Java软件设计基础

第六章 内部类

1. 简介

内部类

某些情况下，会把一个类A放在另一个类B的内部定义，前者A就被称为内部类（嵌套类），B被称为外部类（宿主类）。

- 作用
  - 提供了更好的封装，可以把内部类隐藏在外部类之内，不允许同一个包中的其他类访问该类；
  - 作为外部类的“成员”之一，内部类成员可以直接访问外部类的私有数据；但外部类则不能访问内部类的实现细节。
- 语法
  - 与定义普通类的语法大致相同；
  - 可以多使用三个修饰符：private、protected、static；
  - 非静态内部类不能拥有静态成员。

- 大部分时候，内部类被作为成员内部类定义，而不是作为局部内部类，它的地位与成员变量、方法类似。

```
public class OuterClass{
        内部类的定义……
}
```

Ex1

- 局部内部类和匿名内部类则不是类成员。

2. 非静态内部类

- 简而言之，非静态内部类即不加static修饰符的内部类。

```
public class Cow{
    private double weight;
    public Cow(){}
    public Cow(double weight){this.weight=weight;}
    private class CowLeg{
        private double length;
        private String color;
        public CowLeg(){}
        public CowLeg(double length,String color){
            this.length=length;
            this.color=color;
        }
        public void info(){
            System.out.println("当前牛腿颜色是："+color+",长度"+length);
            System.out.println("本牛腿所在的奶牛重："+weight);
        }
    }
    public void test(){
        CowLeg cl=new CowLeg(1.12,"黑色");
        cl.info();
    }
    public static void main(String[] args) {
        Cow cow=new Cow(378.9);
        cow.test();
    }
}
```

Ex2

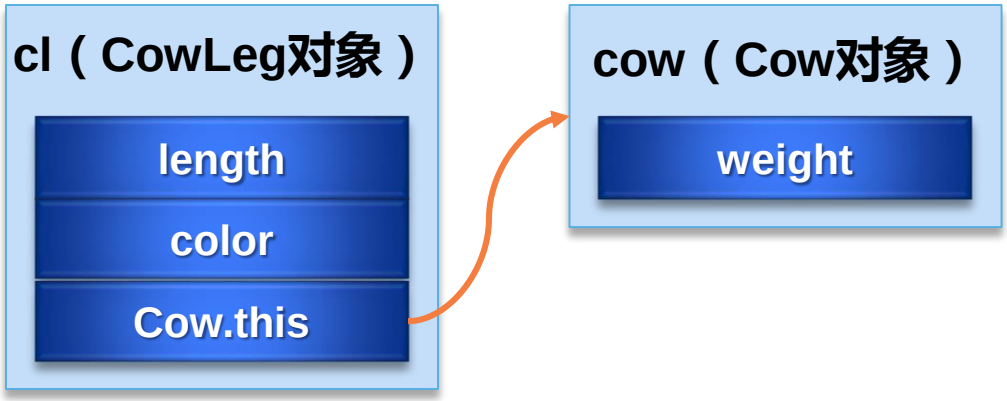
- 编译与运行
  - 产生两个类，前者即为内部类，后者是外部类；

```
Cow$CowLeg.class
Cow.class
```

OuterClass$InnerClass.class

```
当前牛腿颜色是：黑色,长度1.12
本牛腿所在的奶牛重：378.9
```

  - 非静态内部类对象总保存着一个它所寄生的外部类对象的引用：



```
public class Outer2{
	private int h;
	public Outer2(int h){
		this.h=h;
	}
	class Inner{
		public void info(){
		   System.out.println(Outer2.this.h);
		}
	}
	public static void main(String[] args) {
		Outer2 o1=new Outer2(5);
		Outer2 o2=new Outer2(9);
		Outer2.Inner i1=o1.new Inner();
		Outer2.Inner i2=o1.new Inner();
		Outer2.Inner i3=o2.new Inner();
		i1.info();
		i2.info();
		i3.info();
		System.out.println(i1==i2);
	}
}
```

Ex3

```
E:\javadoc\inner>java Outer2
5
5
9
false
```

- 当内部类的局部变量、成员变量以及其外部类的成员变量名相同时，通过以下方式获取相应的变量：
  - varName：局部变量；
  - this.varName：内部类的同名成员变量；
  - OuterClassName.this.varName：外部类的同名成员变量。

⇨非静态内部类对象必须寄生在外部类对象里，即如果存在一个非静态内部类对象，则一定存在一个它所寄生的外部类对象；

⇨反之，外部类对象则不一定寄生了非静态内部类对象！

```
public void test(){
        System.out.println(color);
        CowLeg cl=new CowLeg(1.12,"黑色");
        cl.info();
    }
```

