目录

抽象类介绍

什么是抽象？

百度给出的解释是：从具体事物抽出、概括出它们共同的方面、本质属性与关系等，而将个别的、非本质的方面、属性与关系舍弃，这种思维过程，称为抽象。

这句话概括了抽象的概念，而在Java中，你可以只给出方法的定义不去实现方法的具体事物，由子类去根据具体需求来具体实现。

这种只给出方法定义而不具体实现的方法被称为抽象方法，抽象方法是没有方法体的，在代码的表达上就是没有“{}”。

包含一个或多个抽象方法的类也必须被声明为抽象类。

使用abstract修饰符来表示抽象方法以及抽象类。

```
//有抽象方法的类也必须被声明为abstract
public abstract class Test1 {

        //抽象方法，不能有“{}”
        public abstract void f();

}
```

抽象类除了包含抽象方法外，还可以包含具体的变量和具体的方法。类即使不包含抽象方法，也可以被声明为抽象类，防止被实例化。

抽象类不能被实例化，也就是不能使用new关键字来得到一个抽象类的实例，抽象方法必须在子类中被实现。

```
//有抽象方法的类也必须被声明为abstract
public class Test1 {

        public static void main(String[] args) {
                Teacher teacher=new Teacher("教师");
                teacher.work();

                Driver driver=new Driver("驾驶员");
                driver.work();
        }

}
//一个抽象类
abstract class People{
        //抽象方法
        public abstract void work();
}
class Teacher extends People{
        private String work;
        public Teacher(String work) {
                this.work=work;
        }
        @Override
        public void work() {
                System.out.println("我的职业是"+this.work);
        }

}
class Driver extends People{
        private String work;
        public Driver(String work) {
                this.work=work;
        }
        @Override
        public void work() {
                System.out.println("我的职业是"+this.work);
        }

}
```

> 运行结果： 我的职业是教师 我的职业是驾驶员 几点说明:

抽象类不能直接使用，需要子类去实现抽象类，然后使用其子类的实例。然而可以创建一个变量，其类型也是一个抽象类，并让他指向具体子类的一个实例，也就是可以使用抽象类来充当形参，实际实现类为实参，也就是多态的应用。

```
People people=new Teacher("教师");
people.work();
```

不能有抽象构造方法或抽象静态方法。

如果非要使用new关键在来创建一个抽象类的实例的话，可以这样：

```
People people=new People() {
        @Override
        public void work() {
            //实现这个方法的具体功能
        }
                };
```

个人不推荐这种方法，代码读起来有点累。

在下列情况下，一个类将成为抽象类：

```
当一个类的一个或多个方法是抽象方法时。
当类是一个抽象类的子类，并且不能实现父类的所有抽象方法时。
当一个类实现一个接口，并且不能实现接口的所有抽象方法时。
注意：
上面说的是这些情况下一个类将称为抽象类，没有说抽象类就一定会是这些情况。
抽象类可以不包含抽象方法，包含抽象方法的类就一定是抽象类。
事实上，抽象类可以是一个完全正常实现的类。
```

为什么要用抽象类

老是在想为什么要引用抽象类，一般类不就够用了吗。一般类里定义的方法，子类也可以覆盖，没必要定义成抽象的啊。

看了下面的文章，明白了一点。

其实不是说抽象类有什么用，一般类确实也能满足应用，但是现实中确实有些父类中的方法确实没有必要写，因为各个子类中的这个方法肯定会有不同，所以没有必要再父类里写。当然你也可以把抽象类都写成非抽象类，但是这样没有必要。

