

 public interface 接口名称 {     // 抽象方法

    // 默认方法

    // 静态方法

    // 私有方法

}

**【接口、多态】**

**内容**

接口

三大特征——多态

引用类型转换

**目标**

写出定义接口的格式

写出实现接口的格式

说出接口中成员的特点

能够说出使用多态的前提条件

理解多态的向上转型

理解多态的向下转型

能够完成笔记本电脑案例（方法参数为接口）

**第一章 接口**

**1.1 概述**

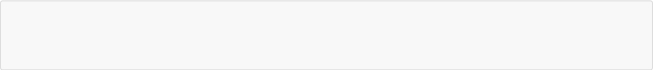
接口，是Java语言中一种引用类型，是方法的集合，如果说类的内部封装了成员变量、构造方法和成员方法，那么 接口的内部主要就是**封装了方法**，包含抽象方法（JDK 7及以前），默认方法和静态方法（JDK 8），私有方法 （JDK 9）。

接口的定义，它与定义类方式相似，但是使用 interface 关键字。它也会被编译成.class文件，但一定要明确它并 不是类，而是另外一种引用数据类型。

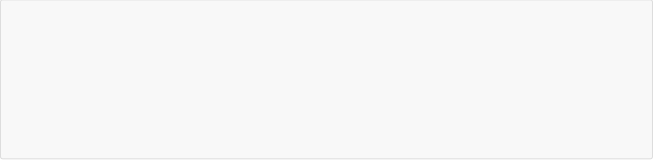
引用数据类型：数组，类，接口。

接口的使用，它不能创建对象，但是可以被实现（ implements ，类似于被继承）。一个实现接口的类（可以看做 是接口的子类），需要实现接口中所有的抽象方法，创建该类对象，就可以调用方法了，否则它必须是一个抽象 类。

**1.2 定义格式**



 public interface InterFaceName {     public abstract void method(); }



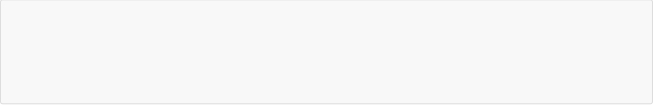
 public interface InterFaceName  {     public default  void method() {         // 执行语句

    }

    public static void method2() {         // 执行语句

    }

}



 public interface InterFaceName  {     private void method() {         // 执行语句

    }

}



**含有抽象方法**

抽象方法：使用 abstract 关键字修饰，可以省略，没有方法体。该方法供子类实现使用。 代码如下：

**含有默认方法和静态方法**

默认方法：使用 default 修饰，不可省略，供子类调用或者子类重写。 静态方法：使用 static 修饰，供接口直接调用。

代码如下：

**含有私有方法和私有静态方法**

私有方法：使用 private 修饰，供接口中的默认方法或者静态方法调用。 代码如下：

**1.3 基本的实现**

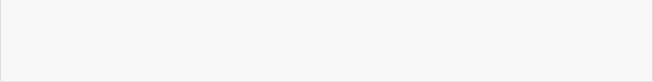
**实现的概述**

类与接口的关系为实现关系，即**类实现接口**，该类可以称为接口的实现类，也可以称为接口的子类。实现的动作类 似继承，格式相仿，只是关键字不同，实现使用 implements 关键字。

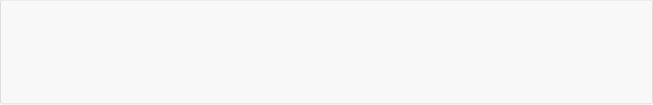
非抽象子类实现接口：

1. 必须重写接口中所有抽象方法。
2. 继承了接口的默认方法，即可以直接调用，也可以重写。

实现格式：

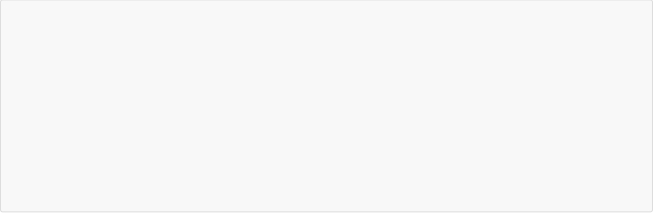


 class 类名 implements 接口名 {     // 重写接口中抽象方法【必须】     // 重写接口中默认方法【可选】 }



 public interface LiveAble {     // 定义抽象方法

    public abstract void eat();     public abstract void sleep(); }



 public class Animal implements LiveAble {     @Override

    public void eat() {

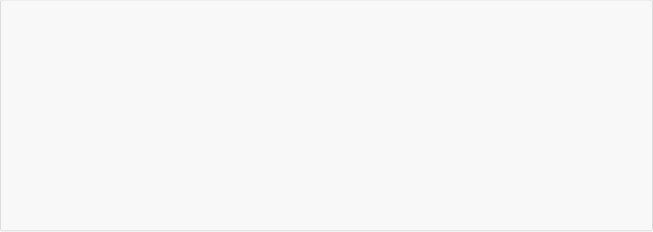
        System.out.println ("吃东西");     }

