CS2030 Lecture 4

Inheritance and Polymorphism

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

Semester 1 2023 / 2024

Outline and Learning Outcomes

- Adherence to the abstraction principle using **inheritance**
- Able to construct **super** (parent) and **sub** (child) classes to realize an inheritance relationship
- Able to model an object to include inheritance
- Understand how inheritance can be used to support **polymorphism**
- Distinguish between **method overriding** and **method overloading**
- Understand **compile-time type** in static binding and **runtime-type** in dynamic binding
- Appreciate the motivation behind the **substitutability** principle

Circle and Filled Circle

- Given the following `Circle` class with `getArea()` method, implement `FilledCircle` class with radius and color

```
class Circle implements Shape {
    private final double radius;

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

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

    public String toString() {
        return "circle with area " + String.format("%.2f", this.getArea());
    }
}
```

- **Abstraction Principle**: *Each significant piece of functionality in a program should be implemented in just one place in the source code. 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*

Sub-Classing with Inheritance

- A child/sub class inherits (extends) from a parent/super class

```
import java.awt.Color;

class FilledCircle extends Circle {
    private final Color color;

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

    public String toString() {
        return "filled " + super.toString() + ", color " + this.color;
    }
}
```

- **super** keyword can be used within the child class:
  - `super.radius` or `super.toString()` to refer to the parent's properties or calling the parent's methods
  - `super(..)` to access the parent's constructor
  - unlike **this**, **super** is not a reference

`protected` Access Modifier

- Where necessary, properties/methods in the super class can be made accessible to a sub-class using the **`protected`** modifier

```
class Circle implements Shape {
    protected final double radius;

    Circle(double radius) {
        this.radius = radius;
    }...
}
class FilledCircle extends Circle {
    private final Color color;

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

    FilledCircle fillColor(Color color) {
        return new FilledCircle(super.radius, color);
    }...
}
```

- Note that **`protected`** gives access to properties/methods to all other classes (not only sub-classes) *within the same package*

Inheritance

- Any behaviour of an object of class `T` *can be invoked* from an object of its sub-class `S`
  - e.g. since `FilledCircle` is a `Circle`, `Circle` methods *can be invoked* from `FilledCircle` objects too

```
jshell> new Circle(1.0)
$.. ==> circle with area 3.14

jshell> new Circle(1.0).getArea()
$.. ==> 3.141592653589793

jshell> new FilledCircle(1.0, Color.BLUE)
$.. ==> filled circle with area 3.14, color java.awt.Color[r=0,g=0,b=255]

jshell> new FilledCircle(1.0, Color.BLUE).getArea()
$.. ==> 3.141592653589793
```

- Unlike interfaces, a child class **cannot** extend from multiple parents; **`class A extends B, C {...}`** is invalid!

Method Overriding

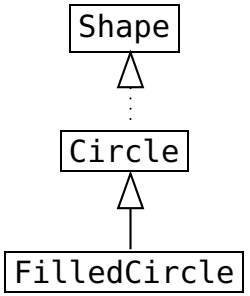
- Defining a method explicitly in a child class that has already been defined in the parent, e.g. `toString()` method
  - all classes inherit from `java.lang.Object`
  - defining a `toString()` method in a sub-class **overrides** the one that is inherited from the parent class

```
jshell> new Circle(1.0) // invokes Circle's toString()
$.. ==> circle with area 3.14

jshell> new FilledCircle(1.0, Color.BLUE) // invokes FilledCircle's toString()
$.. ==> filled circle with area 3.14, color java.awt.Color[r=0,g=0,b=255]
```

- the `@Override` annotation indicates to the compiler that the method overrides the same one in the parent class
  - useful in ensuring that we are overriding the right method

Is-A Relationship

- `FilledCircle` *is a* `Circle`; `Circle` *is a* `Shape`

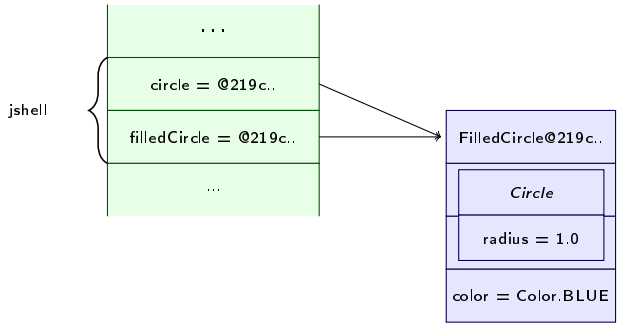



```
jshell> Circle c = new Circle(1.0)
c ==> circle with area 3.14

jshell> c.toString()
$.. ==> "circle with area 3.14"

jshell> Shape s = circle // Circle is a Shape
s ==> circle with area 3.14

jshell> s.toString()
$.. ==> "circle with area 3.14"

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

jshell> c = fc // FilledCircle is a Circle
c ==> filled circle with area 3.14, color java.awt.Color[r=0,g=0,b=255]

jshell> c.toString()
$.. ==> "filled circle with area 3.14, color java.awt.Color[r=0,g=0,b=255]"

jshell> s = fc // FilledCircle is a Shape
s ==> filled circle with area 3.14, color java.awt.Color[r=0,g=0,b=255]

jshell> s.toString()
$.. ==> "filled circle with area 3.14, color java.awt.Color[r=0,g=0,b=255]"
```

