[一、什么是面向对象 3](#_Toc26825)

[1、面向过程思想 3](#_Toc22848)

[2、面向对象思想 3](#_Toc10421)

[二、类与对象 6](#_Toc251)

[三、封装 8](#_Toc26789)

[1、封装的概念 9](#_Toc21216)

[2、封装的好处 9](#_Toc6260)

[3、封装的原则 9](#_Toc17180)

[4、private关键字 9](#_Toc30828)

[四、构造方法 10](#_Toc599)

[1、构造方法详解 10](#_Toc18931)

[2、类的初始化过程 13](#_Toc23099)

[3、this关键字 13](#_Toc28287)

[五、static关键字 14](#_Toc13068)

[1、static的特点 14](#_Toc12102)

[2、static关键字注意事项 15](#_Toc13096)

[3、成员变量和静态变量的区别 15](#_Toc31553)

[4、主方法main解析 15](#_Toc12139)

[5、静态何时使用？ 16](#_Toc5757)

[6、静态代码块、构造代码块、构造方法 18](#_Toc7111)

[六、面向对象——继承 19](#_Toc19670)

[1、继承概述 19](#_Toc26005)

[2、继承注意事项 21](#_Toc2948)

[3、阻止继承：final类和方法 26](#_Toc11620)

[4、访问控制—public、protected、default、private 28](#_Toc8235)

[七、面向对象——抽象类 29](#_Toc17493)

[1、什么是抽象？ 29](#_Toc4424)

[2、抽象类 29](#_Toc12615)

[3、抽象类的几个问题 31](#_Toc25398)

[4、抽象类和一般类的异同点 32](#_Toc25687)

[八、接口 32](#_Toc30235)

[1、接口的概念 33](#_Toc13192)

[2、接口的特点 33](#_Toc19583)

[3、接口成员的特点 33](#_Toc20884)

[4、接口必须掌握的知识 33](#_Toc18266)

[5、抽象类和接口的异同点 34](#_Toc14373)

[九、多态性 38](#_Toc3039)

[1. 编译期多态（静态多态） 38](#_Toc2321)

[2. 运行期多态（动态多态） 38](#_Toc10317)

[3. 总结 45](#_Toc18002)

[十、内部类 45](#_Toc30855)

[1、常规内部类 45](#_Toc32457)

[2、静态内部类 45](#_Toc4174)

[3、局部内部类 46](#_Toc15888)

[4、匿名内部类 47](#_Toc24645)

Java面向对象 ：什么是面向对象、类与对象、封装、构造方法、static关键字、继承、抽象类、接口、多态

**一、什么是面向对象**

**1、面向过程思想**

面向过程：（PO，Procedure Oriented）

在理解面向对象思想之前，我首先来回顾一下以前C语言中的面向过程思想，面向过程思想强调的是过程（动作），在面向过程的开发中，其实就是面向着具体的每一个步骤和过程，把每一个步骤和过程完成，然而由这些功能方法相互调用，完成需求。面向过程的典型代表语言是：C语言。

**2、面向对象思想**

面向对象：（OO, Object-Oriented）

①面向对象思想的由来

随着需求的更改，功能的增多，发现需要面对每一个步骤太麻烦了，这时候就在思考，能不能把这些步骤和方法封装起来，根据不同的功能，进行不同的封装，用的时候，找到对应的类就可以了。这就是面向对象的思想。那么，到底什么是面向对象思想呢？

②面向对象思想概述

a、概念：面向对象是基于对象的编程思想。强调的是对象（实体），典型的语言是：C++、Java。

b、面向对象思想的特点：

* 是一种更符合我们思想习惯的思想
* 可以将复杂的事情简单化
* 是我们从执行者变成了指挥者，角色发生了转变。

c、在完成需求时，按照面向对象的思想，我们应该这样做：

* 首先，我们去寻找具备所需功能的对象来使用。
* 如果不存在这样的对象，那么我们就应该创建一个具备所需功能的对象。
* 这样可以提高代码的复用性，简化开发。

③面向对象思想举例  
a、买电脑：

* 面向过程：我得先了解电脑——了解我自己的需求——找对应的参数信息——去电脑城买电脑——讨价还价——买回电脑。
* 面向对象：我知道我要买电脑——找一个了解电脑和会讨价还价的人去买——别人帮我买回来了。

b、洗衣服：

* 面向过程：把衣服脱下——找一个盆——放点洗衣服——加点水——把衣服扔进去——搓一搓——请洗衣服——晾干——晒干
* 面向对象：我要洗衣服——把衣服脱下来——找到洗衣机——丢进去，开启洗衣机——洗完了，晒干。

④面向对象代码体现：

1. **public** **class** Demo1{
2. /\*
3. 把啤酒放进冰箱。
4. \*/
5. **public** **static** **void** main(String[] args){
6. /\*
7. 面向过程：1、打开冰箱门。
8. 2、把啤酒放进去。
9. 3、关冰箱门
10. \*/
11. //面向过程，调用方法：
12. open();
13. in();
14. close();
16. /\*
17. 面向对象：
18. 如何思考才能更符合面向对象呢？
19. 1、有那些类？
20. 2、每个类有什么东西呢？
21. 3、类与类之间有什么关系呢？
22. 把啤酒放进冰箱的面向分析（名词提取法）
23. a、有几个类？大象，冰箱，Demo类
24. b、每个类有哪些东西呢？
25. ————啤酒：进去
26. ————冰箱：开，关
27. ————Demo：main方法
28. \*/
29. //面向对象，创建对象，对象调用方法
30. Beer beer = **new** Beer();
31. Fridge fri = **new** Fridge();
32. fri.open();
33. fri.in();
34. fri.close();
35. }
37. **public** **static** **void** open(){
38. System.out.println("面向过程：打开冰箱");
39. }
40. **public** **static** **void** in(){
41. System.out.println("面向过程：把啤酒放进冰箱。");
42. }
43. **public** **static** **void** close(){
44. System.out.println("面向过程：关冰箱");
45. }
46. }
47. **class** Beer{
48. **public** **static** **void** in(){
49. System.out.println("面向对象：把啤酒放进去");
50. }
51. }
52. **class** Fridge{
53. **public** **static** **void** open(){
54. System.out.println("面向对象：打开冰箱");
55. }
56. **public** **static** **void** close(){
57. System.out.println("面向对象：关上冰箱");
58. }
59. }

我们在面向对象的开发中，就是不断的创建对象，使用对象，指挥对象做事情。面向对象的设计过程，其实就是管理和维护对象之间的关系。  
⑤面向对象的思维方法

* 首先确定谁来做，其次确定怎么做。
* 首先考虑是整体，其次考虑是局部。
* 首先考虑是抽象，其次考虑是具体。

⑥面向对象的三大特征  
面向对象有三大特征，分别是：封装（encapsulation）、继承（inheritance）、多态（polymorphism）。（若问四大特征：抽象-abstract）

