CS2030 Lecture 4

Interface as the Abstraction Barrier

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

Semester 1 2019 / 2020

Lecture Outline

- Additional forms of inheritance
  - Abstract class
  - Interface
- Polymorphism with interfaces
- Inheritance versus Interface
  - Multiple interfaces
- OO Principles
  - Single Responsibility Principle
  - Open-Closed Principle
  - Interface Segregation Principle
  - Dependency Inversion Principle

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)
$4 ==> Area 3.14 and perimeter 6.28

jshell> new Rectangle(8.9, 1.2)
$5 ==> Area 10.68 and perimeter 20.20
```

- Some design considerations for the `Rectangle` class
  - A rectangle has a width and a height
  - Can obtain the area, as well as perimeter, from a rectangle
- Since `Rectangle` is a shape, and `Circle` is a shape, we can define `Shape` as the super-class of these two classes

"Inheriting" from `Shape`

- Some considerations:
  - `Circle` and `Rectangle` have different properties
  - Both `Circle` and `Rectangle` provide `getArea()` and `getPerimeter()` methods, although computed differently
  - Both `Circle` and `Rectangle` have a common `toString()` method that output the respective area and perimeter
- 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`

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

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

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

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

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

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

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

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

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

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

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

Design #2: **Shape** as an Abstract Class

- No longer useful to instantiate a Shape object

```
jshell> new Shape()
$6 ==> 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

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

    @Override
    public String toString() {
        return "Area " +  String.format("%.2f", getArea()) +
            " and perimeter " + String.format("%.2f", getPerimeter());
    }
}
jshell> new Shape()
|  Error:
|  Shape is abstract; cannot be instantiated
|  new Shape()
|  ^---------^
```

Design #2: **Shape** as an Abstract Class

- Now, we include two other functionalities to `Shape`
  - a `scale` functionality that resizes any concrete shape
  - a `print` functionality that allows any concrete shape to return a `String` representation (how about `toString`)?

```
public abstract class Shape {
    public abstract double getArea();
    public abstract double getPerimeter();
    public abstract Shape scale(double factor);
    public abstract String print();
}
```

- Define the two methods in `Circle` and `Rectangle`, e.g.

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

    @Override
    public String print() {
        return "Circle: area " + getArea() + "; perimeter " + getPerimeter();
    }
}
```

Single Responsibility Principle

- Realize that Shape is no longer just about an object having an area and perimeter
- Additional functionality to scale and print becomes part of Shape too
- 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"
  - Rather than letting Shape be responsible for shape properties, as well as scaling and printing, separate the two latter functionalities into individual classes or entities

Single Responsibility Principle

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

public abstract class Scalable {
    public abstract Object scale(double factor);
}

public abstract class Printable {
    public abstract String print();
}
```

- But a class can **only inherit from one parent class**
- The following is invalid

  ```
  public class Circle extends Shape, Scalable, Printable { ... }
  ```

- Java prohibits multiple inheritance to avoid the creation of *weird* objects, e.g. `class Spork extends Spoon, Fork`

Program Design with Contracts

- Even though a class can only inherit from one parent class, a class **can implement many interfaces**
- An alternative design for constructing shape objects (i.e. circles and rectangles) is to decide on what common **behaviour** each shape object should provide
- In our example, each shape
  - can return an area, as well as perimeter
  - can be scaled
  - can be printed
- The above defines `Shape`, `Scalable` and `Printable` "contracts" between the client and implementer
- In Java, the contract takes the form of an **interface**

Defining an Interface

- Define the `Shape`, `Scalable` and `Printable` interfaces

```
public interface Shape {
    public double getArea();
    public double getPerimeter();
}

public interface Scalable {
    public Object scale(double factor);
}

public interface Printable {
    public abstract String print();
}
```

- Just like abstract classes, interfaces cannot be instantiated
- Inheritance (via **`extends`**) depicts an **is–a** relationship
- On ther other hand, interfaces (via **`implements`**) depicts a
  - **is–a** relationship towards a non-concrete super-class
  - **can–do** relationship

Defining an Interface

- Redefine `Circle` to implement multiple interfaces

```
public class Circle implements Shape, Scalable, Printable {
    private final double radius;

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

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

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

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

    @Override
    public String print() {
        return "Circle: area " + getArea() + "; perimeter " + getPerimeter();
    }
}
```

Polymorphic **Shape** Objects

- Interfaces support polymorphism as well

```
jshell> Shape[] shapes = {new Circle(1.0), new Rectangle(2.0, 3.0)}
jshell> for (Shape s : shapes) ((Scalable)s).scale(2.0)
jshell> for (Shape s : shapes) System.out.println(((Printable)s).print())
Circle: area 3.141592653589793; perimeter 6.283185307179586
Rectangle: area 6.0; perimeter 10.0
```

- Cast the object to respective interfaces to `scale` and `print`
- 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)}
jshell> for (Shape s : shapes) ((Scalable)s).scale(2.0)
jshell> for (Shape s : shapes) System.out.println(((Printable)s).print())
Circle: area 3.141592653589793; perimeter 6.283185307179586
Rectangle: area 6.0; perimeter 10.0
Square: area 16.0; perimeter 16.0
```

Interface Segregation Principle (ISP)

"no client should be forced to depend on methods it does not use."