    @Override

    public void sleep() {

        System.out.println ("晚上睡");     }

}



 public class InterfaceDemo {

    public static void main(String[] args) {         // 创建子类对象

        Animal a = new Animal();

        // 调用实现后的方法

        a.eat();

        a.sleep();

    }

}

输出结果：

吃东西

晚上睡

**抽象方法的使用**

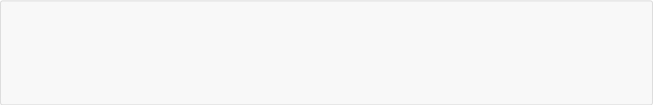
必须全部实现，代码如下： 定义接口：

定义实现类：

定义测试类：

**默认方法的使用**

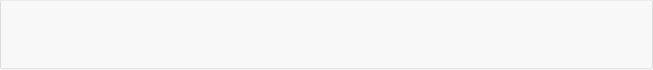
可以继承，可以重写，二选一，但是只能通过实现类的对象来调用。 1. 继承默认方法，代码如下：



 public interface LiveAble {

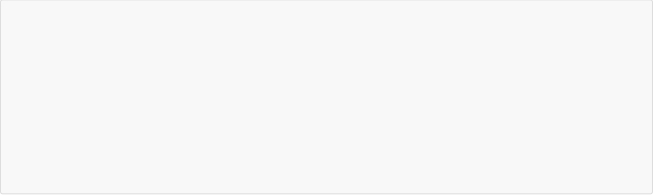
    public default  void fly(){         System.out.println ("天上飞");     }

}



 public class Animal implements LiveAble {     // 继承，什么都不用写，直接调用

}



 public class InterfaceDemo {

    public static void main(String[] args) {         // 创建子类对象

        Animal a = new Animal();

        // 调用默认方法

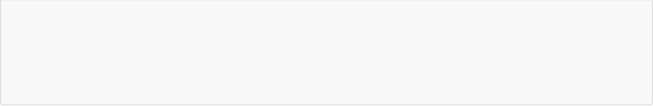
        a.fly();

    }

}

输出结果：

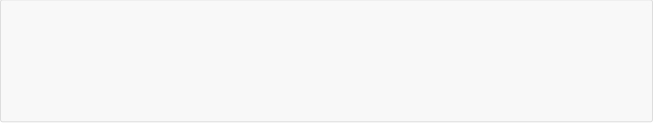
天上飞



 public interface LiveAble {

    public default  void fly(){         System.out.println ("天上飞");     }

}



 public class Animal implements  LiveAble {     @Override

    public void fly() {

        System.out.println ("自由自在的飞");     }

}

定义接口：

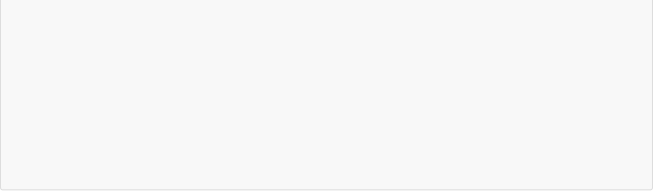
定义实现类：

定义测试类：

2. 重写默认方法，代码如下： 定义接口：

定义实现类：

定义测试类：



 public class InterfaceDemo {

    public static void main(String[] args) {         // 创建子类对象

        Animal a = new Animal();

        // 调用重写方法

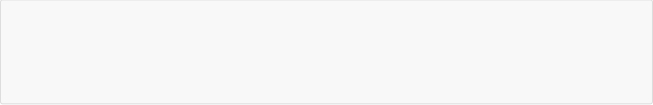
        a.fly();

    }

}

输出结果：

自由自在的飞

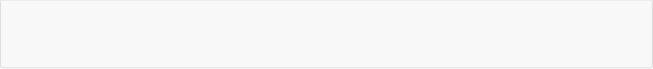


 public interface LiveAble {

    public static void run(){

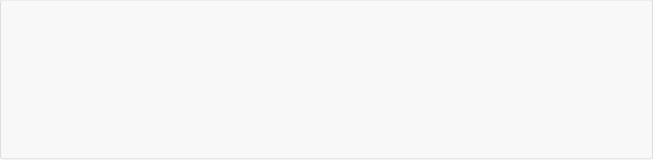
        System.out.println ("跑起来~~~");     }