而写成抽象类，这样别人看到你的代码，或你看到别人的代码，你就会注意抽象方法，而知道这个方法是在子类中实现的，所以，有个提示作用。

抽象类和普通类继承功能区别？
普通类继承适合父类方法是公共方法，不需要子类重写，子类都是用父类的公共方法，虽然父类使用final关键字也可以避免子类重写。

抽象类：作为父类方法是抽象的，抽象类适合子类对这个方法都需要重写。如果是普通类继承，
父类还要重写一遍公共方法，然后子类再重写。

一个抽象类小故事

下面看一个关于抽象类的小故事

```
问你个问题，你知道什么是“东西”吗？什么是“物体”吗？
“麻烦你，小王。帮我把那个东西拿过来好吗”
在生活中，你肯定用过这个词——东西。
小王：“你要让我帮你拿那个水杯吗？”
你要的是水杯类的对象。而东西是水杯的父类。通常东西类没有实例对象，但我们有时需要东西的引用指向它的子类实例。

你看你的房间乱成什么样子了，以后不要把东西乱放了，知道么？

上面讲的只是子类和父类。而没有说明抽象类的作用。抽象类是据有一个或多个抽象方法的类，必须声明为抽象类。抽象类的特点是，不能创建实例。

这些该死的抽象类，也不知道它有什么屁用。我非要把它改一改不可。把抽象类中的抽象方法都改为空实现。也就是给抽象方法加上一个方法体，不过这个方

当你这么尝试之后，你发现，原来的代码没有任何变化。大家都还是和原来一样，工作的很好。你这回可能更加相信，抽象类根本就没有什么用。但总是不列

### 一个抽象类小游戏

接下来，我们来写一个小游戏。俄罗斯方块！我们来分析一下它需要什么类？
我知道它要在一个矩形的房子里完成。这个房子的上面出现一个方块，慢慢的下落，当它接触到地面或是其它方块的尸体时，它就停止下落了。然后房子的上
当然，我们不是真的要写一个游戏。所以我们简化它。我抽象出两个必须的类，一个是那个房间，或者就它地图也行。另一个是方块。我发现方块有很多种，

房子上面总是有方块落下来，房子应该有个属性是方块。当一个方块死掉后，再创建一个方块，让它出现在房子的上面。当玩家要翻转方法时，它翻转的到底

我们写一个方块类，用它来派生出6个子类。而房子类的当前方块属性的类型是方块类型。它可以指向任何子类。但是，当我调用当前方块的翻转方法时，它

那么在父类的这个翻转方法中，我写一些什么代码呢？让它有几种状态呢？因为我们不可能实例化一个方块类的实例，所以它的翻转方法中的代码并不重要。

我们发现，方法类不可能有实例，它的翻转方法的内容可以是任何的代码。而子类必须重写父类的翻转方法。这时，你可以把方块类写成抽象类，而它的抽象

当我看到方块类是抽象的，我会很关心它的抽象方法。我知道它的子类一定会重写它，而且，我会去找到抽象类的引用。它一定会有多态性的体现。

但是，如果你没有这样做，我会认为可能会在某个地方，你会实例化一个方块类的实例，但我找了所有的地方都没有找到。最后我会大骂你一句，你是来欺骗

把那些和“东西”差不多的类写成抽象的。而水杯一样的类就可以不是抽象的了。当然水杯也有几千块钱一个的和几块钱一个的。水杯也有子类，例如，我用的

记住一点，面向对象不是来自于Java，面向对象就在你的生活中。而Java的面向对象是方便你解决复杂的问题。这不是说面向对象很简单，虽然面向对象很复
```

接口介绍

接口（英文：Interface），在JAVA编程语言中是一个抽象类型，是抽象方法的集合，接口通常以interface来声明。一个类通过继承接口的方式，从而来继承接口的抽象方法。

接口并不是类，编写接口的方式和类很相似，但是它们属于不同的概念。类描述对象的属性和方法。接口则包含类要实现的方法。

除非实现接口的类是抽象类，否则该类要定义接口中的所有方法。

接口无法被实例化，但是可以被实现。一个实现接口的类，必须实现接口内所描述的所有方法，否则就必须声明为抽象类。另外，在 Java 中，接口类型可用来声明一个变量，他们可以成为一个空指针，或是被绑定在一个以此接口实现的对象。

