

Android 基本功三 Java 的类

搜索关键字:

内部类

匿名类

匿名对象

回调方法(回调函数)

内存回收机制

1. 内部类知识点简介

【知识点】Android SDK 与 Java SDK 之间的关系。

首先介绍一下什么叫作 SDK。SDK 通常指：软件开发工具包（外语全称：Software Development Kit）一般都是一些被软件工程师用于为特定的软件包、软件框架、硬件平台、操作系统等建立应用软件的开发工具的集合。

SDK 通常是为了开发某一个方面的程序软件，由厂商提供的集成封装的库（library），通常比较底层，通用性强。例如，Windows 的 API 也可以看作是一个 SDK。如：Java Develop Toolkit，就是针对 JAVA 语言的 SDK。

Android 虽然使用 Java 语言作为开发工具，但是在实际开发中发现，还是与 Java SDK 有一些不同的地方。Android SDK 引用了大部分的 Java SDK，少部分被 Android SDK 抛弃，比如说界面部分，`java.awt package` 除了 `java.awt.font` 被引用外，其他都被抛弃，在 Android 平台开发中不能使用。Android SDK 与 Java SDK 的具体细节区别，有兴趣的读者可查阅搜索引擎，在此就不详细展开了。

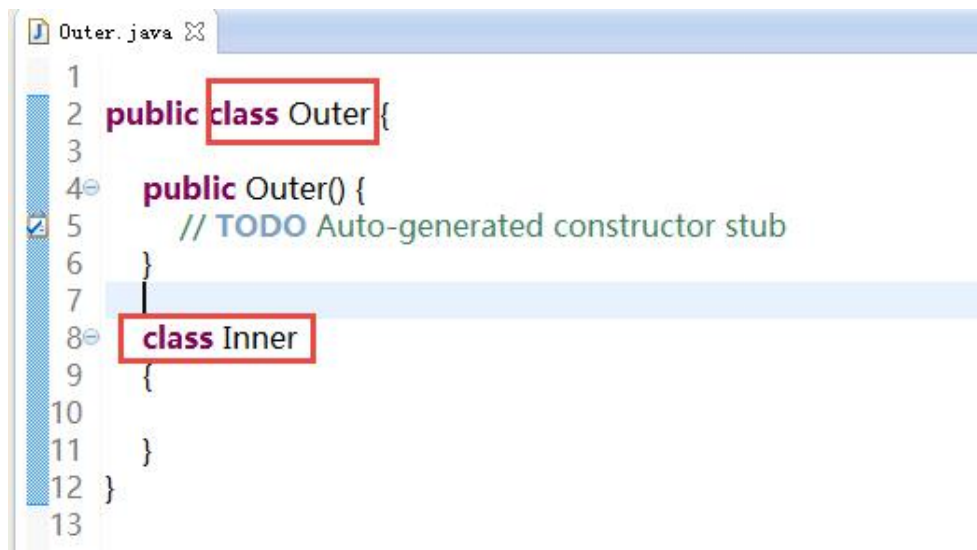
本篇并没有按照“编年体”的方式来讲解 JAVA 语言，因为这里并不是定位专门讲解 JAVA 语言的教程，而是假设读者有一定的编程基础（如 C++），然后选取了 JAVA 相比面向对象语言比较特殊的地方，如“内部类”、“事件监听”、“多线程”和“异常处理”等方面进行讲解。

本章主要讲解 JAVA 中类之间的关系。

1.1 内部类是什么

在一个类的内部定义的类称为内部类(或嵌套类)，包含内部类的类称为外部类。如 Outer 是外部类，Inner 是内部类。内部类与外部类的称呼是相对的。

如图 4A-1 所示，Outer 是一个外部类，Inner 是一个内部类。



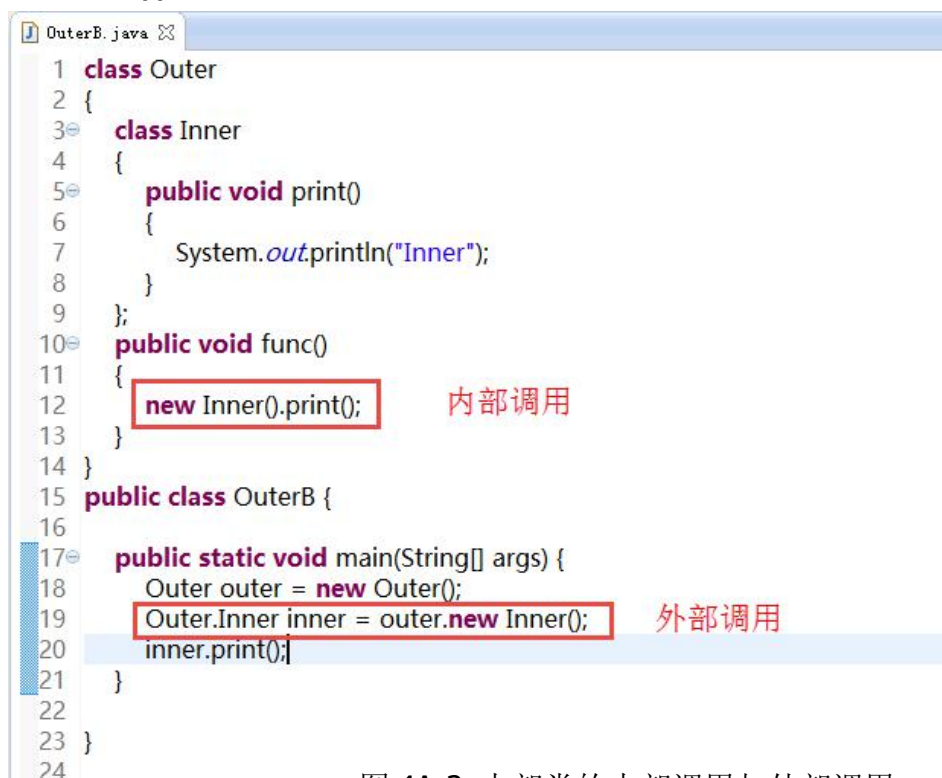
```

1
2 public class Outer {
3
4     public Outer() {
5         // TODO Auto-generated constructor stub
6     }
7
8     class Inner
9     {
10
11     }
12 }
13