**二、类与对象**

1、什么是类？什么是对象？

①类和对象的概念

类是一组相关的属性和行为的集合。是构造对象的模板或蓝图。是一个抽象的概念。

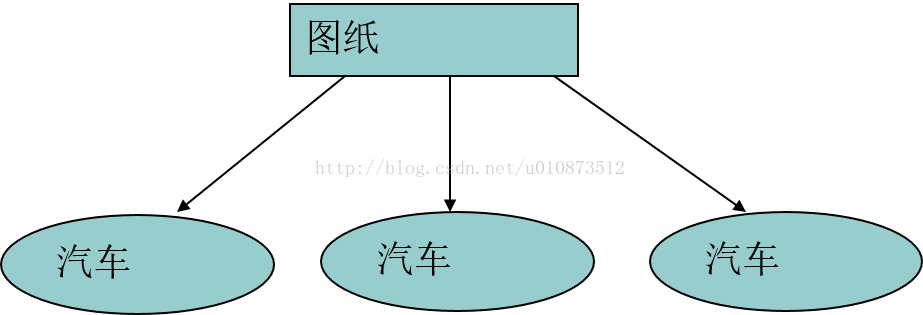
对象是该类事物的具体表现形式。是一个具体存在的个体。

②类和对象的关系

对事物的描述通常包括两方面：一个是属性，一个是行为。只要明确该事物的属性和行为并定义在类中即可。对象其实就是该类事物实实在在存在的个体。

类和对象的关系可以理解成：类——对事物的描述。对象——该类事物的示例，在java中通过new创建来的。

类与对象的关系如图所示（图纸就是类，汽车就是对象）：



③成员变量

定义类其实就是定义类的成员（成员变量和成员方法）

成员变量其实就是类中的属性。成员方法就是类中的行为。

成员变量和局部变量的区别：

* 成员变量定义在类中，整个类都可以访问；局部变量定义在方法中，语句局部代码快中，只在所属的区域有效。
* 成员变量存在于堆内存的对象中；局部变量存在于栈内存的方法中。
* 成员变量随着对象的创建而存在，随着对象的小时而消失；局部变量随着所属区域的执行而存在，随着所属区域的结束而释放。
* 成员变量都有默认的初始化值；局部变量没有默认的初始化值，必须先定义，赋值，才能使用。

成员变量和局部变量能否一样呢？可以，但是使用的时候要注意，先找小范围，再找大范围。  
④类与对象的代码示例：

1. /\*
2. 事物：
3. 属性  事物的信息描述
4. 行为  事物的功能
6. 类：
7. 成员变量    事物的属性
8. 成员方法    事物的行为
10. 定义一个类，其实就是定义该类的成员变量和成员方法。
12. 案例：我们来完成一个学生类的定义。
14. 学生事物：
15. 属性：姓名，年龄，地址...
16. 行为：学习，吃饭，睡觉...
18. 把事物要转换为对应的类：
20. 学生类：
21. 成员变量：姓名，年龄，地址...
22. 成员方法：学习，吃饭，睡觉...
24. 成员变量：和以前变量的定义是一样的格式，但是位置不同，在类中方法外。
25. 成员方法：和以前的方法定义是一样的格式，但是今天把static先去掉。
27. 首先我们应该定义一个类，然后完成类的成员。
28. \*/
29. //这是我的学生类
30. **class** Student {
31. //定义变量
32. //姓名
33. String name;
34. //年龄
35. **int** age;
36. //地址
37. String address;
39. //定义方法
40. //学习的方法
41. **public** **void** study() {
42. System.out.println("学生爱学习");
43. }
44. //吃饭的方法
45. **public** **void** eat() {
46. System.out.println("学习饿了,要吃饭");
47. }
48. //睡觉的方法
49. **public** **void** sleep() {
50. System.out.println("学习累了,要睡觉");
51. }
52. }

**三、封装**

**1、封装的概念**

封装是指隐藏对象的属性和细节，但对外提供公共的访问方式。

**2、封装的好处**

封装的好处主要在于：隐藏实现细节，提供公共的访问方式；将变化隔离；便于使用；提高代码的复用性；提高安全性。

**3、封装的原则**

封装原则主要是以下两点：

a、将不需要对外提供的内容都隐藏起来。

b、把属性都隐藏，提供公共方法对其调用。

**4、private关键字**

①private：私有，是一个权限修饰符，用于修饰成员。被private修饰的成员只有在本类中才能访问。私有仅仅是封装的一种体现而已。

②private常见的应用：将成员变量私有化，对外提供set和get方法对其进行访问，提高数据访问的安全性。

封装的代码示例如下：

1. /\*
2. 封装和private的应用：
3. A:把成员变量用private修饰
4. B:提高对应的getXxx()和setXxx()方法
5. \*/
6. //定义学生类
7. **class** Student {
8. //姓名
9. **private** String name;
10. //年龄
11. **private** **int** age;
13. //姓名获取值
14. **public** String getName() {
15. **return** name;
16. }
18. //姓名设置值
19. **public** **void** setName(String n) {
20. name = n;
21. }
23. //年龄获取值
24. **public** **int** getAge() {
25. **return** age;
26. }
28. //年龄赋值
29. **public** **void** setAge(**int** a) {
30. age = a;
31. }
32. }
33. //测试类
34. **class** StudentTest {
35. **public** **static** **void** main(String[] args) {
36. //创建学生对象
37. Student s = **new** Student();
39. //使用成员变量
40. //错误：被私有修饰了，外界不能直接访问了
41. //System.out.println(s.name+"---"+s.age);
42. System.out.println(s.getName()+"---"+s.getAge());
44. //给成员变量赋值
45. //s.name = "林青霞";
46. //s.age = 27;
47. //通过方法给赋值
48. s.setName("黄祥");
49. s.setAge(20);
50. System.out.println(s.getName()+"---"+s.getAge());
51. }
52. }

**四、构造方法**

**1、构造方法详解**

①什么是构造方法：构造方法是一种特殊的方法。具有以下特点：

a，构造方法的方法名必须与类名相同。

b，构造方法没有返回值类型，也不能定义为void，在方法名面前不声明方法类型。

c，构造方法的主要作用是完成对象的初始化动作，它能把定义对象时的参数传递给对象的域。

d，一个类可以定义多个构造方法，如果在定义类时没有定义构造方法，则编译系统会自动插入一个无參的默认构造器，这个构造器不执行任何代码。

e，构造方法可以重载，以参数的个数，类型，顺序。

②构造方法的作用：给对象进行初始化。

③构造方法和一般方法的区别：

* 构造方法：对象创建时，就会调用与之对应的构造方法，对对象进行初始化；

           一般方法：对象创建后，需要方法功能时才会调用。

* 构造方法，在对象创建时，会调用且只调用一次。

           一般方法，对象创建后，可以被调用多次。

④什么时候定义构造方法：在描述事物时，该事物已存在就具备的一些内容，这些内容都定义在构造方法中。  
⑤构造代码块  
a，作用：给对象进行初始化，对象一建立就运行，而且优于构造方法运行。  
b，和构造方法的区别：

* 构造代码块是给对象进行初始化，构造方法是给指定的对象进行初始化。构造代码块中定义的是不同对象的共性内容。

⑥构造方法代码示例：

1. /\*
2. 构造方法代码演示
3. \*/
4. **class** Student {
5. **private** String name;
6. **private** **int** age;
8. **public** Student() {
9. //System.out.println("我给了，你还给不");
10. System.out.println("这是无参构造方法");
11. }
13. //构造方法的重载格式
14. **public** Student(String name) {
15. System.out.println("这是带一个String类型的构造方法");
16. **this**.name = name;
17. }
19. **public** Student(**int** age) {
20. System.out.println("这是带一个int类型的构造方法");
21. **this**.age = age;
22. }
24. **public** Student(String name,**int** age) {
25. System.out.println("这是一个带多个参数的构造方法");
26. **this**.name = name;
27. **this**.age = age;
28. }
30. **public** **void** show() {
31. System.out.println(name+"---"+age);
32. }
33. }
35. **class** ConstructDemo {
36. **public** **static** **void** main(String[] args) {
37. //创建对象
38. Student s = **new** Student();
39. s.show();
40. System.out.println("-------------");
42. //创建对象2
43. Student s2 = **new** Student("林青霞");
44. s2.show();
45. System.out.println("-------------");
47. //创建对象3
48. Student s3 = **new** Student(27);
49. s3.show();
50. System.out.println("-------------");
52. //创建对象4
53. Student s4 = **new** Student("林青霞",27);
54. s4.show();
55. }
56. }

