

Lesson7

面向对象的三大特称之三：多态 (polymorphism)

主讲老师：申雪萍



主要内容

- 多态的必要性
- 静多态和动多态
- 方法重载，方法覆盖
- 多态的优点及运行机制
- 抽象方法
- 抽象类
- 接口的必要性（将接口用作API）
- 定义接口
- 实现接口
- 将接口用作类型、接口回调（使用接口）
- 接口的进化（通过接口的继承完成）
- 面向接口的编程
- 案例分析
- Upcasting和downcasting

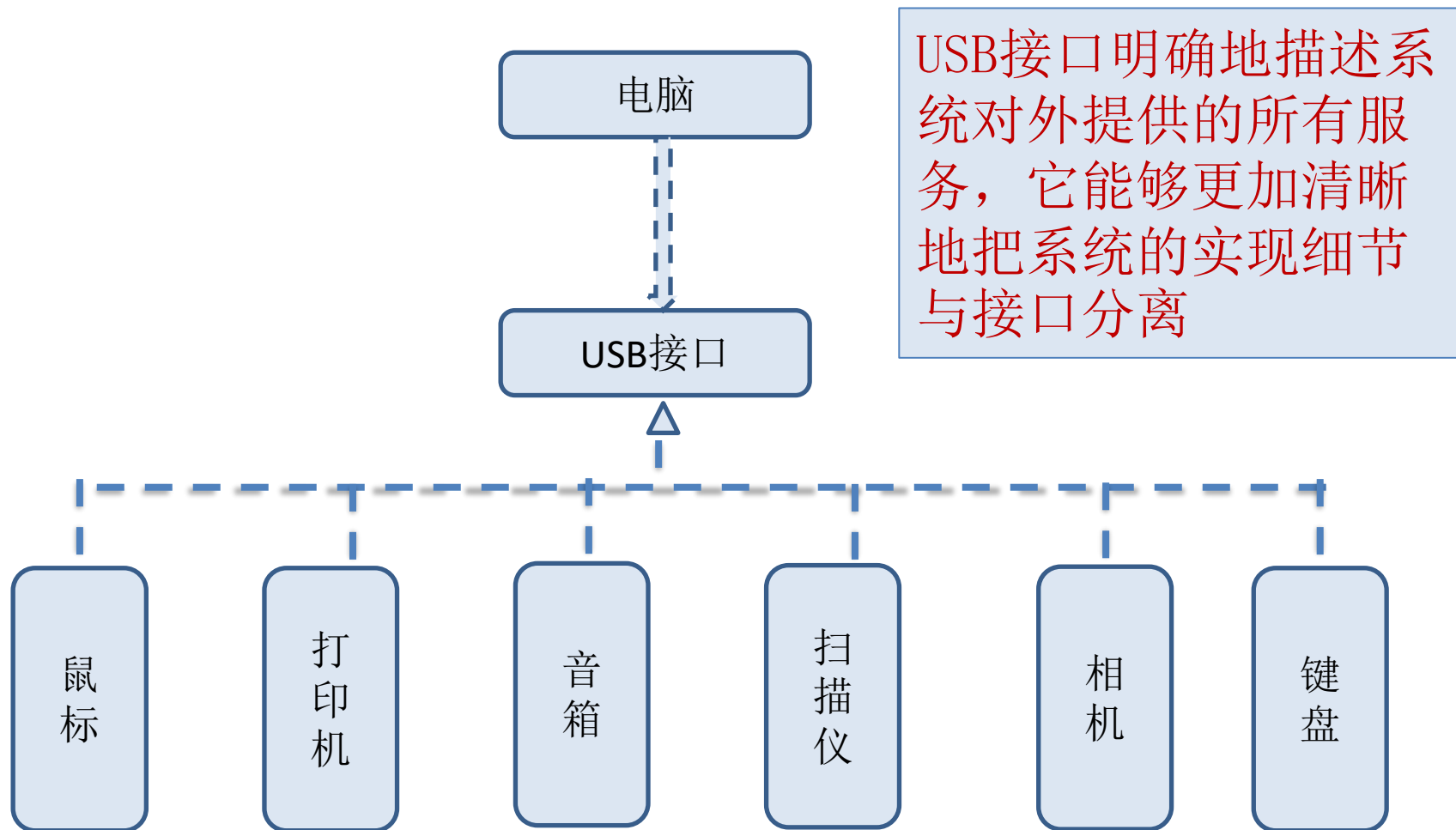
接口的含义(接口有两种意思)



手机的接口



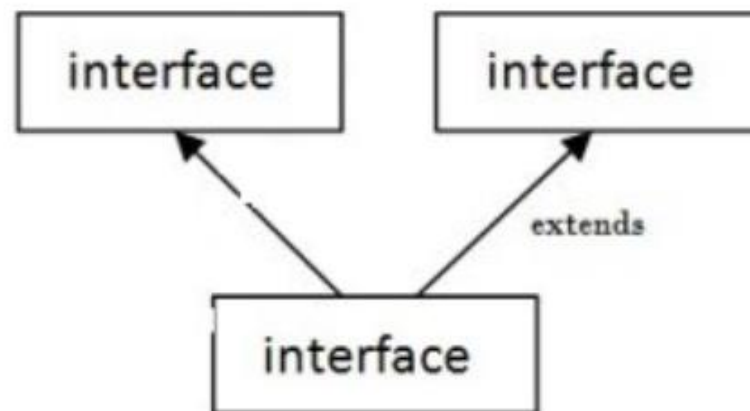
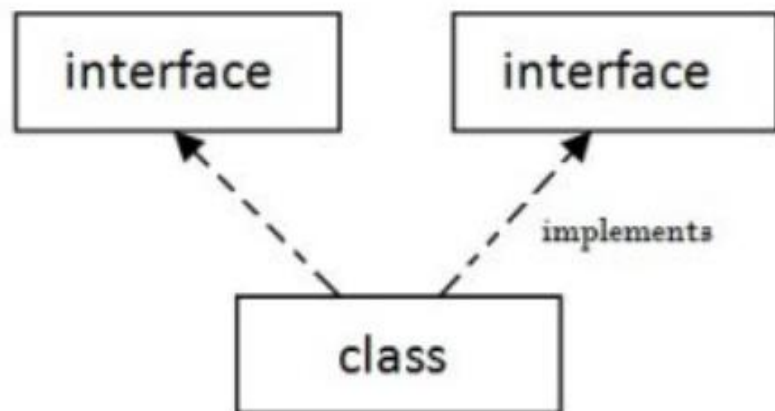
接口的含义



