Java 内部类综述

摘要：

多重继承指的是一个类可以同时从多于一个的父类那里继承行为和特征，然而我们知道 Java 为了保证数据安全，它只允许单继承。但有时候，我们确实是需要实现多重继承，而且现实生活中也真正地存在这样的情况，比如遗传：我们即继承了父亲的行为和特征也继承了母亲的行为和特征。可幸的是，**Java 提供了两种方式让我们曲折地来实现多重继承：接口和内部类。**事实上，**实现多重继承是内部类的一个极其重要的应用。**除此之外，**内部类还可以很好的实现隐藏(例如，私有成员内部类)。**内部类共有四种类型，即成员内部类、静态内部类、局部内部类和匿名内部类。特别地，

- **成员内部类：**成员内部类是外围类的一个成员，是 依附于外围类的，所以，只有先创建了外围类对象才能够创建内部类对象。也正是由于这个原因，成员内部类也不能含有 static 的变量和方法；
- **静态内部类：**静态内部类，就是修饰为 static 的内部类，该内部类对象 不依赖于外部类对象，就是说我们可以直接创建内部类对象，但其只可以直接访问外部类的所有静态成员和静态方法；
- **局部内部类：**局部内部类和成员内部类一样被编译，只是它的 作用域发生了改变，它只能在该方法和属性中被使用，出了该方法和属性就会失效；
- **匿名内部类：定义匿名内部类的前提是，内部类必须要继承一个类或者实现接口，格式为 new 父类或者接口(){定义子类的内容(如函数等)}。也就是说，匿名内部类最终提供给我们的是一个匿名子类的对象。**

一. 内部类概述

1、 内部类基础

内部类指的是在一个类的内部所定义的类，类名不需要和源文件名相同。内部类是一个编译时的概念，一旦编译成功，内部类和外部类就会成为两个完全不同的类。例如，对于一个名为 Outer 的外部类和在其内部定义的名为 Inner 的内部类，在编译完成后，会出现 Outer.class 和 Outer$inner.class 两个类。因此，内部类的成员变量/方法名可以和外部类的相同。内部类可以是静态 static 的，也可用 public，default，protected 和 private 修饰。 特别地，关于 **Java 源文**

件名与类名的关系(java 源文件名的命名与内部类无关,以下 3 条规则中所涉及的类和接口均指的是外部类/接口)，需要符合下面三条规则：

- **如果 java 源文件包含 public 类(public 接口)，则源文件名必须与 public 类名(public 接口名)相同。**

  一个 java 源文件中，如果有 public 类或 public 接口，那么就只能有一个 public 类或一个 public 接口，不能有多个 public 的类或接口。当然，一个 java 源文件中可以有多个包可见的类或接口，即默认访问权限修饰符(类名前没有访问权限修饰符)。public 类(接口) 与 包可见的类(接口)在文件中的顺序可以随意，即 public 类(接口)可以不在第一个的位置。

- **如果 java 源文件不包含 public 类(public 接口)，则 java 源文件名没有限制。**

  只要符合文件名的命名规范就可以，可以不与文件中任一个类或接口同名，当然，也可以与其中之一同名。

- **类和接口的命名不能冲突。**

  同一个包中的任何一个类或接口的命名都不能相同。不同包中的类或接口的命名可以相同，因为通过包可以把它们区分开来。

2、 内部类的作用

使用内部类可以给我们带来以下优点：

- **内部类可以很好的实现隐藏（一般的非内部类，是不允许有 private 与 protected 权限的，但内部类可以）；**
- 内部类拥有外围类的所有元素的访问权限；
- **可以实现多重继承；**
- 可以避免修改接口而实现同一个类中两种同名方法的调用。