接口与类相似点：

- 一个接口可以有多个方法。
- 接口文件保存在 .java 结尾的文件中，文件名使用接口名。
- 接口的字节码文件保存在 .class 结尾的文件中。
- 接口相应的字节码文件必须在与包名称相匹配的目录结构中。

接口与类的区别：

- 接口不能用于实例化对象。
- 接口没有构造方法。
- 接口中所有的方法必须是抽象方法。
- 接口不能包含成员变量，除了 static 和 final 变量。
- 接口不是被类继承了，而是要被类实现。
- 接口支持多继承。

接口特性

- 接口中每一个方法也是隐式抽象的,接口中的方法会被隐式的指定为 **public abstract**（只能是 public abstract，其他修饰符都会报错）。
- 接口中可以含有变量，但是接口中的变量会被隐式的指定为 **public static final** 变量（并且只能是 public，用 private 修饰会报编译错误）。
- 接口中的方法是不能在接口中实现的，只能由实现接口的类来实现接口中的方法。

抽象类和接口的区别

- 1. 抽象类中的方法可以有方法体，就是能实现方法的具体功能，但是接口中的方法不行。
- 2. 抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是 **public static final** 类型的。
- 3. 接口中不能含有静态代码块以及静态方法(用 static 修饰的方法)，而抽象类是可以有静态代码块和静态方法。
- 4. 一个类只能继承一个抽象类，而一个类却可以实现多个接口。

> **注**：JDK 1.8 以后，接口里可以有静态方法和方法体了。

接口的使用：

我们来举个例子，定义一个抽象类People，一个普通子类Student，两个接口。子类Student继承父类People,并实现接口Study，Write

代码演示：

```
package demo;
//构建一个抽象类People
abstract class People{
        //父类属性私有化
        private String name;
        private int age;
        //提供父类的构造器
        public People(String name,int age){
                this.name = name;
                this.age = age;
        }
        //提供获取和设置属性的getter()/setter()方法
        public String getName() {
                return name;
        }

        public int getAge() {
                return age;
        }

        public void setName(String name) {
                this.name = name;
```

```
        }

        public void setAge(int age) {
                this.age = age;
        }

        //提供一个抽象方法
        public abstract void talk();
}

//定义一个接口
interface Study{
        //设置课程数量为3
        int COURSENUM = 3;
        //构建一个默认方法 ，jdk1.8后接口方法可以有默认实现
        default void stu(){
                System.out.println("学生需要学习"+COURSENUM+"门课程");
        }
}

//再定义一个接口
interface Write{
        //定义一个抽象方法
        void print();
}

//子类继承People,实现接口Study,Write
class Student extends People implements Study,Write{
        //通过super关键字调用父类的构造器
        public Student(String name, int age) {
                super(name, age);
        }
        //实现父类的抽象方法
        public void talk() {
                System.out.println("我的名字叫"+this.getName()+",今年"+this.getAge()+"岁");
        }
        //实现Write接口的抽象方法
        public void print() {
                System.out.println("学生会写作业");
        }
}

public class InterfaceDemo{
        public static void main(String[] args) {
                //构建student对象
                Student student = new Student("dodo", 22);
                //调用父类的抽象方法
                student.talk();
                //调用接口Write中的抽象方法
                student.print();
                //调用接口Study中的默认方法
                student.stu();
        }
}
```