```

图 4A-1 外部类与内部类

由图 4A-1 可以看出，内部类是依赖于外部类而存在的，如果需要在外部类之外定义内部类的对象，则应使用 外部类名.内部类名 的格式来表示内部类。如图 4A-2 所示：在外部类 **Outer** 外面使用内部类 **Inner** 对象，必须以 **Outer.Inner** 表示内部类，假若是在内部类 **Outer** 里面使用外部类 **Inner** 对象，则不必写上外部类名和点号(.)。



```

1 class Outer
2 {
3     class Inner
4     {
5         public void print()
6         {
7             System.out.println("Inner");
8         }
9     };
10    public void func()
11    {
12        new Inner().print();    内部调用
13    }
14 }
15 public class OuterB {
16
17    public static void main(String[] args) {
18        Outer outer = new Outer();
19        Outer.Inner inner = outer.new Inner();    外部调用
20        inner.print();
21    }
22
23 }
24

```

图 4A-2 内部类的内部调用与外部调用

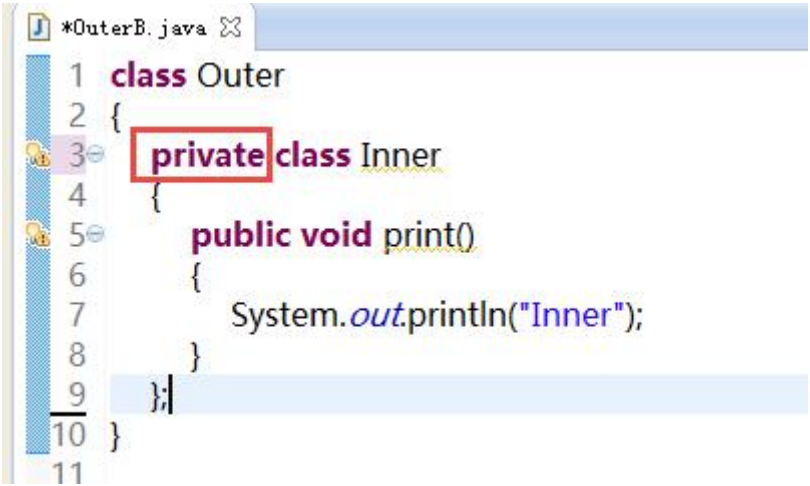
1.2 内部类的种类

类型	说明
成员类	作为类的成员而存在在某一类的内部
内部类	存在于某一方法内的类
静态类	作为类的静态成员存在于某一类的内部，用关键字 static 修饰
匿名内部类	存在于某一类的内部，但无名称的类

1.2.1 成员内部类（成员类）

成员类是作为外部类的成员而存在的，也就是说，成员类与外部类的成员变量和成员方法的地位是一样的。成员类是最常见的内部类，如图 4A-1 所示。

【注意】在内部类中可加上 **private**、**public**、**protected** 修饰符。如图 4A-3 所示。



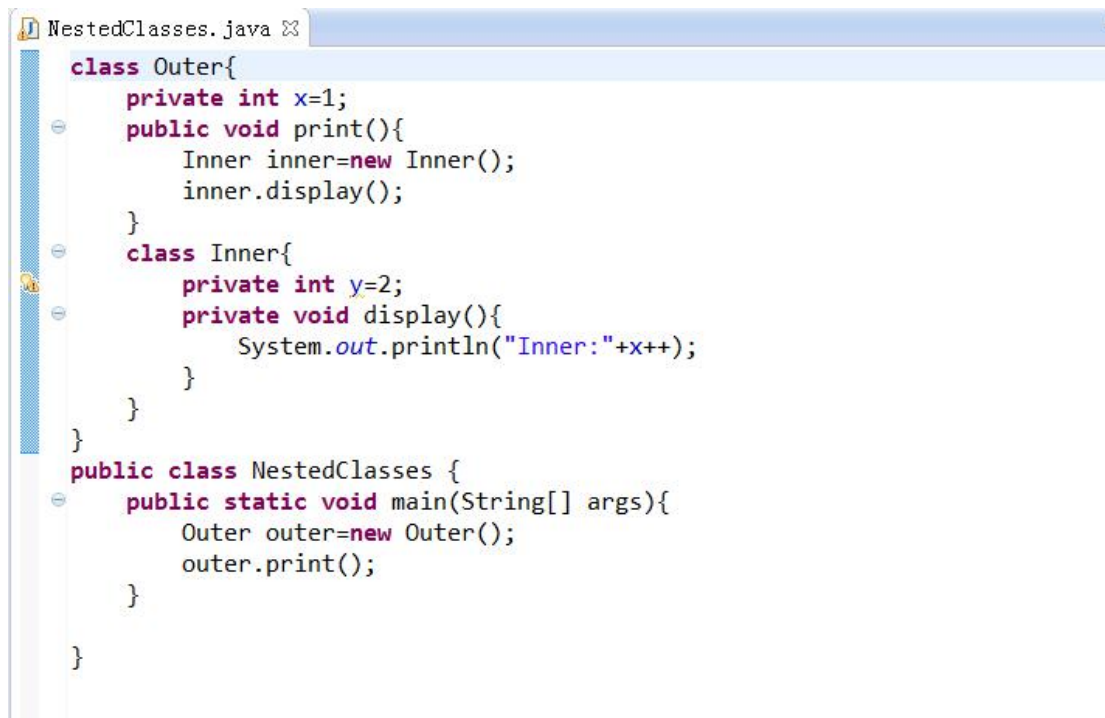
```
*OuterB.java
1 class Outer
2 {
3     private class Inner
4     {
5         public void print()
6         {
7             System.out.println("Inner");
8         }
9     }
10 }
11
```