1）内部类可以很好的实现隐藏

平时我们对类的访问权限，都是通过类前面的访问修饰符来限制的，一般的非内部类，是不允许有 private 与 protected 权限的，但内部类可以，所以我们能通过内部类来隐藏我们的信息。可以看下面的例子：

```
//测试接口
public interface InterfaceTest {
    public void test();
}


//外部类
public class Example {


    //内部类
    private class InnerClass implements InterfaceTest{
        @Override
        public void test() {
            System.out.println("I am Rico.");
        }
    }


    //外部类方法
    public InterfaceTest getInnerInstance(){
```

```
        return new InnerClass();

    }

}



//客户端

public class Client {

    public static void main(String[] args) {

        Example ex = new Example();

        InterfaceTest test = ex.getInnerInstance();

        test.test();

    }

}/* Output:

        I am Rico.

 *///:~
```

对客户端而言，我们可以通过 Example 的 getInnerInstance()方法得到一个 InterfaceTest 实例，但我们并不知道这个实例是如何实现的，也感受不到对应的具体实现类的存在。由于 InnerClass 是 private 的，所以，我们如果不看源代码的话，连实现这个接口的具体类的名字都看不到，所以说内部类可以很好的实现隐藏。

2）内部类拥有外围类的所有元素的访问权限

```
//外部类
public class Example {
    private String name = "example";

    //内部类
    private class Inner{
        public Inner(){
            System.out.println(name);   // 访问外部类的私有属性
        }
    }

    //外部类方法
    public Inner getInnerInstance() {
        return new Inner();
    }
}

//客户端
public class Client {
```

```
    public static void main(String[] args) {

        Example ex = new Example();

        ex.getInnerInstance();

    }

}/* Output:

        example

 *///:~
```

name 这个成员变量是在 Example 里面定义的私有变量，这个变量在内部类中可以被无条件地访问。

3）可以实现多重继承

对多重继承而言，可以这样说，接口只是解决了部分问题，而内部类使得多重继承的解决方案变得更加完整。内部类使得 Java 的继承机制更加完善，是内部类存在的最大理由。Java 中的类只能继承一个类，它的多重继承在我们没有学习内部类之前是用接口来实现的。但使用接口有时候有很多不方便的地方，比如，我们实现一个接口就必须实现它里面的所有方法；而内部类可以使我们的类继承多个具体类或抽象类，规避接口的限制性。看下面的例子：

```
//父类 Example1

public class Example1 {

    public String name() {

        return "rico";

    }

}
```

```java
//父类 Example2
public class Example2 {
    public int age() {
        return 25;
    }
}

//实现多重继承的效果
public class MainExample {

    //内部类 Test1 继承类 Example1
    private class Test1 extends Example1 {
        public String name() {
            return super.name();
        }
    }

    //内部类 Test2 继承类 Example2
    private class Test2 extends Example2 {
```

```
        public int age() {
            return super.age();
        }
    }

    public String name() {
        return new Test1().name();
    }

    public int age() {
        return new Test2().age();
    }

    public static void main(String args[]) {
        MainExample mexam = new MainExample();
        System.out.println("姓名:" + mexam.name());
        System.out.println("年龄:" + mexam.age());
    }
}/* Output:
        姓名:rico
```

```
        年龄:25

*///:~
```

注意到类 MainExample，在这个类中，包含两个内部类 Test1 和 Test2。其中，类 Test1 继承了类 Example1，类 Test2 继承了类 Example2。这样，类 MainExample 就拥有了 类 Example1 和 类 Example2 的方法，也就间接地实现了多继承。

4） 避免修改接口而实现同一个类中两种同名方法的调用

考虑这样一种情形，一个类要继承一个类，还要实现一个接口，可是它所继承的类和接口里面有两个相同的方法（方法签名一致），那么我们该怎么区分它们呢？这就需要使用内部类了。例如，

```
//Test 所实现的接口

public interface InterfaceTest {

    public void test();

}



//Test 所实现的类

public class MyTest {

    public void test(){

        System.out.println("MyTest");

    }

}
```

```
//不使用内部类的情形

public class Test extends MyTest implements InterfaceTest{

    public void test(){

        System.out.println("Test");

    }

}
```

