# 面向对象编程-类与对象（下）

> ## 本节目标
>
> 1. 代码块
> 2. 内部类的定义与使用
> 3. 继承的定义与使用
> 4. 重写（override）
> 5. final关键字
> 6. 多态性

开始本节课之前，我们先来看一道阿里的java校招笔试题

```
public class HelloA {
    //构造方法
    public HelloA(){
        System.out.println("Hello A!父类构造方法");
    }
    //非静态代码块
    {
        System.out.println("i'm A class.父类非静态代码块");
    }
    //静态代码块
    static{
        System.out.println("static A 父类静态代码块");
    }
}

public class HelloB extends HelloA {
    //构造方法
    public HelloB(){
        System.out.println("Hello B! 构造方法");
    }
    //非静态代码块
    {
        System.out.println("i'm B class.非静态代码块");
    }
    //静态代码块
    static{
        System.out.println("static B 静态代码块");
    }
    public static void main(String[] args) {
        System.out.println("---start---");
        new HelloB();
        new HelloB();
        System.out.println("---end---");
```

```
    }
}
```

请写出上述代码的输出

学完本节课，每个同学都应该掌握上述代码的输出。

# 1. 代码块（考点）

代码块定义：使用`{}`定义的一段代码。

根据代码块定义的位置以及关键字，又可分为以下四种：

- 普通代码块
- 构造块
- 静态块
- 同步代码块（后续讲解多线程部分再谈）

## 1.1 普通代码块

普通代码块：定义在方法中的代码块

范例：观察普通代码块

```
public class Test{
    public static void main(String[] args) {
        { //直接使用{}定义，普通方法块
            int x = 10 ;
            System.out.println("x = " +x);
        }
        int x = 100 ;
        System.out.println("x = " +x);
    }
}
```

一般如果方法中代码过长，为避免变量重名，使用普通代码块。（使用较少，了解概念即可）。

## 1.2 构造块

构造块：定义在类中的代码块(不加修饰符)

范例：观察构造块

```
class Person{
    { //定义在类中，不加任何修饰符，构造块
        System.out.println("1.Person类的构造块");
    }
    public Person(){
        System.out.println("2.Person类的构造方法");
```

```
    }
}

public class Test{
    public static void main(String[] args) {
        new Person();
        new Person();
    }
}
```

通过以上代码我们发现：**构造块优先于构造方法执行，每产生一个新的对象就调用一次构造块，构造块可以进行简单的逻辑操作（在调用构造方法前）**

## 1.3 静态代码块

静态代码块：使用static定义的代码块

根据静态块所在的类的不同又可分为以下两种类型

> 1. 在非主类中
> 2. 在主类中

### 1.3.1 在非主类中的静态代码块

范例：观察非主类中的静态块

```
class Person{
    { //定义在类中，不加任何修饰符，构造块
        System.out.println("1.Person类的构造块");
    }
    public Person(){
        System.out.println("2.Person类的构造方法");
    }
    static { //定义在非主类中的静态块
        System.out.println("3.Person类的静态块");
    }
}

public class Test{
    public static void main(String[] args) {
        System.out.println("--start--");
        new Person();
        new Person();
        System.out.println("--end--");
    }
}
```

通过以上代码我们可以发现：

> 1. 静态块优先于构造块执行。
> 2. 无论产生多少实例化对象，静态块都只执行一次。

*静态块的主要作用是为static属性进行初始化*

#### 1.3.2 在主类中的代码块

范例：定义在主类中的代码块

```
public class Test{
    {
        System.out.println("1.Test的构造块");
    }
    public Test(){
        System.out.println("2.Test的构造方法");
    }
    static{
        System.out.println("3.Test的静态块");
    }

    public static void main(String[] args) {
        System.out.println("--start--");
        new Test();
        new Test();
        System.out.println("--end--");
    }
}
```