图 4A-3 内部类可以加上权限修饰符

1. 成员类案例 01

案例设计（成员类）：
外部类 Outer 中包含一个内部类 Inner,在主函数中创建 Outer 的对象 outer,对象 outer 调用 Outer 的 print()方法，在外部类 Outer 中又包含了一个内部类 Inner。在外部类中的 print()方法中我们创建了内部类 Inner 的对象 inner，并且调用了其方法 display()。

案例实现，如图 4A-4 所示。



```


class Outer{
    private int x=1;
    public void print(){
        Inner inner=new Inner();
        inner.display();
    }
    class Inner{
        private int y=2;
        private void display(){
            System.out.println("Inner:"+x++);
        }
    }
}

public class NestedClasses {
    public static void main(String[] args){
        Outer outer=new Outer();
        outer.print();
    }
}

```

图 4A-4 在外部类调用内部类的方法

输出，如图 4A-5 所示：



```

<terminated> NestedClasses [Java Application] D:\Program Files (x86)\Java\jdk1
Inner:1

```

图 4A-5 输出结果

案例注意的问题：

1) 访问权限的规则

- 1: 内部类可以直接访问外部类的成员(包括私有成员)。
- 2: 外部类不可以直接访问内部类成员(因为内部类的成员只有在内部类范围内是可见的),可以通过先创建外部类实例,再创建内部类实例来调用内部类的变量和方法。

2) 内部类的适用场合：一个类的程序代码要用到另一个类的实例，而另一个类的程序代码又要访问第一个类的成员，将另一个类做成第一个类的内部类，程序代码的编写就要容易得多。这种情况在实际应用中很多(如事件处理)。

3) 如果内部类与外部类的变量同名，则可以在内部类中采用如下格式进行区别：**this.变量名** -> 内部类的变量、**外部类名.this.变量名** -> 外部类的变量。