此时非静态内部类对象还未创建，即外部类对象**cow**中还没有**CowLeg**的实例寄生

Ex4

```
public static void main(String[] args) {
        Cow cow=new Cow(378.9);
        new CowLeg();
        cow.test();
}
```

Ex5

根据静态成员不能访问非静态成员的规则，外部类的静态方法不能访问非静态内部类中的变量、方法等（包括构造方法）。

```
private class CowLeg{
    static int num=4;
    static void getNum(){
        System.out.println(num);
    }
}
```

Ex6

非静态内部类里不允许定义静态成员。

假设需要创建Cow类，Cow类则需要组合一个CowLeg对象，反之，CowLeg类只有在Cow类里才有效，离开了Cow类就没有存在的意义。
该例从某种程度上说明内部类的存在意义和应该定义的场合。

3. 静态内部类

- 静态内部类就是使用static修饰的内部类，也叫类内部类；该静态内部类属于外部类本身，而不属于外部类的某个对象。

```
public class StaticInnerClassTest{
    private int prop1=5;
    private static int prop2=9;
    static class StaticInnerClass{
        private static int age;
        public void accessOuterProp(){
            System.out.println(prop1);
            System.out.println(prop2);
        }
    }
}
```

Ex7

静态内部类里可定义静态、非静态成员

静态内部类里可访问外部类的静态成员

⇨静态内部类是外部类相关的，而不是外部类的对象相关的。内部类存在的时候并没有一个它寄生的外部类的对象，因此静态内部类不能访问外部类实例。

⇨静态内部类对象只持有外部类的类引用！

```
public class StaticInnerClassTest{
    private int prop1=5;
    private static int prop2=9;
    static class StaticInnerClass{
        private static int age=18;
        private String name="John";
        public void accessOuterProp(){
            System.out.println(prop2);
            System.out.println(new StaticInnerClassTest().prop1);
        }
    }
    public static void main(String args[]){
        System.out.println(StaticInnerClass.name);
        StaticInnerClass.accessOuterProp();
        System.out.println(StaticInnerClass.age);
        System.out.println(new StaticInnerClass().name);
        new StaticInnerClass().accessOuterProp();
    }
}
```

静态内部类里需通过外部类的实例才可以访问外部类的非静态成员

外部类静态方法中不可直接访问静态内部类的非静态成员

```
18
John
9
5
```

Ex8

4. 内部类的使用

- 在外部类内部使用
  - 不要在外部类的静态成员中使用非静态内部类（如Ex4）；
  - 在外部类内部定义内部类的子类与一般的定义子类方法相同；
- 在外部类以外使用非静态内部类
  - 访问控制修饰符对非静态内部类的作用与外部类的其他类成员相同；
  - 声明内部类对象的语法格式：

```
OuterClass.InnerClass varName
```

  - 创建内部类对象的语法格式：

```
OuterInstance.new InnerClassConstructor()
```

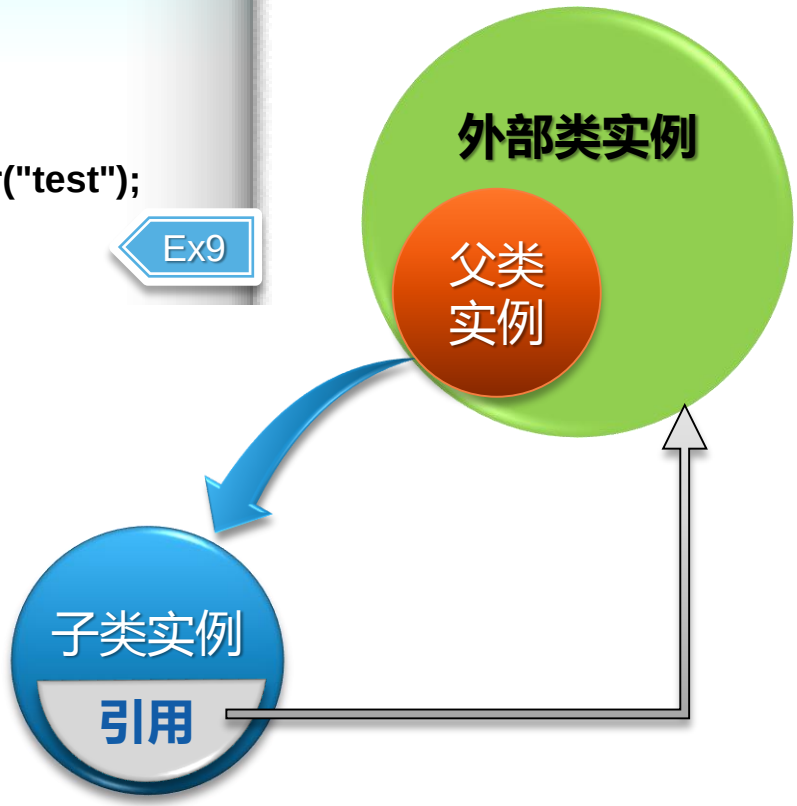
  即先要有内部类对象寄生的外部类对象，然后通过外部类对象调用该非静态内部类的构造方法创建内部类对象。

```
class Outer{
      class Inner{
            public Inner(String s){
               System.out.println(s);
            }
      }
}
public class CreateInnerInstance{
      public static void main(String[] args) {
            Outer.Inner in=new Outer().new Inner("test");
      }
}
```

