CS2030 Lecture 4

Abstract Class and Interface

Henry Chia (hchia@comp.nus.edu.sg)

Semester 1 2021 / 2022

Lecture Outline and Learning Outcomes

- Know when to define a **concrete class**, and when to define an **abstract class**
- Know how to define and implement an **interface**
- Understand when to use inheritance and/or interfaces
- Understand how inheritance and interfaces can both support polymorphism
- Demonstrate the application of SOLID principles in the design of object-oriented software

Adding More Shapes

- Suppose we would like to design a `Rectangle` class, as well as a `Circle` class

```
jshell> new Circle(1.0, Color.BLUE)
$.. ==> area 3.14; perimeter 6.28; java.awt.Color[r=0,g=0,b=255]

jshell> new Rectangle(8.9, 1.2, Color.GREEN)
$.. ==> area 10.68; perimeter 20.20; java.awt.Color[r=0,g=255,b=0]
```

- Some design considerations
  - circle has a radius
  - rectangle has a width and a height
  - shapes have color
  - able to compute area and perimeter of each shape
- Since both `Rectangle` and `Circle` are shapes with color, define a `FilledShape` class as the parent of these two classes

"Inheriting" from `FilledShape`

- Some implementation considerations:
  - `Circle` and `Rectangle` have different dimension properties
  - `Circle` and `Rectangle` have a common color property
  - both `Circle` and `Rectangle` must provide `getArea()` and `getPerimeter()` methods, although computed differently
- Redefine the `Circle` and `Rectangle` classes so that it now extends from `FilledShape`
- How to ensure that `Circle` and `Rectangle` must have `getArea` and `getPerimeter` methods?
  - define `getArea` and `getPerimeter` in `FilledShape` and have them overridden in `Circle` and `Rectangle`
  - how should the methods be implemented in `FilledShape`?

Design #1: `FilledShape` as a Concrete Class

```
class FilledShape {
    private final Color color;

    FilledShape(Color color) {
        this.color = color;
    }

    double getArea() { return -1.0; }
    double getPerimeter() { return -1.0; }
}
```

```
class Circle extends FilledShape {
    private final double radius;

    Circle(double radius, Color color) {
        super(color);
        this.radius = radius;
    }

    @Override
    double getArea() {
        return Math.PI * radius * radius;
    }

    @Override
    double getPerimeter() {
        return 2.0 * Math.PI * radius;
    }
}
```

```
class Rectangle extends FilledShape {
    private final double width;
    private final double height;

    Rectangle(double width, double height, Color color)
        super(color);
        this.width = width;
        this.height = height;
    }

    @Override
    double getArea() {
        return width * height;
    }

    @Override
    double getPerimeter() {
        return 2.0 * (width + height);
    }
}
```

Design #2: `FilledShape` as an Abstract Class

- Does not make sense to instantiate a `FilledShape` object!

```
jshell> new FilledShape(Color.BLUE).getArea()
$.. ==> -1.0
```

- Redefine `FilledShape` as an **abstract** class with abstract methods; these will be implemented in the child classes

```
abstract class FilledShape {
    private final Color color;

    FilledShape(Color color) {
        this.color = color;
    }

    abstract double getArea();
    abstract double getPerimeter();
}
```

```
jshell> new FilledShape(Color.BLUE)
|  Error:
|  FilledShape is abstract; cannot be instantiated
|  new FilledShape(Color.BLUE)
|  ^-------------------------^
```

Design #2: `FilledShape` as an Abstract Class

- Properties and method/constructor implementations can be included in an abstract class to be inherited by the subclasses

```
abstract class FilledShape {
    private final Color color; // property

    protected FilledShape(Color color) { // constructor
        this.color = color;
    }

    abstract double getArea(); // abstract method
    abstract double getPerimeter();

    @Override
    public String toString() { // concrete method
        return "area " + String.format("%.2f", this.getArea()) +
            "; perimeter " + String.format("%.2f", this.getPerimeter()) +
            "; " + this.color;
    }
}

jshell> new Circle(1.0, Color.BLUE)
$.. ==> area 3.14; perimeter 6.28; java.awt.Color[r=0,g=0,b=255]
```

Inheriting from Multiple Parents?

- Define another abstract class `Scalable` to scale a shape

```
abstract class Scalable {
    abstract Scalable scale(double factor);
}
```

- But a child class *can only inherit from one parent class!*

```
jshell> class Circle extends FilledShape, Scalable { }
|  Error:
|  '{' expected
|  class Circle extends FilledShape, Scalable { }
```

- Java prohibits multiple inheritance to avoid the creation of *weird* objects, e.g. `class Spork extends Spoon, Fork`
  - if parents have the same method signature but different implementation, which method should the child invoke?

