Java Lang Review, Part II

Properties of classes and objects

Jianjun Chen (Jianjun.Chen@xjtlu.edu.cn)

Class and Object

- It's helpful to group related variables and their functions together.
- Class serves as the blueprint that allows instances of such bundles (objects) to be created easily.

```java
public class Bicycle {
    int speed = 0;
    int gear = 1;

    void changeGear(int newValue) {
        gear = newValue;
    }

    void speedUp(int increment) {
        speed = speed + increment;
    }

    void applyBrakes(int decrement) {
        speed = speed - decrement;
    }

    void printStates() {
        System.out.println(" speed:" +
                speed + " gear:" + gear);
    }
}
```

```java
public class BicycleDemo {
    public static void main(String[] args) {
        // Create two different Bicycle objects
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();

        // Invoke methods on those objects
        bike1.speedUp(10);
        bike1.changeGear(2);
        bike1.printStates();

        bike2.speedUp(10);
        bike2.changeGear(2);
```

Method Overloading

- Methods with the same name can exist in one class, given that their method signatures are different.
  - A <u>method signature</u> is the method name and the number, type and order of its parameters.

- Recall how you invoke a function:
  - `foo(params);`
- If such an expression does not lead to ambiguity, then it's usually fine.

Method Overloading

- In the example below, functions 1, 2 and 3 are correct method overloading.
- Functions (2, 4) are ambiguous.

```
1. static void foo(int x, int y) {}
2. static void foo(int x) {}
3. static void foo(int f, double z) {}
4. static int foo(int x) {return 1;}
```

- Remember: the method name and the number, type and order of its parameters.

Static/Non-static functions

- A non-static function can be considered as a static function that takes an additional object called "this" as its parameter.

- That's why you can't call non-static functions without creating an object first.
  - `Object.function()` rather than `function(object)` is just a language design.

Static/Non-static functions

```java
public class Test {
    public int x = 0;

    static void setX1(Test thisobj) {
        thisobj.x = 1;
    }

    void setX2() {
        this.x = 1;
    }
}
```

- `setX1()` and `setX2()` are functionally equivalent.

```java
public class TestUser {
    public static void main(String[] args) {
        Test t = new Test();
        Test.setX1(t); // sets t.x to 1
        t.setX2(); // also sets t.x to 1
    }
}
```

Static/Non-static Variables

- Static variables are class variables.
  - Initialised when classes are defined.
  - Expressions used in variable definitions inside classes are run before main function is called. (See the next slide)

- Non-static variables are object variables.
  - Initialised when objects are created.
  - Expressions used in variable definitions inside classes are run before the constructor is called. (See the next slide)

Initialisation Order of Members

```java
import java.io.PrintStream;

public class StaticTest {
    static PrintStream x = System.out.printf("var x\n");
    PrintStream y = System.out.printf("var y\n");

    public StaticTest() {
        System.out.println("constructor");
    }

    public static void main(String args[]) {
        System.out.println("main func");
        new StaticTest();
    }
}
```

```
$ java StaticTest
var x
main func
var y
constructor
```

Access Modifiers

A class can inherit another class from another package, thus it's placed between "package" and "world"

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Set up a project to test these

Follow the tutorial here if you forget something:

https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Type Casts

- Casting an object to a more general type (superclass) is called **upcasting**, and is always legal.
  - `Object obj = new String();`

- Casting an object to a more specific type (subclass) is called **downcasting**, and Java inserts a run-time check to ensure that the cast is legal.
  - `String str = (String) obj;`

Type Casts

- Java will implicitly cast subclass to superclass when necessary.
- For example, we can call function `f(Object x)` using a `String` object.
  - Because `String` is a subclass of `Object`.
- Also works for interfaces and those classes that implements interfaces.

```
static void f(Object x) {
    System.out.println(x);
}
```

```
public static void main(String[] args) {
    f("");
}
```

Inheritance

- Reuse the fields and methods of the existing class without having to write them yourself.
  - The subclass contains a copy of the superclass

- Constructors are not inherited
- The constructors of the base class is only invoked when you put `super()` at the first line of your subclass constructor.
  - Because base classes should be initialised first.

