第十二章 抽象类和接口

1. 抽象类

1.1 抽象类的概述

1. 具体类
   a. 是对现实世界一种实体的抽象定义
   b. 比如：鸟，猫和狗等是一种实体
2. 抽象类
   a. 是对现实世界一种类型，多种实体的抽象定义
   b. 比如：宠物就是一种类型，多种实体
   c. 抽象类的抽象级别，比具体类更高
   d. 一个类的具体方法，如果在不清楚的情况下，可以把这个方法可以声明为抽象的，那么这个类必须抽象类，抽象方法，交给子类来实现

1.2 抽象类的特点

代码块

```
package com.powernode.abstract01;

/**
 * 1.定义抽象类的关键字是abstract
 * 2.抽象类中可以包含非抽象方法
 * 3.抽象类中可以包含实例|静态变量
 * 4.抽象类可以包含构造器
 * 5.抽象类不可以直接创建对象（通过子类来创建）
 * 6.抽象类适用于多态
 */
abstract class Pet{
    /* public abstract void eat();*/
     private String name;
     //public static String city;

     public Pet(String name) {
         this.name = name;
     }

```

```
    public String getName() {
        return name;
    }
}
class Cat extends Pet{
    private String hireColor;
    Cat(String name,String hireColor){
        super(name);
    }
}
public class Test {
    public static void main(String[] args) {
        //抽象类适用于多态
        Pet pet = new Cat("喵喵","red");
    }
}
```

1.3 抽象方法

代码块

```
package com.powernode.abstract02;
 abstract  class Pet{
    /**
     * 1.什么样的方法，应该声明为抽象的呢?
     *    1.一个类的具体行为不清楚，我们可以把这个方法声明为抽象的
     *    2.比如：宠物，都有吃饭和睡觉的方法，但是我们不确定具体的行为，可以声明为抽象的
     * 2.抽象方法的特点
     *    1.使用abstract去修饰方法
     *    2.没有方法体
     *    3.抽象方法必须存在于抽象类中
     * 3.抽象的继承
     *    1.子类如果不是抽象类，必须重写所有的抽象方法
     *    2.子类如果是抽象类，不强制重写抽象方法
     *
     */
    public abstract void eat();

    public abstract void sleep();
}
class Cat extends Pet{

    @Override
    public void eat() {
        System.out.println("Cat.eat");
    }
```

```
26
27      @Override
28      public void sleep() {
29          System.out.println("Cat.sleep");
30      }
31  }
32  public class Test {
33      public static void main(String[] args) {
34          Pet pet = new Cat();
35          pet.eat();
36          pet.sleep();
37      }
38  }
```

1.4 抽象类的应用场景（模版方法）

- 抽象类
- 抽象方法
- final
- 经典案例（模版方法）

代码块

```
1   package com.powernode.abstract03;
2
3
4   import java.io.File;
5
6   //文件处理类
7   abstract class FileProcessor{
8       //定义一个文件处理流程（模版方法）
9       public final void processFile(){
10          //1.打开文件
11          openFile();
12          //2.处理文件
13          processContents();
14          //3.关闭文件
15          closeFile();
16      }
17
18      public abstract void processContents();
19
20
21      public void openFile(){
22          System.out.println("打开文件");
```

```
23      }
24      public void closeFile() {
25          System.out.println("关闭文件");
26      }
27  }
28  //统计文件行数
29  class LineCounter extends FileProcessor{
30
31    /*  @Override
32      public void processFile() {
33          System.out.println("LineCounter.processFile");
34      }*/
35
36      @Override
37      public void processContents() {
38          System.out.println("只专注于统计行数");
39      }
40  }
41  //统计文件字数
42  class WordCounter extends FileProcessor{
43
44      @Override
45      public void processContents() {
46          System.out.println("只专注于统计文件字数");
47      }
48  }
49
50
51  public class Test {
52      public static void main(String[] args) {
53          FileProcessor fileLineCounter = new LineCounter();
54          fileLineCounter.processFile();
55
56          FileProcessor fileWordCounter = new WordCounter();
57          fileWordCounter.processFile();
58      }
59  }
```