**在主类中定义的静态块，优先于主方法（main）执行**

*总结：针对以上对代码块的讲解，如果一些属性需要在使用前做处理，可以考虑使用代码块。*

# 2.内部类的定义与使用

## 2.1 内部类的基本概念

*内部类：所谓内部类就是在一个类的内部进行其他类结构的嵌套的操作*

范例：观察内部类的简单定义

```
class Outer{
    private String msg = "Hello World" ;
    // ********************************
    class Inner{ //定义一个内部类
        public void print(){ //定义一个普通方法
            System.out.println(msg); //调用msg属性
        }
    }
    // ********************************
    //在外部类中定义一个方法，该方法负责产生内部类对象并且调用print()方法
    public void fun(){
        Inner in = new Inner(); //内部类对象
        in.print(); // 内部类提供的print()方法
```

```
    }
}


public class Test{
    public static void main(String[] args) {
        Outer out = new Outer(); //外部类对象
        out.fun(); //外部类方法
    }
}
```

通过以上代码我们会发现，引入内部类后，程序的结构有些混乱。虽然内部类破坏了程序的结构，但是从另一方面来讲，内部类可以方便的操作外部类的私有访问。

范例：修改上述代码，要求把内部类拆开到外部，主方法代码不变，实现相同的功能

```
class Outer{
    private String msg = "Hello World" ;
    public String getMsg(){ //通过此方法才能取得msg属性
        return this.msg ;
    }
    public void fun(){ //3.现在由out对象调用了fun()方法
        Inner in = new Inner(this);  //4.this表示当前对象
        in.print(); //7.调用方法
    }
}

class Inner{
    private Outer out;
    public Inner(Outer out){ //5.Inner.out = mian.out
        this.out = out ; //6.引用传递
    }
    public void print(){ //8.执行此方法
        System.out.println(out.getMsg());
        }
    }

public class Test{
    public static void main(String[] args) {
        Outer out = new Outer(); //1. 实例化Outter类对象
        out.fun(); //2.调用Outter类方法
    }
}
```



## 2.2 内部类为什么存在

> 1. 内部类方法可以访问该类定义所在作用域中的数据，包括被 private 修饰的私有数据
> 2. 内部类可以对同一包中的其他类隐藏起来
> 3. 内部类可以实现 java 单继承的缺陷

> 4. 当我们想要定义一个回调函数却不想写大量代码的时候我们可以选择使用匿名内部类来实现

范例:使用内部类来实现"多继承"

```
class A {
    private String name = "A类的私有域";
    public String getName() {
        return name;
    }
}
class B {
    private int age = 20;
    public int getAge() {
        return age;
    }
}

class Outter {
    private class InnerClassA extends A {
        public String name() {
            return super.getName();
        }
    }
    private class InnerClassB extends B {
        public int age() {
            return super.getAge();
        }
    }
    public String name() {
        return new InnerClassA().name();
    }
    public int age() {
        return new InnerClassB().age();
    }
}
public class Test {
    public static void main(String[] args) {
        Outter outter = new Outter();
        System.out.println(outter.name());
        System.out.println(outter.age());
    }
}
```



## 2.3 内部类与外部类的关系

> - 对于非静态内部类，内部类的创建依赖外部类的实例对象，在没有外部类实例之前是无法创建内部类的
> - 内部类是一个相对独立的实体，与外部类不是is-a关系
> - 内部类可以直接访问外部类的元素(包含私有域)，但是外部类不可以直接访问内部类的元素

- 外部类可以通过内部类引用间接访问内部类元素

范例:内部类可以直接访问外部类的元素

```
class Outter {
    private String outName;
    private int outAge;

    class Inner {
        private int InnerAge;
        public Inner() {
            Outter.this.outName = "I am Outter class";
            Outter.this.outAge = 20;
        }
        public void display() {
            System.out.println(outName);
            System.out.println(outAge);
        }
    }
}

public class Test {
    public static void main(String[] args) {
        Outter.Inner inner = new Outter().new Inner();
        inner.display();
    }
}
```

