## 3.1、继承性（重点）

继承性是面向对象的第二大主要特征。

### 3.1.1、继承问题的引出

任何的概念出现都有其自己的目的以及可以结局的问题范畴，那么下面首先编写两个程序：Person类、Student类。

|  |  |
| --- | --- |
| **Person.java：** | **Student.java：** |
| class Person {  private String name ;  private int age ;  public void setName(String name) {  this.name = name ;  }  public void setAge(int age) {  this.age = age ;  }  public String getName() {  return this.name ;  }  public int getAge(){  return this.age ;  }  } | class Student {  **private String name ;**  **private int age ;**  private String school ;  **public void setName(String name) {**  **this.name = name ;**  **}**  **public void setAge(int age) {**  **this.age = age ;**  **}**  public void setSchool(String school) {  this.school = school ;  }  **public String getName() {**  **return this.name ;**  **}**  **public int getAge(){**  **return this.age ;**  **}**  public String getSchool() {  return this.school ;  }  } |

以上两个程序所使用到的概念都是利用之前所学习过的知识点来完成，但是现在通过代码的比较就可以清楚的发现一点：代码之中存在了重复，而按照之前所学，这种重复是不可能消除的，在所有的代码之中，最具有重复意义的就是链表类，针对于每一个简单Java类或者是说其他的任何类，都编写一个链表程序，代码量庞大，而且不方便维护。

### 3.1.2、继承的概念

继承性严格来讲就是指扩充一个类已有的功能。在Java之中，如果要实现继承的关系，可以使用如下的语法：

|  |
| --- |
| class 子类 **extends** 父类 {} |

· 对于extends而言，应该翻译为扩充，但是为了理解方便，统一称为继承；

· 子类又被称为派生类；

· 父类又被称为超类（Super Class）。

**范例：**观察继承的基本实现

|  |
| --- |
| class Person {  private String name ;  private int age ;  public void setName(String name) {  this.name = name ;  }  public void setAge(int age) {  this.age = age ;  }  public String getName() {  return this.name ;  }  public int getAge(){  return this.age ;  }  }  **class Student extends Person { // Student类继承了Person类**  **}**  public class TestDemo {  public static void main(String args[]) {  Student stu = new Student() ; // 实例化的是子类  stu.setName("张三") ; // Person类定义  stu.setAge(20) ; // Person类定义  System.out.println("姓名：" + stu.getName() + "，年龄：" + stu.getAge()) ;  }  } |

现在通过代码就可以发现了，子类（Student）并没有定义任何的操作，而在主类之中所使用的全部操作都是由Person类定义的，就证明，子类即使不扩充父类，那么也属于维持功能的状态。

**范例：**在子类之中扩充父类的功能

|  |
| --- |
| class Person {  private String name ;  private int age ;  public void setName(String name) {  this.name = name ;  }  public void setAge(int age) {  this.age = age ;  }  public String getName() {  return this.name ;  }  public int getAge(){  return this.age ;  }  }  class Student **extends** Person { // Student类继承了Person类  **private String school ; // 子类的属性**  **public void setSchool(String school) {**  **this.school = school ;**  **}**  **public String getSchool() {**  **return this.school ;**  **}**  }  public class TestDemo {  public static void main(String args[]) {  Student stu = new Student() ; // 实例化的是子类  stu.setName("张三") ; // Person类定义  stu.setAge(20) ; // Person类定义  stu.setSchool("清华大学") ; // Student类扩充方法  System.out.println("姓名：" + stu.getName() + "，年龄：" + stu.getAge() + "，学校：" + stu.getSchool()) ;  }  } |

以上的代码，子类对于父类的功能进行了扩充（扩充了一个属性和两个方法）。但是思考一下：子类从外表看是扩充了父类的功能，但是对于以上的代码，子类还有一个特点：**子类实际上是将父类定义的更加的具体化的一种手段**。父类表示的范围大，而子类表示的范围小。

### 3.1.3、继承的限制

虽然继承可以进行类功能的扩充，但是其在定义的时候也是会存在若干种限制的。

**限制一：**一个子类只能够继承一个父类，存在单继承局限

这个概念实际上是相对于其他语言而言，在其他语言之中，一个子类可以同时继承多个父类，就好比如下代码：

**范例：**错误的程序

|  |
| --- |
| class A {}  class B {}  class C extends **A,B** {} // 一个子类继承了两个父类 |

以上操作称为**多重继承**，实际上以上的做法就是希望一个子类，可以同时继承多个类的功能，但是以上的语法不支持而已，但是可以换种方式完成同样的操作。

**范例：**正确的程序

|  |
| --- |
| class A {}  class B extends A {}  class C extends B {} |

C实际上是属于（孙）子类，这样一来就相当于B类继承了A类的全部方法，而C类又继承了A和B类的方法，这种操作称为**多层继承**。结论：**Java之中只允许多层继承，不允许多重继承**，Java存在单继承局限。

严格来讲，按照以上的代码格式，可以一直无限制的继承下去，但是从实际来讲，应该不超过三层。

**限制二：**在一个子类继承的时候，实际上会继承父类之中的所有操作（属性、方法），但是需要注意的是，对于所有的非私有（no private）操作属于显式继承（可以直接利用对象操作），而所有的私有操作属于隐式继承（间接完成）。

|  |
| --- |
| class A {  **private String msg ;**  public void setMsg(String msg) {  this.msg = msg ;  }  public String getMsg() {  return this.msg ;  }  }  class B extends A {  public void print() {  System.out.println(msg) ; // 错误: msg可以在A中访问private  }  }  public class Test {  public static void main(String args[]) {  B b = new B() ;  b.setMsg("张三") ;  System.out.println(b.getMsg()) ;  }  } |

此时对于A类之中的msg这个私有属性发现无法直接进行访问，但是却发现可以通过setter、getter方法间接的进行操作。

**限制三：**在继承关系之中，如果要实例化子类对象，会默认先调用父类构造，为父类之中的属性初始化，之后再调用子类构造，为子类之中的属性初始化，即：默认情况下，子类会找到父类之中的无参构造方法。

|  |
| --- |
| class A {  public A() { // 父类无参构造  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*") ;  }  }  class B extends A {  public B() { // 子类构造  System.out.println("#########################");  }  }  public class Test {  public static void main(String args[]) {  B b = new B() ; // 实例化子类对象  }  } |