此时，Test 中的 test() 方法是属于覆盖 MyTest 的 test() 方法呢，还是实现 InterfaceTest 中的 test() 方法呢？我们怎么能调到 MyTest 这里的方法？显然这是不好区分的。而我们如果用内部类就很好解决这一问题了。看下面代码：

```
//Test 所实现的接口

public interface InterfaceTest {

    public void test();

}



//Test 所实现的类

public class MyTest {

    public void test(){

        System.out.println("MyTest");

    }

}
```

```
//使用内部类的情形
public class AnotherTest extends MyTest {


    private class InnerTest implements InterfaceTest {
        @Override
        public void test() {
            System.out.println("InterfaceTest");
        }
    }


    public InterfaceTest getCallbackReference() {
        return new InnerTest();
    }


    public static void main(String[] args) {
        AnotherTest aTest = new AnotherTest();
        aTest.test(); // 调用类 MyTest 的 test() 方法
        aTest.getCallbackReference().test(); // 调用 InterfaceTest 接口中的
 test() 方法
    }
```

```
}
```

通过使用内部类来实现接口，就不会与外围类所继承的同名方法冲突了。

3、 内部类的种类

在 Java 中，内部类的使用共有两种情况：

(1) 在类中定义一个类(成员内部类，静态内部类)；
(2) 在方法中定义一个类(局部内部类，匿名内部类)。

二. 成员内部类

1、定义与原理

成员内部类是最普通的内部类，它是外围类的一个成员，在实际使用中，一般将其可见性设为 private。成员内部类是依附于外围类的，所以，只有先创建了外围类对象才能够创建内部类对象。也正是由于这个原因，成员内部类也不能含有 static 的变量和方法，看下面例子：

```
public class Outter {

    private class Inner {



        private final static int x=1;   // OK



        /* compile errors for below declaration

        * "The field x cannot be declared static in a non-static inner ty
pe,
```

```
        * unless initialized with a constant expression" */

        final static Inner a = new Inner();     // Error



        static Inner a1=new Inner();     // Error



        static int y;     // Error

    }

}
```

如果上面的代码编译无误，那么我们就可以直接通过 Outter.Inner.a 拿到内部类 Inner 的实例。 由于内部类的实例一定要绑定到一个外部类的实例的，所以矛盾。因此，成员内部类不能含有 static 变量/方法。此外，成员内部类与 static 的关系还包括：

- 包含 static final 域，但该域的初始化必须是一个常量表达式；
- 内部类可以继承含有 static 成员的类。

2、交互

成员内部类与外部类的交互关系为：

- 成员内部类可以直接访问外部类的所有成员和方法，即使是 private 的；
- 外部类需要通过内部类的对象访问内部类的所有成员变量/方法。

```
//外部类

class Out {

    private int age = 12;
```

```
    private String name = "rico";



    //内部类

    class In {

        private String name = "livia";

        public void print() {

            String name = "tom";

            System.out.println(age);

            System.out.println(Out.this.name);

            System.out.println(this.name);

            System.out.println(name);

        }

    }



    // 推荐使用 getxxx()来获取成员内部类的对象

    public In getInnerClass(){

        return new In();

    }

}
```

```
public class Demo {

    public static void main(String[] args) {



        Out.In in = new Out().new In();   // 片段 1

        in.print();



        //或者采用注释内两种方式访问

        /*

         * 片段 2

        Out out = new Out();



        out.getInnerClass().print();  // 推荐使用外部类 getxxx()获取成员内部类对象



        Out.In in = out.new In();

        in.print();



        */

    }

}/* Output:

        12
```

```
        rico

        livia

        tom

*///:~
```