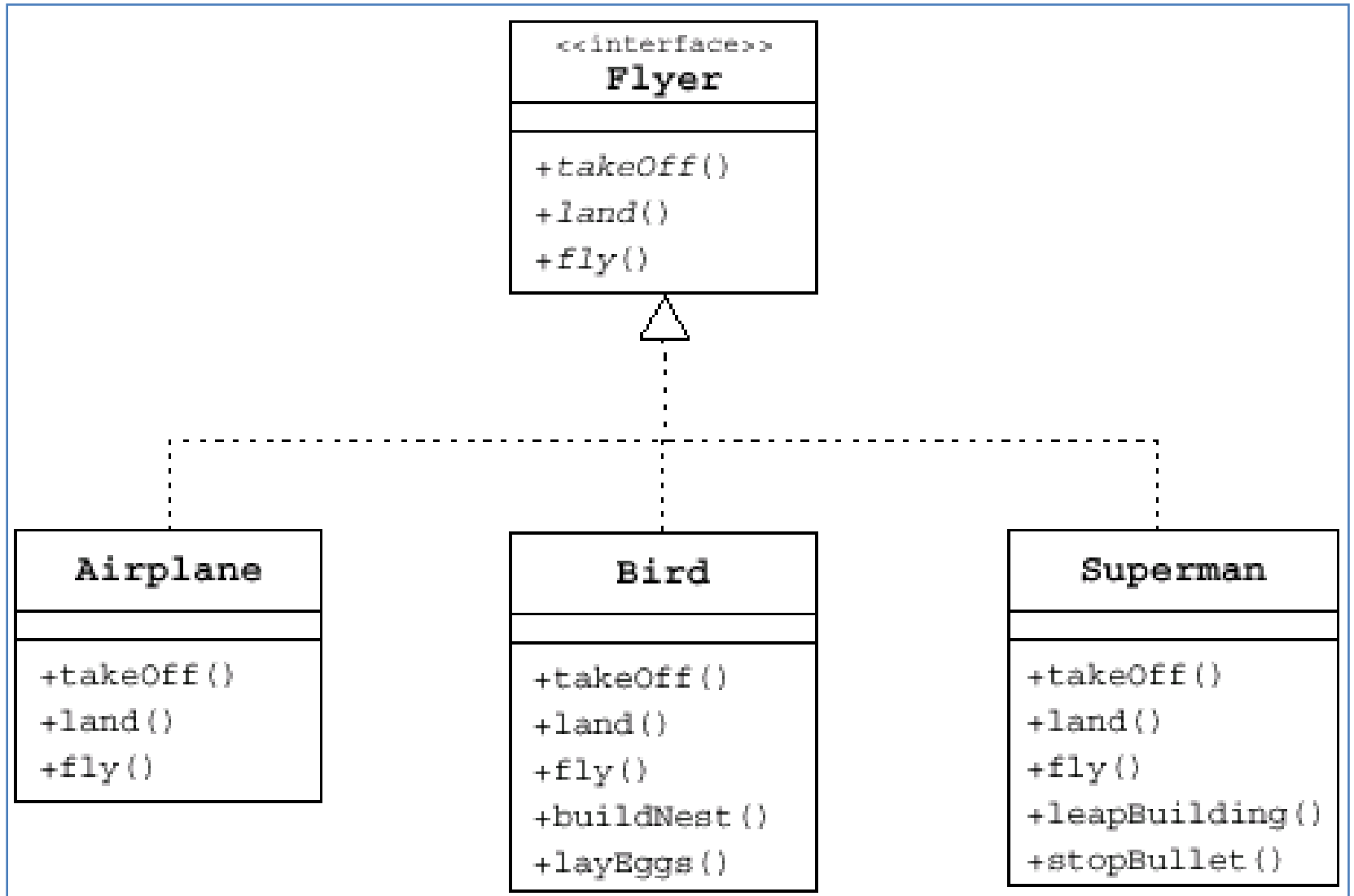
接口的含义

- 在Java语言中，接口有两种意思：
 - ① 一是指概念性的接口，即指类对外提供的所有服务。
例如：类的所有能被其他程序访问的方法构成了类的接口、USB接口等
 - ② 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**。Java接口是一系列方法的声明，是一些方法特征的集合，**一个接口只有方法的特征没有方法的实现**，因此这些方法可以在不同的地方被不同的类实现，而这些实现可以具有不同的行为（功能）

Java中接口和类，接口和接口的关系



实例:

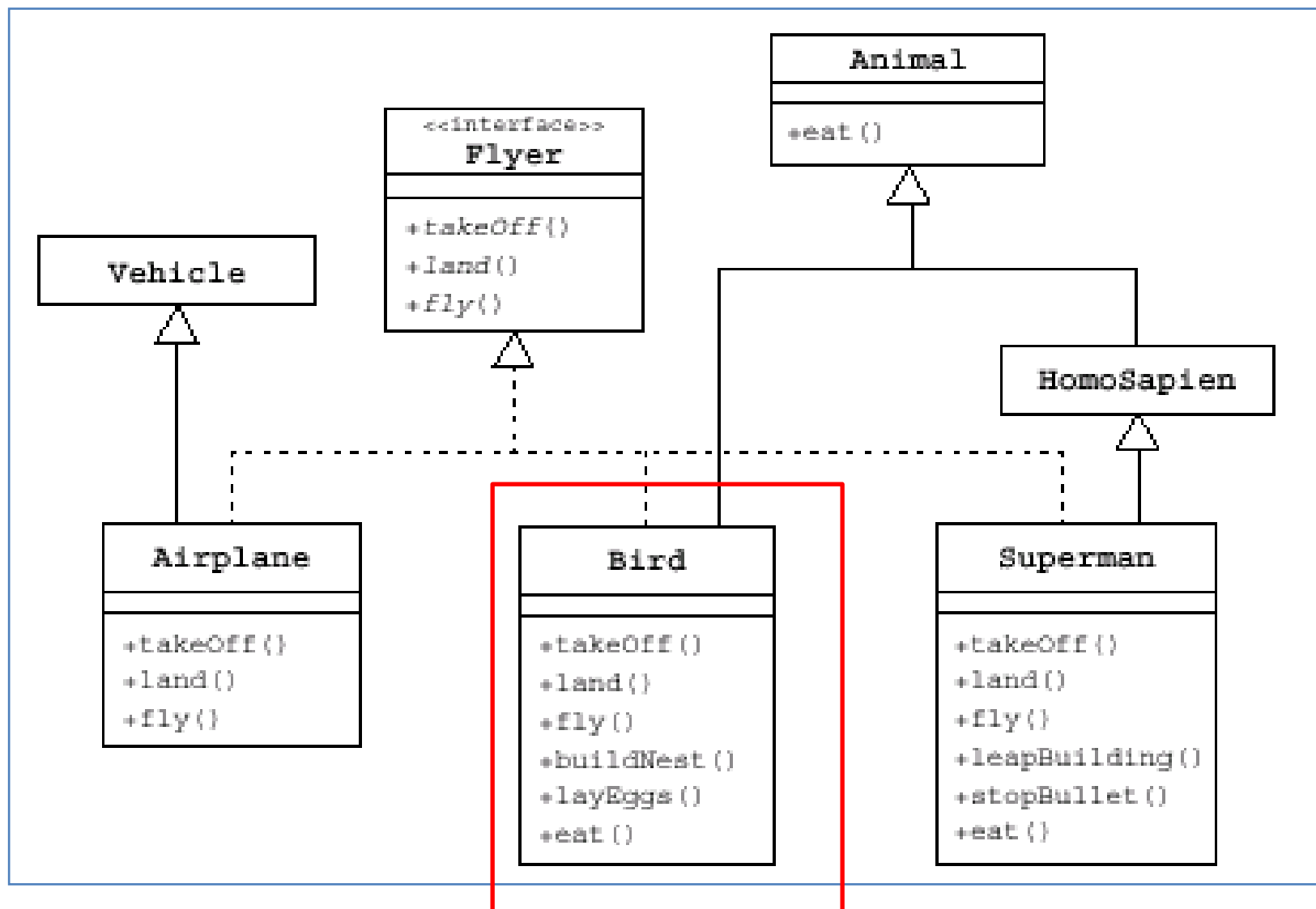


接口的含义

```
public interface Flyer {  
    public void takeOff();  
    public void land();  
    public void fly();  
}
```

接口是用来实现类间（不相关类）多重继承功能的结构

接口的含义



```
public class Bird extends Animal implements Flyer {  
    public void takeOff() {  
        /* take- off implementation */ }  
    public void land() {  
        /* landing implementation */ }  
    public void fly() {  
        /* fly implementation */ }  
    public void buildNest() {  
        /* nest building behavior */ }  
    public void layEggs() {  
        /* egg laying behavior */ }  
    public void eat() {  
        /* override eating behavior */ }  
}
```

接口的概念

- 从本质上讲，接口是一种特殊的抽象类，这种抽象类中只包含常量和方法的定义，而没有方法的实现。接口是抽象方法和常量值的定义的集合。
- 接口是用来实现类间（不相关类）多重继承功能的结构。
- 从语法上看，接口是一种与“类”很相似的结构，只是接口中的所有方法都是抽象的，只有声明、没有方法体。
- 接口声明的关键字是**interface**。

接口要点

- ① 接口是Java中的一种复合数据类型，是用 `interface` 关键字来定义的；
- ② 接口是一种特殊的“类”，一种特殊的“抽象类”；
- ③ 接口中所有的方法都默认是 **public abstract** 的，并且只有方法头和参数列表，没有方法体；
- ④ 接口中所有的 **变量都默认是 public static final** 的；

接口要点（续）

- ⑤ 接口中没有构造方法；
- ⑥ 一个类可以实现多个接口。
- ⑦ 接口中的方法体可以由 java 语言书写，也可以由其他语言书写，**用其他语言书写时，接口方法需要用 native 关键字修饰**

定义(声明)接口

```
[public][interface]接口名称[extends父接口名列表]
{
//静态常量
[public][static][final]数据类型 变量名=常量名;
//抽象方法
[public][abstract][native]返回值类型
        方法名（参数列表）；
}
```

实现接口

```
[修饰符]class类名[extends父类名]implements接口A,接口B,...
```

```
{
```

类的成员变量和成员方法;

为接口A中的所有方法编写方法体, 实现接口A;

为接口A中的所有方法编写方法体, 实现接口B;