代码讲解：上述例子结合了抽象类和接口的知识，内容较多，同学们可以多看多敲一下，学习学习。

接口的实现：类名 implements 接口名，有多个接口名，用“，”隔开即可。

接口的作用——制定标准 接口师表尊，所谓的标准，指的是各方共同遵守一个守则，只有操作标准统一了，所有的参与者才可以按照统一的规则操作。

如电脑可以和各个设备连接，提供统一的USB接口，其他设备只能通过USB接口和电脑相连

代码实现：

```
package demo;

interface USB
{
    public void work() ;    // 拿到USB设备就表示要进行工作
}

class Print implements USB              //实现类（接口类）
{                                                       // 打印机实现了USB接口标准（对接口的方法实现）
    public void work()
   {
        System.out.println("打印机用USB接口，连接,开始工作。") ;
    }
}
class Flash implements USB              //实现类（接口类）
```

```
class Flash implements USB                  //实现类（接口类）
{                                                                   // U盘实现了USB接口标准（对接口的方法实现）
    public void work()
    {
        System.out.println("U盘使用USB接口，连接,开始工作。") ;
    }
}

class Computer
{
    public void plugin(USB usb)                         //plugin的意思是插件，参数为接收接口类
    {
        usb.work() ;    // 按照固定的方式进行工作
    }
}
public class InterfaceStandards { public static void main(String args[]) { Computer computer = new Computer() ;
computer.plugin(new Print()) ; //实例化接口类， 在电脑上使用打印机 computer.plugin(new Flash()) ; //实例化接口类， 在电脑上使用U盘
}}
```

代码讲解：上述例子，就给我们展示了接口制定标准的作用，怎么指定的呢？看下面代码

```
class Computer
{
    public void plugin(USB usb)                         //plugin的意思是插件，参数为接收接口类
    {
        usb.work() ;    // 按照固定的方式进行工作
    }
}
```

我们可以看到，Computer类里面定义了一个方法plugin()，它的参数内写的是USB usb,即表示plugin()方法里，接收的是一个usb对象，而打印机和U盘对象可以通过向上转型当参数，传入方法里。我们来重新写一个main方法帮助大家理解

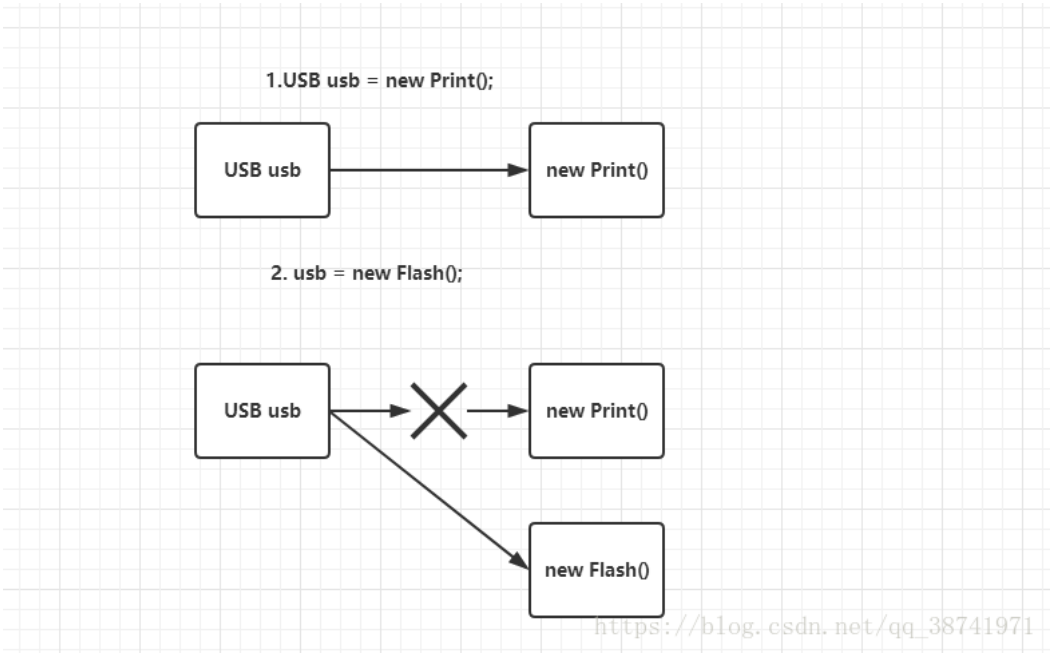
代码演示：

```
public class InterfaceStandards
{
    public static void main(String args[])
    {
        Computer computer = new Computer() ;

        USB usb = new Print();
        computer.plugin(usb) ;   //实例化接口类， 在电脑上使用打印机
        usb = new Flash();
        computer.plugin(usb) ;   //实例化接口类， 在电脑上使用U盘
    }
}
```

