CS2030 Lecture 3

Substitutability Principle in Object-Oriented Design

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

Semester 1 2019 / 2020

Lecture Outline

- OO Principles
  - Abstraction
  - Encapsulation
  - **Inheritance**
    - Super–sub (Parent–child) classes
  - **Polymorphism**
    - Dynamic vs Static binding
- Method overloading
- Mental-modeling objects with inheritance
- Liskov Substitution Principle

Overriding Revisited: `equals` Method

- Other than the `toString` method, another commonly overridden method is the `equals` method
- Within the `Object` class, the `equals` method compares if two object references refer to the same object

```
jshell> new Point(0, 0) == new Point(0, 0)
$2 ==> false

jshell> new Point(0, 0).equals(new Point(0, 0))
$3 ==> false

jshell> new Point(0, 0).toString() == new Point(0, 0).toString()
$4 ==> false

jshell> new Point(0, 0).toString().equals(new Point(0, 0).toString())
$5 ==> true
```

- If points with the same coordinate values are equal, we need to override the `equals` method inherited from `Object`

Overriding `Object`'s `equals` Method

- A naïve way of overriding
  the `equals` method is to define the method in the following way:

```
public boolean equals(Object obj) {
    Point p = (Point) obj;
    return Math.abs(this.x - p.x) < 1E-15 &&
        Math.abs(this.y - p.y) < 1E-15;
}
```

```
jshell> new Point(0,0).equals(new Point(0,0))
$2 ==> true
```

- Since the `equals` method takes in a parameter of `Object`
  - need to **type-cast** `obj` from `Object` type to `Point` type before accessing the radius in order to check for equality
- But what if the an object of different type is compared?
  - A ClassCastException is thrown

Overriding `Object`'s `equals` Method

- With a good sense of type awareness, the correct way to override the `equals` method is

```
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    } else if (obj instanceof Point) {
        Point p = (Point) obj;
        return Math.abs(this.x - p.x) < 1E-15 &&
            Math.abs(this.y - p.y) < 1E-15;
    } else {
        return false;
    }
```

- In essence,
  - first check if it's the same object
  - then check if it's the same type
  - then check the associated equality property

Constructing Tests with `equals`

- Suppose we would like to test the `midPoint` method

```
public Point midPoint(Point otherPoint) {
    return new Point((this.x + otherPoint.x)/2,
            (this.y + otherPoint.y)/2);
}
```

- Rather than "test" the actual output of the returned `Point` object via the `toString` method

```
jshell> new Point(0, 0).midPoint(new Point(1, 1))
$2 ==> point (0.5, 0.5)
```

- The proper way is to test the equality between the actual `Point` object that is returned with the expected one

```
jshell> new Point(0, 0).midPoint(new Point(1, 1)).equals(
   ...> new Point(0.5, 0.5))
$3 ==> true
```

Designing a Filled Circle

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

```
public class Circle {
    private final 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;
    }

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

Designing a Filled Circle

- Let's create a `FilledCircle` object to be filled with a color that uses the `java.awt.Color` class provided by Java

```
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

```
import java.awt.Color;

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

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

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

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

    public 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`

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

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

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

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

    public 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`

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

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

    public Color getColor() {
        return color;
    }

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

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

Inheritance

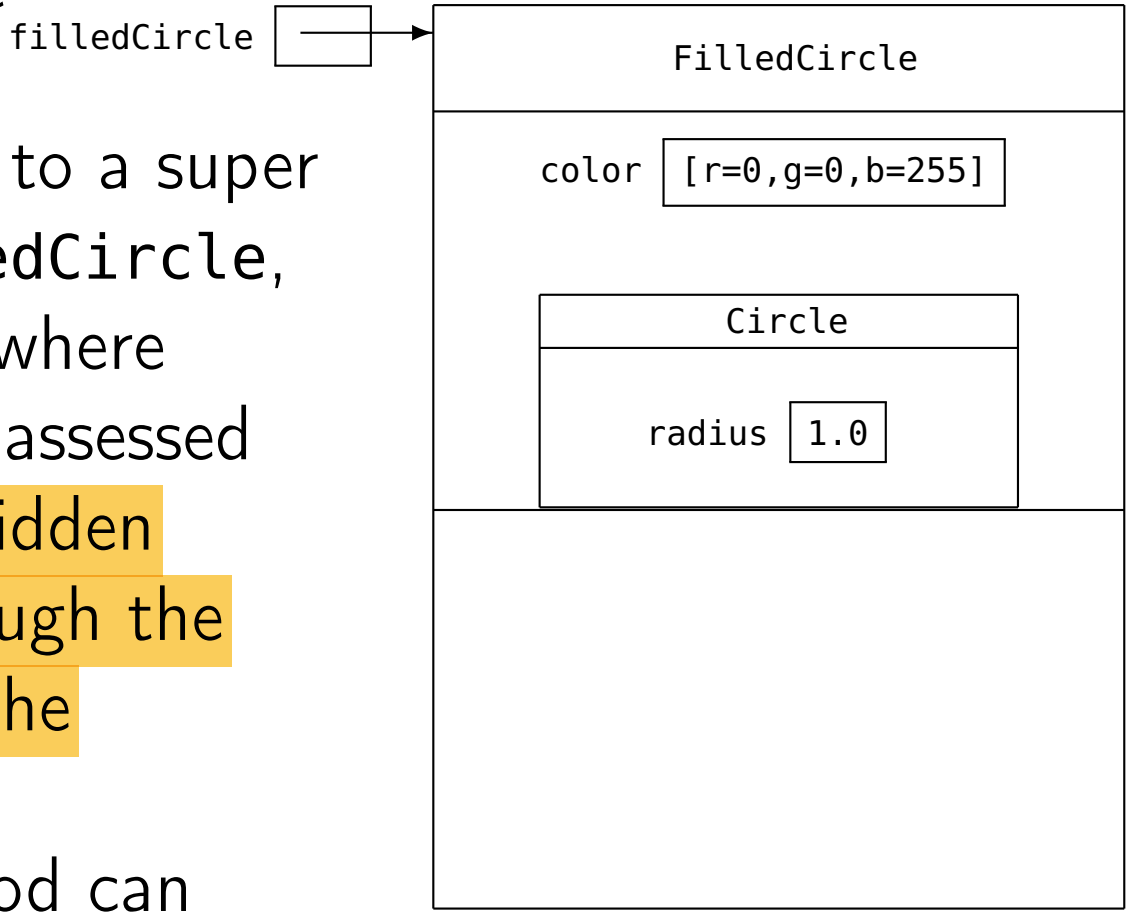
- `FilledCircle` invokes the parent class `Circle`'s constructor using `super(radius)` within it's own constructor
- The `radius` variable in `Circle` can also be made accessible to the child class by changing the access modifier