这个时候虽然实例化的是子类对象，但是发现它会默认先执行父类构造，调用父类构造的方法体执行，而后再实例化子类对象，调用子类的构造方法。而这个时候，对于子类的构造而言，就相当于隐含了一个super()的形式。

|  |
| --- |
| class B extends A {  public B() { // 子类构造  **super() ; // 调用父类构造**  System.out.println("#########################");  }  } |

现在默认调用的是无参构造，而如果这个时候父类没有无参构造，则子类必须通过super()调用指定参数的构造方法。

|  |
| --- |
| class A {  public A(String msg) { // 父类构造  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*") ;  }  }  class B extends A {  public B() { // 子类构造  **super("Hello") ; // 调用父类构造**  System.out.println("#########################");  }  }  public class Test {  public static void main(String args[]) {  B b = new B() ; // 实例化子类对象  }  } |

在任何的情况下，子类都逃不出父类构造的调用，但是，既然super()可以调用父类构造，那么这个语法和this()很相似，那么很明显了，**super调用父类构造的时候，一定要放在构造方法的首行上**。

**疑问？**有没有可能性，不让子类去调用父类构造？

既然super()和this()都是调用构造方法，而且都要放在构造方法的首行上，那么如果说this()出现了，那么super()应该就不会出现了，所以他编写了如下的程序：

|  |
| --- |
| class A {  public A(String msg) { // 父类构造  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*") ;  }  }  class B extends A {  public B(String msg) {  **this(msg,10) ; // 调用本类构造**  }  public B(String msg,int age) { // 子类构造  **this(msg) ; // 调用本类构造**  System.out.println("#########################");  }  }  public class Test {  public static void main(String args[]) {  B b = new B("",20) ; // 实例化子类对象  }  } |

在之前讲解this关键字的时候强调过一句话：**如果一个类之中有多个构造方法之间使用this()互相调用的话，那么至少要保留有一个构造方法作为出口，而这个出口就一定会去调用父类构造**。

此时在某种程度上来讲，有一个问题解释了一半：**一个简单Java类一定要保留有一个无参构造方法**。

## 3.2、覆写（重点）

既然现在出现了继承的关系，那么就存在了子类和父类的联系，而在子类之中有可能定义和父类完全相同的方法或属性的名称，这个时候就称为覆写了。

### 3.2.1、方法的覆写

当子类定义了和父类在方法名称、返回值类型、参数类型及个数完全相同的方法的时候，称为方法的覆写。

**范例：**没有覆写的操作

|  |
| --- |
| class A {  public void print() {  System.out.println("Hello World .") ;  }  }  class B extends A {  }  public class Test {  public static void main(String args[]) {  B b = new B() ;  b.print() ; // 方法从父类继承而来  }  } |

这个时候实例化的是子类对象，但是在子类之中，没有print()方法，那么就使用从父类之中继承而来的pritn()方法。

**范例：**实现覆写

|  |
| --- |
| class A {  public void print() {  System.out.println("Hello World .") ;  }  }  class B extends A {  public void print() { // 方法名称、参数类型及个数、返回值全相同  System.out.println("世界，你好！") ;  }  }  public class Test {  public static void main(String args[]) {  B b = **new B()** ;  b.print() ; // 方法从父类继承而来  }  } |

**当一个类之中的方法被覆写之后，如果实例化的是这个子类对象，则调用的方法就是被覆写过的方法。**

但是在进行方法覆写的时候有一个点需要注意：**被子类所覆写的方法不能拥有比父类更严格的访问控制权限，对于访问控制权限现在已经接触过三种：private < default（不写） < public**；

如果此时父类之中的方法是default权限，那么子类覆写的时候只能是default或public权限，而如果父类的方法是public，那么子类之中方法的访问权限只能是public。

**范例：**错误的操作

|  |
| --- |
| class A {  **public void print()** {  System.out.println("Hello World .") ;  }  }  class B extends A {  **void print()** {  System.out.println("世界，你好！") ;  }  } |

**提示一下：别没事干自己给自己找事，以后只要是方法都是public，99.9999%可以解决问题。**

当一个子类覆写了一个父类方法的时候，那么在这种情况下，子类要想调用父类的被覆写过的方法，则在方法前要加上“super”。

|  |
| --- |
| class A {  public void print() {  System.out.println("Hello World .") ;  }  }  class B extends A {  public void print() {  super.print() ;  System.out.println("世界，你好！") ;  }  }  public class Test {  public static void main(String args[]) {  B b = new B() ;  b.print() ; // 方法从父类继承而来  }  } |

在以后一定要记住操作范围：

· this.方法()：先从本类查找是否存在指定的方法，如果没有找到，则调用父类操作；

· super.方法()：直接由子类调用父类之中的指定方法，不再找子类。

**提问：**请问以下的操作是覆写吗？

|  |
| --- |
| class A {  **private** void print() {  System.out.println("Hello World .") ;  }  public void fun() {  this.print() ;  }  }  class B extends A {  **public** void print() { // 不叫覆写  System.out.println("世界，你好！") ;  }  }  public class Test {  public static void main(String args[]) {  B b = new B() ;  b.fun() ; // 方法从父类继承而来  }  } |

首先从覆写的概念上来讲：现在父类的权限是private，而子类是public，的确是扩大了权限，而且方法的参数名称及个数、返回值类型都一样。这种情况在开发之中是绝对不会出现的，即：使用了private定义的操作都无法真正覆写。

### 3.2.2、属性的覆盖（别了解了）

当一个子类定义了和父类重名的属性名称的时候，就表示属性的覆盖了。

|  |
| --- |
| class A {  public String msg = "Hello World ." ;  }  class B extends A {  public int msg = 100 ; // 属性同名  public void print() {  System.out.println("msg = " + this.msg) ;  System.out.println("msg = " + super.msg) ;  }  }  public class Test {  public static void main(String args[]) {  B b = new B() ;  b.print() ; // 方法从父类继承而来  }  } |

这种操作几乎没有意义，因为从开发来讲，属性一定要封装，封装之后就没有覆盖这一概念了。

**面试题：**请解释一下this和super的区别？

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **区别** | **this** | **super** |
| 1 | 定义 | 表示本类对象 | 表示父类对象 |
| 2 | 使用 | 本类操作：this.属性、this.方法()、this() | 父类操作：super.属性、super.方法()、super() |
| 3 | 调用构造 | 调用本类构造，要放在首行 | 子类调用父类构造，放在首行 |
| 4 | 查找范围 | 先从本类查找，找不到查找父类 | 直接由子类查找父类 |
| 5 | 特殊 | 表示当前对象 | - |