1.5 模版方法的深度剖析

1. 设计原理
   a. 抽象类：模版方法设计模式中，抽象类定义了一个流程（定义流程的方法称为模版方法）方法和其他方法
   b. 模版方法：使用final修饰一下模版方法，确保该方法不可以被重写

```
代码块
1  //定义一个文件处理流程（模版方法）
2  public final void processFile(){
3      //1.打开文件
4      openFile();
5      //2.处理文件
6      processContents();
7      //3.关闭文件
8      closeFile();
9  }
```

c. 业务方法：

i. 处理具体业务的方法（processContents）声明为抽象的，该方法在模版方法的流程中，不做具体的实现，是交给子类来实现的

ii. 其他公用的方法 openFile()和closeFile()

2. 模版方法的

a. 优点

i. 代码复用率高

1. 这个案例中， openFile()和closeFile()，是通用功能，这些方法放入了抽象类的流程方法中（ processFile()）中
2. 无论是LineCounter还是WordCounter都不需要重复编写 openFile()和closeFile()
3. 未来如果需要添加其他的文件处理功能，都可以复用openFile()和closeFile()，减少了代码的冗余，提高复用率

ii. 结构清晰易于维护

1. 模版方法明确的把处理流程（processFile）和具体的实现细节（processContents）分开，这种实现方式，使代码的结构更加清晰
2. 对于开发人员来说，看一下processFile模版方法，就可以快速的了解文件的处理流程，而看一下LineCounter和WordCounter的processContents方法就可以找到具体的处理细节

iii. 易于程序的扩展

1. 添加一个标点符号统计的功能，只需要创建一个类继承FileProcessor，并重写processContents方法即可
2. 新的类就直接加入了处理流程中，不需要修改任何其他类的代码，使代码具有更好的扩展性，可以适应不断变化的需求，遵循了软件开发的开闭原则

iv. 注：开闭原则

1. 开闭原则（OCP:Open/Closed Principle）：是面向对象的重要原则，它是指软件的实体（方法，类，模块）应该对扩展开放，对修改关闭

2. 在软件的开发过程中，要扩展功能，不修改已有稳定的代码，而是通过其他方式来扩展（创建一个类，继承FileProcessor），这样我们开发的软件在**提高扩展的同时，更增强了程序的稳定性**

b. 缺点：

i. 灵活性受限
模板方法中的骨架流程被父类固定，子类只能通过重写特定方法来扩展，难以修改整体流程结构。如果业务需求需要改变流程的核心步骤，可能需要修改父类，违反 “开闭原则”。

ii. 继承带来的耦合性
子类与父类的依赖关系较强，父类的任何改动（如抽象方法的增减或签名变化）都可能影响所有子类，增加了维护成本。

iii. **可能导致类数量膨胀**
每一种业务变体都需要创建一个子类来实现，当变体较多时，会产生大量子类，增加系统复杂度。

iv. 学习成本
新开发者需要理解父类的整体逻辑才能正确实现子类，对于复杂的模板方法，可能需要花费更多时间梳理流程与扩展点的关系。

v. 逆扩展性差
如果父类后续需要新增抽象方法，所有已有的子类都必须实现该方法，否则会导致编译错误，对已有代码的侵入性较强。

作业

1. **练习题(理解静态变量属于类，实例变量属于对象)**

a. 在Studnet类中声明私有的静态属性currentStudnetNO，初始值为0，作为班级学生的起始值。

b. 声明公有的静态方法getCurrentStudnetNO ，作为生成学生唯一序列号的方法。每调用一次，将currentStudnetNO增加1，并作为返回值。

