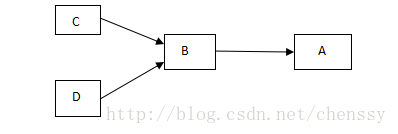
1：父类有默认构造器，子类默认调用父类的构造器，如果父类没有默认构造器，我们就要必须显示的使用super()来调用父类构造器，否则编译器会报错：无法找到符合父类形式的构造器。

2：其实在继承链中对象方法的调用存在一个优先级：this.show(O)、super.show(O)、this.show((super)O)、super.show((super)O)。

1. 经典案例：**public** **class** A {
2. **public** String show(D obj) {
3. **return** ("A and D");
4. }
6. **public** String show(A obj) {
7. **return** ("A and A");
8. }
10. }
12. **public** **class** B **extends** A{
13. **public** String show(B obj){
14. **return** ("B and B");
15. }
17. **public** String show(A obj){
18. **return** ("B and A");
19. }
20. }
22. **public** **class** C **extends** B{
24. }
26. **public** **class** D **extends** B{
28. }
30. **public** **class** Test {
31. **public** **static** **void** main(String[] args) {
32. A a1 = **new** A();
33. A a2 = **new** B();
34. B b = **new** B();
35. C c = **new** C();
36. D d = **new** D();
38. System.out.println("1--" + a1.show(b));
39. System.out.println("2--" + a1.show(c));
40. System.out.println("3--" + a1.show(d));
41. System.out.println("4--" + a2.show(b));
42. System.out.println("5--" + a2.show(c));
43. System.out.println("6--" + a2.show(d));
44. System.out.println("7--" + b.show(b));
45. System.out.println("8--" + b.show(c));
46. System.out.println("9--" + b.show(d));
47. }
48. }
49. **结果：**
50. 1--A and A
51. 2--A and A
52. 3--A and D
53. 4--B and A
54. 5--B and A
55. 6--A and D
56. 7--B and B
57. 8--B and B
58. 9--A and D

从上面的程序中我们可以看出A、B、C、D存在如下关系。



首先我们分析5，a2.show(c)，a2是A类型的引用变量，所以this就代表了A，a2.show(c),它在A类中找发现没有找到，于是到A的超类中找(super)，由于A没有超类（Object除外），所以跳到第三级，也就是this.show((super)O)，C的超类有B、A，所以(super)O为B、A，this同样是A，这里在A中找到了show(A obj)，同时由于a2是B类的一个引用且B类重写了show(A obj)，因此最终会调用子类B类的show(A obj)方法，结果也就是B and A。

当超类对象引用变量引用子类对象时，被引用对象的类型而不是引用变量的类型决定了调用谁的成员方法，但是这个被调用的方法必须是在超类中定义过的，也就是说被子类覆盖的方法。例如：a2.show(b)；

这里a2是引用变量，为A类型，它引用的是B对象，因此按照上面那句话的意思是说有B来决定调用谁的方法,所以a2.show(b)应该要调用B中的show(B obj)，产生的结果应该是“B and B”，但是为什么会与前面的运行结果产生差异呢？这里我们忽略了后面那句话“但是这儿被调用的方法必须是在超类中定义过的”，那么show(B obj)在A类中存在吗？根本就不存在！所以这句话在这里不适用？那么难道是这句话错误了？非也！其实这句话还隐含这这句话：它仍然要按照继承链中调用方法的优先级来确认。所以它才会在A类中找到show(A obj)，同时由于B重写了该方法所以才会调用B类中的方法，否则就会调用A类中的方法。

所以多态机制遵循的原则概括为：当超类对象引用变量引用子类对象时，被引用对象的类型而不是引用变量的类型决定了调用谁的成员方法，但是这个被调用的方法必须是在超类中定义过的，也就是说被子类覆盖的方法，但是它仍然要根据继承链中方法调用的优先级来确认方法，该优先级为：this.show(O)、super.show(O)、this.show((super)O)、super.show((super)O)。

3： abstract不能与final并列修饰同一个类。abstract 不能与private、static、final或native并列修饰同一个方法。

4：接口本身就不是类，接口是用来建立类与类之间的协议，它所提供的只是一种形式，而没有具体的实现。所有方法访问权限自动被声明为public。确切的说只能为public，实现接口的非抽象类必须要实现该接口的所有方法。抽象类可以不用实现。可以使用 instanceof 检查一个对象是否实现了某个特定的接口。例如：if(anObject instanceof Comparable){}。在某种程度上来说，接口是抽象类的特殊化。