**面试题：**请解释一下方法重载与覆写的区别？当方法重载的时候能否改变其返回值类型？

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **区别** | **重载** | **覆写** |
| 1 | 英文单词 | Overloading | Override |
| 2 | 定义 | 方法名称相同、参数的类型及个数不同 | 方法名称、参数类型及个数、返回值类型完全相同 |
| 3 | 权限 | 没有权限要求 | 被子类所覆写的方法不能拥有比父类更严格的访问控制权限 |
| 4 | 范围 | 发生在一个类之中 | 发生在继承关系类之中 |

方法重载的时候可以改变返回值类型，一般设计的时候不会这样去做。

## 3.3、思考题（总结，核心）

现在要求定义一个整型数组的操作类，数组的大小由外部决定，用户可以向数组之中增加数据，以及取得数组中的全部数据，也可以根据外部提供的数组的增长大小，在原本的数组之上扩充指定的容量，另外，在此类上派生两个子类：

· **排序类：**取得的数组内容是经过排序出来的结果；

· **反转类：**取得的数组内容是反转出来的结果；

首先要完成的是定义父类，根本就不需要考虑子类。

**范例：**定义了父类 —— Array

|  |
| --- |
| class Array { // 数组操作类  private int [] data ;  private int foot = 0 ; // 控制脚标  public Array(int len) { // 由外部传递大小  if (len > 0) {  this.data = new int [len] ;  } else {  this.data = new int [1] ; // 维持一个大小  }  }  public boolean add(int num) {  if (this.foot < this.data.length) { // 有位置  this.data[this.foot ++] = num ; // 保存数据  return true ;  }  return false ;  }  public int [] getData() {  return this.data ;  }  public void increment(int num) {  int [] newArr = new int [this.data.length + num] ;  System.arraycopy(this.data,0,newArr,0,this.data.length) ;  this.data = newArr ; // 改变引用  }  }  public class Test {  public static void main(String args[]) {  Array arr = new Array(5) ;  System.out.println(arr.add(8)) ;  System.out.println(arr.add(10)) ;  System.out.println(arr.add(2)) ;  System.out.println(arr.add(5)) ;  System.out.println(arr.add(3)) ;  System.out.println(arr.add(9)) ;  arr.increment(3) ;  System.out.println(arr.add(20)) ;  System.out.println(arr.add(10)) ;  System.out.println(arr.add(30)) ;  int result [] = arr.getData() ;  for (int x = 0 ; x < result.length ; x ++) {  System.out.println(result[x]) ;  }  }  } |

**范例：**定义排序类

|  |
| --- |
| class SortArray extends Array {  public SortArray (int len) {  super(len) ; // 调用父类的有参构造  }  public int [] getData() {  java.util.Arrays.sort(super.getData()) ;  return super.getData() ;  }  }  public class Test {  public static void main(String args[]) {  SortArray arr = new SortArray(5) ;  System.out.println(arr.add(8)) ;  System.out.println(arr.add(10)) ;  System.out.println(arr.add(2)) ;  System.out.println(arr.add(5)) ;  System.out.println(arr.add(3)) ;  System.out.println(arr.add(9)) ;  arr.increment(3) ;  System.out.println(arr.add(20)) ;  System.out.println(arr.add(10)) ;  System.out.println(arr.add(30)) ;  int result [] = arr.getData() ;  for (int x = 0 ; x < result.length ; x ++) {  System.out.println(result[x]) ;  }  }  } |

由于所有的操作方法是以父类操作为主，所以在使用的时候此处只是换了一个类的名称，其他的地方都没有改变。

**范例：**定义反转子类

|  |
| --- |
| class ReverseArray extends Array {  public ReverseArray(int len) {  super(len) ;  }  public int [] getData() { // 覆写  int head = 0 ;  int tail = super.getData().length - 1 ;  int center = super.getData().length / 2 ;  for (int x = 0 ; x < center ; x ++) {  int temp = super.getData()[head] ;  super.getData()[head] = super.getData()[tail] ;  super.getData()[tail] = temp ;  head ++ ;  tail -- ;  }  return super.getData() ;  }  }  public class Test {  public static void main(String args[]) {  ReverseArray arr = new ReverseArray(5) ;  System.out.println(arr.add(8)) ;  System.out.println(arr.add(10)) ;  System.out.println(arr.add(2)) ;  System.out.println(arr.add(5)) ;  System.out.println(arr.add(3)) ;  System.out.println(arr.add(9)) ;  arr.increment(3) ;  System.out.println(arr.add(20)) ;  System.out.println(arr.add(10)) ;  System.out.println(arr.add(30)) ;  int result [] = arr.getData() ;  for (int x = 0 ; x < result.length ; x ++) {  System.out.println(result[x]) ;  }  }  } |

**在整个程序的开发之中，可以明显感觉到，所有的操作都是围绕着父类功能的扩充进行的，但是方法并没有改变，所以在开发之中，父类的设计是最重要的，子类最好的继承或者说是覆写操作，都应该以父类的方法为主。**

**3.4、final关键字（重点）**

在Java中，final关键字表示的是一个终结器的概念，使用final可以定义类、方法、变量。

**1、 使用final定义的类不能有子类，太监类**

|  |
| --- |
| **final** class A {  }  class B extends A {  } |

**2、 使用final定义的方法不能被子类所覆写**

|  |
| --- |
| class A {  public **final** void print() {}  }  class B extends A {  public void print() {}  } |

**3、 使用final定义的变量，就表示常量，常量在定义的时候必须设置默认值，并且无法修改**

|  |
| --- |
| class A {  final String INFO = "hello world" ; // 常量  public final void print() {  INFO = "world" ; // 无法修改  }  } |

而如果说现在使用了public static来定义的常量，那么这个常量就称为全局常量。

|  |
| --- |
| public static final String INFO = "hello world" ; // 全局常量 |

而对于以上final关键字定义的三个操作，只有全局常量的概念是被开发之中所使用的，像类或方法定义上使用final的情况，几乎是不会在我们自己编写的代码过程之中出现。**定义final常量的时候每个单词的字母都要大写**。

