Inheritance & Polymorphism Recap

Introduction

- Besides composition, another form of reuse is inheritance.
- With inheritance, an object can inherit behavior from another object, thus reusing its code.
- The class inheriting is called a subclass (or derived class).
- The class inherited from is called a superclass (or base class).

Inheritance & Polymorphism

- Inheritance is a form of software reusability in which new classes are created from existing classes by
  - absorbing their attributes and behaviours and
  - enhance these, with capabilities the new classes require.
- Polymorphism
  - enables us to write programs (and methods) in a general fashion to handle a wide variety of existing **and** yet-to-be-specified related classes.
  - makes it easy to add new capabilities to a system.
- Inheritance and Polymorphism are effective techniques for dealing with software complexity.

Case Study - Point,Circle,Cylinder

- Suppose a set of shape classes such as Circle, Triangle , Square etc. are all derived from superclass Shape, and each class has the ability to draw itself (has its own draw() method,which would be different in each case).
- When drawing a shape, whatever shape it might be, it would be nice to be able to treat all these shapes generically as objects of the superclass Shape.
- Then to draw any shape, we could call the draw() method of superclass Shape and let the program determine *dynamically* (at run time) which subclass draw() method should be called (depending on the objects type).
- To enable this kind of behaviour, we declare draw() in the superclass, and override draw() in each of the subclasses to draw the appropriate shape.

The *abstract* keyword

- An abstract method is not actually implemented in the class. The body of the method is implemented in subclasses of that class.

```
public abstract void draw();
```

- An abstract method must be part of an abstract class.

```
public abstract class Shape extends Object
```

- Abstract classes cannot be instantiated. It is a compile-time error to try something like

```
Shape m = new Shape();
```

where Shape has been declared to be abstract

Abstract Classes

- A variable of an abstract type can refer to an object of a subclass of that type.
- This permits polymorphism.
- Sometimes a collection, such as an array, of a superclass or abstract superclass type contains objects of subclasses.
- An iterator is used to traverse all objects in the collection.
- A message sent to each object behaves in a polymorphic manner.

Case Study : Point,Circle,Cylinder (1)

```
public abstract class Shape extends Object
{
        public double area() {
                return 0.0;
        }
        public double volume() {
                return 0.0;
        }
        public abstract String getName();
        public abstract void Draw();
}
```

Case Study : Point,Circle,Cylinder (2)

```
import javax.swing.JOptionPane;

public class Point extends Shape
{
   protected int x, y; // coordinates of the Point

   public Point(){ setPoint( 0, 0 ); }

   public Point( int x, int y ){ setPoint( x, y ); }

   public void setPoint( int x, int y )
   {
      this.x = x;
      this.y = y;
   }
```

Case Study : Point,Circle,Cylinder (3)

```
public int getX() { return x; }

public int getY() { return y; }

public String toString()
    { return "[" + x + ", " + y + "]"; }

public String getName(){ return "Point"; }

public void Draw()
    {
    JOptionPane.showMessageDialog(null,getName() + ": " + this);
    }
}
```

Case Study : Point,Circle,Cylinder (4)

```
import javax.swing.JOptionPane;

public class Circle extends Point {  // inherits from Point
   protected double radius;

   public Circle(){
      // implicit call to superclass constructor
      setRadius( 0 );
   }

   public Circle( double r, int x, int y ){
      super( x, y );  // call to superclass constructor
      setRadius( r );
   }

   public void setRadius( double r )
      { radius = ( r >= 0.0 ? r : 0.0 ); }
```

Case Study : Point,Circle,Cylinder (5)

```
public double getRadius() { return radius; }

    public double area() { return Math.PI * radius * radius; }

    public String toString()
    {
        return "Center = " + "[" + x + ", " + y + "]" +
                "; Radius = " + radius;
    }

    public String getName()
        { return "Circle"; }

    public void Draw()
        {
        JOptionPane.showMessageDialog(null,getName() + ": " + this);
        }
}
```

Case Study : Point,Circle,Cylinder (6)

```
import javax.swing.JOptionPane;

public class Cylinder extends Circle {
   protected double height;  // height of Cylinder

   // no-argument constructor
   public Cylinder()
   {
   // implicit call to superclass constructor here
      setHeight( 0 );
   }

   // constructor
   public Cylinder( double h, double r, int x, int y )
   {
      super( r, x, y );    // call to superclass constructor
      setHeight( h );
   }
```

