

# IECS273/274 [1111 3670/3671] 物件導向設計與實習

## 07#1 OOP Fundamentals Polymorphism





01

► Polymorphism

# ► Polymorphism

- **Polymorphism** is an object that can be used as an object of various concrete classes. It means *a same operation can behave differently* (be implemented by different methods).
  - Three necessary conditions for polymorphism:
    - ✓ inheritance,
    - ✓ rewrite, and
    - ✓ superclass reference pointing to the subclass object.
  - ***When using polymorphism to call a method, first check whether the method exists in the parent class, if not, it is a compiler error; if so, call the method of the same name in the subclass.***
  - The advantage of polymorphism: it can make the program have good expansion, and can handle all classes of objects in general.

# ► Polymorphism

- Polymorphism in object-oriented programming languages makes it easy to design and implement extensible programs.
  - Can generically handle superclass objects.
  - Ease of adding classes to a hierarchy.
    - ✓ little or no modification to other programs, only the parts of the program that need to be directly related to the new class must be changed
  - Polymorphism has the following advantages:
    - ✓ eliminate coupling relationships between types.
    - ✓ replaceability,
    - ✓ expandability,
    - ✓ interface,
    - ✓ flexibility,
    - ✓ simplify.

Polymorphism.java

```
public class PolymorphismDemo {  
    /** Main method */  
    public static void main(String[] args) {  
        // Display circle and rectangle properties  
        displayObject(new Circle(3, "blue", false));           //Polymorphism  
        displayObject(new Rectangle(4, 6, "red", true));        //Polymorphism  
    }  
  
    /** Display geometric object properties */  
    public static void displayObject(GeometricObject object) {  
        System.out.println("Created on " + object.getDateCreated() +  
            ". Color is " + object.getColor());  
    }  
}
```

Polymorphism.class

Created on Mon Oct 03 18:42:26 EDT 2022. Color is blue  
Created on Mon Oct 03 18:42:26 EDT 2022. Color is red



02

## ► Polymorphism in Java (Abstract Class)

# ► Abstract Class

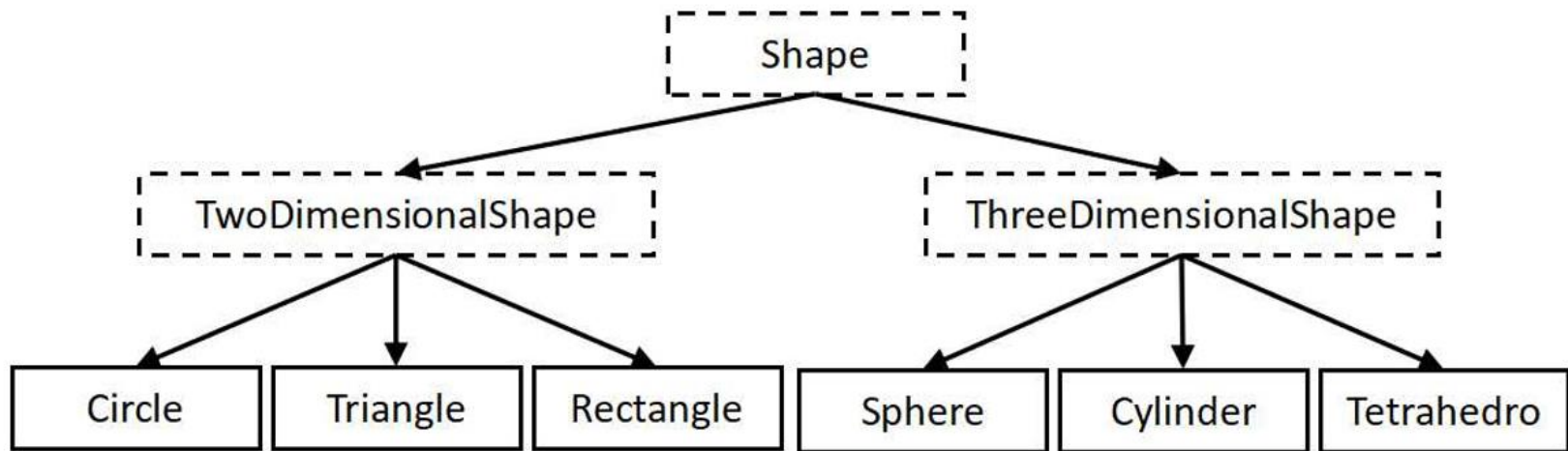
- Java is an object-oriented programming language that supports abstract class/superclass to achieve **polymorphism**.



