CS2030 Lecture 3

Polymorphism

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

Semester 1 2021 / 2022

Lecture Outline and Learning Outcomes

- □ Appreciate the motivation behind the **substitutability principle**
- □ Understand the relationship between **inheritance** and **polymorphism**
- □ Distinguish between **compile-time** and **run-time** types
- □ Know the difference between **static (early) and dynamic (late) binding** in overloaded and overriding methods, and their relation to compile-time and run-time types
- □ Be able to define an appropriate overriding `equals` method
- □ Appreciate the substitution principle in context of *return types and accessibility of overriding methods*

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."

□ The **substitutability** principle says 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

Polymorphism

□ Poly–morphism: *many–forms*

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

jshell> fc.getArea()
$.. ==> 3.141592653589793

jshell> fc.fillColor(Color.RED)
$.. ==> area 3.14, perimeter 6.28, Color[r=255,g=0,b=0]

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

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

jshell> c.getArea()
$.. ==> 3.141592653589793

jshell> c.fillColor(Color.RED) // but isn’t c referencing FilledCircle?
|  Error:
|  cannot find symbol
|    symbol:   method fillColor(java.awt.Color)
|  c.fillColor(Color.RED)
|  ^---------^
```

Polymorphism

- □ Passing parameters — assignment across methods

```
jshell> String foo(Circle c) { // Circle or FilledCircle can be passed to c
   ...> double area = c.getArea(); // ok
   ...> c = c.fillColor(Color.RED); // ??
   ...> return c.toString(); // which toString ??
   ...> }
|  created method foo(Circle), however, it cannot be invoked
|  until method fillColor(java.awt.Color) is declared
```

- □ Converting between super-type and sub-type

```
jshell> c = fc // sub-type to super-type widening conversion
c ==> area 3.14, perimeter 6.28, Color[r=0,g=0,b=255]

jshell> fc = c // super-type to sub-type narrowing conversion
|  Error:
|  incompatible types: Circle cannot be converted to FilledCircle
|  fc = c
|       ^

jshell> fc = (FilledCircle) c // Risky! Is c referencing Circle or FilledCircle
fc ==> area 3.14, perimeter 6.28, java.awt.Color[r=0,g=0,b=255]
```

Compile-Time vs Run-Time Type

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

□ variable `c` 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. `c.getArea`, but not `c.fillColor`

□ `c` has a **run-time type** of `FilledCircle`

- the type of the object that the variable is pointing to
- determines the actual method called *during runtime*, e.g. `FilledCircle::toString()`, not `Circle::toString()`

□ A variable's compile-type is determined at compile time; its run-time type varies depending on the object assigned to it

Testing Object Equality

- Comparing two objects using the `==` operator returns **`true`** only if both refers to the same object instance

```
jshell> new Circle(1.0) == new Circle(1.0)
$.. ==> false
```

- Inherited `Object::equals(Object)` method same as `==`

```
jshell> new Circle(1.0).equals(new Circle(1.0))
$.. ==> false
```

- However, if we compare the `String` objects (returned from the `toString` method) using `equals`

```
jshell> new Circle(1.0).toString() == new Circle(1.0).toString()
$.. ==> false
```

```
jshell> new Circle(1.0).toString().equals(new Circle(1.0).toString())
$.. ==> true
```

  – Despite distinct instances, `String::equals` returns **`true`**

Overloading `Circle::equals`

□ Let’s define `equals(Circle)` to overload `equals(Object)`

```
class Circle {
    ...
    boolean equals(Circle c) { // Overloads equals(Object) method
        return Math.abs(this.radius - c.radius) < 1e-15;
    }
```

```
jshell> new Circle(1.0).equals(new Circle(1.0)) // equals(Circle)
$.. ==> true

jshell> new Circle(1.0).equals(new Circle(2.0)) // equals(Circle)
$.. ==> false

jshell> new Circle(1.0).equals("Circle") // equals(Object)
$.. ==> false
```

□ Which overloaded method is called in the following?

```
jshell> new Circle(1.0).equals(new FilledCircle(1.0, Color.BLUE))
$.. ==> true
```

Static Binding

□ **Static (Early) binding**

- the compile-time type decides which overloaded `equals` method to call *during compilation*

□ Which `equals` method is called?