c. 在Test类的main方法中，分两次调用getCurrentStudnetNO 方法，获取序列号并打印输出，

d. 创建Student对象，使用student对象再次调用getCurrentStudnetNO 获取并打印。

e. 把 currentStudnetNO，getCurrentStudnetNO 修改实例变量和实例方法

f. 把main方法内之前写的全注释掉

g. 在main方法中，创建Student 对象 stu ，使用stu对象分两次调用getCurrentStudnetNO 方法，获取序列号并打印输出

h. 继续创建Student 对象 newstu ，使用newstu再次调用getCurrentStudnetNO 方法，获取序列号并打印输出

2. **单例设计模式练习**
   a. 总经理（CEO）只能创建一个对象，使用单例设计模式解决
3. **实例方法访问，静态常量**
   a. 在MyMath类中声明公有静态常量PI ，值为3.14 。
   b. 设计一个类圆-Circle， 至少包含实例变量半径（radius ），和实例方法获得面积getArea()。
   c. 在Test类的main方法中计算半径为2.6的圆面积。
4. **抽象类和抽象方法**
   a. 将Frock类声明为抽象类，在类中声明抽象方法calcArea(double size)方法，用来计算衣服的布料面积。
   b. 通过编写代码来验证抽象类中是否可包含实例属性color,具体方法setColor/getColor和**构造器**。
   c. 定义一个Shirt类，声明一个price（价格）实例变量，创建构造器为price赋值，并提供get和set方法
   d. Shirt类继承Frock类，重写 calcArea(double size)方法，用来计算衬衣所需的布料面积（size*1.3）。
   e. 在TestShirt类的main方法中：
      i. 使用本态引用创建Shirt对象，并调用calcArea方法，打印计算结果。
      ii. 使用Frock 多态引用创建Shirt对象，并调用calcArea方法，打印计算结果。
      iii. 备注：Frock（女装）,Shirt(衬衫)

2. 接口

2.1 接口的概述

1. 具体类
   a. 对现实世界**一种实体**的抽象定义
   b. 比如：鸟，猫和狗都是一种实体
2. 抽象类
   a. 对现实世界**一种类型，多种实体**的抽象定义
   b. 比如：宠物，就是一种类型多种实体
3. 接口

- 对现实世界**多种类型，多种实体**的抽象定义
- 比如：飞行物，宠物可以飞，飞机也可以飞

a. 接口的特点

i. 特殊的抽象类

1. 属性：所有的属性都是 **public static final**的
2. 方法：所有的方法都是抽象的（jdk8之前）
3. 构造器：接口中不可以包含构造器

ii. 接口可以多实现（类只能单继承，接口可以多实现）【理解：继承是找亲爹，实现认干爹】

iii. 声明接口的关键字是interface，声明类的关键字是class

b. 什么情况下使用接口

i. 在实际的开发过程中，对父类进行抽象定义时，如果这个父类的方法都不清楚具体实现

1. 可以把父类定义为抽象类
2. 也可以把父类定义为接口
3. 如果它的所有属性都是 public static final的，最好定义为接口

2.2 接口中的属性

代码块

```
package com.powernode.interface04;
//飞行物
interface Flyer{
    /**
     * 1.接口中的属性都是：public static final的
     * 2.所以 public static final 可以省略
     */
   // public static final int SPEED = 120;
    int SPEED = 120;

}
public class Test {
    public static void main(String[] args) {
        //说明是static
        System.out.println(Flyer.SPEED);
        //Flyer.SPEED = 130;Cannot assign a value to final variable 'SPEED' 说明是final的
    }
}
```

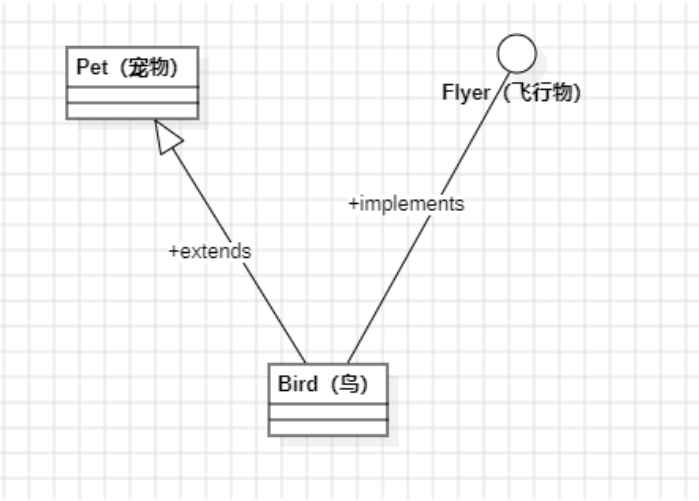
2.3 接口中的方法

代码块

```
package com.powernode.interface05;

public interface Flyer {
    /**
     * 1.在JDK8之前，接口是用于定义规范的，不做具体实现
     * 2.public abstract 可以省略不写，默认就是public abstract
     * 3.在接口中不可以包含实例块{} 和静态块 static{}
     *    1.实例块（{}）用于初始化实例对象时执行代码（属于对象的初始化逻辑），
     *    2.静态块（static {}）用于初始化类的静态资源（属于类的初始化逻辑），
     *      两者都属于具体实现代码（接口的核心定位：行为规范而非实现）。
     * 4.接口中不可以包含构造器
     */
    // {}
    //static {}
    public abstract void fly();
    void land();
}
```