Syntax: **public abstract class** <ClassName> {...}

- An abstract class cannot be used to create objects. An abstract class can contain abstract methods that are implemented in concrete subclasses.
- **In order to postpone the definition of a method, Java allows an abstract method to be declared.**
  - An abstract method has a heading, but no method body
  - The body of the method is defined in the derived classes
  - The class that contains an abstract method is called an abstract class.
  - An abstract superclass must contain **one or more abstract methods**; an abstract method is a method *with no function code*.
  - Abstract superclasses can also have *data members, constructors, and non-abstract methods*, but **cannot be instantiated**.
  - If you *instantiate* an abstract superclass, a compilation *syntax error* will occur.
  - An abstract superclass can be a **multi-layer hierarchy**, but the subclasses must be **concrete** subclasses that can be substantiated.



# ► Abstract Class



abstract class:   
concrete class: 



# ► Abstract Class

- A class that has at least one abstract method is called an **abstract class**
  - An abstract class must have the modifier **abstract** included in its class heading:
  - An abstract class can have any number of abstract and/or fully defined methods
  - If a derived class of an abstract class adds to or does not define all of the abstract methods, then it is abstract also, and must add abstract to its modifier
- A class that has no abstract methods is called a **concrete class**

```
public abstract class Employee
{
    private instanceVariable;
    ...
    public abstract double getPayment();
    ...
}
```

# ► Abstract Method

- An abstract method is like a placeholder for a method that will be fully defined in a descendent class
- It has a complete method heading, to which has been added the modifier abstract
- It cannot be private
- It has no method body, and ends with a semicolon in place of its body

```
public abstract double getPayment();
```

```
public abstract void doLoop(int count);
```

Shape.java

```
package shapeAbstractClass;
```

```
// Define an abstract base class of shape.
```

```
public abstract class Shape { // An abstract without any data members. // abstract superclass
```

```
    public double area() {return 0.0;} // Non-abstract method. // non-abstract method
```

```
    public double volume() {return 0.0;} // Non-abstract method. // If the subclass is not overloaded, the code here is executed
```

```
    public abstract String getName(); // Abstract method. No function body. // abstract method, empty function body,
```

```
} // cannot execute.
```

Point.java

```
package shapeAbstractClass;
```

```
// A subclass inherits from Shape.
```

```
public class Point extends Shape{  
    protected int x, y; // Coordinates of the point.
```

// Concrete subclass, inheriting the abstract  
// superclass **Shape**.

```
// Point constructor without arguments.
```

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

// two constructors

```
// Point constructor with two integers.
```

```
public Point(int a, int b){  
    setPoint(a, b);  
}
```

```
// Set the coordinate [x, y] of the point.
```

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

Point.java

// Get x coordinate.

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

// Get y coordinate.

**public int** getY() { **return** y; }

// Convert to the output string of point [x, y].

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

// non-abstract method, every subclass has this method

// Return the class name. Implementation of the abstract method.

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

// Overloading abstract methods, each subclass has overloads

Circle.java

```
package shapeAbstractClass;
```

```
// A subclass inherits from Point. Also, a subclass of Shape.
```

```
public class Circle extends Point {
```

// Concrete subclass, inheriting the superclass **Point**,  
// and also indirectly inheriting the abstract  
// superclass **Shape**.

```
    protected double radius; // Radius of circle. Data member in circle.
```

```
    // Constructor without argument.
```

```
public Circle() {
```

```
    // Implicitly call superclass constructor here.
```

```
    setRadius(0.0); // Set radius to 0.
```

```
}
```

// two constructors

```
public Circle(double r, int a, int b) {
```

```
    super(a, b); // Explicit call to the superclass constructor.
```

```
    setRadius(r); // Set radius.
```

```
}
```

```
// Set the value to radius.
```

```
public void setRadius(double r) {
```

```
    radius = (r >= 0.0 ? r : 0.0);
```

```
}
```

## Circle.java

```
// Get radius of the circle.
public double getRadius() { return radius; }

// Calculate area of the circle.
public double area() { return Math.PI * radius * radius; }

// Convert to the output string of circle of center and radius.
// Override method in the superclass.
public String toString(){ // non-abstract method, every subclass has this method
    return "Center = " + super.toString() + "; Radius = " + radius; // Invoke a method of the superclass
}

// Return the class name. Implementation of the abstract method.
public String getName(){return "Circle";} // Overloading abstract methods, each subclass has overloads
}
```