5：利用序列化实现对象的拷贝；

1. **public** **class** CloneUtils {
2. @SuppressWarnings("unchecked")
3. **public** **static** <T **extends** Serializable> T clone(T obj){
4. T cloneObj = **null**;
5. **try** {
6. //写入字节流
7. ByteArrayOutputStream out = **new** ByteArrayOutputStream();
8. ObjectOutputStream obs = **new** ObjectOutputStream(out);
9. obs.writeObject(obj);
10. obs.close();
12. //分配内存，写入原始对象，生成新对象
13. ByteArrayInputStream ios = **new** ByteArrayInputStream(out.toByteArray());
14. ObjectInputStream ois = **new** ObjectInputStream(ios);
15. //返回生成的新对象
16. cloneObj = (T) ois.readObject();
17. ois.close();
18. } **catch** (Exception e) {
19. e.printStackTrace();
20. }
21. **return** cloneObj;
22. }
23. }

1. **public** **class** Client {
2. **public** **static** **void** main(String[] args) {
3. //写封邮件
4. Email email = **new** Email("请参加会议","请与今天12:30到二会议室参加会议...");
6. Person person1 =  **new** Person("张三",email);
8. Person person2 =  CloneUtils.clone(person1);
9. person2.setName("李四");
10. Person person3 =  CloneUtils.clone(person1);
11. person3.setName("王五");
12. person1.getEmail().setContent("请与今天12:00到二会议室参加会议...");
14. System.out.println(person1.getName() + "的邮件内容是：" + person1.getEmail().getContent());
15. System.out.println(person2.getName() + "的邮件内容是：" + person2.getEmail().getContent());
16. System.out.println(person3.getName() + "的邮件内容是：" + person3.getEmail().getContent());
17. }
18. }
19. -------------------
20. Output:
21. 张三的邮件内容是：请与今天12:00到二会议室参加会议...
22. 李四的邮件内容是：请与今天12:30到二会议室参加会议...
23. 王五的邮件内容是：请与今天12:30到二会议室参加会议...

所以使用该工具类的对象只要实现Serializable接口就可实现对象的克隆，无须继承Cloneable接口实现clone()方法。

6：使用内部类最吸引人的原因是：每个内部类都能独立地继承一个（接口的）实现，所以无论外围类是否已经继承了某个（接口的）实现，对于内部类都没有影响。只能使用内部类才能实现多重继承了。

在成员内部类中要注意两点，第一：成员内部类中不能存在任何static的变量和方法；第二：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。

局部内部类是嵌套在方法和作用于内的，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

静态内部类：它的创建是不需要依赖于外围类的。 它不能使用任何外围类的非static成员变量和方法。

7：强制转换：Father father = new Father();           Son son = (Son) father;这个系统会抛出ClassCastException异常信息。在继承中，子类可以自动转型为父类，但是父类强制转换为子类时只有当引用类型真正的身份为子类时才会强制转换成功，否则失败。例： Father father = new Son();       Son son = (Son) father；

8：构造代码块：编译器会将代码块按照他们的顺序(假如有多个代码块)插入到所有的构造函数的最前端，这样就能保证不管调用哪个构造函数都会执行所有的构造代码块

1. **public** **class** Test {
2. /\*\*
3. \* 构造代码
4. \*/
5. {
6. System.out.println("执行构造代码块...");
7. }
9. /\*\*
10. \* 无参构造函数
11. \*/
12. **public** Test(){
13. System.out.println("执行无参构造函数...");
14. }
15. /\*\*
16. \* 有参构造函数
17. \* @param id  id
18. \*/
19. **public** Test(String id){
20. System.out.println("执行有参构造函数...");
21. }
22. }

等同于：

1. **public** **class** Test {
2. /\*\*
3. \* 无参构造函数
4. \*/
5. **public** Test(){
6. System.out.println("执行构造代码块...");
7. System.out.println("执行无参构造函数...");
8. }
10. /\*\*
11. \* 有参构造函数
12. \* @param id  id
13. \*/
14. **public** Test(String id){
15. System.out.println("执行构造代码块...");
16. System.out.println("执行有参构造函数...");
17. }
19. }
21. 运行结果
23. **public** **static** **void** main(String[] args) {
24. **new** Test();
25. System.out.println("----------------");
26. **new** Test("1");
27. }
28. ------------
29. Output:
30. 执行构造代码块...
31. 执行无参构造函数...
32. ----------------
33. 执行构造代码块...
34. 执行有参构造函数...