范例:外部类可以通过内部类引用间接访问内部类元素

```
class Outter {
    public void display() {
        // 外部类访问内部类元素，需要通过内部类引用来访问
        Inner inner = new Inner();
        inner.display();
    }
    class Inner {
        public void display() {
            System.out.println("I am InnerClass");
        }
    }
}

public class Test {
    public static void main(String[] args) {
        Outter out = new Outter();
        out.display();
    }
```

```
}
```

## 2.4 内部类

### 2.4.1 内部类分类

在Java中内部类主要分为：

- 成员内部类
- 静态内部类
- 方法内部类
- 匿名内部类

### 2.4.1 在使用内部类的时候创建内部类对象

```
外部类.内部类 内部类对象 = new 外部类().new 内部类();

Outter.Inner in = new Outter().new Inner();
```

### 2.4.2 在外部类内部创建内部类对象

在外部类内部创建内部类，就像普通对象一样直接创建

```
Inner in = new Inner();
```

## 2.5 内部类详解

### 2.5.1 成员内部类

在成员内部类中要注意两点：

> 1. 成员内部类中不能存在任何static的变量和方法
> 2. 成员内部类是依附于外围类的，所以只有先创建了外围类才能够创建内部类

### 2.5.2 静态内部类

关键字static可以修饰成员变量、方法、代码块，其实它还可以**修饰内部类**，使用static修饰的内部类我们称之为静态内部类。静态内部类与非静态内部类之间存在一个最大的区别，我们知道非静态内部类在编译完成之后会隐含地保存着一个引用，该引用是指向创建它的外围类，但是静态内部类却没有。没有这个引用就意味着：

> 1. 静态内部类的创建是不需要依赖于外围类，可以直接创建
> 2. 静态内部类不可以使用任何外围类的非static成员变量和方法，而内部类则都可以

外部类的创建语法:

```
外部类.内部类 内部类对象 = new 外部类.内部类();
```

范例:使用static创建静态内部类

```
class Outer{
        private static String msg = "Hello World" ;
        // ********************************
        static class Inner{ //定义一个内部类
                public void print(){ //此时只能使用外部类中的static操作
                        System.out.println(msg); //调用msg属性
                }
        }
        // ********************************
        //在外部类中定义一个方法，该方法负责产生内部类对象并且调用print()方法
        public void fun(){
                Inner in = new Inner(); //内部类对象
                in.print(); // 内部类提供的print()方法
        }
}

public class Test{
        public static void main(String[] args) {
                Outer.Inner in = new Outer.Inner();
                in.print();
        }
}
```

### 2.5.3 方法内部类

方法内部类定义在外部类的方法中，局部内部类和成员内部类基本一致，只是它们的作用域不同，方法内部类只能在该方法中被使用，出了该方法就会失效。 对于这个类的使用主要是应用与解决比较复杂的问题，想创建一个类来辅助我们的解决方案，到那时又不希望这个类是公共可用的，所以就产生了局部内部类。

> 1. 局部内类不允许使用访问权限修饰符 public private protected 均不允许
> 2. 局部内部类对外完全隐藏，除了创建这个类的方法可以访问它其他的地方是不允许访问的。
> 3. 局部内部类要想使用方法形参，该形参必须用final声明(JDK8形参变为隐式final声明).

范例:使用方法内部类

```
class Outter {
    private int num;
    public void display(int test) {
        class Inner {
            private void fun() {
                num++;
                System.out.println(num);
                System.out.println(test);
            }
        }
        new Inner().fun();
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Outter out = new Outter();
        out.display(20);
    }
}
```

### 2.5.4 匿名内部类

匿名内部类其实就是一个没有名字的方法内部类，所以它符合方法内部类的所有约束.除此之外，还有一些地方需要注意:

> 1. 匿名内部类是没有访问修饰符的。
> 2. 匿名内部类必须继承一个抽象类或者实现一个接口
> 3. 匿名内部类中不能存在任何静态成员或方法
> 4. 匿名内部类是没有构造方法的，因为它没有类名。
> 5. 与局部内部相同匿名内部类也可以引用方法形参。此形参也必须声明为 final