## Cylinder.java

```
package shapeAbstractClass;
```

```
// A subclass inherits from Circle. Also, a subclass of Shape and Point.
```

```
public class Cylinder extends Circle {    // Concrete subclass, inherits superclass Circle, also indirectly subclass Point  
    protected double height; // Height of cylinder. Data member in cylinder.  
                                   // and inherits abstract superclass Shape.
```

```
// Constructor without argument.
```

```
public Cylinder() {  
    // Implicitly call superclass constructor here.  
    setHeight(0.0); // Set height to 0.  
}
```

// two constructors

```
public Cylinder(double h, double r, int a, int b) {  
    super(r, a, b); // Explicit call to the superclass constructor.  
    setHeight(h); // Set height.  
}
```

```
// Set the value to height.
```

```
public void setHeight(double h) {  
    height = (h >= 0.0 ? h : 0.0);  
}
```

## Cylinder.java

```
// Get height of the cylinder.
public double getHeight() { return height; }

// Calculate surface area of cylinder.
public double area() {
    return 2 * super.area() + 2 * Math.PI * radius * height;
}

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

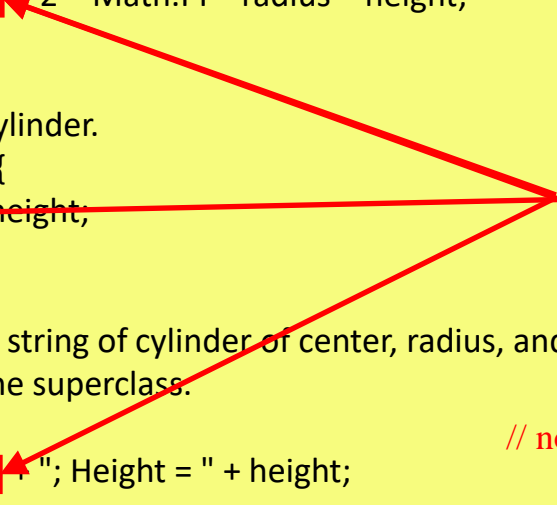
// Convert to the output string of cylinder of center, radius, and height.
// Override method in the superclass.
public String toString() {
    return super.toString() + "; Height = " + height;
}

// Return the class name. Implementation of the abstract method.
public String getName() {return "Cylinder";}
}
```

**// Invoke a method of the superclass**

**// non-abstract method, every subclass has this method**

**// Overloading abstract methods, each subclass has overloads**



## ShapeAbstractClass.java

```
package shapeAbstractClass;
```

```
// Test of abstract class.
```

```
public class ShapeAbstractClass {
```

```
public static void main(String[] args) {
```

```
Point point = new Point(10, 20); // Create a point object.
```

```
Circle circle = new Circle(5.6, 50, 50); // Create a circle object.
```

```
// Create objects of three subclasses
```

```
Cylinder cylinder = new Cylinder(15.8, 6.2, 40, 40); // Create a cylinder object.
```

```
Shape arrayOfShapes[] = new Shape[3]; // Create an array of three shapes. // Abstract class references an array of
// subclass objects
```

```
arrayOfShapes[0] = point; // Refer to a point shape.
```

```
arrayOfShapes[1] = circle; // Refer to a circle shape.
```

```
// Each array element references a subclass object
```

```
arrayOfShapes[2] = cylinder; // Refer to a cylinder shape.
```

```
// Print the point data.
```

[illegible]

```
// Print the circle data.
```

[illegible]

## ShapeAbstractClass.java

```
// Print the cylinder data. // Print the subclass Cylinder name and data
System.out.println(cylinder.getName() + ": " + cylinder.toString()); // directly through the subclass object
System.out.println("\n-----\n");

// Iterate the three shapes and print the shape data, area, and volume of each shape.
for (int i=0; i<3; i++) { // The abstract method is called, and the concrete method of the subclass can be determined at
    System.out.printf("%s: %s, area=%4.2f, volume=%4.2f\n", arrayOfShapes[i].getName(), // execution time
        arrayOfShapes[i].toString(), arrayOfShapes[i].area(), arrayOfShapes[i].volume())
    }
    // Calls a non-abstract method; if the method is not overridden in the subclass, executes
    // the method in the superclass Shape.
}
```