2. 成员类案例 02

案例实现：如图 4A-6 所示

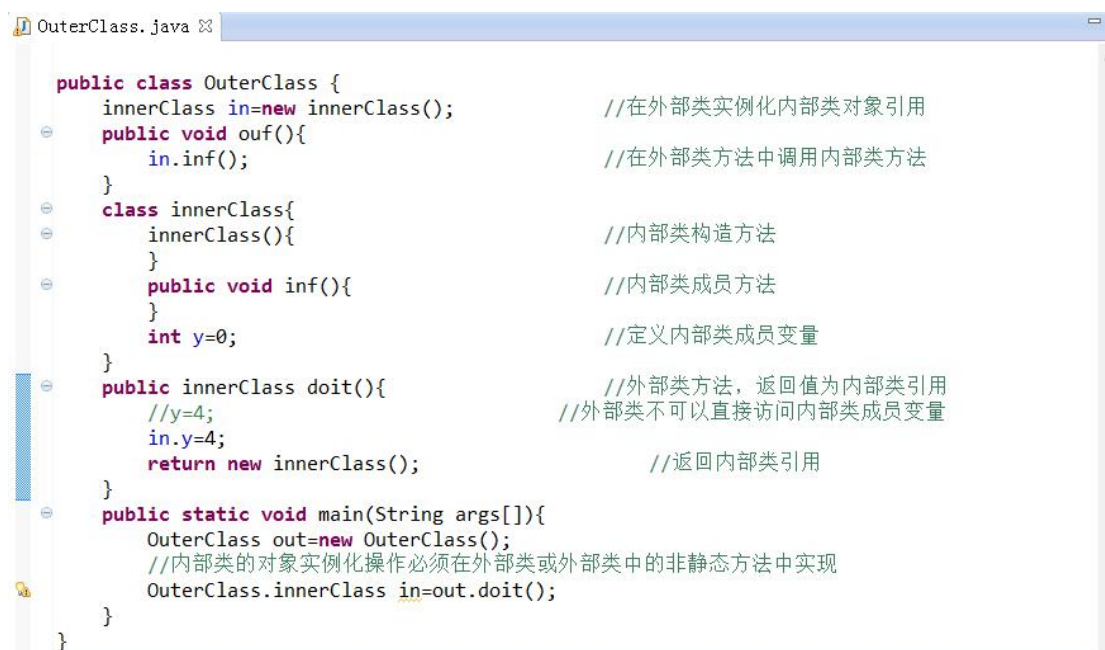


图 4A-6 在外部类调用内部类方法

案例注意的问题：

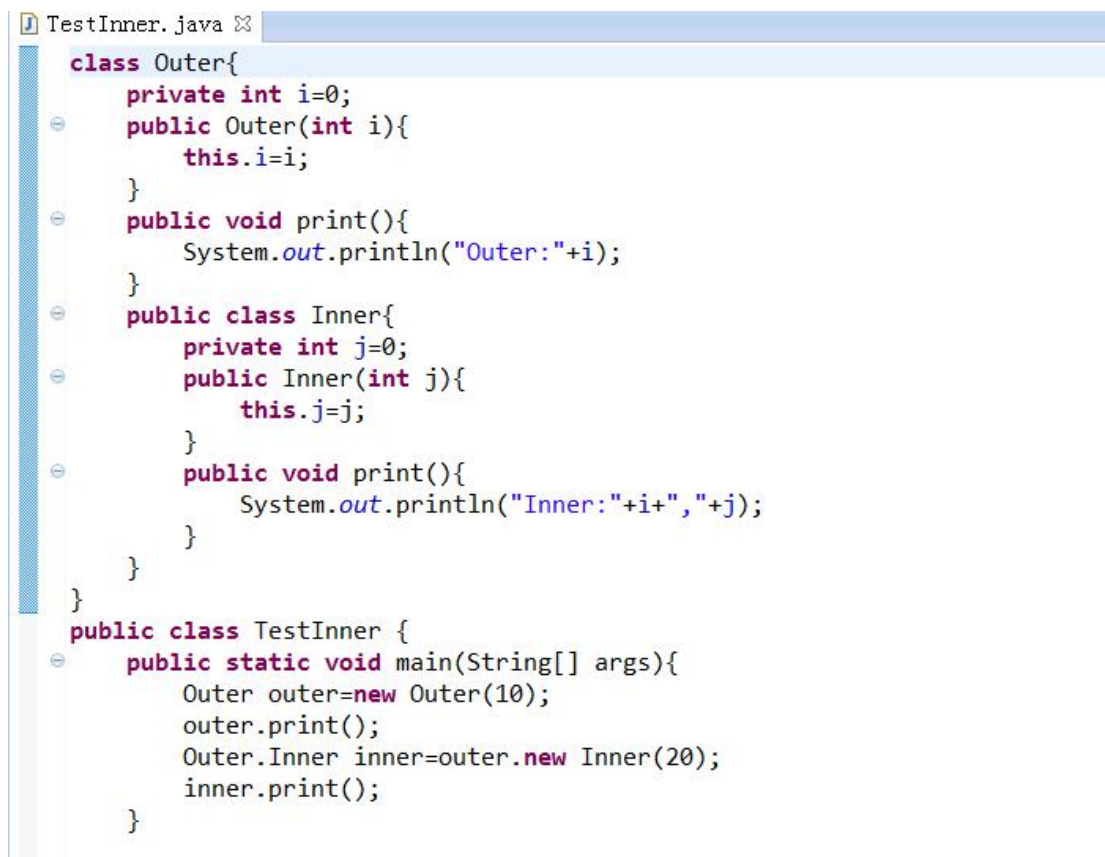
此例子的外部类创建内部类实例时与其他类创建对象引用时相同。内部类可以访问它的外部类的成员，但内部类的成员只有在内部类的范围之内是可知的，不能被外部类使用。如果将内部类的成员变量 **y** 再次赋值时将会出错，但是如果使用内部类对象引用调用成员变量 **y** 即可。

3. 成员类案例 03

案例设计：

主函数中创建 **Outer** 外部类的对象 **outer** 并传入形参 **10** 调用了其构造方法进行初始化，继而通过对象 **outer** 调用了其 **print()** 方法，输出属性 **i** 的值.通过类似的方法我们可以利用外部类 **Outer** 的对象 **outer** 创建出内部类 **Inner** 的对象 **inner**，利用 **inner** 对象调用其内部类所特有的 **print()** 方法。

案例实现，如图 4A-7 所示：



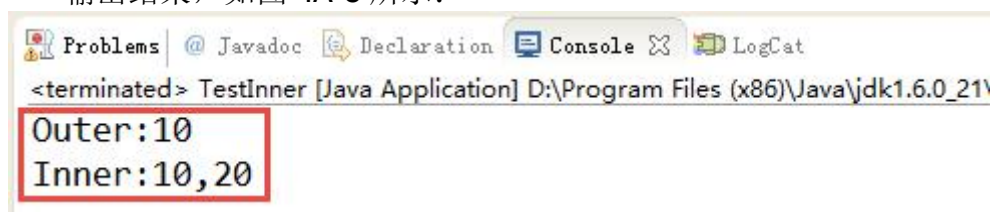
```

class Outer{
    private int i=0;
    public Outer(int i){
        this.i=i;
    }
    public void print(){
        System.out.println("Outer:"+i);
    }
    public class Inner{
        private int j=0;
        public Inner(int j){
            this.j=j;
        }
        public void print(){
            System.out.println("Inner:"+i+","+j);
        }
    }
}

public class TestInner {
    public static void main(String[] args){
        Outer outer=new Outer(10);
        outer.print();
        Outer.Inner inner=outer.new Inner(20);
        inner.print();
    }
}

```

图 4A-7 外部类分别调用自己以及内部类的方法
输出结果，如图 4A-8 所示：



```

<terminated> TestInner [Java Application] D:\Program Files (x86)\Java\jdk1.6.0_21\
Outer:10
Inner:10,20

```

案例注意的问题： 图 4A-8 输出结果

必须首先创建外围对象然后才可用它创建内部对象。

1. 静态成员类的优点在于：允许在成员类中声明静态成员。

其特点如下：

- a. 实例化静态类时，在 new 前面不需要用对象变量。
- b. 静态类中只能访问其外部类的静态成员。
- c. 静态方法中不能不带前缀地实例化一个非静态类。