...

```
}
```

```
[修饰符] class A implements IA {...}
```

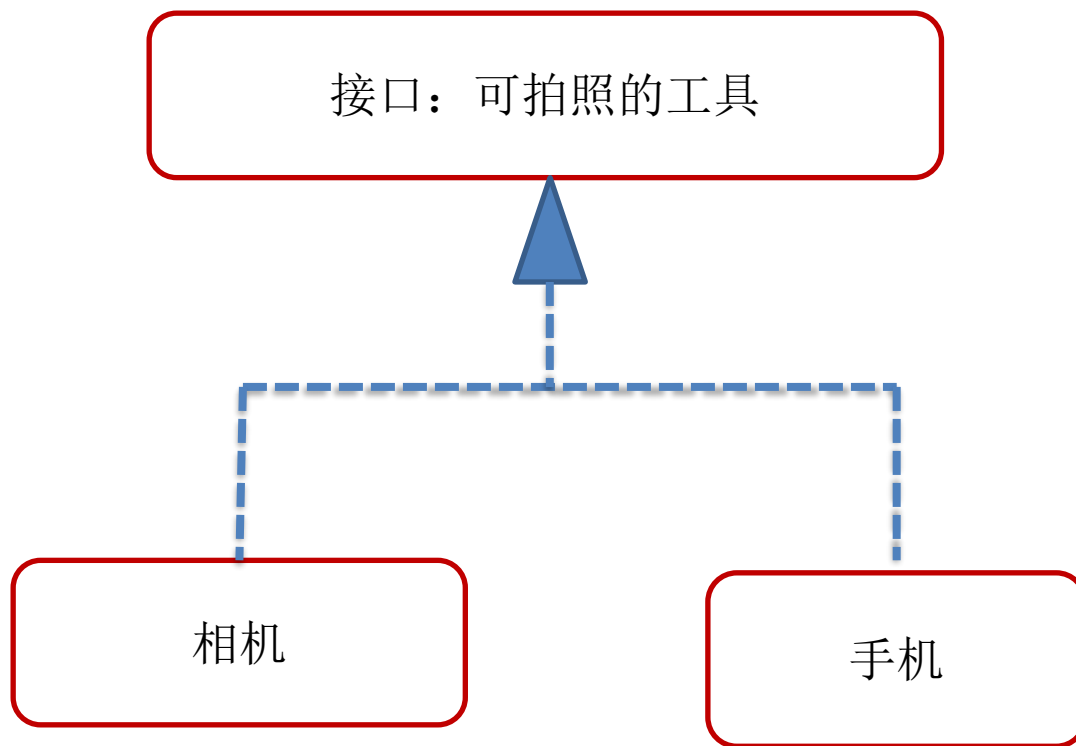
```
[修饰符] class B extends A implements IB, IC  
{...}
```

```
interface IExample {  
    void method1();  
    void method2();  
}
```

```
abstract class Example1 implements IExample {  
    public void method1(){  
        //... ..  
    }  
}
```

因为只实现了一个方法，所以类Example1需要定义成抽象类。

案例分析：接口声明，接口实现



接口的声明:

```
/** 表示所有能拍照的工具类型 */  
public interface Photographable{  
    /** 拍照 */  
    public void takePhoto();  
}
```

实现接口：类实现接口的关键字为implements

```
public class Camera implements Photographable {  
    public void takePhoto(){...}; //实现拍照功能  
}
```

```
public class CellPhone implements Photographable {  
    public void takePhoto(){...}; //实现拍照功能  
}
```

注意事项

- ① 一个类在实现某个接口的抽象方法时，必须以完全相同的方法头。否则，只是在重载一个新方法，而不是实现已有的抽象方法。
- ② 接口的抽象方法的访问限制符默认为 `public`，所以类在实现这些抽象方法时，必须显式的使用 `public` 修饰符，否则将被警告为缩小了接口中定义的方法的访问控

接口的必要性（小结）

- ① 通过接口可以实现不相关类的相同行为，而不需要考虑这些类之间的层次关系。通过接口可以指明多个类需要实现的方法。
- ② 通过接口可以了解对象的**交互界面**，而不需要了解内部细节。某种程度上，接口是一种更高级的封装
 - ① 系统之间的交互界面
 - ② 模块之间的交互界面
 - ③ 子模块之间的交互界面
 - ④ 软件与硬件之间交互界面

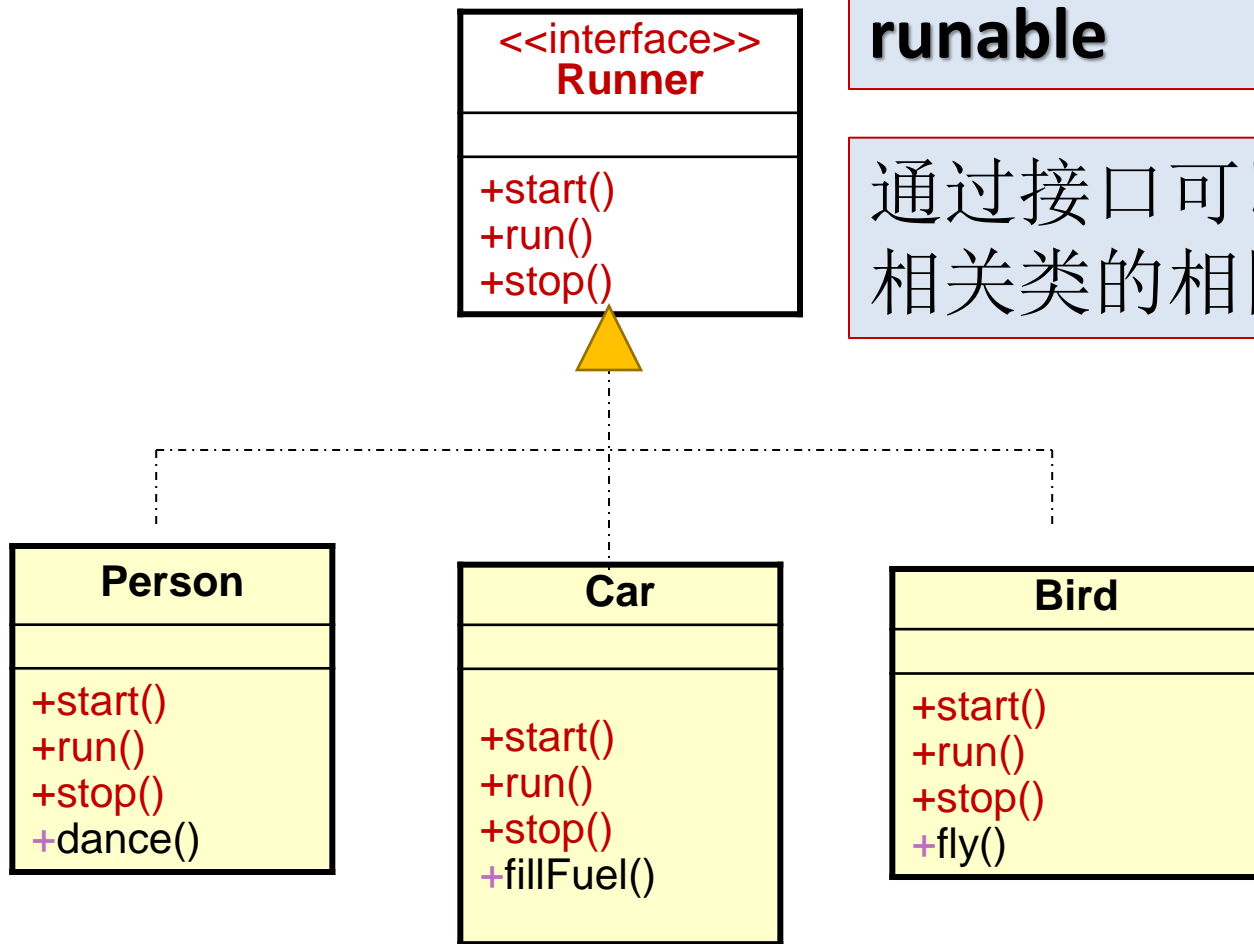
- 1、**丰富Java面向对象的思想**：在Java语言中，`abstract class` 和 `interface` 是支持抽象类定义两种机制。正是由于这两种机制的存在，才赋予了Java强大的面向对象能力。
- 2、**有利于代码的规范性**：如果一个项目比较庞大，那么就需要一个能理清所有业务的架构师来定义一些主要的接口，这些接口不仅告诉开发人员你需要实现那些业务，而且也将命名规范限制住了

- 3、**有利于对代码进行维护**：可以一开始定义一个接口，把功能菜单放在接口里，然后定义类时实现这个接口，以后要换的话只不过是引用另一个类而已，这样就达到维护、拓展的方便性。
- 4、**增强安全、严密性**：接口是实现软件松耦合的重要手段，它描叙了系统对外的所有服务，而不涉及任何具体的实现细节。这样就比较安全、严密一些(一般软件服务商考虑的比较多)。

使用接口：将接口用作类型、接口回调

接口代表一种能力
runable

通过接口可以实现不
相关类的相同行为




```
interface Runner {           //接口 1
```

```
    public void run();
```

```
}
```

```
interface Swimmer {          //接口 2
```

```
    public void swim();
```

```
}
```

```
abstract class Animal {      //抽象类,去掉关键字abstract是否可行?
```

```
    public abstract void eat();
```

```
}
```

```
class Person extends Animal implements Runner,Swimmer { //继承类, 实现接口
```

```
    public void run() {
```

```
        System.out.println("我是飞毛腿,跑步速度极快!");
```

```
    }
```

```
    public void swim(){
```

```
        System.out.println("我游泳技术很好,会蛙泳、自由泳、仰泳、蝶泳...");
```

```
    }
```

```
    public void eat(){
```

```
        System.out.println("我牙好胃好,吃啥都香!");
```

```
    }
```

```
}
```

```
public class InterfaceTest{
```

```
    public void m1(Runner r) { r.run(); } //接口作参数  
    public void m2(Swimmer s) {s.swim();} //接口作参数  
    public void m3(Animal a) {a.eat();} //抽象类引用
```

```
    public static void main(String args[]){  
        InterfaceTest t = new InterfaceTest();  
        Person p = new Person();
```

```
        t.m1(p); //接口回调  
        t.m2(p); //接口回调  
        t.m3(p); //接口回调
```

```
    }
```

```
}
```

程序运行结果：
我是飞毛腿,跑步速度极快!
我游泳技术很好,会蛙泳、自由泳、仰泳、蝶泳...
我牙好胃好,吃啥都香!