ShapeAbstractClass.class

Point: [10, 20]

Circle: Center = [50, 50]; Radius = 5.6

Cylinder: Center = [40, 40]; Radius = 6.2; Height = 15.8

-----

Point: [10, 20], area=0.00, volume=0.00

Circle: Center = [50, 50]; Radius = 5.6, area=98.52, volume=0.00

Cylinder: Center = [40, 40]; Radius = 6.2; Height = 15.8, area=857.03, volume=1908.05

# ► Static Binding

## ■ static binding (early binding)

- which method is to be called is decided **at compile time**
- **overloading**: an invocation can be operated on arguments of more than one type
- Java uses static binding with **private, final, and static methods**
  - ✓ In the case of private and final methods, late binding would serve no purpose
  - ✓ However, in the case of a static method invoked using a calling object, it does make a difference

```
public class GoodMorning {  
    public String GoodMorning(String name){  
        return "Good Morning! "+ name;  
    }  
    public String GoodMorning(String name, String gender){  
        if(gender.equals("man")){  
            return " Good Morning! Mr. "+ name;  
        }  
        else if(gender.equals("woman")){  
            return " Good Morning! Miss. "+ name;  
        } else {  
            return " Good Morning! "+ name;  
        }  
    }  
    public static void main(String[] args){  
        GoodMorning hello = new GoodMorning();  
        System.out.println(hello.GoodMorning("Brien")); //decided at compile time  
        System.out.println(hello.GoodMorning("Brien.", "man")); //decided at compile time  
    }  
}
```



# ► Dynamic Binding

- Object-oriented programming languages cannot instantiate abstract superclasses, and a superclass variable does not correspond to an instance object of the superclass; a superclass variable **refers** to an instance object of a subclass.
  - Therefore, at compile time, the compiler cannot determine from the code which subclass instance the superclass variable refers to; that is to say, at compile time, the compiler *cannot determine which subclass corresponds to the abstract method to be executed*.
  - The method corresponding to the subclass cannot determine the instance object of the subclass referenced by the superclass variable *until it is executed*, and then execute the method corresponding to the subclass.
  - That is, the binding of abstract methods is determined at execution time, a phenomenon called **dynamic method binding**.

# ► Dynamic Binding

## ■ dynamic binding (late binding)

- which method is to be called is decided **at run time**
- **overriding**: a derived class inherits methods from the base class, it can change or override an inherited method

```
public class CreditCardPayment extends Payment{
    public void checkout() {
        System.out.println("Checkout with credit card");
    }
}

public class Store {
    public static void main(String[] args) {
        Payment p1 = new Payment();
        p1.checkout(); //decided at run time
        Payment p2 = new CreditCardPayment();
        p2.checkout(); //decided at run time
    }
}
```

```
public class DynamicBinding {  
    public static void main(String[] args) {  
        demo(new GraduateStudent());  
        demo(new Student());  
        demo(new Person());  
        demo(new Object());  
    }  
    public static void demo(Object x) {  
        System.out.println(x.toString());  
    }  
}  
class GraduateStudent extends Student {  
}  
class Student extends Person {  
    public String toString() {  
        return "Student";  
    }  
}  
class Person extends Object {  
    public String toString() {  
        return "Person";  
    }  
}
```

dynamicBinding.class

Student

Student

Person

java.lang.Object@

# ► final Variables, Methods, and Classes

- Neither a final class nor a final method can be extended. A final data field is a constant.
- Java programming language can prefix variables, methods, and classes with the modifier **final**.
  - **final variables**: after the variable is defined, it cannot be modified.
    - ✓ The value of **final** variables must be initialized at definition time.
    - ✓ Similar to constant identifiers in the C programming language.
  - **final methods**: overloading is not available in subclasses.
    - ✓ **static** methods and **private** methods are implicit **final** methods.
    - ✓ Programs can *inline* final methods
      - *Inlining* is the code that replaces the method call directly with the method. (Like a *macro function*).
  - **final classes**: cannot be used as superclasses, that is, cannot be inherited.
    - ✓ Methods in a **final** class are implicitly **final**.