2.4 在实际工作中接口的几种形式

2.4.1 具体类继承抽象类并实现接口



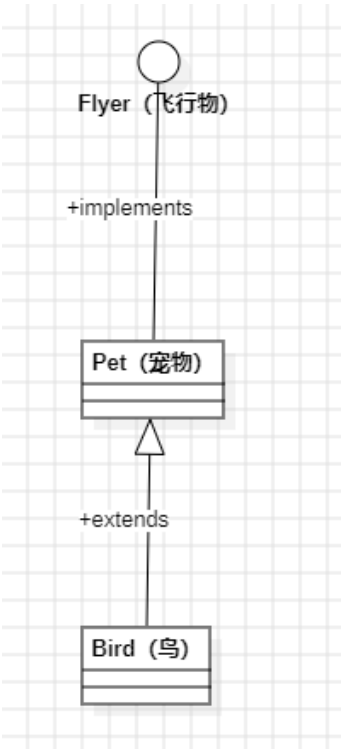
代码块

```
package com.powernode.interface06;
//飞行物（接口）
public interface Flyer {
    //起飞
    void takeOff();
    //飞行
```

```
    void fly();
    //着陆
    void land();
}
//宠物（抽象类）
abstract class Pet{
    private String name;
    public abstract void eat();
    public abstract void sleep();
}
//具体类(Bird)：Bird继承了Pet，并实现了Flyer
class Bird extends Pet implements Flyer{

    @Override
    public void takeOff() {
        System.out.println("Bird.takeOff");
    }

    @Override
    public void fly() {
        System.out.println("Bird.fly");
    }

    @Override
    public void land() {
        System.out.println("Bird.land");
    }

    @Override
    public void eat() {
        System.out.println("Bird.eat");
    }

    @Override
    public void sleep() {
        System.out.println("Bird.sleep");
    }
}
class Test{
    public static void main(String[] args) {
        //1.本太调用
        Bird bird = new Bird();
        //可以调用5个方法
        bird.fly();
        bird.eat();

        //2.抽象类多态
```

```
54          Pet pet = new Bird();
55          //调用自己
56          pet.eat();
57          pet.sleep();
58
59          //3.接口多态
60          Flyer flyer = new Bird();
61          //调用直接的
62          flyer.fly();
63          flyer.land();
64          flyer.takeOff();
65
66          //多态指的方法多态：编译看父类，运行看子类
67
68      }
69  }
```

2.4.2 具体类继承抽象类，抽象类实现接口



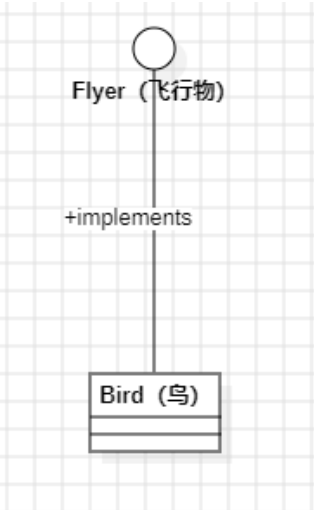
代码块

```
1   package com.powernode.interface07;
2   //飞行物（接口）
```

```java
public interface Flyer {
    //起飞
    void takeOff();
    //飞行
    void fly();
    //着陆
    void land();
}
//宠物（抽象类）
abstract class Pet implements Flyer{
    private String name;
    public abstract void eat();
    public abstract void sleep();
}
//具体类(Bird)：Bird继承了Pet
class Bird extends Pet {

    @Override
    public void eat() {
        System.out.println("Bird.eat");
    }

    @Override
    public void sleep() {
        System.out.println("Bird.sleep");
    }

    @Override
    public void takeOff() {
        System.out.println("Bird.takeOff");
    }

    @Override
    public void fly() {
        System.out.println("Bird.fly");
    }

    @Override
    public void land() {
        System.out.println("Bird.land");
    }
}
class Test{
    public static void main(String[] args) {
        //1.本太调用
        Bird bird = new Bird();
        //可以调用5个方法
```

```
50          bird.fly();
51          bird.eat();
52
53          //2.抽象类多态
54          Pet pet = new Bird();
55          //调用自己 + 干爹的  = 5个
56          pet.eat();
57          pet.sleep();
58          pet.fly();
59
60          //3.接口多态
61          Flyer flyer = new Bird();
62          //调用直接的
63          flyer.fly();
64          flyer.land();
65          flyer.takeOff();
66
67          //多态指的方法多态：编译看父类，运行看子类
68
69      }
70  }
```