范例: 使用匿名内部类

```
interface MyInterface {
    void test();
}
class Outter {
    private int num;
    public void display(int para) {
        // 匿名内部类，实现了MyInterface接口
        new MyInterface(){
            @Override
            public void test() {
                System.out.println("匿名内部类"+para);
            }
        }.test();
    }
}

public class Test {
    public static void main(String[] args) {
        Outter outter = new Outter();
        outter.display(20);
    }
}
```

总结：**内部类的使用暂时不作为设计的首选，内部类的特点如下**

- 1.破坏了程序的结构
- 2.方便进行私有属性的访问。（外部类也可以访问内部类的私有域）

- 3.如果发现类名称上出现了`.`，应当立即想到内部类的概念。

# 3.继承的定义与使用

*面向对象的第二大特征：继承。继承的主要作用在于，在已有基础上继续进行功能的扩充。*

范例：定义两个类（人、学生）

```
class Person{
   private String name;
   private int age;

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
class Student{
    private String name;
    private  int age;
    private String school;

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

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

```
    public String getSchool() {
        return school;
    }

    public void setSchool(String school) {
        this.school = school;
    }
}
```

以上程序就是我们之前一直采用的模式，单独的java类，含有大量重复性代码。不仅代码上重复，而且从概念上讲，学生一定是人，学生和人相比学生更加具体，学生类描述的范围更小，具备的属性更多，具有的方法也更多。

这个时候要想消除结构定义上的重复，就要用到继承。

## 3.1 继承的实现

*在Java中，继承使用extends关键字来实现，定义的语法如下：*

**class 子类 extends 父类**

关于继承定义的说明：

子类在一些书上也被称为派生类，父类也被称为超类(Super Class)

范例：继承的基本实现

```
class Person{
   private String name;
   private int age;

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
class Student extends Person{ //定义了一个子类
}

public class Test {
```

```
    public static void main(String[] args) {
       Student student = new Student();
       student.setName("Steven");
       student.setAge(18);
       System.out.println("姓名: "+student.getName()+", 年
龄: "+student.getAge());
    }
}
```

通过上述代码可以发现，当发生了类继承关系之后，子类可以直接继承父类的操作，可以实现代码的重用。子类最低也维持和父类相同的功能。子类可以进行功能的扩充。例如：扩充属性和方法。

范例：子类进行功能的扩充

```
class Student extends Person{ //定义了一个子类
    private String school; //扩充的新属性

    public String getSchool() {
        return school;
    }

    public void setSchool(String school) {
        this.school = school;
    }
}

public class Test {
    public static void main(String[] args) {
       Student student = new Student();
       student.setName("Steven");
       student.setAge(18);
       student.setSchool("高新一中");
       System.out.println("姓名: "+student.getName()+", 年
龄: "+student.getAge()+",学校: "+student.getSchool());
    }
}
```

*继承的主要作用是对类进行扩充以及代码的重用。*

## 3.2 继承的限制

> - 子类对象在进行实例化前一定会首先实例化父类对象。默认调用父类的构造方法后再调用子类构造方法进行子类对象初始化。

范例：观察子类对象创建

```
class Person{
        public Person(){
                System.out.println("**Person类对象产生**");
```

```
        }
}
class Student extends Person{
        public Student(){
                super() ; //此语句在无参时写于不写一样
                System.out.println("**Student类对象产生**");
        }
}
public class Test{
        public static void main(String[] args) {
                new Student();
        }
}
```

以上代码我们发现，没有任何一条语句调用父类构造方法。因此，子类对象实例化之前一定先实例化父类对象。

*注意：实际上在子类的构造方法之中，相当于隐含了一个语句 super();*

*同时需要注意的是，如果父类里没有提供无参构造，那么这个时候就必须使用super()明确指明你要调用的父类构造方法。*

> - Java只允许单继承，不允许多继承。

一个子类只能继承一个父类。

范例：错误的多继承

```
class A{}
class B{}
class C extends A,B{}
```

C同时继承A和B的主要目的是同时拥有A和B中的操作，为了实现这样的目的，可以采用多层继承的形式完成。