**2、类的初始化过程**

Student s = new Student();在内存中究竟做了哪些事情呢？  
①加载student.class文件进内存。  
②为栈内存s开辟空间。  
③为堆内存学生对象开辟空间。  
④对学生对象的成员变量进行默认初始化。  
⑤对学生对象的成员变量进行显示初始化。  
⑥通过构造方法对学生对象的成员变量进行赋值。  
⑦学生对象初始化完毕，将对象地址赋值给s变量。

**3、this关键字**

①this关键字的含义：代表对象，this就是所在方法的所属对象的引用。简单的说，哪个对象调用了this所在的方法，this就代表哪个对象。

②this关键字的应用

a、当成员变量和局部变量重名，可以用关键字this来区分。

b、this也可以用于在构造方法中调用其它构造方法。

注意：this只能定义在构造方法的第一行，因为初始化动作必须要先执行。

如下代码所示：

1. /\*
2. this的实例。
4. this:哪个对象调用那个方法，this就代表那个对象
5. \*/
6. **class** Student {
7. **private** String name;
8. **private** **int** age;
10. **public** String getName() {
11. **return** name; //这里其实是隐含了this
12. }
14. **public** **void** setName(String name) {
15. **this**.name = name;
16. }
18. **public** **int** getAge() {
19. **return** age;
20. }
22. **public** **void** setAge(**int** age) {
23. **this**.age = age;
24. }
25. }
27. **class** StudentTest2 {
28. **public** **static** **void** main(String[] args) {
29. //创建一个对象
30. Student s1 = **new** Student();
31. s1.setName("欧阳");
32. s1.setAge(27);
33. System.out.println(s1.getName()+"---"+s1.getAge());
35. //创建第二个对象
36. Student s2 = **new** Student();
37. s2.setName("黄祥");
38. s2.setAge(30);
39. System.out.println(s2.getName()+"---"+s2.getAge());
40. }
41. }

**五、static关键字**

**1、static的特点**

①static是一个修饰符，用于修饰成员。

②static修饰的成员被所有的对象所共享。

③static优先于对象而存在，因为static的成员随着类的加载就已经存在了。

④static修饰的成员多了一种调用方式，就是可以直接被类名所调用，类名.静态成员。

⑤static修饰的数据是共享数据，对象中存在的是特有数据。

**2、static关键字注意事项**

①在静态方法中不能使用this或者super关键字。

②静态方法只能访问静态的成员变量和成员方法。（非静态既可以访问静态也能访问非静态）

③主方法是静态的。

**3、成员变量和静态变量的区别**

①两个变量的生命周期不同

* 成员变量随着对象的创建而存在，随着对象的回收而释放。
* 静态变量随着类的加载而存在，随着类的消失而消失。

②两种变量的调用方式不同

* 成员变量只能被对象调用。
* 静态变量可以被对象调用，还能被类名调用。

③成员变量也称实例变量；静态变量也称类变量。  
④成员变量数据存储在堆内存的对象中，所以也叫对象的特有数据；静态变量数据存储在方法区（共享数据），所以也叫对象的共享数据。

**4、主方法main解析**

①主方法的特殊之处：格式是固定的；被JVM所识别和调用。  
主方法：public static void main (String[] args)

* public ：因为权限必须是最大的。
* static：不需要对象的，直接用主方法所属类名调用即可。
* void：主方法没有具体的返回值。
* main：方法名，不是关键字，只是一个被JVN所识别的固定的名字。
* String[] args：这是主方法的参数列表，是一个数组类型的参数，而且元素都是字符串类型。

**5、静态何时使用？**

①静态变量  
a、当分析对象中所具备的成员变量的值都是固定的，这时这个成员就可以使用静态修饰。  
b、只要数据在对象中都是不同的，就是对象的特有数据，必须存储在对象中，是非静态的。  
c、如果是相同的数据，对象不需要做修改，只需要使用即可，不需要存储在对象中，定义成静态的。  
②静态方法  
a、方法是否用静态修饰，就参考一点，就是该方法的功能是否有访问到对象中特有数据。  
b、简单的说，从源代码看，该功能是狗需要访问到非静态的成员变量，如果需要，该功能就是非静态的；如果不需要，该功能就可以定义成静态的。  
c、但是，非静态需要被对象调用，而仅创建对象调用非静态的，没有访问特有数据的方法，该对象的创建是没有意义的。  
静态方法代码示例：

1. /\*
2. 静态方法
3. \*/
4. **class** Teacher {
5. **public** **int** num = 10;
6. **public** **static** **int** num2 = 20;
8. **public** **void** show() {
9. System.out.println(num); //隐含的告诉你访问的是成员变量
10. System.out.println(**this**.num); //明确的告诉你访问的是成员变量
11. System.out.println(num2);
13. //function();
14. //function2();
15. }
17. **public** **static** **void** method() {
18. //无法从静态上下文中引用非静态 变量 num
19. //System.out.println(num);
20. System.out.println(num2);
22. //无法从静态上下文中引用非静态 方法 function()
23. //function();
24. function2();
25. }
27. **public** **void** function() {
29. }
31. **public** **static** **void** function2() {
33. }
34. }
36. **class** TeacherDemo {
37. **public** **static** **void** main(String[] args) {
38. //创建对象
39. Teacher t = **new** Teacher();
40. t.show();
41. System.out.println("------------");
42. t.method();
43. }
44. }

静态变量代码示例：

1. /\*
2. 静态变量示例
3. \*/
4. **class** Student {
5. //非静态变量
6. **int** num = 10;
8. //静态变量
9. **static** **int** num2 = 20;
10. }
12. **class** StudentDemo {
13. **public** **static** **void** main(String[] args) {
14. Student s = **new** Student();
15. System.out.println(s.num);
17. System.out.println(Student.num2);//静态可通过类名调用。
18. System.out.println(s.num2);//静态变量也可以通过类名调用。
19. }
20. }

**6、静态代码块、构造代码块、构造方法**

1、静态代码块：static{代码} 在类加载时执行，但只会执行一次，不会重复执行。程序启动时需要做的初始化运算或操作，可以用来给静态属性赋值或初始化。静态语句块放在所有静态属性之后定义，为的是方便初始化所有属性，如果定义在某些属性之前，可能造成静态属性的初始化失败。

2、构造代码块：{代码} 构造代码块在创建对象时被调用，每次创建对象都会被调用，并且构造代码块的执行次序优先于类构造函数。

public class Student {

/\*\*

\* 静态代码块，在加载类的时候已经开始执行了 只执行一次

\*/

static {

System.out.println("我是静态代码块！");

}

/\*\*

\* 构造代码块，创建对象实例时执行，发生在调用构造方法前 只执行一次

\*/

{

System.out.println("我是构造代码块！");

}

/\*\*

\* 构造方法，创建对象实例时执行 只执行一次

\*/

public Student() {

System.out.println("我是构造方法！");

}

}

**六、面向对象——继承**

**1、继承概述**

①什么是继承？

继承是面向对象的一个重要方面。当多个类存在相同属性和行为时，将这些类抽取到单独一个类中，那么多个类无需再定义这些属性和行为，只需要继承那个类即可。关键字extends表明正在构造的新生类派生于一个已存在的类。已存在的类被称为超类（superclss）、基类（base class）或父类（parent class）；新类被称为子类（subclass）。“is-a”关系是继承的一个明显特征。

