M301: Software Systems & their Development

Block 1: Introduction to Java

Unit 3: Using Inheritance

Aims of the unit:

- reinforce the concepts of inheritance and interfaces;
- show how to capture and deal with exceptions;.

Block 1: Introduction to Java

Unit 3: Using Inheritance

Section 1: Pinball Game: Collection Classes, Mouse Events, Inheritance and Polymorphism

Objectives:

On completing this section you should be able to:

- develop a program that will respond to mouse-related events;
- use the `Vector` class collection class for holding objects and primitive data types;
- describe what an *exception* is in Java and write simple exception handlers;
- develop programs using inheritance and interfaces;
- use polymorphism and polymorphic variables in your programs.

Summary of Chapter 7

The Pinball Game

The application we will develop in this chapter, 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. Ball 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.

First Version of Game

Because we will later be creating objects that will need to communicate with the window object, the variable `world` is declared as `public static` value, rather than as a local variable to the `main` method.

The `PinBallGame` class is shown below.

```
import java.awt.event.*;
import java.util.Vector;
import javax.swing.*;
import java.awt.*;

public class PinBallGame extends JFrame {

  public static void main(String[] args) {
    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; //A vector to hold balls
  private PinBallFire fireButton = new PinBallFire(new Point(FrameWidth -
40, FrameHeight - 40));

  public static PinBallGame world;

  private class MouseKeeper extends MouseAdapter {…}

  public void run() {…}

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

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

```
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));}
}
```

The class PinBall (shown below) extends the class CannonBall (already discussed in the previous unit). Differences are that:

- Here the initial direction is slightly to the left of vertical, and includes a small random number perturbation (to be less predictable).
- Also, a PinBall can return an instance of the standard class Rectangle that represents the bounding rectangle for the ball (to help detect when a ball has hit a target).

```
public 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);
  }
}
```

Collection Classes

Unlike the Cannon game, the pinball game allows several balls to be moving at one time.

To manage this we need a data structure that can hold a collection of values. One collection data structure we have already seen before is the array. However, when we

allocate a new array we need to state the number of elements the array will hold. In our case, we can not predict the number of balls (array elements).

Java provides an indexed data structure which can dynamically grow as new values are inserted into the collection. This data structure is called Vector.

To access a vector, the programmer must first import the vector class definition.

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

The vector is declared by simply providing a name:

```
private Vector balls;
```

Unlike an array, it is not necessary to state the type of values that a Vector will hold. A vector is restricted to holding only objects (e.g. one cannot create a vector of integer values (ints) but can create a vector of instances of class Integer).

The space for a Vector must be created and assigned as follows:

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

Note that no fixed limit is set for the space.

To insert a new element into the vector we use the method addElement, as follows:

```
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++)
```

To access values in a vector we use the method elementAt. 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. For example:

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

Here we cast the value into the type PinBall.

Mouse Listeners

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 are of interest. Hence, the interface for a MouseListener has the following structure:

```
public interface MouseListener {
        public void mouseClicked (MouseEvent e);
        public void mouseEntered (MouseEvent e);
        public void mouseExisted (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 Java insists that the programmer provide a definition for all operations.

To simplify such cases, the Java library provides a class named MouseAdapter. This class implements the MouseListener interface but uses an empty method for each method. That is, a MouseAdapter does nothing in response to any mouse event.

The programmer can write a new class that *inherits* from MouseAdapter, and overrides (or redefines) the methods of interest.

In our 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.

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

The class MouseKeeper implements one method of MouseAdapter. MouseKeeper is defined as follows:

```
private class MouseKeeper extends MouseAdapter {
   public void mousePressed(MouseEvent e) {
     //locate position of mouse cursor when moused clicked
     int x = e.getX();
     int y = e.getY();
     //if mouse clicked within firing region, create new
     //ball and thread, and start thread
     if (fireButton.includes(x, y))
       balls.addElement(fireButton.fire(x, y));
   }
 }
```

Running the Application

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.

```
public 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:

```
   private void 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. There are different types of targets, some add values to the score, some move balls in a new direction, and some swallow the ball.

To simplify the program, we will maintain all the different types of targets in a single data structure, a vector.

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. Inheritance is a mechanism for sharing *structure* (i.e. data fields and methods, including the implementation of those methods); a PinBall, for example, is simply a type of Ball that has the same structure and behavior as a Ball, adding a few features, but maintaining all the characteristics of the original.

There is little in the way of common structure between different targets. What is needed in this case is the ability to state that the targets concepts share the same *behavior*, although they have nothing in common in *structure*.

In Java this is accomplished by describing common behavior as an interface, and declaring the targets objects implement the same interface.

An interface for our pinball targets can be described as follows:

```
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 in a pinball game.

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 pain itself, and each can provide 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.

The class descriptions (along with explanation) of the different targets in our program can be found in Budd, pp. 110-117. Note that each target class must explicitly state that it implements the PinBallTarget interface, and must provide a specific meaning for each of the four elements of that interface. For example a target known as a Wall is implemented as follows:

```
public class Wall implements PinBallTarget
{
        public Rectangle location;

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

        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 (PinBall aBall)
        {
                return location.intersects(aBall.box());
        }

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

The advantage of declaring 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. Hence, a variable declared as maintaining a PinBallTarget could, in fact, be a Wall, a Spring, or any other target.

