类和接口

13、使类和成员的可访问性最小化

(1) 要区别设计良好的模块与设计不好的模块，最重要的因素在于，这个模块对于外部的其他模块而言，是否隐藏其内部数据和其他实现细节。设计良好的模块会隐藏所有的实现细节，把它的 API 与它的实现清晰的隔离开来。然后，模块之间只通过它们的 API 进行通信，一个模块不需要知道其他模块的内部工作情况。这被称为信息隐藏或者封装

(2) 第一条规则：尽可能使每个类和成员不被外界访问。尽可能使用最小的访问级别。

对于类和接口，只有两种可能的访问级别：包级私有的和公有的。使用 public 修饰符声明了类和接口，那么这个类和接口就是公有的;如果省略 public,那么这个类就是包级私有的。通过把类或者接口做成包级私有的，它实际上成了这个包的实现的一部分,而不是该包导出的 API 的一部分。以后对它的修改，只需要担心这个包内的其他类对它的使用，而不用担心客户端代码对它的使用。特别，如果一个类只是在某一个类的类的内部被使用到，那么我们可以考虑将这个类做成需要它的类的私有嵌套类（私有内部类），而不是包级私有类，这样就进一步降低了访问级别。

(3) 对于成员（五类：域，初始化块，构造器，方法，内部类、内部接口），有四种可能的访问级别：

私有的	private
包级私有的	缺省
受保护的	protected
公有的	public

(4) 应该把所有其他的成员都变成私有的,只有当同一个包中的另一个类真正需要访问一个成员时,才应该删除 private 修饰符，使该成员变成包级私有的。

(5) 对于静态域也建议尽量少使用公有的。

14、在公有类中使用访问方法而非公有域

```
Class Point {
    public double x;
    public double y;
}
```

(1)上面 Point 类的数据域是可以直接被访问的，这样的类没有提供封装。如果不改变 API，就无法改变它的数据表示法（比如，使用一个比 double 更高精度的类来表示 x 和 y），也无法强加任何约束条件（比如以后我们可能会希望 x 和 y 不会超过某个值）。应该用包含私有域和公有设值得类代替，如下

```
Class Point {
    private double x;
    private double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() { return x; }
    public double getY() { return y; }

    public void setX(double x) { this.x = x; }
    public void setY(double y) { this.y = y; }
}
```

(2) 让公有类暴露域不是好办法，但如果域是不可变的，这种做法的危害会较小,如下：

```
public final class Time {
    private static final int HOURS_PER_DAY = 24;
    private static final int MINUTES_PER_HOURS = 60;

    public final int hour;//hour 是不可变域
    public final int minute;//minute 是不可变域

    public Time(int hour, int minute) {//无法改变类的表示法，但是可以强加约束条件
        if(hour < 0 || hour > HOURS_PER_DAY)
            throw new IllegalArgumentException("Hour: " + hour);
        if(minute < 0 || minute >= MINUTES_PER_HOUR)
            throw new IlllllegalArgumentException("Minute: " + minute);
        this.hour = hour;
        this.minute = minte;
    }
}
```

(3) 总之，公有类永远都不应该暴露可变的域，让公有域暴露不可变域的危害较小（可以强加约束条件，不能改变域表示法），

15、使可变性最小化

(1)不可变类只是其实例不能被修改的类。每个实例中包含的所有信息都必须在创建该实例的时候就提供，并在对象的整个生命周期内固定不变。

为了使类成为不可变，要遵循下面五条规则：

a.不要提供任何会修改对象状态的方法

b.保证类不会被扩展

c.使所有的域都是 final 的

d.使所有的域都成为私有的

e.确保对于任何可变组件的互斥访问。

Demo:

```
public final class Complex{
    private final double re;
    private final double im;
    public Complex(double im, double re) {
        this.im = im;
        this.re = re;
    }
    public double realPart(){
        return re;
    }
    public double imaginaryPart(){
        return im;
    }
    public Complex subtract(Complex c){
        return new Complex(re-c.re,im-c.im);
    }

    public Complex multiply(Complex c){
        return new Complex(re*c.re-im*c.im,im*c.re+im*c.re);
    }

    public Complex divide(Complex c){
        double tmp = c.re*c.re+c.im*c.im;
        return new Complex((re*c.re+im*c.im)/tmp,(im*c.re-re*c.im)/tmp);
    }
}
```