②通过extends关键字来实现类与类的继承，如：  
class 子类名 extends 父类名｛｝  
有了继承以后，我们在定义一个类的时候，可以在一个已经存在的类的基础上，还可以定义自己的新成员。  
但是，我们需要注意的是，在Java中支持单继承，不直接支持多继承。但对c++中的多继承机制进行了改良。  
a、单继承：一个子类只能有一个直接父类  
b、多继承：一个子类可以有多个直接父类。（java中不允许）因为多个父类中有相同成员时，会产生调用的不确定性。  
c、java支持多层继承（多重继承）：如C继承B，B继承A，这样就出现了继承体系。  
当我们需要使用一个继承体系时，我们首先要查看该体系中的顶层类，了解该体系的基本功能，然后再创建体系中的最子类对象，完成功能的使用。  
③继承的设计原则："高内聚低耦合"  
所谓高内聚低耦合，简单的理解，内聚就是指自己完成某件事情的能力，耦合就是类与类之间的关系。我们在设计继承时候的原则就是：自己能完成的就不要麻烦别人，这样将来别人产生了修改，就对我的影响较小。由此可见：在开发中使用继承其实是在使用一把双刃剑。今天我们还是以继承的好处来使用，因为继承还有很多其他的特性。  
④继承的好处

* 将多个类相同的成员可以放到一个类中，提高了代码的复用性。
* 如果功能的代码需要修改，修改一处即可，提高了代码的维护性。
* 让类与类之间产生了关系，是多态的前提。但这也造成了高耦合，也是其弊端之一。

继承的一个简单示例如下：

1. /\*
2. 继承概述
3. \*/
5. //使用继承前
6. /\*
7. class Student {
8. public void eat() {
9. System.out.println("吃饭");
10. }
12. public void sleep() {
13. System.out.println("睡觉");
14. }
15. }
17. class Teacher {
18. public void eat() {
19. System.out.println("吃饭");
20. }
22. public void sleep() {
23. System.out.println("睡觉");
24. }
25. }
26. \*/
28. //使用继承后，老师和学生都是人的范畴，抽取了老师和学生的共性功能，作为人的功能，让老师和学生去继承。
29. **class** Person {
30. **public** **void** eat() {
31. System.out.println("吃饭");
32. }
34. **public** **void** sleep() {
35. System.out.println("睡觉");
36. }
37. }
39. **class** Student **extends** Person {}
41. **class** Teacher **extends** Person {}
43. **class** ExtendsDemo {
44. **public** **static** **void** main(String[] args) {
45. Student s = **new** Student();
46. s.eat();
47. s.sleep();
48. System.out.println("-------------");
50. Teacher t = **new** Teacher();
51. t.eat();
52. t.sleep();
53. }
54. }

**2、继承注意事项**

①子类只能继承父类所有非私有的成员（成员方法和成员变量）

②子类不能继承父类的构造方法，但是可以通过super关键字去访问父类构造方法。

③不要为了部分功能去继承。

④当类之间出现了”is-a“的关系时，我们才可以去继承。不能因为两个类中有部分代码相同就使用继承，这是不对的。

代码演示如下：

1. /\*
2. 继承的注意事项：
3. class A {
4. public void show1(){}
5. public void show2(){}
6. }
8. class B {
9. public void show2(){}
10. public void show3(){}
11. }
13. //我们发现B类中出现了和A类一样的show2()方法，所以，我们就用继承来体现
14. class B extends A {
15. public void show3(){}
16. }
17. 这样其实不好，因为这样你不但有了show2(),还多了show1()。
18. 有可能show1()不是你想要的。
20. 继承其实体现的是一种关系："is a"。
22. 采用假设法。
23. 如果有两个类A,B。只有他们符合A是B的一种，或者B是A的一种，就可以考虑使用继承。
24. \*/
25. **class** Father {
26. **private** **int** num = 10;
27. **public** **int** num2 = 20;
29. //私有方法，子类不能继承
30. **private** **void** method() {
31. System.out.println(num);
32. System.out.println(num2);
33. }
35. **public** **void** show() {
36. System.out.println(num);
37. System.out.println(num2);
38. }
39. }
41. **class** Son **extends** Father {
42. **public** **void** function() {
43. //num可以在Father中访问private
44. //System.out.println(num); //子类不能继承父类的私有成员变量
45. System.out.println(num2);
46. }
47. }
49. **class** ExtendsDemo3 {
50. **public** **static** **void** main(String[] args) {
51. // 创建对象
52. Son s = **new** Son();
53. //s.method(); //子类不能继承父类的私有成员方法
54. s.show();
55. s.function();
56. }
57. }

3、super关键字

①super关键字和this的用法很像，this代表的是对应的引用，而super代表父类存储空间的标识（父类引用）。

②super关键字的用法：

* 访问成员变量：super.成员变量
* 访问构造方法：super（……）
* 访问成员方法：super.成员方法()。

③继承中构造方法的关系  
为什么子类所有的构造方法默认都会访问父类空参数的构造方法？  
因为子类会继承父类的数据，可能还会使用父类的数据。所以，子类初始化之前，一定要先完成父类数据的初始化。为了实现这个效果，在子类构造的第一条语句上默认有一个super()。

1. /\*
2. 继承中构造方法的关系
3. A:子类中所有的构造方法默认都会访问父类中空参数的构造方法
4. B:为什么呢?
5. 因为子类会继承父类中的数据，可能还会使用父类的数据。
6. 所以，子类初始化之前，一定要先完成父类数据的初始化。
8. 注意：子类每一个构造方法的第一条语句默认都是：super();
9. \*/
10. **class** Father {
11. **int** age;
13. **public** Father() {
14. System.out.println("Father的无参构造方法");
15. }
17. **public** Father(String name) {
18. System.out.println("Father的带参构造方法");
19. }
20. }
22. **class** Son **extends** Father {
23. **public** Son() {
24. //super();
25. System.out.println("Son的无参构造方法");
26. }
28. **public** Son(String name) {
29. //super();
30. System.out.println("Son的带参构造方法");
31. }
32. }
34. **class** ExtendsDemo6 {
35. **public** **static** **void** main(String[] args) {
36. //创建对象
37. Son s = **new** Son();
38. System.out.println("------------");
39. Son s2 = **new** Son("林青霞");
40. }
41. }

④继承中成员方法的关系  
a、当子父类中方法声明不一样的时候，通过子类对象去访问方法，这个很容易访问。  
b、当子父类方法声明一样的时候，首先是通过子类对象去访问方法，先查找子类中有没有该方法，如果有该方法，就使用。如果子类中没有该方法，则去父类中查找有没有该方法，如果父类中有该方法，则使用。如果字父类中都没有该方法，则报错。  
如下代码所示：

1. /\*
2. 继承中成员方法的关系:
3. \*/
4. //父类
5. **class** Father {
6. **public** **void** show() {
7. System.out.println("show Father");
8. }
9. }
10. //子类
11. **class** Son **extends** Father {
12. **public** **void** method() {
13. System.out.println("method Son");
14. }
16. **public** **void** show() {
17. System.out.println("show Son");
18. }
19. }
21. **class** ExtendsDemo8 {
22. **public** **static** **void** main(String[] args) {
23. //创建对象
24. Son s = **new** Son();
25. s.show();//找子类。
26. s.method();//method son。先找子类中，子类有则调用。
27. //s.fucntion(); //找不到符号，子父类中都没有该方法。
28. }
29. }

⑤方法重载（overroad）和方法覆盖（override）  
a、什么是方法重载？（同一个类中）方法重载是指在同一个类中，出现方法名相同，参数列表不同的情况。  
b、什么是方法覆盖？（子父类中）方法覆盖是指在子类中，出现和父类一模一样的方法声明的时候，会运行子类的函数，这种现象称为覆盖操作。  
方法覆盖会发生在有继承关系的父类和子类之间，而且是在子类类型中，子类继承到父类的方法之后，觉得方法实现已经不足以满足新一代的要求了，于是就给出了新的方法实现。  
覆盖注意事项：