We will use this property, by storing all the targets in our program in a single data structure, and testing the motion of the ball against the location of each target in turn.

```
private Vector targets;
…
targets = new Vector();
…
for (int j = 0; j < targets.size(); j++) {
          PinBallTarget target =
           (PinBallTarget) targets.elementAt(j);
         if (target.intersects(theBall)) target.hitBy(theBall);
       }
```

Similarly a variable declared as maintaining a value of type Ball could, in fact, be holding a cannonball, pinball, or any other value derived from a class that extends the original class Ball.

The Hole target

A Hole consumes any ball it comes in contact with, removing it from the playing surface. A Hole is actually structurally similar to a Ball (both are circular and have a location). The hole inherits much of its behavior from the class Ball, including the paint and moveTo methods.

In addition, the Hole declares that it implements the PinBallTarget interface (it should implement the interface methods).

```
public class Hole extends Ball implements PinBallTarget
{
	public Hole (int x, int y)
	{
		super(new Point(x,y), 12);
		setColor(Color.black);
	}

	public boolean intersects (PinBall 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 (PinBall aBall)
	{
		aBall.moveTo(0, PinBallGame.FrameHeight + 30);
		aBall.setMotion(0, 0);
	}
}
```

A class that inherits from an existing class that implements an interface must of necessity also implements the interface. For example, a ScorePad is like a hole represents a circle region; however, the score pad adds a certain amount to the player's score when it is struck by a ball. The class description of ScorePad is shown below.

```
public class ScorePad extends Hole
{
	protected int value;

	public ScorePad (int x, int y, int v)
	{
		super (x, y);
		value = v;
		setColor (Color.red);
	}

	public void hitBy(PinBall aBall)
	{
		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);
        }
}
```

Note how the ScorePad class 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.

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.

Adding a Label to Our Pinball Game

We will also add a new graphical element to our program. This element is a textual label in a banner across the top of the window. This is done by declaring a new Label, and adding it to the "North" part of the window. As the user scores new points, the text of the label is updated.

```
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);
  }
```

Pinball Game Construction Kit: Mouse Events Reconsidered

We will improve our program by providing a pallet of targets from which the user can select and place on the game surface. Hence, the user can click the mouse down in one of these targets, 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.

This is done by overriding both the mousePressed and the mouseReleased methods inherited from the mouse adapter.

The two methods communicate with each other via a variable named element.

The mousePressed method creates a potential target, determined by the coordinates of the point at which the mouse goes down.

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, then it adds a new target to the game.

```
private class MouseKeeper extends MouseAdapter{
   public void mousePressed(MouseEvent e) {
      element = null;
                  //locate position of mouse cursor when moused clicked
     int x = e.getX();
     int y = e.getY();
                  //if mouse clicked within firing region, create new
                  //ball and thread, and start thread
     if (fireButton.includes(x, y))
        balls.addElement(fireButton.fire(x, y));
    if (x < 40) { //each target occupies a 40x40 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();
     int y = e.getY();
     if ((element != null) && (x > 50)) {
       element.moveTo(x, y);
       targets.addElement (element);
       repaint();
     }
   }
}
```

Block 1: Introduction to Java

Unit 3: Using Inheritance

Section 2: Understanding Inheritance

Objectives:

On completing this section you should be able to:

- give an intuitive description of inheritance and the use of interfaces;
- describe the methods of the class Object from which all Java classes inherit;
- describe the concept of substitutability and use it as a programming mechanism;
- describe the difference between a subtype and a subclass;
- describe and use the forms of inheritance to which Budd refers under the headings of *specialization* , *specification* and *construction* .

IMPORTANT: Read Section 2.3 in Block 1

Summary of Chapter 8

The first step in learning OO programming is understanding the basic philosophy of organizing a computer program as the interaction of loosely coupled software components.

The next step in learning OO programming is organizing classes into a hierarchical structure based on the concept of inheritance.

By *inheritance*, we mean the property that instances of a child class (or subclass) can access both data and behavior (methods) associated with a parent class (superclass).

By *inheritance of code*, we mean the case where a class extends another class.

By *inheritance for specification*, we mean the case where a class implements an interface.

An Intuitive Description of Inheritance

In programming languages, inheritance means the following.

- The relationship between the classes that inherit from each other is transitive. For example, this means that the class Dog inherits all the data fields and methods of Mammal (since it extends Mammal), but also inherits all the methods and data fields of Animal (since Mammal inherits from Animal).
- The methods and data fields of a child class are an extension of the parent (i.e. they include the methods and data fields of the parent).

- The child class is also a contraction of the parent class, in the sense that it is more specialized.
- The two seemingly opposed concepts, *extension* and *contraction*, are easily reconciled if one realizes that *extension* applies at the class level and *contraction* at the object level. For example, consider the class Employee which inherits from the class Person. At the object level, there will be many more persons than employees since not all persons will be employees. In this sense, Employee is a contraction of Person. At the class level, employees will have all the methods and data fields associated with being a person, as well as those which are specialized to being an employee. In this sense, Employee is an extension of Person.