代码讲解：我们修改了主函数后，发现，使用了两次的向上转型给了USB，虽然使用的都是usb对象，但赋值的子类对象不一样，实现的方法体也不同，这就很像现实生活，无论我使用的是打印机，还是U盘，我都是通过USB接口和电脑连接的，这就是接口的作用之一——制定标准

我们来个图继续帮助大家理解一下：



上面的图：我们学习前面的章节多态可以知道对象的多态可以通过动态绑定来实现，即使用向上转型，我们知道类，数组，接口都是引用类型变量，什么是引用类型变量？

引用类型变量都会有一个地址的概念，即指向性的概念，当USB usb = new Print(),此时usb对象是指向new Print()的，当usb = new Flash()后，这时候usb变量就会指向new Flash()，我们会说这是子类对象赋值给了父类对象usb，而在内存中，我们应该说，usb指向了new Flash();

接口最佳实践：设计模式中的工厂模式

首先我们来认识一下什么是工厂模式？工厂模式是为了解耦：把对象的创建和使用的过程分开。就是Class A 想调用 Class B ，那么A只是调用B的方法，而至于B的实例化，就交给工厂类。

其次，工厂模式可以降低代码重复。如果创建对象B的过程都很复杂，需要一定的代码量，而且很多地方都要用到，那么就会有很多的重复代码。我们可以这些创建对象B的代码放到工厂里统一管理。既减少了重复代码，也方便以后对B的创建过程的修改维护。

由于创建过程都由工厂统一管理，所以发生业务逻辑变化，不需要找到所有需要创建B的地方去逐个修正，只需要在工厂里修改即可，降低维护成本。同理，想把所有调用B的地方改成B的子类C，只需要在对应生产B的工厂中或者工厂的方法中修改其生产的对象为C即可，而不需要找到所有的new B（）改为newC()。

代码演示：

```
package demo;

import java.util.Scanner;

interface Fruit                                          //定义一个水果标准
{
        public abstract void eat();
}

class Apple implements Fruit
{
        public void eat()
        {
                System.out.println("吃苹果");
        }
}
class Orange implements Fruit
{
        public void eat()
        {
                System.out.println("吃橘子");
        }
}

class factory
{
        public static Fruit getInstance(String className)  //返回值是Fruit的子类
        {
                if("apple".equals(className))
                {
                        return new Apple();
                }
```

```
                else if("orange".equals(className))
                {
                        return new Orange();
                }
                else
                {
                        return null;
                }
        }
}

public class ComplexFactory {
        public static void main(String[] args)
        {
                System.out.println("请输入水果的英文名:");
                Scanner sc = new Scanner(System.in);
                String ans = sc.nextLine();
                Fruit f = factory.getInstance(ans);   //初始化参数
                f.eat();
                sc.close();
        }
}
```

代码讲解：上述代码部分我们讲一下factory这个类，类中有一个getInstance方法，我们用了static关键字修饰，在使用的时候我们就在main中使用类名.方法名调用。

Fruit f = factory.getInstance(ans); //初始化参数 在Factory的getInstance()方法中，我们就可以通过逻辑的实现，将对象的创建和使用的过程分开了。

> 总结点评：在接口的学习中，大家可以理解接口是特殊的抽象类，java中类可以实现多个接口，接口中成员属性默认是public static final修饰，可以省略；成员方法默认是public abstract修饰，同样可以省略，接口中还可定义带方法体的默认方法，需要使用default修饰。利用接口我们还可以制定标准，还能够使用工厂模式，将对象的创建和使用过程分开。

接口与抽象类的本质区别是什么?