对于代码片段 1 和 2，可以用来生成内部类的对象，这种方法存在两个小知识点需要注意：

1) **开头的 Out 是为了标明需要生成的内部类对象在哪个外部类当中；**
2) **必须先有外部类的对象才能生成内部类的对象。**

因此，成员内部类，外部类和客户端之间的交互关系为：

- 在成员内部类使用外部类对象时，使用 outer.this 来表示外部类对象；
- 在外部类中使用内部类对象时，需要先进行创建内部类对象；
- 在客户端创建内部类对象时，需要先创建外部类对象。

特别地，对于成员内部类对象的获取，外部类一般应提供相应的 getxxx() 方法。

3、私有成员内部类

如果一个成员内部类只希望被外部类操作，那么可以使用 private 将其声明私有内部类。例如，

```
class Out {

    private int age = 12;



    private class In {
```

```
        public void print() {
            System.out.println(age);
        }
    }
    public void outPrint() {
        new In().print();
    }
}

public class Demo {
    public static void main(String[] args) {

        /*
        * 此方法无效
        Out.In in = new Out().new In();
        in.print();
        */

        Out out = new Out();
        out.outPrint();
```

```
    }

}/* Output:

        12

 *///:~
```

在上面的代码中，**我们必须在 Out 类里面生成 In 类的对象进行操作，而无法再使用 Out.In in = new Out().new In() 生成内部类的对象。也就是说，此时的内部类只对外部类是可见的，其他类根本不知道该内部类的存在。**

三. 静态内部类

1、定义与原理

静态内部类，就是修饰为 static 的内部类，该内部类对象不依赖于外部类对象，就是说我们可以直接创建内部类对象。看下面例子：

```
3  class Outt {
4      private static int age = 12;
5      private String name = "rico";
6
7      static class In {
8          public void print() {
9              System.out.println(age);
10             System.out.println(name); // Error
11         }
12     }
13 }
14
15 public class Demoo {
16     public static void main(String[] args) {
17
18         // 通过类名访问static，生不生成外部类对象都没关系
19         Outt.In in = new Outt.In();
20
21         in.print();
22     }
23 }
```

2、交互

静态内部类与外部类的交互关系为：

- 静态内部类可以直接访问外部类的所有静态成员和静态方法，即使是 private 的；
- 外部类可以通过内部类对象访问内部类的实例成员变量/方法；对于内部类的静态域/方法，外部类可以通过内部类类名访问。

3、成员内部类和静态内部类的区别

成员内部类和静态内部类之间的不同点包括：

- **静态内部类对象的创建不依赖外部类的实例，但成员内部类对象的创建需要依赖外部类的实例；**
- **成员内部类能够访问外部类的静态和非静态成员，静态内部类不能访问外部类的非静态成员；**

四. 局部内部类

1、定义与原理

有这样一种内部类，它是嵌套在方法和作用域内的，对于这个类的使用主要是应用与解决比较复杂的问题，想创建一个类来辅助我们的解决方案，但又不希望这个类是公共可用的，所以就产生了局部内部类。局部内部类和成员内部类一样被编译，只是它的作用域发生了改变，它只能在该方法和属性中被使用，出了该方法和属性就会失效。

```
// 例 1：定义于方法内部
```

```
public class Parcel4 {

    public Destination destination(String s) {

        class PDestination implements Destination {

            private String label;



            private PDestination(String whereTo) {

                label = whereTo;

            }



            public String readLabel() {

                return label;

            }

        }

        return new PDestination(s);

    }



    public static void main(String[] args) {

        Parcel4 p = new Parcel4();

        Destination d = p.destination("Tasmania");

    }
```

