内部类与接口

内部类

根据类间的关系将类创建在不同的包中，同一个包中的类

- 类间耦合相对紧密，同一个包中的内是一个层次，`import mypack.level1.*;`
- 类还可以定义在另一个类的内部
  - 在一个类里面定义的类称为**内部类(InnerClass)或嵌套类**，把外面定义的类称为**外部类(OutClass)或宿主类**

eg.

```java
class Outer{  // 外部类，宿主类
        ...
        class Inner{  // 内部类，嵌套类
                ...
        }
}
```

- 内部类与外部类最大不同：
  - 内部类**实例对象不能单独存在**
  - 内部类对象**必须依赖于外部类对象存在**
  - 外部类声明的时候不能用`private`和`protected`，但内部类却可以声明为以上权限（跟继承关系有点像，就是外部类声明为私有或保护权限后就不能被访问/继承了，所以就没办法再创建内部类，为避免这种情况发生，所以作出的规定）

成员内部类的声明

需要定义外部类对象才能定义内部类对象

- 成员内部类与外部类的关系
  - 外部类私有成员：整个外部类代码块中均可见，内部类也在外部类的代码块内
  - *内部类可以访问外部类的私有成员*
  - *外部类也可访问内部类的私有成员*（对外部类不隐藏）
- 成员内部类与外部类：
  - 不完全是两个类，更像是**一个类**，只是类间有一个分隔
  - 从内部类来看：x，y，z 都是属性，只是`x, y`要用`Outer.this.x(y)`来访问。当外部成员跟内部类成员**不同名**时，`Outer.this.`可省略
  - 从外部类来看：属性`x,y`和类`Inner`平等，Inner内还有一个属性z。这个z只是要用Inner对象来访问，本质上x,y,z都是属性

JAVA

```java
public class Outer {
        public int x = 1;
        private int y = 2;

        public void test(){
                Inner inObj = new Inner();
                System.out.println(inObj.z);
        }
        class Inner{
                public int z = 3;

                public int sum() {
                        return(x + y + z);
                }
        }
        public static void main(String[] args) {
                Outer outer = new Outer();
                Outer.Inner inner = outer.new Inner(); // 注意这里的定义方式，需要先定义了外部类对象才能定义内部类的对象
                outer.test();
                System.out.println("sum: " + inner.sum());
        }
}
```

成员内部类的访问

- 访问成员内部类的方式跟**访问的位置**有关
- 外部类访问：
  - 可直接**通过内部类的类名访问**内部类
- 外部类以外的其他类中：
  - 必须使用**完整类名**`(OuterName.InnerName)`来访问内部类
- *非静态*内部类不能像普通类那样访问，需要**先创建内部类实例**
- *静态*内部类可通过外部类名来访问
- 外部类实例可对于多个内部类实例

JAVA

```java
public class TestInner {
         public static void main(String args[]) {
                 OuterClass outer = new OuterClass();
                 OuterClass.InnerClass inner1 =outer.new InnerClass();
                 inner1.m1();
                 OuterClass.InnerClass inner2 = outer.new InnerClass();   // 另一种创建内部类方法
                 inner2.m2();
         }
}
```

成员内部类基本语法

- 外部类创建内部类对象
  `内部类 对象名 = new 内部类构造器()`
- 外部类以外的其他类或外部类的静态方法创建内部类对象
  `内部类 对象名 = new 外部类构造器().new 内部类构造器()`

JAVA

```
public class OuterClass {
        static String name = "JC, Zhang";
        private int age = 48;
        public void show() {
                InnerClass inner = new InnerClass(); //this.new Inner...
                inner.m1();
                InnerClass.m2(); // 调用内部类的静态方法
        }
        public static void main(String args[]) {
                OuterClass outer = new OuterClass();
                outer.show();
                System.out.println(InnerClass.b);
                InnerClass inner = outer.new InnerClass();
                System.out.println(inner.age);
                inner.m1(); inner.m2();
        }
        public class InnerClass{
                private int age = 10;
                static int b = 20;
                public void m1() {
                        System.out.println("Inner variable:" + age);
                        System.out.println(“Outer variable:” + OuterClass.this.age);
                        System.out.println("Outer static variable:" + name);
                }
                public static void m2() {
                        System.out.println("Inner static variable:" + b);
                        System.out.println("Outer static variable:" + name);
                }
        }
}
```