2.4.3 具体类直接实现接口

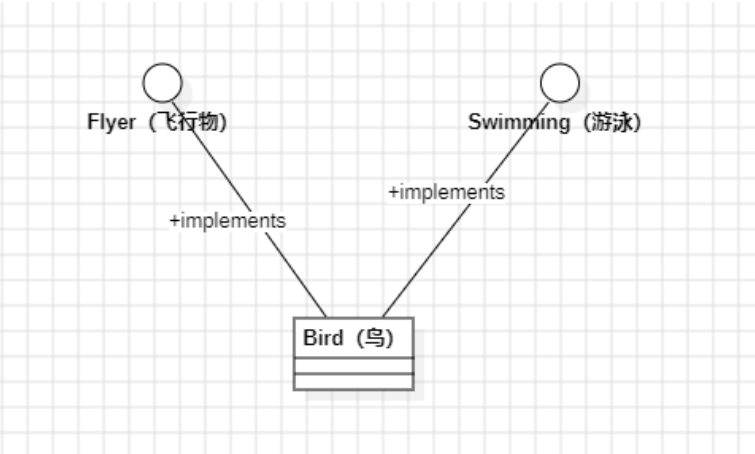


代码块

```
1   package com.powernode.interface08;
2
3   public interface Flyer {
4       void takeOff();
5       void fly();
6       void land();
```

```
}
class Bird implements Flyer{

    @Override
    public void takeOff() {
        System.out.println("Bird.takeOff");
    }

    @Override
    public void fly() {
        System.out.println("Bird.fly");
    }

    @Override
    public void land() {
        System.out.println("Bird.land");
    }
}
class Test{
    public static void main(String[] args) {
        Flyer flyer = new Bird();
        flyer.fly();
        flyer.land();
        flyer.takeOff();
    }
}
```

2.4.4 具体类实现多个接口

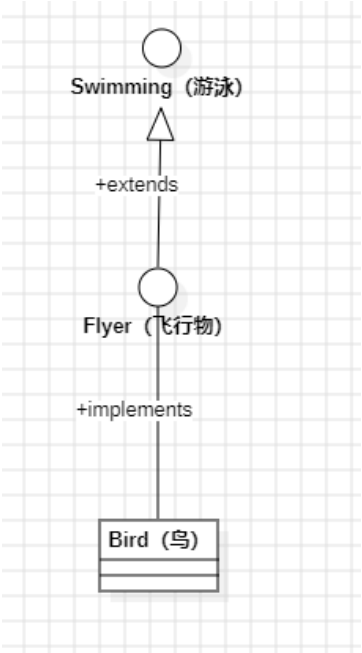


代码块

```
package com.powernode.interface09;

```

```java
import javax.print.attribute.standard.PagesPerMinute;

public interface Flyer {
    void takeOff();
    void land();
    void fly();
}
interface Swimming{
    void swim();
}
//接口多实现，中间使用逗号隔开
class Bird implements Flyer,Swimming{

    @Override
    public void takeOff() {
        System.out.println("Bird.takeOff");
    }

    @Override
    public void land() {
        System.out.println("Bird.land");
    }

    @Override
    public void fly() {
        System.out.println("Bird.fly");
    }

    @Override
    public void swim() {
        System.out.println("Bird.swim");
    }
}
class Test{
    public static void main(String[] args) {
        Flyer flyer = new Bird();
        flyer.land();
        flyer.fly();
        flyer.takeOff();
        //两个干爹之间没有关系
        Swimming swimming = new Bird();
        swimming.swim();
    }
}
```

2.4.5 接口继承接口



代码块

```
package com.powernode.interface10;

public interface Swimming {
    void swim();
}
interface Flyer extends Swimming{
    void takeOff();
    void fly();
    void land();
}
class Bird implements Flyer{

    @Override
    public void takeOff() {
        System.out.println("Bird.takeOff");
    }

    @Override
    public void fly() {
        System.out.println("Bird.fly");
    }

    @Override
    public void land() {
        System.out.println("Bird.land");
    }
```

```
27
28      @Override
29      public void swim() {
30          System.out.println("Bird.swim");
31      }
32  }
33  class Test{
34      public static void main(String[] args) {
35          Bird bird = new Bird();
36          bird.fly();
37          bird.swim();
38
39          Flyer flyer = new Bird();
40          flyer.fly();
41          flyer.swim();
42
43          Swimming swimming = new Bird();
44          swimming.swim();
45      }
46  }
```