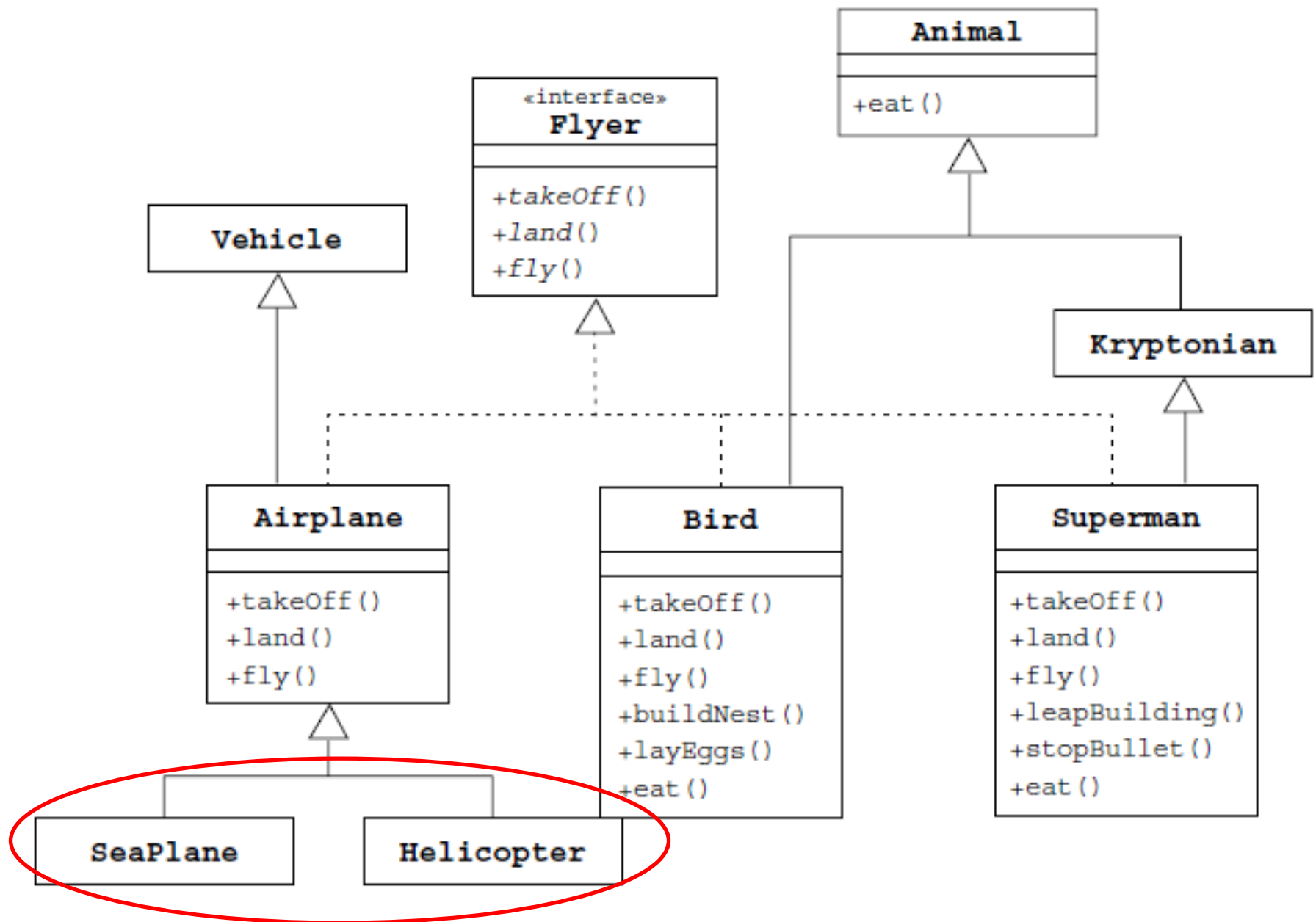
接口的使用:将接口用作类型

- 接口变量:
 - 声明格式: 接口 变量名(又称为引用)
 - 接口做参数: 如果一个方法的参数是接口类型, 就可以将任何实现该接口的类的实例的引用传递给接口参数, 那么接口参数就可以回调类实现的接口方法。

什么是接口回调？

- 接口回调：
 - 把实现某一接口的类创建的对象引用赋给该接口声明的接口变量
 - 该接口变量就可以调用被类实现的接口中的方法。
 - 即：

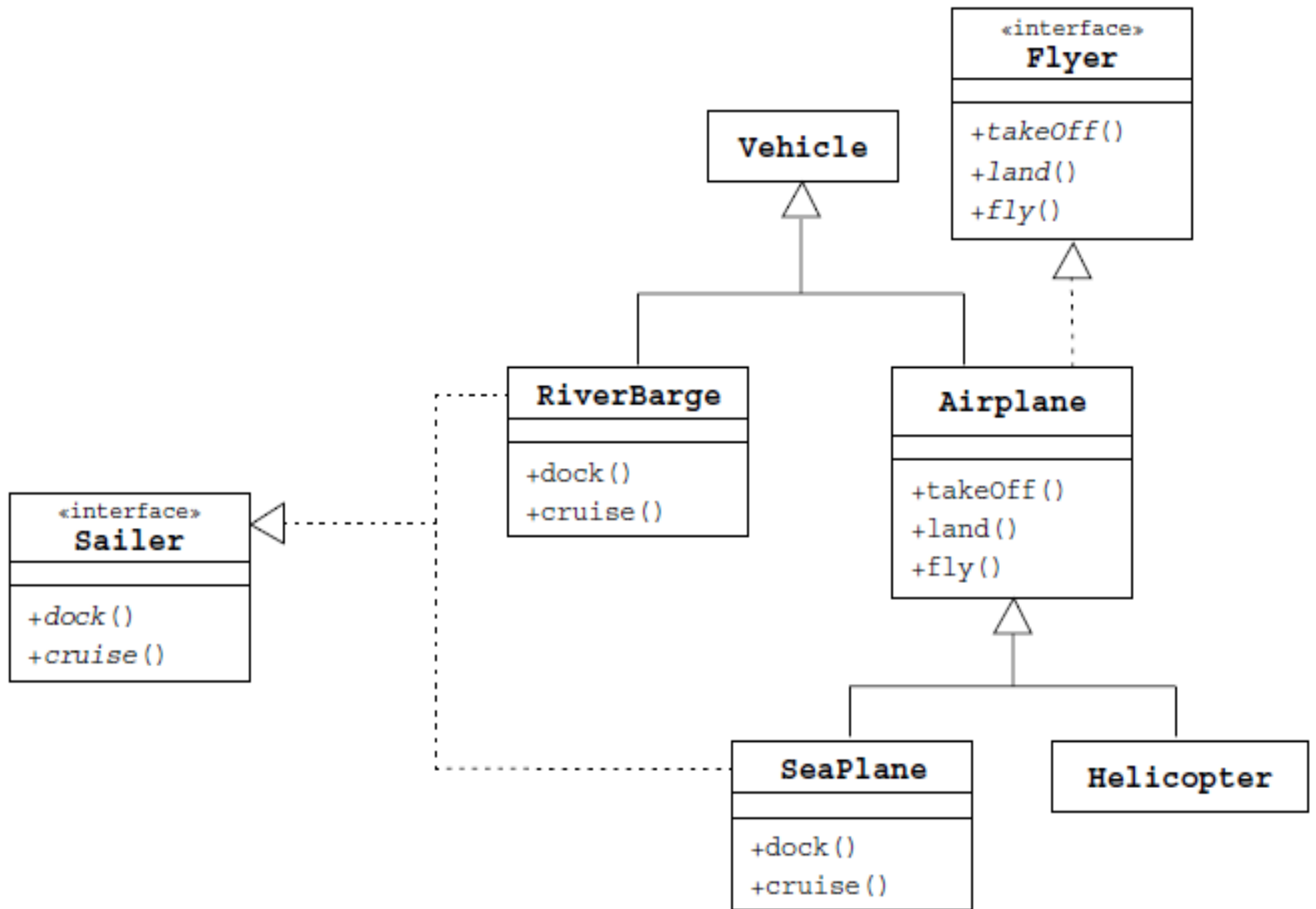
接口变量=实现该接口的类所创建的对象；
接口变量.接口方法([参数列表])；



```
public class Airport {  
    public static void main(String[] args) {  
        Airport metropolisAirport = new Airport();  
        Helicopter copter = new Helicopter();  
        SeaPlane sPlane = new SeaPlane();  
  
        metropolisAirport.givePermissionToLand(copter);  
        metropolisAirport.givePermissionToLand(sPlane);  
    }  
  
    private void givePermissionToLand(Flyer f) {  
        f.land();  
    }  
}
```

