Abstract Classes

CS 3: Computer Programming in Java
Objectives

- Abstractly define and discuss the properties of the abstract class
- Look at an example of an abstract class and its extension
- Taking the inheritance design and converting it into source code
- Finally, show how we test the source code
Abstract Classes

- Object oriented languages such as Java and C++ use a concept of abstract classes, so that an object of certain type can be conceptualized and then extended further.

- For example, take the concept of round shapes in geometry:
  - The roundness is a property that is present in many different forms in real objects.
  - The circle, spheres, and elliptical objects have roundness.
  - However, the concept roundness is a bit abstract, so if we wrote a class to capture this concept, we may never be able to truly find a real object for which the class Roundness can be a template.

- The concepts and behaviors, that apply to other objects, but are too abstract to make concrete, are often written in the form of abstract classes.
Main Properties of Abstract Classes

- The abstract class has at least one abstract method
  - An abstract method is a method, which has no body or definition
  - It just has a signature and return type
- The usefulness of abstract class lies in it being extended by other classes or other classes being derived from it
- An abstract class cannot be instantiated, though it can be used as a reference type for the sub class objects
Inheritance Design

Shape

Point (Coordinate X, Coordinate Y)

A point is-a shape

A square is-a point

Square (Inherits X and Y as center from Point; has edge/length)

A cube is-a square

Cube (Inherits X, Y, and edge from Square; needs no new fields)

A cube is-a square
A shape is a general abstract concept
We know that a shape can have area and in some cases volume
A shape certainly has a name
However, shape is abstract enough that we can define an abstract class called Shape to model it
We would never need an instance of class Shape because there is no real object in the world that is called “Shape”
Now there can be many types of shapes

A point is a shape, which is infinitesimal in size

However, it has properties such as coordinates x and y in the Cartesian system

Therefore, the Class Point can extend the general concept of Shape to a concrete form

Hence, we derive the class Point from Shape
A Square has a center point just like a Point and a class Square can inherit the coordinates x and y from class Point.

Nevertheless, Square shape has additional property like a length or edge.

Therefore, we derive the class Square from Point.
Finally, a Cube has the same properties or fields as the Square does, such as a central point and an edge. However, it also has a volume, which Square does not have. Yet, no new fields are needed if we were to derive the class Cube from class Square.
Converting the Inheritance Design to Code - Shape Class

- Now we convert the above design into source code, first by writing various classes in the inheritance hierarchy and then writing a driver class program to test them.

```java
public abstract class Shape {
    public double area() {
        return 0;
    }
    public double volume() {
        return 0;
    }
    public abstract String getName();
}
```

- **Uses the keyword** `abstract`
- **Implemented methods**
- **Abstract method**
Converting the Inheritance Design to Code - Shape Class (2)

- The class Shape is an abstract class because it has one un-implemented (abstract) method: `getName()`
  - The syntax requires that the keyword `abstract` be included in the declaration of method signature
- A class can also become abstract if it derives from an abstract class and does not provide implementation for the abstract methods for its parent
  - The keyword `abstract` would be then required in class declaration
Converting the Inheritance Design to Code - Point Class

```java
public class Point extends Shape {
    protected int x;
    protected int y;

    public Point() {
        this(0);
    }

    public Point(int a) {
        this(a,0);
    }

    public Point(int a, int b) {
        this.x = a;
        this.y = b;
    }
}
```
Converting the Inheritance Design to Code - Point Class (2)

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

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

public void setX(int x1)
{
    this.x = x1;
}

public void setY(int y1)
{
    this.y = y1;
}
```
Converting the Inheritance Design to Code - Point Class (3)

```java
public int getX()
{
    return this.x;
}

public int getY()
{
    return this.y;
}

public String toString()
{
    return "[" + this.x + ", " + this.y + "]";
}
```
Converting the Inheritance Design to Code - Square Class