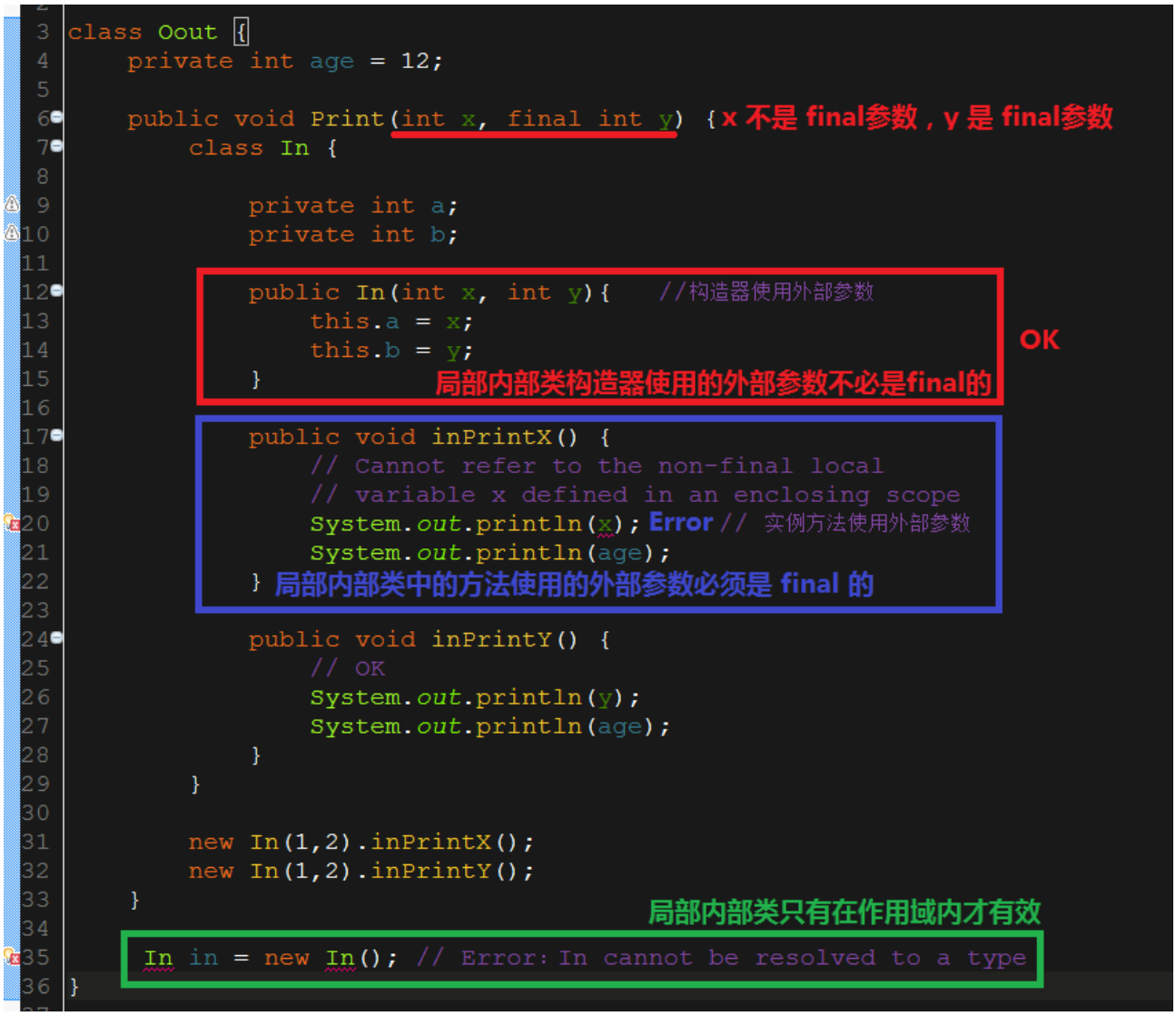
```
}

// 例 2：定义于作用域内部

public class Parcel5 {

    private void internalTracking(boolean b) {

        if (b) {

            class TrackingSlip {

                private String id;

                TrackingSlip(String s) {

                    id = s;

                }

                String getSlip() {

                    return id;

                }

            }

            TrackingSlip ts = new TrackingSlip("slip");

            String s = ts.getSlip();

        }

    }



    public void track() {
```

```
        internalTracking(true);
    }


    public static void main(String[] args) {
        Parcel5 p = new Parcel5();
        p.track();
    }
}
```