2.5 接口的新特性（jdk8及之后）

2.5.1 接口中定义非抽象方法

代码块

```
1   package com.powernode.interface11;
2
3   public interface Flyer {
4       /**
5        * 1.默认方法
6        *    1.默认方法的作用
7        *    2.解决接口升级问题(新增方法)
8        *      1.解决方案：在接口中添加抽象方法，子类重写抽象方法，工程量比较大
9        *      2.解决方案：使用默认方法，只需要在接口中添加具体方法，对子类不做修改，对子类
    也做了扩展
10       *    3.默认方法的限制：所有的子类都是一样的
11       *
12       * 2.私有方法（jdk9）
13       *   1.私有方法主要为默认方法提供服务的
```

```
14          *   2.在很多的默认方法中，有代码相同的片段处理同样的功能，可以把这个代码片段抽取出来，放到一个私有方法中，
15          *   供其他默认的方法调用
16          *   3.减少了代码的冗余，提高程序的内聚性
17          *
18          */
19      default void method1(){
20          method();
21          System.out.println("-------处理其他业务-------");
22      }
23      default void method2(){
24          method();
25          System.out.println("-------处理其他业务-------");
26      }
27      private void method(){
28          for (int i = 0; i < 3; i++) {
29              for (int j = 0; j < 5; j++) {
30                  System.out.print("*");
31              }
32              System.out.println();
33          }
34      }
35
36      /*default void method1(){
37          for (int i = 0; i < 3; i++) {
38              for (int j = 0; j < 5; j++) {
39                  System.out.print("*");
40              }
41              System.out.println();
42          }
43          System.out.println("-------处理其他业务-------");
44      }
45      default void method2(){
46          for (int i = 0; i < 3; i++) {
47              for (int j = 0; j < 5; j++) {
48                  System.out.print("*");
49              }
50              System.out.println();
51          }
52          System.out.println("-------处理其他业务-------");
53      }*/
54      /**
55       * 3.静态方法
56       *    1.类似于默认方法
57       *    2.用于完成通用的功能
58       */
59      public static void land(){
```

```
60          System.out.println("Flyer.land");
61      }
62      /**
63       * 4.私有静态方法（jdk9）
64       *    1.私有静态方法主要为静态方法提供服务的
65       *    2.在很多的静态方法中，有代码相同的片段处理同样的功能，可以把这个代码片段抽取出来，放到一个私有静态方法中，
66       * 供其他静态的方法调用
67       *    3.减少了代码的冗余，提高程序的内聚性
68       */
69      private static int max(int x , int y){
70          return x > y ? x :y;
71      }
72
73  }
74  class Dog implements Flyer{}
75  class Cat implements Flyer{}
```

2.5.2 接口中默认方法的重写问题

代码块

```
1   package com.powernode.interface12;
2
3   public interface Flyer {
4       //默认方法
5       public default void land(){
6           System.out.println("Flyer.land");
7       }
8   }
9   class Bird implements Flyer{
10      //默认方法重写: 去掉default
11      @Override
12      public void land() {
13          System.out.println("Bird.land");
14      }
15  }
16  class Test{
17      public static void main(String[] args) {
18          Flyer flyer = new Bird();
19          flyer.land();
20      }
21  }
```

2.5.3 接口多实现，多接口默认方法相同

代码块

```
package com.powernode.interface13;

public interface Flyer {
    default void land(){
        System.out.println("Flyer.land");
    }
    default void swim(){
        System.out.println("Flyer.swim");
    }
}
interface Swimming{
    default void swim(){
        System.out.println("Swimming.swim");
    }
}
class Bird implements Flyer,Swimming{

    /**
     * 如果多个接口有相同的默认方法时，子类必须重写一个
     */
    @Override
    public void swim() {
        Flyer.super.swim();//调用Flyer的默认方法
        Swimming.super.swim();//调用Swimming的默认方法
    }
}
class Test{
    public static void main(String[] args) {
        Bird bird = new Bird();
       /* bird.swim()*/
    }
}
```