CS2030 Lecture 4

Interface as an Abstraction Barrier

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

Semester 2 2019 / 2020

Lecture Outline

- Abstract class
- Interface
- Polymorphism revisited
- OO design principles
  - Single Responsibility Principle
  - Open-Closed Principle
  - Liskov-Substitution Principle
  - Interface Segregation Principle
  - Dependency Inversion Principle
- Preventing inheritance and overriding

Adding More Shapes

- Suppose we would like to create a rectangle, in addition to the `Circle` class that we have developed previously

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

jshell> new Rectangle(8.9, 1.2)
$.. ==> Area 10.68 and perimeter 20.20
```

- Some design considerations for the `Rectangle` class
  - a rectangle has a width and a height
  - obtain the area and perimeter from a rectangle
- Since both `Rectangle` and `Circle` are shapes, define a `Shape` class as the parent of these two classes

“Inheriting” from `Shape`

- Some considerations:
  - `Circle` and `Rectangle` have different properties
  - 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 `Shape`
- How to ensure that `Circle` and `Rectangle` must have `getArea` and `getPerimeter` methods?
  - define `getArea` and `getPerimeter` in `Shape` and have them overridden in `Circle` and `Rectangle`
  - how should the methods be implemented in `Shape`?

Design #1: **Shape** as a Concrete Class

```java
class Shape {
    double getArea() { return -1; }
    double getPerimeter() { return -1; }
}
```

```java
class Circle extends Shape {
    private final double radius;

    Circle(double radius) {
        this.radius = radius;
    }

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

    @Override
    double getPerimeter() {
        return 2 * Math.PI * radius;
    }
}
```

```java
class Rectangle extends Shape {
    private final double width;
    private final double height;

    Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

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

    @Override
    double getPerimeter() {
        return 2 * (width + height);
    }
}
```

Design #2: `Shape` as an Abstract Class

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

```
jshell> new Shape()
$.. ==> Area -1.00 and perimeter -1.00
```

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

```
abstract class Shape {
    abstract double getArea();
    abstract double getPerimeter();
}
```

```
jshell> new Shape()
|  Error:
|  Shape is abstract; cannot be instantiated
|  new Shape()
|  ^---------^
```

Design #2: `Shape` as an Abstract Class

- Method implementations can be included within an abstract class to be inherited by the subclasses

```
abstract class Shape {
    abstract double getArea();
    abstract double getPerimeter();

    void print() {
        System.out.println("Area " + getArea() +
                " and perimeter " + getPerimeter());
    }
}
```

```
jshell> new Rectangle(2.0, 3.0).print()
Area 6.0 and perimeter 10.0
```

- `Shape` implementation needs to be changed when:
  - shape properties are modified, e.g. **`double`** to Double
  - changes to where output is redirected, e.g. print to a file

Responsibilities of a Class

- Realize that now `Shape` has multiple *responsibilities*
- This violates the **Single Responsibility Principle**

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

- Responsibility is defined as the "reason to change"
- In our example,
  - let `Shape` be responsible for shape related properties and methods
  - `Shape` class returns a `String` representation instead
  - responsibility of output redirection given to another class

Responsibilities of a Class

```java
abstract class Shape {
    abstract double getArea();
    abstract double getPerimeter();

    String print() {
        return "Area " +  String.format("%.2f", getArea()) +
            " and perimeter " + String.format("%.2f", getPerimeter());
    }
}
```

```
jshell> new Rectangle(2.0, 3.0).print()
$.. ==> "Area 6.00 and perimeter 10.00"
```

- Consider a `scale` functionality to resize any concrete shape
  - Scaling a rectangle is different from scaling a circle
  - Scaling is not only relevant to `Shape`, but also to `3DShapes`

Inheriting from Multiple Parents?

- Define another abstract class `Scalable`

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

in C you can inherit from multiple parents

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

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

- Java prohibits multiple inheritance to avoid the creation of *weird* objects, e.g. `class Spork extends Spoon, Fork`
  - not desirable to inherit **properties** from different parents
  - but still appropriate to inherit functionality as specified by the **methods** from different parents

Defining an Interface as a Contract

- Even though a class can only inherit from one parent class, a class **can implement multiple interfaces**
- In our example, each shape
  - has associated properties and methods to support area and perimeter computations
  - can be scaled by a given factor and returned as a new shape
    - define a `Scalable` interface as a contract between the client and implementer