2、final 参数

对于 final 参数，若是将引用类型参数声明为 final，我们无法在方法中更改参数引用所指向的对象；若是将基本类型参数声明为 final，我们可以读参数，但却无法修改参数（这一特性主要用来向局部内部类和匿名内部类传递数据）。

如果定义一个局部内部类，并且希望它的方法可以直接使用外部定义的数据，那么我们必须将这些数据设为是 final 的；特别地，如果只是局部内部类的构造器需要使用外部参数，那么这些外部参数就没必要设置为 final，例如：




```
class Oout {
    private int age = 12;

    public void Print(int x, final int y) {
        class In {

            private int a;
            private int b;

            public In(int x, int y){    //构造器使用外部参数
                this.a = x;
                this.b = y;
            }

            public void inPrintX() {
                // Cannot refer to the non-final local
                // variable x defined in an enclosing scope
                System.out.println(x); Error // 实例方法使用外部参数
                System.out.println(age);
            }

            public void inPrintY() {
                // OK
                System.out.println(y);
                System.out.println(age);
            }
        }

        new In(1,2).inPrintX();
        new In(1,2).inPrintY();
    }

    In in = new In(); // Error: In cannot be resolved to a type
}
```

五. 匿名内部类

有时候我为了免去给内部类命名，便倾向于使用匿名内部类，因为它没有名字。匿名内部类的使用需要注意以下几个地方：

- 匿名内部类是没有访问修饰符的；
- 匿名内部类是没有构造方法的 (因为匿名内部类连名字都没有)；

- 定义匿名内部类的前提是，内部类必须是继承一个类或者实现接口，格式为 **new** 父类或者接口**(){**子类的内容**(**如函数等**)}**。也就是说，匿名内部类最终提供给我们的是一个匿名子类的对象，例如：

```
// 例 1

abstract class AbsDemo

{

    abstract void show();

}



public class Outer

{

    int x=3;

    public void function()//可调用函数

    {

        new AbsDwmo()//匿名内部类

        {

            void show()

            {

                System.out.println("x==="+x);

            }

            void abc()
```

```
            {
                System.out.println("haha");
            }
        }.abc();  //匿名内部类调用函数,匿名内部类方法只能调用一次
    }
}
// 例 2
interface Inner {    //注释后，编译时提示类 Inner 找不到
    String getName();
}

public class Outer {

    public Inner getInner(final String name, String city) {
        return new Inner() {
            private String nameStr = name;

            public String getName() {
                return nameStr;
            }
```

```
        };

    }



    public static void main(String[] args) {

        Outer outer = new Outer();

        Inner inner = outer.getInner("Inner", "gz");

        System.out.println(inner.getName());



        System.out.println(inner instanceof Inner);  //匿名内部类实质上是一个匿名子类的对象

    } /* Output:

        Inner

        true

      *///:~

}
```