* 子类方法覆盖父类方法时，子类权限必须大于等于父类中的权限。
* 静态只能覆盖静态或者被静态覆盖。

c、如何判断方法是不是重载呢？

* 方法名必须相同
* 返回值类型可能不同
* 参数列表必须不同：参数类型不同，参数个数不同，参数顺序不同。

**3、阻止继承：final类和方法**

有时候，可能希望阻止人们利用某个类定义子类。不允许扩展的类被称为final类。如果在定义类的时候使用了final修饰符就表明这个类是final类。

①final关键字是最终的意思，可以修饰类，成员变量，成员方法。

* final关键字修饰的类不可以被继承。
* final修饰的方法不可以被覆盖。
* final修饰的变量是一个常量，只能被覆盖。

②类中的方法也可以被声明为final。如果这样做，子类就不能覆盖这个方法（final类中的所有方法自动称为final方法。）将方法或类声明为final主要是鉴于以下原因：

* 确保它们不会在子类中改变语义。例如：Calendar类中的getTime个setTime方法都声明为final。这表明Calendar类的设计者负责实现Date类与日历状态之间的转换，而不允许子类处理这些问题。同样的，String类也是final类，这意味着不允许任何人定义String的子类。换而言之，如果有一个String的引用，它引用的一定是一个String对象，而不可能是其它对象。

③为什么要用final修饰变量？  
其实在程序中如果一个数据是固定的，那么直接使用这个数据就可以了。但是这样阅读性很差，所以就给该数据起个名称，而且这个变量名称的值不能再变化，所以加上final固定。  
写法规范：常量所有字符都大写，如果多个单词，中间用"\_"连接。  
代码示范如下：

1. /\*
2. final可以修饰类，方法，变量
3. \*/
5. //final class Fu //无法从最终Fu进行继承
7. **class** Fu {
8. **public** **int** num = 10;
9. **public** **final** **int** NUM\_2 = 20;
11. // Zi中的show()无法覆盖Fu中的show()
12. /\*  public final void show() {
13. System.out.println(num);
14. }
15. \*/
16. }
18. **class** Zi **extends** Fu {
19. // Zi中的show()无法覆盖Fu中的show()
20. **public** **void** show() {
21. num = 100;
22. System.out.println(num);
24. //无法为最终变量num2分配值
25. //NUM\_2 = 200;
26. System.out.println(NUM\_2);
27. }
28. }
30. **class** FinalDemo {
31. **public** **static** **void** main(String[] args) {
32. Zi z = **new** Zi();//100
33. z.show();//20
34. }
35. }

④final关键字面试题  
final修饰局部变量

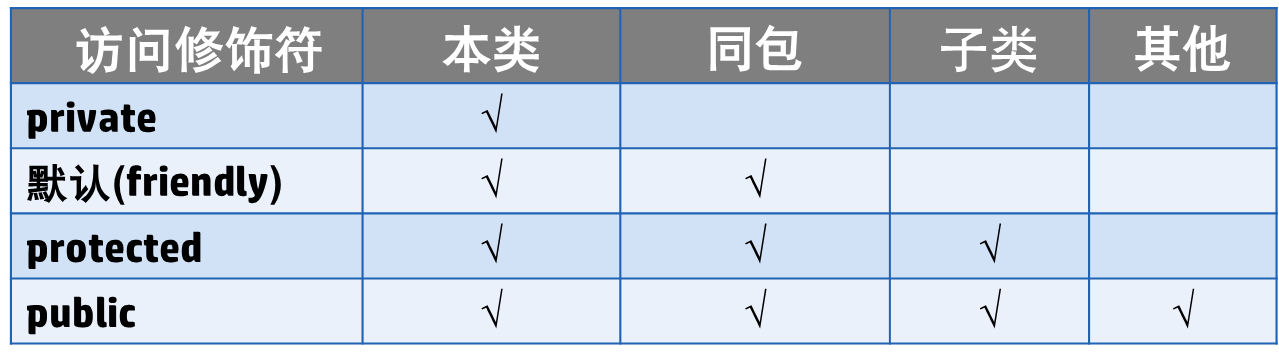
* 在方法内部，该变量不可改变。
* 在方法声明上，如果是基本类型，则值不能改变；如果是引用类型，则是地址值不能改变。

代码示例如下：

1. /\*
2. 面试题：final修饰局部变量的问题
3. 基本类型：基本类型的值不能发生改变。
4. 引用类型：引用类型的地址值不能发生改变，但是，该对象的堆内存的值是可以改变的。
5. \*/
6. **class** Student {
7. **int** age = 10;
8. }
10. **class** FinalTest {
11. **public** **static** **void** main(String[] args) {
12. //局部变量是基本数据类型
13. **int** x = 10;
14. x = 100;
15. System.out.println(x);//100
16. **final** **int** y = 10;
17. //无法为最终变量y分配值（基本类型）
18. //y = 100;
19. System.out.println(y);//10
20. System.out.println("--------------");
22. //局部变量是引用数据类型
23. Student s = **new** Student();
24. System.out.println(s.age);//10
25. s.age = 100;
26. System.out.println(s.age);//100
27. System.out.println("--------------");
28. //引用数据类型的地址值不可改变，但堆内存的值是可变的！
29. **final** Student ss = **new** Student();
30. System.out.println(ss.age);//10
31. ss.age = 100;
32. System.out.println(ss.age);//100
34. //重新分配内存空间
35. //错误：无法为最终变量ss分配值（引用类型）
36. //ss = new Student();
37. }
38. }

  final修饰变量的初始化时机：在构造对象前完毕即可。

**4、访问控制—public、protected、default、private**



* public：公有的

被声明为public的类、方法、构造方法和接口能够被任何其他类访问。

* protected：受保护的

被声明为protected的变量、方法和构造方法能被同一个包中的任何其他类访问，也能够被不同包中的子类访问。

* private：私有的

私有访问修饰符是最严格的访问级别，所以被声明为private的方法、变量和构造方法只能被所属类访问，并且类和接口不能声明为private。

* 默认的：不使用任何关键字

不使用任何修饰符声明的属性和方法，对同一个包内的类是可见的。接口里的变量都隐式声明为public static final，而接口里的方法默认情况下访问权限为public。

**七、面向对象——抽象类**

**1、什么是抽象？**

定义：抽象类就从多个事物中，将共性的、本质的东西提取出来。

例如：老师和学生，都是人，都具有如姓名、性别、年龄等属性。将其抽取出来，放置于继承关系较高层次的通用超类人中。

**2、抽象类**

①什么是抽象类？

Java中可以定义没有方法体的方法，该方法的具体实现由子类完成，该方法称为抽象方法，包含抽象方法的类就是抽象类。

抽象类的由来：多个对象都具备相同的功能，但是功能具体内容有所不同，那么在抽取过程中，只抽取了功能定义，并未抽取功能主体，那么只有功能声明，没有功能主题的方法称为抽象方法。

②抽象类的特点

* 方法只有声明没有实现时，该方法就是抽象方法，需要abstract修饰，抽象方法必须定义在抽象类中，该类也必须被abstract修饰。
* 抽象类不能被实例化，因为调用抽象方法没有意义。抽象类按照多态的方式，由具体的子类实例化。
* 抽象类必须有其子类覆盖了所有的抽象方法后，该子类才可以初始化，否则这个子类还是抽象类。
* 抽象类不一定有抽象方法，有抽象方法的类一定是抽象类。

如下代码所示，演示一下抽象类的使用：

1. /\*
2. 抽象类的概述：
3. 动物不应该定义为具体的东西，而且动物中的吃，睡等也不应该是具体的。
4. 我们把一个不是具体的功能称为抽象的功能，而一个类中如果有抽象的功能，该类必须是抽象类。
6. 抽象类的实例化其实是靠具体的子类实现的。是多态的方式。
7. People p = new Student();
8. \*/
10. //abstract class People //抽象类的声明格式
11. **abstract** **class** People {
12. //抽象方法
13. //public abstract void eat(){} //空方法体,这个会报错。抽象方法不能有主体
14. **public** **abstract** **void** eat();
16. **public** People(){}
17. }
19. //子类是抽象类
20. **abstract** **class** Teacher **extends** People {}//没有实现父类的抽象方法。还是抽象类。
22. //子类是具体类，重写抽象方法
23. **class** Student **extends** People {
24. **public** **void** eat() {
25. System.out.println("学生吃饭");
26. }
27. }
29. **class** AbstractDemo {
30. **public** **static** **void** main(String[] args) {
31. //创建对象
32. //people是抽象的; 无法实例化
33. //People p = new People();
34. //通过多态的方式
35. People p = **new** Student();
36. p.eat();
37. }
38. }