Inheritance: Example

```java
public class A {
    public int x = 2;

    public void printX() {
        System.out.println("x is " + x);
    }
}
```

```java
public class B extends A {
    public int y = 3;
    public void printY() {
        System.out.println("y is " + y);
    }

    public void printX(int add) {
        System.out.println("x+" + add + " is "
                + (x + add));
    }
}
```

```java
public static void main(String[] args) {
    B b = new B();
    b.printX(); // function of class A
    b.printY(); // function of class B
    b.printX(5); // overloaded function
}
```

B can call functions of A and use A's variables

Inheritance: Identifier Lookup

```
int x = 5;
```

Domain of A

```
void printX() {
    println(x);
}
```

Domain of B

```
void printX(int add) {
    println(x + add);
}
```

Identifier x requested from `A.printX()`

1. Is x a local variable of `printX()`? No.
2. Is x in Class A? Found!

Identifier x requested from `B.printX(int add)`

1. Is x a local variable of `printX(int add)`? No.
2. Is x in Class B? No.
3. Is x in A? Found!

Test 1

```
int x = 5;

void printX() {
    println(x);
}
```

Domain of A

```
int x = 7;

void printX(int add) {
    println(x + add);
}
```

Domain of B

Now consider:

```
B b = new B();
b.printX();
b.printX(5);
```

What is the result?

Test 1

- The result is: 5 and then 12

- An object of A is contained within B.
  - Access to `A.printX()` is not “blocked”.

- `A.printX()` will always use `A.x` instead of `B.x`.
  - Respecting original author’s choice.

Test 2

```
int x = 5;

void printX() {
   println(x);
}
```

Domain of A

```
int x = 7;
```

Domain of B

Now consider:

```
B b = new B();
b.printX();
((A) b).printX();
```

What is the result?

Test 2

- The result is: 5 and then 5

- B does not have `printX()`, Java will use `A.printX()` instead.
  - Which in term uses `A.x`

Test 3

Domain of A

```
int x = 5;

void printX() {
   println(x);
}
```

Domain of B

```
int x = 7;

void printX() {
   println(x);
}
```

Now consider:

```
B b = new B();
b.printX();
((A) b).printX();
```

What is the result?

Test 3: Override

- The result is: 7 and then 7

- The original object is of type B. Any of its behaviours will first respect its designer's decision.
  - If `x` or `printX()` is not defined, it will look for base class definitions.
- we are <u>overriding</u> the original `printX()` in Class A. To call the original one:
  - In B, use `super.printX()`
  - Outside of B: You cannot call `A.printX()` using the object of B.

@Override

- A common mistake is that we overload by accident when we actually intended to override.
  - i.e. we don't want `printX()` to be called through B.
- By adding @Override before function definition, The Java compiler will report error if you accidentally override.
  - see: https://www.baeldung.com/java-override

```java
public class B extends A {
    public int y = 3;
    public void printY() {
        System.out.println("y is " + y);
    }

    @Override
    public void printX(int add) {
        System.out.println("x+" + add + " is "
                + (x + add));
    }
}
```

Abstract

- An abstract method is a method that is declared but not defined.
- An abstract class contains one or more abstract methods
  - it must itself be declared with the **abstract** keyword.
- Abstract classes cannot be instantiated, but they can be subclassed.

```
abstract class GraphicObject {
    int x, y;
    void moveTo(int newX, int newY) {
        this.x = newX;
        this.y = newY;
    }
    abstract void draw();
    abstract void resize();
}
```

```
class Circle extends GraphicObject {
    void draw() {
        //code implementing draw.
    }
    void resize() {
        //code implementing resize.
    }
}
```

Interface

```
interface Drawable{
    void draw(int color);
    void setPosition(double x, double y);
}
```

- An interface may contain only public abstract methods and definitions of constants
- Methods in interfaces do not have bodies
- Interface methods can only be **implemented** by classes

```
public class Point implements Drawable{
    public double x = 0;
    public double y = 0;

    //constructor
    public Point(double a, double b) {
        x = a;
        y = b;
    }

    @Override public void draw(int color) {
        System.out.println("Point drawn at " +
                toString() + " in color " + color);
    }

    @Override public void setPosition(double x, double y) {
        this.x = x;
        this.y = y;
    }

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

Homework

Think about the differences between

- Class A implements interface B
- Class A extends Class B

What are the suitable situations to use them?

Check discussions online.