03

► Interface in Java

# ► *Interface* in Java

- An interface is a class-like construct for defining common operations for objects.
- An interface is something like an extreme case of an abstract class
  - **an interface is not a class**
  - It is a type that can be satisfied by any class that implements the interface
- The syntax for defining an interface is similar to that of defining a class except the word interface is used in place of class
- An interface specifies a set of methods that any class that implements the interface must have
  - It contains method headings and constant definitions only
    - ✓ Any variables defined in an interface must be public, static, and final
  - It contains no instance variables nor any complete method definitions

```
public interface Shape {  
    int color = 1; // => public static final int color = 1;  
}  
public class Paint {  
    public static void main(String[] args) {  
        System.out.println(Shape.color);  
    }  
}
```



# ► Interface in Java

- An abstract class in the Java programming language can use another keyword **interface** instead of **abstract**. Syntax:

```
public interface <interfaceName> [extends <interfaceName1>
                                   {<interfaceName2>}] {
    public abstract <returnType> <methodName> (<parameterList>);
    {public abstract <returnType> <methodName> (<parameterList>);}
    {public static final <type> <dataName> = <dataValue>;}
}
```

- All methods in an **interface** must be **abstract methods**.
- An **interfaces** can have data members, but they must be **static final** members.
- Implement an **interface** using the keyword **implements**. Syntax:

```
public class <className> [extend <className>]
    implements <interfaceName> {, <interfaceName>} {...}
```

# ► **Interface in Java**

- An **interface** is a replacement for an **abstract** class without a default method implementation.
- A class can implement **multiple interfaces** at the same time, and the use of interfaces allows for *multiple inheritance*.
- After the keyword **implements**, list multiple interface names separated by commas ','
- The interface should be in a single file, and the file name and interface name should be the **same**.

# ► *Interface* in Java

- All methods in an interface are implicitly public and abstract, so you can omit the public modifier.
  - They cannot be given private or protected.

```
public interface ISpecs {  
    private void run(); //Not allowed  
    protected void run(); //Not allowed  
    void run(); // Allowed. Equal to the following definition  
    public abstract void run(); //Allowed  
}  
  
public interface Shape {  
    int color = 1; // => public static final int color = 1;  
    public abstract double area(); //=> double area();  
}
```

# ► *Interface* in Java

- To implement an interface, a concrete class must do two things:
  - ① implements Interface\_Name
  - ② the class must implement all the method headings listed in the definition(s) of the interface(s).
- A class can implement multiple interfaces, but it can only extend one superclass.
- Multiple inheritance is not allowed in Java. Instead, Java's way of approximating multiple inheritance is through interfaces.

```
public class ConcreteClass implements ISpec1, ISpec2, ISpec3{  
    ...  
}
```

Shape.java

```
package shapeInterface;
```

```
public interface Shape { // An interface for Shape, no method implementation.  
    public abstract double area(); // Abstract method for computing area.  
    public abstract double volume(); // Abstract method for computing volume.  
    public abstract String getName(); // Abstract method for getting shape's name.  
}
```

Redefined as an **interface**,  
not an **abstract** class.

All are **abstract** methods.

Point.java

```
package shapeInterface;

//A subclass implements Shape interface.
public class Point implements Shape {
    protected int x, y; // Coordinates of the point.

    // Point constructor without arguments.
    public Point() {
        setPoint(0, 0);
    }

    // Point constructor with two integers.
    public Point(int a, int b) {
        setPoint(a, b);
    }

    // Set the coordinate [x, y] of the point.
    public void setPoint(int a, int b) {
        x = a;
        y = b;
    }
}
```

**Implement** interfaces, alternative inheritance.

```
// Get x coordinate.  
public int getX() { return x; }  
  
// Get y coordinate.  
public int getY() { return y; }  
  
// Convert to the output string of point [x, y].  
public String toString() {  
    return "[" + x + ", " + y + "];"  
}  
  
// Implementation of the abstract method.  
public double area() {return 0.0;}  
  
// Implementation of the abstract method.  
public double volume() {return 0.0;}  
  
// Return the class name. Implementation of the abstract method.  
public String getName() {return "Point";}   
}
```

## Circle.java

```
package shapeInterface;

//A subclass inherits from Point. Also, a subclass implements Shape interface.
public class Circle extends Point {
    protected double radius; // Radius of circle. Data member in circle.

    // Constructor without argument.
    public Circle() {
        // Implicitly call superclass constructor here.
        setRadius(0.0); // Set radius to 0.
    }

    public Circle(double r, int a, int b) {
        super(a, b); // Explicit call to the superclass constructor.
        setRadius(r); // Set radius.
    }

    // Set the value to radius.
    public void setRadius(double r) {
        radius = (r>=0.0 ? r : 0.0);
    }
}
```