1.2.2 局部内部类(简称局部类)

局部类是包含在外部类的某一方法中的内部类，其作用域是在局限于该方法的范围，地位与方法中的局部变量一样。**局部类只有在方法内部才能创建对象，一旦方法执行完毕，它就会释放内存而消亡。**如图 4A-9 所示。

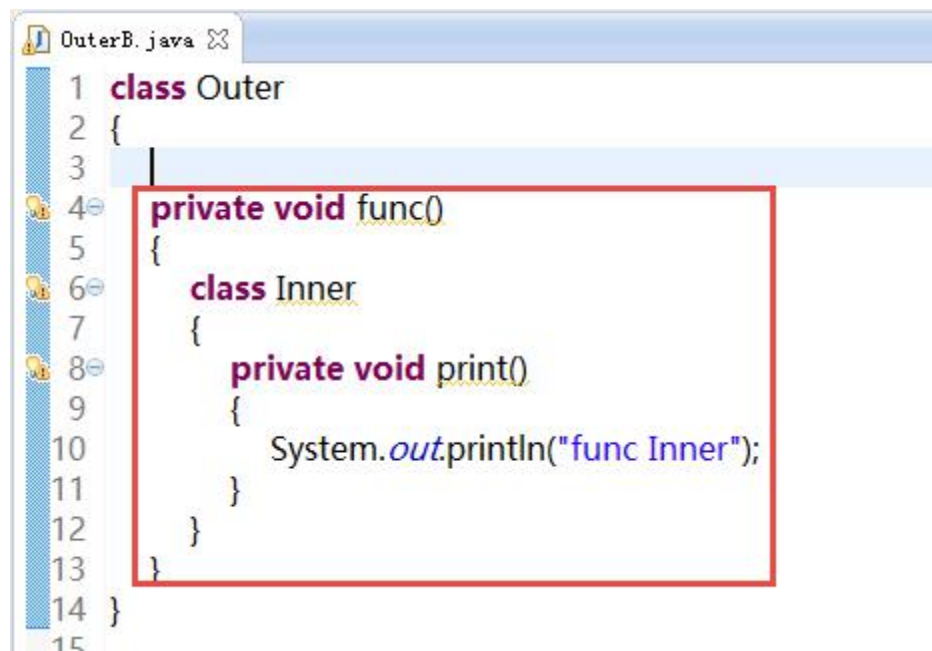


图 4A-9 局部内部类

- 1) 局部类可直接访问外部类成员，但是对象的创建是在方法内进行的。
- 2) 局部类只访问用 **final** 修饰的局部变量和形参。原因是：局部变量会随着方法的退出而消亡，通过将其定义为 **final** 变量，可以扩展其生命周期，可与访问其类实例的生命期相配合。因为类实例的生命期是由内存的回收机制决定的。
- 3) 局部类的作用域仅限于其直接外围块。因而局部类不可使用访问控制修饰符 **public** 、 **protected** 、 **private**。

