什么是多态？它的实现机制是什么呢？重载和重写的区别在那里？这就是这一次我们要回顾的四个十分重要的概念：继承、多态、重载和重写。 ) |6 L( R. }' K. d/ V0 I; q  
6 ^7 t. t6 T5 {\* L  
继承(inheritance)   
  
简单的说，继承就是在一个现有类型的基础上，通过增加新的方法或者重定义已有方法（下面会讲到，这种方式叫重写）的方式，产生一个新的类型。继承是面向对象的三个基本特征--封装、继承、多态的其中之一，**我们在使用JAVA时编写的每一个类都是在继承**，因为在JAVA语言中，java.lang.Object类是所有类最根本的基类（或者叫父类、超类），如果我们新定义的一个类没有明确地指定继承自哪个基类，那么JAVA就会默认为它是继承自Object类的。   
+ `3 ^5 z- D2 w( X  
我们可以把JAVA中的类分为以下三种： 3 X- G1 }7 ~3 T9 i  F1 |4 I7 c  
9 n0 F7 k& c& J  
类：使用class定义且不含有抽象方法的类。 : l% a0 ~8 s' B0 B  
抽象类：使用abstract class定义的类，它可以含有，也可以不含有抽象方法。 " Z& r) s7 G0 v% J# G- u; `  
接口：使用interface定义的类。   
1 q( |( }, b- Y6 {  
在这三种类型之间存在下面的继承规律：   
6 ]9 p\* A- g! i4 {\* h\* H0 g  
类可以继承（extends）类，可以继承（extends）抽象类，可以继承（implements）接口。 . O\* B7 |2 m0 {\* Q( l' b\* u! q  
抽象类可以继承（extends）类，可以继承（extends）抽象类，可以继承（implements）接口。 + G6 C1 A' |7 v% U+ k: r9 y  `  
接口只能继承（extends）接口。   
  
请注意上面三条规律中每种继承情况下使用的不同的关键字extends和implements，它们是不可以随意替换的。大家知道，一个普通类继承一个接口后，必须实现这个接口中定义的所有方法，否则就只能被定义为抽象类。我在这里之所以没有对implements关键字使用“实现”这种说法是因为从概念上来说它也是表示一种继承关系，而且对于抽象类implements接口的情况下，它并不是一定要实现这个接口定义的任何方法，因此使用继承的说法更为合理一些。   
  
以上三条规律同时遵守下面这些约束：   
: j\* G+ H+ ^/ \_( N6 j; l  
类和抽象类都只能最多继承一个类，或者最多继承一个抽象类，并且这两种情况是互斥的，也就是说它们要么继承一个类，要么继承一个抽象类。 - a8 w  u3 F: e8 ?( B+ v6 m  g  
类、抽象类和接口在继承接口时，不受数量的约束，理论上可以继承无限多个接口。当然，对于类来说，它必须实现它所继承的所有接口中定义的全部方法。   
抽象类继承抽象类，或者实现接口时，可以部分、全部或者完全不实现父类抽象类的抽象（abstract）方法，或者父类接口中定义的接口。   I; H5 j1 C1 v! H5 G; q. y0 T- {7 v, M  
类继承抽象类，或者实现接口时，必须全部实现父类抽象类的全部抽象（abstract）方法，或者父类接口中定义的全部接口。 ' u' W9 x; J. J2 s/ ]9 d  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_8 \_) k9 H+ g, G$ Z+

新的一行

继承给我们的编程带来的好处就是对原有类的**复用**（重用）。就像模块的复用一样，类的复用可以提高我们的开发效率，实际上，模块的复用是大量类的复用叠加后的效果。除了继承之外，**我们还可以使用组合的方式来复用类**。

所谓**组合**就是把原有类定义为新类的一个属性，通过在新类中调用原有类的方法来实现复用。如果新定义的类型与原有类型之间不存在被包含的关系，也就是说，从抽象概念上来讲，新定义类型所代表的事物并不是原有类型所代表事物的一种，比如黄种人是人类的一种，它们之间存在包含与被包含的关系，那么这时组合就是实现复用更好的选择。下面这个例子就是组合方式的一个简单示例： 3 h0 X) P9 g\* H/ c6 ?8 p  
Java代码 . {7 n3 J\* J1 P' J# q! D! \/ W( D  
public class Sub {   7 t4 y1 x- @9 f" ~  
    private Parent p = new Parent();   8 j, U0 r\* j1 ^" c  
    
