

## Final 关键字

### Final01.java

**final** 中文意思:最后的, 最终的.

**final** 可以修饰类、属性、方法和局部变量.

在某些情况下,程序员可能有以下需求, 就会使用到**final**:

- 1) 当不希望类被继承时,可以用**final**修饰. 【案例演示】
- 2) 当不希望父类的某个方法被子类覆盖/重写(override)时,可以用**final**关键字修饰. 【案例演示: 访问修饰符 **final** 返回类型 方法名】
- 3) 当不希望类的某个属性的值被修改,可以用**final**修饰. 【案例演示: `public final double TAX_RATE=0.08`】
- 4) 当不希望某个局部变量被修改, 可以使用**final**修饰 【案例演示: `final double TAX_RATE=0.08`】

## Final 细节:

- 1) **final**修饰的属性又叫常量,一般用 `XX_XX_XX` 来命名
- 2) **final**修饰的属性在定义时,必须赋初值,并且以后不能再修改,赋值可以在如下位置之一【选择一个位置赋初值即可】:
  - ① 定义时: 如 `public final double TAX_RATE=0.08;`
  - ② 在构造器中
  - ③ 在代码块中.
- 3) 如果**final**修饰的属性是静态的,则初始化的位置只能是
  - ① 定义时
  - ② 在静态代码块不能在构造器中赋值。
- 4) **final**类不能继承,但是可以实例化对象. [A2类]
- 5) 如果类不是**final**类,但是含有**final**方法,则该方法虽然不能重写,但是可以被继承. [A3类]

- 5) 一般来说,如果一个类已经是**final**类了,就没有必要再将方法修饰成**final**方法.
- 6) **final**不能修饰构造方法(即构造器)
- 7) **final** 和 **static** 往往搭配使用,效率更高,不会导致类加载.底层编译器做了优化处理.

```
class Demo{
    public static final int i=16; //
    static{
        System.out.println("韩顺平教育~");
    }
}
```

- 8) 包装类(Integer,Double,Float, Boolean等都是**final**),String也是**final**类。

## 抽象类:

父类的某些方法,需要声明,又不确定该如何实现时,可以将其声明为抽象方法,这个类就是抽象类。

## 抽象类介绍:

- 1) 用abstract 关键字来修饰一个类时,这个类就叫抽象类  
访问修饰符 abstract 类名{  
}
- 2) 用abstract 关键字来修饰一个方法时,这个方法就是抽象方法  
访问修饰符 abstract 返回类型 方法名(参数列表);//没有方法体
- 3) 抽象类的价值更多作用是在于设计,是设计者设计好后,让子类继承并实现抽象类()
- 4) 抽象类,是考官比较爱问的知识点,在框架和设计模式使用较多

注意:

- 1) 抽象类不能被实例化 [举例]
- 2) 抽象类不一定要包含abstract方法。也就是说,抽象类可以没有abstract方法 [举例]
- 3) 一旦类包含了abstract方法,则这个类必须声明为abstract [说明]
- 4) abstract 只能修饰类和方法,不能修饰属性和其它的。[说明]

5) 抽象类可以有任意成员【抽象类本质还是类】,比如:非抽象方法、构造器、静态属性等等 [举例]

6) 抽象方法不能有主体,即不能实现.如图所示

```
abstract void aaa() { };
```

7) 如果一个类继承了抽象类,则它必须实现抽象类的所有抽象方法,除非它自己也声明为abstract类。[举例 A类,B类,C类]

8) 抽象方法不能使用private、final 和 static来修饰,因为这些关键字都是和重写相违背的。

接口:

接口就是给出一些没有实现的方法,封装到一起,到某个类要使用的时候,在根据具体情况把这些方法写出来。语法:

```
interface 接口名{  
    //属性  
    //抽象方法  
}  
class 类名 implements 接口{  
    自己属性;  
    自己方法;  
    必须实现的接口的抽象方法  
}
```

**小结:**接口是更加抽象的抽象的类,抽象类里的方法可以有方法体,接口里的所有方法都没有方法体 [jdk7.0]。接口体现了程序设计的多态和高内聚低耦合的设计思想。

特别说明: Jdk8.0后接口类可以有静态方法,默认方法,也就是说接口中可以有方法的具体实现

注意事项:



1. 接口不能被实例化
2. 接口中所有的方法是 `public` 方法, 接口中抽象方法, 可以不用 `abstract` 修饰
3. 一个普通类实现接口, 就必须将该接口的所有方法都实现, 可以使用 `alt+enter` 来解决
4. 抽象类去实现接口时, 可以不实现接口的抽象方法
5. 一个类同时可以实现多个接口
6. 接口中的属性, 是 `public static final`
7. 接口不能继承其它的类, 但是可以继承多个别的接口
8. 接口的修饰符 只能是 `public` 和默认, 这点和类的修饰符是一样的

接口与继承:

当子类继承了父类, 就自动的拥有父类的功能。

如果子类需要扩展功能, 可以通过实现接口的方式扩展。

可以理解 实现接口 是 对 java 单继承机制的一种补充。

➤ 接口和继承解决的问题不同

继承的价值主要在于: 解决代码的**复用性和可维护性**。  
接口的价值主要在于: 设计, 设计好各种规范(方法), 让其它类去实现这些方法。即更加的灵活..