```java
public class Square extends Point {
    protected double edge;

    public Square() {
        this(0.0);
    }

    public Square(double edge1) {
        this(edge1, 0, 0);
    }

    public Square(double edge1, int a) {
        this(edge1, a, 0);
    }
}```
Converting the Inheritance Design to Code - Square Class (2)

```java
public Square(double m_edge, int a, int b) {
    super(a, b);
    setEdge(m_edge);
}

public void setEdge(double m_edge) {
    this.edge = (m_edge >= 0 ? m_edge : 0);
}

public double getEdge() {
    return this.edge;
}
```
Converting the Inheritance Design to Code - Square Class (3)

```java
public class Square {

    private double edge;

    public double area() {
        return this.edge * this.edge;
    }

    public String toString() {
        return "center = " + super.toString() + " ; Edge = " + this.edge;
    }

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

    //End of class Square
}
```
Converting the Inheritance Design to Code - Cube Class

```java
public class Cube extends Square {
    public Cube() {
        this(0.0);
    }
    public Cube(double edge1) {
        this(edge1,0,0);
    }
    public Cube(double edge1, int a) {
        this(edge1, a, 0);
    }
    public Cube(double m_edge, int a, int b) {
        super(m_edge, a, b);
    }
}
```
Converting the Inheritance Design to Code - Cube Class (2)

```java
public double volume()
{
    return this.edge * this.edge * this.edge;
}

public double area()
{
    return 6 * this.edge * this.edge;
}
```
Converting the Inheritance Design to Code - Cube Class (3)

```java
public String toString()
{
    //return super.toString() + " ; Height = " + this.edge;
    return super.toString();
}

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

}//end of class Cube
```
Unified Modeling Language for the Inheritance Design
import javax.swing.JOptionPane;
import java.text.*;

public class TestShapes
{
    public static void main(String[] args)
    {
        DecimalFormat precision2 = new DecimalFormat( "0.00" );
        boolean done = false;
        String Input = "";
        String Output = "";
while(!done)
{
    Input = JOptionPane.showInputDialog("Please enter the "
            +"x coordinate of the shape.\n");
    int x_cord = Integer.parseInt(Input);
    Input = JOptionPane.showInputDialog("Please enter the "
            +"y coordinate of the shape.\n");
    int y_cord = Integer.parseInt(Input);
    Input = JOptionPane.showInputDialog("Please enter the "
            +"edge of the shape. Enter 0 for the point.\n");
    double edge = new Double(Input).doubleValue();
    Shape Gen_Shape = null;
if(edge == 0)
{
    Gen_Shape = new Point(x_cord,y_cord);
    Output = Gen_Shape.getName() + ": " + Gen_Shape.toString() + "; "
    + "Area of Point = " + precision2.format(Gen_Shape.area())+ "; "+
    "Volume of Point = " + precision2.format(Gen_Shape.volume());
}
else
{
    Input = JOptionPane.showInputDialog("Please enter "+
        +"whether the data you just entered are for [S]quare or [C]ube");
}
Testing the Inheritance Design (4)

```java
if(Input.equals("S") || Input.equals("s"))
{
    Gen_Shape = new Square(edge,x_cord,y_cord);
    Output = Gen_Shape.getName() + ": " + Gen_Shape.toString() + " ; 
    + "Square Area = " + precision2.format(Gen_Shape.area()) + " ; 
    + "Square Volume = " + precision2.format(Gen_Shape.volume());
}
else if(Input.equals("C") || Input.equals("c"))
{
    Gen_Shape = new Cube(edge,x_cord,y_cord);
    Output = Gen_Shape.getName() + ": " + Gen_Shape.toString() + " ; 
    + "Cube Area = " + precision2.format(Gen_Shape.area()) + " ; 
    + "Cube Volume = " + precision2.format(Gen_Shape.volume());
}
```
Testing the Inheritance Design (5)

JOptionPane.showMessageDialog(null, Output);
Input = JOptionPane.showInputDialog("More shapes? 