1. 局部内部类案例 01

案例设计：

在外部类的 **sell** 方法中创建 **Apple** 局部内部类，然后创建该内部类的实例，并调用其定义的 **price()** 方法输出单价信息。

案例实现，如图 4A-10 所示：

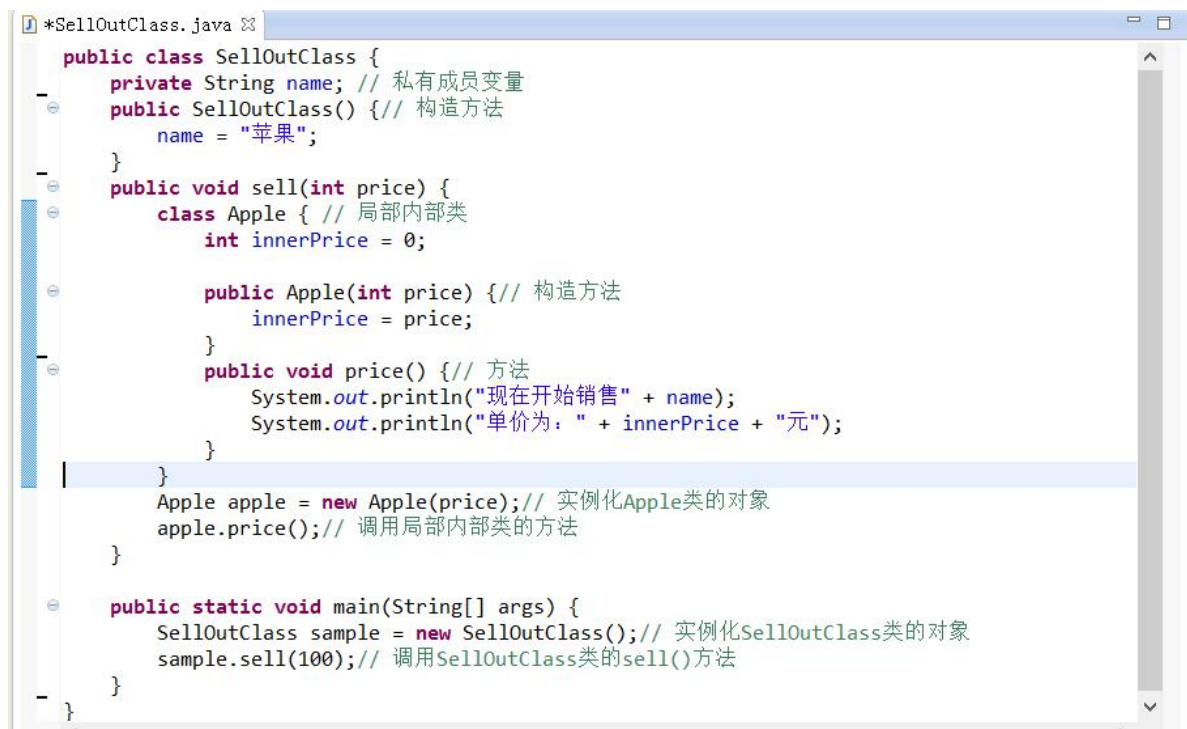


图 4A-10 调用局部内部类

输出，如图 4A-11 所示：



图 4A-11 输出结果

案例注意的问题：

将内部类定义在 `sell()` 方法内部。但是有一点值得注意，内部类 `Apple` 是 `sell()` 方法的一部分，并非 `SellOutClass` 类的一部分，所以在 `sell()` 方法的外部不能访问该内部类，但是该内部类可以访问当前代码块的常量以及此外部类的所有成员。

1.2.3 静态内部类(简称嵌套类)

静态类是最简单的内部类形式，与其他静态变量一样，静态内部类只需要在定义的时候加上 `static` 关键字，同样，静态内部类只能访问外部类中的静态成员变量。如图 4A-12 所示。


```

public class staticInner {
    public static void main(String[] args) {
        OutClass.InnerClass A = new OutClass.InnerClass();
        A.print();
    }
}

class OutClass{
    static int a = 1;
    public static class InnerClass{
        void print(){
            System.out.println(a);
        }
    }
}

```

图 4A-12 静态内部类

一般静态内部类都被称为嵌套类，当一个内部类是 **static** 的时候，这意味着：

- 1) 创建嵌套类对象并不需要外部类的对象。
- 2) 不能从嵌套类的对象中访问非静态类的外部类对象。

1. 静态内部类案例 01

案例设计：

在 `main()` 中访问使用静态内部类方法。

案例实现，如图4A-13所示：

```

2
3 public class Outer {
4
5     static int x = 1;
6
7     static class Nest {
8         void print(){
9             System.out.println("Nest "+x);
10        }
11    }
12
13    public static void main(String[] args) {
14        Outer.Nest nest = new Outer.Nest();
15        nest.print();
16    }
17 }

```

图4A-13 访问静态内部类方法

输出如图4A-14所示：



图 4A-14 输出结果

案例注意问题：

因为静态嵌套类和其他静态方法一样只能访问其它静态的成员，而不能访问实例成员。因此静态嵌套类和外部类（封装类）之间的联系就很少了，他们之间可能也就是命名空间上的一些关联。而上例需要注意的就是静态嵌套类的声明方法 `new Outer.Nest()` 连续写了两个类名，以至于会怀疑前面的 `Outer` 是个包名。

1.2.4 匿名内部类（简称内部类）

匿名类是指没有自己名字的内部类，而且必须是非静态类。匿名类是不能有名称的类，所以没办法引用它们。必须在创建时，作为 `new` 语句的一部分来声明它们。如图4A-15所示。

```
abstract class Person {  
    public abstract void name();  
}  
  
public class Contents {  
    public static void main(String[] args) {  
        Person p = new Person() {  
            public void name() {  
                System.out.println("Andy");  
            }  
        };  
        p.name();  
    }  
}
```

图 4A-15 匿名内部类

在使用匿名内部类时，要记住以下几个原则：

- 1) 匿名内部类不能有构造方法。
- 2) 匿名内部类不能定义任何静态成员、方法和类。
- 3) 匿名内部类不能是 **public,protected,private,static**。
- 4) 只能创建匿名内部类的一个实例。
- 5) 一个匿名内部类一定是在 `new` 的后面，用其隐含实现一个接口或实现一个类。
- 6) 因匿名内部类为局部内部类，所以局部内部类的所有限制都对其生效。

【注意】匿名类和内部类中的中的 **this** :

有时候,我们会用到一些内部类和匿名类。当在匿名类中用 **this** 时,这个 **this** 则指的是匿名类或内部类本身。这时如果我们要使用外部类的方法和变量的话,则应该加上外部类的类名。

1. 匿名内部类案例 01

案例设计:

在 **main()** 方法中编写匿名内部类去除字符串中的全部空格。首先声明 **IStringDeal** 接口,在接口中又声明了一个过滤字符串中的空格的方法,在主函数中我们实现了 **IStringDeal** 接口并且重写了 **filterBlankChar()** 方法。

案例实现,如图 4A-16 所示:



图 4A-16 调用匿名内部类

输出,图 4A-17:



图 4A-17 输出结果

案例注意的问题:

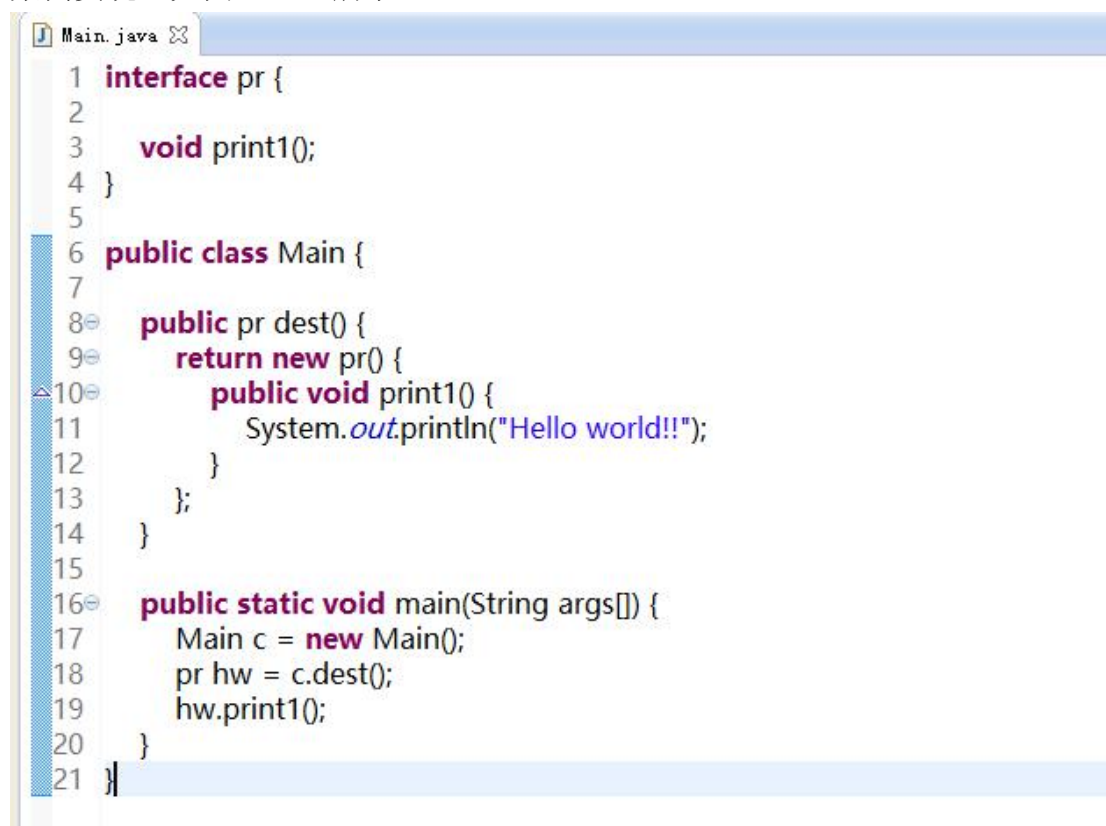
匿名类因为没有名字,所以匿名类不能有自己的构造方法。所以在初始化问题上,一般匿名类利用局部变量或形式参数完成初始化。在 **GUI** 事件编程中,大量使用匿名内部类。

2. 匿名内部类案例 02

案例设计：

匿名类继承事件类重写回调方法，生成匿名对象，再调用匿名对象的方法。

案例实现，如图 4A-18 所示。



```
1 interface pr {
2
3     void print1();
4 }
5
6 public class Main {
7
8     public pr dest() {
9         return new pr() {
10             public void print1() {
11                 System.out.println("Hello world!!");
12             }
13         };
14     }
15
16     public static void main(String args[]) {
17         Main c = new Main();
18         pr hw = c.dest();
19         hw.print1();
20     }
21 }
```

图 4A-18 重写回调方法调用匿名内部类
输出字符串，如图 4A-19 所示。



```
<terminated> Main [Java Application] D:\Program Files (x86)\Java\jdk1.6.0_21\bin\javaw.exe
Hello world!!
```

图 4A-19 输出结果

案例需要注意的问题：

使用匿名内部类必须继承一个父类或实现一个接口。

1.3 为什么要使用内部类

结束了基础内部类的学习之后，相信还有不少读者不知道在 JAVA 中，内部类到底起着什么样的作用。这是因为内部类是 JAVA 中高级编程的应用，那么我们就来总结一下内部类的一些特点。

1. 内部类只能依靠其外部类存在，在内部类的定义中我们可以声明 public、

protected、private 等访问权限，但是内部类可以访问其外部类中的所有成员，无论是否 **private**。这样，就可以起到一个隐藏代码的作用，同时也提供了一个进入外部类的窗口。

2. 说起内部类，最吸引人的就是内部类可以直接继承一个接口，而并不用管外部类是否继承这个接口。这样，内部类使得 **JAVA** 不支持的“多重继承”的解决方案变得完整。

2. 匿名对象

2.1 匿名对象概念

概念：匿名对象是在一个对象被创建之后，调用对象的方法或属性时可以不定义对象的引用变量。

2.2 匿名对象案例

例一：

如图 4A-20 所示。

```
2 public class Contents {
3     public static void main(String[] args) {
4         test(new A());
5     }
6 }
7
```

图 4A-20 调用匿名对象

其中图 4A-20 中第 4 行代码的 `test(new A());` 语句等价于图 4A-21 中的第 5 行到第 6 行所示。而在图 4A-20 中的 `new A()` 就是一个匿名对象。

```
2
3 public class Contents {
4     public static void main(String[] args) {
5         A a = new A();
6         test(a);
7     }
8 }
```

图 4A-21 平时的调用方法

例二：

如图 4A-22 所示。“abc”.equals(str) 一个字符串能够调用一个函数，我们就可以看出来：一个字符串是 **String** 的匿名对象。


```

2 public class Contents {
3     public static void main(String[] args) {
4         String str = new String("abc");
5         if("abc".equals(str)){
6             System.out.println(1);
7         }
8         else{
9             System.out.println(2);
10        }
11    }
12 }