```
class A{}
class B extends A{}
class C extends B{}
```

*这个层数不建议太多。类的继承关系最多3层。*

**总结：Java不允许多重继承，但是允许多层继承**

> - 在进行继承的时候，子类会继承父类的所有结构。（包含私有属性、构造方法、普通方法）但是这个时候需要注意的是，所有的非私有操作属于显示继承（可以直接调用），所有的私有操作属于隐式继承（通过其他形式调用，例如setter或getter）

范例：显示继承与隐式继承

```
class Person{
```

```
    private String name;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
class Student extends Person{
    public void fun(){
        System.out.println(getName());
    }
}

public class Test {
    public static void main(String[] args) {
       Student student = new Student();
       student.setName("Steven");
       System.out.println(student.getName());
       student.fun();
    }
}
```

此时父类中的属性的确被子类所继承了，但是发现子类能够使用的是所有非private操作，而所有的private操作无法被直接使用，所以称为隐式继承。

*继承总结：*

> 1. 继承的语法以及继承的目的（扩展已有类的功能，使代码重用）
> 2. 子类对象的实例化流程：不管如何操作，一定要先实例化父类对象。
> 3. 不允许多重继承，只允许多层继承。

# 4.覆写（override）

*在清楚继承的概念后，此时就有可能出现以下情况*

*如果子类定义了与父类相同的方法或属性的时候，这样的操作就称为覆写（override）*

## 4.1 方法的覆写（重点）

> 备注：研究抽象类和接口的第一步

方法的覆写：子类定义了与父类方法名称、参数类型及个数完全相同的方法。但是被覆写不能够拥有比父类更为严格的访问控制权限。

范例：观察简单覆写

```
class Person{
```

```
        public void print(){
                System.out.println("1.[Person]类的print方法");
        }
}

class Student extends Person{
        public void print(){
                System.out.println("2.[Student]类的print方法");
        }
}
public class Test{
        public static void main(String[] args) {
                new Student().print();
        }
}
```

以后在进行覆写操作的时候注意关注以下两点：

> 1. 你当前使用的对象是通过哪个类new的。
> 2. 当调用某个方法，如果该方法已经被子类所覆写了，那么调用的一定是被覆写过的方法。

在进行方法覆写的时候，有明确的要求:被覆写不能够拥有比父类更为严格的访问控制权限。

现在已经接触了三种访问控制权限：`private<default<public`。那么也就意味着如果父类使用public进行方法声明，那么子类必须也使用public；如果父类使用default，那么子类可以使用default或者public。

范例：错误的覆写

```
class Person{
        public void print(){
                System.out.println("1.[Person]类的print方法");
        }
}

class Student extends Person{
         void print(){ //更严格的访问控制权限
                System.out.println("2.[Student]类的print方法");
        }
}
public class Test{
        public static void main(String[] args) {
                new Student().print();
        }
}
```

建议： 1.以后写方法时，99.99%的情况下建议使用public。

2.写属性，98%情况下建议使用private。

*问题：如果现在父类方法使用private定义，子类中使用public覆写，对吗?*

范例：父类使用private定义的方法，子类中使用public覆写

```
class Person{
        public void fun(){
                this.print();
        }
        //如果现在父类方法使用了private定义，那么就表示该方法只能被父类使用，子类无法使用。换言之，子类根本就不知道父类有这样的方法。
        private void print(){
                System.out.println("1.[Person]类的print方法");
        }
}

class Student extends Person{
        //这个时候该方法只是子类定义的新方法而已，并没有和父类的方法有任何关系。
        public void print(){
                System.out.println("2.[Student]类的print方法");
        }
}

public class Test{
        public static void main(String[] args) {
                new Student().fun();
        }
}
```

**面试题：请解释重载(overload)和覆写(override)的区别**

| No | 区别 | 重载（overload） | 覆写(override) |
| --- | --- | --- | --- |
| 1 | 概念 | 方法名称相同，参数的类型及个数不同 | 方法名称、返回值类型、参数的类型及个数完全相同 |
| 2 | 范围 | 一个类 | 继承关系 |
| 3 | 限制 | 没有权限要求 | 被覆写的方法不能拥有比父类更严格的访问控制权限 |

再次强调：为了良好的设计，在重载时请保持方法返回类型一致。

## 4.2 属性的覆写（了解）

当子类定义了和父类属性名称完全相同的属性的时候，就成为属性的覆盖。

范例：属性覆盖