静态内部类

- 用`static`修饰符的成员内部类
  - 静态内部类可以*直接访问外部类静态成员*
  - 静态内部类可以*定义静态成员*
  - 静态内部类可以*独立外部类对象实例化*，不需要先实例化外部类对象再创建内部类对象

JAVA

```java
public class TestStaticInner {
        public static void main(String[] args) {
                Outer.Inner inner = new Outer.Inner();
                inner.show();
        }
}
        class Outer{
                int t;
                static String dest = "123";
                static class Inner{
                static int x = 10;
                public void show(){
                        //System.out.println("Outer vriable t:" + Outer.this.t);
                        Outer outer = new Outer();
                        System.out.println("Outer vriable t:" + outer.t);
                        System.out.println("Inner vriable dest:" + dest);
                }
        }
}
```

局部内部类

- 定义在**方法代码块内部**的内部类
  - 局部内部类修饰符就是局部变量可用的修饰符
  - 局部内部类不能声明为`static`，局部变量也不能用`static`修饰
  - 局部内部类没有访问权限修饰符
  - 外部类变量与内部类定义的同名的时候，局部内部类可以用`Outer.this.`访问外部类成员
  - 方法外面不能访问方法内定义的局部变量，方法之外也不能访问局部内部类

JAVA

```java
class LocalInner {
        public void test() {
                int x = 20;
                class Inner{
                        public void print(){
                                System.out.println("I am " + x);
                        }
                }
                new Inner().print();
        }
        public static void main(String args[]){
                LocalInner outer = new LocalInner();
                outer.test();
        }
}
```

匿名内部类

局部内部类的特例：匿名内部类

- 特点：
  - 没有显式的类名，通常在创建对象的时候顺便定义
  - 表达式直接生成对象，场景包括
    - 方法参数，引用变量初始化，返回值等
  - **不需要单独声明类名**
  - 这个匿名内部类**只使用一次**
  - 可以**访问外部类所有成员**
  - 必须**继承父类或实现接口**
  - 形式上**用父类或接口构造器创建对象**

JAVA

```
public class TestAnonymous {
        public static void main(String args[]) {
                int array[] = {1,3,3,3,15};
                A instance = new A(array) {  // 匿名内部类
                        public double getMean() { // 重写父类方法
                                int min, max;
                                if (a.length <= 2) return 0;
                                min = a[0]; max = a[0];
                                for (int e: a) {
                                        if(e < min) min = e;
                                        if(e > max) max = e;
                                }
                                double sum = super.getMean() * a.length;
                                return (sum- min - max)/(a.length-2);
                        }
                };
                System.out.println(instance.getMean());
                //创建后实现内部类方法
        }
}
////////////////////////////////////////
class A{
        int[] a;
        public A(int a[]) {this.a = a;}
        public double getMean() {
                int sum = 0;
                for (int i=0; i<a.length; i++) sum += a[i];
                if (a.length<1) return 0;
                else return ((double)sum/a.length). //(double)(sum/a.length);)
        }
}
```

匿名内部类对象的构建

- 匿名内部类没有类名，也没有构造器

- 生成对象调用的是父类或接口构造器
- 编译后生成类名为：SuperClass$1.class，SuperClass$2.class
- 多态：匿名内部类对象赋给父类或接口变量；
- 子类可以重写父类成员，但子类新增成员隐藏

`SuperClass instance = new SuperConstructor(参数列表){\\ 匿名内部类定义}`
调用匿名类：`test(new A(){})`

借助内部类实现多重继承

JAVA

```java
public class MultiParents {
        public static void main(String[] args) {
                Amphibious son = new Amphibious();
                son.carry();
                son.dive();
        }
}
class Vehicle{
        public void run() {
                System.out.println("I can run in land");
        }
}
class Boat{
        public void sail(){
                System.out.println("I can sial in water");
        }
}
```