Ex9

- 非静态内部类的子类
  - 外部类实例
  - 父类实例
  - 子类实例

非静态内部类的子类实例需要保留指向其父类所在外部类的对象的引用。用于通过该外部类对象调用其父类的构造方法



```
class Outer{
    class Inner{
        public Inner(String s){
            System.out.println(s);
        }
    }
}
class subClass extends Outer.Inner{
    public subClass(Outer o){
        o.super("hello");
    }
}
public class CreateInnerInstance{
    public static void main(String[] args) {
        Outer.Inner in=new Outer().new Inner("test");
        new subclass(new Outer());
    }
}
```

Ex10

super以subClass的角度来看，代表其父类Inner的构造方法的显式调用；由于Inner的构造需要由其外部类的对象调用，因此传递一个外部类对象o。

- 在外部类以外使用静态内部类
  - 因为静态内部类是外部类相关而非外部类对象相关的，因此无需创建外部类对象。

OuterClass.new InnerClassConstructor()

```
class StaticOut{
    static class StaticIn{
        public StaticIn(){
            System.out.println("This is the constructor of In");
        }
    }
}
class subClass extends StaticOut.StaticIn{ }
public class CreateStaticInnerInstance{
    public static void main(String[] args){
        StaticOut.StaticIn si=new StaticOut.StaticIn();
        new subClass();
    }
}
```

可直接通过外部类名调用构造方法

不再需要显式导入外部类对象来构造

Ex11

⇨使用静态内部类比使用非静态内部类简单得多，只要把外部类看成静态内部类的“包空间”即可。因此当程序需要使用内部类时，应优先考虑使用静态内部类。

5. 局部内部类

- 如果把内部类视作变量，那么定义在方法中的内部类就是局部内部类。其定义仅在该方法内部有效，不能使用访问控制修饰符和static修饰符。

```
public class LocalInnerClass{
    public static void main(String[] args){
        class InnerBase{
            int a;
        }
        class InnerSub extends InnerBase{
            int b;
        }
        InnerSub is=new InnerSub();
        is.a=5;
        is.b=7;
        System.out.println(is.a+","+is.b);
    }
}
```

Ex12

```
LocalInnerClass$1InnerBase.class
LocalInnerClass$1InnerSub.class
LocalInnerClass.class
```

⇨ Java为局部内部类的class文件名中加入了数字，因为不同的方法中有可能定义了同名的局部内部类。

6. 匿名内部类

- 用于创建只需要一次使用的类，创建时会立即创建一个该类的实例，而这个类的定义则立即消失，不能再次使用。

```
new 实现接口( ) | 父类构造方法(实参列表){
        匿名内部类的类体
}
```

  - 匿名内部类必须继承一个父类或实现一个接口，但最多只能继承一个父类或实现一个接口；
  - 匿名内部类不能为抽象类；
  - 匿名内部类不能显式定义构造方法（因为没有类名），因此其内部只有一个隐式的无参构造方法，new接口名的括号里不能传入参数值；
  - 可以定义初始化模块。

- 实现接口的匿名内部类

```
interface Product{
    public double getPrice();
    public String getName();
}
public class AnonymousTest{
    public void test(Product p){
        System.out.println(p.getName()+","+p.getPrice());
    }
    public static void main(String[] args){
        AnonymousTest ta=new AnonymousTest();
        ta.test(new Product(){
            public double getPrice(){
                return 67.8;
            }
            public String getName(){
                return "mouse";
            }
        });
    }
}
```

