早上漏了一句话：方法的重载是面向对象多态性的一个体现

主要内容：

·继承

·抽象类

·接口

·多态

1. 继承：

举例：现在有一个人类，有姓名、年龄的属性和说话的方法

有一个学生类，有姓名、年龄、学号的属性和说话、考试的方法。

可以说学生类是继承自人类的，通过上面例子可以总结为什么需要继承：

·通过继承可以简化类的定义

·通过继承可以扩展类的功能

怎么实现继承？

子类 extends 父类

|  |
| --- |
| **package** com.wanczy.oopdemo01;  **public** **class** Person {  String name;  **int** age;    **public** **void** say(){  System.*out*.println("Hello");  }  } |
| **package** com.wanczy.oopdemo01;  **public** **class** Student **extends** Person{  String stu\_num;  **public** **void** test(){  System.*out*.println("我在考试");  }  } |
| **package** com.wanczy.oopdemo01;  **public** **class** OopDemo01 {  **public** **static** **void** main(String[] args) {  Student s = **new** Student();  s.say();  s.age = 10;  s.name = "李四";  }  } |

运行之后发现，子类继承了父类的属性和方法，这样子类就可以简化定义。

将父类中的属性和方法改为private（私有），则子类对象不能访问父类的属性和方法。所有说子类可以继承父类一切非私有的属性和方法。

继承的特点：继承具有层次结构，继承中父类可以继承爷爷类的所有的非私有的属性和方法，子类可以继承父类中非私有的属性和方法，就是说java中继承是一层一层继承的。和现实生活中一样，你爷爷将所有的积蓄继承给你爸爸，你爸爸将所有的积蓄再继承给你。

继承的规定：java只支持单继承，一个人不可能同时有两个亲生父亲。在java中体现就是一个类只能有一个父类， 但是在c++中是允许多继承的。

子类实例化的过程：

|  |
| --- |
| **package** com.wanczy.oopdemo01;  **public** **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;  }  **public** Person() {  System.*out*.println("这是父类Person的构造方法");  }  **public** Person(String name, **int** age) {  **this**.name = name;  **this**.age = age;  }  **public** **void** say(){  System.*out*.println("Hello");  }  } |
| **package** com.wanczy.oopdemo01;  **public** **class** Student **extends** Person{  **private** String stu\_num;    **public** String getStu\_num() {  **return** stu\_num;  }  **public** **void** setStu\_num(String stuNum) {  stu\_num = stuNum;  }  **public** **void** test(){  System.*out*.println("我在考试");  }    **public** Student(){  System.*out*.println("这是子类Student的构造方法");  }    } |
| **package** com.wanczy.oopdemo01;  **public** **class** OopDemo01 {  **public** **static** **void** main(String[] args) {  Student s = **new** Student();  }  } |

以上的程序运行结果发现，子类实例化的时候，会先去调用父类的构造方法， 然后再调用子类的构造方法，这就是我们子类实例化的过程，调用父类的构造方法的时候是先给子类对象开辟空间，并为类中的属性赋值。

super关键字：在java中，父类也称为superclass超类，super关键字的几个用法如下：

·super可以调用父类的构造方法（开辟空间并类中的属性赋值）

·super可以调用父类的方法

·super关键字可以调用父类的属性

调用父类的构造方法：

|  |
| --- |
| **package** com.wanczy.oopdemo01;  **public** **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;  }  **public** Person() {  System.*out*.println("这是父类Person的构造方法");  }  **public** Person(String name, **int** age) {  System.*out*.println("父类有参数的构造方法");  **this**.name = name;  **this**.age = age;  }  **public** **void** say(){  System.*out*.println("Hello");  }  } |
| **package** com.wanczy.oopdemo01;  **public** **class** Student **extends** Person{  **private** String stu\_num;    **public** String getStu\_num() {  **return** stu\_num;  }  **public** **void** setStu\_num(String stuNum) {  stu\_num = stuNum;  }  **public** **void** test(){  System.*out*.println("我在考试");  }    **public** Student(){  System.*out*.println("这是子类Student的构造方法");  }  **public** Student(String name, **int** age,String stuNum) {  **super**(name, age);//调用父类的构造方法  stu\_num = stuNum;  System.*out*.println("子类有参数的构造方法");  }    } |