(2)不可变对象本质上是线程安全的，它们不要求同步。所以不可变对象可以自由地共享。

(3)不仅可以共享不可变对象，甚至也可以共享它们的内部信息。

(4)不可变对象为其他对象提供了大量的构件，无论是可变的还是不可变的对象。

(5)不可变类真正唯一的缺点是，对于每个不同的值都需要一个单独的对象。创建这种对象的代价可能很高，特别是对于大型对象的情形。

(6)为了确保不可变性，类绝对不允许自身被子类化。除了“使类成为 final 的”这种方法之外，还有另一种更加灵活的办法可以做到这一点。让不可变的类变成 final 的另一种办法就是，让类的所有构造器都变成私有的或者包级私有的，并添加公有的静态工厂来代替公有的构造器。如下：

```
public class Complex{
    private final double re;
    private final double im;

    private Complex(double im, double re) {
        this.im = im;
        this.re = re;
    }

    public static Complex valueOf(double re,double im){
        return new Complex(re,im);
    }
}
```

16、复合优先于继承

继承（inheritance）是实现代码重用的有力手段，但并非总是最好的选择。继承打破了封装性，因为子类依赖于超类中特定功能的实现细节。超类的实现有可能随着发行版本的不同而有所变化，导致子类遭到破坏。

(1)使用继承，子类遭到破话的案例

```
public class InstrumentedHashSet<E> extends HashSet<E> {
    private int addCount = 0;
    public InstrumentedHashSet() {}
    public InstrumentedHashSet(int initCap, float loadFactor) {
        super(initCap, loadFactor);
    }
    @Override
    public boolean add(E e) {
        addCount ++;
        return super.add(e);
    }
    @Override
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() {
        return addCount;
    }
}
```

这段代码看上去没什么问题，假如执行下面的程序，我们期望 getAddCount 返回 3，但它实际上返回的是 6。

```
InstrumentedHashSet<String> s = new InstrumentedHashSet<String>();
s.addAll(Arrays.asList("Snap", "Crackle", "Pop"));
System.out.println(s.getAddCount());
```

因为：在 HashSet 内部，addAll 方法是基于 add 方法来实现的，因此 InstrumentedHashSet 中的 addAll 方法首先把 addCount 增加了 3，然后利用 super.addAll()调用 HashSet 的 addAll 实现，在该实现中又调用了被 InstrumentedHashSet 覆盖了的 add 方法，每个元素调用一次，这三次又分别给 addCount 增加了 1，所以总共增加了 6。

因此，使用继承扩展一个类很危险，父类的具体实现很容易影响子类的正确性。而复合优先于继承告诉我们，不用扩展现有的类，而是在新类中增加一个私有域，让它引用现有类的一个实例。这种设计称为复合（Composition）。

17、要么为继承而设计，并提供文档说明，要么就禁止继承

(1)第 16 条告诉了我们对一个不是继承而设计并且没有文档的类是很危险的。那么一个为了设计而使用继承并且有很好的文档则应该做以下事情：

a.该类的文档必须精确地说明每个改写的方法作用以及影响。好的 API 文档：应该是描述该方法做了什么工作，而不是如何做的。

b. 一个为了继承而设计的类（超类）应该是：

b1.为了允许被继承，无论是直接还是间接，构造方法都不能够调用非 final 方法。否则程序有可能失败。由于超类的构造方法比子类的构造方法先执行，所以子类中改写版本的方法将会在子类的构造方法运行之前先被调用。如果该改写版本的方法依赖于子类构造方法所执行的初始化工作，那么该方法将不会如预期执行。

b2.设计的超类实现 Serializable 或者 Cloneable 接口都是很麻烦的事情。因为 clone 和 readObject 都类似一个构造方法。所以超类要实现这两个接口就必须遵循这个规则：clone 和 readObject 都不能够调用可改写的方法，无论是直接还是间接方式。

b3.如果超类实现了 Serializable 接口，并且该类有一个 readResolve 或者 writeReplace 方法，那么就必须是 readResolve 或者 writeReplace 方法成为 protectd，而不是 private。如果是私有的话，那么子类会忽略掉这两个方法。