Modeling Inheritance

- Java memory model for the statement

```
jshell> FilledCircle filledCircle = new FilledCircle(1.0, Color.BLUE)
filledCircle ==> filled circle with area 3.14, java.awt.Color[r=0,g=0,b=255]

jshell> Circle circle = filledCircle // circle assigned with reference to FilledCircle object
circle ==> filled circle with area 3.14, java.awt.Color[r=0,g=0,b=255]
```



- Notice how the child object "wraps-around" the parent

Compile-Time vs Run-Time Type

- Consider the following assignment statement:
  `Circle circle = new FilledCircle(1.0, Color.BLUE);`
- variable `circle` has a **compile-time type** of `Circle`
  - the type of the variable declared
  - restricts the *methods it can call during compilation*
    - e.g. `getArea()`, `toString()`, but not `fillColor(Color)`
- `circle` has a **run-time type** of `FilledCircle`
  - the type of the object that the variable is referencing
  - determines the *actual method called during runtime*
    - e.g. `FilledCircle::toString()`, not `Circle::toString()`

Polymorphism

- Poly–morphism means "many forms"
- Suppose `S` is a *sub-type* of `T`,
  - a child class is a sub-type of its parent
  - a class that implements an interface is a sub-type of that interface
- *Substitutability*: variable of type `T` can be assigned with an object of type `T` or sub-type `S` (or `S` is *substitutable* for `T`)

```
jshell> Circle circle = new FilledCircle(1.0, Color.BLUE)
circle ==> filled circle with area 3.14, java.awt.Color[r=0,g=0,b=255]

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

jshell> circle.fillColor(Color.GREEN) // compile-time type decides whether method can be called
|  Error:
|  cannot find symbol
|    symbol:   method fillColor(java.awt.Color)
|  circle.fillColor(Color.GREEN)
|  ^--------------^
```


Polymorphism

- Another example of polymorphism
  - parameter passing, i.e. assignment across methods

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

  - Not compilable as `fillColor` not defined in `Circle` class!
- `Circle` and `FilledCircle` objects can be passed to `foo`
  - `foo` only knows that parameter `circle` would eventually reference "something" that behaves like a `Circle`
  - implementation in the method body must work for `Circle` objects, as well as objects of its sub-classes

Method Overloading

- Methods of the same name can co-exist if their method *signatures (number, type, order of arguments)* are different
  - e.g. defining an overloaded method `toString(String)` in Circle class

```
String toString(String prompt) {
    return prompt + " " + this.toString(); // calls toString()
}
```

```
jshell> Circle circle = new Circle(1.0)
c ==> circle with area 3.14

jshell> circle.toString()
$.. ==> "circle with area 3.14"

jshell> circle.toString("myshell>")
$.. ==> "myshell> circle with area 3.14"

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

jshell> circle.toString()
$.. ==> "filled circle with area 3.14, color java.awt.Color[r=0,g=0,b=255]"

jshell> circle.toString("myshell>")
$.. ==> "myshell> filled circle with area 3.14, color java.awt.Color[r=0,g=0,b=255]"
```

  - What if `toString(String)` is defined in `FilledCircle` instead?

Method Overloading

- Method overloading is very common among constructors within the same class

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

Circle() { // circle with default radius 1.0
    this.radius = 1.0;
}
```

- Use **`this(..)`** to call one constructor from another

```
Circle() {
    this(1.0);
}
```

- Note that **`this(..)`** or **`super(..)`** must be the first statement within the body of the constructor

Static vs Dynamic Binding

- During compilation, *static binding* resolves the method to call (including overloaded methods)
  - in the following, `circle.toString()` calls the `toString()` method with no arguments

```
jshell> ImList<Circle> circles = new ImList<Circle>().
   ...> add(new Circle(1.0)).
   ...> add(new FilledCircle(1.0, Color.BLUE))
circles ==> [circle with area 3.14, circle with area 3.14, java.awt.Color[r=0,g=0,b=255]]

jshell> for (Circle circle : circles) {
   ...>     System.out.println(circle); // or System.out.println(circle.toString())
   ...> }
circle with area 3.14
circle with area 3.14, java.awt.Color[r=0,g=0,b=255]
```

- During runtime, *dynamic binding* resolves the actual method that is called among all overriding methods
  - if `circle` references an object of type `FilledCircle`, the `toString()` method of `FilledCircle` is invoked

From Concrete Class to Interfaces

- **Concrete class** defines the actual implementation with data (properties) and behaviour (methods)
- **Interface** specifies methods to be implemented, with no data
- **Abstract class** is a trade off between the two
  - can have properties to be inherited by child classes
  - can have some methods defined; hence cannot instantiate

```
abstract class FilledShape {
    protected final Color color;

    FilledShape(Color color) {
        this.color = color;
    }

    // declare method as abstract
    abstract double getArea();

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

```
class FilledCircle extends FilledShape {
    private final int radius;

    FilledCircle(int radius, Color color) {
        super(color);
        this.radius = radius;
    }
    double getArea() {
        return Math.PI * radius * radius;
    }
}
```

- Multiple inheritance, even for abstract classes, is not allowed

fyi, as of Java 8 "impure" interfaces can include default methods with implementations; in CS2030 we use only "pure" interfaces.