CS2030 Programming Methodology II

Lecture 2

Shengdong Zhao

Spring 2019

Acknowledge: slides are adapted from Henry Chia

Lecture Outline

- OO Principles
  - Abstraction
  - Encapsulation
  - Inheritance
    - Super–sub (Parent–child) classes
  - Polymorphism
    - Dynamic vs Static binding
  - Method overloading
  - Mental-modeling objects with inheritance
  - Class variables and methods

A Simplified Circle class

- We consider a simplified version of the Circle class

```java
public class Circle {
  private double radius;
  public Circle(double radius) {
    this.radius = radius;
  }
  public static void main(String[] args) {
    Circle circle = new Circle(1.0);
    System.out.println(circle);
  }
}
```

- What is the output when the above is compiled and run?
- How do we test the Circle class without explicitly writing another Java class?

A Simplified Circle class

- jshell was introduced in Java 9 to provide an interactive shell
  - allows us to enter a command that is immediately executed with result displayed
  - uses REPL to provide an immediate feedback loop

```
$ jshell Circle.java
| Welcome to JShell -- Version 9.0.4
| For an introduction type: /help intro

jshell> Circle c = new Circle(1.0)
c ==> Circle@5f341870

jshell> /exit
| Goodbye
```

Printing the Circle class

- Suppose we would like to create a Circle object and output in the following format:

```
$ jshell Circle.java
| Welcome to JShell -- Version 9.0.4
| For an introduction type: /help intro
jshell> Circle c = new Circle(1.0)
c ==> Circle with area 3.14 and perimeter 6.28
jshell> /exit
| Goodbye
```

- What are the attributes and methods of the Circle class?
- Specifically, you will need to define an overriding toString method

Overriding toString method

- The toString method of the Circle class can be defined as:

```
/**
* Returns a string representation of the Circle, showing
its centre coordinates, area and perimeter.
*
* @return a string representation of the Circle object.
*/
@Override
public String toString() {
  return "Circle with area " +
    String.format("%.2f", getArea()) + " and perimeter "+
    String.format("%.2f", getPerimeter());
}
```

- The annotation @Override indicates to the compiler that the method overrides another one

Overriding toString method

- Invoking javadoc Circle.java produces the following:

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

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

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)

Object's equals Method

- The other commonly overridden method is the equals method
- Within the Object class, the equals method compares if two object references refer to the same object
- As an example, consider the following

```
jshell> Circle c1 = new Circle(1.0);
c1 ==> Circle with area 3.14 and perimeter 6.28

jshell> Circle c2 = new Circle(1.0);
c2 ==> Circle with area 3.14 and perimeter 6.28

jshell> c1 == c2
$4 ==> false

jshell> c1.equals(c2)
$5 ==> false
```

- If circles of the same radius are deemed equal, then we need to override the equals method inherited from Object

Overriding Object's equals Method

- A naïve way of overriding equals method is to include the following method in the Circle class

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

class casting, change the scope to look for info

- Since the equals method takes in a parameter of Object
  - type-cast obj from Object type to Circle type before accessing the radius in order to check for equality
- But what if the equals method of Circle was invoked as

```
(new Circle(1.0)).equals(new Point(0.0, 0.0))
```

  - A ClassCastException is thrown

Overriding Object's equals Method

- Hence, with a sense of type awareness, the correct way to override the equals method is

```
@Override
public boolean equals(Object obj) {
  if (this == obj) {
    return true;
  } else if (obj instanceof Circle) {
    return this.radius ==
      ((Circle) obj).radius;
  } 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

Designing a Filled Circle

- Suppose we would like to create a FilledCircle object that is a circle filled with a color

```
jshell> /open FilledCircle.java
jshell> new FilledCircle(1.0, Color.BLUE)
$3 ==> Circle with area 3.14, perimeter 6.28
and color java.awt.Color[r=0,g=0,b=255]
```

- Uses the Color class provided by Java

```
import java.awt.Color;
```

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

Design #1: As a Standalone Class

```
import java.awt.Color;
public class FilledCircle {
  private double radius;
  private 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 "Filled Circle with area " +String.format("%.2f", getArea())+
      ", perimeter " + String.format("%.2f", getPerimeter()) +
      "\nand color " + getColor();
  }
}
```

constructor

not good, functions defined under Circle, need type again

Design #2: Using Composition

- has-a relationship: FilledCircle has a Circle

```
public class FilledCircle {
  private Circle circle;
  private 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 "Filled Circle with area " +String.format("%.2f", getArea())+
      ", perimeter " + String.format("%.2f", getPerimeter()) +
      "\nand color " + getColor();
  }
}
```

Mental Modeling

- is-a relationship: FilledCircle is a Circle

