that can move around the display surface.

Constructors

The following statement create a new instance of the class BallWorld,

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

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. This is illustrated in the following figure.



Hence, the meaning of the statement is:

create a new object of type BallWorld, initialized using the constructor, and assign it to the variable, world.

Java uses a concept called a constructor to guarantee that objects are placed into a proper initial state the moment they are created.

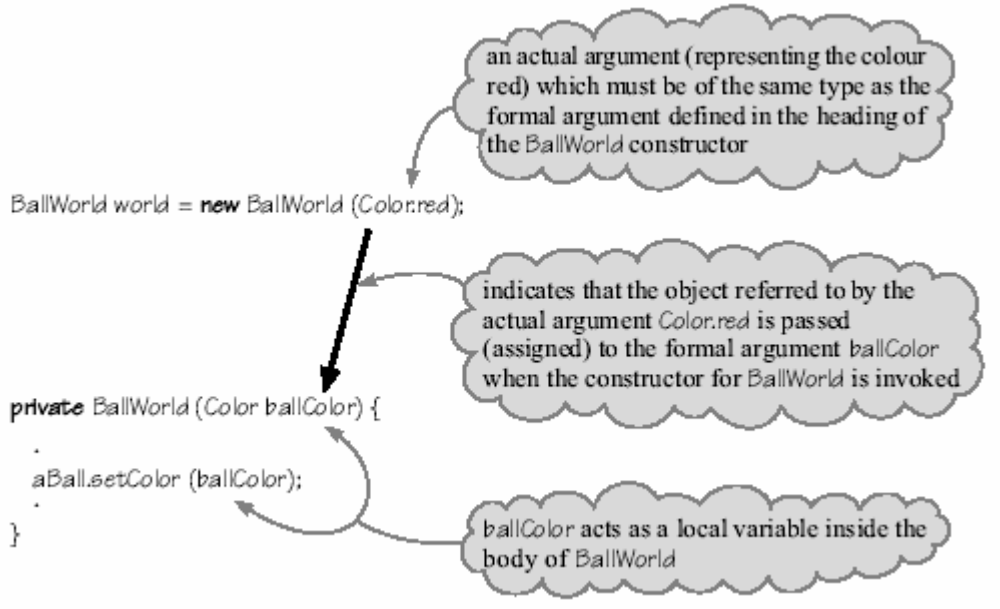
Differences between constructors and methods

- The name of the constructor matches the name of the class in which it appears.
- The constructor does not specify a return type.
- The use can never execute the constructor except as part of creating a new object.

Similarities between constructors and methods

- Both can have arguments.
- Their body consists of a sequence of statements.

When an object is created, the first method invoked using the newly created object is the constructor method. The arguments passed to the constructor are the arguments supplied in the new expression. This can be illustrated in the following figure.



The constructor of the BallWorld class is:

```
private BallWorld (Color ballColor) {
        setSize (FrameWidth, FrameHeight);
        setTitle ("Ball World");

        aBall.setColor (ballColor);
        aBall.setMotion (3.0, 6.0);
}
```

Inheritance (sometimes called *extension*)

Java uses the class JFrame to represent a generic window. By saying that the class BallWorld extends the class JFrame, we indicate that our new class, BallWorld, is a type of frame but a more specialized one.

The class JFrame defines code to perform actions such as resizing the window, arranging for the window to be displayed on the screen, etc.

By extending the class JFrame, our new class *inherits* this functionality which means the abilities are made available to the new class without rewriting them again!

The following statement represents inheritance in our program.

```
public class BallWorld extends JFrame {
```

We can observe the use of inheritance in the variety of methods that are invoked in our program, but are not defined by the class BallWorld. These methods are instead inherited from the parent class JFrame, or from the classes that JFrame inherits from. Two examples are the methods: setSize and setTitle invoked in the BallWorld constructor.

Note. A graphical application will not halt simply because the end of the main method is encountered, but must explicitly call the system method System.exit to terminate execution.

The Java Graphics Model

Graphics in Java is provided as part of the Abstract Windowing Toolkit (AWT).

The Java AWT is an example of a software framework. The idea of a framework is to provide the structure of a program but no application-specific details.

The show method (inherited from JFrame) creates the window in which the action will take place. In order to draw the image shown in the window the show method invokes a method called paint passing as an argument a *graphics object*.

