CS2030 Lecture 3

Substitutability in Object-Oriented Design

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

Semester 2 2019 / 2020

Lecture Outline

- OO Principles
  - Abstraction
  - Encapsulation
  - **Inheritance**
    - Abstraction principle
    - Super–sub (Parent–child) classes
  - **Polymorphism**
    - Dynamic vs Static binding
    - Compile-time vs run-time type
- Method overriding vs method overloading
- Liskov Substitution Principle

Designing a Filled Circle

- Given below is a simplified `Circle` class having one `radius` property and methods `getArea()` and `getPerimeter()`

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

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

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

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

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

Designing a Filled Circle

- Creating a `FilledCircle` object to be filled with a color using `java.awt.Color`

```
jshell> /open Circle.java

jshell> /open FilledCircle.java

jshell> new Circle(1.0)
$4 ==> circle: area 3.14, perimeter 6.28

jshell> new FilledCircle(1.0, Color.BLUE)
$5 ==> circle: area 3.14, perimeter 6.28, java.awt.Color[r=0,g=0,b=255]
```

- What are the different ways in which `FilledCircle` class can be defined?

Design #1: As a Stand-alone Class

```java
import java.awt.Color;

class FilledCircle {
    private final double radius;
    private final Color color;

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

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

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

    Color getColor() {
        return color;
    }

    @Override
    public String toString() {
        return "circle: area " + String.format("%.2f", getArea()) +
            ", perimeter " + String.format("%.2f", getPerimeter()) +
            ", " + getColor();
    }
}
```

Abstraction Principle

Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts

— *Benjamin C. Pierce*

Design #2: Using Composition

- **has-a** relationship: `FilledCircle` *has a* `Circle`

```
class FilledCircle {
    private final Circle circle;
    private final Color color;

    FilledCircle(double radius, Color color) {
        circle = new Circle(radius);
        this.color = color;
    }

    double getArea() {
        return circle.getArea();
    }

    double getPerimeter() {
        return circle.getPerimeter();
    }

    Color getColor() {
        return color;
    }

    @Override
    public String toString() {
        return "circle: area " + String.format("%.2f", getArea()) +
            ", perimeter " + String.format("%.2f", getPerimeter()) +
            ", " + getColor();
    }
}
```

Design #3: Using Inheritance

- **is-a** relationship: `FilledCircle` is a `Circle`

```
class FilledCircle extends Circle {
    private final Color color;

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

    Color getColor() {
        return color;
    }

    @Override
    public String toString() {
        return super.toString() + ", " + getColor();
    }
}
```

- Parent/Super class: `Circle`; child/sub class: `FilledCircle`

Inheritance

- `FilledCircle` invokes the parent class `Circle`'s constructor using `super(radius)` within it's own constructor
- The `radius` variable in `Circle` can also be made accessible to the child class by changing the access modifier

```
class Circle {
    protected final double radius;
    ...
```

- The `super` keyword is used for the following purposes:
  - `super(..)` to access the parent's constructor
  - `super.radius` or `super.toString()` can be used to make reference to the parent's properties or methods; especially useful when there is a conflicting property of the same name in the child class

Polymorphism

- Other than as an “aggregator” of common code fragments in similar classes, inheritance is used to support **polymorphism**
- Polymorphism means “many forms”

```
jshell> Circle c = new Circle(1.0)
c ==> circle: area 3.14, perimeter 6.28

jshell> c = new FilledCircle(1.0, Color.BLUE)
c ==> circle: area 3.14, perimeter 6.28, java.awt.Color[r=0,g=0,b=255]

jshell> FilledCircle fc = new FilledCircle(1.0, Color.BLUE)
fc ==> circle: area 3.14, perimeter 6.28, java.awt.Color[r=0,g=0,b=255]

jshell> fc = new Circle(1.0)
|  Error:
|  incompatible types: Circle cannot be converted to FilledCircle
|  fc = new Circle(1.0)
|       ^-------------^
```

Static binding

- Consider an array `Circle[] circles`

```
jshell> Circle[] circles = {new Circle(1), new FilledCircle(1, Color.BLUE)}
```

- How do we output the objects one at a time?
- Using static (early) binding
  - check the types (more specific first):

```
for (Circle circle : circles) {
    if (circle instanceof FilledCircle) {
        System.out.println((FilledCircle) circle);
    } else if (circle instanceof Circle) {
        System.out.println((Circle) circle);
    }
}
```

- Static binding occurs during compile time, i.e. decide which specific method to call during program compilation

Dynamic binding

- Contrast static binding with dynamic (or late) binding

```
for (Circle circle : circles) {
    System.out.println(circle);
}
```

- Notice that the exact type of circle, and the exact `toString` method to invoke, is not known until runtime
- Polymorphism and dynamic binding leads to extensible implementations
  - Simply add a new sub-class of circle that extends the `Circle` class and overriding the appropriate methods
  - Does not require the client code (above) to be modified

Compile-Time vs Run-Time Type

- Consider the following statement:

  ```
  Circle circle = new FilledCircle(1.0, Color,BLUE);
  ```

- `circle` has a compile-time type of `Circle`
  - the type in which the variable is declared
  - restricts the methods it can call during compilation, e.g. `circle.getArea()`, but not `circle.getColor()`
- `circle` has a run-time type of `FilledCircle`
  - the type of the object that the variable is pointing to
  - determines the actual method called, e.g. `toString()` in `FilledCircle`, rather than `Circle`
- Clearly, a variable's compile-type is fixed at compile time, while its run-time type may vary as the program runs

Liskov Substitution Principle (LSP)

- Introduced by Barbara Liskov

  > "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."