```

图 4A-22 字符串调用方法

在 JAVA 中使用匿名对象能够使代码简洁并提高代码的可阅读性。

3.回调函数

所谓回调，就是客户程序 C 调用服务程序 S 中的某个函数 A，然后 S 又在某个时候反过来调用 C 中的某个函数 B，对于 C 来说，这个 B 便叫做回调函数。

一般说来，C 不会自己调用 B，C 提供 B 的目的就是让 S 来调用它，而且是 C 不得不提供。由于 S 并不知道 C 提供的 B 姓甚名谁，所以 S 会约定 B 的接口规范（函数原型），然后由 C 提前通过 S 的一个函数 R 告诉 S 自己将要使用 B 函数，这个过程称为回调函数的注册，R 称为注册函数。Web Service 以及 Java 的 RMI 都用到回调机制，可以访问远程服务器程序。

3.1 回调原理图

如图 4A-23 所示。

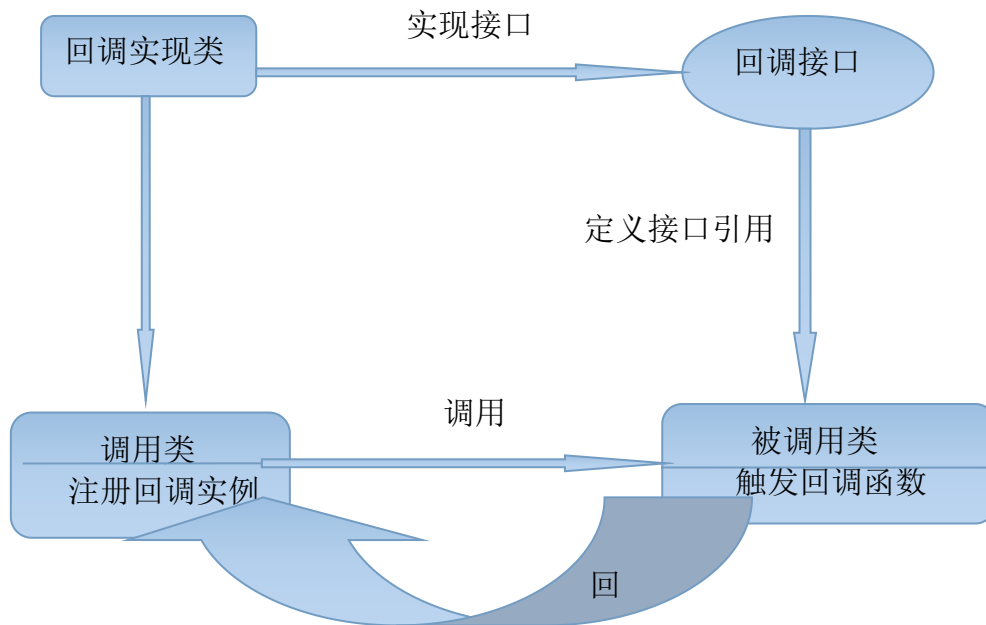


图 4A-23 回调原理

3.2 回调函数案例

案例设计：

Test 是一个用于测试的调用者类，它通过 **main** 方法中实例化一个 **FooBar**，并用实现的 **ICallBack** 的匿名类作为参数传递给 **FooBar** 的被调用方法 **setCallBack**，而在这个虚拟方法中，**FooBar** 调用了匿名类的匿名类的 **postExec** 方法的动作，这个动作就是回调（**Callback**）。

案例实现：如图 4A-24 所示。



图 4A-24 使用回调函数

输出结果如图 4A-25 所示：



图 4A-25 回调函数输出结果

4.回收机制

回收机制概念：在 Java 程序运行过程中，垃圾回收器是以后台线程方式运行，它会被不定时地被唤醒以检查是否有不再被使用的对象，以释放它们所占的内存空间。

1. finalize() 是 Object 类的一个方法，它可以被其它类继承或改写。finalize() 的作用：在对象被当做垃圾从内存中释放前调用，并不是在对象变成垃圾前被调用。由于垃圾回收器的启用是不定时的，因此，finalize() 方法的调用并不可靠。

2. System.gc() 作用：强制启动垃圾回收器来回收垃圾（指不再使用的对象）。

3. 在 Java 语言中，采用 new 为对象申请内存空间，至于内存空间的释放和回收，是由 Java 运行系统来完成的，用户可以完全不管，这样可以避免内存泄漏和无用内存的调用两类错误的产生，这种机制被称为垃圾回收机制。C++ 程序

使用内存空间的策略：用 new 申请空间，用 delete 归还空间，否则，会产生内存泄漏。

4. 垃圾回收器的启动不用程序员控制，也无规律可循。并不会一产生了垃圾，它就被唤醒，甚至可能程序终止，它都没有启动的机会。当内存空间严重不足或调用相关外部命令时，它会被唤醒。

5. 项目心得

在本章学习中，主要讲述了 Java 中的内部类、匿名对象、回调函数以及垃圾回收机制。

本章的重点是内部类的学习。下面来总结下四种的内部类的使用场景：

静态内部类：作为类的静态成员存在于某一类的内部，用关键字 **static** 修饰。最简单的内部类，主要用于**访问静态对象**。

成员内部类：作为类的成员而存在在某一类的内部。最常用的**内部类种类**。

匿名内部类：存在于某一类的内部，但无名称的类。常用于需要**隐藏的类**。

局部内部类：存在于某一方法内的类。常用于用完就需要丢弃的类，会**随着方法的消逝而消逝**。

通过学习本章的内容，希望能加深读者对 Android 中对 Java 语法代码以及系统运行机制的理解。