CS2030 Programming Methodology II

Lecture III

Shengdong Zhao
Spring 2019

Acknowledge: slides are adapted from Henry Chia

Lecture Outline

- **Abstract class**
- **Interface**
  - Supporting polymorphism
- **Inheritance versus Interface**
  - Multiple interfaces
- **Object equality**
- **Access modifiers**
- **Packaging**
- **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> Circle c = new Circle(1.0)
c ==> Area 3.14 and perimeter 6.28

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

- How should we design the `Rectangle` class?
  - A rectangle has a width and a height
  - We can get 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

- Redefine the `Circle` class so that it now `extends` from Shape

```
public class Circle extends Shape {
        private double radius;
                public Circle(double radius) {
                        this.radius = radius;
        }
        public double getArea() {
                return Math.PI * radius * radius;
        }
        public double getPerimeter() {
                return 2 * Math.PI * radius;
        }
        public String toString() {
                return "Area " + getArea() +
                        " and perimeter " + getPerimeter();
        }
}
```

- So what's the definition of the Shape class?

Design #1: Shape as a Concrete Class

- Shape as an empty class?

```
public class Shape { }
```

- But how to ensure that `Circle` and `Rectangle` must have `getArea` and `getPerimeter` methods?
- Shape with dummy `getArea` and `getPerimeter` methods

```
public class Shape {
        public double getArea() { return 0; }
        public double getPerimeter() { return 0; }
        @Override
        public String toString() {
        return "Area " + getArea() +
                " and perimeter " + getPerimeter();
        }
}
```

Abstract Classes

- Whether `Shape` is an empty class, or contains dummy methods, it is no longer useful as a `concrete` class
- Redefine `Shape` as an `abstract` class with abstract methods; these methods are to be implemented in the child classes

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

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

Shapes, Circles and Rectangles Revisited

- 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
  - can return an perimeter
  - can return a string representation for output purposes
- The above defines a Shape "contract" between what the user expects of the implementer of a shape object
- In Java, the contract takes the form of an **interface**

Defining an Interface

- The Shape interface is defined as

```
public interface Shape {
        static final double PI = 22.0 / 7;

        public double getArea();

        public double getPerimeter();

        @Override
        public String toString();
}
```

- Note that an interface does not allow for instance properties; apart from constant declarations (e.g PI defined above)
- Just like an abstract class, interfaces cannot be instantiated

Interfaces support Polymorphism

- We have seen that `Circle` (Rectangle) inherits from Shape
  - We say that `Circle` (Rectangle) is a Shape
- Here, we see that `Circle` (Rectangle) implements Shape
  - We say that `Circle` (Rectangle) has the capabilities of Shape
- Hence, inheritance (via extends) depicts an **is–a** relationship

Interfaces support Polymorphism

- On the other hand, interfaces (via implements) depicts a
  - **can–do** relationship
  - **is–a** relationship towards a non-concrete super-class
- Both inheritance and interface allows a `Circle` (Rectangle) to take on the form of a Shape — polymorphism

Re-defining the Circle Class

- Circle class now implements the Shape interface

```
public class Circle implements Shape {
        private double radius;
        public Circle(double radius) {
                this.radius = radius;
        }
        @Override
        public double getArea() {
                return Shape.PI * radius * radius;
        }
        @Override
        public double getPerimeter() {
                return Shape.PI * radius;
        }
        @Override
        public String toString() {
                return "Circle with area " + getArea() +
                        " and perimeter " + getPerimeter();
        }
}
```

Re-defining the Rectangle Class

- Rectangle class also implements the Shape interface

```
public class Rectangle implements Shape {
        private double width;
        private 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);
        }
        @Override
        public String toString() {
                return "Rectangle with area " + getArea() +
                        " and perimeter " + getPerimeter();
        }
}
```

Polymorphic Shape Objects

- Notice how polymorphism, as well as dynamic (or late) binding, takes effect in the same way as inheritance

```
class Main {
	public static void main(String[] args) {
		Shape[] shapes = {new Circle(1.0),
			new Rectangle(8.9, 1.2)};
		for (Shape shape : shapes) {
			System.out.println(shape);
		}
	}
}
```

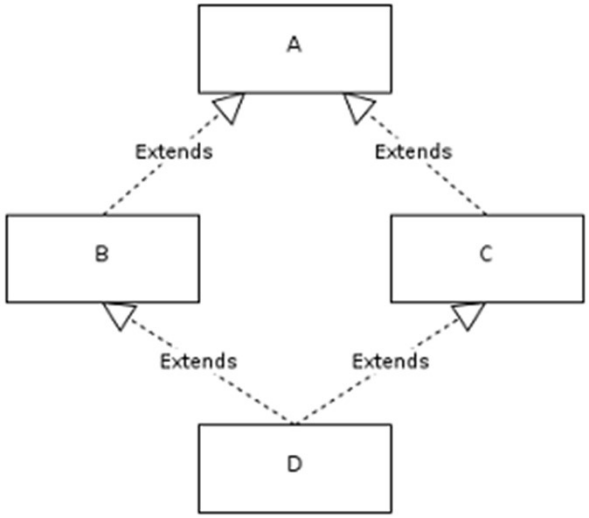
- Running the program gives the following output