**3、抽象类的几个问题**

①抽象类中有构造方法吗？  
有，用于给子类对象进行初始化。  
②抽象类可以定义非抽象方法吗？  
可以。但是很少见，目的是不让该类创建对象，AWT的适配器对象就是这种类，通常这个类中的方法有方法体，但是却没有内容。  
③抽象的关键字不可以和哪些关键字共存？  
private不行，static不行，final不行。  
原因在于：

* final：被final修饰的类不能有子类。而abstract修饰的类一定是父类。
* private：抽象类中的私有的抽象方法，不能被子类所知，也就无法复写，而抽象方法的出现就是要被复写。
* static：如果static修饰抽象方法，那么对象都不用new了。直接类名.方法调用就行了。

如下列代码所示：

1. /\*
2. 一个类如果没有抽象方法，可不可以定义为抽象类?如果可以，有什么意义?
3. A:可以。
4. B:不让创建对象。
6. abstract不能和哪些关键字共存?
7. private 冲突
8. final   冲突
9. static  无意义
10. \*/
11. **abstract** **class** Fu {
12. //public abstract void show();
13. //非法的修饰符组合: abstract和private
14. //private abstract void show();
16. //非法的修饰符组合
17. //final abstract void show();
19. //非法的修饰符组合
20. **static** **abstract** **void** show();
22. **public** **static** **void** method() {
23. System.out.println("method");
24. }
25. }
27. **class** Zi **extends** Fu {
28. **public** **void** show() {}
29. }
31. **class** AbstractDemo {
32. **public** **static** **void** main(String[] args) {
33. Fu.method();
34. }
35. }

**4、抽象类和一般类的异同点**

①相同点：抽象类和一般类都是用来描述事物的，都是内部定义了成员。  
②不同点：

* 一般类有足够的信息描述事物；抽象类中描述的信息可能不足。
* 一般类不能定义抽象方法，只能定义非抽象方法；抽象类可以定义抽象方法，也可以定义非抽象方法。
* 一般类可以实例化；抽象类不可以实例化。

**八、接口**

**1、接口的概念**

接口可以被看成是一个特殊的抽象类。当一个抽象类中的方法都是抽象的时候，这是可以将抽象类用另一种形式来定义和表示，就是接口（interface）。定义接口使用的关键字不是class，而是interface，接口当中的常见成员，而且这些成员都有固定的修饰符：全局常量，抽象方法。

**2、接口的特点**

①接口用关键字interface表示。格式：interface接口名{}。

②类实现接口用implements表示。格式：class类名implements接口名{}。

③接口是一种特殊的抽象类，不能实例化。尤其不能用new运算符实例化一个接口。按照多态的方式，由具体的子类实例化。其实这也是多态的一种，接口多态。

④接口的子类要么是抽象类，要么重写接口中的所有方法。

⑤不能构造接口的对象，但却能声明接口的变量。如：Comparable x；

⑥接口变量必须引用实现了接口的类对象。如：x = new XXXX（）；

⑦接口中不能包含实例域或静态方法，但却可以包含常量。

**3、接口成员的特点**

①成员变量：只能是常量。默认修饰符是public static final。

②构造方法：没有，因为接口主要是扩展功能的，而没有具体存在

③成员方法：只能是抽象方法，默认修饰符public abstract。

**4、接口必须掌握的知识**

* 接口当中的成员都是公共的，是自动属于public的。因此，在接口方法声明中，可以不写public。
* 类与类之间的是继承关系（extends）；类与接口之间是实现关系（implments）。
* 接口不可以实例化，只能由实现类接口的子类并覆盖了接口中所有的抽象方法后，该子类才可以实例化，否则，这个子类就是一个抽象类。
* 接口实现的步骤：a、将类声明为实现给定的接口。b、对接口中的所有方法进行定义。
* 在java中不直接支持多继承，因为会出现调用的不确定性，所以java将多继承机制进行了改良，在java中可以多实现，即一个类可以实现多个接口。
* 一个类在继承另一个类的同时，还能实现多个接口。接口的出现避免了单继承的局限性。
* 接口与接口之间可以继承，而且接口可以多继承。

**5、抽象类和接口的异同点**

①相同点：都是不断向上抽取而来的。

②不同点：

* 抽象类需要被继承，而且只能单继承；接口需要被实现，而且能多实现。
* 抽象类中可以定义抽象方法和非抽象方法，子类继承后，可以直接使用非抽象方法；接口中只能定义抽象方法，必须由接口去实现。
* 抽象类的继承是is-a关系，在定义该体系内的基本共性内容；接口的实现是has-a关系，在定义体系的额外功能。

接口应用案例代码演示：

1. /\*
2. 老师和学生案例,加入抽烟的额外功能
4. 分析：从具体到抽象
5. 老师：姓名，年龄，吃饭，睡觉
6. 学生：姓名，年龄，吃饭，睡觉
8. 由于有共性功能，我们提取出一个父类，人类。
10. 人类：
11. 姓名，年龄
12. 吃饭();
13. 睡觉(){}
15. 抽烟的额外功能不是人或者老师，或者学生一开始就应该具备的，所以，我们把它定义为接口
17. 抽烟接口。
19. 部分老师抽烟：实现抽烟接口
20. 部分学生抽烟：实现抽烟接口
22. 实现：从抽象到具体
24. 使用：具体
25. \*/
26. //定义抽烟接口
27. **interface** Smoking {
28. //抽烟的抽象方法
29. **public** **abstract** **void** smoke();
30. }
32. //定义抽象人类
33. **abstract** **class** Person {
34. //姓名
35. **private** String name;
36. //年龄
37. **private** **int** age;
39. **public** Person() {}
40. //构造器初始化
41. **public** Person(String name,**int** age) {
42. **this**.name = name;
43. **this**.age = age;
44. }
46. **public** String getName() {
47. **return** name;
48. }
50. **public** **void** setName(String name) {
51. **this**.name = name;
52. }
54. **public** **int** getAge() {
55. **return** age;
56. }
58. **public** **void** setAge(**int** age) {
59. **this**.age = age;
60. }
62. //吃饭方法;
63. **public** **abstract** **void** eat();
65. //睡觉睡觉方法{}
66. **public** **void** sleep() {
67. System.out.println("睡觉觉了");
68. }
69. }
71. //具体老师类
72. **class** Teacher **extends** Person {
73. **public** Teacher() {}
75. **public** Teacher(String name,**int** age) {
76. **super**(name,age);
77. }
79. **public** **void** eat() {
80. System.out.println("吃大白菜");
81. }
82. }
84. //具体学生类
85. **class** Student **extends** Person {
86. **public** Student() {}
88. **public** Student(String name,**int** age) {
89. **super**(name,age);
90. }
92. **public** **void** eat() {
93. System.out.println("吃红烧肉");
94. }
95. }
97. //抽烟的老师
98. **class** SmokingTeacher **extends** Teacher **implements** Smoking {
99. **public** SmokingTeacher() {}
101. **public** SmokingTeacher(String name,**int** age) {
102. **super**(name,age);
103. }
104. //抽烟方法
105. **public** **void** smoke() {
106. System.out.println("抽烟的老师");
107. }
108. }
110. //抽烟的学生
111. **class** SmokingStudent **extends** Student **implements** Smoking {
112. **public** SmokingStudent() {}
114. **public** SmokingStudent(String name,**int** age) {
115. **super**(name,age);
116. }
117. //实现抽烟方法
118. **public** **void** smoke() {
119. System.out.println("抽烟的学生");
120. }
121. }
123. **class** InterfaceTest2 {
124. **public** **static** **void** main(String[] args) {
125. //测试学生
126. SmokingStudent ss = **new** SmokingStudent();
127. ss.setName("黄祥");
128. ss.setAge(23);
129. System.out.println(ss.getName()+"---"+ss.getAge());
130. ss.eat();
131. ss.sleep();
132. ss.smoke();
133. System.out.println("-------------------");
135. //测试老师
136. SmokingTeacher st = **new** SmokingTeacher();
137. st.setName("刘老师");
138. st.setAge(45);
139. System.out.println(st.getName()+"---"+st.getAge());
140. st.eat();
141. st.sleep();
142. st.smoke();
143. }
144. }