接口回调

接口作为参数



```
public class Harbor {  
    public static void main(String[] args) {  
        Harbor bostonHarbor = new Harbor();  
        RiverBarge barge = new RiverBarge();  
        SeaPlane sPlane = new SeaPlane();  
  
        bostonHarbor.givePermissionToDock(barge);  
        bostonHarbor.givePermissionToDock(sPlane);  
    }  
  
    private void givePermissionToDock(Sailer s) {  
        s.dock();  
    }  
}
```

接口回调

接口用做参数

接口的进化（通过接口的继承完成）

接口可以继承，而且可以多重继承

```
interface IA {...}
```

```
interface IB {...}
```

```
interface IC {...}
```

```
interface ID extends IA, IB, IC {...}
```

```
interface A {  
    char a = 'A';  
    void showa();  
}  
interface B extends A {  
    char b = 'B';  
    void showb();  
}  
interface C {  
    char c = 'C';  
    void showc();  
}  
interface D extends B, C {  
    char d = 'D';  
    void showd();  
}
```

```
class E implements D {  
    char e = 'E';  
    public void showa() {  
        System.out.println("这里是接口" + a);  
    }  
    public void showb() {  
        System.out.println("这里是接口" + b);  
    }  
    public void showc() {  
        System.out.println("这里是接口" + c);  
    }  
    public void showd() {  
        System.out.println("这里是接口" + d);  
    }  
    public void showe() {  
        System.out.println("这里是类" + e);  
    }  
}
```

```
class InterfaceTest1 {  
    public static void main(String args[]) {  
        E e = new E();  
        e.showa();  
        e.showb();  
        e showc();  
        e.showd();  
        e.showe();  
    }  
}
```

这里是接口A

这里是接口B

这里是接口C

这里是接口D

这里是类E

抽象类与接口共同点：

- 都不能被实例化
- 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。
- 抽象类与接口都是为了继承与多态，它们都需要子类来继承或实现才有意义，最终目的是为了多态；子类重写了父类的方法，再通过向上转型，由父类对象引用指向子类对象，达到运行时动态调用子类方法的目的。

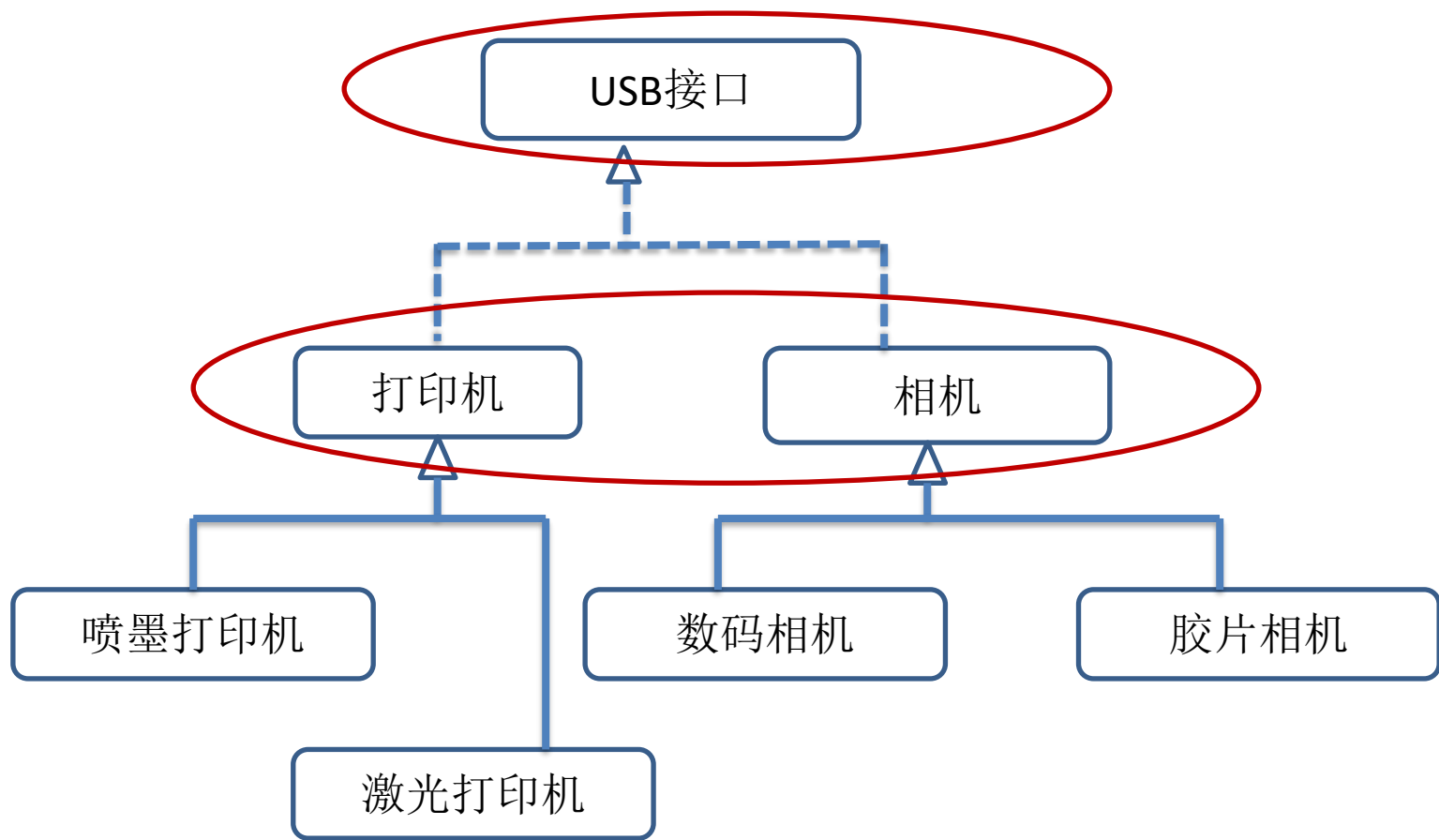
抽象类与接口区别（1）：

- 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。
- 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。
- 接口中没有构造方法，抽象类中有构造方法。接口中所有方法都是抽象方法，抽象类中可以有，也可以没有抽象方法。抽象类比接口包含了更多的实现细节。

抽象类与接口区别（2）：

- 抽象类是某一类事物的一种抽象，而接口不是类，它只定义了某些行为；
 - 例如，“生物”类虽然抽象，但有“狗”类的雏形，接口中的run方法可以由狗类实现，也可以由汽车实现。
- 在语义上，接口表示更高层次的抽象，声明系统对外提供的服务。而抽象类则是各种具体类型的抽象。

抽象类与接口的区别




```
public interface USBInterface{ /* USB接口 */  
    public void transportData();  
}  
  
public abstract class Printer implements USBInterface{  
    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();  
}  
  
public class DigitalCamera extends Camera implements  
USBInterface{...}  
  
public class FilmCamera extends Camera{...} /* 胶片照相机 */
```