## 3.5、构造方法私有化（重点）

在讲解本操作之前，首先来观察如下的程序。

|  |
| --- |
| class Singleton { // 定义一个类  public void print() {  System.out.println("Hello World .") ;  }  }  public class Test {  public static void main(String args[]) {  Singleton inst = null ; // 声明对象  inst = new Singleton() ; // 实例化对象  inst.print() ; // 调用方法  }  } |

在以上的程序之中，Singleton类里面是存在构造方法的（因为如果一个类之中没有明确的定义一个构造方法的话，则会自动生成一个无参的，什么都不做的构造方法），但是下面要将构造方法改变一下。

|  |
| --- |
| class Singleton { // 定义一个类  **private Singleton() {} // 构造方法私有化**  public void print() {  System.out.println("Hello World .") ;  }  } |

现在发现在实例化Singleton类对象的时候，程序出现了编译错误，因为构造方法被私有化了，无法在外部调用，即：无法在外部实例化Singleton类的对象。

**那么现在就需要思考：在保证Singleton类之中的构造方法不修改不增加，以及print()方法不修改的情况下，如何操作，才可以让类的外部通过实例化对象再去调用print()方法？**

**思考过程一：**使用private访问权限定义的操作只能被本类所访问，外部无法调用，那么现在既然构造方法被私有化了，就证明，这个类的构造方法只能被本类所调用，即：现在在本类之中产生本类实例化对象。

|  |
| --- |
| class Singleton { // 定义一个类  **Singleton instance = new Singleton() ;**  private Singleton() {} // 构造方法私有化  public void print() {  System.out.println("Hello World .") ;  }  } |

**思考过程二：**对于一个类之中的普通属性，默认情况下一定要在本类存在了实例化对象之后才可以进行调用，可是现在在Singleton类的外部无法产生实例化对象，那么必须想一个办法，让Singleton类中的instance属性可以在没有Singleton类实例化对象的时候来进行调用，可以使用static完成，static定义的属性特点：由类名称直接调用，并且在没有实例化对象的时候也可以调用。

|  |
| --- |
| class Singleton { // 定义一个类  **static Singleton instance = new Singleton() ;**  private Singleton() {} // 构造方法私有化  public void print() {  System.out.println("Hello World .") ;  }  }  public class Test {  public static void main(String args[]) {  Singleton inst = null ; // 声明对象  inst = **Singleton.instance ;** // 实例化对象  inst.print() ; // 调用方法  }  } |

**思考过程三：**类之中的全部属性都应该封装，所以以上的instance属性应该进行封装，而封装之后要想取得属性要编写getter方法，只不过这个时候的getter方法应该也由类名称直接调用，定义为static型。

|  |
| --- |
| class Singleton { // 定义一个类  **private static Singleton instance = new Singleton() ;**  **private** Singleton() {} // 构造方法私有化  public void print() {  System.out.println("Hello World .") ;  }  public static Singleton getInstance() {  return instance ;  }  }  public class Test {  public static void main(String args[]) {  Singleton inst = null ; // 声明对象  **inst = Singleton.getInstance() ; // 实例化对象**  inst.print() ; // 调用方法  }  } |

**思考过程四：**这样做的目的？此时程序之中的instance属性，属于static，那么表示所有Singleton类的对象，不管有多少个都共同拥有同一个instance属性，那么既然是有同一个，那么又有什么意义呢？

现在做一个简单的思考：如果说现在一个类只希望有唯一的一个实例化对象出现，应该控制构造方法，如果构造方法对外部不可见了，那么现在肯定无法执行对象的实例化操作，必须将构造方法隐藏，使用private隐藏。

既然清楚了这个目的，不过本程序依然有一个问题。

|  |
| --- |
| public static Singleton getInstance() {  **instance = new Singleton() ;**  return instance ;  } |

本操作语法没有错误，也不需要考虑是否有意义，但是现在的代码是允许这样做的，而这样做发现之前表示唯一一个实例化对象的所有努力就白费了，那么必须想办法废除掉这种做法，可以在定义instance的时候增加一个final关键字。

|  |
| --- |
| class Singleton { // 定义一个类  private static final Singleton INSTANCE = new Singleton() ;  private Singleton() {} // 构造方法私有化  public void print() {  System.out.println("Hello World .") ;  }  public static Singleton getInstance() {  return INSTANCE ;  }  }  public class Test {  public static void main(String args[]) {  Singleton inst = null ; // 声明对象  inst = Singleton.getInstance() ; // 实例化对象  inst.print() ; // 调用方法  }  } |

这样的设计在设计模式上讲就称为单例设计模式（Singleton）。

**面试题：**请编写一个Singleton程序，并说明其主要特点？

|  |
| --- |
| class Singleton { // 定义一个类  private static final Singleton INSTANCE = new Singleton() ;  private Singleton() {} // 构造方法私有化  public void print() {  System.out.println("Hello World .") ;  }  public static Singleton getInstance() {  return INSTANCE ;  }  }  public class Test {  public static void main(String args[]) {  Singleton inst = null ; // 声明对象  inst = Singleton.getInstance() ; // 实例化对象  inst.print() ; // 调用方法  }  } |

特点：构造方法被私有化，只能够通过getInstance()方法取得Singleton类的实例化对象，这样不管外部如何操作，最终也只有一个实例化对象，在单例设计模式之中，一定会存在一个static方法，用于取得本类的实例化对象。

对于单例设计模式，在日后的开发之中，只会用到此概念，但是具体的代码很少去编写。

**扩展（可以不会）：**

对于单例设计模式按照设计模式的角度而言，分为两种：

· 饿汉式：之前写的程序就属于饿汉式，因为在类之中的INSNTACE属性是在定义属性的时候直接实例化；

· 懒汉式：在第一次使用一个类实例化对象的时候才去实例化。

**范例：**观察懒汉式

|  |
| --- |
| class Singleton { // 定义一个类  private static Singleton instance ;  private Singleton() {} // 构造方法私有化  public void print() {  System.out.println("Hello World .") ;  }  public static Singleton getInstance() {  if (instance == null) { // 没有实例化  instance = new Singleton() ; // 实例化  }  return instance ;  }  } |