18、接口优于抽象类

如果一个方法是抽象方法的话，那么该类一定是抽象类，并且该类的类名前必须加 abstract。抽象类中的抽象方法也必须加 abstract。

(1) Java 程序设计语言提供了接口和抽象类两种机制来定义允多个实现的类型。这两种机制的主要区别有

a.抽象类允许包含某些方法的实现，但是接口则不允许。

b.为了实现由抽象类定义的类型，类必须成为抽象类的一个子类。

c.因为 Java 只允许单继承，所以抽象类作为类型定义受到了极大的限制，但是一个类可以实现多个接口。

(2) 接口使得安全地增强类的功能成为可能。如果使用抽象类来定义类型，那么程序员除了使用继承的手段来增加功能，没有其他的选择。这样得到的类与包装类相比，功能更差，也更加脆弱

(3) 骨架实现类。

虽然接口不允许包含方法的实现，,，但是可以通过对你导出的每个重要接口都提供一个抽象的骨架实现类，把接口和抽象的优点结合起来。接口的作用仍然是定义类型，但是骨架实现类接管了所有与接口实现相关的工作

（4）接口和抽象各有各自的好处与缺点：当接口增加一个方法的时候，那么它的实现类也必须增加对该方法的实现，这会打破原有的类设计，但是使用抽象类就没有此缺点。

19、接口只用于定义类型

(1) 常量接口的概念：只有静态 final 成员变量的接口（接口中成员变量默认为 static final 的）。常量接口是对接口的不良使用。这就相当于实现该接口的类将他要使用的常量暴露于外部

(2) 如果一个类实现了这样的接口，那么就可以不通过类名+常量的模式来访问常量了。最好不要使用接口来导出常量，这种方式不好，比如说当接口增加某些常量或者删除某些常量的话，对于实现了该接口的类就需要做一些更改

(3) 如果要导出常量的话，可以使用一个不可以实例化的类（将类的构造方法声明为私有的）；或者使用枚举类型定义常量。如：Integer。

总之，接口只是用来定义类型的，而不应该用来导出常量。

20、类层次优于标签类

(1) 什么是标签类？如下

```
class Figure{
    enum Shape{RECTANGLE,CIRCLE};
    //Tag field - the shape of this figure
    final Shape shape;
    //There field are used only if shape is RECTANGLE
    double length;
    double width;
    //This field is used only if shape is CIRCLE
    double radius;
    //Constructor for circle
    Figure(double radius){
        shape = Shape.CIRCLE;
        this.radius = radius;
    }
    //Constructor for rectangle
    Figure(double length,double width){
        shape = Shape.RECTANGLE;
        this.length = length;
        this.width = width;
    }
    double area(){
        switch(shape){
            case RECTANGLE:
                return length * width;
            case CIRCLE:
                return Math.PI * (radius * radius);
            default:
                throw new AssertionError();
        }
    }
}
```

这种类中有许多样板代码充斥在一个单类中，破坏可读性，内存占用增加，因为实例承担着不相关的域，实例化域不当，会引起程序 bug。过于冗余，容易出错，并且效率低下。

（2）什么是类层次？

```
abstract class Figure{
    abstract double area();
}
class Circle extends Figure{
    final double radius;
    Circle(double radius){this.radius = radius}
    double area(){return Math.PI * (radius * radius);}
}
class Rectangle extends Figure{
    final double length;
    final double width;
    Rectangle(double length,double width){
        this.length = length;
        this.width = width;
    }
    double area(){
        return (length * width);
    }
}
```