案例分析：接口声明、实现、使用、接口回调

- 问题域：
 - 编写程序模拟动物园里饲养员给各种动物喂养各种不同食物的过程。

分析设计（1）：

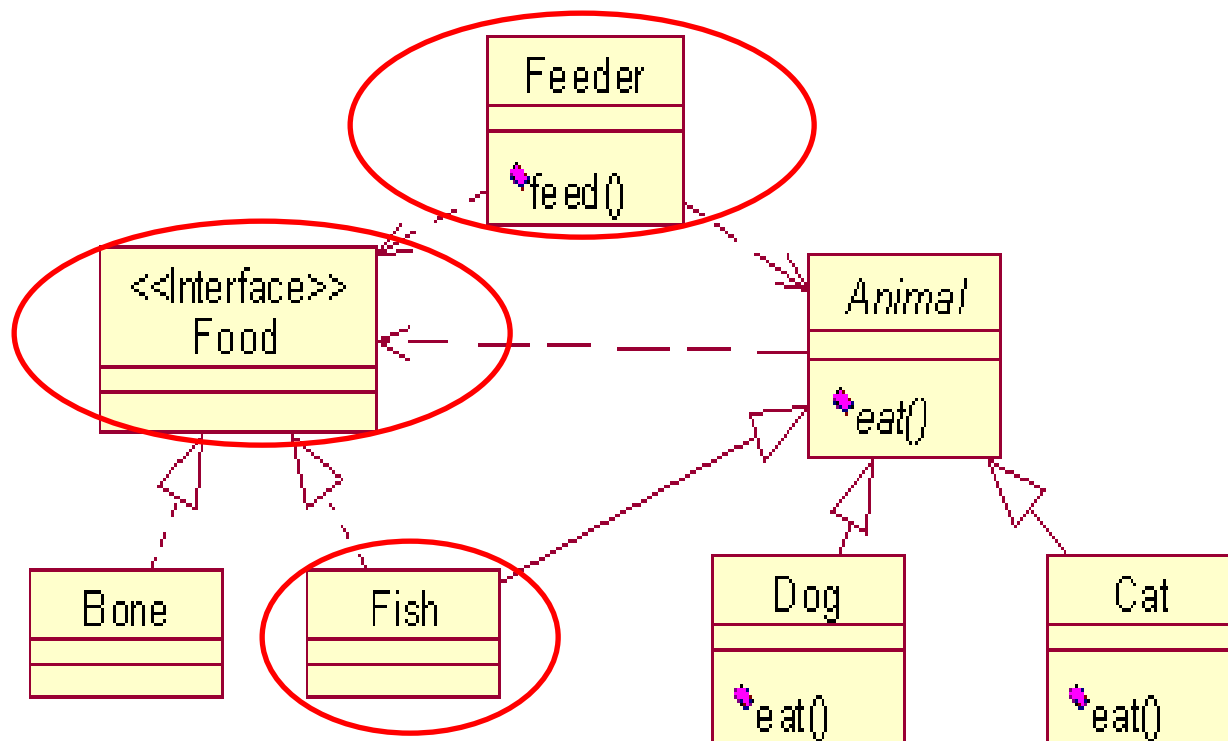
- 在这个动物园里，涉及的对象有
 - 饲养员
 - 各种不同动物
 - 以及各种不同的食物。
- 当饲养员给动物喂食时，动物发出欢快的叫声。
- 很容易抽象出3个类Feeder、Animal和Food。
- 假设只考虑猫和狗，则由Animal类派生出Cat类和Dog类
- 由Food类可以进一步派生出其子类Bone、Fish。因为他们之间存在着明显的is-a关系。

分析设计（2）（Java中的类是单继承的）

- 鱼是一种食物，但实际上，鱼也是一种动物
- 如果将Animal定义为接口，则Animal中是不能定义成员变量和成员方法的，而Cat类和Dog类继承了Animal的很多属性和方法，这是冲突的。
- Food类中虽然也可能有变量但是数量比Animal少，所以我们可以考虑将Food定义为接口。此时可以说“鱼是一种动物，同时也是一种食物”。

用到的知识点

- 继承
- 多态
- 抽象类
- 接口



重点代码段

```
public interface Food{
```

```
public abstract class Animal{  
    public abstract void eat(Food food);  
}
```

```
public class Fish extends Animal  
implements Food{  
    public void eat(Food food){..... }  
}
```




```
public class Feeder{
```

```
    public void feed(Animal animal, Food  
food){  
        animal.eat(food);  
    }
```

```
} 2022/10/20
```

```
Feeder feeder=new  
Feeder();  
Cat cat=new Cat();  
Fish fish=new Fish();  
  
feeder.feed(cat,fish);
```

 com.buaa.zooSimulationEx

- ▷  Animal.java
- ▷  Bone.java
- ▷  Cat.java
- ▷  Dog.java
- ▷  Feeder.java
- ▷  Fish.java
- ▷  Food.java
- ▷  TestDemo.java

```
public interface Food {  
    public abstract String getName();  
}
```



```
public abstract class Animal {  
    private String name;  
  
    public Animal(String name) {  
        this.name = name;  
    }  
  
    public abstract void shout();  
    public abstract void eat(Food food);  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

```
public class Cat extends Animal{  
  
    public Cat(String name) {  
        super(name);  
    }  
  
    public void shout() {  
        System.out.println("喵呜.....");  
    }  
  
    public void eat(Food food) {  
        System.out.println(getName() + "正在吃着香喷喷的" + food.getName());  
    }  
  
}
```

```
public class Dog extends Animal {  
  
    public Dog(String name) {  
        super(name);  
    }  
  
    @Override  
    public void shout() {  
        System.out.println("汪汪汪.....");  
    }  
  
    @Override  
    public void eat(Food food) {  
        System.out.println(getName() + "正在啃着香喷喷的" + food.getName());  
    }  
}
```

```
public class Fish extends Animal implements Food{
```

```
    public Fish(String name) {  
        super(name);  
    }
```

```
    @Override  
    public void shout() {  
  
    }
```

```
    @Override  
    public void eat(Food food) {  
  
    }
```

```
}
```

```
public class Bone implements Food{
```

```
    @Override  
    public String getName() {  
        return "bone";  
    }
```

```
}
```

```
public class Feeder {  
    private String name;  
    public Feeder(String name) {  
        this.name = name;  
    }  
    public void speak() {  
        System.out.println("欢迎来到动物园！");  
        System.out.println("我是饲养员 "+getName());  
    }  
    public void feed(Animal a, Food food) {  
        a.eat(food);  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

```
public class TestDemo {  
  
    public static void main(String[] args) {  
        Feeder feeder = new Feeder("花花");  
        feeder.speak();  
  
        Dog dog = new Dog("小布丁");  
        Food food = new Bone();  
        feeder.feed(dog, food);  
  
        Cat cat = new Cat("小猫崽");  
        food = new Fish("黄花鱼");  
        feeder.feed(cat, food);  
    }  
}
```

接口的意义

- 接口规定了类“做什么”，而不关心“怎样做”，这样既规范了类行为，又给了实现接口的类很大自由度和灵活性，以适应不断发展变化的客观现实，接口在J2ME和JavaEE中大量使用。
- 所以，接口不仅是一种规范，而是Java编程思想的体现。

接口表示一种能力

- “做这项工作需要一个钳工（木匠/程序员）”

钳工是一种“能力”，不关心具体是谁

- 接口是一种能力 **体现在接口的方法上**

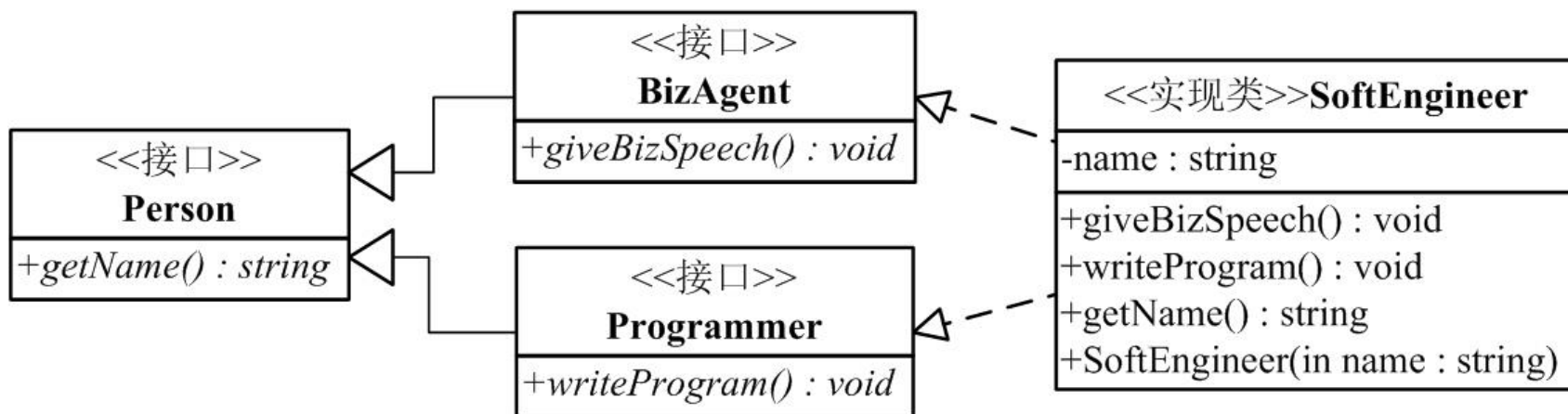
- 面向接口编程

**程序
设计
时**

关心实现类有何能力，而不关心实现细节

面向接口的约定而不考虑接口的具体实现

面向接口编程应用



具备讲解业务的能力

定义Programmer接口
定义BizAgent接口

具备编码的能力

编写SoftEngineer类







实现两个接口

编写测试类

案例分析:接口规定了类 “做什么”，而不关心 “怎样做”