这些概念清楚就行了，对于单例设计再次强调：记清楚代码的结构以及操作的特点，以后这种代码虽然不会由你自己去写，但是概念一定会用到。

## 3.6、多例设计模式（理解）

单例设计模式只留有一个类的一个实例化对象，而多例设计模式，会定义出多个对象，例如：定义一个表示星期X的类，这个类的对象只有7个取值，定义一个表示性别的类，只有2个取值，定义一个表示颜色基色的操作类，颜色只有三个：红、绿、蓝，这种情况下，这样的类就不应该由用户无限制的去创造实例化对象，应该只使用有限的几个，这个就属于多例设计，但不管是单例设计还是多例设计，有一个核心不可动摇 —— 构造方法私有化。

|  |
| --- |
| class Sex {  **private static final Sex MALE = new Sex("男") ;**  **private static final Sex FEMALE = new Sex("女") ;**  private String title ;  private Sex(String title) { // 构造方法私有化  this.title = title ;  }  public static Sex getInstance(String msg) {  switch(msg) {  case "male" :  return MALE ;  case "female" :  return FEMALE ;  default :  return null ;  }  }  public String getTitle() {  return this.title ;  }  }  public class Test {  public static void main(String args[]) {  Sex male = Sex.getInstance("male") ;  System.out.println(male.getTitle()) ;  }  } |

构造方法的封装是实现所有操作的基础，但是在我们所编写的代码过程之中，首先考虑的不是构造方法的封装。

## 3.7、多态性（重点）

多态是面向对象的最后一个主要特征，它本身主要分为两个方面：

· 方法的多态性：重载与覆写

|- 重载：同一个方法名称，根据不同的参数类型及个数可以完成不同的功能；

|- 覆写：同一个方法，根据操作的子类不同，所完成的功能也不同。

· 对象的多态性：父子类对象的转换。

|- 向上转型：子类对象变为父类对象，格式：父类 父类对象 = 子类实例，自动；

|- 向下转型：父类对象变为子类对象，格式：子类 子类对象 = (子类) 父类实例，强制；

**范例：**编写一个简单的程序，观察程序输出

|  |
| --- |
| class A {  public void print() {  System.out.println("A、public void print(){}") ;  }  }  class B extends A {  **public void print() { // 方法覆写**  **System.out.println("B、public void print(){}") ;**  **}**  }  public class Test {  public static void main(String args[]) {  B b = **new** B() ; // 实例化子类对象  b.print() ;  }  } |

这种操作主要观察两点：

· 看new的是那一个类；

· 看new的这个类之中是否被覆写了父类要调用的方法。

**范例：**向上转型

|  |
| --- |
| public class Test {  public static void main(String args[]) {  A a = new B() ; // 向上转型  a.print() ;  }  } |

**范例：**向下转型

|  |
| --- |
| public class Test {  public static void main(String args[]) {  **A a = new B() ; // 向上转型**  **B b = (B) a ; // 向下转型**  b.print() ;  }  } |

**范例：**观察如下程序

|  |
| --- |
| public class Test {  public static void main(String args[]) {  **A a = new A() ; // 没有转型**  **B b = (B) a ; // 向下转型，java.lang.ClassCastException: A cannot be cast to B**  b.print() ;  }  } |

以上的程序在编译的时候没有发生任何的错误信息，但是在执行的时候出现了“ClassCastException”错误提示，表示的是类转换异常，即：两个没有关系的类互相发生了对象的强制转型。

在整个故事之中可以发现：

· 如果两个人真的有关系，可以分配遗产；

· 如果两个人真的没有关系，不能分配遗产。

**转型因素：**

· 在实际的工作之中，对象的向上转型为主要使用，80%，向上转型之后，所有的方法以父类的方法为主，但是具体的实现，还是要看子类是否覆写了此方法；

· 向下转型，10%，因为在进行向下转型操作之前，一定要首先发生向上转型，以建立两个对象之间的联系，如果没有这种联系，是不可能发生向下转型的，一旦发生了运行中就会出现“ClassCastException”，当需要调用子类自己特殊定义方法的时候，才需要向下转型；

· 不转型，10%，在一些资源较少的时候，例如：移动开发。

|  |
| --- |
| class A {  public void print() {  System.out.println("A、public void print(){}") ;  }  }  class B extends A {  public void print() { // 方法覆写  System.out.println("B、public void print(){}") ;  }  public void getB() {  System.out.println("B、getB()") ;  }  }  public class Test {  public static void main(String args[]) {  A a = new B() ; // 向上转型  B b = (B) a ; // 向下转型  b.getB() ; // 父类没有的操作，只能通过子类对象调用  }  } |

既然在发生向下转型的操作之中会存在一些问题，那么能不能想判断一下再转型呢？如果要想判断某一个对象是否是某一个类的实例，只需要使用instanceof关键字即可，此操作的语法如下：

|  |
| --- |
| 对象 instanceof 类 🡺 返回boolean型 |

**范例：**观察instanceof验证

|  |
| --- |
| public class Test {  public static void main(String args[]) {  A a = new A() ;  System.out.println(a instanceof A) ;  System.out.println(a instanceof B) ;  if (a instanceof B) {  B b = (B) a ;  b.getB() ;  }  }  } |

为了日后的操作方便，在编写代码的时候，尽量不要去执行向下转型操作。子类尽量不要去扩充新的方法名称（父类没有的方法名称），依据父类定义的操作完善方法。

**范例：**观察完善的操作

|  |
| --- |
| class Person {  private String name ;  private int age ;  public Person(String name,int age) {  this.name = name ;  this.age = age ;  }  public String getInfo() {  return "姓名：" + this.name + "，年龄：" + this.age ;  }  }  class Student extends Person { // Student类继承了Person类  private String school ; // 子类的属性  public Student(String name,int age,String school) {  super(name,age) ;  this.school = school ;  }  public String getInfo() {  return super.getInfo() + "，学校：" + this.school ;  }  }  public class TestDemo {  public static void main(String args[]) {  Person per = new Student("张三",20,"清华大学") ;  System.out.println(per.getInfo()) ;  }  } |

一切的操作标准都要以父类为主，这是向上转型的意义所在，另外，除了以上的意义之外，向上转型还可以解决一个最为麻烦的问题。例如：

|  |
| --- |
| class A {  public void print() {  System.out.println("A、public void print(){}") ;  }  } |

现在要求定义一个方法，这个方法可以接收A类的任意子类对象。

**实现方式一：**不使用对象转型

|  |
| --- |
| class A {  public void print() {  System.out.println("A、public void print(){}") ;  }  }  class B extends A {  public void print() { // 方法覆写  System.out.println("B、public void print(){}") ;  }  }  class C extends A {  public void print() { // 方法覆写  System.out.println("C、public void print(){}") ;  }  }  public class Test {  public static void main(String args[]) {  fun(new B()) ;  fun(new C()) ;  }  public static void fun(B b) {  b.print() ;  }  public static void fun(C c) {  c.print() ;  }  } |