执行顺序静态代码块 > 构造代码块 > 构造函数。

9：在[Java](http://lib.csdn.net/base/java)中int类型数据的大小比较可以使用双等号，double和float类型则不能使用双等号来比较大小，如果使用的话得到的结果将永远是不相等，即使两者的精度是相同的也不可以。如果要比较的两个double数据的字符串精度相等，可以将数据转换成string然后借助string的equals方法来间接实现比较两个double数据是否相等。注意这种方法只适用于比较精度相同的数据，并且是只用用于比较是否相等的情况下，不能用来判断大小。

float类型: 使用Float.foatToIntBits转换成int类型，然后使用==。

double类型: 使用Double.doubleToLongBit转换成long类型，然后使用==。

10：判断一个对象是否是某个类的实例：用instancof或者if(s.getClass()==String.getClass())；覆写equals时推荐使用getClass进行类型判断。而不是使用instanceof。

1. **public** **class** Person {
2. **protected** String name;
4. **public** String getName() {
5. **return** name;
6. }
8. **public** **void** setName(String name) {
9. **this**.name = name;
10. }
12. **public** Person(String name){
13. **this**.name = name;
14. }
16. **public** **boolean** equals(Object object){
17. **if**(object **instanceof** Person){
18. Person p = (Person) object;
19. **if**(p.getName() == **null** || name == **null**){
20. **return** **false**;
21. }
22. **else**{
23. **return** name.equalsIgnoreCase(p.getName());
24. }
25. }
26. **return** **false**;
27. }
28. }
29. **public** **class** Employee **extends** Person{
30. **private** **int** id;
32. **public** **int** getId() {
33. **return** id;
34. }
36. **public** **void** setId(**int** id) {
37. **this**.id = id;
38. }
40. **public** Employee(String name,**int** id){
41. **super**(name);
42. **this**.id = id;
43. }
45. /\*\*
46. \* 重写equals()方法
47. \*/
48. **public** **boolean** equals(Object object){
49. **if**(object **instanceof** Employee){
50. Employee e = (Employee) object;
51. **return** **super**.equals(object) && e.getId() == id;
52. }
53. **return** **false**;
54. }
55. }

1. **public** **class** Test {
2. **public** **static** **void** main(String[] args) {
3. Employee e1 = **new** Employee("chenssy", 23);
4. Employee e2 = **new** Employee("chenssy", 24);
5. Person p1 = **new** Person("chenssy");
7. System.out.println(p1.equals(e1));
8. System.out.println(p1.equals(e2));
9. System.out.println(e1.equals(e2));
10. }
11. }

结果：true、true、false。但是p1即等于e1也等于e2，这是非常奇怪的，因为e1、e2明明是两个不同的类，但为什么会出现这个情况？首先p1.equals(e1)，是调用p1的equals方法，该方法使用instanceof关键字来检查e1是否为Person类，这里我们再看看instanceof：判断其左边对象是否为其右边类的实例，也可以用来判断继承中的子类的实例是否为父类的实现。他们两者存在继承关系，肯定会返回true了，而两者name又相同，所以结果肯定是true。

故覆写equals时推荐使用getClass进行类型判断。

11：String并不是基本数据类型，而是一个对象，并且是不可变的对象。查看源码就会发现String类为final型的（当然也不可被继承），而且通过查看JDK文档会发现几乎每一个修改String对象的操作，实际上都是创建了一个全新的String对象。

””、null、new String()三者的区别。null 表示string还没有new ，也就是说对象的引用还没有创建，也没有分配内存空间给他，而””、new String()则说明了已经new了，只不过内部为空，但是它创建了对象的引用，是需要分配内存空间的。

在字符串中存在一个非常特殊的地方，那就是字符串池。每当我们创建一个字符串对象时，首先就会检查字符串池中是否存在面值相等的字符串，如果有，则不再创建，直接放回字符串池中对该对象的引用，若没有则创建然后放入到字符串池中并且返回新建对象的引用。这个机制是非常有用的，因为可以提高效率，减少了内存空间的占用。所以在使用字符串的过程中，推荐使用直接赋值（即String s=”aa”），除非有必要才会新建一个String对象（即String s = new String(”aa”)）。

String name = ”I ” + ”am ” + ”chenssy ” ;

StringBuffer name = new StringBuffer(”I ”).append(” am ”).append(” chenssy ”);

对于这两种方式，你会发现第一种比第二种快太多了，JVM做了一下优化处理，其实String name = ”I ” + ”am ” + ”chenssy ” ;在JVM眼中就是String name = ”I am chenssy ”

字符串拼接：append()速度最快，concat()次之，+最慢；除append外其他两个在每次拼接时都创建了新对象。

12：所有被final标注的方法都是不能被继承、更改的，所以对于final方法使用的第一个原因就是方法锁定，以防止任何子类来对它的修改；父类的final方法是不能被子类所覆盖的，也就是说子类是不能够存在和父类一模一样的方法的。如果将一个方法指明为final，就是同意编译器将针对该方法的所有调用都转为内嵌调用。类似于c++中的内联，函数函数之间的调用其实是比较耗性能的，内联函数相当于在编译时直接拷贝一个代码的副本，省去了函数调用时内存跳转。但是如果内联函数很庞大会消耗多余的内存。