+"Enter [Y]es to continue or [N]o to exit.

if(Input.equals("Y") || Input.equals("y"))
   done = false;
else
   done = true;
}
System.exit(0);
About the Test Code

- The purpose of this program is to get from the user the properties of shapes and then display properties, such as central coordinates of shape, and, if relevant, then display its area and volume.
- The coordinates of the center of a Shape, be it a point, square, or cube would not change with its type.
- First, the user input for Shape’s center x, y coordinates are accepted and parsed.
The user is prompted to input the edge of the Shape or zero if the Shape is a point

Understand that all instantiable objects such as Point, Square, and Cube are derived from super class Shape

Therefore the reference type Shape can hold the address of any of them

We declare a general Shape type reference Gen_Shape (L10.16#24)

If data for edge entered by the user were zero, then it is certain that the shape is a point

In that case the first if block is executed

Inside the if block the final point object is constructed with user provided center coordinates x_coord and y_coord and a super class reference Gen_Shape points to this derived class Point object
About the Test Code (3)

- An Output String is built to describe the properties of this Shape object, first by getting its name, and central coordinates by calling member methods getName and toString.
- Then to add its area and volume to the overall description, the methods area and volume are called.
  - Understand that class Point does not actually have the coded area and volume methods and it automatically binds to its super class Shape’s implemented methods area and volume for their execution.
- The JOptionPane outside the else block displays the properties of the Shape.
- The other JOptionPane shows the system of displays when the user constructs a Point shape.
Sample Run of the Test Code

- **Input**
  - Please enter the x coordinate of the shape:
    - Value: 10
  - OK, Cancel

- **Input**
  - Please enter the y coordinate of the shape:
    - Value: 10
  - OK, Cancel

- **Input**
  - Please enter the edge of the shape. Enter 0 for the point:
    - Value: 0
  - OK, Cancel

- **Message**
  - Point: [10, 10]; Area of Point = 0.00; Volume of Point = 0.00
  - OK
About the Sample Run

- User is given, through the display, all the correct properties of Point type object constructed by them.

- Notice that even though point object created was represented by a Shape type reference due to the property of dynamic binding, Java correctly recognizes the object to be of Point type and only calls the methods from Point class to build the property String Output.

- If the edge value inputted is not zero then user may wish to construct either a square or a cube (both these shapes having an edge).
  - In that case, the else block is executed.
Creating the Cube

Input
Please enter the edge of the shape. Enter 0 for the point.
10
OK  Cancel

Input
Please enter whether the data you just entered are for [S]quare or [C]ube
C
OK  Cancel

Message
Cube: center = [10, 10]; Edge = 10.0; Cube Area = 600.00; Cube Volume = 1000.00
OK
About Creating the Cube

- If, after entering the center coordinates of 10, and 10, the user also entered an edge value of 10 for a cube object, the displays print the properties of Cube constructed by the user.

- Here also, though the reference of type Shape (Gen_Shape) holds the address of Cube object created, Java does the correct dynamic binding by calling the appropriate methods for area and volume for the Shape object Cube.

- We shall soon discuss advantages of extending other Java abstract classes called adapter classes, which implement some methods for a Java interface but leave one of more methods unimplemented.

- Adapter classes simplifies coding in creating some graphical user interface programs.
Summary

- Abstract classes
  - Written for the concepts and behaviors that apply to other objects, but are too abstract to make concrete
  - Properties
    - Has at least one abstract method
    - Usefulness lies in it being extended by other classes or other classes being derived from it
    - An abstract class cannot be instantiated
Summary (2)

- Looked at the concept with the following inheritance design

  - **Shape**
    - **Point** (Coordinate X, Coordinate Y)
      - A point is-a shape
    - **Square**
      - A square is-a point
      - **Cube** (Inherits X, Y, and edge from Square; needs no new fields)
    - **A cube is-a square**
Explored the contents of the classes
  - Shape
  - Point
  - Square
  - Cube

Verified its workability with test code TestShapes