通过内部类+依赖关系实现多重继承：

JAVA

```
class Amphibious{
        class SubVehicle extends Vehicle{
                public void run(){
                        super.run();
                        System.out.println("I can carry somthing");
                }
         }
         class SubBoat extends Boat{
                public void sail(){
                        super.sail();
                        System.out.println("I can swim under water");
                }
         }
         public void carry() {
                new SubVehicle().run();
         }
         public void dive() {
                new SubBoat().sail();
         }
  }
```

接口

接口是系统各对象协作的窗口

- 在Java语言中，接口有两种意思：
  - 一是指概念性的接口，即指类对外提供的所有服务。类的所有能被其他程序访问的方法构成了类的接口
  - 二是指用`interface`关键字定义的实实在在的接口，也称为接口类型。它用于明确地描述系统对外提供的所有服务，它能够更加清晰地把系统的实现细节与接口分离

面向设计方法强调对象协作

Java中的接口

Java语言中，指用`interface`关键字定义的实实在在的接口，也称为接口类型

接口定义非常简洁

- 接口是隐式抽象的
  - 当声明一个接口时，不必使用`abstract`关键字
  - 系统会自动像抽象类那样处理接口，**不能创建对象**
- 接口中每个方法也是隐式抽象的
  - 声明时不用`abstract`关键字
  - 系统自动将接口中方法视为抽象方法，**没有实现体**
- 接口中的方法都是公有的
  - 实现接口的类中，方法重写必须加上`public`，否则重写就将权限由`public`缩小到`default`，编译器报错

接口的语法：成员变量

- 接口可看作是由**常量和抽象方法**组成的**特殊的抽象类**
  - 接口中的变量默认都是`public`、`static`、`final`类型的，必须被显式初始化。并且接口中智能包含`public`、`static`、`final`类型的成员变量
  - 可以用**`接口名.变量名`**直接调用，如`A.v3`
  - 接口中`int v3`相当于`public static final int v3 = 3`

JAVA

```
public interface A{
        int v1; //编译出错，v1变量被看作静态常量，必须被显式初始化
        protected int v2=0; //编译出错，v2变量必须是public类型
        int v3=3; //合法，v3变量默认为public、static、final类型
}
```

接口的语法：成员方法

- 接口的实现：
  - 接口定义的仅仅是**实现某一个特定功能**的一组功能的对外接口和规范，功能的真正实现要由实现接口的各个类来定义接口中各抽象方法的方法体
  - 一般来说，接口中的方法默认`abstract`修饰，没有方法体的抽象方法
- 允许在接口中定义带有实现体的默认方法
  - 默认方法用`default`关键字来声明，拥有默认的实现，此处`default`不是访问权限。接口中的方法默认是`public`类型的，并且必须是`public`类型的
  - 有了`default`方法，后面实现类中**可不用每个方法都重写**
  - 接口中的静态方法只能**用类名访问**，不能用变量访问

JAVA

```
public interface MyIFC {
        default void method1(){ //声明一个默认方法
                System.out.println("default method1");
        }
        static void method2(){ //声明一个静态方法
                System.out.println("static method2");
        }
        void method3(); //声明一个抽象方法
}
public class MyImpl implements MyIFC{
        public void method3(){...} //必须实现抽象方法
        public static void main(String[] args){
                MyIFC a=new MyImpl();
                MyIFC.method2(); // 必须这么访问静态方法
                a.method1(); // 缺省方法可不用实现
                a.method3();
        }
}
```

- 接口的实现，是指一个类类**实现其中的抽象方法**

    - 如果一个类没有实现接口全部的抽象方法，这个类必须声明为抽象类，不能创建对象，有待子类进一步继承，重写抽象方法
- 接口**没有构造器，无法实例化**

接口中的省略修饰符

- 接口中带方法实现的方法是`static`或`default`修饰的方法
- 其他方法必须是`public abstract`修饰的没有实现的方法

```
public interface collection{
   int MAX_NUM = 100;
   void add(Object obj)
   int curentCount();
   abstract Object getTop();
   public void remove(int index);
}
```

```
public interface collection{
   public static final int MAX_NUM = 100;
   public abstract void add(Object obj)
   public abstract int curentCount();
   public abstract Object getTop();
   public abstract void remove(int index);
}
```