}



 public class Animal implements LiveAble {     // 无法重写静态方法

}



 public class InterfaceDemo {

    public static void main(String[] args) {

        // Animal.run(); // 【错误】无法继承方法,也无法调用         LiveAble.run(); //

    }

}

输出结果：

跑起来~~~

**静态方法的使用**

静态与.class 文件相关，只能使用接口名调用，不可以通过实现类的类名或者实现类的对象调用，代码如下： 定义接口：

定义实现类：

定义测试类：

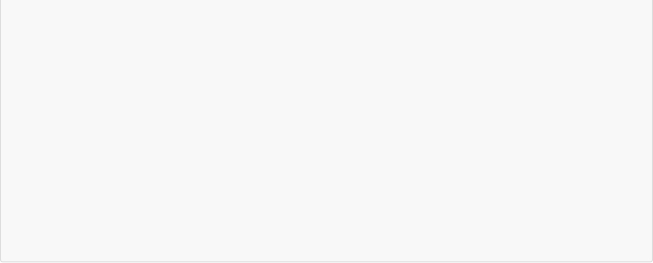
**私有方法的使用**

私有方法：只有默认方法可以调用。

私有静态方法：默认方法和静态方法可以调用。

如果一个接口中有多个默认方法，并且方法中有重复的内容，那么可以抽取出来，封装到私有方法中，供默认方法 去调用。从设计的角度讲，私有的方法是对默认方法和静态方法的辅助。同学们在已学技术的基础上，可以自行测 试。

定义接口：



 public interface LiveAble {

    default void func(){

        func1();

        func2();

    }

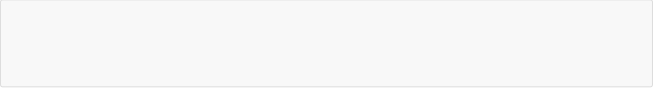
    private void func1(){

        System.out.println ("跑起来~~~");     }

    private void func2(){

        System.out.println ("跑起来~~~");     }

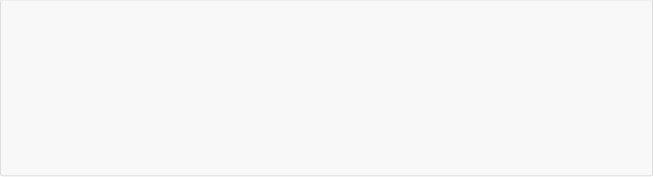
}



 class 类名 [extends 父类名] implements 接口名1,接口名2,接口名3... {     // 重写接口中抽象方法【必须】

    // 重写接口中默认方法【不重名时可选】

}



 interface A {

    public abstract void showA();     public abstract void show(); }

interface B {

    public abstract void showB();     public abstract void show(); }



**1.4 接口的多实现**

之前学过，在继承体系中，一个类只能继承一个父类。而对于接口而言，一个类是可以实现多个接口的，这叫做接 口的**多实现**。并且，一个类能继承一个父类，同时实现多个接口。

实现格式：

[ ]： 表示可选操作。

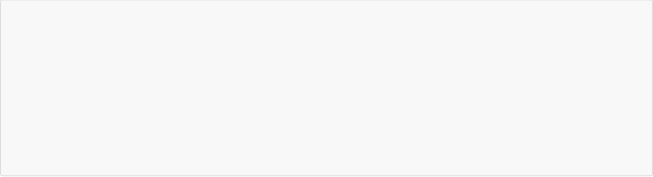
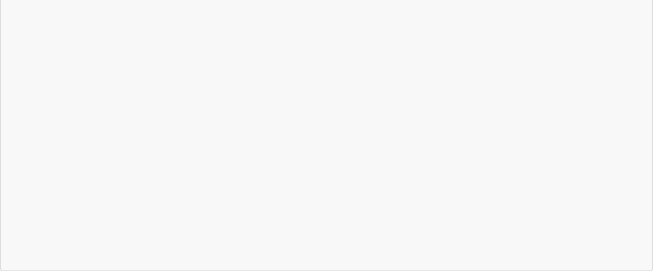
**抽象方法**

接口中，有多个抽象方法时，实现类必须重写所有抽象方法**。如果抽象方法有重名的，只需要重写一次。**代码如 下：

定义多个接口：

定义实现类：

public class C implements A,B{

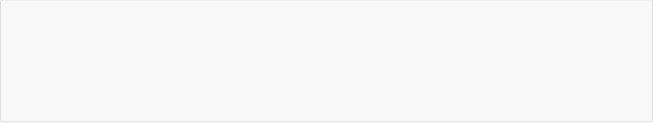


 interface A {

    public default  void methodA(){}     public default  void method(){} }

interface B {

    public default  void methodB(){}     public default  void method(){} }



 public class C implements A,B{

    @Override

    public void method() {

        System.out.println ("method");     }

}

    @Override

    public void showA() {

        System.out.println ("showA");