但是，如果真的依靠了以上的方式进行代码开发的话，最为严重的事情：如果A类有5000W个子类，方法重载5000W次，并且一旦有新的子类产生，那么方法都要被重复的修改，这样不现实。

**实现方式二：**利用对象向上转型完成

|  |
| --- |
| public class Test {  public static void main(String args[]) {  fun(new B()) ;  fun(new C()) ;  }  public static void fun(A a) {  a.print() ;  }  } |

这样一来，参数的类型就得到了统一，就算有再多的子类出现，方法或者是类也不需要进行修改了，但是在这块必须强调的是：子类操作的过程之中，尽量向父类靠拢。

|  |
| --- |
| class A {  public void print() {  System.out.println("A、public void print(){}") ;  }  }  class B extends A {  public void print() { // 方法覆写  this.getB() ;  System.out.println("B、public void print(){}") ;  }  public void getB() {  System.out.println("B、getB()") ;  }  }  class C extends A {  public void print() { // 方法覆写  this.getC() ;  System.out.println("C、public void print(){}") ;  }  public void getC() {  System.out.println("C、getC()") ;  }  }  public class Test {  public static void main(String args[]) {  fun(new B()) ;  fun(new C()) ;  }  public static void fun(A a) {  a.print() ;  }  } |

以后所有的代码之中，都会存在对象的转型问题，并且向上转型居多。

在日后的所有开发之中，像之前程序那样，一个类去继承另外一个已经实现好的类的情况，是不可能出现的。**即：一个类不能去继承一个已经实现好的类，只能继承抽象类或实现接口。对于抽象类和接口如果要想真正的清楚其概念，需要一段很长的时间，今天讲解它们的基本语法、定义形式、使用方式，给出一些代码的结构。**

## 3.8、抽象类（核心）

### 3.8.1、抽象类的基本概念

普通类就是一个完善的功能类，可以直接产生对象并且可以使用，里面的方法都是带有方法体的，而抽象类之中最大的特点是包含了抽象方法，而抽象方法是只声明而未实现（没有方法体）的方法，而抽象方法定义的时候要使用abstract关键字完成，而抽象方法一定要在抽象类之中，抽象类要使用abstract关键字声明。

**范例：**定义一个抽象类

|  |
| --- |
| abstract class A {  private String info = "Hello World ." ;  public void print() {  System.out.println(info) ;  }  public abstract void get() ; // 只声明没有方法体  } |

**范例：**错误的使用 —— 直接实例化对象

|  |
| --- |
| public class Test {  public static void main(String args[]) {  **A a = new A () ; // Test.java:10: 错误: A是抽象的; 无法实例化**  }  } |

**思考：**为什么抽象类对象不能够直接new？

一个类的对象实例化之后，可以调用类中的属性和方法，但是抽象类之中的抽象方法没有方法体，如果这样直接调用，那么不就乱了吗。

**抽象类的使用原则：**

· 抽象类必须有子类，使用extends继承，一个子类只能继承一个抽象类；

· 子类（如果不是抽象类）则必须覆写抽象类之中的全部抽象方法；

· 抽象类对象可以使用对象的向上转型方式，通过子类来进行实例化操作。

**范例：**使用抽象类

|  |
| --- |
| abstract class A {  private String info = "Hello World ." ;  public void print() {  System.out.println(info) ;  }  public **abstract** void get() ; // 只声明没有方法体  }  class Impl extends A {  public void get() {  System.out.println("Hello MLDN .") ;  }  }  public class Test {  public static void main(String args[]) {  A a = new Impl() ; // 向上转型  a.print() ; // 自己类定义  a.get() ; // 子类负责实现  }  } |

通过以上的一个程序，现在就可以清楚的发现，与之前类的继承不一样的是，抽象类定义出了子类必须要覆写的方法，而之前的类子类可以有选择性的来决定是否需要覆写。而且可以发现，抽象类实际上就比普通类多了一些抽象方法而已，其他的定义和普通类完全一样。**如果把普通类比喻成一盘炒熟的菜，那么抽象类就是一盘半成品。**

**关于抽象类的若干种疑问？**

· 抽象类能否使用final定义？

不能，因为抽象类必须有子类，final定义的类太监类，不能有子类；

· 抽象类之中能否包含构造方法？

可以，因为抽象类之中除了包含抽象方法之外，还包含了普通方法和属性，而属性一定要在构造方法执行完毕之后才可以进行初始化操作；

· 抽象类之中能否不包含抽象方法？

可以，抽象类之中可以没有抽象方法，但是反过来讲，如果有抽象方法，则一定是抽象类，即使抽象类之中没有抽象方法，也不能够被直接实例化；

· 抽象类能否使用static声明？

|  |
| --- |
| abstract class A {  private String info = "Hello World ." ;  static abstract class B {// 外部类  public abstract void print() ;  }  }  class Impl extends A.B {  public void print() {  System.out.println("Hello MLDN .") ;  }  }  public class Test {  public static void main(String args[]) {  A.B b = new Impl() ;  b.print() ;  }  } |