    public void doSomething() {   & A" O+ O8 `1 Z& R1 u  
        // 复用Parent类的方法     
        p.method();     
        // other code     
    }   - a& Z$ f) b8 b7 u$ S  
}   ! d+ f! A6 O! k. r\* w: C  
    
class Parent {   : D" J\* s( N" x" `1 c) c  
    public void method() {   " ?( S# e8 r; c. R! m\* Q  
        // do something here   8 [5 r/ F+ x  M# ~; y; r  
    }   ; a- {% V' [& I! w4 O\* N+ r2 N  
}    
  
public class Sub {  
        private Parent p = new Parent();  
  
        public void doSomething() {7 ~' G( V  \9 `& H) I8 G  
                // 复用Parent类的方法/ v7 h- C\* N) g9 Y\* {) O" O  
                p.method();, k' W7 z2 X. `  
                // other code  
        }; w/ a6 N8 J8 i1 {; X5 |1 M  
}  
  
class Parent {  
        public void method() {  W, p) x. Z3 ?  
                // do something here  
        }  
}  
" Z0 t% |3 {# b# @9 ~  
当然，为了使代码更加有效，我们也可以在需要使用到原有类型（比如Parent p）时，才对它进行初始化。 ' ~. [# s, T0 ?8 n0 n+ N% E! U  
  
使用继承和组合复用原有的类，都是一种增量式的开发模式，这种方式带来的好处是不需要修改原有的代码，因此不会给原有代码带来新的BUG，也不用因为对原有代码的修改而重新进行测试，这对我们的开发显然是有益的。因此，如果我们是在维护或者改造一个原有的系统或模块，尤其是对它们的了解不是很透彻的时候，就可以选择增量开发的模式，这不仅可以大大提高我们的开发效率，也可以规避由于对原有代码的修改而带来的风险。   
' ?  z4 I9 p% A0 q$ n0 O& O  
多态(Polymorphism) 7 `. y. d6 l\* r; `$ w  
  
多态是又一个重要的基本概念，上面说到了，它是面向对象的三个基本特征之一。究竟什么是多态呢？我们先看看下面的例子，来帮助理解： & r$ h2 b  d' h/ }  u  
Java代码 ; \_, k1 N: ~4 a& F6 `4 t+ e  
//汽车接口   9 j; T+ x: W4 E5 [$ s. q1 A  
interface Car {     
    // 汽车名称     
    String getName();   0 x8 ^5 R7 n% F) x7 [  
    
    // 获得汽车售价     
    int getPrice();     
}   ! }6 y. [/ L3 l. H  
    
// 宝马     
class BMW implements Car {   - ^7 O: m3 N: `# d% T+ A" \_\* Y7 Z! t  
    public String getName() {     
        return "BMW";   ; [- v( q( x( D( S! B\* }0 B  
    }   ) q& p) m0 |% a2 C\* ~) M  
    
    public int getPrice() {     
        return 300000;   3 @6 J/ ^0 u; M6 H  
    }   / z, r\* L0 j0 V; W  
}     
    
// 奇瑞QQ     
class CheryQQ implements Car {   + p2 M9 K8 {; j\* b' {% d. o  
    public String getName() {   8 \) |4 m' D/ }1 c  
        return "CheryQQ";   4 d- b  b. O9 V3 S/ ^  
    }     
    
    public int getPrice() {     
        return 20000;   6 M4 ^0 W\* @) H# t+ p: p  
    }   $ [8 H( s. g\* m8 ]1 l  
}     
    
// 汽车出售店     
public class CarShop {     
    // 售车收入     
    private int money = 0;     
  6 v7 I' }# U  n) B$ c3 R, s  
    // 卖出一部车   4 i4 g5 O\* ^. s/ ^! K( l  
    public void sellCar(Car car) {   ; W( R5 h. G$ {  p  
        System.out.println("车型：" + car.getName() + "  单价：" + car.getPrice());     
        // 增加卖出车售价的收入     
        money += car.getPrice();     
    }   ) l, G$ T8 U- q( P\* y  @( k9 g  
  2 g. Z6 U# w0 M8 @. l/ C; \_0 h  
    // 售车总收入     
    public int getMoney() {     
        return money;   4 {- Q1 Q6 ?' U. K6 W; ?+ h0 h  
    }     
  0 d8 n1 ?7 S( e1 M6 ]: K; J) c5 s  
    public static void main(String[] args) {     
        CarShop aShop = new CarShop();   3 G$ k9 |4 l( l  
        // 卖出一辆宝马     
        aShop.sellCar(new BMW());   9 K9 X/ {5 P6 |6 R# \_  
        // 卖出一辆奇瑞QQ   / ]% R6 r% ~5 u4 D  
        aShop.sellCar(new CheryQQ());     
        System.out.println("总收入：" + aShop.getMoney());     
    }   " T- K- N# w7 U  
}  9 \_$ a6 Z: l: H! d  
  