```
class Person{
        public String info = "Person";
}

class Student extends Person{
        // 按照就近取用原则，肯定找被覆盖的属性。
        public String info = "Student";
}
public class Test{
        public static void main(String[] args) {
                System.out.println(new Student().info);
        }
}
```

这种操作本身没有任何意义，其核心的原因在于：类中的属性都要求使用private封装，一旦封装了，子类不知道父类具有什么属性，那么也就不存在属性覆盖的问题了。了解概念即可。

## 4.3 super关键字

上面讲到子类对象实例化操作的时候讲过super()，当时的主要作用是子类调用父类构造方法时才使用的。

*那么在进行覆写的操作过程之中，子类也可以使用super.方法()/super.属性明确调用父类中的方法或属性*

范例：使用super调用父类的同名方法

```
class Person{
        public void print(){
                System.out.println("1.I am father");
        }
}

class Student extends Person{
        public void print(){
                super.print();
                System.out.println("2.I am child");
        }
}
public class Test{
        public static void main(String[] args) {
                new Student().print();
        }
}
```

范例：使用super调用父类属性

```
class Person{
        public String info = "爸爸! ";
}

class Student extends Person{
        public String info = "儿子! " ;

        public void print(){
                //不找本类中的属性
                System.out.println(super.info);
                System.out.println(this.info);
        }
}
public class Test{
        public static void main(String[] args) {
                new Student().print();
        }
}
```

通过上述讲解可以发现super和this在使用上非常的相似，但是两者最大的区别是super是子类访问父类的操作，而this是本类的访问处理操作。

| No | 区别 | this | super |
| --- | --- | --- | --- |
| 1 | 概念 | 访问本类中的属性和方法 | 由子类访问父类中的属性、方法 |
| 2 | 查找范围 | 先查找本类，如果本类没有就调用父类 | 不查找本类而直接调用不累定义 |
| 3 | 特殊 | 表示当前对象 | 无 |

能使用super.方法()一定要明确标记出是父类的操作。

**1.子类覆写父类的方法是因为父类的方法功能不足才需要覆写。**

**2.方法覆写的时候使用的就是public权限**

# 5.final关键字

在Java中final被称为终结器。

- 使用final修饰类、方法、属性
- final成员变量必须在声明的时候初始化或者在构造器中初始化，否则就会报编译错误
- 使用final定义的类不能有子类(String类便是使用final定义)
- final一旦修饰一个类之后，该类的所有方法默认都会加上final修饰。(不包含成员变量)

```
final class A{} //A类不能有子类
```

- 使用final定义的方法不能被子类所覆写

```
class A{
        public final void fun(){}
}
```

- 使用final定义的变量就成为了常量，**常量必须在声明时赋值**，并且不能够被修改。

```
publc final int a = 100 ;
```

- 使用final修饰的变量不能再次赋值
- 定义常量(public static final )，常量全用大写字母，多个单词间以_分隔。

```
public static final int MAX_AGE = 120;
```

# 6.多态性

## 6.1 多态表现

在Java中，对于多态的核心表现主要有以下两点：

- 方法的多态性：
  - ①方法的重载：同一个方法名称可以根据参数的类型或个数不同调用不同的方法体
  - ②方法的覆写：同一个父类的方法，可能根据实例化子类的不同也有不同的实现。
- 对象的多态性【抽象类和接口才能体会到实际用处】（前提：方法覆写）：
  - 【自动，90%】①对象的向上转型：父类 父类对象 = 子类实例。
  - 【强制， 1%】②对象的向下转型：子类 子类对象 =（子类）父类实例。

范例：观察向上转型