如果定义的是外部抽象类，则不能够使用static声明，可是如果定义的是内部抽象类，那么这个内部的抽象类使用了static声明之后，就表示是一个外部的抽象类。

### 3.8.2、抽象类的应用 —— 模板设计模式（体会）

下面首先通过一个简单的程序来分析一下，例如：现在有三种类型：狗、机器人、人；

· 狗具备三种功能：吃、睡、跑；

· 机器人具备两个功能：吃、工作；

· 人具备四个功能：吃、睡、跑、工作。

现在就要求设计一个程序，可以让这三类不同的类型，进行工作。现在给出的三个类实际上并没有任何的联系，唯一的联系就是在于一些行为上。

|  |
| --- |
| abstract class Action {  public static final int EAT = 1 ;  public static final int SLEEP = 3 ;  public static final int WORK = 5 ;  public static final int RUN = 7 ;  public void order(int flag) {  switch (flag) {  case EAT :  this.eat() ;  break ;  case SLEEP:  this.sleep() ;  break ;  case WORK :  this.work() ;  break ;  case RUN :  this.run() ;  break ;  case EAT + SLEEP + RUN :  this.eat() ;  this.sleep() ;  this.run() ;  break ;  case EAT + WORK :  this.eat() ;  this.work() ;  break ;  case EAT + SLEEP + RUN + WORK :  this.eat() ;  this.sleep() ;  this.run() ;  this.work() ;  break ;  }  }  public abstract void eat() ;  public abstract void sleep() ;  public abstract void run() ;  public abstract void work() ;  }  class Dog extends Action {  public void eat() {  System.out.println("小狗在吃。") ;  }  public void sleep() {  System.out.println("小狗在睡。") ;  }  public void run() {  System.out.println("小狗在跑步。") ;  }  public void work() {}  }  class Robot extends Action {  public void eat() {  System.out.println("机器人喝油。") ;  }  public void sleep() {}  public void run() {}  public void work() {  System.out.println("机器人在工作。") ;  }  }  class Person extends Action {  public void eat() {  System.out.println("人在吃饭。") ;  }  public void sleep() {  System.out.println("人在睡觉。") ;  }  public void run() {  System.out.println("人在跑步。") ;  }  public void work() {  System.out.println("人在工作。") ;  }  }  public class Test {  public static void main(String args[]) {  Action act1 = new Dog() ;  act1.order(Action.EAT + Action.SLEEP + Action.RUN) ;  Action act2 = new Robot() ;  act2.order(Action.EAT + Action.WORK) ;  }  } |

所有的子类如果要想正常的完成操作，必须按照指定的方法进行覆写才可以，而这个时候抽象类所起的功能就是一个类定义模板的功能。

## 3.9、接口（重点）

### 3.9.1、接口的基本概念

接口属于一种特殊的类，如果一个类定义的时候全部由**抽象方法和全局常量**所组成的话，那么这种类就称为接口，但是接口是使用interface关键字进行定义的。

|  |
| --- |
| interface A { // 定义接口  public static final String INFO = "Hello World ." ;  public abstract void print() ;  }  interface B {  public abstract void get() ;  } |

那么在接口之中，也同样存在了抽象方法，很明显，接口对象无法直接进行对象的实例化操作，那么接口的使用原则如下：

· 每一个接口必须定义子类，子类使用implements关键字实现接口；

· 接口的子类（如果不是抽象类）则必须覆写接口之中所定义的全部抽象方法；

· 利用接口的子类，采用对象的向上转型方式，进行接口对象的实例化操作。

下面给出子类实现接口的语法格式：

|  |
| --- |
| class 子类 [extends 父类] [implemetns 接口1,接口2,...] {} |

通过格式可以发现，每一个子类可以同时实现多个接口，但是只能继承一个父类。

**范例：**让子类实现接口

|  |
| --- |
| interface A { // 定义接口  public static final String INFO = "Hello World ." ;  public abstract void print() ;  }  interface B {  public abstract void get() ;  }  class X **implements** A,B { // 同时实现了两个接口  public void print() { // 方法覆写  System.out.println("Hello World .") ;  }  public void get() {  System.out.println(INFO) ;  }  }  public class Test {  public static void main(String args[]) {  A a = new X() ;  B b = new X() ;  a.print() ;  b.get() ;  }  } |

那么如果一个类现在即要实现接口又要继承抽象类的话，则应该采用先继承后实现的方式完成。

|  |
| --- |
| interface A { // 定义接口  public static final String INFO = "Hello World ." ;  public abstract void print() ;  }  interface B {  public abstract void get() ;  }  abstract class C {  public abstract void fun() ;  }  class X extends C implements A,B { // 同时实现了两个接口  public void print() { // 方法覆写  System.out.println("Hello World .") ;  }  public void get() {  System.out.println(INFO) ;  }  public void fun() {  System.out.println("世界，你好！") ;  }  }  public class Test {  public static void main(String args[]) {  A a = new X() ;  B b = new X() ;  C c = new X() ;  a.print() ;  b.get() ;  c.fun() ;  }  } |

**但是需要说明的是：接口之中的全部组成就是抽象方法和全局常量，那么在开发之中以下的两种定义接口的最终效果是完全一样的：**

|  |  |
| --- | --- |
| **完整定义：** | **简化定义：** |
| **interface A { // 定义接口**  **public static final String INFO = "Hello World ." ;**  **public abstract void print() ;**  **}** | **interface A { // 定义接口**  **public String INFO = "Hello World ." ;**  **public void print() ;**  **}** |

**接口之中的访问权限只有一种：public，即：定义接口方法的时候就算没有写上public，那么最终也是public。**

在Java之中每一个抽象类都可以实现多个接口，但是反过来讲，一个接口却不能继承抽象类，可是Java之中，一个接口却可以同时继承多个接口，以实现接口的多继承操作。

|  |
| --- |
| interface A {  public void printA() ;  }  interface B {  public void printB() ;  }  interface C **extends A,B** { // 一个接口继承了多个接口  public void printC() ;  }  class X implements C {  public void printA() {}  public void printB() {}  public void printC() {}  } |

而在开发之中，内部类是永远不会受到概念限制的，在一个类中可以定义内部类，在一个抽象类之中也可以定义抽象内部类，在一个接口里面也可以定义内部抽象类或内部接口，但是从实际的开发来讲，用户自己去定义内部抽象类或内部接口的时候是比较少见的（Android开发中见过内部接口），而且**在定义内部接口的时候如果使用了static，表示是一个外部接口**。

|  |
| --- |
| interface A {  public void printA() ;  static interface B { // 外部接口  public void printB() ;  }  }  class X implements A.B {  public void printB() {  System.out.println("Hello World .") ;  }  }  public class Test {  public static void main(String args[]) {  A.B temp = new X() ;  temp.printB() ;  }  } |

以上对于接口的概念并不是很难理解，但是需要强调的是，在实际的开发之中，接口有**三大主要功能：**

**· 制订操作标准；**

**· 表示一种能力；**

**· 将服务器端的远程方法视图暴露给客户端。**

### 3.9.2、使用接口定义标准

在日常的生活之中，接口这一名词经常听到的，例如：USB接口、打印接口、充电接口等等。

现在假设每一个USB设备只有两个功能：安装驱动程序、工作。

**范例：**定义出一个USB的标准

|  |
| --- |
| interface USB { // 操作标准  public void install() ;  public void work() ;  } |

**范例：**在电脑上应用此接口

|  |
| --- |
| class Computer {  public void plugin(USB usb) {  usb.install() ;  usb.work() ;  }  } |