//汽车接口4 L. W7 ?2 I! E' n4 N6 d3 I$ |  
interface Car {1 e+ x, `8 o$ N' l6 P; C# a  |  
        // 汽车名称  
        String getName();  
  
        // 获得汽车售价& k8 U\* ~- x7 m$ D  
        int getPrice();  
}  
  
// 宝马3 o$ T; b! B6 c) L( Q! S3 ]  
class BMW implements Car {- \_7 c( t! k2 M+ [\* x4 t4 {  
        public String getName() {+ ^! \_( h5 @6 j/ v\* s+ ^7 Z8 l+ Y  
                return "BMW";: f' \_) R% t3 N2 }1 G7 z) N% ?  
        }  
0 j. d% U, X. Y+ ?! r7 c5 M  
        public int getPrice() {  
                return 300000;  
        }  
}  
  
// 奇瑞QQ& |\* o" }! D" |& @; K" Y  
class CheryQQ implements Car {  
        public String getName() {- e& F# }! J- t\* \; M  
                return "CheryQQ";9 q; E! R, p1 J( o  
        }" p( r' A5 D- \_3 `& J  
( {7 u! R1 b, Y$ O  
        public int getPrice() {3 K) S9 z5 g6 F) z  f  
                return 20000;# C) u: M\* g$ V  
        }. d" K1 c, ~' l% g\* T: s\* j  
}  
5 H/ P" U  e\* a' V6 f% }$ j# S  
// 汽车出售店+ Z! {5 b0 }9 \_; M, @( U6 U) k  
public class CarShop {/ L8 v' F8 I: d2 i& A. O& T  
        // 售车收入) o: R7 v- c\* {$ v  
        private int money = 0;  
  
        // 卖出一部车% o5 V& l: x9 A5 t5 @  
        public void sellCar(Car car) {  
                System.out.println("车型：" + car.getName() + "  单价：" + car.getPrice());  
                // 增加卖出车售价的收入  
                money += car.getPrice();  
        }  
/ }. \6 I1 a1 u\* b; l0 n  
        // 售车总收入: E4 n% X\* m1 o& ^, B5 e- L  
        public int getMoney() {  
                return money;  
        }0 w! A4 V% E8 ?$ ^# x9 Y" b) ~4 C  
  
        public static void main(String[] args) {$ L. d: i) s' d, S  
                CarShop aShop = new CarShop();  
                // 卖出一辆宝马' [+ Z' k8 J0 n: m  F  
                aShop.sellCar(new BMW());  
                // 卖出一辆奇瑞QQ  
                aShop.sellCar(new CheryQQ());  
                System.out.println("总收入：" + aShop.getMoney());; M/ p% {# q& t  
        }  
}  
  
运行结果： 8 R+ f% I& f4 \_3 e( Z: f0 x  
/ S0 R: e5 Q- b1 u$ Z  
车型：BMW  单价：300000   
车型：CheryQQ  单价：20000 5 B  \_0 ?# W6 H! \_# r  
总收入：320000 + j; P2 e\* K' k, q0 d1 w: G9 R  
  
继承是多态得以实现的基础。从字面上理解，多态就是一种类型（都是Car类型）表现出多种状态（宝马汽车的名称是BMW，售价是300000；奇瑞汽车的名称是CheryQQ，售价是2000）。将一个方法调用同这个方法所属的主体（也就是对象或类）关联起来叫做绑定，分前期绑定和后期绑定两种。下面解释一下它们的定义： ' d3 Z& f  b# b+ O$ O3 d" R9 v  
  
前期绑定：在程序运行之前进行绑定，由编译器和连接程序实现，又叫做静态绑定。比如static方法和final方法，注意，这里也包括private方法，因为它是隐式final的。 \* i' x9 V\* o3 |  \_3 v3 \_  
后期绑定：在运行时根据对象的类型进行绑定，由方法调用机制实现，因此又叫做动态绑定，或者运行时绑定。除了前期绑定外的所有方法都属于后期绑定。   
  