Case Study : Point,Circle,Cylinder (7)

```
// Set height of Cylinder
  public void setHeight( double h )
     { height = ( h >= 0 ? h : 0 ); }

  // Get height of Cylinder
  public double getHeight() { return height; }

  // Calculate area of Cylinder (i.e., surface area)
  public double area()
  {
     return 2 * super.area() +
            2 * Math.PI * radius * height;
  }
```

Case Study : Point,Circle,Cylinder (8)

```
// Calculate volume of Cylinder
  public double volume() { return super.area() * height; }

  // Convert a Cylinder to a String
  public String toString()
     { return super.toString() + "; Height = " + height; }

  // Return the class name
  public String getName() { return "Cylinder"; }

   public void Draw()
     {
     JOptionPane.showMessageDialog(null,getName() + ": " + this);
     }
}
```

Case Study : Point,Circle,Cylinder (9)

```
import java.text.DecimalFormat;
import javax.swing.JOptionPane;

public class Tester {
   public static void main( String args[] )
   {
      String output;
      Point point = new Point( 7, 11 );
      Circle circle = new Circle( 3.5, 22, 8 );
      Cylinder cylinder = new Cylinder( 10, 3.3, 10, 10 );

      Shape arrayOfShapes[];

      arrayOfShapes = new Shape[ 3 ];
```

Case Study : Point,Circle,Cylinder (10)

```
// aim arrayOfShapes[0] at subclass Point object
arrayOfShapes[ 0 ] = point;

// aim arrayOfShapes[1] at subclass Circle object
arrayOfShapes[ 1 ] = circle;

// aim arrayOfShapes[2] at subclass Cylinder object
arrayOfShapes[ 2 ] = cylinder;

    point.Draw();
    circle.Draw();
    cylinder.Draw();

DecimalFormat precision2 = new DecimalFormat( "0.00" );
```

Case Study : Point,Circle,Cylinder (11)

```
// Loop through arrayOfShapes and print the name,
      // area, and volume of each object.
 for ( int i = 0; i < arrayOfShapes.length; i++ )
   {
    arrayOfShapes[ i ].Draw();


  output = "\nArea = " + precision2.format( arrayOfShapes[ i ].area() ) +
       "\nVolume = " + precision2.format( arrayOfShapes[ i ].volume() );
        JOptionPane.showMessageDialog( null, output,"Demonstrating Polymorphism",
        JOptionPane.INFORMATION_MESSAGE );

 }
      System.exit( 0 );
   }
}
```

Polymorphism & Dynamic Binding

Interfaces (1)

- Interfaces provide some features of multiple inheritance:
  - Like an **abstract** class, an interface defines a set of methods (and perhaps constants as well), but no implementation.
  - By using the **implements** keyword, a class can indicate that it implements that set of methods.
  - This makes it unnecessary for related classes to share a common superclass or to directly subclass **object**.
  - It's possible for a class to implement several interfaces.

Interfaces (2)

- An interface is like a class with nothing but abstract methods and final, static fields. All methods and fields of an interface must be public.
- However, unlike a class, an interface can be added to a class that is already a subclass of another class. Furthermore an interface can apply to members of many different classes
- When you introduce a new class, you can choose to “support” any number of interfaces
- For each interface you support you must implement member functions defined in the interface

Interfaces (3)

- All interface members are public.
- Methods are abstract.
- Constants are static and final.
- Generally, the keywords public, abstract, static and final are not used in an interface declaration since they will have these characteristics by default.

Interfaces (4)

- A class can inherit from only one direct superclass, but it can implement multiple interfaces.

```
public class MyClass extends MySuperClass implements IFace1,
   IFace2, IFace3, … {
```

- If a class implements an interface but not all of its methods, it must be an abstract class.
- Interface methods are implicitly abstract.
- Interfaces are defined in their own .java file named with the interface name.

Interfaces (5)

```
public interface IShape
{
        public String getName();
        public void Draw();
}
```

```
public class Point extends Shape implements IShape {
        //Previous implementation ...

        public String getName() {…}
        public void Draw() {…}
}
```

Inheritance Principles

- Common operations and fields belong to a superclass
- Use inheritance to model the is-a relationship
- Don't use inheritance unless ALL inherited methods make sense.
- Use Polymorphism not type information