**范例：**定义USB设备

|  |
| --- |
| class Phone implements **USB** {  public void install() {  System.out.println("安装手机驱动程序。") ;  }  public void work() {  System.out.println("手机与电脑进行工作。") ;  }  } |

**范例：**定义USB设备

|  |
| --- |
| class Print implements USB {  public void install() {  System.out.println("安装打印机驱动程序。") ;  }  public void work() {  System.out.println("进行文件打印。") ;  }  } |

**范例：**连接

|  |
| --- |
| interface USB { // 操作标准  public void install() ;  public void work() ;  }  class Computer {  public void plugin(USB usb) {  usb.install() ;  usb.work() ;  }  }  class Phone implements USB {  public void install() {  System.out.println("安装手机驱动程序。") ;  }  public void work() {  System.out.println("手机与电脑进行工作。") ;  }  }  class Print implements USB {  public void install() {  System.out.println("安装打印机驱动程序。") ;  }  public void work() {  System.out.println("进行文件打印。") ;  }  }  public class Test {  public static void main(String args[]) {  Computer c = new Computer() ;  c.plugin(new Phone()) ; // USB usb = new Phone() ;  c.plugin(new Print()) ;  }  } |

按照这种方式开发下去的话，不管有多少个USB接口的子类，都可以在电脑上使用。

### 3.9.3、接口的实际作用 —— 工厂设计模式（Factory）

下面首先来观察如下的程序代码。

|  |
| --- |
| interface Fruit {  public 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。") ;  }  }  public class Test {  public static void main(String args[]) {  Fruit f = new Apple() ;  f.eat() ;  }  } |

本程序非常简单就是通过接口的子类为接口对象实例化，但是本操作存在什么样的问题呢？

之前一直在强调，主方法或者是主类是一个客户端，客户端的操作应该越简单越好。但是现在的程序之中，有一个最大的问题：客户端之中，一个接口和一个固定的子类绑在一起了。

在本程序之中，最大的问题在于耦合上，发现在主方法之中，一个接口和一个子类紧密耦合在一起，这种方式比较直接，可以简单的理解为由：A 🡺 B，但是这种紧密的方式不方便于维护，所以后来使用了**A 🡺 C 🡺 B**，中间经历了一个过渡，这样一来B去改变，C去改变，但是A不需要改变，就好比JAVA的JVM一样：程序 🡺 JVM 🡺 操作系统。

**范例：**修改代码

|  |
| --- |
| interface Fruit {  public 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) {  if ("apple".equals(className)) {  return new Apple() ;  }  if ("orange".equals(className)) {  return new Orange () ;  }  return null ;  }  }  public class Test {  public static void main(String args[]) {  Fruit f = Factory.getInstance(args[0]) ;  f.eat() ;  }  } |

这个时候发现客户端不再和一个具体的子类耦合在一起了，就算以后增加了新的子类，那么只需要修改Factory类即可实现。

### 3.9.4、接口的实际作用 —— 代理设计模式（Proxy）

张金宇的不幸人生。。。555555。

|  |
| --- |
| interface Subject { // 操作主题  public void get() ; // 要银子  }  class RealSubject implements Subject { // 真正的要银子  public void get() {  System.out.println("真实业务主题") ;  }  }  class ProxySubject implements Subject {  private Subject sub = null ;  public ProxySubject(Subject sub) {  this.sub = sub ;  }  public void prepare() {  System.out.println("准备操作。") ;  }  public void destroy() {  System.out.println("收尾操作。") ;  }  public void get() {  this.prepare() ;  this.sub.get() ;  this.destroy() ;  }  }  public class Test {  public static void main(String args[]) {  Subject sub = new ProxySubject(new RealSubject()) ;  sub.get() ;  }  } |

通过以上的分析就可以得出结论：代理负责完成与真实业务有关的所有辅助性操作。

## 3.9.5、抽象类和接口的区别（面试题）

通过如上的分析，感觉抽象类和接口在使用上似乎区别不大，那么下面就通过一个表格给出这两者的区别。

（面试题：请解释抽象类和接口的区别？）

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **区别** | **抽象类** | **接口** |
| 1 | 定义关键字 | abstract class | interface |
| 2 | 组成 | 常量、变量、抽象方法、普通方法、构造方法 | 全局常量、抽象方法 |
| 3 | 权限 | 可以使用各种权限 | 只能是public |
| 4 | 关系 | 一个抽象类可以实现多个接口 | 接口不能够继承抽象类，却可以继承多接口 |
| 5 | 使用 | 子类使用extends继承抽象类 | 子类使用implements实现接口 |
| 抽象类和接口的对象都是利用对象多态性的向上转型，进行接口或抽象类的实例化操作 | |
| 6 | 设计模式 | 模板设计模式 | 工厂设计模式、代理设计模式 |
| 7 | 局限 | 一个子类只能够继承一个抽象类 | 一个子类可以实现多个接口 |

通过上面的分析可以得出结论：在开发之中，抽象类和接口实际上都是可以使用的，并且使用那一个都没有明确的限制，可是抽象类有一个最大的缺点 —— 一个子类只能够继承一个抽象类，存在单继承的局限，所以当遇到抽象类和接口都可以使用的情况下，**优先考虑接口，避免单继承局限**。

到此时已经学习过了：对象、类、抽象类、接口、继承、实现等等，这些都属于什么样的关系呢？

接口就是在类的基础之上的进一步具体的抽象。

# 4、总结

1、 继承性用于扩充类的功能；

2、 方法的覆写与对象多态性的联系；

3、 final关键字的使用；

4、 单例设计模式；

5、 对象的多态性，转型问题；

6、 抽象类和接口的概念；

7、 今天的三个设计模式（背下结构）：单例、工厂、代理。

# 5、预习任务

匿名内部类、Object类、异常的捕获及处理、包及访问控制权限。

# 6、作业

|  |  |  |
| --- | --- | --- |
| **No.** | **表达式** | **描述** |
| 1 |  |  |
| 2 |  |  |
| 3 |  |  |
| 4 |  |  |
| 5 |  |  |
| 6 |  |  |
| 7 |  |  |
| 8 |  |  |
| 9 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **方法名称** | **类型** | **描述** |
| 1 |  |  |  |
| 2 |  |  |  |
| 3 |  |  |  |
| 4 |  |  |  |
| 5 |  |  |  |
| 6 |  |  |  |
| 7 |  |  |  |
| 8 |  |  |  |
| 9 |  |  |  |