CS2030 Lecture 3

Substitutability in OO Design and Interfaces

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

Semester 1 2020 / 2021

Lecture Outline

- Liskov Substitution Principle
  - Program design
  - Return types of overriding methods
- Abstract class
- Interface
- Polymorphism revisited
- Guiding principles in OO design — SOLID

Liskov Substitution Principle (LSP)

- **Substitutability Principle** — if `S` is a subclass of `T`, then an object of type `T` can be replaced by that of type `S` *without changing the desirable property* of the program

  > "Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T."
  >
  > — Barbara Liskov

- As an example, if `FilledCircle` is a subclass of `Circle`, then everywhere we can expect areas and perimeters of circles to be computed, we can always replace a circle with a filled-circle
  - Example, using `getArea()` and `getPerimeter()`

LSP: OOP Program Design

```java
class Circle {
    Point centre;
    double radius;

    Circle(Point centre, double radius) {
        this.centre = centre;
        this.radius = radius;
    }

    @Override
    public String toString() {
        return "Circle centered at " + this.centre +
            " with radius " + this.radius;
    }
}

class UnitCircle extends Circle {
    UnitCircle(Point centre) {
        super(centre, 1.0);
    }
}
```

- Now consider including a `scaleBy(double factor)` functionality in the `Circle` class

LSP: OOP Program Design

- Defining a **`void`** method to "mutate" the object :

```java
class Circle {
    ...
    void scaleBy(double factor) {
        this.radius = this.radius * factor;
    }
}

jshell> UnitCircle uc = new UnitCircle(new Point(1, 1))
uc ==> Circle centered at (1.0, 1.0) with radius 1.0

jshell> uc.scaleBy(2.0)

jshell> uc
uc ==> Circle centered at (1.0, 1.0) with radius 2.0

jshell> uc instanceof UnitCircle // uc is still a UnitCircle?
$.. ==> true
```

- Breaks the integrity of the `UnitCircle` object
- Let's define an overriding method in the `UnitCircle` class

LSP: OOP Program Design

```java
class UnitCircle extends Circle {
    ...
    @Override
    void scaleBy(double factor) {
        // do nothing
    }
}
```

```
jshell> UnitCircle uc = new UnitCircle(new Point(1, 1))
uc ==> Circle centered at (1.0, 1.0) with radius 1.0

jshell> uc.scaleBy(2.0) // tries to scale UnitCircle by 2

jshell> uc
uc ==> Circle centered at (1.0, 1.0) with radius 1.0 // uc remains :)

jshell> Circle c = uc
c ==> Circle centered at (1.0, 1.0) with radius 1.0 // c refers to Circle

jshell> c.scaleBy(2.0) // scale c by 2

jshell> c
c ==> Circle centered at (1.0, 1.0) with radius 1.0 // c is not scaled?
```

- Breaks substitutability: `UnitCircle` cannot substitute `Circle`

LSP: OOP Program Design

- How about making `Circle` immutable? *Let's ponder...*

```
class Circle {
    ...
    Circle scaleBy(double factor) {
        return new Circle(this.centre, this.radius * 2);
    }
}

class UnitCircle extends Circle {
    ...
    @Override
    UnitCircle scaleBy(double factor) {
        return this;
    }
}

jshell> UnitCircle uc = new UnitCircle(new Point(1, 1))
uc ==> Circle centered at (1.0, 1.0) with radius 1.0

jshell> uc.scaleBy(2.0)
$.. ==> Circle centered at (1.0, 1.0) with radius 1.0 // Still valid?

jshell> Circle c = uc
c ==> Circle centered at (1.0, 1.0) with radius 1.0

jshell> c.scaleBy(2.0)
$.. ==> Circle centered at (1.0, 1.0) with radius 1.0 // Arghh!!
```

LSP: Overriding Method Return Types

- Notice that the return type of the `scaleBy` method in the superclass is `Circle`, while that of the subclass is `UnitCircle`
- In general, suppose `S` is a sub-class of `T`, i.e.

```
class S extends T {
    ...
}
```

  - if `T` has a method `foo` defined, what are the possible ways that a method `foo` defined in `S` overrides that of `T`?

- Consider how a client uses a variable of type `T`

```
T t = new T();
... = t.foo();
t = new S();
... = t.foo();
```

LSP: Overriding Method Return Types

```
jshell> class T {
   ...> Circle foo() { return new Circle(new Point(1, 1), 2); }
   ...> }
|  modified class T

jshell> class S extends T {
   ...> UnitCircle foo() { return new UnitCircle(new Point(0, 0)); }
   ...> }
|  modified class S

jshell> Circle c = new T().foo()
c ==> Circle centered at (1.0, 1.0) with radius 2.0

jshell> c = new S().foo()
c ==> Circle centered at (0.0, 0.0) with radius 1.0
```

- The compile-time type of `c` is `Circle`, but the runtime-type of `c` can be `Circle` or any of it's sub-class, e.g. `UnitCircle`

LSP: Overriding Method Return Types

- Return type of an overriding method cannot be more general than that of the overridden one

```
jshell> class S extends T {
   ...> Object foo() { return new UnitCircle(new Point(0, 0)); }
   ...> }
|  Error:
|  foo() in S cannot override foo() in T
|    return type java.lang.Object is not compatible with Circle
|  Object foo() { return new UnitCircle(new Point(0, 0)); }
|  ^-------------------------------------------------------^
```

- Allowing the above will make the following invalid:

```
c = new S().foo()
c.scaleBy(2.0) // what is scaleBy of an Object?
```

- How about accessibility modifiers of the methods?

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().getArea()
$.. ==> -1.0

jshell> new Shape().getPerimeter()
$.. ==> -1.0
```

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

```java
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

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

    @Override
    public String toString() {
        return "Area " + getArea() +
            " and perimeter " + getPerimeter();
    }
}
```

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

- An abstract class can contain both abstract methods as well as concrete methods

Inheriting from Multiple Parents?

- Define another abstract class `Scalable`

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

- 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?

```java
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)
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)
Area 3.14 and perimeter 6.28
Area 6.00 and perimeter 10.00
Area 16.00 and 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

“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*

> "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

```
final class Circle {
    ⋮
}
```

  - To allow inheritance but prevent overriding

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

Lecture Summary

- Appreciate Liskov Substitution Principle so as to avoid incorrect inheritance implementations
- Know when to define a concrete class, and when an abstract class is more appropriate
- Know how to define and implement an interface
- Understand when to use inheritance or interfaces
- Understand how inheritance and interfaces can also support polymorphism
- Demonstrate the application of SOLID principles in the design of object-oriented software, focusing on