
Java 程序设计 LAB05

实验目的：

- 理解静多态和动多态的概念
- 理解多态的必要性和实现机制
- 理解并灵活使用方法重载和方法覆盖
- 理解并灵活使用抽象类和抽象方法
- 理解接口的必要性（将接口用作 API）
- 掌握如何定义接口、实现接口
- 将接口用作类型、使用接口回调
- 理解并掌握接口的继承
- 面向接口的编程

实验题目

1. 多态 1

阅读下面这段代码：

```
// Test.java

class PrivateOverride {

    private void f() { System.out.println("private f()"); }

    public static void main(String[] args) {

        PrivateOverride po = new Derived();

        po.f();

    }

}

class Derived extends PrivateOverride {

    public void f() { System.out.println("public f()"); }

}

public class Test {

    public static void main(String[] args) {

        PrivateOverride.main(args);

    }

}
```

运行 `java Test`，程序的输出是什么？

如果将父类中的方法声明为 `public`，而子类为 `private`，编译能通过吗？如果能，最后会输出什么？

题外话:

private 方法被默认是 final 的

2. 多态 2

阅读下面这段代码:

```
// Test.java
class Super {
    public int field = 0;
    public int getField() { return field; }
}

class Sub extends Super {
    public int field = 1;
    public int getField() { return field; }
    public int getSuperField() { return super.field; }
}

public class Test {
    public static void main(String[] args) {
        Super sup = new Sub(); // Upcast
        System.out.println("sup.field = " + sup.field +
            ", sup.getField() = " + sup.getField());
        Sub sub = new Sub();
        System.out.println("sub.field = " + sub.field +
            ", sub.getField() = " + sub.getField() +
            ", sub.getSuperField() = " + sub.getSuperField());
    }
}
```

运行 java Test, 程序的输出是什么? 类的非静态属性能体现多态性吗?

3. 多态 3

阅读下面这段代码:

```
// Test.java
class StaticSuper {
    public static String staticGet() {
        return "Base staticGet()";
    }
}
```

```

    }

    public String dynamicGet() {
        return "Base dynamicGet()";
    }
}

class StaticSub extends StaticSuper {
    public static String staticGet() {
        return "Derived staticGet()";
    }

    public String dynamicGet() {
        return "Derived dynamicGet()";
    }
}

public class Test {
    public static void main(String[] args) {
        StaticSuper sup = new StaticSub(); // Upcast
        System.out.println(sup.staticGet());
        System.out.println(sup.dynamicGet());
    }
}

```

运行 `java Test`, 程序的输出是什么? 类的静态属性和静态方法能体现多态性吗?

4. 多态 4

阅读下面这段有点儿长的代码:

```

// Test.java

class A {
    void draw() { System.out.println("A.draw()"); }

    A() {
        System.out.println("A() before draw()");
        draw();
        System.out.println("A() after draw()");
    }
}

class B extends A {

```

```
private int b = 1;

B(int b) {
    this.b = b;
    System.out.println("B(), b = " + this.b);
}

void draw() {
    System.out.println("B.draw(), b = " + this.b);
}
}

public class Test {
    public static void main(String[] args) {
        new B(5);
    }
}
```

运行 `java Test`，程序的输出是什么？

结合之前实验的初始化顺序和多态，给出程序这样输出的解释。

5. ShapeFactory1

在 LAB04 的 Shape 的基础上，实现一个满足如下需求的 ShapeFactory 类：

- 提供一个 ShapeType 的枚举类，其中有表示矩形、菱形、椭圆的枚举量；
- 具有方法 `public Shape makeShape(ShapeType type, double a, double b)`，返回一个由 type 指定类型，a 和 b 指定大小的形状；
 - ◆ 参数不合法时，返回 null 或抛出异常
- 具有方法 `public Shape randomNextShape()`，返回一个随机类型，随机大小的形状；
 - ◆ 随机得到的形状要合法
 - ◆ 不能随机出来 null

编写测试类：

- 使用以上的两种生成形状的方式，分别随机生成五个形状并存储到 Shape 类型的数组（或其他容器）中，最后使用 foreach 循环将他们的面积输出

注意：

不能修改上一次的 Shape。

题外话：

这题不是设计模式中的工厂方法模式（factory method pattern）。

本题中的工厂，如果你为 Shape 添加了一个新的子类（比如直角三角形类），那么你就需要给 ShapeType 添加新的枚举量，还要重新写 makeShape 和 randomNextShape。

6. Overload? Override?

阅读下面这段代码：

```
// Test.java
interface I0 {
    void f(); // 默认是 abstract public 的
}

interface I1 {
    void f();
}

interface I2 {
    int a = 2; // 默认是 static public final 的
    int f();
}

interface I3 {
    int a = 3;
    int f(int i);
}

interface I4 {
    void f(int i);
}

class Test01 implements I0, I1{
    @Override
    void f() {}
}

class Test02 implements I0, I2{
    @Override
    void f() {}

    @Override
    int f() {return 0;}
}

class Test23 implements I2, I3{
    @Override
    int f() {return a;}

    @Override
    int f(int i) {return i;}
}
```