但是有一点要注意，在super调用父类构造方法的时候，super必须写在子类构造方法的首行。是根据super（）参数的类型、个数、顺序而去调用相应的父类中的构造方法。

super调用父类的方法：

|  |
| --- |
| **public** **void** eat(){  **super**.say();//通过super 关键字调用父类的方法  } |

super关键字调用父类的属性。

|  |
| --- |
| **public** Student(String name, **int** age,String stuNum) {  //super();//调用父类的构造方法  **super**.age = age;  **super**.name = name;//这就是使用super关键字调用父类的属性  stu\_num = stuNum;  System.*out*.println("子类有参数的构造方法");  } |

可以发现super关键字和this关键字有非常多的相似之处，那么我们拉一个表格来说明this和super的区别

|  |  |  |
| --- | --- | --- |
| 功能 | this | super |
| 调用属性 | 调用本类的属性 | 调用父类的属性 |
| 调用方法 | 调用本类的方法 | 调用父类的方法 |
| 调用构造方法 | 调用本类的构造方法 | 调用父类的构造方法 |
| 表示对象 | 表示当前对象 | 不能表示对象 |

方法的重写（覆写）：和方法的重载一样，都是面向对象多态性的一个体现。方法名称一样，返回值，参数也一样，用在继承关系中，实现不同的功能。简单一点就是说父类中有一个方法，子类中也有一个此方法，就是子类覆写了父类的方法。

|  |
| --- |
| **package** com.wanczy.oopdemo01;  **public** **class** Person {  String name;  **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;  }  **public** Person() {  System.*out*.println("这是父类Person的构造方法");  }  **public** Person(String name, **int** age) {  System.*out*.println("父类有参数的构造方法");  **this**.name = name;  **this**.age = age;  }  **public** **void** say(){  System.*out*.println("Hello");  }  } |
| **package** com.wanczy.oopdemo01;  **public** **class** Student **extends** Person{  **private** String stu\_num;    **public** String getStu\_num() {  **return** stu\_num;  }  **public** **void** setStu\_num(String stuNum) {  stu\_num = stuNum;  }  **public** **void** test(){  System.*out*.println("我在考试");  }    **public** Student(){  System.*out*.println("这是子类Student的构造方法");  }  **public** Student(String name, **int** age,String stuNum) {  //super();//调用父类的构造方法  **super**.age = age;  **super**.name = name;//这就是使用super关键字调用父类的属性  stu\_num = stuNum;  System.*out*.println("子类有参数的构造方法");  }  **public** **void** eat(){  **super**.say();//通过super 关键字调用父类的方法  }  **public** **void** say(){  System.*out*.println("我是学生 Hello");  }  } |

我们发现子类覆写了父类的say方法之后，子类对象调用的是子类覆写后的方法，而不是父类的方法，即在方法的覆写中，子类对象调用的永远是子类覆写后的方法，而不会去调用父类的被覆写的方法。

覆写的规定：子类覆写后的方法的访问权限不能小于父类被覆写方法的访问权限。

访问权限的比较：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 位置 | private | default | protected | public |
| 同类 | √ | √ | √ | √ |
| 同包的类 | × | √ | √ | √ |
| 不同包的子类 | × | × | √ | √ |
| 不同包并且不是子类 | × | × | × | √ |

重载和覆写的区别：

|  |  |  |
| --- | --- | --- |
|  | 覆写 | 重载 |
| 概念 | 方法名称相同，返回值参数也相同，覆写后的访问权限要大于父类被覆写的访问权限 | 方法名称一样， 参数的个数、类型、顺序不同 |
| 范围 | 用在继承关系中 | 同一个类中 |

final：在java中，final可以修饰类，变量，方法；

·使用final修饰的类为太监类，不能被继承；

·使用final修饰的变量即为常量，只能赋值一次，不能改变（常量的名称全部大写）；

·使用final修饰的方法不能被子类覆写。

在java中，final使用最多的就是用来定义常量。

二、抽象类和接口

抽象类：使用abstract修饰的类为抽象类

抽象方法：用abstract修饰的方法为抽象方法

有一个抽象方法的类必须定义为抽象类。

抽象类和普通类的区别：抽象类要比普通类至少多一个抽象方法，普通类中所有的定义都可以在抽象类中定义，抽象类和抽象方法都需要使用abstract声明。

抽象类的使用规则：

·抽象类本身不能直接实例化。

·抽象类中的抽象方法只需要定义，不需要实现

·含有抽象方法的类必须定义为抽象类，抽象类必须要有子类，不然这个抽象类就没有任何意义，抽象类的子类必须覆写抽象类中全部的抽象方法。

·抽象类可以被抽象类继承

思考：抽象类能不能有构造方法呢？抽象类中可以定义构造方法

抽象了能不能使用final声明？抽象类不能使用final修饰

抽象类子类的实例化过程：

|  |
| --- |
| **package** com.wanczy.oopDemo02;  **public** **abstract** **class** OopDemo01 {  **public** **abstract** **void** sayHello();//抽象方法只需要声明，不需要实现  **public** OopDemo01(){  System.*out*.println("This is Abstract class 构造方法");  }  } |
| **package** com.wanczy.oopDemo02;  **public** **class** OopDemo02 **extends** OopDemo01 {  @Override  **public** **void** sayHello() {  System.*out*.println("just so so");  }  **public** OopDemo02(){  System.*out*.println("This is Abstract class 子类的 构造方法");  }  } |