|  |  |  |
| --- | --- | --- |
| 一般类  (class) | 抽象类  (abstract class) | 接口  (interface) |
| 静态属性  静态方法  一般属性  一般方法  构造方法 | 静态属性  静态方法  一般属性  一般方法  构造方法  抽象方法 | 静态属性（静态常量）  抽象方法 |
| 可以实例化 | 不能实例化 | 不能实例化 |
| 可以继承抽象类，实现多个接口 | 被多个类继承，其子类可以是抽象类也可以是一般类，抽象类的继承是is-a关系 | 可以被多个类实现，也可以被多个抽象类实现，接口的实现是like-a关系 |

**九、多态性**

   多态性是面向对象编程中常用的特性，所以JAVA作为面向对象的主流语言自然也有多态性，它的多态性其实和C++的概念差不多，只是实现形式和表现形式不一样。在C++中可能还会提到多态的划分，但是在JAVA中可能很多人都不会听到编译期多态和运行期多态这种划分，一般我们说到多态都是指运行期多态，因为这才是面向对象思想的真正体现之处，即OOP(面向对象)的多态性的体现，所以JAVA中我们不再讨论编译期多态这个问题，只重点讨论运行期多态，下面简称运行期多态为JAVA中的多态性。

**1. 编译期多态（静态多态）**

学习JAVA的话就不要深究这个问题了，直接进入运行期多态。（重载属于编译期多态）

**2. 运行期多态（动态多态）**

    运行期多态主要是指在程序运行的时候，动态绑定所调用的函数，动态地找到了调用函数的入口地址，从而确定到底调用哪个函数。

（1）前提

    A. 要有继承关系。  
    B. 要有方法重写。其实没有也是可以的，但是如果没有这个就没有意义。  
    C. 要有父类引用指向子类对象。

（2）多态中的成员访问特点  
    A. 成员变量：编译看左边，运行看左边。  
    B. 构造方法：创建子类对象的时候，访问父类的构造方法，对父类的数据进行初始化。  
    C. 成员方法：编译看左边，运行看右边。由于成员方法存在方法重写，所以它运行看右边。

    D. 静态方法：编译看左边，运行看左边。静态和类相关，算不上重写，所以，访问还是左边的。  
（3）多态性的例子

1. // 例1：多态性
2. **class** Fu {
3. **public** **int** num = 100;
5. **public** **void** show() {
6. System.out.println("show Fu");
7. }
9. **public** **static** **void** function() {
10. System.out.println("function Fu");
11. }
12. }
14. **class** Zi **extends** Fu {
15. **public** **int** num = 1000;
16. **public** **int** num2 = 200;
18. **public** **void** show() {
19. System.out.println("show Zi");
20. }
22. **public** **void** method() {
23. System.out.println("method zi");
24. }
26. **public** **static** **void** function() {
27. System.out.println("function Zi");
28. }
29. }
31. **class** DuoTaiDemo {
32. **public** **static** **void** main(String[] args) {
33. //要有父类引用指向子类对象。
34. //父 f =  new 子();
35. Fu f = **new** Zi();
36. System.out.println(f.num);
37. //找不到符号
38. //System.out.println(f.num2);
40. f.show();
41. //找不到符号
42. //f.method();
43. f.function();
44. }
45. }
46. 运行结果：
48. 100
49. show Zi
50. function Fu

例1是一个经典的多态性例子，父类中和子类中有一些同样名称的成员方法和成员变量，那么这里的访问原则就遵从（2）中的原则，从运行结果可以很清楚地看到这一点。我们这里关键来说说多态性到底体现在哪里。从main方法中可以看到，我们定义了一个父类的引用然后指向了子类的对象，我们关键来研究show()方法的调用，因为这才是多态性的关键之处。这里其实存在一个向上转型，即将子类对象向上转型到了一个父类引用，然后我们利用这个父类引用调用show()方法的时候，由于在子类中重写了show()方法，并且该方法是一个普通的成员方法，并不是什么静态方法，所以，遵从“编译看左边，运行看右边”的原则，这个原则正是多态性的体现。这是因为在运行期间才实现的动态绑定，将父类的引用绑定到了子类的对象上，从而用父类的引用去调用父类和子类都有的方法时，实际上调用的是子类的方法，这就是多态性。虽然是父类的引用，但是在运行期间却绑定到了子类对象上。

    例2很形象地说明了向上转型和向下转型的问题，这里的向上转型就是多态性的一个体现。注意，这个例子只是用来让大家理解，并不能直接运行，大家可以修改成合理的字母表示，然后运行。这个例子是摘自风清扬的一个经典的例子，十分形象合理。

1. /\*
2. 例3  多态的好处：
3. A:提高了代码的维护性(继承保证)
4. B:提高了代码的扩展性(由多态保证)
6. 猫狗案例代码
7. \*/
8. **class** Animal {
9. **public** **void** eat(){
10. System.out.println("eat");
11. }
13. **public** **void** sleep(){
14. System.out.println("sleep");
15. }
16. }
18. **class** Dog **extends** Animal {
19. **public** **void** eat(){
20. System.out.println("狗吃肉");
21. }
23. **public** **void** sleep(){
24. System.out.println("狗站着睡觉");
25. }
26. }
28. **class** Cat **extends** Animal {
29. **public** **void** eat() {
30. System.out.println("猫吃鱼");
31. }
33. **public** **void** sleep() {
34. System.out.println("猫趴着睡觉");
35. }
36. }
38. **class** Pig **extends** Animal {
39. **public** **void** eat() {
40. System.out.println("猪吃白菜");
41. }
43. **public** **void** sleep() {
44. System.out.println("猪侧着睡");
45. }
46. }
48. //针对动物操作的工具类
49. **class** AnimalTool {
50. **private** AnimalTool(){}
52. /\*
53. //调用猫的功能
54. public static void useCat(Cat c) {
55. c.eat();
56. c.sleep();
57. }
59. //调用狗的功能
60. public static void useDog(Dog d) {
61. d.eat();
62. d.sleep();
63. }
65. //调用猪的功能
66. public static void usePig(Pig p) {
67. p.eat();
68. p.sleep();
69. }
70. \*/
71. **public** **static** **void** useAnimal(Animal a) {
72. a.eat();
73. a.sleep();
74. }
76. }
78. **class** DuoTaiDemo2 {
79. **public** **static** **void** main(String[] args) {
80. //我喜欢猫，就养了一只
81. Cat c = **new** Cat();
82. c.eat();
83. c.sleep();
85. //我很喜欢猫，所以，又养了一只
86. Cat c2 = **new** Cat();
87. c2.eat();
88. c2.sleep();
90. //我特别喜欢猫，又养了一只
91. Cat c3 = **new** Cat();
92. c3.eat();
93. c3.sleep();
94. //...
95. System.out.println("--------------");
96. //问题来了,我养了很多只猫，每次创建对象是可以接受的
97. //但是呢?调用方法，你不觉得很相似吗?仅仅是对象名不一样。
98. //我们准备用方法改进
99. //调用方式改进版本
100. //useCat(c);
101. //useCat(c2);
102. //useCat(c3);
104. //AnimalTool.useCat(c);
105. //AnimalTool.useCat(c2);
106. //AnimalTool.useCat(c3);
108. AnimalTool.useAnimal(c);
109. AnimalTool.useAnimal(c2);
110. AnimalTool.useAnimal(c3);
111. System.out.println("--------------");
113. //我喜欢狗
114. Dog d = **new** Dog();
115. Dog d2 = **new** Dog();
116. Dog d3 = **new** Dog();
117. //AnimalTool.useDog(d);
118. //AnimalTool.useDog(d2);
119. //AnimalTool.useDog(d3);
120. AnimalTool.useAnimal(d);
121. AnimalTool.useAnimal(d2);
122. AnimalTool.useAnimal(d3);
123. System.out.println("--------------");
125. //我喜欢宠物猪
126. //定义一个猪类，它要继承自动物，提供两个方法，并且还得在工具类中添加该类方法调用
127. Pig p = **new** Pig();
128. Pig p2 = **new** Pig();
129. Pig p3 = **new** Pig();
130. //AnimalTool.usePig(p);
131. //AnimalTool.usePig(p2);
132. //AnimalTool.usePig(p3);
133. AnimalTool.useAnimal(p);
134. AnimalTool.useAnimal(p2);
135. AnimalTool.useAnimal(p3);
136. System.out.println("--------------");
138. //我喜欢宠物狼，老虎，豹子...
139. //定义对应的类，继承自动物，提供对应的方法重写，并在工具类添加方法调用
140. //前面几个必须写，我是没有意见的
141. //但是，工具类每次都改，麻烦不
142. //我就想，你能不能不改了
143. //太简单：把所有的动物都写上。问题是名字是什么呢?到底哪些需要被加入呢?
144. //改用另一种解决方案。
146. }
148. /\*
149. //调用猫的功能
150. public static void useCat(Cat c) {
151. c.eat();
152. c.sleep();
153. }
155. //调用狗的功能
156. public static void useDog(Dog d) {
157. d.eat();
158. d.sleep();
159. }
160. \*/
161. }