    }

    @Override

    public void showB() {

        System.out.println ("showB");

    }

    @Override

    public void show() {

        System.out.println ("show");

    }

}

**默认方法**

接口中，有多个默认方法时，实现类都可继承使用。**如果默认方法有重名的，必须重写一次。**代码如下： 定义多个接口：

定义实现类：

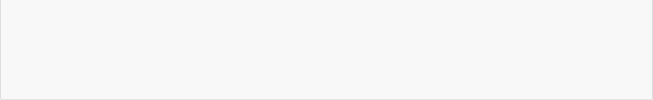
**静态方法**

接口中，存在同名的静态方法并不会冲突，原因是只能通过各自接口名访问静态方法。

**优先级的问题**

当一个类，既继承一个父类，又实现若干个接口时，父类中的成员方法与接口中的默认方法重名，子类就近选择执 行父类的成员方法。代码如下：

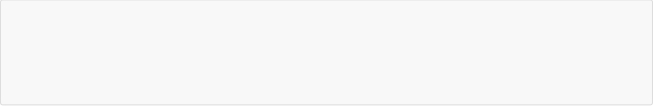
定义接口：



 interface A {

    public default  void methodA(){         System.out.println ("AAAAAAAAAAAA");     }

}

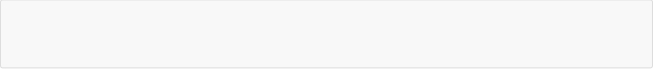


 class D {

    public void methodA(){

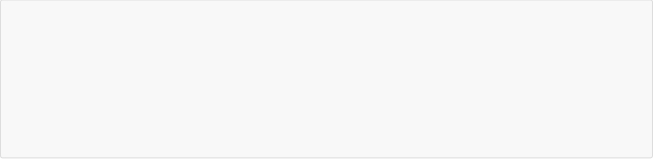
        System.out.println ("DDDDDDDDDDDD");     }

}



 class C extends D implements A {     // 未重写methodA方法

}



 public class Test {

    public static void main(String[] args) {         C c = new C();

        c.methodA ();

    }

}

输出结果:

DDDDDDDDDDDD



定义父类：

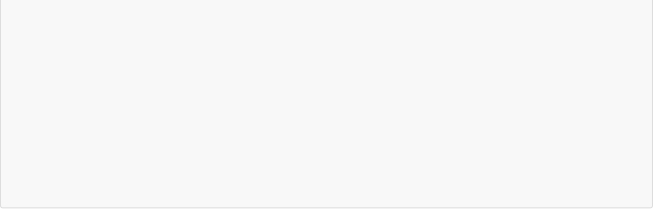
定义子类：

定义测试类：

**1.5 接口的多继承【了解】**

一个接口能继承另一个或者多个接口，这和类之间的继承比较相似。接口的继承使用 extends 关键字，子接口继 承父接口的方法。**如果父接口中的默认方法有重名的，那么子接口需要重写一次。**代码如下：

定义父接口：



 interface A {

    public default  void method(){

        System.out.println ("AAAAAAAAAAAAAAAAAAA");     }

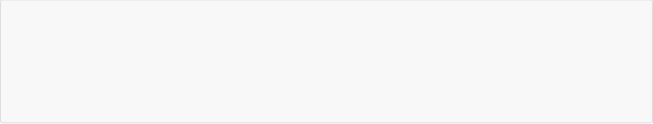
}

interface B {

    public default  void method(){

        System.out.println ("BBBBBBBBBBBBBBBBBBB");     }

}



 interface D extends A,B{

    @Override

    public default  void method() {

        System.out.println ("DDDDDDDDDDDDDD");     }

}

定义子接口：

小贴士：

子接口重写默认方法时，default关键字可以保留。

子类重写默认方法时，default关键字不可以保留。

**1.6 其他成员特点**

接口中，无法定义成员变量，但是可以定义常量，其值不可以改变，默认使用public static ﬁnal修饰。 接口中，没有构造方法，不能创建对象。

接口中，没有静态代码块。

**第二章 多态**

**2.1 概述**

**引入**

多态是继封装、继承之后，面向对象的第三大特性。

生活中，比如跑的动作，小猫、小狗和大象，跑起来是不一样的。再比如飞的动作，昆虫、鸟类和飞机，飞起来也 是不一样的。可见，同一行为，通过不同的事物，可以体现出来的不同的形态。多态，描述的就是这样的状态。