The programmer defines the appearance of the window by providing an implementation of the method paint.

The graphics object passed as argument provides the ability to draw a host of items (e.g. lines, polygons and text). Hence, this object can be used to create a variety of graphical images.

In our program the paint is implemented as follows

```
public void paint (Graphics g) {
            super.paint(g);
            aBall.paint (g);
      }
```

The method invokes the paint method in the parent class JFrame. The method in the parent class erases any previous contents of the window.

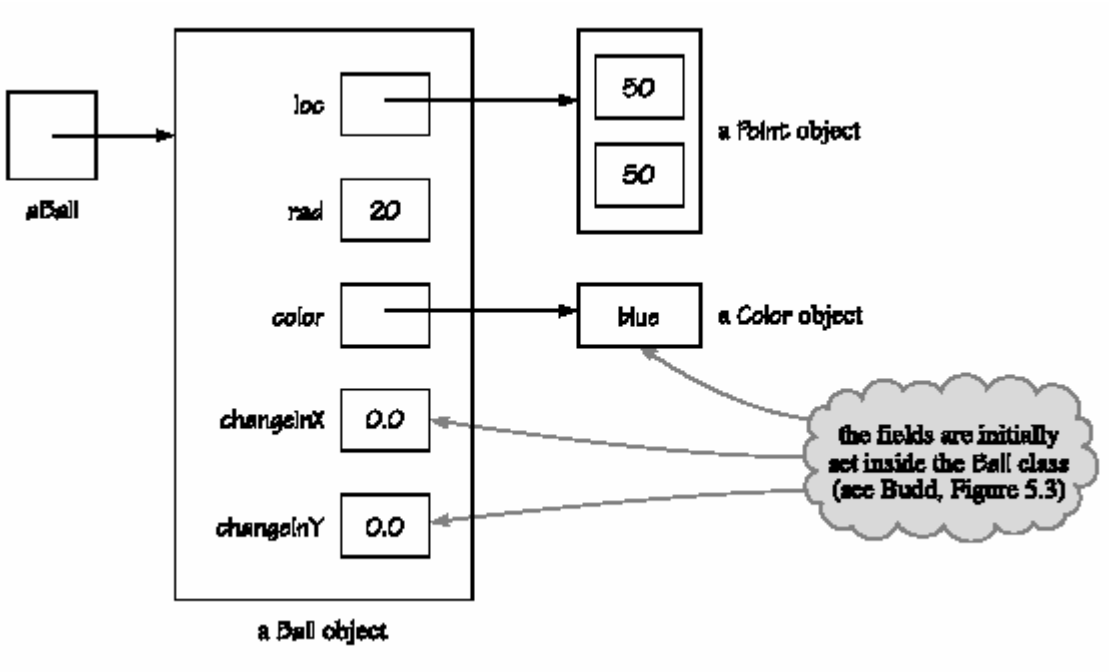
The variable super is used to distinguish the parent method from the child method, as both have the same name, paint.

The run method instructs the ball to move and requests that window to be repainted. To do this the method invokes repaint; an application does not call the paint method directly, but only indirectly by requesting a repainting.

Graphical applications (e.g. repainting) are relatively slow. Hence, it is necessary to halt the program for a short period of time so that the graphics system can catch up. This is done using the command Thread.sleep(). This command will suspend the program for a small period of time (measured in milliseconds).

The sleep method can raise an exception. Hence it must be surrounded by a try block.

The Class Ball
The Ball class defines a round colored object that moves. A Ball object can be pictured as follows.



The data fields of the Ball class are declared as protected. It is a good practice to declare data fields protected, rather than private, even if you do not anticipate extending the class to make new classes.

Methods that allow access to a data field in a class are termed *accessor* methods.

The use of accessor methods is strongly encouraged in preference to making the data fields themselves public, because accessor methods ensures that any modification to a data field will be mediated by the proper method.

For the details of the Ball class read Budd pp. 77-78.

Multiple Objects of the Same Class

The class MultiBallWorld is similar to the BallWorld class except that it creates a collection of balls rather than just a single ball.

To create this collection an array is used as follows:

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

where BallArraySize is a symbolic constant. Although the array is created the array elements cannot be accessed. Each array element must be individually created, once more using a new operation:

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

Each ball is created, and then initialized. Hence, we have multiple instances of the same class each maintain their own separate data fields.