— *Uncle Bob*

- Keep interfaces minimal; avoid "fat" interfaces
  - Classes should not implement methods that they can't
  - Clients should not know of methods they don't need
- Split interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them

ie when importing methods, only take what is necessary rather than the whole module

Interface Segregation Principle (ISP)

- The many faces of you...

```
public interface MummysBoy {
    public String hi();
}

public class Me implements MummysBoy, Romantic, PartyAnimal {
    @Override
    public String hi() {
        return "Hi mum... :)";
    }

    @Override
    public String hey() {
        return "Hey babe... :x";
    }

    @Override
    public String supp() {
        return "Supp dudes... :P";
    }
}
```

- `Mummy` only wants to know the "filial" side of her son
- `Mummy` references `Me` via the interface `MummysBoy`

Interface Segregation Principle (ISP)

```
jshell> Me me = new Me()
me ==> Me@14acaea5

jshell> me.hi()
$6 ==> "Hi mum... :)"

jshell> me.hey()
$7 ==> "Hey babe... :x"

jshell> me.supp()
$8 ==> "Supp dudes... :P"
```

Notice that Me@14acaea5 is the same object as above

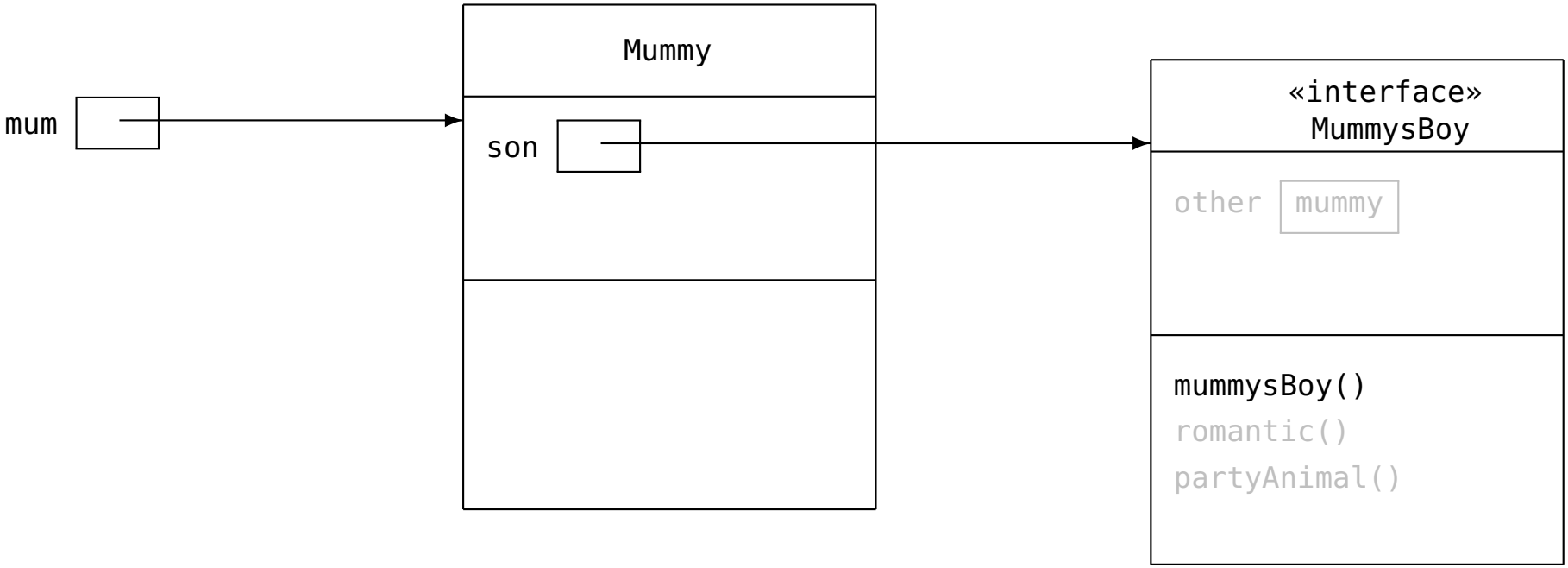
```
jshell> MummysBoy ahBoy = me
ahBoy ==> Me@14acaea5

jshell> ahBoy.hi()
$10 ==> "Hi mum... :)"
```

```
jshell> ahBoy.hey()
|  Error:
|  cannot find symbol
|    symbol:   method hey()
|  ahBoy.hey()
|  ^-------^
jshell> ahBoy.supp()
|  Error:
|  cannot find symbol
|    symbol:   method supp()
|  ahBoy.supp()
|  ^--------^
```