- This **substitutability** principle means that 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
- 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 and Type/Sub-type Consistency

- Suppose `class B extends A`, and `A` has a method `foo`
- What are the possible ways that a method `foo` defined in `B` overrides that of `A`?
  - Consider how can clients use a variable of type `A`

    ```
    A a = new A();
    a.foo();
    a = new B();
    b.foo();
    ```

- Return type cannot be more general than that of the overridden method
- How about accessibility modifiers of the methods?
- Parameter type cannot be more specific than the overridden method, *but need to also consider method overloading...*

Access Modifiers

- We have seen **public**, **private** and **protected** modifiers
- There is also 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`
- 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)

Method Overloading

- Methods of the same name can co-exist if the *signatures (number, order, and type of arguments)* are different
- Method overloading is very common among constructors

```
Circle() {
    this.radius = 1.0;
}

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

- **Static binding** occurs during method overloading
  - method to be called is determined during compile time

```
class A {
    Number foo(Number x) { ... }
    Number foo(String x) { ...}
}
```

```
A a = new A();
a.foo(123)
a.foo("123")
```

Overriding or Overloading?

- We have considered defining **`equals`** as an overriding method

```java
@Override
public boolean equals(Object obj) {
    return this == obj ||
        (obj instanceof Circle &&  this.radius == ((Circle) obj).radius);
}
```

- Can we define as an overloaded method instead?

```java
public boolean equals(Circle c) {
    return this.radius == c.radius;
}
```

- Using an overloaded method, would it be possible for a client to invoke the **`equals`** method of the superclass **`Object`**?
- With an overriding **`equals`** method, is it possible for a client to invoke the overridden one?
  - *Ponder... can an overridden method ever be invoked?*

Effective use of LSP in OOP Design

- Consider a 20% salary upgrade for the salary bracket $(0, 1000)$

```
class Salary {
    protected final double amount;

    protected Salary(double amount) {
        this.amount = amount;
    }

    Salary upgrade() {
        assert this.amount > 0 && this.amount < 1000;
        return new Salary(this.amount * 1.2);
    }

    @Override
    public String toString() {
        return "Salary: $" + this.amount;
    }
}
```

- Consider a client's `upgradeSalary` method as follows:

```
static Salary upgradeSalary(Salary salary) {
    salary = salary.upgrade();
    return salary;
}
```

Liskov Substitution Principle (LSP)

- Which `SalaryTest` classes is/are substitutable for `Salary`?

```java
public static void main(String[] args) {
    Salary s = new SalaryTest(Double.valueOf(args[0]));
    System.out.println(upgradeSalary(s));
}
```

```java
class SalaryTest extends Salary {
    ...
    Salary upgrade() {
        assert amount > 0 && amount < 100;
        return new SalaryTest(amount * 1.2);
    }
    ...
}
```

```java
class SalaryTest extends Salary {
    ...
    Salary upgrade() {
        assert amount > 0 && amount < 10000;
        return new SalaryTest(amount * 1.2);
    }
    ...
}
```

Inheritance Misuse

- Keeping in mind the *substitutability* principle can help us avoid incorrect usage of inheritance
- The following is incorrectly designed, although looks functional

```java
class FilledCircle {
    private final double radius;
    private final Color color;

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

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

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

    Color getColor() {
        return this.color;
    }

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

```java
class Circle extends FilledCircle {

    Circle(double radius) {
        super(radius, null);
    }

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

Inheritance Misuse

```
jshell> FilledCircle[] fcs = {new FilledCircle(1.0, Color.BLUE), new Circle(2.0)}
fcs ==> FilledCircle[2] { 3.141592653589793 6.28318530717 ... 59172 12.56637061435917
```

```
jshell> fcs[0].getArea()
$5 ==> 3.141592653589793

jshell> fcs[1].getArea()
$6 ==> 12.566370614359172
```

- However, when testing the property of color, substitutability implies that `FilledCircle` can be replaced by `Circle`

```
jshell> fcs[0].getColor()
$7 ==> java.awt.Color[r=0,g=0,b=255]

jshell> fcs[1].getColor()
$8 ==> null
```

Inheritance Misuse

- Do not confuse a **has-a** relationship with **is-a**

```
class Point {
    protected double x;
    protected double y;

    Point(double x, double y) {
        this.x = x; this.y = y;
    }

    @Override
    public String toString() {
        return "(" + this.x + ", " + this.y + ")";
    }
}

class Circle extends Point {
    private double radius;

    Circle(Point point, double radius) {
        super(point.x, point.y);
        this.radius = radius;
    }

    @Override
    public String toString() {
        return "circle: radius " + radius + " centered at " + super.toString();
    }
}
```

Lecture Summary

- □ Understand the object-oriented principles of abstraction, encapsulation, inheritance and polymorphism
- □ Know the difference between static (early) and dynamic (late) binding, and understand their use in relation to compile-time type and run-time type
- □ Differentiate between method overloading and method overriding, and circumstances in which they are used
- □ Distinguish between an is-a relationship and a has-a relationship, and choose the appropriate one during object-oriented design
- □ Appreciate Liskov Substitution Principle so as to avoid incorrect inheritance implementations