13: asList返回的是一个长度不可变的列表。数组是多长，转换成的列表是多长，我们是无法通过add、remove来增加或者减少其长度的。

14: 与ArrayList相似，但是Vector是同步的。所以说Vector是线程安全的动态数组。它的操作与ArrayList几乎一样。如果集合中的元素的数目大于目前集合数组的长度时，vector增长率为目前数组长度的100%,而arraylist增长率为目前数组长度的50%.如过在集合中使用数据量比较大的数据，用vector有一定的优势。

15: 由于TreeMap需要维持内部元素的顺序，所以它通常要比HashMap和HashTable慢。

16: Java中最常用的两种结构是数组和模拟指针(引用)，几乎所有的数据结构都可以利用这两种来组合实现，HashMap也是如此。实际上HashMap是一个“链表散列” ,HashMap底层实现还是数组，只是数组的每一项都是一条链.

17: 对于HashSet而言，它是基于HashMap来实现的，底层采用HashMap来保存元素, HashSet中的所有元素都是保存在HashMap的key中

18: fail-fast”机制的（也就是快速失败）。所谓快速失败就是在并发集合中，其进行迭代操作时，若有其他线程对其进行结构性的修改，这时迭代器会立马感知到，并且立即抛出ConcurrentModificationException异常，而不是等到迭代完成之后才告诉你（你已经出错了）。

19: 1、判断两个对象的hashcode是否相等，若不等，则认为两个对象不等，完毕，若相等，则比较equals。

 2、若两个对象的equals不等，则可以认为两个对象不等，否则认为他们相等。

20: treeMap 底层通过红黑树实现，一般用于单线程

21: “快速失败”也就是fail-fast，它是Java集合的一种错误检测机制。当多个线程对集合进行结构上的改变的操作时，有可能会产生fail-fast机制。记住是有可能，而不是一定。例如：假设存在两个线程（线程1、线程2），线程1通过Iterator在遍历集合A中的元素，在某个时候线程2修改了集合A的结构（是结构上面的修改，而不是简单的修改集合元素的内容），那么这个时候程序就会抛出 ConcurrentModificationException 异常，从而产生fail-fast机制。

在遍历过程中所有涉及到改变modCount值得地方全部加上synchronized或者直接使用Collections.synchronizedList，这样就可以解决。但是不推荐，因为增删造成的同步锁可能会阻塞遍历操作。或者使用CopyOnWriteArrayList来替换ArrayList。推荐使用该方案。CopyOnWriteArrayList为何物？ArrayList 的一个线程安全的变体，其中所有可变操作（add、set 等等）都是通过对底层数组进行一次新的复制来实现的。 该类产生的开销比较大，但是在两种情况下，它非常适合使用。1：在不能或不想进行同步遍历，但又需要从并发线程中排除冲突时。2：当遍历操作的数量大大超过可变操作的数量时。遇到这两种情况使用CopyOnWriteArrayList来替代ArrayList再适合不过了。CopyOnWriterArrayList的无论是从数据结构、定义都和ArrayList一样。它和ArrayList一样，同样是实现List接口，底层使用数组实现。在方法上也包含add、remove、clear、iterator等方法。CopyOnWriterArrayList不会抛ConcurrentModificationException异常。他们所展现的魅力就在于copy原来的array，再在copy数组上进行add操作，这样做就完全不会影响COWIterator中的array了。

22: 在使用asList时不要将基本数据类型当做参数。要使用基本数据类型的封装类型..asList返回的列表只不过是一个披着list的外衣，它并没有list的基本特性（变长）。该list是一个长度不可变的列表，传入参数的数组有多长，其返回的列表就只能是多长。

23: 使用subString方法来对String对象进行分割处理，同时我们也可以使用subList、subMap、subSet来对List、Map、Set进行分割处理. subList返回的SubList同样也是AbstractList的子类，同时它的方法如get、set、add、remove等都是在原列表上面做操作，它并没有像subString一样生成一个新的对象。所以subList返回的只是原列表的一个视图，它所有的操作最终都会作用在原列表上。对于子列表视图，它是动态生成的，生成之后就不要操作原列表了，否则必然都导致视图的不稳定而抛出异常。最好的办法就是将原列表设置为只读状态，要操作就操作子列表.. 例如，有一个列表存在1000条记录，我们需要删除100-200位置处的数据，可能我们会这样处理：list1.subList(100,200).clear();