The Base Class Object

In Java all classes use inheritance. Unless specified otherwise, all classes are derived from a single root class named Object.

If no parent class is explicitly provided, the class Object is implicitly assumed.

The class Object provides minimal functionality guaranteed to be common to all objects. These includes the following methods: equals(), getClass(), hashCode() and toString().

Subclass, Subtype and Substitutability

The idea of substitutability is that the type given in a declaration of a variable does not have to match the type associated with a value the variable is holding.

An example of this concept was used in the Pin Ball Game program. The variable target was declared as a PinBallTarget, but in fact held a variety of different types of values that were created using implementation of PinBallTarget.

```
PinBallTarget target = (PinBallTarget) targets.elementAt(j);
```

Because Object is a parent class to all objects, a variable declared using this type can hold any nonprimitive value. The collection class Vector makes use of this property, holding its values in an array of Object values. Because the array is declared as Object, any object value can be stored in a Vector.

Substitutability can also occur through the use of interfaces (for details refer to Budd, pp. 129).

A *subtype* explicitly recognizes the principle of substitutability. This means that a subtype object must behave in the same way as the supertype object, that is, it must respond to the same operations. So, for example, Employee is a subtype of a Person because employee objects behave like person objects as well as having their own specific operations.

In contrast, a *subclass* represents a way of creating a new class by extending another. It is easy to recognize subclass from the source description of a program by the presence of the keyword `extends`. Being a subclass does not guarantee that all the behavior inherited from the parent class is sensible behavior for the subclass. For example, it is perfectly possible to define `Employee` by extending the class `Window` but doing so would be rather nonsensical because not all the behavior of a window is sensible behavior for an employee.

In the majority of situations a subclass is also a subtype. However, in some cases (we will see later) this may not be true. In addition, subtypes can be formed using interfaces, linking types that have no inheritance relationship whatsoever.

Forms of Inheritance

Inheritance is employed in a surprising variety of ways. Presented here are a few of its more common uses.

Inheritance for Specialization

Inheritance for specialization involves the child class being a more specialized variety of the parent class, but satisfying the specifications of the parent behavior in all relevant respects.

Thus, this form always creates a subtype, and the principle of substitutability is explicitly upheld.

Along with the following category (i.e. inheritance for specification) this is the most ideal form of inheritance, and something that a good design should strive for.

The behavior of a ball is in all respects appropriate to that of a cannon ball, and in this sense `CannonBall` is a subtype of `Ball` as well as a subclass of `Ball`. Note the fact that the `move` method of `Ball` has been overridden in `CannonBall`; this does not necessarily negate the subtyping relationship.

We say that inheritance for specialization occurs in this example because the child class (i.e. `CannonBall`) satisfies all the properties that we expect of the parent class (`Ball`). Furthermore, the new class overrides one or more methods, specializing them with application-specific behavior.

Inheritance for Specification

The main characteristic of *inheritance for specification* is that it guarantees that the inheriting classes maintain a common interface, that is, they implement methods having the same headings. It is a form of inheritance for specialization except that the child class must provide the implementations.

There are two different mechanisms provided by Java to support the idea of inheritance of specification. The most obvious technique is the use of interfaces (as we already have seen in the previous sections).

Inheritance of specification can also take place with inheritance of classes formed using extension. One way to guarantee that a subclass must be constructed is to use the keyword abstract.

A class declared as abstract must be subclasses; it is not possible to create an instance of such a class using the operator new. In addition, individual methods can also be declared as abstract, and they too must be overridden before instances can be constructed.

An example abstract class in Java library is Number, a parent class for the numeric wrapper classes Integer, Long, Double, and so on. The class description is as follows:

```
public abstract class Number {

        public abstract int intValue();
        public abstract long longValue();
        public abstract float floatValue();
        public abstract double doubleValue();

        public byte byteValue()
        { return (byte) intValue(); }

        public short shortValue()
        { return (short) intValue(); }
}
```

Subclass of Number must override the methods intValue, longValue, floatValue and doubleValue.

Notice that not all methods in an abstract class must themselves be declared abstract. Subclasses of Number need not override byteValue or shortValue, as these methods are provided with an implementation that can be inherited without change.

Inheritance for Construction

Inheritance for construction represents the case where one class extends another in order to inherit *some* of the methods of the parent class. In other words, the two classes are not related as subtype and supertype, but they simply form a 'marriage of convenience'. This form of inheritance is *not* considered good practice because, even though the objects of the subclass will be substitutable for those of the superclass, it does not make sense to substitute one for the other.

An example of inheritance for construction occurs in the Java library. There, the class Stack is constructed using inheritance from the class Vector (see implementation details in Budd pp. 134). As abstractions, the concept of the stack and the concept of a vector have little in common; however, from a pragmatic point of view using the Vector class as a parent greatly simplifies the implementation of the stack. Another example is the Hole and Ball in the pin ball game of the previous section.