这段程序是无法通过编译的，都有哪些原因呢？尝试从继承、覆盖、重载的角度考虑。
如果 I1 extends I0，会引入新的错误吗？I2、I3 也 extends I0 呢？

题外话：

在任何支持多重继承的语言中，多个父类拥有相同的函数名都会带来误会，如果不是万不得已，千万不要这么做。

还有更多复杂的情况：

- 比如 class B extends A implements I
- 比如 class A implements I1, class B extends A implements I2
- 更多情况请自行尝试并理解。

7. 策略模式 (Strategy Pattern)

如果一些方法只保留了业务中逻辑固定不变的部分，只依据参数的不同来产生不同的行为，符合这样的方法，就是符合**策略模式 (Strategy Pattern)**。接口经常用于策略模式。

定义 `interface ITextProcess`：

- 具有方法 `String process(String s)`

利用 `interface ITextProcess` 完成几个类：

- `class Splitter`，其 `process` 方法将 `s` 中的所有空格都去掉。
- `class UpperCaseProcessor`，其 `process` 方法将 `s` 中的所有字符都变为大写的。
- `class SnakeCaseProcessor`，其 `process` 方法将 `s` 转变为 `snake_case` 的
 - ◆ `snake_case` 指的是不用空格而用下划线分隔单词
 - ◆ `I hate ddl -> I_hate_ddl`
 - ◆ `have a good time -> have_a_good_time`

编写测试类：

- 提供方法 `public static void process(ITextProcess p, String s)`，在其中使用 `p` 处理 `s`，并输出处理结果
- 在 `main` 中测试你的功能

题外话：

本题中，业务逻辑不变的是使用一个文本处理器处理文本，变化的是使用的处理器和文本内容。测试类的 `process` 方法是符合策略模式的。这样可以保证也实现了 `ITextProcess` 的类在添加到系统时，无需对原有代码产生影响，在这种情况下，保证了原有代码的可复用性。
`interface` 在这里的作用，就是定义一个标准，定义一种框架。

如果没有 `interface`，那么我们就需要像第五题的 `ShapeFacotry` 一样，通过参数指定处理器的类型，为原有代码增加更多的特殊情况判断。

使用 `interface` 而不使用继承的另一个原因是：现实场景下，`Splitter` 等可能需要继承其他类，而 `TextProcess` 并没有必须要有的属性，所以没有必要让他们都继承一个 `abstract class TextProcessor`。如果出现了不得不同时继承多个类的情况，后续实验会提到装饰器模式 (`Decorator Pattern`)，也可以通过拆解类结构来化继承为组合。

8. ShapeFactory2

在 LAB04 的 Shape 的基础上，定义一个满足如下需求的 IShapeFactory 接口：

- 具有方法 `Shape makeShape(double a, double b)`，返回一个由 a 和 b 指定大小的形状；
 - ◆ 参数不合法时，返回 null 或抛出异常

为每一种形状编写它的工厂类：

- 比如生成矩形的工厂类 RectangleFactory 要 implements 接口 IShapeFactory。

编写测试类 ShapeFactoriesTest：

- 具有 `static` 方法 `Shape makeShape(IShapeFactory factory, double a, double b)`，在其中使用 `factory.makeShape(a, b)` 方法生成形状并返回
- 在 `main` 方法中声明所有 3 种工厂，将他们存入一个 IShapeFactory 类型的数组 (或其他容器)
- 对工厂数组 (或容器) 使用 `foreach` 循环遍历，利用 `ShapeFactories.makeShape` 方法生成所有种类的形状并输出他们的面积

注意：

和第 5 题的 ShapeFactory1 没什么关系。

题外话：

这题才是设计模式中的工厂方法模式 (factory method pattern)，第五题虽然也是比较常用的“工厂”，但并不是工厂方法模式，甚至不是一个专门的设计模式。

工厂方法模式将实例化延迟到子类，由专门的工厂类生成特定类型的产品 (比如 Shape)。本次实验没有关于 Random 的需求，主要是因为设计上的问题，他和本题的初衷“简单了解工厂方法模式”不符，它不是特定的，它需要知道所有的类型信息，类似本题中的 ShapeFactoriesTest 的地位。

第五题中的方法，需要使用诸如 ShapeType 的标准来指定类型，这就导致了如果有新的需求 (新的形状种类) 出现，整个 ShapeFactory 类都要重新编写并编译。而使用工厂方法模式，你需要做的是编写一个新的工厂类并编译这个新的类，对原有的工厂代码无需进行修改 (重构过图书馆的同学应该能体会到“不用修改代码”是一种多么幸福的事)。

编写的时候可能觉得引入过多的类比较反人类，但是程序不是开发出来就完事了，还有维护和迭代更新。提倡在开工之前的设计环节为未来做足打算，但是也不要因此变成设计狂魔。上机题是为了在比较小的工作量下让大家了解基础内容，所以才会抽象出各种各样不现实的场景。如果像某次实验的“文件”那样，在工作量上并不友好。

本题的 ShapeFactories.makeShape 也是策略模式的应用。