➤ 接口比继承更加灵活

接口比继承更加灵活, 继承是满足 `is - a`的关系, 而接口只需满足 `like - a`的关系。

➤ 接口在一定程度上实现代码解耦 [即: 接口规范性+动态绑定机制]

接口的多态性:

- (1) 多态参数
- (2) 多态数组
- (3) 接口存在多态传递现象

内部类:

如果定义类在局部位置(方法中/代码块): (1) 局部内部类 (2) 匿名内部类

定义在成员位置 (1) 成员内部类 (2) 静态内部类

基本语法:

```

class Outer{    //外部类
    class Inner{    //内部类
    }
}
class Other{    //外部其他类
}
//InnerClass01.java

```

内部类分类:

- 定义在外部类局部位置上 (比如方法内) :
  - 1) 局部内部类 (有类名)
  - 2) 匿名内部类 (没有类名, 重点!!!!!!!)
- 定义在外部类的成员位置上:
  - 1) 成员内部类 (没用static修饰)
  - 2) 静态内部类 (使用static修饰)

局部内部类:

说明: 局部内部类是定义在外部类的局部位置, 比如方法中, 并且有类名。

1. 可以直接访问外部类的所有成员, 包含私有的
2. 不能添加访问修饰符, 因为它的地位就是一个局部变量。局部变量是不能使用修饰符的。但是可以使用final 修饰, 因为局部变量也可以使用final
3. 作用域: 仅仅在定义它的方法或代码块中。
4. 局部内部类---访问---->外部类的成员 [访问方式: 直接访问]
5. 外部类---访问---->局部内部类的成员  
访问方式: 创建对象, 再访问(注意: 必须在作用域内)

记住: (1) 局部内部类定义在方法中/代码块  
(2) 作用域在方法体或者代码块中  
(3) 本质仍然是一个类

6. 外部其他类---不能访问----->局部内部类 (因为 局部内部类地位是一个局部变量)
7. 如果外部类和局部内部类的成员重名时, 默认遵循就近原则, 如果想访问外部类的成员, 则可以使用 (外部类名.this.成员) 去访问 【演示】

```

System.out.println("外部类的n2=" + 外部类名.this.n2);

```

匿名内部类:



//(1) 本质是类(2) 内部类(3) 该类没有名字  
(4)同时还是一个对象

说明：匿名内部类是定义在外部类的局部位置，比如方法中，并且没有类名

### 1. 匿名内部类的基本语法

```
new 类或接口(参数列表){  
    类体  
};
```

### 【案例演示 AnonymousInnerClass.java】

#### AnonymousInnerClassDetail.java

2. 匿名内部类的语法比较奇特，请大家注意，因为匿名内部类既是一个类的定义，同时它本身也是一个对象，因此从语法上看，它既有定义类的特征，也有创建对象的特征，对前面代码分析可以看出这个特点，因此可以调用匿名内部类方法。

3. 可以直接访问外部类的所有成员，包含私有的 [案例演示]

4. 不能添加访问修饰符，因为它的地位就是一个局部变量。 [过]

5. 作用域：仅仅在定义它的方法或代码块中。 [过]

6. 匿名内部类---访问---->外部类成员 [访问方式：直接访问]

7. 外部其他类---不能访问----->匿名内部类（因为匿名内部类地位是一个局部变量）  
8. 如果外部类和匿名内部类的成员重名时，匿名内部类访问的话，默认遵循就近原则，如果想访问外部类的成员，则可以使用（外部类名.this.成员）去访问

成员内部类：

#### MemberInnerClass01.java

说明：成员内部类是定义在外部类的成员位置，并且没有static修饰。

1. 可以直接访问外部类的所有成员，包含私有的

```
class Outer01{ //外部类  
    private int n1 = 10;  
    public String name = "张三";  
    class Inner01{  
        public void say(){  
            System.out.println("Outer01 的 n1 = " + n1 + " outer01 的 name = " + name );  
        }  
    }  
}
```

2. 可以添加任意访问修饰符(public、protected、默认、private),因为它的地位就是一个成员。

### 3. 作用域 **MemberInnerClass01.java**

和外部类的其他成员一样，为整个类体  
比如前面案例，在外部类的成员方法中创建成员内部类对象，再调用方法。

4. 成员内部类---访问---->外部类成员(比如：属性) [访问方式：直接访问] (说明)

5. 外部类---访问----->成员内部类 (说明)  
访问方式：创建对象，再访问

6. 外部其他类---访问---->成员内部类

7. 如果外部类和内部类的成员重名时，内部类访问的话，默认遵循就近原则，如果想访问外部类的成员，则可以使用 (外部类名.this.成员) 去访问

静态内部类：

说明：静态内部类是定义在外部类的成员位置，并且有static修饰

1. 可以直接访问外部类的所有静态成员，包含私有的，但不能直接访问非静态成员  
2. 可以添加任意访问修饰符(public、protected、默认、private),因为它的地位就是一个成员。

3. 作用域：同其他的成员，为整个类体

4. 静态内部类---访问---->外部类(比如：静态属性) [访问方式：直接访问所有静态成员]

5. 外部类---访问----->静态内部类 访问方式：创建对象，再访问

6. 外部其他类---访问----->静态内部类

7. 如果外部类和静态内部类的成员重名时，静态内部类访问的时，默认遵循就近原则，如果想访问外部类的成员，则可以使用 (外部类名.成员) 去访问