- 设计实现发声接口，通过该接口可以播放，调节音频
 - ① 发出声音
 - **public void playSound();**
 - ② 降低声音
 - **public void decreaseVolume();**
 - ③ 停止声音
 - **public void stopSound();**
- 收音机，随身听和手机可以发出声音
- SampleDisplay类可以实现播放的功能
- 测试类

 com.buaa.soundControlEx

- ▷  MobilePhone.java
- ▷  Radio.java
- ▷  SampleDisplay.java
- ▷  Soundable.java
- ▷  TestDemo.java
- ▷  Walkman.java

```
public interface Soundable {
```

```
// 发出声音
```

```
public void playSound() ;
```

```
// 降低声音
```

```
public void decreaseVolume();
```

```
// 停止声音
```

```
public void stopSound();
```

```
}
```

```
public class MobilePhone implements Soundable{
```

```
    @Override  
    public void playSound() {  
        System.out.println("手机发出来电铃声：叮当、叮当");  
    }
```

```
    @Override  
    public void decreaseVolume() {  
        System.out.println("降低手机音量");  
    }
```

```
    @Override  
    public void stopSound() {  
        System.out.println("关闭手机");  
    }
```

```
public class Radio implements Soundable{
```

```
    @Override
```

```
    public void playSound() {
```

```
        System.out.println("收音机播放广播：中央人民广播电视台。");
```

```
    }
```

```
    @Override
```

```
    public void decreaseVolume() {
```

```
        System.out.println("降低收音机音量。");
```

```
    }
```

```
    @Override
```

```
    public void stopSound() {
```

```
        System.out.println("关闭收音机。");
```

```
    }
```

```
}
```

```
public class Walkman implements Soundable{
```

```
    @Override  
    public void playSound() {  
        System.out.println("随身听发出音乐");  
    }
```

```
    @Override  
    public void decreaseVolume() {  
        System.out.println("降低随身听音量");  
    }
```

```
    @Override  
    public void stopSound() {  
        System.out.println("关闭随身听");  
    }
```


接口用做参数，接口回调

```
class SampleDisplay {  
    public void display(Soundable soundable) {  
        soundable.playSound();  
        soundable.decreaseVolume();  
        soundable.stopSound();  
    }  
}
```

```
package com.buaa.soundControlEx;
import java.util.Scanner;
public class TestDemo {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.println("你想听什么? 请输入: ");
        System.out.println("0-收音机 1-随身听 2-手机");
        int choice;
        choice = in.nextInt();
        SampleDisplay sampledisplay = new SampleDisplay();
        if (choice == 0)
            sampledisplay.display(new Radio());
        else if(choice == 1)
            sampledisplay.display(new Walkman());
        else if(choice == 2)
            sampledisplay.display(new MobilePhone());
        else
            System.out.println("haha, 你输错了! ");
        in.close();
    }
}
```

Native关键字

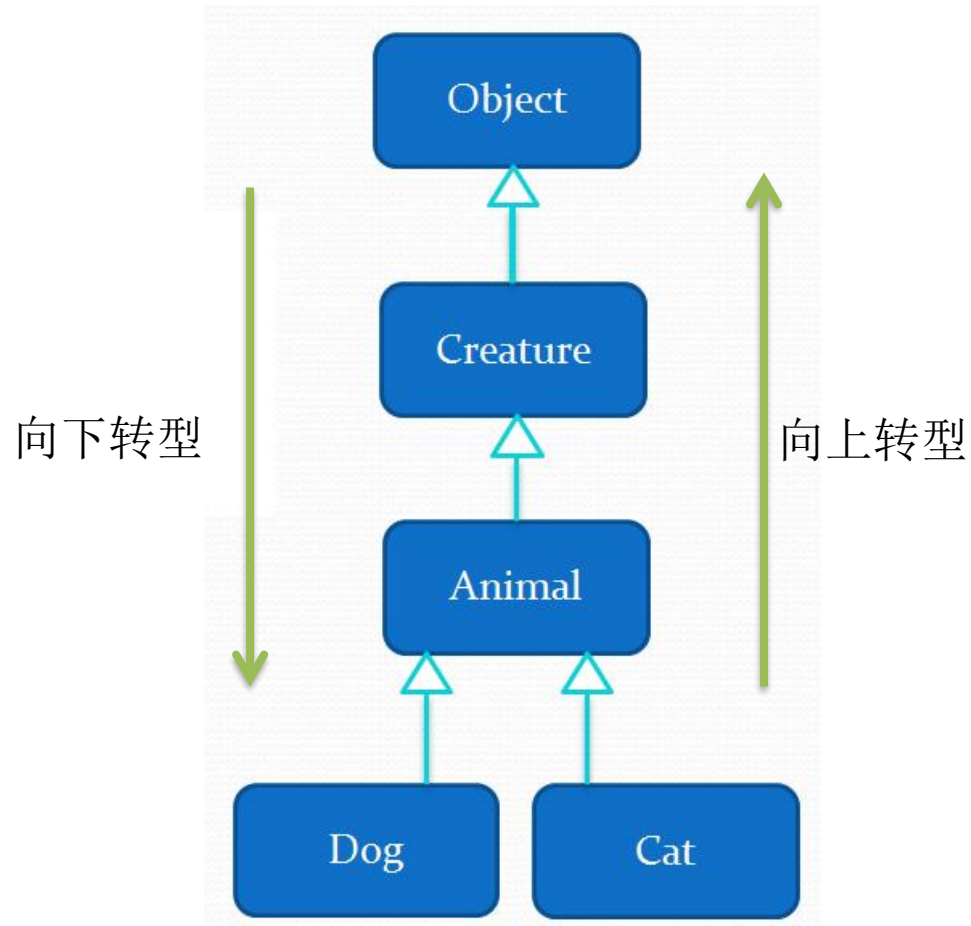
- java是跨平台的语言，既然是跨了平台，所付出的代价就是牺牲一些对底层的控制，而java要实现对底层的控制，就需要一些其他语言的帮助，这个就是native的作用。
- Native用来声明一个方法是由机器相关的语言（如C/C++语言）实现的。通常，native方法用于一些比较消耗资源的方法，该方法用c或其他语言编写，可以提高速度。
- native 定义符说明该方法是一个使用本地其他语言编写的非java类库的方法，它是调用的本地（也就是当前操作系统的方法或动态链接库）。最常见的就是c/c++封装的DLL里面的方法，这是java的 JNI技术。它在类中的声明和抽象方法一样没有方法体。

pcasting和downcasting

主讲老师：申雪萍



Upcasting和downcasting



父类和子类的对象之间转换

Java允许在父类和子类的对象之间进行转换：

1. 自动转换（向上映射）
2. 强制类型转换（向下映射）

向上转型 upcasting

- 向上转型：当有子类对象赋值给一个父类引用时，便是向上转型，多态本身就是向上转型的过程。
- 使用格式：
 - 父类类型 变量名 = new 子类类型();
 - 如：Person p = new Student();

测试类：向上映射upcasting

```
public class animalTest {  
    public static void main(String[] args) {  
        Animal aMouse = new Mouse();  
        Animal aGiraffa = new Giraffe();  
        Animal aLion = new Lion();  
        aMouse.eat();  
        aMouse.sleep();  
        aLion.eat();  
        aLion.sleep();  
        aGiraffa.eat();  
        aGiraffa.sleep();  
        System.out.println("-----");  
        Animal[] aArray = new Animal[3];  
        aArray[0] = aMouse;  
        aArray[1] = aGiraffa;  
        aArray[2] = aLion;  
        for (Animal i : aArray) {  
            i.eat();  
            i.sleep();  
        }  
    }  
}
```


上转型对象的使用

- 一. 上转型对象可以访问子类继承或隐藏的
成员变量，也可以调用子类继承的方法或
子类重写的实例方法。
- 二. 如果子类重写了父类的某个实例方法后，
当用上转型对象调用这个实例方法时一定
是调用了子类重写的实例方法。
- 三. 上转型对象不能操作子类新增的成员变
量；不能调用子类新增的方法。

- 向下转型(映射): 一个已经向上转型的子类对象可以使用强制类型转换的格式, 将父类引用转为子类引用, 这个过程是向下转型。
- 使用格式:
- 子类类型 变量名 = (子类类型) 父类类型的变量;
 - 如: `Person p = new Student();`
`Student stu = (Student) p`
- 如果是直接创建父类对象, 是无法向下转型的! , 会产生运行时异常
 - 如: `Person p = new Peron();`
`Student stu = (Student) p`