```
interface Scalable {
    Scalable scale(double factor);
}
```

Java Interface

- Just like abstract classes, interfaces cannot be instantiated
- Methods in interfaces are implicitly **public**
  - What is an appropriate return type and access modifier?

public

```
class Circle extends Shape implements Scalable {
    private final double radius;

    Circle(double radius) {
        this.radius = radius;
    }

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

    @Override
    double getPerimeter() {
        return 2 * Math.PI * radius;
    }

    @Override
    public Circle scale(double factor) {
        return new Circle(this.radius * factor);
    }
}
```

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.print())
Area 3.14 and perimeter 6.28
Area 6.00 and perimeter 10.00
```

- Can *extend* a new shape (say `Square`) without *modifying* the client's implementation — **Open-Closed Principle**

```
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.print())
Area 3.14 and perimeter 6.28
Area 6.00 and perimeter 10.00
Area 16.00 and perimeter 16.00
```

Polymorphism Revisited

```
jshell> Circle c = new Circle(1.0); Shape sh = c; Scalable sc = c
c ==> Circle@14acaea5
sh ==> Circle@14acaea5
sc ==> Circle@14acaea5

jshell> c.print()
$.. ==> "Area 3.14 and perimeter 6.28"

jshell> c.scale(2.0)
$.. ==> Circle@59494225

jshell> sh.print()
$.. ==> "Area 3.14 and perimeter 6.28"

jshell> sh.scale(2.0)
|  Error:
|  cannot find symbol
|    symbol:   method scale(double)
|  sh.scale(2.0)
|  ^------^
```

```
jshell> sc.scale(2.0)
$.. ==> Circle@5cb9f472

jshell> sc.print()
|  Error:
|  cannot find symbol
|    symbol:   method print()
|  sc.print()
|  ^------^
```

“Fat” Interface

- Why not combine scalability into Shape?

```
abstract class Shape {
    abstract double getArea();
    abstract double getPerimeter();

    abstract Shape scale(double factor);

    String print() {
        return "Area " +  String.format("%.2f", getArea()) +
            " and perimeter " + String.format("%.2f", getPerimeter());
    }
}
```

- **Interface Segregation Principle**

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

  – Classes should not implement methods that they can’t
  – Clients should not know of methods they don’t need

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

“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

SOLID Principles

- **S**ingle Responsiblity Principle*
- **O**pen-Closed Principle*
- **L**iskov Substitution Principle*
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle
  - *Program to an interface, not an implementation*

interface ensure that changes in implementor's classes don't sabotage codes in client's classes

> "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*

Preventing Inheritance and Overriding

- The **final** keyword can also be applied to methods or classes
  - Use the **final** keyword to explicit prevently inheritance

```
public final class Circle {
    :
}
```

final for class: no subclass
final for method: no override

  - To allow inheritance but prevent overriding

```
public class Circle {
    :
    @Override
    public final double getArea() {
        :
    }
    :
    @Override
    public final double getPerimeter() {
        :
    }
}
```

Lecture Summary

- Know how to define concrete/abstract classes or an interface
- Understand when to use inheritance or interfaces
- Understand how interfaces can also support polymorphism
- Demonstrate the application of SOLID principles in the design of object-oriented software, focusing on
  - **Single responsibility principle (SRP)**
  - **Open-closed principle (OCP)**
  - **Liskov substitution principle (LSP)**
- Appreciate “programming to an interface” that supports the maintainability, extensibility, and testing of the software