这种类的优点：

(1) 这个类层次纠正了前面提到的标签类的所有缺点。

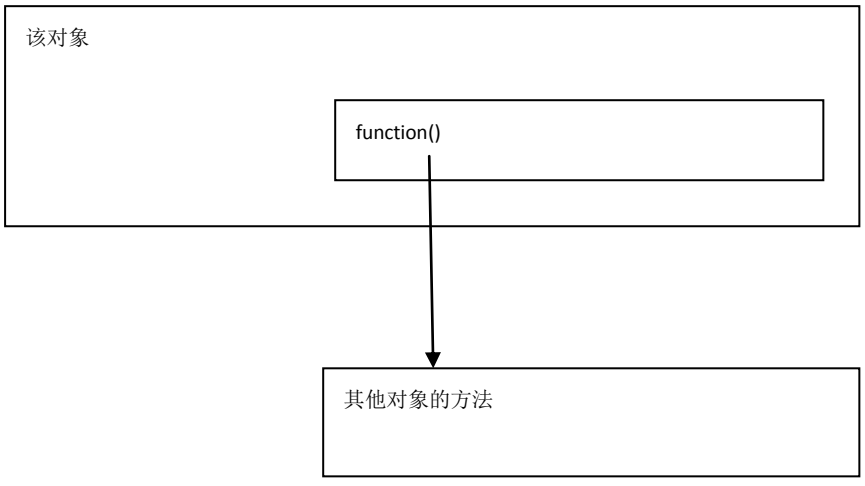
(1) 他们可以用来反映类之间本质上的层次关系，有助于增强灵活性，并进行更好的编译时类型检查。

21、用函数对象表示策略

(1)引入情景：不像 C 语言等语言，Java 没有提供函数指针。所以要实现相同的功能就要用到“函数对象”。

(2)解释：调用对象的方法通常就是执行该对象（that object）上的某项操作。然而也可以定义这样的对象，他的方法执行其他对象（other object）上的操作。

(3)什么是函数对象，图解如下：



```
public class StringLengthComparator {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

指向 StringLengthComparator 对象的引用可以被当做是一个指向该比较器的“函数指针（function pointer)”，可以在任意一对象字符串上被调用。换句话说，StringLengthComparator 实例是用于字符串比较操作的具体策略(concrete strategy)。

(4)当一个具体策略只使用一次时，通常使用匿名类来声明和实例化这个具体策略类，如下：

a.定义一个策略接口，如下

```
public interface Comparable<T> {
    public int compareTo(T t1, T t2);
}
```

b. 具体的策略类往往使用匿名内部类，如

```
Arrays.sort(stringArray, new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        return 0;
    }
});
```

(5) 当一个具体策略类是设计用来重复使用的时候，他的类通常就要被实现为私有的静态成员类，并通过共有的静态 final 域导出（策略接口类型），如

```
public class Host {
    private static class StrLenCmp implements Comparator<String>, Serializable {
        @Override
        public int compare(String o1, String o2) {
            return o1.length() - o2.length();
        }
    }

    public static final Comparator<String> STRING_LENGTH_COMPARATOR = new StrLenCmp();
}
```

22、优先考虑静态成员类

(1)嵌套类（nested class）是指被定义在另一个类的内部的类。嵌套类存在的目的应该只是为他的外围类（enclosing class）提供服务。

(2)嵌套类有四种：静态成员类（static member class)、非静态成员类（nonstatic member class)、匿名类（anonymous class）和局部类（local class)。除了第一种之外，其他三种都称为内部类（inner class）

(3)从语法上讲，静态成员类和非静态成员类之间的唯一区别是，静态成员类的声明中包含修饰符 static。尽管他们的语法非常相似，但是两种嵌套类有很大的不同

```
package cn.partner4java.nestedclass;


public class HelloWorld2 {


    public void test1(){}
    public static void test2(){}

    //如果要调用下面这个类的方法，HelloWorld2 必须实例化
    class NestedClass1{
        public NestedClass1() {
            test1();   //在非静态成员类的实例方法内部，可以调用外围实例上的方法
        }
    }

    //要调用下面这个类的方法，HelloWorld2 不用实例化，这是较内部类的区别或者说优点
    static class NestedClass2{
        public NestedClass2() {
            test2();
        }
    }
}
```