- **若匿名内部类 (匿名内部类没有构造方法) 需要直接使用其所在的外部类方法的参数时，该形参必须为 final 的；如果匿名内部类没有直接使用其所在的外部类方法的参数时，那么该参数就不必为 final** 的，例如：

```
// 情形 1：匿名内部类直接使用其所在的外部类方法的参数 name
```

```
public class Outer {

    public static void main(String[] args) {

        Outer outer = new Outer();

        Inner inner = outer.getInner("Inner", "gz");

        System.out.println(inner.getName());

    }



    public Inner getInner(final String name, String city) {  // 形参 name 被设为 final

        return new Inner() {

            private String nameStr = name;       // OK

            private String cityStr = city;       // Error: 形参 city 未被设为 final



            public String getName() {

                return nameStr;

            }

        };

    }

}
```

```
// 情形 2：匿名内部类没有直接使用其所在的外部类方法的参数
public class Outer {
    public static void main(String[] args) {
        Outer outer = new Outer();
        Inner inner = outer.getInner("Inner", "gz");
        System.out.println(inner.getName());
    }

    //注意这里的形参 city，由于它没有被匿名内部类直接使用，而是被抽象类 Inner 的构造函数所使用，所以不必定义为 final
    public Inner getInner(String name, String city) {
        return new Inner(name, city) {     // OK，形参 name 和 city 没有被匿名内部类直接使用
            private String nameStr = name;

            public String getName() {
                return nameStr;
            }
        };
```

```
    }

}



abstract class Inner {

    Inner(String name, String city) {

        System.out.println(city);

    }



    abstract String getName();

}
```

从上述代码中可以看到，当匿名内部类直接使用其所在的外部类方法的参数时，那么这些参数必须被设为 final 的。为什么呢？本文所引用到的一篇文章是这样解释的：

“这是一个编译器设计的问题，如果你了解 java 的编译原理的话很容易理解。首先，内部类被编译的时候会生成一个单独的内部类的.class 文件，这个文件并不与外部类在同一 class 文件中。当外部类传的参数被内部类调用时，从 java 程序的角度来看是直接的调用，例如：

```
public void dosome(final String a,final int b){

  class Dosome{

      public void dosome(){

          System.out.println(a+b)

      }

  };
```

```
  Dosome some=new Dosome();
  some.dosome();
}
```

从代码来看，好像是内部类直接调用的 a 参数和 b 参数，但是实际上不是，在 java 编译器编译以后实际的操作代码是:

```
class Outer$Dosome{
  public Dosome(final String a,final int b){
      this.Dosome$a=a;
      this.Dosome$b=b;
  }
  public void dosome(){
      System.out.println(this.Dosome$a+this.Dosome$b);
  }
}
```

从以上代码来看，内部类并不是直接调用方法传进来的参数，而是内部类将传进来的参数通过自己的构造器备份到了自己的内部，自己内部的方法调用的实际是自己的属性而不是外部类方法的参数。这样就很容易理解为什么要用 final 了，因为两者从外表看起来是同一个东西，实际上却不是这样，如果内部类改掉了这些参数的值也不可能影响到原参数，然而这样却失去了参数的一致性，因为从编程人员的角度来看他们是同一个东西，如果编程人员在程序设计的时候在内部类中改掉参数的值，但是外部调用的时候又发现值其实没有被改掉，这就让人非常的难以理解和接受，为了避免这种尴尬的问题存在，所以编译器设计人员把内部类能够使用的参数设定为必须是 final 来规避这种莫名其妙错误的存在。”

以上关于匿名内部类的每个例子使用的都是默认无参构造函数，下面我们介绍 **带参数构造函数的匿名内部类**：

```
public class Outer {

    public static void main(String[] args) {

        Outer outer = new Outer();

        Inner inner = outer.getInner("Inner", "gz");

        System.out.println(inner.getName());

    }



    public Inner getInner(final String name, String city) {

        return new Inner(name, city) {   //匿名内部类

            private String nameStr = name;

            public String getName() {

                return nameStr;

            }

        };

    }

}
```

```
abstract class Inner {

    Inner(String name, String city) {   // 带有参数的构造函数

        System.out.println(city);

    }



    abstract String getName();

}
```

