

OO programming (Abstract Classes)

Abstract classes are classes that cannot be instantiated. Abstract classes are used to factor out common behavior into a general class. You cannot create objects whose type is abstract but you can have abstract class variables that refer to some extended class object.

Abstract classes may contain abstract methods: methods that have no definition in the original class and must be defined in non-abstract subclasses (pure virtual classes in C++).

```
abstract class Message{
    public abstract void play();//must be overridden
}
class TextMessage extends Message{
    public void play() { System.out.println("Text
                                                Message");
}
}
```

OO programming (Abstract Classes) 2

Rules about abstract classes/methods:

- Any class with an abstract method is automatically abstract and must be declared as such. Also, an abstract class cannot be instantiated.
- A subclass of an abstract class can be instantiated only if it overrides each of the abstract methods of its superclass and provides an implementation for all of them. (Concrete class, as opposed to abstract)
- If a subclass of an abstract class does not implement all the abstract methods it inherits, that subclass is itself abstract.
- static, private and final methods cannot be abstract, since they cannot be overridden by a subclass.
- A class can be declared abstract even though it does not have any abstract methods.

OO programming (Abstract Classes) 3

As mentioned before, the `Object` class is the ultimate ancestor; every class in Java extends this class. You can use a variable or object of type `Object` to refer to any type:

```
Object obj=new Employee("H. Hacker", 10000);
```

But you need to cast this object to the original type to use **Employee** objects capabilities:

```
Employee e=(Employee) obj;
```

The `Object` class has the method `equals` which returns true if two object references **refer** to the same object. For your new classes, you should always override this method so that it tests for object equality not reference equality. For example, for the `Employee` class:

```
public boolean equals(Object obj){
    if(!(obj instanceof Employee)) return
false;

    Employee e= (Employee) obj;
    return name.equals(e.name) &&
salary.equals(e.salary) ...}
```

OO programming (Abstract Classes) 4

Note that the `equals` method for the `String` class is overridden by Java. Another method of `Object` is `toString`, which we saw earlier. Since objects of any type or class can be held in variables of type `Object`, we can use this for generic programming. For example, suppose we need a method that takes an array and a value as parameters and we want the method to return the index of that value in the array:

```
static int find(Object[] ob, Object value)
{
    for (int i=0; i<ob.length; i++)
        if(ob[i].equals(value)) return i;
    return -1;    //not found
}
Employee[] staff=new Employee[10];
Employee e=new Employee("Hacker", 10000);
//...
int n=find(staff, e); //works on any object array
```

OO programming (Interfaces)

Java does not support multiple inheritance. A Java class can extend only one class. But in some situations we need classes that inherit behavior from more than one parent class. The solution to this is using Java **interfaces**.

Many believe that multiple inheritance, in C++, introduces more complexity and work on the part of the programmer than solve problems. For example, it causes complications in dealing with multiple copies of inherited members.

An interface is a class but a class which can only contain abstract methods and constants. It cannot contain any implementation for its methods nor can have any instance fields because an interface is a specification and has no implementation detail. Its methods are implicitly abstract.

Any class that implements an interface must define the interface methods or must itself be an abstract class.

OO programming (Interfaces) 2

A class can implement as many interfaces as it needs. The following example is taken from the textbook:

```
interface Shape {
    public abstract double area();
    public abstract double volume();
    public abstract String getName();
}

class Point implements Shape {
    protected int x, y; // coordinates of the Point
    public Point() { setPoint( 0, 0 ); }
    public Point( int a, int b ) { setPoint( a, b ); }
    public void setPoint( int a, int b ){
        x = a;
        y = b;
    } //see next page
}
```

OO programming (Interfaces) 3

```
public int getX() { return x; }
    public int getY() { return y; }
    public String toString() {
        return "[" + x + ", " + y + "];"
    }
    public double area() { return 0.0; }
    public double volume() { return 0.0; }
    public String getName() { return "Point"; }
}

public class Circle extends Point {
    protected double radius;
    public Circle() {
        setRadius( 0 );
    } //see next page
}
```

OO programming (Interfaces) 4

```
public Circle( double r, int a, int b ){
    super( a, b );
    setRadius( r );
}
public void setRadius( double r ){
    radius = ( r >= 0 ? r : 0 );
}
public double getRadius() {
    return radius;
}
public double area() {
    return Math.PI * radius * radius;
}
//see next page
```


OO programming (Interfaces) 5

```
public String toString(){
    return "Center = " + super.toString() +
        "; Radius = " + radius;
}
public String getName() { return "Circle"; }
}
public class Cylinder extends Circle {
    protected double height; // height of Cylinder
    public Cylinder()        {
        setHeight( 0 );
    }
    public Cylinder( double h, double r, int a, int b) {
        super( r, a, b );
        setHeight( h );
    }
    public void setHeight( double h ){
        height = ( h >= 0 ? h : 0 ); } //see next page
```