◈ 接口中带方法实现的方法是static或default修饰的方法

◈ 其它方法必须是public abstract修饰的没有实现的方法

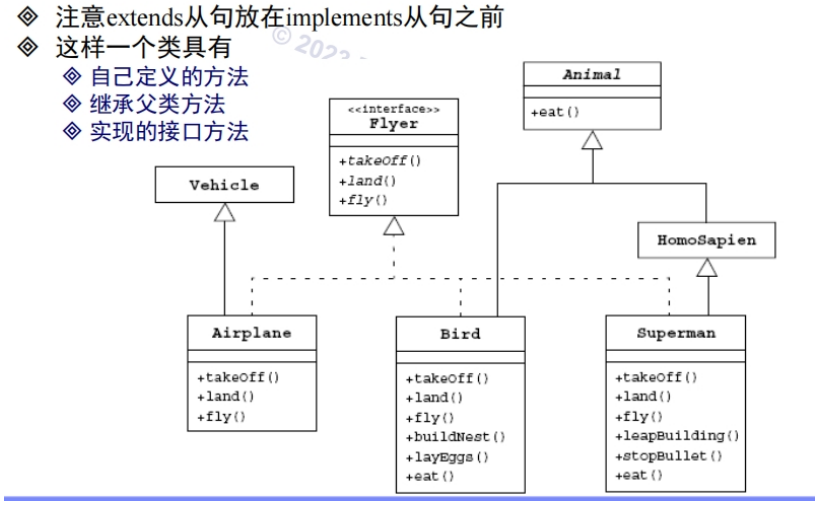
```
public interface A{
      int var;                    //编译出错，public static final静态变量没有初始化
      void method1(){}            //编译出错，  public abstract方法不能有实现
      protected void method2();   //编译出错，方法必须是public
      static void method3();      //编译出错，接口中不能有静态方法没有实现的方法
}
```

理解接口

- 继承在前，实现在后
- 接口是一个纯的抽象类
- 继承讲的是**泛化**关系，接口讲的是**实现**关系
- 接口仅仅声明了一个桥梁，并没有具体实现；一个具体类实现接口，就是说它实现了接口中**所有声明**的方法
- 接口使得不同层次，不相关的类具有相同的行为，为多重继承问题提供了一种解决方案
    - 一个类只能继承一个父类，但可以实现多个接口
    - 接口可以继承

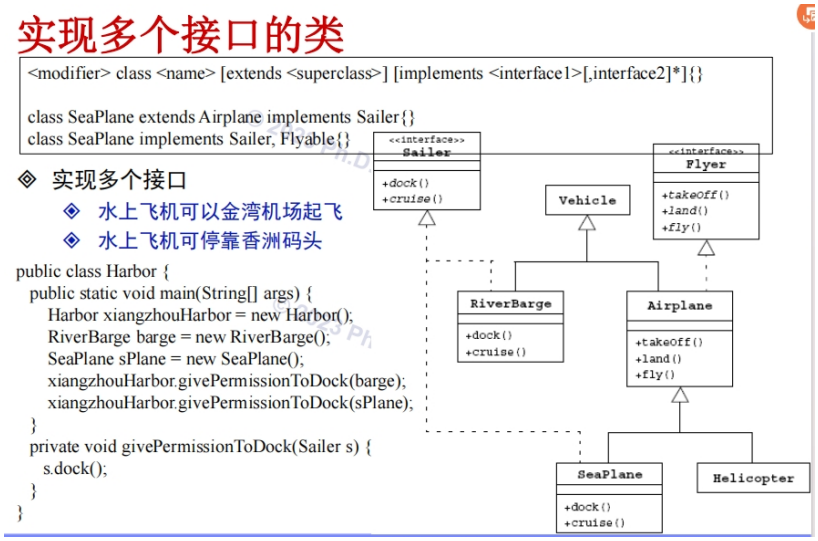
```java
interface BaseIFC{
        void method1();
        void method2();
}
interface SubIFC extends BaseIFC{
        void method3();
}
```

可以既有继承又有实现接口



实现多个接口的类

`<modifier> class <name> [extends <superclass>] [implements<interface1>[,interface2]*]{}`