```
jshell> Circle c1 = new Circle(1.0)
c1 ==> area 3.14, perimeter 6.28

jshell> Object o1 = new Circle(1.0)
o1 ==> area 3.14, perimeter 6.28

jshell> o1.equals(c1)
.. ==> false

jshell> c1.equals(o1)
.. ==> false
```

□ How to make the outcome **`true`** instead?

Overriding `Object::equals(Object)`

- Let's override the `Object::equals(Object)` method

```
class Circle {
    ...
    @Override
    public boolean equals(Object obj) {
        Circle c = (Circle) obj;
        return this.radius - c.radius < 1e-15;
    }
```

```
jshell> c1.equals(o1)
$.. ==> true

jshell> o1.equals(c1)
$.. ==> true
```

- During *compile-time*, which `equals` method to call?
- During *run-time*, which `method` is actually called?
- Since the `equals` method takes in `Object`, need to type-cast `Object` to `Circle` before accessing the radius

Overriding **equals**

- But what if an object of a different type is passed to **equals**?
  - A ClassCastException is thrown during runtime
- With a good sense of type awareness, the correct way to override the **equals** method is

```java
class Circle {
    ...
    @Override
    public boolean equals(Object obj) {
        if (this == obj) { // same object?
            return true;
        } else if (obj instanceof Circle) { // same type?
            Circle c = (Circle) obj;
            return this.radius - c.radius < 1e-15; // equals?
        } else {
            return false;
        }
    }
```

Polymorphism and Dynamic Binding

- In contrast to static binding in overloaded methods, dynamic (or late) binding occurs in overriding methods
- **Dynamic Binding**
  - the exact `equals` method to invoke (e.g. `Object` or `Circle`) is not known *until runtime*
- Which `equals(Object)` method is invoked below?

```
jshell> boolean isUnitCircle(Object obj) {
   ...> return obj.equals(new Circle(1.0));
   ...> }
|  created method isUnitCircle(Object)

jshell> isUnitCircle(new Circle(1.0))
$.. ==> true

jshell> isUnitCircle("Circle")
$.. ==> false
```

Overriding or Overloading?

□ Having considered defining `equals` as both an overloading and overriding method, which one works?

□ Using an overloaded method

```
jshell> new Circle(1.0).equals(new Circle(1.0))
$.. ==> true

jshell> new Circle(1.0).equals((Object) new Circle(1.0))
$.. ==> false
```

□ Using an overriding method

```
jshell> new Circle(1.0).equals(new Circle(1.0))
$.. ==> true

jshell> ((Object) new Circle(1.0)).equals((Object) new Circle(1.0)
$.. ==> true
```

- client cannot invoke an overridden method

LSP and Type/Sub-type Consistency

- Consider the following classes A and B

```java
import java.awt.Color;

class A {
    Circle foo() {
        return new Circle(1.0);
    }
}

class B extends A {
    @Override
    FilledCircle foo() {  // FilledCircle and not Circle?
        return new FilledCircle(1.0, Color.BLUE);
    }
}
```

- Does the above compile?
- What are the possible valid return types of method `B::foo()` that can override `A::foo()`?

LSP and Type/Sub-type Consistency

□ Consider how clients could use a variable of type `A`

```
jshell> Circle bar(A a) {
   ...> return a.foo();
   ...> }
|  created method bar(A)

jshell> Circle c = bar(new A())
c ==> area 3.14, perimeter 6.28

jshell> Circle c = bar(new B())
c ==> area 3.14, perimeter 6.28, Color[r=0,g=0,b=255]
```

□ Return type cannot be more general than that of the overridden method

□ How about parameter types?

- cannot be more specific than the overridden method?
- *but don't forget method overloading...*

LSP and Accessibility

- How about the accessibility modifier of the methods?

```java
import java.awt.Color;

class A {
    Circle foo() {
        return new Circle(1.0);
    }
}

class B extends A {
    @Override
    private FilledCircle foo() { // private modifier
        return new FilledCircle(1.0, Color.BLUE);
    }
}
```

- Does the above compile?
- Accessibility modifier cannot be more restricted than that of the overridden method