Interface Segregation Principle (ISP)

- `Mummy` has restricted knowledge of (restricted access to) certain methods of `Me`



Interface as an Abstraction

"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."

Low level, the abstract details that define the full boundaries of a program
Interface -> Contract between the low level and high level
Client level details -> the implementation of the details according to the contract

— *Uncle Bob*

- **Dependency Inversion Principle**
  - Abstractions: interface
  - Details: concrete implementations
- *Program to an interface, not an implementation*

Programming to an Interface

- Difference between concrete, abstract classes and interface:
  - A **concrete class** is the actual implementation
  - An **interface** is a contract that specifies the abstraction that its implementer should implement
  - An **abstract class** is a trade off between the two, i.e. when we need to have the partial implementation of the contract
    - Typically used as a base class
- A client should always use the interfaces from an application rather than the concrete implementations
- The benefits of this approach are **maintainability**, **extensibility** and **testablility**

Stubbing

This is to be able to implement your Circle class and allow for unit testing even without the actual implementation of the Point class by only implementing the few methods and attributes of the actual point class relevent to your class

- Using Lab #1 as an example, suppose you are implementing the `Circle` class, while your friend is implementing `Point`
- How to commence testing `createCircle` (and possibly higher level methods) without waiting for `Point` to complete?

```
static Circle createCircle(Point p, Point q, double radius) {
    double distPQ = p.distanceTo(q);
    if (distPQ < 2 * radius + 1E-15 && distPQ > 0) {
        Point c = p.midPoint(q);
        double cp = Math.sqrt(Math.pow(radius, 2) - Math.pow(p.distanceTo(c), 2)
        double theta = p.angleTo(q);
        return Circle.getCircle(c.moveTo(theta + Math.PI / 2, cp), radius);
    } else {
        return null;
    }
}
```

- Let's restrict the test using two points that are 2 units apart, i.e. the midpoint is exactly the centre of the unit circle
- *Dependency injection* via inheritance: `PointStub` is-a `Point`

Stubbing

- Need only minimally functional `midPoint` and `moveTo`

```
class PointStub extends Point {
    private final double x;
    private final double y;

    public PointStub(double x, double y) {
        super(x, y);
        this.x = x;
        this.y = y;
    }

    @Override
    public Point midPoint(Point q) {
        PointStub qs = (PointStub)q;
        return new PointStub((this.x + qs.x) / 2, (this.y + qs.y) / 2);
    }

    @Override
    public PointStub moveTo(double theta, double d) {
        return new PointStub(this.x, this.y);
    }
}
```

PointStub will just remain at the same point for a unit test with the 2 points in createCircle at a diameter line-> the point of this is to show that I can create a circle at the given point, such that I can focus on further higher level implementations based on my Circle

Stubbing

- Notice that the behaviour of the correct implementation of `Point` and it's stubbed version `PointStub` is exactly the same

```
jshell> Main.createCircle(new Point(1, 0), new Point(-1, 0), 1)
$6 ==> circle of radius 1.0 centered at point (0.000, 0.000)

jshell> Main.createCircle(new Point(0.5, 0), new Point(-1.5, 0), 1)
$7 ==> circle of radius 1.0 centered at point (-0.500, 0.000)

jshell> Main.createCircle(new PointStub(1, 0), new PointStub(-1, 0), 1)
$8 ==> circle of radius 1.0 centered at point (0.000, 0.000)

jshell> Main.createCircle(new PointStub(0.5, 0), new PointStub(-1.5, 0), 1)
$9 ==> circle of radius 1.0 centered at point (-0.500, 0.000)
```

- More importantly, there is no need to modify the `createCircle` method to handle `PointStub` objects

Preventing Inheritance and Overriding

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

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

  - We can also 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
  - Single responsibility principle (SRP)
  - Open-closed principle (OCP)
  - Liskov substitution principle (LSP)
  - Interface Segregation Principle (ISP)
  - Dependency Inversion Principle (DIP)
- Appreciate "programming to an interface" that supports the maintainability, extensibility, and testing of the software