Note:
The **Java platform** is the predefined set of Java classes in the installation, which are contained in packages delivering graphics, input/output, networking, user interfaces, security and more.
The Java platform is also referred to as the Java runtime environment or the core Java APIs (application programming interfaces). We use the Java 2 Platform on this course. One significant addition in Java 2 is a revised run-time library with a windowing interface that is more platform independent than the AWT. In practical terms, part of the AWT has been replaced by a new set of classes known as **Swing classes**.

Read Example 2.1 in Section 2.2 of Block 1.

Block 1: Introduction to Java

Unit 2: Basic Constructs in Java

Section 3: Cannon Game: Events, Inner Classes and Interfaces

Objectives:

On completion of this section you should be able to:

- understand the need for the 'wrapper' class Integer;
- use inheritance to create a new class based on the Ball class introduced in
- Section 1;
- use the pseudovariable `super`;
- explain the use of inner classes and say why they are useful;
- use interface classes and explain why they are useful;
- describe Java's event model, involving events and listeners for events, and use buttons and scrollbars which rely on this model;
- describe and use the default window layout manager;
- write code which enables a window application in Java to be terminated by closing its window.

Summary of Chapter 6

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. The user can control the angle of elevation for the cannon and fire a cannonball. The objective of the game is to hit the target.

The Simple Cannon Game

In this version of the game, the user enters the angle of the cannon from the command argument line, the cannon fires once, and the program halts.

The principal class for our cannon application is shown below.

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

public class CannonGame extends JFrame {

   public static void main (String [] args) {
      //Convert the string command line argument to be an Integer
      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;
   private String message;

   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);
   }

   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;
     }
   }

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

Description of the CannonGame **class:**

The main method:
An instance of class CannonGame is created. The message show displays the window. The method run repeatedly moves the ball and redraws the window. The angle of the cannon is read from the command line argument.

The cannon game is declared to be of type JFrame which is how Java declares a new type of window.

Data fields:
Two public constants, the cannon ball (which is an instance of class Ball), a message, the target and the cannon.

The CannonGame constructor:
The constructor resizes the window frame, 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 message string is set to a value. Finally, the cannonball is created using the angle to determine the initial direction of movement.

The moveCannonBall method:
This method will move the cannon ball slightly and see if the target has been hit, updating the message accordingly.

The run method:
It cycles over a loop that moves the cannon ball, redraws the window, then sleeps while the slower graphics operations are being performed.

The paint method:
It paints the window, the cannon and the cannon ball (if any), the target and the message in the middle of the screen.

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 integer.

```
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);
  }
}
```

The Cannon

The cannon draws the cannon, manipulates the angle of the cannon and the firing of cannon balls. For details read Budd, pp. 88-91.

The Cannon ball

The CannonBall inherits from the class Ball (described in the previous section). 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 to simulate the effect of gravity by reducing the change in the vertical direction by a small amount (stored in constant GravityEffect) each update cycle.

```
import java.awt.*;
//to be more specific, java.awt.Point does the job

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

  public double GravityEffect = 0.3;

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

When a method overrides (i.e. replaces) a similarly named method in the parent class, the pseudovariable super is used to indicate that the method should invoke the method inherited from the parent class.

For example, by invoking super.move() the method move is asking that the move method from the parent class to be executed, and not the version overridden by the class CannonBall.

Integers and ints

int is the name of a primitive data type but not the name of a class. Therefore, the operations of int are not represented by methods and are confined to the normal arithmetic operations such as add and multiply (the symbols +, .] and represent int operations, not methods). If you want to use a value of type int where an object is expected, or want a method that manipulates integers (e.g. converts an integer to a string), you should use the class Integer. Instances of the class Integer are objects such that each Integer object contains (or 'wraps up') a data field of type int.

The Integer class has a number of useful methods, one of which, valueOf, is used in CannonBall to convert a String object, args (previously read from the command line) into an Integer object. Subsequently, this object, referred to by the local argument theta, is transformed into an int value by the message theta.intValue() where intValue is another method of the class Integer.

You can also convert a value of type int (such as 42) into an Integer object by writing,

```
Integer(42)
```

Adding user interaction

In the second version of the cannon game, we improve user interaction by providing 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.

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.

The program for the revised game, now named CannonWorld, is shown below.

Note. We must now import the Java libraries java.awt.event.* and javax.swing.event.*, in order to include the definitions of the event-handling routines for the Java system.