Defining an Interface as a Contract

- Even though a class can only inherit from one parent class, a class **can implement one or more interfaces**
- Each interface is a contract to that specifies methods to be defined in the implementation class
- Just like abstract classes, interfaces cannot be instantiated
- Implementing the `Scalable` interface

```
class Circle extends FilledShape implements Scalable {
    ...
    @Override
    public Circle scale(double factor) {
        return new Circle(this.radius * factor, super.color);
    }
```

  - Note return type and access modifier of `Circle::scale`

- Methods in interfaces are implicitly **`public`**

Implementing Multiple Interfaces

```
interface Shape {
    double getArea();
    double getPerimeter();
}
interface Scalable {
    Scalable scale(double factor);
}
class Circle implements Shape, Scalable { // multiple implementation
    private final double radius;

    Circle(double radius) {
        this.radius = radius;
    }
    @Override
    public double getArea() {
        return Math.PI * this.radius * this.radius;
    }
    @Override
    public double getPerimeter() {
        return 2 * Math.PI * this.radius;
    }
    @Override
    public Circle scale(double factor) {
        return new Circle(this.radius * factor);
    }
    public String toString() {
        return "Area " +  String.format("%.2f", getArea()) +
            " and perimeter " + String.format("%.2f", getPerimeter());
    }
}
```

Polymorphism Revisited

- Abstract classes and interfaces also support polymorphism

```
jshell> Shape[] shapes = {new Circle(1.0), new Rectangle(2.0, 3.0)}
shapes ==> Shape[2] { Circle@14acaea5, Rectangle@46d56d67 }

jshell> for (Shape s : shapes) System.out.println(s)
area 3.14; perimeter 6.28
area 6.00; perimeter 10.00
```

- **Open-Closed Principle** — Bertrand Meyer
  - "*open for extension, but closed for modification*"
  - we *can extend* a new shape (say `Square`) *without modifying* the client's implementation

```
jshell> /open Square.java
jshell> Shape[] shapes = {new Circle(1), new Rectangle(2, 3), new Square(4)}
shapes ==> Shape[3] { Circle@d8355a8, Rectangle@59fa1d9b, Square@28d25987 }

jshell> for (Shape s : shapes) System.out.println(s)
area 3.14; perimeter 6.28
area 6.00; perimeter 10.00
area 16.00; perimeter 16.00
```

From Concrete Class to Interfaces

- Difference between concrete, abstract classes and interface:
  - **concrete class** is the actual implementation
  - **interface** is a contract specifying the abstraction between
    - what the client can use, and
    - what the implementer should provide
  - **abstract class** is a trade off between the two, i.e. partial implementation of the contract
    - typically used as a base class
- *"Impure" interfaces*...
  - Since Java 8, default methods with implementations can be included into interfaces

SOLID Principles in OO Design

- **S**ingle Responsiblity Principle

  A class should have only one reason to change.
  — Robert C. Martin (aka Uncle Bob)

- **O**pen-Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle

  no client should be forced to depend on methods it does not use. — Uncle Bob

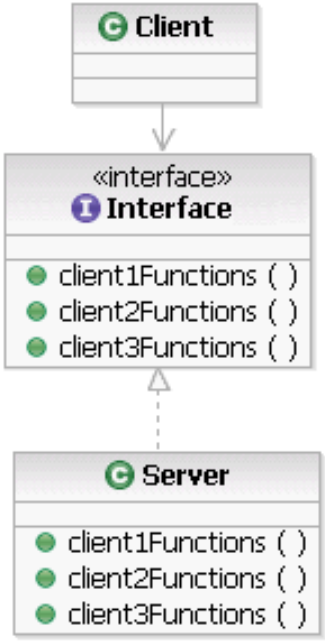
- **D**ependency Inversion Principle

  High-level modules should not depend on low-level modules. Both should depend on abstractions.
  Abstractions should not depend on details. Details should depend on abstractions. — Uncle Bob