我们发现抽象类子类的实例化过程和继承中子类的实例化过程是一样，首先会实例化父类（即调用父类的构造方法），再调用子类的构造方法。

练习：一个公司有两种员工，一种是技术员工，一种是销售员工，现在公司要为技术员工和销售员工提升工资，技术员工按照职称提升（高级工程师提升10%，中级工程师提升20%，初级工程师提升30%），销售员工按照去年的销售业绩提升工资（业绩在500w的提升20%，业绩在200w-499w的提升10%，业绩在100w-200w提升5%，业绩低于100w的不提升）

接口：使用interface定义的就是接口，接口中所有的方法都是抽象方法，所有变量即为常量，接口中只有抽象方法和常量。

接口定义的语法：

interface 接口名{

抽象方法；

常量；

}

如果一个抽象类中只含有抽象方法，那么这个抽象类就可以使用接口来表示。

接口的使用规则：

·接口中抽象方法和常量的访问权限永远是public

·接口中的抽象方法可以省略abstract，但是省略之后的方法仍然是抽象方法

·接口中的常量可以省略final ，但是省略之后的变量仍然是常量

·接口中抽象方法和常量的访问权限可以省略public，但是省略之后抽象方法和常量的访问权限仍然是public。

·接口在程序中叫做实现，即接口必须通过子类实现，但是此时不叫做子类，而叫做实现类，实现类必须覆写接口中所有的抽象方法，实现类可以直接使用接口中的所有常量。实现类 implements 接口  
 ·一个实现类可以实现多个接口，如果一个实现类实现的多个接口中有相同的抽象方法， 那么次实现类只需要实现此方法一次。

·接口和接口之间可以继承，一个接口可以使用extends去继承一个或者多个接口，子接口继承父接口中的所有的抽象方法和常量。

·接口不能直接实例化

通过抽象类和接口的讲解，大家可以发现，接口和抽象类其实有很多相同之处吗，那么我们在开发的时候是优先选择抽象类还是接口呢？

在以后的开发过程中，优先使用接口进行开发，因为接口是多继承多实现，而抽象类有java中单继承的局限性。  
接口是我们以后开发程序的一套标准，尤其是在以后的数据库开发中。

练习：使用接口进行对象数组的增删改查和打印。

面向对象的多态性

·方法的重载和覆写是面向对象多态性的一个体现

·子类对象可以作为父类对象使用（子类对象为父类实例化）

·抽象类子类对象可以作为抽象类对象使用（子类对象为父类实例化）

·实现类可以作为接口的对象使用（实现类对象为接口实例化）

|  |
| --- |
| **package** com.wanczy.oopdemo01;  **public** **class** OopDemo01 {  **public** **static** **void** main(String[] args) {  // Student s = new Student("张三",10,"06119212");  // Person p = s;//子类对象作为父类对象  Person p = **new** Student("张三",10,"06119212");    // System.out.println(s.getName());  // s.eat();  p.say();  }  } |
| **package** com.wanczy.oopDemo02;  **public** **class** Test {  **public** **static** **void** main(String[] args) {  OopDemo01 oop = **new** OopDemo02();//抽象类的子类的对象可以作为抽象类的对象使用  }  } |
| **package** com.wanczy.oopdemo03;  **public** **class** Demo02 {  **public** **static** **void** main(String[] args) {  InterfaceDemo03 inter3 = **new** Demo01();  InterfaceDemo02 inter2 = **new** Demo01();  InterfaceDemo01 inter1 = **new** Demo01();//实现类的对象可以作为接口对象使用  }  } |

抽象类和接口的区别：

|  |  |  |
| --- | --- | --- |
|  | 抽象类 | 接口 |
| 相似 | 不能直接实例化，必须依靠子类（实现类）实例化 | |
| 不同 | 包含抽象方法和普通类所有的定义 | 只有常量和抽象方法 |
| 可以有构造方法 | 不能有构造方法 |
| 抽象类可以实现多个接口 | 接口不能继承抽象类 |
| 存在单继承的局限性 | 可以多继承多实现 |