基本语法区别

在 Java 中，接口和抽象类的定义语法是不一样的。这里以动物类为例来说明，其中定义接口的示意代码如下：

```
public interface Animal
{
    //所有动物都会吃
    public void eat();

    //所有动物都会飞
    public void fly();
}
```

定义抽象类的示意代码如下：

```
public abstract class Animal
{
    //所有动物都会吃
    public abstract void eat();

    //所有动物都会飞
    public void fly(){};
}
```

可以看到，在接口内只能是功能的定义，而抽象类中则可以包括功能的定义和功能的实现。在接口中，所有的属性肯定是 public、static 和 final，所有的方法都是 abstract，所以可以默认不写上述标识符；在抽象类中，既可以包含抽象的定义，也可以包含具体的实现方法。

在具体的实现类上，接口和抽象类的实 现类定义方式也是不一样的，其中接口实现类的示意代码如下：

```
public class concreteAnimal implements Animal
{
    //所有动物都会吃
    public void eat(){}

    //所有动物都会飞
    public void fly(){}
}
```

抽象类的实现类示意代码如下：

```
public class concreteAnimal extends Animal
{
    //所有动物都会吃
    public void eat(){}

    //所有动物都会飞
    public void fly(){}
}
```

可以看到，在接口的实现类中使用 implements 关键字；而在抽象类的实现类中，则使用 extends 关键字。一个接口的实现类可以实现多个接口，而一个抽象类的实现类则只能实现一个抽象类。

设计思想区别

从前面抽象类的具体实现类的实现方式可以看出，其实在 Java 中，抽象类和具体实现类之间是一种继承关系，也就是说如果采用抽象类的方式，则父类和子类在概念上应该是相同的。接口却不一样，如果采用接口的方式，则父类和子类在概念上不要求相同。

接口只是抽取相互之间没有关系的类的共同特征，而不用关注类之间的关系，它可以使没有层次关系的类具有相同的行为。因此，可以这样说：抽象类是对一组具有相同属性和方法的逻辑上有关系的事物的一种抽象，而接口则是对一组具有相同属性和方法的逻辑上不相关的事物的一种抽象。

仍然以前面动物类的设计为例来说明接口和抽象类关于设计思想的区别，该动物类默认所有的动物都具有吃的功能，其中定义接口的示意代码如下：

```
public interface Animal
{
    //所有动物都会吃
    public void eat();
}
```

定义抽象类的示意代码如下：

```
public abstract class Animal
{
    //所有动物都会吃
    public abstract void eat();
}
```

不管是实现接口，还是继承抽象类的具体动物，都具有吃的功能，具体的动物类的示意代码如下。

接口实现类的示意代码如下：

```
public class concreteAnimal implements Animal
{
    //所有动物都会吃
    public void eat(){}
}
```

抽象类的实现类示意代码如下：

```
public class concreteAnimal extends Animal
{
    //所有动物都会吃
    public void eat(){}
}
```

当然，具体的动物类不光具有吃的功能，比如有些动物还会飞，而有些动物却会游泳，那么该如何设计这个抽象的动物类呢？可以别在接口和抽象类中增加飞的功能，其中定义接口的示意代码如下：

```
public interface Animal
{
    //所有动物都会吃
    public void eat();

    //所有动物都会飞
    public void fly();
}
```

定义抽象类的示意代码如下：

```
public abstract class Animal
{
    //所有动物都会吃
    public abstract void eat();

    //所有动物都会飞
    public void fly(){};
}
```

这样一来，不管是接口还是抽象类的实现类，都具有飞的功能，这显然不能满足要求，因为只有一部分动物会飞，而会飞的却不一定是动物，比如飞机也会飞。那该如何设计呢？有很多种方案，比如再设计一个动物的接口类，该接口具有飞的功能，示意代码如下：

```
public interface AnimaiFly
{
    //所有动物都会飞
    public void fly();
}
```

那些具体的动物类，如果有飞的功能的话，除了实现吃的接口外，再实现飞的接口，示意代码如下：

```
public class concreteAnimal implements Animal,AnimaiFly
{
    //所有动物都会吃
    public void eat(){}

    //动物会飞
    public void fly();
}
```

那些不需要飞的功能的具体动物类只实现具体吃的功能的接口即可。另外一种解决方案是再设计一个动物的抽象类，该抽象类具有飞的功能，示意代码如下：

```
public abstract class AnimaiFly
{
    //动物会飞
    public void fly();
}
```