  `Circle` with area 3.0 and perimeter 3.0

  `Rectangle` with area 10.68 and perimeter 20.2

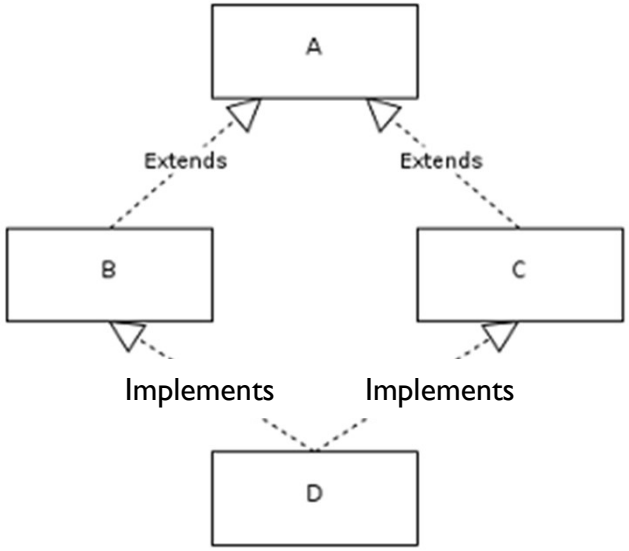
Inheritance vs Interface

- While both inheritance and interfaces supports polymorphism, there is one important difference between them
- A class can only inherit from one parent class, but a class can implement many interfaces
  - Java prohibits multiple inheritance to avoid common-sense ambiguities, e.g. *spork is both a spoon and a fork*



The Diamond Problem: We have two classes B and C inheriting from A. Assume that B and C are overriding an inherited method and they provide their own implementation. Now D inherits from both B and C doing multiple inheritance. D should inherit that overridden method, which overridden method will be used? Will it be from B or C? Here we have an ambiguity.

Why it is ok to implement multiple Interfaces?



The Diamond Problem is not a problem for Interfaces. We have two interfaces B and C inheriting from interface A. Assume that B and C are overriding an inherited method x(). Now D is concrete class implements both B and C. D needs to implement x(). Since neither x() has any details from interface B or C, D's implementation of method x() works for both interface B or C, so there is no ambiguity on which implementation to use when method x() is called. It will always be D's implementation.

Implementing Multiples Interfaces

- Let's suppose `Circle (Rectangle)` has a scalable capability, i.e. it should implement the Scalable interface in addition to the Shape interface

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

- `Circle` class implements both Shape and `Scalable` interfaces

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

        public Circle(double radius) {
                this.radius = radius;
        }
        @Override
        public double getArea() {
                return Math.PI * radius * radius;
        }
        @Override
        public double getPerimeter() {
                return Math.PI * radius;
        }
        @Override
        public String toString() {
                return "Circle with area " + getArea() +
                        " and perimeter " + getPerimeter();
        }
        @Override
        public void scale(double factor) {
                this.radius *= factor;
        }
}
```

Scaling and Printing Circles and Rectangles

- To scale each shape, and then ouput

```
class Main {
        public static void main(String[] args) {
                Shape[] shapes = {
                        new Circle(1.0),
                        new Rectangle(8.9, 1.2)};
                for (Shape shape : shapes) {
                        ((Scalable) shape).scale(2.0);
                        System.out.println(shape);
                }
        }
}
```

- ((Scalable) shape).scale(2.0) casts the object Shape object to a Scalable in order to invoke the scale method
  - The Shape object does not know that Circle (or Rectangle) implements other interfaces apart from its own

Access Modifiers

- In the discussion of an abstraction barrier, we have seen the use of the **public**, **private** and **protected** modifiers
- Other than these three, there is a default modifier
- Java adopts an additional **package** abstraction mechanism that allows the grouping of relevant classes/interfaces together under a *namespace*, just like java.lang

Access Modifiers

- In particular, a **protected** field can be accessed by other classes within the same package
- The access level (most restrictive first) is given as follows:
  - private (visible to the class only)
  - default (visible to the package)
  - protected (visible to the package and all sub classes)
  - public (visible to the world)

non static protected method can only be accessed by non static method

Access Modifiers

Access Modifiers ->	private	Default/no-access	protected	public
Inside class	Y	Y	Y	Y
Same Package Class	N	Y	Y	Y
Same Package Sub-Class	N	Y	Y	Y
Other Package Class	N	N	N	Y
Other Package Sub-Class	N	N	Y	Y

Creating Packages

- Let's use an example where we desire to have Shape, Scalable, Circle and Rectangle classes/interfaces reside in the cs2030.shapes package
- Include the following line at the top of the java files **package** cs2030.shapes;
- Compile the four Java files using
  `javac -d . *.java`
- This will create the cs2030/shapes directory with the associated class files stored within

Creating Packages

- The client, say Main.java, now requires the files in the cs2030.shapes package to be imported

```
import cs2030.shapes.Shape;
import cs2030.shapes.Scalable;
import cs2030.shapes.Circle;
import cs2030.shapes.Rectangle;

class Main {
        public static void main(String[] args) {
                Shape[] shapes = {new Circle(1.0),
                        new Rectangle(8.9, 1.2)};
                for (Shape shape : shapes) {
                        ((Scalable) shape).scale(2.0);
                        System.out.println(shape);
                }
        }
}
```

Preventing Inheritance and Overriding

- We have seen how the **final** keyword can be used to create final variables, or variables containing values that cannot be changed; in other words, constants
- The **final** keyword can also be applied to methods or classes
- Sometimes we need to prevent a class from being inherited
  - As an example, java.lang.Math and java.lang.String classes cannot be inherited from **(Why?)**
  - We can use the **final** keyword to explicit prevently inheritance

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

String important, lots of java mechanism rely on it, hackers can use to create security holes if override

Preventing Inheritance and Overriding

- We can also allow inheritance but prevent overriding

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

Lecture Summary

- Know when to define a concrete class, and when an abstract class is more appropriate
- Know how to define and implement an interface
- Understand how interfaces can support polymorphism
- Understand when to use inheritance and when to use interfaces
- Understand the restriction levels of different access modifiers
- Appreciate how packages can be created to realize another abstraction level
- Know how to prevent inheritance and overriding when necessary