**定义**

**多态**： 是指同一行为，具有多个不同表现形式。

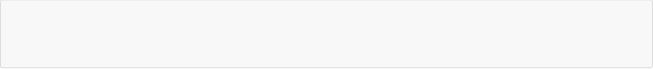
**前提【重点】**



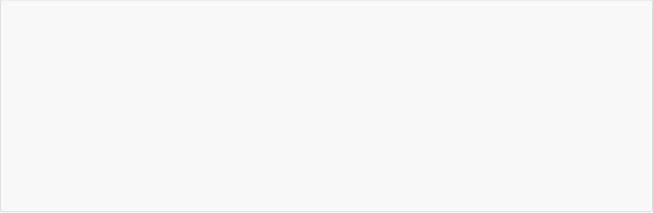
 父类类型 变量名 = new 子类对象； 变量名.方法名();



 Fu f = new Zi(); f.method();



 public abstract class Animal {      public abstract void eat();  }



 class Cat extends Animal {

    public void eat() {

        System.out.println ("吃鱼");      }

}

class Dog extends  Animal {

    public void eat() {

        System.out.println ("吃骨头");      }

}

1. 继承或者实现【二选一】
2. 方法的重写【意义体现：不重写，无意义】
3. 父类引用指向子类对象【格式体现】

**2.2 多态的体现**

多态体现的格式：

父类类型：指子类对象继承的父类类型，或者实现的父接口类型。

代码如下：

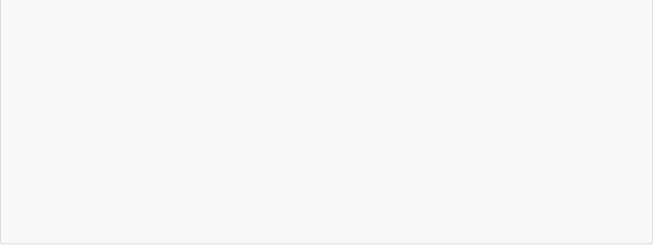
**当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误；如果有，执行的是子类重写 后方法。**

代码如下：

定义父类：

定义子类：

定义测试类：



 public class Test {

    public static void main(String[] args) {         // 多态形式，创建对象

        Animal a1 = new Cat();

        // 调用的是 Cat的 eat

        a1.eat();

        // 多态形式，创建对象

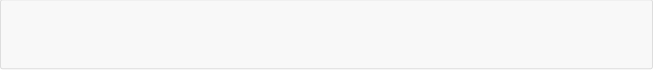
        Animal a2 = new Dog();

        // 调用的是 Dog 的 eat

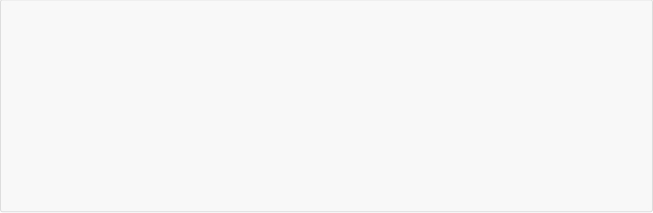
        a2.eat();

    }

}



 public abstract class Animal {      public abstract void eat();  }



 class Cat extends Animal {

    public void eat() {

        System.out.println ("吃鱼");      }

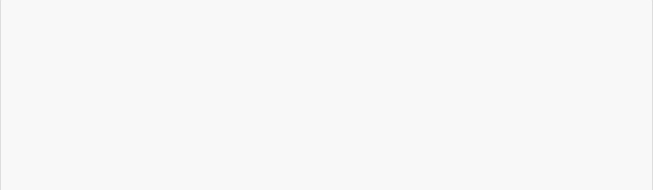
}

class Dog extends  Animal {

    public void eat() {

        System.out.println ("吃骨头");      }

}



**2.3 多态的好处**

实际开发的过程中，父类类型作为方法形式参数，传递子类对象给方法，进行方法的调用，更能体现出多态的扩展 性与便利。代码如下：

定义父类：

定义子类：

定义测试类：

public class Test {

    public static void main(String[] args) {         // 多态形式，创建对象

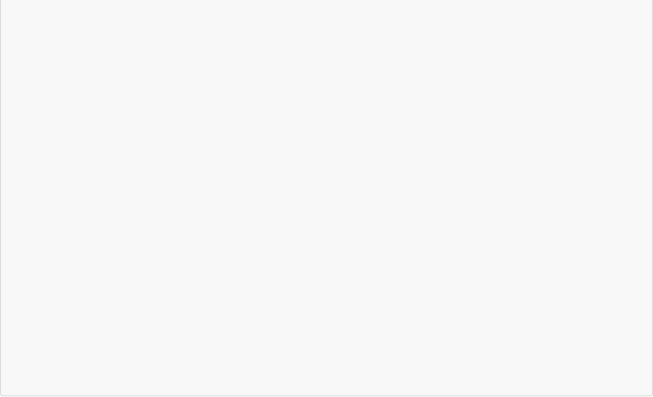
        Cat c = new Cat();