```
import java.awt.Color;
public class FilledCircle extends Circle {
  private Color color;
  public FilledCircle(double radius, Color color) {
    super(radius);
    this.color = color;
  }
  public Color getColor() {
    return color;
  }
  @Override
  public String toString() {
    return "Filled Circle with area " +String.format("%.2f", getArea())+
      ", perimeter " + String.format("%.2f", getPerimeter()) +
      "\nand color " + getColor();
  }
}
```

- Circle is the parent(super) class, while FilledCircle is the child(sub) class

Inheritance

- Notice the child class 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

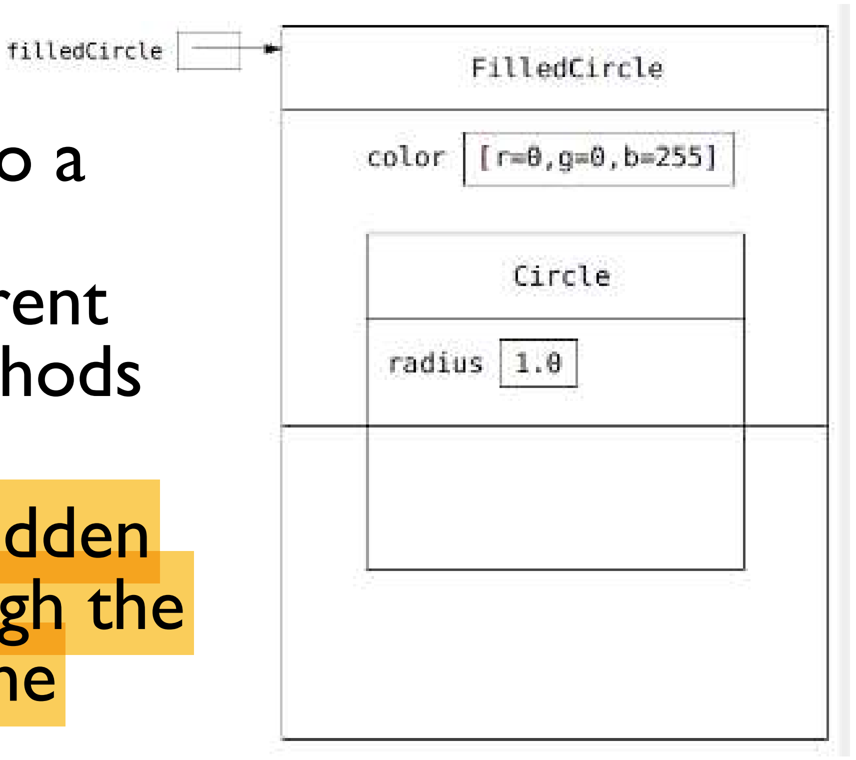
```
public class Circle {
  protected double radius;
  ...
```

accessible to child class use protected

- The super keyword is used for the following purposes:
  - super(..) to access the parent's constructor
  - super.radius or super.getArea() 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



if child class overrides, will call child class’ method even call from parent

Inheritance Misuse

- Do not confuse a has-a relationship with is-a
- Despite that the classes on the left is functional, it does not make sense!

```
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 with radius " + radius +
      " centered at " + super.toString();
  }
}
```

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 with area 3.14 and perimeter 6.28
jshell> c = new FilledCircle(1.0, Color.BLUE)
c ==> Filled Circle with area 3.14, perimeter 6.28
and ... a.awt.Color[r=0,g=0,b=255]
jshell> FilledCircle fc = new FilledCircle(1.0, Color.BLUE)
fc ==> Filled Circle with area 3.14, perimeter 6.28
and ... a.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, 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;
}
```

- Whichever method is called is determined during compile time

```
Circle c1 = new Circle();
Circle c2 = new Circle(1.2);
```

- 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
  - 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

Class Variables and Methods

- Having gone through designing a class and allowing objects of that class to be created, how do we keep track of the number objects instantiated at any point of time?
- Clearly, such an aggregate value cannot be stored in every object, since every new instance created would entail that this value be updated in every object
- Use static modifier to create class variables and methods

```java
public class Circle {
  private double radius;
  private static int numOfCircles = 0;
  public Circle(double radius) {
    this.radius = radius;
    numOfCircles++;
  }
  public static int getNumOfCircles() {
    return numOfCircles;
  }
```

Class Variables and Methods

- Class variables and methods can be called through the class or the object
- Calling through the class is preferred as it makes clear the intent

```
jshell> Circle c = new Circle(1.0)
c ==> Circle with area 3.14 and perimeter 6.28
jshell> FilledCircle fc = new FilledCircle(2.3, Color.BLUE)
fc ==> Filled Circle with area 16.62, perimeter 14.45
an ... a.awt.Color[r=0,g=0,b=255]
jshell> c = new FilledCircle(8.9, Color.WHITE)
c ==> Filled Circle with area 248.85, perimeter 55.92
a ... t.Color[r=255,g=255,b=255]
jshell> Circle.getNumOfCircles()
$7 ==> 3
jshell> c.getNumOfCircles()
$8 ==> 3
jshell> fc.getNumOfCircles()
$9 ==> 3
```

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 use of class variables and methods for aggregation purposes