定义了一个实现了接口Product的匿名内部类。

Ex13

- 继承父类的匿名内部类

```
abstract class Device{
    private String name;
        public Device(){}
        public Device(String name){this.name=name;}
        public abstract double getPrice();
        public String getName(){return name;}
}
public class AnonymousInner{
    public void test(Device d){
        System.out.println(d.getName()+","+d.getPrice());
    }
    public static void main(String[] args) {
        AnonymousInner ai=new AnonymousInner();
        ai.test(new Device("mouse"){
            public double getPrice(){return 67.8;}
        });
        Device d=new Device(){
            {System.out.println("匿名内部类的初始化模块：");}
            public double getPrice(){return 777;}
            public String getName(){return "Monitor";}
        };
        ai.test(d);
    }
}
```

通过继承父类来创建匿名内部类时，匿名内部类将拥有和父类相似的构造方法，即拥有相同的形参列表

如有需要，可以重写父类的普通方法

Ex14

- Java 8以前的版本要求被局部内部类、匿名内部类访问的局部变量必须是显式final修饰的，从Java 8开始，如果局部变量被匿名内部类访问，则相当于自动使用了final修饰符。

```
interface A{
    void test();
}
public class ATest{
    public static void main(String[] args){
        int age=8;
        A a=new A(){
            public void test(){
                System.out.println(age);
                age=2;
            }
        };
        a.test();
    }
}
```

```
错误：从内部类引用的本地变量必须是最终变量或实际上的最终变量
                age=2;
                ^
```

Ex15

6. Lambda表达式

- 先看一个匿名内部类

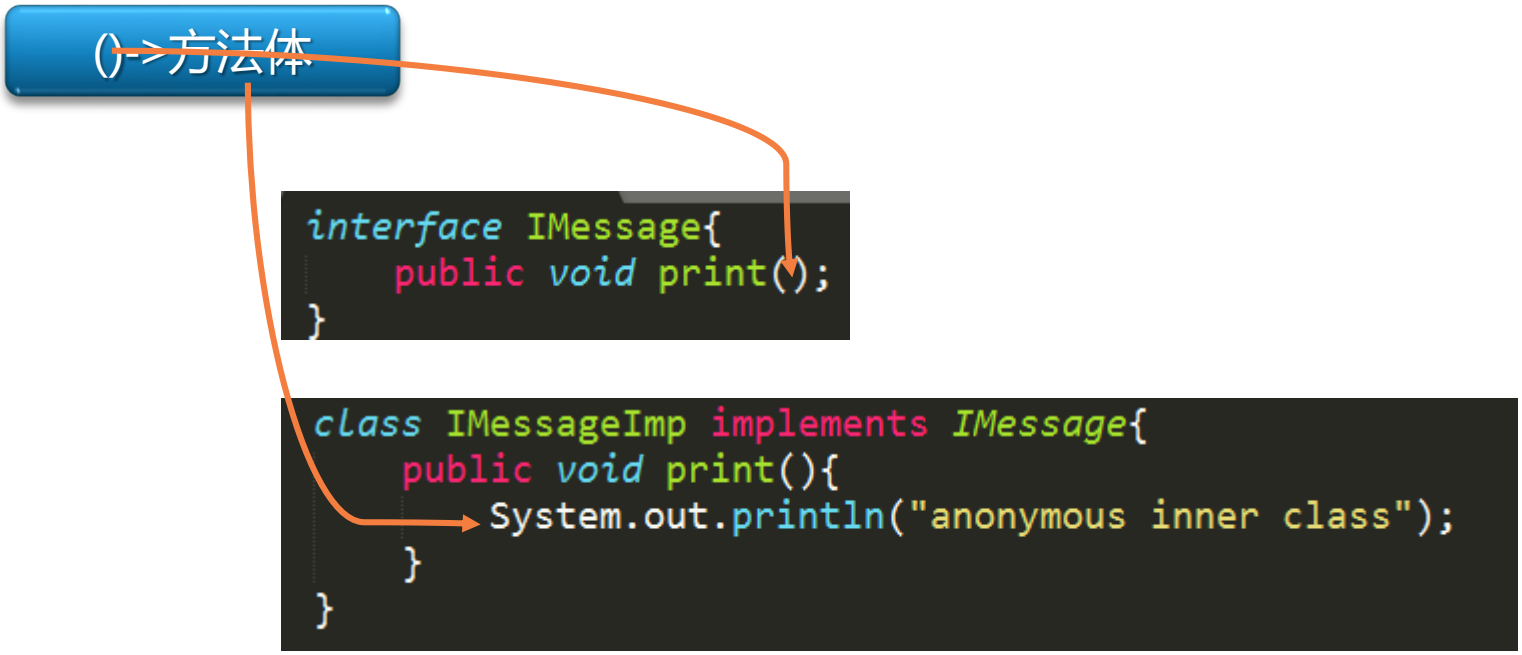
```
interface IMessage{
    public void print();
}
public class InnerTest{
    public static void main(String[] args){


        fun(()->System.out.println("anonymous inner class"));


    }
    public static void fun(IMessage msg){
        msg.print();
    }
}
```