        Dog d = new Dog();

        // 调用showCatEat

        showCatEat(c);

        // 调用showDogEat



 父类类型  变量名 = new 子类类型(); 如：Animal a = new Cat();

        showDogEat(d);

        /\*

        以上两个方法, 均可以被showAnimalEat(Animal a)方法所替代

        而执行效果一致

        \*/

        showAnimalEat(c);

        showAnimalEat(d);

    }

    public static void showCatEat  (Cat c){

        c.eat();

    }

    public static void showDogEat  (Dog d){

        d.eat();

    }

    public static void showAnimalEat  (Animal a){

        a.eat();

    }

}

由于多态特性的支持，showAnimalEat方法的Animal 类型，是Cat和Dog的父类类型，父类类型接收子类对象，当 然可以把Cat对象和Dog对象，传递给方法。

当eat方法执行时，多态规定，执行的是子类重写的方法，那么效果自然与showCatEat、showDogEat方法一致， 所以showAnimalEat完全可以替代以上两方法。

不仅仅是替代，在扩展性方面，无论之后再多的子类出现，我们都不需要编写showXxxEat方法了，直接使用 showAnimalEat都可以完成。

所以，多态的好处，体现在，可以使程序编写的更简单，并有良好的扩展。

**2.4 引用类型转换**

多态的转型分为向上转型与向下转型两种：

**向上转型**

**向上转型**：多态本身是子类类型向父类类型向上转换的过程，这个过程是默认的。

当父类引用指向一个子类对象时，便是向上转型。

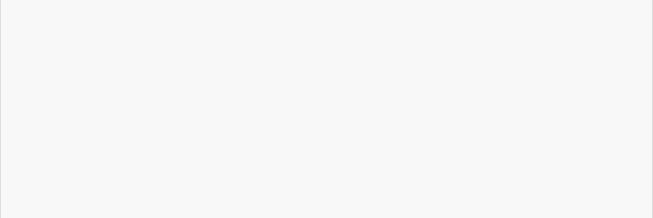
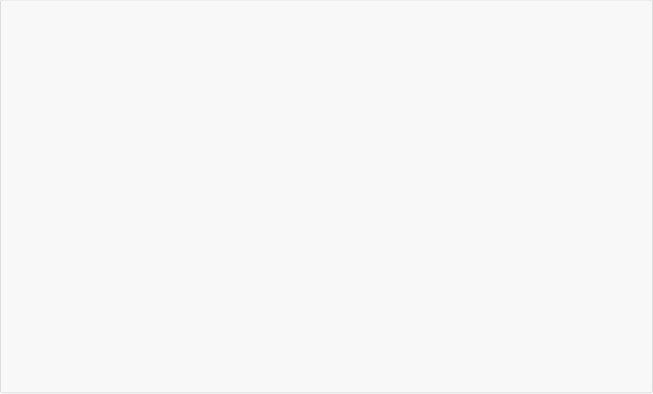
使用格式：

**向下转型**

**向下转型**：父类类型向子类类型向下转换的过程，这个过程是强制的。



 子类类型 变量名 = (子类类型) 父类变量名; 如:Cat c =(Cat) a;



一个已经向上转型的子类对象，将父类引用转为子类引用，可以使用强制类型转换的格式，便是向下转型。 使用格式：

**为什么要转型**

当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误。也就是说，**不能调用**子类拥 有，而父类没有的方法。编译都错误，更别说运行了。这也是多态给我们带来的一点"小麻烦"。所以，想要调用子 类特有的方法，必须做向下转型。

转型演示，代码如下：

定义类：

 abstract  class Animal {

    abstract  void eat();

}

class Cat extends  Animal {

    public void eat() {

        System.out.println ("吃鱼");

    }

    public void catchMouse () {

        System.out.println ("抓老鼠");

    }

}

class Dog extends  Animal {

    public void eat() {

        System.out.println ("吃骨头");

    }

    public void watchHouse () {

        System.out.println ("看家");