```
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() {
      setSize(FrameWidth, FrameHeight);
      setTitle("Cannon World");
      addWindowListener(new CloseQuit());
        //add graphical objects and their listeners
      JButton fire = new JButton("fire");
      fire.addActionListener(new FireButtonListener());
      getContentPane().add("North", fire);
      slider.addAdjustmentListener(new JScrollBarListener());
      getContentPane().add("East", slider);
  }

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

  public void paint (Graphics g) {…}

  private void moveCannonBall() {…}

  public void run(){…}

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

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

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 main responsibility is to wait for a specific event. When the event occurs, the listener wakes up, and performs the necessary action.

To provide a platform independent way to halt our program, we define a listener and attach it to the close-box of a 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 (part of the standard library class WindowAdapter) defines the behavior we want. By using inheritance to subclass from WindowAdapter we can attach whatever actions we want to this method.

In our program we want to halt the application when the user clicks on the close-box. Hence, we create the following class:

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

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

The argument of the method windowClosing contains information about the mouse-click event (e.g. location of the mouse-click action).

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

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

Inner Classes

In the CannonWorld class note the declaration of two new classes within the application class itself. These classes are known as *inner classes*.

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 (e.g. JScrollBarListener is allowed to access the data fields cannon and message).

Interfaces

Both the inner classes created in CannonWorld use the keyword implements in their header. The *implementation of an interface* is yet another program-structuring mechanism.

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 static.

An interface for ActionListener, used in our program might be written as follows:

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

The Java library, particularly in those sections that related to the handling of events, makes extensive use of interfaces.

When a class is declared as implementing an interface, it is a guarantee that the class must provide a certain behavior (i.e. implements the methods of the interface).

Interfaces vs. Inheritance

Using inheritance, methods and data fields that are declared in the parent class may be used in the child class. Hence, the child class inherits the structure of the parent class.

Using an interface, 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. Hence, a child class inherits a *specification* of methods from an interface, but no structure, no data fields, and no member functions.

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. For examples refer to Budd, pp. 96.

The Java Event Model

Modern graphical user interfaces are structured around the concept of events.

An *event* is an action (e.g. a user clicking the mouse on a button, pressing a key).

The program responds to an event by performing certain actions.

Such interfaces are said to be *event-drive*.

For each type of event, there is an associated listener. When the event occurs, the listener goes into action and performs its assigned behavior.

Here is an overview of the process involving the fire button, which should help to clarify the issues for you (a similar process applies to the scroll bar).

- There are three activities associated with programming widgets: they have to be located at a chosen position in the window; they must be associated with a listener object (which detects when the widget has been used, that is, an event has occurred); and the action to be performed in response to the event (a mouse click, for example) must be specified.
- The action to be performed for a button, for example, must be implemented by a method named actionPerformed (declared in the ActionListener interface contained in the event handling package java.awt.event). This means that you must include, in the class that extends JFrame, an inner class, such as FireButtonListener which implements ActionListener containing a definition for actionPerformed.

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

- The association of the widget object with a listener object is achieved by sending a message (such as addActionListener in the case of a button) to the widget. This method passes the listener object (for example, an instance of FireButtonListerner) as an argument to the widget.

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

- When fire is pressed by the user, the Java system automatically creates an event object of type ActionEvent, which contains useful information about the event (for example, the location of the mouse pointer when the mouse was clicked).

The above discussion illustrates why inner classes are useful. The method actionPerformed needs to access the private data fields (aBall, and cannon) of the CannonWorld class. This is made easier by implementing FireButtonListerner as an inner class.

More examples and explanation can be found in Budd, pp. 96-97.

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 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. This is done in the constructor of our program to place a button on the top of the window, and the scroll bar on the right hand side.

```
public CannonWorld() {
   …
    //add graphical objects and their listeners
   JButton fire = new JButton("fire");
   fire.addActionListener(new FireButtonListener());
   getContentPane().add("North", fire);
   slider.addAdjustmentListener(new JScrollBarListener());
   getContentPane().add("East", slider);
}
```