Dependency Inversion Principle

- *Program to an interface, not an implementation — GoF*

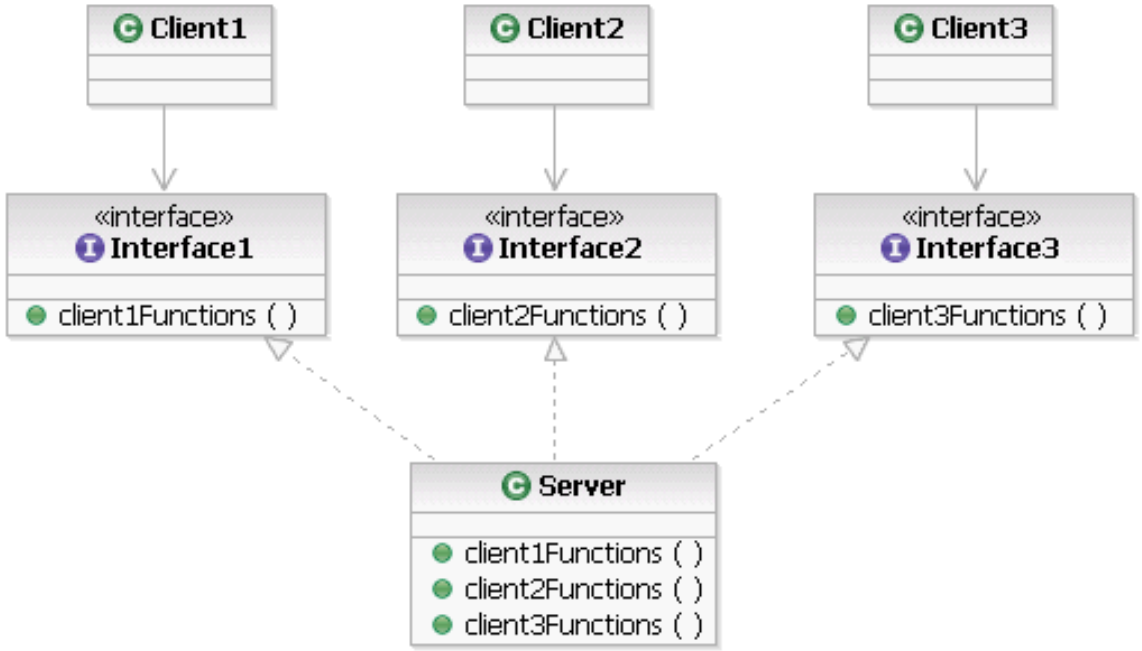
```
jshell> /list Shape

   1 : interface Shape { // the contract
           double getArea();
           double getPerimeter();
       }

jshell> Shape s = new Circle(1.0)
s ==> Area 3.14 and perimeter 6.28

jshell> s.scale() // client cannot use scale
|  Error:
|  cannot find symbol
|    symbol:   method scale(double)
|  s.scale(2.0)
|  ^------^

jshell> class Circle implements Shape { // implementer must implment getArea
   ...> private final double radius;
   ...> public double getPerimeter() {
   ...> return 2.0 * Math.PI * this.radius;
   ...> }}
|  Error:
|  Circle is not abstract and does not override abstract method getArea() in Shape
|  class Circle implements Shape {
|  ^-----------------------------...
```



Interface Segregation Principle

- Clients should not know of methods they don't need

```
jshell> Circle c = new Circle(1.0)
c ==> Area 3.14 and perimeter 6.28

jshell> Shape s = c
s ==> Area 3.14 and perimeter 6.28

jshell> s.getArea()
$.. ==> 3.141592653589793

jshell> s.scale(0.5)
|  Error:
|  cannot find symbol
|    symbol:   method scale(double)
|  s.scale(0.5)
|  ^-----^

jshell> Scalable k = c
k ==> Area 3.14 and perimeter 6.28

jshell> k.scale(0.5)
$.. ==> Area 0.79 and perimeter 3.14

jshell> k.getArea()
|  Error:
|  cannot find symbol
|    symbol:   method getArea()
|  k.getArea()
|  ^-------^
```



"Sub-classing" Arrays

- Since `Circle` is a sub-class (sub-type) of `Shape`, `Circle[]` is also a sub-type of `Shape[]`
  - Arrays are covariant *(variance of types covered later...)*

```
jshell> Circle[] circles = {new Circle(1.0), new Circle(2.0)}
circles ==> Circle[2] { Circle@59fa1d9b, Circle@28d25987 }

jshell> Shape[] shapes = circles
shapes ==> Circle[2] { Circle@59fa1d9b, Circle@28d25987 }
```

- Caution!! May lead to heap pollution

```
jshell> shapes[0] = new Rectangle(2.0, 3.0)
|  java.lang.ArrayStoreException thrown: REPL.$JShell$14$Rectangle
|        at (#8:1)
```

- Above assignment still allows the program to compile, but an `ArrayStoreException` is thrown during run-time
- *Make an array immutable?*