```
public 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

Modeling Inheritance

- Notice how the child object "wraps-around" the parent
- Type-casting a child object to a super class, e.g. `(Circle) filledCircle`, refers to the parent object where attributes/methods can be assessed
- The only exception is overridden methods; calling them through the parent or child will invoke the overriding methods
- An overridden parent method can only be called within the child class via **super**



Polymorphism

- How is inheritance useful?
- 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

- Given an array `Circle[] circles` comprising both `Circle` and `FilledCircle` objects, e.g

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

  output these objects one at a time
- In static (or early) binding, we can do something like this:

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

- Static binding occurs during compile time, i.e. all information needed to call a specific method can be known at compile time

Method Overloading

- Static binding also occurs during **method overloading**
- Method overloading commonly occurs in constructors

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

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

- The method to be called is determined during compile time

```
jshell> new Circle(2.0)
$4 ==> circle: area 12.57, perimeter 12.57

jshell> new Circle()
$5 ==> circle: area 3.14, perimeter 6.28
```

meaning I can construct from diff datatypes or order
Although mixing the order would just mess up your mental model

- Methods of the same name can co-exist if the *signatures (number, order, and type of arguments)* are different

Dynamic binding

- Contrast static binding with dynamic (or late) binding

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

- The above will give the same output as in the previous case
- Notice that the exact type of circle, and the exact `toString` method to be overridden, is not known until runtime
- Polymorphism with dynamic binding leads to more easily extensible implementations (*open-closed principle*)
  - 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

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."

  IE where a object x can be used, if y is a subclass of x, it can be used in any method of x for the same property

- 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 circles to be used, we can replace a circle with a filled-circle
  - Example, using `getArea()` and `getPerimeter()`

Liskov Substitution Principle (LSP)

- Given the following `Account` class

```
public class Account {
    protected final int balance;

    public Account(int balance) {
        this.balance = balance;
    }

    public boolean checkBalance() {
        return this.balance > 0 && this.balance < 100;
    }
}
```

Account A cannot as it has a max of only 10!

- Which one of the following accounts is **not** substitutable?

```
public class AccountA extends Account {
    public AccountA(int balance) {
        super(balance);
    }

    public boolean checkBalance() {
        return this.balance > 0 &&
            this.balance < 10;
    }
}
```

```
public class AccountB extends Account {
    public AccountB(int balance) {
        super(balance);
    }

    public boolean checkBalance() {
        return this.balance > 0 &&
            this.balance < 1000;
    }
}
```

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

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

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

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

    public Color getColor() {
        return this.color;
    }

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

```java
public class Circle extends FilledCircle {

    public 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 12.566370614359
```

```
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**

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

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

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


public class Circle extends Point {
    private double radius;

    public 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 OO principles of abstraction, encapsulation, inheritance and polymorphism
- Know the difference between static (early) and dynamic (late) binding
- Differentiate between method overloading and method overriding
- Distinguish between an is-a relationship and a has-a relationship and apply the appropriate design
- Extend the mental model of program execution for an object to include inheritance
- Appreciate the Liskov Substitution Principle so as to avoid incorrect inheritance implementations