**3. 总结**

总之，在JAVA中大家就不要去过多纠结编译期多态和运行期多态，只要掌握好常用的多态性即运行期多态即可。

**十、内部类**

一般来说，有4中内部类：常规内部类、静态内部类、局部内部类、匿名内部类。

**1、常规内部类**

  (1) 在外部类的作用范围内可以任意创建内部类对象，即使内部类是私有的(私有内部类)。即内部类对包围它的外部类可见。

1. //代码1：内部类对外部类可见
2. **class** Outer{
3. //创建私有内部类对象
4. **public** Inner in=**new** Inner();
5. //私有内部类
6. **private** **class** Inner{
7. ...
8. }
9. }

      (2) 在内部类中可以访问其外部类的所有域，即使是私有域。即外部类对内部类可见。

1. //代码2：外部类对内部类可见
2. **class** Outer{
3. //外部类私有数据域
4. **private** **int** data=0;
5. //内部类
6. **class** Inner{
7. **void** print(){
8. //内部类访问外部私有数据域
9. System.out.println(data);
10. }
11. }
12. }

**2、静态内部类**

内部类也有静态的区别，这就是静态内部类，我们来看看代码：

2. //代码3：静态内部类对外部变量的引用
3. **public** **class** Outer{
4. **private** **static** **int** i=0;
5. //创建静态内部类对象
6. **public** Inner in=**new** Inner();
7. //静态
8. **private** **static** **class** Inner{
9. **public** **void** print(){
10. System.out.println(i);   //如果i不是静态变量，这里将无法通过编译。
11. }
12. }
14. }

静态内部类和私有内部类最大的区别在于，静态内部类中无法引用到其外围类的非静态成员。**静态内部类只能访问其外围类的静态成员，除此之外与非静态内部类没有任何区别。**

**3、局部内部类**

 局部内部类也叫方法内部类，其也有两个特点

      (1)  方法中的内部类没有访问修饰符， 即方法内部类对包围它的方法之外的任何东西都不可见。

      (2)  方法内部类只能够访问该方法中的局部变量，所以也叫局部内部类。而且这些局部变量一定要是final修饰的常量。

1. **class** Outter{
2. **public** **void** outMethod(){
3. **final** **int** beep=0;
4. **class** Inner{
5. //使用beep
6. }
7. Inner in=**new** Inner();
8. }
9. }

我们可以这样解释Inner类中的这个备份常量域，首先当JVM运行到需要创建Inner对象之后，Outter类已经全部运行完毕，这是垃圾回收机制很有可能释放掉局部变量beep。那么Inner类到哪去找beep变量呢？

**编译器又出来帮我们解决了这个问题，他在Inner类中创建了一个beep的备份**，也就是说即使Ouuter中的beep被回收了，Inner中还有一个备份存在，自然就不怕找不到了。

但是问题又来了。如果Outter中的beep不停的在变化那。那岂不是也要让备份的beep变量无时无刻的变化。**为了保持局部变量与局部内部类中备份域保持一致。**编译器不得不规定死这些局部域必须是常量，一旦赋值不能再发生变化了。所以为什么局部内部类应用外部方法的域必须是常量域的原因所在了。

**4、匿名内部类**

匿名内部类也就是没有名字的内部类

正因为没有名字，所以匿名内部类只能使用一次，它通常用来简化代码编写

但使用匿名内部类还有个前提条件：必须继承一个父类或实现一个接口

实例1:不使用匿名内部类来实现抽象方法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | abstract class Person {      public abstract void eat();  }    class Child extends Person {      public void eat() {          System.out.println("eat something");      }  }    public class Demo {      public static void main(String[] args) {          Person p = new Child();          p.eat();      }  } |

**运行结果：**eat something

可以看到，我们用Child继承了Person类，然后实现了Child的一个实例，将其向上转型为Person类的引用

但是，如果此处的Child类只使用一次，那么将其编写为独立的一个类岂不是很麻烦？

这个时候就引入了匿名内部类

实例2：匿名内部类的基本实现

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | abstract class Person {      public abstract void eat();  }    public class Demo {      public static void main(String[] args) {          Person p = new Person() {              public void eat() {                  System.out.println("eat something");              }          };          p.eat();      }  } |

**运行结果：**eat something

可以看到，我们直接将抽象类Person中的方法在大括号中实现了

这样便可以省略一个类的书写

并且，匿名内部类还能用于接口上

实例3：在接口上使用匿名内部类

|  |
| --- |
| interface Person {      public void eat();  }    public class Demo {      public static void main(String[] args) {          Person p = new Person() {              public void eat() {                  System.out.println("eat something");              }          };          p.eat();      }  } |

**运行结果：**eat something

由上面的例子可以看出，只要一个类是抽象的或是一个接口，那么其子类中的方法都可以使用匿名内部类来实现

最常用的情况就是在多线程的实现上，因为要实现多线程必须继承Thread类或是继承Runnable接口

实例4：Thread类的匿名内部类实现

|  |
| --- |
| public class Demo {      public static void main(String[] args) {          Thread t = new Thread() {              public void run() {                  for (int i = 1; i <= 5; i++) {                      System.out.print(i + " ");                  }              }          };          t.start();      }  } |

**运行结果：**1 2 3 4 5

实例5：Runnable接口的匿名内部类实现

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class Demo {     public static void main(String[] args) {       Runnable r = new Runnable() {           public void run() {             for (int i = 1; i <= 5; i++) {                System.out.print(i + " ");              }           }          };          Thread t = new Thread(r);          t.start();      }  } |

**运行结果：**1 2 3 4 5