    }

}

定义测试类：

 public class Test {

    public static void main(String[] args) {

        // 向上转型

        Animal a = new Cat();

        a.eat();                // 调用的是 Cat 的 eat

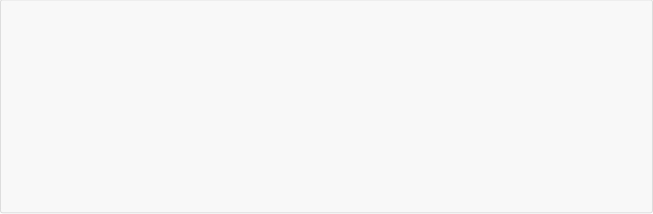
        // 向下转型

        Cat c = (Cat)a;

        c.catchMouse ();         // 调用的是 Cat的 catchMouse

    }

}



 public class Test {

    public static void main(String[] args) {

        // 向上转型

        Animal a = new Cat();

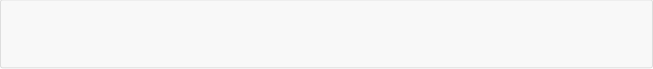
        a.eat();               // 调用的是 Cat的 eat

        // 向下转型

        Dog d = (Dog)a;

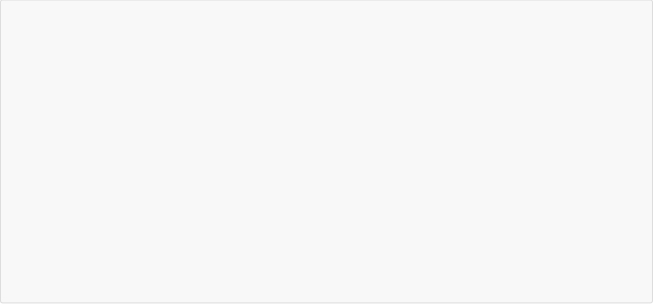
        d.watchHouse ();        // 调用的是 Dog 的 watchHouse 【运行报错】     }

}



 变量名 instanceof  数据类型

如果变量属于该数据类型，返回true。 如果变量不属于该数据类型，返回false。



 public class Test {

    public static void main(String[] args) {

        // 向上转型

        Animal a = new Cat();

        a.eat();               // 调用的是 Cat的 eat

        // 向下转型

        if (a instanceof Cat){

            Cat c = (Cat)a;

            c.catchMouse();        // 调用的是 Cat 的 catchMouse         } else if (a instanceof Dog){

            Dog d = (Dog)a;

            d.watchHouse();       // 调用的是 Dog的 watchHouse         }

    }

}

**转型的异常**

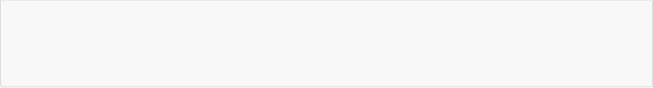
转型的过程中，一不小心就会遇到这样的问题，请看如下代码：

这段代码可以通过编译，但是运行时，却报出了 ClassCastException ，类型转换异常！这是因为，明明创建了 Cat类型对象，运行时，当然不能转换成Dog对象的。这两个类型并没有任何继承关系，不符合类型转换的定义。

为了避免ClassCastException 的发生，Java提供了 instanceof 关键字，给引用变量做类型的校验，格式如下：

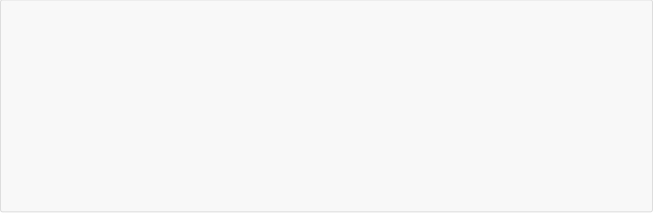
所以，转换前，我们最好先做一个判断，代码如下：

**第三章 接口多态的综合案例 3.1 笔记本电脑**



 interface USB {

    void open();// 开启功能     void close();// 关闭功能 }



 class Mouse implements  USB {

    public void open() {

        System.out.println ("鼠标开启，红灯闪一闪");     }

    public void close() {

        System.out.println ("鼠标关闭，红灯熄灭");     }

    public void click(){

        System.out.println ("鼠标单击");

    }

}

笔记本电脑（laptop）通常具备使用USB设备的功能。在生产时，笔记本都预留了可以插入USB设备的USB接口， 但具体是什么USB设备，笔记本厂商并不关心，只要符合USB规格的设备都可以。

定义USB接口，具备最基本的开启功能和关闭功能。鼠标和键盘要想能在电脑上使用，那么鼠标和键盘也必须遵守 USB规范，实现USB接口，否则鼠标和键盘的生产出来也无法使用。