抽象类与接口相似

> 接口和继承都是面向对象编程中的重要概念。它们各自有不同的作用，但也可以相互配合使用。
> 接口的主要作用是**定义一种规范或者是一种协议**。它规定了一个类应该有哪些方法，但是这些方法的具体实现由实现这个接口的类来完成。接口可以使得不同的类有相同的行为，提高了代码的复用性和可替换性。
> 而继承则是**从一个已有的类创建新类的一种方式**，子类可以继承父类的属性和方法，并且可以在此基础上进行扩展和修改。继承可以使得代码更加简洁，避免了重复的代码。
> 接口和继承的关系主要体现在接口继承和类继承上。接口可以继承一个或多个其他接口，从而扩展或组合多个接口的行为。而类可以实现一个或多个接口，通过实现接口来承诺遵循接口的规范。同时，类也可以继承其他类，并可以通过重写方法来改变继承的行为。

- 共同点：
  - 都**不能被实例化**，不可用`new`调用构造器，但可以用来声明变量
  - 都可以包含抽象方法，以及包含具体实现的方法
  - 在语义上，都位于系统的抽象层，需要其他类来进一步提供实现细节
- 区别：
  - 抽象类：强 `is a`关系
  - 接口：弱`is a` 关系，强`kind of`关系

- 接口中的成员变量和方法只能是`public`类型的，而抽象类中的成员变量和方法可以处于*各种访问级别*。抽象类比接口包含了更多的实现细节
- 接口中的成员变量只能是`public、static和final`类型的，而在抽象类中可以定义*各种类型的实例变量和静态变量*
- 一个类只能继承一个直接的父类，这个父类有可能是抽象类；但一个类可以实现多个接口

JAVA

```
public interface USBInterface{ /* USB接口 */
        public void transportData();
}
public abstract class Printer implements USBInterface{ /* 打印机 */
        public abstract void print();
}
public class InkjetPrinter extends Printer{...} /* 喷墨打印机 */
public class LaserPrinter extends Printer{...} /* 激光打印机 */
public abstract class Camera { /* 照相机 */
        public abstract void takePhoto();
}
/* 数码照相机 */
public class DigitalCamera extends Camera implements USBInterface{...}
public class FilmCamera extends Camera{...} /* 胶片照相机 */
```

接口与多态

- 允许将一个引用变量声明为接口类型，而实际上引用实现接口的类的实例
- `instanceof`操作符左边的操作元是一个引用类型，右边的操作元元是一个**类名或接口名**

接口Comparable

```
package java.lang;

public interface Comparable<E> {
    public int compareTo(E o);
}
```

◈ 8种基本类型都有一个包装类

◈ 每个包装类都重写了object类中的**toString, equals, hashCode方法**

◈ 数字类和Character类都实现了Comparable接口，都实现了compareTo方法

```
public class Integer extends Number
    implements Comparable<Integer> {
  // class body omitted

  @Override
  public int compareTo(Integer o) {
    // Implementation omitted
  }
}
```

```
public class BigInteger extends Number
    implements Comparable<BigInteger> {
  // class body omitted

  @Override
  public int compareTo(BigInteger o) {
    // Implementation omitted
  }
}
```

```
public class String extends Object
    implements Comparable<String> {
  // class body omitted

  @Override
  public int compareTo(String o) {
    // Implementation omitted
  }
}
```

```
public class Date extends Object
    implements Comparable<Date> {
  // class body omitted

  @Override
  public int compareTo(Date o) {
    // Implementation omitted
  }
}
```

接口Comparable（续）

```
1 System.out.println(new Integer(3).compareTo(new Integer(5)));
2 System.out.println("ABC".compareTo("ABE"));
3 java.util.Date date1 = new java.util.Date(2013, 1, 1);
4 java.util.Date date2 = new java.util.Date(2012, 1, 1);
5 System.out.println(date1.compareTo(date2));
```

n: **Integer** object
s: **String** object
d: **Date** object
下面语句全部为 **true**

```
n instanceof Integer
n instanceof Object
n instanceof Comparable
```

```
s instanceof String
s instanceof Object
s instanceof Comparable
```

```
d instanceof java.util.Date
d instanceof Object
d instanceof Comparable
```