## Circle.java

```
// Get radius of the circle.  
public double getRadius() { return radius; }  
  
// Calculate area of the circle.  
public double area() { return Math.PI * radius * radius; }  
  
// Convert to the output string of circle of center and radius.  
// Override method in the superclass.  
public String toString() {  
    return "Center = " + super.toString() + "; Radius = " + radius;  
}  
  
// Return the class name. Implementation of the abstract method.  
public String getName() {return "Circle";}  
}
```

## Cylinder.java

```
package shapeInterface;
```

```
//A subclass inherits from Point and Circle. Also, a subclass implements Shape interface.
```

```
public class Cylinder extends Circle {
```

```
    protected double height; // Height of cylinder. Data member in cylinder.
```

```
    // Constructor without argument.
```

```
    public Cylinder() {
```

```
        // Implicitly call superclass constructor here.
```

```
        setHeight(0.0); // Set height to 0.
```

```
    }
```

```
    public Cylinder(double h, double r, int a, int b) {
```

```
        super(r, a, b); // Explicit call to the superclass constructor.
```

```
        setHeight(h); // Set height.
```

```
    }
```

```
    // Set the value to height.
```

```
    public void setHeight(double h) {
```

```
        height = (h>=0.0 ? h : 0.0);
```

```
    }
```

## Cylinder.java

```
// Get height of the cylinder.  
public double getHeight() { return height; }  
  
// Calculate surface area of cylinder.  
public double area() {  
    return 2 * super.area() + 2 * Math.PI * radius * height;  
}  
  
// Calculate volume of cylinder.  
public double volume() {  
    return super.area() * height;  
}  
  
// Convert to the output string of cylinder of center, radius, and height.  
// Override method in the superclass.  
public String toString() {  
    return super.toString() + "; Height = " + height;  
}  
  
// Return the class name. Implementation of the abstract method.  
public String getName() {return "Cylinder";}  
}
```

## ShapeInterface.java

```
package shapeInterface;

public class ShapeInterface {

    // Test of interface.
    public static void main(String[] args) {
        Point point = new Point(5, 5); // Create a point object.
        Circle circle = new Circle(8.4, 20, 30); // Create a circle object.
        Cylinder cylinder = new Cylinder(14.5, 8.2, 15, 25); // Create a cylinder object.

        Shape arrayOfShapes[] = new Shape[3]; // Create an array of three shapes.

        arrayOfShapes[0] = point; // Refer to a point shape.
        arrayOfShapes[1] = circle; // Refer to a circle shape.
        arrayOfShapes[2] = cylinder; // Refer to a cylinder shape.

        // Print the point data.
        System.out.println(point.getName() + ": " + point.toString());

        // Print the circle data.
        System.out.println(circle.getName() + ": " + circle.toString());
    }
}
```

## ShapeInterface.java

```
// Print the cylinder data.
System.out.println(cylinder.getName() + ": " + cylinder.toString());
System.out.println("\n-----\n");

// Iterate the three shapes and print the shape data, area, and volume of each shape.
for (int i=0; i<3; i++) {
    System.out.printf("%s: %s, area=%4.2f, volume=%4.2f\n", arrayOfShapes[i].getName(),
        arrayOfShapes[i].toString(), arrayOfShapes[i].area(), arrayOfShapes[i].volume());
}
}
```

ShapeInterface.class

Point: [5, 5]

Circle: Center = [20, 30]; Radius = 8.4

Cylinder: Center = [15, 25]; Radius = 8.2; Height = 14.5

-----

Point: [5, 5], area=0.00, volume=0.00

Circle: Center = [20, 30]; Radius = 8.4, area=221.67, volume=0.00

Cylinder: Center = [15, 25]; Radius = 8.2; Height = 14.5, area=1169.55, volume=3062.99



# 04 ▶ Programming Practice

# ► Practice 1 : *Employees Salary*

*A company pays its employees monthly. The employees are of three types:*

- ✓ *Salaried employees are paid a fixed monthly salary,*
- ✓ *Commission employees are paid a percentage of their sales, and*
- ✓ *Base-salary-plus-commission employees receive a base salary plus a percentage of their sales.*

*For the current pay period, the company has decided to reward base-salary-plus-commission employees by adding 10 percent to their base salaries.*

*Please implement a Java console application that performs its payroll calculations polymorphically.*



## ► Practice 1 : *Employees Salary*

Please use *abstract class Employee* to represent the general concept of an employee.