```
class Person{
        public void print(){
                System.out.println("1.我是爸爸! ");
        }
}

class Student extends Person{
        public void print(){
                System.out.println("2.我是儿子! ");
        }
}
public class Test{
        public static void main(String[] args) {
                Person per = new Student(); //向上转型
                per.print();
        }
}
```

不管是否发生了向上转型，核心本质还是在于：你使用的是哪一个子类（new在哪里），而且调用的方法是否被子类所覆写了。

向下转型指的是将父类对象变为子类对象，但在这之前我们需要明确：为什么我们需要向下转型？当你需要子类扩充操作的时候就要采用向下转型

范例：观察向下转型

```
class Person{
```

```
        public void print(){
                System.out.println("1.我是爸爸! ");
        }
}

class Student extends Person{
        public void print(){
                System.out.println("2.我是儿子! ");
        }
        public void fun(){
                System.out.println("只有儿子有! ");
        }
}
public class Test{
        public static void main(String[] args) {
                Person per = new Student();
                per.print();
                //这个时候父类能够调用的方法只能是本类定义好的方法
                //所以并没有Student类中的fun()方法，那么只能够进行向下转型处理
                Student stu = (Student) per;
                stu.fun();
        }
}
```

【此概念一般开发用不到】但是并不是所有的父类对象都可以向下转型：如果要想进行向下操作之前，一定要首先发生向上转型，否则在转型时会出现`ClassCastException`。

范例：观察错误转型

```
Person per = new Person(); //父类对象
Student stu = (Student) per; //强转
```

问题：如果向下转型存在安全隐患，那么如何转型才靠谱呢？最好的做法就是先进行判断，而后在进行转型，那么就可以依靠`instanceof`关键字实现，该关键字语法如下：

子类对象`instanceof`类，返回boolean类型

范例：观察`instanceof`操作

```
Person per = new Student();
                System.out.println(per instanceof Person);
                System.out.println(per instanceof Student);
                if (per instanceof Student) { //避免ClassCastException
                        Student stu = (Student) per ;
                        stu.fun();
                }
```

这种转换到底有什么意义？

范例：要求定义一个方法，这个方法可以接收Person类的所有子类实例，并调用Person类的方法。

```
class Person{
        public void print(){
                System.out.println("1.我是人类！");
        }
}

class Student extends Person{
        public void print(){
                System.out.println("2.我是学生！");
        }
}
class Worker extends Person{
        public void print(){
                System.out.println("3.我是工人！");
        }
}
public class Test{
        public static void main(String[] args) {
                whoYouAre(new Student());
                whoYouAre(new Worker());
        }
        public static void whoYouAre(Person per){
                per.print();
        }
}
```

通过以上分析就可以清楚，对象的向上转型有一个最为核心的用途：*操作参数统一*。

多态性总结：

> - 对象多态性的核心在于方法的覆写。
> - 通过对象的向上转型可以实现接收参数的统一，向下转型可以实现子类扩充方法的调用（一般不操作向下转型，有安全隐患）。
> - 两个没有关系的类对象是不能够进行转型的，一定会产生ClassCastException。

# 课后练习

范例: 请选出下列程序出错的语句

```
public void test() {
    byte b1=1,b2=2,b3,b6,b8;
    final byte b4=4,b5=6,b7=9;
    b3=(b1+b2);  /*语句1*/
    b6=b4+b5;    /*语句2*/
    b8=(b1+b4);  /*语句3*/
    b7=(b2+b5);  /*语句4*/
    System.out.println(b3+b6);
}
```

**总结数据类型转换**

当使用 +、-、*、/、%、运算操作时，遵循如下规则：

- 只要两个操作数中有一个是double类型的，另一个将会被转换成double类型，并且结果也是double类型
- 两个操作数中有一个是float类型的，另一个将会被转换为float类型，并且结果也是float类型
- 两个操作数中有一个是long类型的，另一个将会被转换成long类型，并且结果也是long类型，否则**（操作数为：byte、short、int 、char），两个数都会被转换成int类型，并且结果也是int类型。**
- final修饰的变量类型不会发生变化