但此时没有办法实现那些既有吃的功能，又有飞的功能的具体动物类。因为在 Java 中具体的实现类只能实现一个抽象类。一个折中的解决办法是，让这个具有飞的功能的抽象类，继承具有吃的功能的抽象类，示意代码如下：

```
public abstract class AnimaiFly extends Animal
{
    //动物会飞
    public void fly();
}
```

此时，对那些只需要吃的功能的具体动物类来说，继承 Animal 抽象类即可。对那些既有吃的功能又有飞的功能的具体动物类来说，则需要继承 AnimalFly 抽象类。

但此时对客户端有一个问题，那就是不能针对所有的动物类都使用 Animal 抽象类来进行编程，因为 Animal 抽象类不具有飞的功能，这不符合面向对象的设计原则，因此这种解决方案其实是行不通的。

还有另外一种解决方案，即具有吃的功能的抽象动物类用抽象类来实现，而具有飞的功能的类用接口实现；或者具有吃的功能的抽象动物类用接口来实现，而具有飞的功能的类用抽象类实现。

具有吃的功能的抽象动物类用抽象类来实现，示意代码如下：

```
public abstract class Animal
{
    //所有动物都会吃
    public abstract void eat();
}
```

具有飞的功能的类用接口实现，示意代码如下：

```
public interface AnimaiFly
{
    //动物会飞
    public void fly();
}
```

既具有吃的功能又具有飞的功能的具体的动物类，则继承 Animal 动物抽象类，实现 AnimalFly 接口，示意代码如下：

```
public class concreteAnimal extends Animal implements AnimaiFly
{
    //所有动物都会吃
    public void eat(){}

    //动物会飞
    public void fly();
}
```

或者具有吃的功能的抽象动物类用接口来实现，示意代码如下：

```
public interface Animal
{
    //所有动物都会吃
    public abstract void eat();
}
```

具有飞的功能的类用抽象类实现，示意代码如下：

```
public abstract class AnimaiFly
{
    //动物会飞
    public void fly(){};
}
```

既具有吃的功能又具有飞的功能的具体的动物类，则实现 Animal 动物类接口，继承 AnimaiFly 抽象类，示意代码如下：

```
public class concreteAnimal extends AnimaiFly implements Animal
{
    //所有动物都会吃
    public void eat(){}

    //动物会飞
    public void fly();
}
```

这些解决方案有什么不同呢？再回过头来看接口和抽象类的区别：抽象类是对一组具有相同属性和方法的逻辑上有关系的事物的一种抽象，而接口则是对一组具有相同属性和方法的逻辑上不相关的事物的一种抽象，因此抽象类表示的是“is a”关系，接口表示的是“like a”关系。

假设现在要研究的系统只是动物系统，如果设计人员认为对既具有吃的功能又具有飞的功能的具体的动物类来说，它和只具有吃的功能的动物一样，都是动物，是一组逻辑上有关系的事物，因此这里应该使用抽象类来抽象具有吃的功能的动物类，即继承 Animal 动物抽象类，实现 AnimalFly 接口。

如果设计人员认为对既具有吃的功能，又具有飞的功能的具体的动物类来说，它和只具有飞的功能的动物一样，都是动物，是一组逻辑上有关系的事物，因此这里应该使用抽象类来抽象具有飞的功能的动物类，即实现 Animal 动物类接口，继承 AnimaiFly 抽象类。

假设现在要研究的系统不只是动物系统，如果设计人员认为不管是吃的功能，还是飞的功能和动物类没有什么关系，因为飞机也会飞，人也会吃，则这里应该实现两个接口来分别抽象吃的功能和飞的功能，即除实现吃的 Animal 接口外，再实现飞的 AnimalFly 接口。

从上面的分析可以看出，对于接口和抽象类的选择，反映出设计人员看待问题的不同角度，即抽象类用于一组相关的事物，表示的是“is a”的关系，而接口用于一组不相关的事物，表示的是“like a”的关系。

如何回答面试题：接口和抽象类的区别?