Lambda表达式是用于在单一抽象方法接口环境下的简化定义形式，用于解决匿名内部类的定义复杂问题

- 语法格式


()->方法体
interface IMessage{
public void print();
}
class IMessageImp implements IMessage{
public void print(){
System.out.println("anonymous inner class");
}
}

```java
@FunctionalInterface
interface IMessage{
    public int add(int x,int y);
}
public class InnerTestAdd{
    public static void main(String[] args){
        fun((s1,s2)->{
            return s1+s2;
        });
    }
    public static void fun(IMessage msg){
        System.out.println(msg.add(10,20));
    }
}
```

Lambda表达式明确要求是操作的接口只包含一个抽象方法。可以在接口上使用**@FunctionalInterface**来注解声明，表示这是函数式接口，里面只定义一个抽象方法

```java
@FunctionalInterface
interface IMessage{
    public int add(int ... args);
    static int sum(int ... args){
        int sum=0;
        for(int i:args)
            sum+=i;
        return sum;
    }
}
public class InnerTestArg{
    public static void main(String[] args){
        fun((int ... param)->IMessage.sum(param));
    }
    public static void fun(IMessage msg){
        System.out.println(msg.add(10,20,30));
    }
}
```

7. GUI事件处理中的内部类

- 事件监听与处理有如下形式
  - 外部类实现
    - 事件监听通常属于特定的组件或GUI界面，定义成外部类不利于提高程序的内聚性；
    - 外部类形式的事件监听器不能自由访问创建GUI界面类中的组件；
    - 编程不够简洁

```
class Handler implements ActionListener{
    TextField tf;
    public Handler(TextField tf){
        this.tf=tf;
    }
    public void actionPerformed(ActionEvent e){
        ……
    }
}
```

```
public class MyFrame{
    Frame f=new Frame("test");
    TextField tf=new TextField(40);
    Button b=new Button("push");
    public void init(){
        f.add(b);
        b.addActionListener(new Handler(tf));
        ……
    }
    ……
}
```

- 类本身作为事件监听器类
  - 是早期AWT事件编程中较流行的形式（在chp8/chp9大部分使用这种范例）
  - 可能造成混乱的程序结构，GUI界面的职责主要是完成界面初始化工作，但还需要包含事件处理器方法，从而降低程序的可读性；
  - GUI继承事件适配器会导致GUI界面无法继承其他类

```
public class Dlog implements ActionListener{
        Frame f = new Frame("Dialog owner");
        Dialog d = new Dialog(f,"This is a Dialog Window",false);
        Button b = new Button("Confirm");
        Label l = new Label("welcome");
        public void go(){
                f.setBounds(50,50,230,180);
                f.setLayout(null);
                f.add(b);
                b.setBounds(50,100,50,20);
                d.add(l);
                d.setLayout(null);
                d.setBounds(50,100,250,150);
                l.setBounds(90,70,150,20);
                b.addActionListener(this);
                f.setVisible(true);
        }
        public void actionPerformed(ActionEvent e){
                d.setVisible(true);
```

- 使用内部类实现监听器
  - 可较好的复用该监听器类;
  - 可自由访问外部类的所有GUI组件

```
public class EventQs{
	private Frame f=new Frame("测试");
	private Button ok=new Button("push");
	private Label l=new Label("waiting");
	public void init(){
		ok.addActionListener(new OkListener());
		f.setLayout(new FlowLayout());
		f.add(l);
		f.add(ok);
		f.pack();
		f.setVisible(true);
	}
	class OkListener implements ActionListener{
		public void actionPerformed(ActionEvent e){
			l.setText("be pushed");
		}
	}
	public static void main(String[] args) {
		new EventQs().init();
	}
}
```

- 匿名内部类实现监听器

```
public class EventQs{
	private Frame f=new Frame("测试");
	private Button ok=new Button("push");
	private Label l=new Label("waiting");
	public void init(){
		ok.addActionListener(new ActionListener(){
			public void actionPerformed(ActionEvent e){
				l.setText("be pushed");
			}
		});
		f.setLayout(new FlowLayout());
		f.add(l);
		f.add(ok);
		f.pack();
		f.setVisible(true);
	}
	public static void main(String[] args) {
		new EventQs().init();
	}
}
```