OO programming (Interfaces) 6

```
public double getHeight() { return height; }
    public double area(){
        return 2 * super.area() +
            2 * Math.PI * radius * height;
    }
    public double volume() {
        return super.area() * height;
    }
    public String toString(){
        return super.toString() + "; Height = " +
            height;
    }
    public String getName() { return "Cylinder"; }
} //see next page
```

OO programming (Interfaces) 7

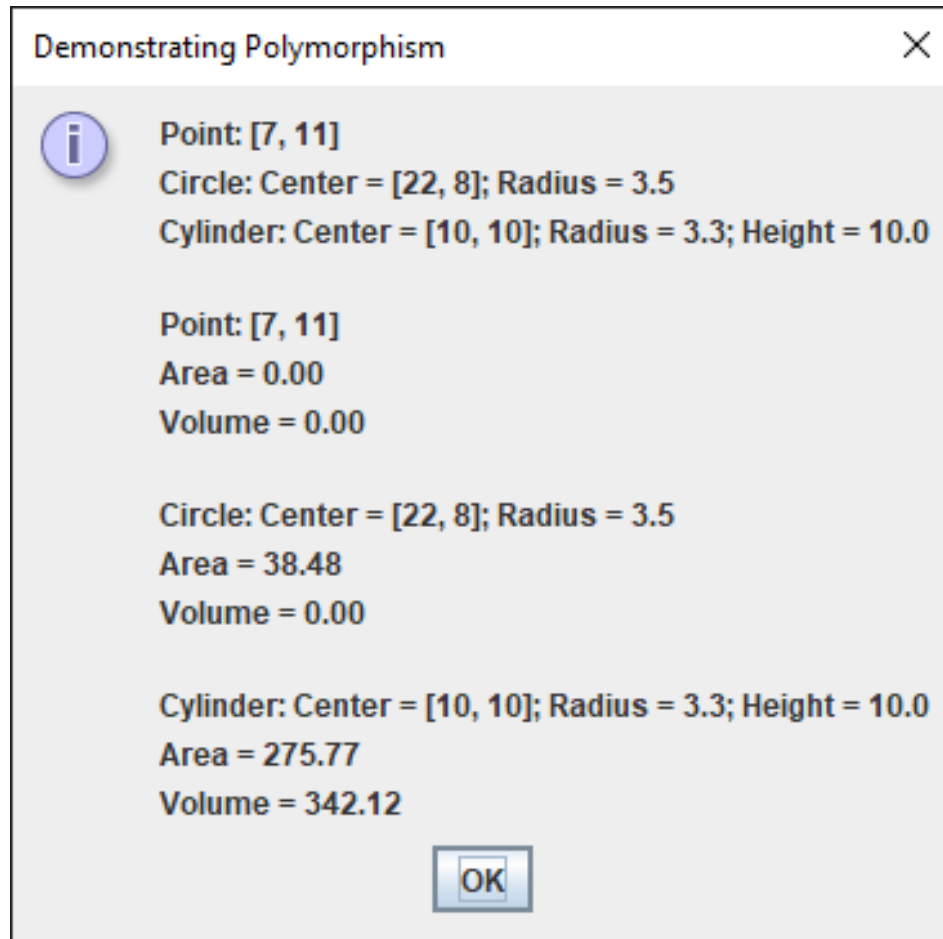
```
import javax.swing.JOptionPane;
import java.text.DecimalFormat;
public class Test {
    public static void main( String args[] ){
        Point point = new Point( 7, 11 );
        Circle circle = new Circle( 3.5, 22, 8 );
        Cylinder cylinder = new Cylinder( 10, 3.3, 10, 10 );
        Shape arrayOfShapes[];
        arrayOfShapes = new Shape[ 3 ];
        arrayOfShapes[ 0 ] = point;
        arrayOfShapes[ 1 ] = circle;
        arrayOfShapes[ 2 ] = cylinder;
        String output =
            point.getName() + ": " + point.toString() + "\n" +
            circle.getName() + ": " + circle.toString() + "\n" +
            cylinder.getName() + ": " + cylinder.toString();
    }
}
```

OO programming (Interfaces) 8

```
DecimalFormat precision2 = new DecimalFormat( "0.00" );
    for ( int i = 0; i < arrayOfShapes.length; i++ ) {
        output += "\n\n" + arrayOfShapes[ i ].getName() + ":
"        + arrayOfShapes[ i ].toString() + "\nArea = " +
            precision2.format( arrayOfShapes[ i ].area() ) +
            "\nVolume = " +
            precision2.format( arrayOfShapes[ i ].volume() );
    }
    JOptionPane.showMessageDialog( null, output,
        "Demonstrating Polymorphism",
        JOptionPane.INFORMATION_MESSAGE ); System.exit( 0 );
}
}
```

This example demonstrates how interfaces are used in a class hierarchy. In C++ this would be achieved by a combination of multiple inheritance and pure virtual functions.

