CS2030 Lecture 2

Advanced Object-Oriented Programming Concepts — Inheritance and Polymorphism

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

Semester 1 2020 / 2021

Lecture Outline

- Recap
  - OOP Principles 1 & 2: Abstraction and Encapsulation
- OOP principle 3: Inheritance
  - Super–sub (Parent–child) classes
  - is-a relationship
- OOP principle 4: Polymorphism
  - Overriding methods
- Abstraction principle
- Compile-time vs run-time type
- Dynamic vs Static binding
  - Method overriding vs method overloading

`toString` Method

- Thus far, creating an object using JShell results in the address of the object being displayed

```
jshell> new Point(1.0, 2.0)
$.. ==> Point@5c3bd550
```

- Make it more meaningful by defining a `toString` method with the following method header:

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

jshell> new Point(1.0, 2.0)
?.. ==> (1.0, 2.0)
```

- More details on `toString` when we discuss method overriding

Defining a `UnitCircle`

- Suppose we would like to represent another unit-circle object with an arbitrary point as its centre

```
jshell> Circle createUnitCircle(Point centre) {
   ...> return new Circle(centre, 1.0);
   ...> }
|  created method createUnitCircle(Point)

jshell> createUnitCircle(new Point(1.0, 1.0))
$.. ==> Circle centered at (1.0, 1.0) with radius 1.0
```

- We could define an *overloaded* constructor in `Circle` class

```
class Circle {
    private final Point centre;
    private final double radius;

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

    Circle(Point centre) {
        this(centre, 1.0); // calls the two-argument constructor
    }
```

- **`this`**`(..)` calls another constructor that matches arguments

Inheritance: `UnitCircle` is a `Circle`

- Since a unit-circle is a circle, the **is-a** relationship is indicative of another OOP principle, namely **inheritance**
  - **is-a** relationship (inheritance): `UnitCircle` is a `Circle`
  - Sub-classing: `Circle` is the parent(super) class, while `UnitCircle` is the child(sub) class
  - **has-a** relationship (composition): `Circle` has a `Point`
- Define a sub-class `UnitCircle` with a constructor that invokes the parent `Circle`'s constructor

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

Inheritance: `UnitCircle` is a `Circle`

- The **`super`** keyword is used within the child class for the following purposes:
  - **`super`**`(..)` to access the parent's constructor
  - **`super`**`.radius` or **`super`**`.contains()` can be used to make reference to the parent's properties or methods
- Since `UnitCircle` is a `Circle`, `Circle` methods can be invoked from `UnitCircle` objects too

```
jshell> /open UnitCircle.java

jshell> new UnitCircle(new Point(1.0, 1.0))
$.. ==> Circle centered at (1.0, 1.0) with radius 1.0

jshell> new UnitCircle(new Point(1.0, 1.0)).contains(new Point(1.0, 1.0))
$.. ==> true

jshell> new UnitCircle(new Point(1.0, 1.0)).contains(new Point(2.0, 2.0))
$.. ==> false
```

Overriding `toString` method

- Invoking: `javadoc -d doc Circle.java`

```
public class Circle
extends java.lang.Object
...
public java.lang.String toString()

Returns a string representation of the Circle, showing its centre coordinates
and radius.

Overrides:
toString in class java.lang.Object

Returns:
a string representation of the Circle object.
```

- This indicates that there is an equivalent `toString` method being overridden in the `java.lang.Object` class from which `Circle` extends (inherits)

Overriding `toString` Method

- All classes in Java inherit from the `Object` class
  - Methods defined in the `Object` class can be called from all objects of its child classes
- An example is the `toString` method
  - When an expression in JShell evaluates to an object, it invokes the `toString` method of that object
- Explicitly defining this `toString` method in our classes **overrides** the same method that is inherited from `Object`
  - The annotation `@Override` indicates to the compiler that the method overrides the same one in the parent class

Overriding `equals` Method

- Another commonly overridden method is the `equals` method
- Within the `Object` class, the `equals` method compares if two object references are the same (i.e. refer to the same object)

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

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

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

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

- To have points with the same coordinate values deemed equal, we need to override the `equals` method inherited from `Object`

Overriding `equals` Method

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

```
@Override
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))
$.. ==> true
```

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

Overriding `equals` Method

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

```
@Override
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 there is a `midPoint` method

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

- Rather than "eyeballing" the outcome of the object's `toString` method

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

Exercise: Designing a Filled Circle

```
class Circle {
    private final double radius;

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

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

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

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

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

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

- How do we define the `FilledCircle` class?

Design #1: As a Stand-alone Class

```
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 * this.radius * this.radius;
    }

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

    Color getColor() {
        return this.color;
    }

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

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

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

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

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):

```
String output = "";
for (Circle circle : circles) {
    if (circle instanceof FilledCircle) {
        output = output + (FilledCircle) circle + "\n";
    } else if (circle instanceof Circle) {
        output = output + (Circle) circle + "\n";
    }
}
```

- 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

```
String output = "";
for (Circle circle : circles) {
    output = output + circle + "\n";
}
```

- 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

Method Overloading

- Methods of the same name can co-exist if the *signatures (number, type, and order 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 {
    void foo(int x) { ... }
    void foo(String x) { ...}
}
```

```
new A().foo(123)
new A().foo("123")
```

Overriding or Overloading?

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

```
@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?

```
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?*

Lecture Summary

- Understand the object-oriented principles of abstraction, encapsulation, inheritance and polymorphism
- Distinguish between an is-a relationship and a has-a relationship, and choose the appropriate one during object-oriented design
- Extend the mental model of program execution for an object to include 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