多态就是在后期绑定这种机制上实现的。多态给我们带来的好处是消除了类之间的耦合关系，使程序更容易扩展。比如在上例中，新增加一种类型汽车的销售，只需要让新定义的类继承Car类并实现它的所有方法，而无需对原有代码做任何修改，CarShop类的sellCar(Car car)方法就可以处理新的车型了。新增代码如下： 3 U6 d0 J" p& y  
Java代码   
// 桑塔纳汽车     
class Santana implements Car {     
    public String getName() {   & \: C# y3 ^' ?+ A7 f  
        return "Santana";   % A5 K' o4 N) o  
    }     
  7 g$ a2 f& c\* }  
    public int getPrice() {     h6 g  u+ S0 }" S5 x  
        return 80000;   ; [0 p" M6 e5 T! ]6 }) i  
    }   " u8 Y, ~, [9 o. j( ?: C6 y+ ~: \  
}  % o+ v3 T4 K, h# x! ~9 a+ U  
  
// 桑塔纳汽车  
class Santana implements Car {4 P\* X2 m9 P$ |3 x  
        public String getName() {  
                return "Santana";  
        }4 L! m" u9 I9 X1 Q6 J# D5 l7 R  
  
        public int getPrice() {  
                return 80000;  
        }  
}  
  
7 P+ b3 ~/ F+ E" e  
重载(overloading)和重写(overriding) 3 g6 n0 F8 W5 R) P  
  
重载和重写都是针对方法的概念，在弄清楚这两个概念之前，我们先来了解一下什么叫方法的型构（英文名是signature，有的译作“签名”，虽然它被使用的较为广泛，但是这个翻译不准确的）。型构就是指方法的组成结构，具体包括方法的名称和参数，涵盖参数的数量、类型以及出现的顺序，但是不包括方法的返回值类型，访问权限修饰符，以及abstract、static、final等修饰符。比如下面两个就是具有相同型构的方法： ( m4 K1 ~$ O, c\* m" I& R9 D" ]( R9 x  
Java代码 - I& W1 M$ n/ G  j+ u\* N& [  
public void method(int i, String s) {     
    // do something     
}     
  ' X) H6 c; N" c0 F7 G  
public String method(int i, String s) {   6 v5 y; r+ y8 Z$ C1 u& ]4 L- O. |  
    // do something     
}    
8 C% k4 J/ \_4 L% d$ a  
public void method(int i, String s) {3 ]& j1 n4 {6 @4 R0 a9 B  
        // do something% P1 v\* @5 Y( B$ @7 V/ h: s# ]  
}  
0 g0 B+ ]( G, Q$ G  
public String method(int i, String s) {3 n9 I5 `% B1 w# V: T: {( b  
        // do something  
}  
, H0 c+ J4 S+ Z9 Z- }9 v9 d+ k# M  
而这两个就是具有不同型构的方法： : X8 z\* \' ^6 F  
Java代码   
public void method(int i, String s) {     
    // do something     
}     
  : \- m8 ~! G. J+ h  
public void method(String s, int i) {     
    // do something     
}  1 L/ Z" g2 a6 z- J; D) h  
  
public void method(int i, String s) {4 v7 @& r9 s8 `, M/ R  
        // do something  `8 I& Y# l  ]9 H/ Y2 I  
}  
\* P  X, G+ z  K/ `  
public void method(String s, int i) {  
        // do something" k3 J6 g6 l5 J7 B\* }  
}+ ], U: n. G/ r, Y  
  
了解完型构的概念后我们再来看看重载和重写，请看它们的定义：   
  
**重写，英文名是overriding**，是指在继承情况下，子类中定义了与其基类中方法具有相同型构的新方法，就叫做子类把基类的方法重写了。这是实现多态必须的步骤。   
**重载，英文名是overloading**，是指在同一个类中定义了一个以上具有相同名称，但是型构不同的方法。在同一个类中，是不允许定义多于一个的具有相同型构的方法的。 1 `0 n+ L, ]7 n7 I0 z  
- T; X0 Z6 W8 S; x0 h9 n' R  
我们来考虑一个有趣的问题：构造器可以被重载吗？答案当然是可以的，我们在实际的编程中也经常这么做。实际上构造器也是一个方法，构造器名就是方法名，构造器参数就是方法参数，而它的返回值就是新创建的类的实例。但是构造器却不可以被子类重写，因为子类无法定义与基类具有相同型构的构造器。