**3.2 案例分析**

进行描述笔记本类，实现笔记本使用USB鼠标、USB键盘

USB接口，包含开启功能、关闭功能

笔记本类，包含运行功能、关机功能、使用USB设备功能

鼠标类，要实现USB接口，并具备点击的方法

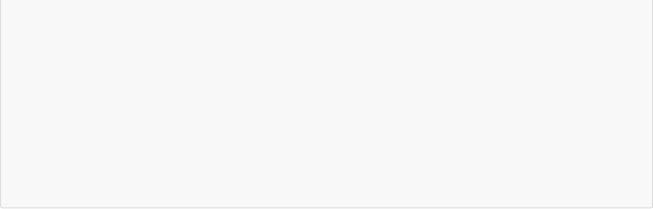
键盘类，要实现USB接口，具备敲击的方法

**3.3 案例实现**

定义USB接口：

定义鼠标类：

定义键盘类：



 class KeyBoard  implements USB {

    public void open() {

        System.out.println ("键盘开启，绿灯闪一闪");     }

    public void close() {

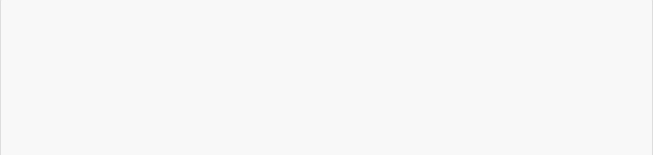
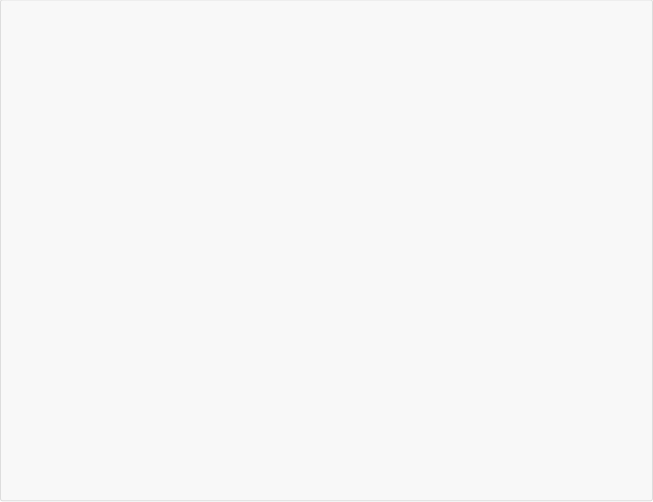
        System.out.println ("键盘关闭，绿灯熄灭");     }

    public void type(){

        System.out.println ("键盘打字");

    }

}



定义笔记本类：

 class Laptop {

    // 笔记本开启运行功能

    public void run() {

        System.out.println ("笔记本运行");

    }

    // 笔记本使用usb设备，这时当笔记本对象调用这个功能时，必须给其传递一个符合USB规则的USB设备     public void useUSB(USB usb) {

        // 判断是否有USB设备

        if (usb != null) {

            usb.open();

            // 类型转换,调用特有方法

            if(usb instanceof Mouse){

                Mouse m = （Mouse）usb；

                    m.click();

            }else if (usb instanceof KeyBoard){

                KeyBoard kb = (KeyBoard )usb;

                kb.type();

            }

            usb.close();

        }

    }

    public void shutDown () {

        System.out.println ("笔记本关闭");

    }

}

测试类，代码如下：

 public class Test {

    public static void main(String[] args) {

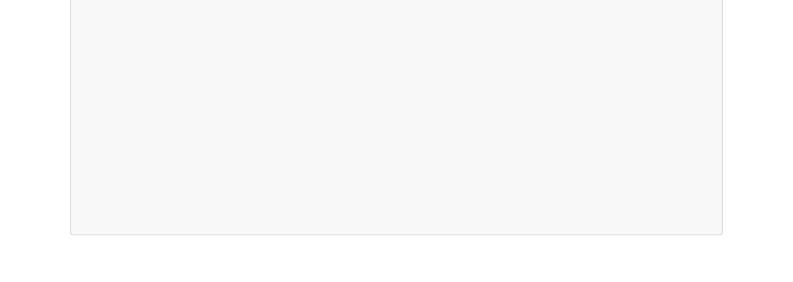
        // 创建笔记本实体对象

        Laptop lt = new Laptop();

        // 笔记本开启

        lt.run();

        // 创建鼠标实体对象



        Usb u = new Mouse();         // 笔记本使用鼠标

        lt.useUSB(u);

        // 创建键盘实体对象

        KeyBoard kb = new KeyBoard();         // 笔记本使用键盘

        lt.useUSB(kb);

        // 笔记本关闭

        lt.shutDown();

    }

}