The classes that derive directly from *Employee* are *SalariedEmployee* and *CommissionEmployee*. Class *BasePlusCommissionEmployee*—derived from *CommissionEmployee*—represents the last employee type.

Abstract base class *Employee* declares the “interface” to the hierarchy—that is, the set of member functions that a program can invoke on all *Employee* objects. Each employee, regardless of the way his or her earnings are calculated, has a first name, a last name and a social security number, so private data members *firstName*, *lastName* and *socialSecurityNumber* appear in abstract base class *Employee*.

## ▶ Practice 2 : *Comparable Interface*

Write a Java program ScoreSorter.java to sort the score rank in the class using Array.sort.

There are three data fields in Score class.

They are Chinese score, English score, and Mathematics score respectively. The rank priority is by total score, by Chinese score, by English score, and by mathematics score.

Please implement the Comparable interface to handle this sorting problem.



# 05 ▶ Assignment #3

# ► Assignment #3: Vending Machine == Team

Suppose a vending machine sells the following eight products:

- A: Coca Cola, unit price US \$1.65
- B: Minute Maid Orange Juice, unit price US \$3.50
- C: Evian Mineral Water, unit price US \$2.80
- D: M&M's Chocolate, unit price US \$1.50
- E: Hershey's Chocolate Bar, unit price US \$1.85
- F: Oreo Cookies, unit price US \$1.00
- G: Doritos Tortilla Chips, unit price US \$3.25
- H: Pringles Potato Chips, unit price US \$3.40

*Write a Java program to define a vending machine to simulate this vending machine with the following rules.*

The vending machine starts with the five operational buttons as below:

- a. Deposit bill(s) or coin(s),
- b. Select product(s),
- c. Cancel a product,
- d. Purchase product(s),
- e. Abort transaction.

# ► Assignment #3: Vending Machine

1. When button a is selected, the vending machine enters the money deposit state and the customer can deposit either bills or coins. Before each money deposit, enter B for bill deposit and C for coin deposit.
  - ① When the money deposit is B, the customer can deposit a 1-dollar, 5-dollar, 10-dollar, or 20-dollar bill by entering integer 1, 5, 10, or 20 to represent the value of a bill, respectively. The money deposit state ends when the value is 0. If the integer value is a number other than 0, 1, 5, 10, and 20, the money deposit is ignored, the vending machine stays in the money deposit state.
  - ② When the money deposit is C, the customer can deposit a nickel (5 cents), a dime (10 cents), or a quarter (25 cents) using integer 5, 10, or 25 to represent the value of a coin, respectively. The money deposit state ends when the value is 0. If the integer value is a number other than 0, 5, 10, and 25, the money deposit is ignored and the vending machine stays in the money deposit state.
  - ③ If the money deposit is neither B nor C, the input is ignored and the money deposit state ends. The vending machine is back to wait for the next operational mode.
2. When button b is selected, the vending machine enters the product selection state and the customer can select a product by entering one of characters A, B, C, D, E, F, G, and H as described above. In state b, a customer may select multiple items until he/she enters Q to quit; if any of the other characters is entered, it is ignored.
3. When button c is selected, the customer can cancel a selected product item by entering one of A, B, C, D, E, F, G, and H. Each time a button c is selected, only one item can be canceled. If any of the other characters is entered, it is ignored. When a product is canceled, its quantity in the current transaction must be greater than 0 and the quantity is decreased by 1; otherwise, the cancellation has no effect.

# ► Assignment #3: Vending Machine

4. When button d is selected, the total price of the selected products is calculated.
  - ① If the deposited money is greater than or equal to the total price, print the selected products and quantities to simulate the vending machine dropping the selected products and return the changes to the customer. The returned changes contain coins only and must be the least number of coins. You may assume the vending machine has sufficient product inventory, bills, and coins. A transaction is completed when the changes are returned. The vending machine may begin a new transaction.
  - ② If the deposited money is less than the total price, show a message "Insufficient Deposit" and enter state a to deposit additional money.
5. When button e is selected, the current transaction is canceled, return all the deposit money to customer. Assume the vending machine can return both bills and coins. Show the value and the number for each bill or coin deposited by the customer and the total amount returned. When the transaction is canceled, the deposited money is returned. The vending machine may begin a new transaction.

**At the end of each of the above steps, show the current transaction status including the total deposit amount, the name of each selected product with its unit price, and the total price of the selected products.**