OO programming (Interfaces) 9



Demonstrating Polymorphism

i Point: [7, 11]
Circle: Center = [22, 8]; Radius = 3.5
Cylinder: Center = [10, 10]; Radius = 3.3; Height = 10.0

Point: [7, 11]
Area = 0.00
Volume = 0.00

Circle: Center = [22, 8]; Radius = 3.5
Area = 38.48
Volume = 0.00

Cylinder: Center = [10, 10]; Radius = 3.3; Height = 10.0
Area = 275.77
Volume = 342.12

OK

OO programming (Interfaces) 10

Abstract classes or interfaces: Interfaces are used in place of abstract classes when there is no default implementation and no instance fields. In addition to providing support for multiple inheritance, interfaces are commonly used in GUI programming as will soon see. Usually interfaces are defined in classes of their own with the same name as the interface name and in a `.java` file.

Since they both may contain abstract methods it is not possible to instantiate objects from them and they may not define constructor methods. Interfaces can only contain abstract instance methods and constants whereas abstract classes can contain instance fields and a mixture of abstract and instance methods. If you add a new method to an interface which has already been implemented by some class, you break that subclass. This is not a problem with abstract classes.

An interface defines a public API. All methods of an interface are implicitly `public`, even if the `public` modifier is omitted. Also, it is an error to define `protected` or `private` methods in an interface.

OO programming (Interfaces) 11

Here is another example using interfaces:

```
import java.util.*;
public class Interface1{
    public static void main(String[] l){
        Employee[] staff=new Employee[3];
        staff[0]=new Employee("Harry", 35000, new
            Date(1990,1,2));
        staff[1]=new Employee("Barry", 32000, new
            Date(1992,5,6));
        staff[2]=new Employee("Jerry", 29000, new
            Date(1998,11,2));
        ArrayAlg.Sort(staff);
        for(int i=0; i<staff.length;i++)
            System.out.println(staff[i]);
    }
}
//---->
```

OO programming (Interfaces) 12

```
interface Sortable{
    public int compareTo(Sortable b);
}
class Employee implements Sortable {
    public Employee(String n, double s, Date d)
    {
        name=n;
        salary=s;
        hireDate=d;    }
    public void raiseSalary(double byPercent){
        salary*=1+byPercent/100;
    }
    public String getName(){return name;}
    public double getSalary(){return salary;}
    public String toString(){
        return name + " " + salary + " " + hireYear();
    }
}
```


OO programming (Interfaces) 13

```
public int hireYear()
{
    return hireDate.getYear();
}
public int compareTo(Sortable b)
{
    Employee eb=(Employee) b;
    if(salary<eb.salary) return -1;
    if (salary>eb.salary) return 1;
    return 0;
}
private String name;
private double salary;
private Date hireDate;
}
//---->
```

OO programming (Interfaces) 14

```
class ArrayAlg{
    public static void Sort(Sortable[] a){
        for(int i=0; i<a.length; i++)
        {
            for(int j=0; j<a.length-1; j++)
            {
                if(a[j].compareTo(a[j+1])<0)
                {
                    Sortable temp=a[j];
                    a[j]=a[j+1];
                    a[j+1]=temp;
                }
            }
        }
    }
}
```

OO programming (Interfaces) 15

In this example, the `employee` class implements the `Sortable` interface which has only one method, the `compareTo` method. Any class which implements this interface must override its methods.

The class `ArrayAlg` contains a static method `Sort` which takes an array of objects of any class which implements the `Sortable` interface and sorts the elements of the array in descending order. This example uses the bubble sort algorithm, but you can replace it with any other sorting routine.

This example used an interface; you could have used an abstract class to achieve the same effect. But, if the `employee` class had already inherited (extended) another class, say a class `Person`, then you wouldn't have been able to use abstract classes. Why is this?

Instead of the user-defined `Sortable` interface, you could use the Java library `Comparable` interface, which has a similar definition to our `Sortable` interface.

OO programming (Interfaces) 16

Since interfaces has no default implementation for its methods, adding new methods to the existing interfaces would break its implementing concrete classes.

Java 8 introduced a facility to create default methods inside the interface, that provides a default implementation for a method. So, implementing classes can choose to override or use the default implementation.

```
interface Shape {  
    public default double area()  
    {  
        return 0.0;  
    }  
    public abstract double volume();  
    public abstract String getName();  
}
```