特别地，**匿名内部类通过实例初始化 (实例语句块主要用于匿名内部类中)，可以达到类似构造器的效果，**如下：

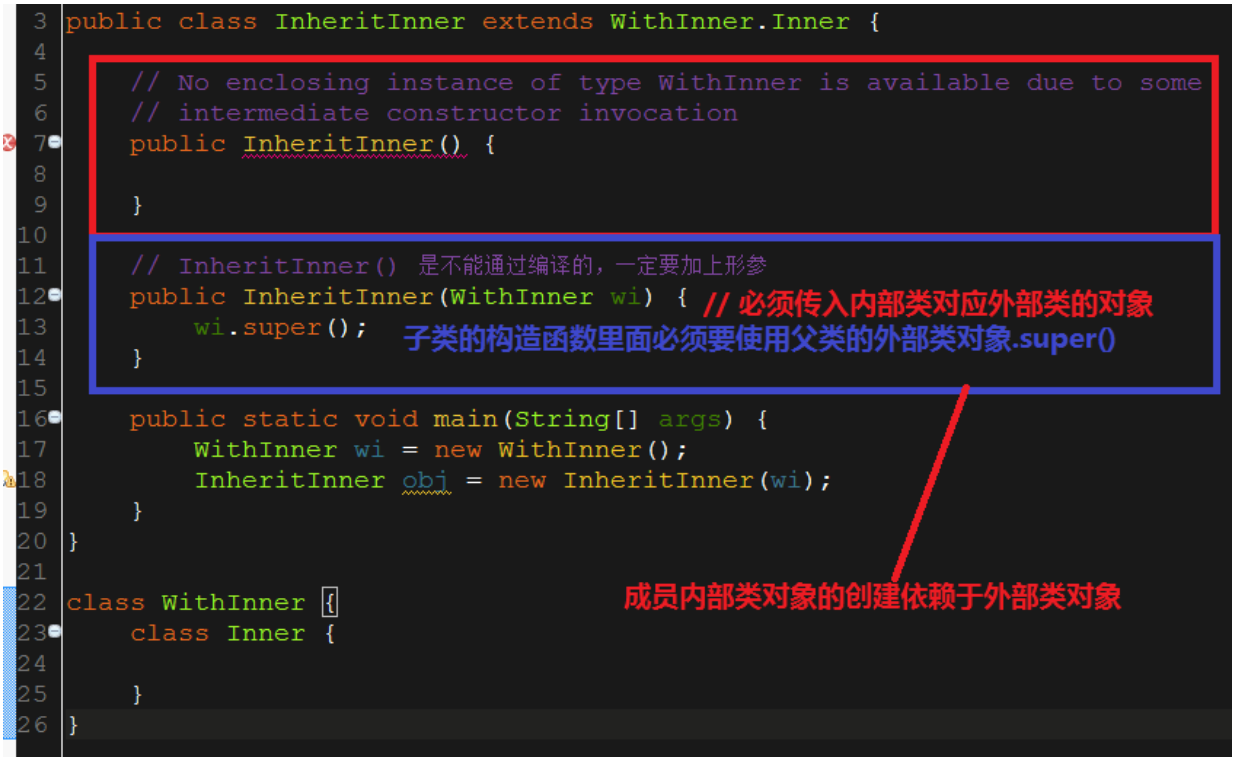
```
public class Outer {

    public static void main(String[] args) {

        Outer outer = new Outer();

        Inner inner = outer.getInner("Inner", "gz");

        System.out.println(inner.getName());

        System.out.println(inner.getProvince());

    }



    public Inner getInner(final String name, final String city) {

        return new Inner() {

            private String nameStr = name;
```

```
        private String province;


        // 实例初始化
        {
            if (city.equals("gz")) {
                province = "gd";
            }else {
                province = "";
            }
        }

        public String getName() {
            return nameStr;
        }


        public String getProvince() {
            return province;
        }
    };
}
```

```
}
```

六. 内部类的继承

内部类的继承，是指内部类被继承，普通类 extents 内部类。而这时候代码上要有点特别处理，具体看以下例子：



可以看到，**子类的构造函数里面要使用父类的外部类对象.super() [成员内部类对象的创建依赖于外部类对象]；而这个外部类对象需要从外面创建并传给形参。**