接口(interface)和抽象类(abstract class)是支持抽象类定义的两种机制。

接口是公开的，不能有私有的方法或变量，接口中的所有方法都没有方法体，通过关键字interface实现。

抽象类是可以有私有方法或私有变量的，通过把类或者类中的方法声明为abstract来表示一个类是抽象类，被声明为抽象的方法不能包含方法体。子类实现方法必须含有相同的或者更低的访问级别(public->protected->private)。抽象类的子类为父类中所有抽象方法的具体实现，否则也是抽象类。

接口可以被看作是抽象类的变体，接口中所有的方法都是抽象的，可以通过接口来间接的实现多重继承。接口中的成员变量都是public static final类型，由于抽象类可以包含部分方法的实现，所以，在一些场合下抽象类比接口更有优势。

相同点

（1）都不能被实例化 （2）接口的实现类或抽象类的子类都只有实现了接口或抽象类中的方法后才能实例化。

不同点

(1)接口只有定义，不能有方法的实现，java 1.8中可以定义default方法体，而抽象类可以有定义与实现，方法可在抽象类中实现。

(2)实现接口的关键字为implements，继承抽象类的关键字为extends。一个类可以实现多个接口，但一个类只能继承一个抽象类。所以，使用接口可以间接地实现多重继承。

(3)接口强调特定功能的实现，而抽象类强调所属关系。

(4)接口成员变量默认为public static final，必须赋初值，不能被修改；其所有的成员方法都是public、abstract的。抽象类中成员变量默认default，可在子类中被重新定义，也可被重新赋值；抽象方法被abstract修饰，不能被private、static、synchronized和native等修饰，必须以分号结尾，不带花括号。

(5)接口被用于常用的功能，便于日后维护和添加删除，而抽象类更倾向于充当公共类的角色，不适用于日后重新对立面的代码修改。功能需要累积时用抽象类，不需要累积时用接口。

JDK8j接口增强

原来接口中方法必须是public abstract修饰，不能带方法体，实现必须依靠子类实现抽象方法。

```
public interface JDK8Interface {
    // static修饰符定义静态方法
    static void staticMethod() {
        System.out.println("接口中的静态方法");
    }
    // default修饰符定义默认方法
    default void defaultMethod() {
        System.out.println("接口中的默认方法");
    }
```

为什么要引入 default 方法
和接口中定义的其他方法一样，default 方法默认是 public
接口是用来定义类的行为的，如果要在接口中新添加方法，那么所有实现此接口的类都需要强制的实现新添加的方法。而 default 方法就可以规避该问题。默认有实现好的方法。default方法在接口中实现后，可以不用在接口的实现类中重写，因此这是一种向下兼容的妥协。

默认方法default带来的问题?
当多个接口定义了相同 default 方法时候，当一个类实现的多个接口定义了相同的 default 方法，那么编译时会失败。需要子类 Override 该方法实现。

static方法

静态方法，只能通过接口名调用，不可以通过实现类的类名或者实现类的对象调用。default方法，只能通过接口实现类的对象来调用。JDK8Interface.staticMethod()

函数式接口(Functional Interface)是只有一个抽象方法的接口。

1. 函数式接口中的抽象函数就是为了支持 lambda表达式；为什么出现了函数式接口
2. 函数式接口可以被隐式转换为lambda表达式；
3. 为确保函数式接口符合语法，可以添加@FunctionalInterface注解；

```
@FunctionalInterface
public interface FuncInterface {

   //只有一个抽象方法
   public void reference();

         //还可以有其他方法
   //interface default method
   default void defaultMehtod() {
      System.out.println("This is a default method~");
   }

   //interface second default method
   default void anotherDefaultMehtod() {
      System.out.println("This is the second default method~");
   }

   //interface static method
   static void staticMethod() {
      System.out.println("This is a static method~");

   }

   //interface second static method
   default void anotherStaticMethod() {
      System.out.println("This is the second static method~");
   }
}
```