对象的向上映射和向下映射

- 对象的向上映射总是安全的，可靠的。
- 对象的向下映射就不一定了，有时可以，有时不可以，*如果不可以转的话，程序是不会报语法错误的，发生的是一个运行时异常。*
- *怎么解决这样的问题，让我们避免这个运行时异常呢？*

instanceof操作符

- instanceof操作符用于判断一个引用类型所引用的对象是否是一个类的实例。instanceof运算符是Java独有的双目运算符
- instanceof操作符左边的操作元**是一个引用类型的对象（可以是null）**，右边的操作元是一个类名或接口名。
- 形式如下：
obj instanceof ClassName
或者
obj instanceof InterfaceName

instanceof操作符

```
Fish fish=new Fish();
```

```
//XXX表示一个类名或接口名
```

```
System.out.println(fish instanceof XXX);
```

- 当“XXX”是以下值时，instanceof表达式的值为**true**:
 - Fish类。
 - Fish类的直接或间接父类。
 - Fish类实现的接口。

instanceof 操作符

```
Fish fish=new Fish();
```

```
System.out.println(fish instanceof Fish); //打印true
```

```
System.out.println(fish instanceof Animal); //打印true
```

```
System.out.println(fish instanceof Object); //打印true
```

```
System.out.println(fish instanceof Food); //打印true
```

```

class ASuper {
    String s = "class:A";
}

class BSub extends ASuper {    //继承关系
    String s = "class:B";    //变量隐藏
}

```

```

public class TypeV {
    public static void main(String args[]) {
        BSub b1,b3;
        ASuper a1, a2, a3;
        BSub b2 = new BSub();////
        a1 = b2;    //向上映射, 自动转换
        a2 = b2;    //向上映射, 自动转换
        System.out.println(a1.s);////
        System.out.println(a2.s);////
        b1 = (BSub) a1;    //向下映射, 强制转换
        System.out.println(b1.s);
        a3=new ASuper();////////
        // b3=(BSub)a3;
        if(a3 instanceof BSub)//instanceof的用法
            b3=(BSub)a3;
        else
            System.out.println("can not be transfered!");
    }
}

```

```

class:A
class:A
class:B
can not be transfered!

```

示例代码

```
import java.util.ArrayList;
```

```
import java.util.Vector;
```

对象是 `java.util.ArrayList` 类的实例

```
public class Main {
```

```
    public static void main(String[] args) {
```

```
        Object testObject = new ArrayList();
```

```
        displayObjectClass(testObject);
```

```
    }
```

```
    public static void displayObjectClass(Object o) {
```

```
        if (o instanceof Vector)
```

```
            System.out.println("对象是 java.util.Vector 类的实例");
```

```
        else if (o instanceof ArrayList)
```

```
            System.out.println("对象是 java.util.ArrayList 类的实例");
```

```
        else
```

```
            System.out.println("对象是 " + o.getClass() + " 类的实例");
```

```
    }
```

```
}
```



```
//Teacher和Student继承Person
//Object>String
//Object>Person>Teacher
//Object>Person>Student
Object object = new Student();
    System.out.println(object instanceof Student);//true
    System.out.println(object instanceof Person);//true
    System.out.println(object instanceof Object);//true
    System.out.println(object instanceof Teacher);//false
    System.out.println(object instanceof String);//false
    System.out.println("=====");
    Person person = new Student();
    System.out.println(person instanceof Student);//true
    System.out.println(person instanceof Person);//true
    System.out.println(person instanceof Object);//true
    System.out.println(person instanceof Teacher);//false
//        System.out.println(person instanceof String);//编译错误
    System.out.println("=====");
    Student student = new Student();
    System.out.println(student instanceof Student);//true
    System.out.println(student instanceof Person);//true
    System.out.println(student instanceof Object);//true
//        System.out.println(student instanceof Teacher);//编译错误
//        System.out.println(student instanceof String);//编译错误
```

小结

- instanceof是Java中的二元运算符
- 表达式 **obj instanceof T**, instanceof 运算符的 **obj** 操作数的类型必须是引用类型或空类型; 否则, 会发生编译时错误。
- 如果 **obj** 强制转换为 **T** 时发生编译错误, 则关系表达式的 **instanceof** 同样会产生编译时错误。
- 如果 **obj** 不为 **null** 并且 **(T) obj** 不抛 **ClassCastException** 异常则该表达式值为 **true** , 否则值为 **false** 。

示例代码：通过向下转型，使用子类特有功能。

```
//描述动物类，并抽取共性eat方法
abstract class Animal {
    abstract void eat();
}

// 描述狗类，继承动物类，重写eat方法，增加lookHome方法
class Dog extends Animal {
    void eat() {
        System.out.println("啃骨头");
    }

    void lookHome() {
        System.out.println("看家");
    }
}
```

示例代码

```
// 描述猫类，继承动物类，重写eat方法，增加catchMouse方法
class Cat extends Animal {
    void eat() {
        System.out.println("吃鱼");
    }

    void catchMouse() {
        System.out.println("抓老鼠");
    }
}
```

示例代码： instanceof操作符

```
public class Test {  
    public static void main(String[] args) {  
        Animal a = new Dog(); // 多态形式，创建一个狗对象  
        a.eat(); // 调用对象中的方法，会执行狗类中的eat方法  
        // a.lookHome(); // 使用Dog类特有的方法，需要向下转型，不能直接使用  
        // 为了使用狗类的lookHome方法，需要向下转型  
        // 向下转型过程中，可能会发生类型转换的错误，即ClassCastException异常  
        // 那么，在转之前需要做健壮性判断  
        if (!(a instanceof Dog)) { // 判断当前对象是否是Dog类型  
            System.out.println("类型不匹配，不能转换");  
            return;  
        }  
        Dog d = (Dog) a; // 向下转型  
        d.lookHome(); // 调用狗类的lookHome方法  
    }  
}
```

示例代码： instanceof操作符

```
package com.buaa.test;

public interface Electronics{

}
```

```
package com.buaa.test;

public class Thinkpad implements Electronics {
    // Thinkpad引导方法
    public void boot() {
        System.out.println("welcome,I am Thinkpad");
    }
    // 使用Thinkpad编程
    public void program() {
        System.out.println("using Thinkpad program");
    }
}
```

示例代码： instanceof操作符

```
package com.buaa.test;

public class Mouse implements Electronics {
    // 鼠标移动
    public void move() {
        System.out.println("move the mouse");
    }
    // 鼠标点击
    public void onClick() {
        System.out.println("a click of the mouse");
    }
}
```

示例代码： instanceof 操作符

```
package com.buaa.test;

public class Keyboard implements Electronics {
    // 使用键盘输入
    public void input() {
        System.out.println("using Keyboard input");
    }
}
```


示例代码： instanceof 操作符

```
package com.buaa.test;|
import java.util.ArrayList;
import java.util.List;
public class ShopCar {
    private List<Electronics> mlist = new ArrayList<Electronics>();
    public void add(Electronics electronics) {
        mlist.add(electronics);
    }
    public int getSize() {
        return mlist.size();
    }
    public Electronics getListItem(int position) {
        return mlist.get(position);
    }
}
```

```
package com.buaa.test;
public class Test {
    public static void main(String[] args) {
        // 添加进购物车
        ShopCar shopcar = new ShopCar();
        shopcar.add(new Thinkpad());
        shopcar.add(new Mouse());
        shopcar.add(new Keyboard());
        // 获取大小
        System.out.println("购物车存放的电子产品数量为 —> " + shopcar.getSize());
        for(int i=0;i<shopcar.getSize();i++){
            if(shopcar.getListItem(i) instanceof Thinkpad){
                Thinkpad thinkpad = (Thinkpad) shopcar.getListItem(i);
                thinkpad.boot();
                thinkpad.program();
                System.out.println("-----");
            }
            else if (shopcar.getListItem(i) instanceof Mouse){
                Mouse mouse = (Mouse) shopcar.getListItem(i);
                mouse.move();
                mouse.onClick();
                System.out.println("-----");
            }
            else if (shopcar.getListItem(i) instanceof Keyboard){
                Keyboard keyboard = (Keyboard) shopcar.getListItem(i);
                keyboard.input();
                System.out.println("-----");
            }
        }
    }
}
```

示例代码： instanceof 操作符

购物车存放的电子产品数量为 —> 3

```
welcome,I am Thinkpad  
using Thinkpad program
```

```
move the mouse  
a click of the mouse
```

```
using Keyboard input
```

- 1、什么时候使用向上转型：
 - 如果不需要使用子类特有功能时，使用向上转型，采用动态联编，可以给开发人员带来实际的价值。
- 2、什么时候使用向下转型
 - 当要使用子类特有功能时，就需要使用向下转型，调用子类特有的功能。

- 3、向下转型的好处：可以使用子类特有功能。
- 4、向下转型的弊端：向下转型时容易发生 `ClassCastException` 类型转换异常。在转换之前必须做类型判断。
如： `if(!a instanceof Dog) {···}`

小结

- 接口的必要性（将接口用作**API**）
- 定义接口
- 实现接口
- 将接口用作类型、接口回调（使用接口）
- 接口的进化（通过接口的继承完成）
- 面向接口的编程