**面向对象**

## 一、面向对象的思想

**1 面向对象和面向过程的思想有着本质上的区别,**作为面向对象的思维来说，当你拿到一个问题时，你分析这个问题不再是第一步先做什么，第二步再做什么，这是面向过程的思维，你应该分析这个问题里面有哪些类和对象，这是第一点，然后再分析这些类和对象应该具有哪些属性和方法。这是第二点。最后分析类和类之间具体有什么关系，这是第三点。

**2 面向对象有一个非常重要的设计思维：合适的方法应该出现在合适的类里面**

**3 简单理解面向对象**

就是在程序里面首先分解出来的应该是注意不再是一步一步的过程了，而是首先考虑在这个问题域里面或者程序里面应该具有有哪些对象，所以从现在开始考虑任何问题脑子里不要再想着我实现这件事我第一步应该干什么，第二步应该干什么，如果这样想，那就是面向过程的思维了。面向对象的思维是，当我碰到这个问题域的时候，碰到这个程序的时候，我首先应该把这个问题里有哪些对象，对象与对象之间有什么关系抽象出来。

**4 面向对象的设计思想**

（1）面向对象的基本思想是，从现实世界中客观存在的事物出发来构造软件系统，并在系统的构造中尽可能运用人类的自然思维方式。

（2）面向对象更加强调运用人类在日常生活的逻辑思维中经常采用的思想方法与原则，如抽象、分类，继承、聚合、多态等。

（3）人在思考的时候，首先眼睛里看到的是一个一个的对象。

**5 什么是类?什么是对象?**

简单来说,类是抽象出某一类事物的行为和特征,并进行的封装,特征称之为属性,行为称之为方法;

对象就是这一类的具体实例,对象拥有这个类的所有属性和方法

（1）举个例子

所有的人是一个类,人所具有的属性:身高和体重...,人具有的方法:吃饭,睡觉..

具体到某一个人是一个对象,比如小明就是一个对象,小明有身高,体重,可以吃饭,可以睡觉..

（2）java编程思想中是这么描述对象的:

**对象具有状态,行为,和标识.这意味着每一个对象都可以拥有内部数据(它们给出了该对象的状态)和方法(它们产生行为),并且每一个对象都可以与其他对象区分开来,具体来说,就是每一个对象在内存中都有一个唯一的地址.**

**6 什么是面向对象?为什么要面向对象?**

（1）官方话

面向对象（Object Oriented）是一种新兴的程序设计方法，或者是一种新的程序设计规范(paradigm)，其基本思想是使用对象、类、继承、封装、多态等基本概念来进行程序设计。

从现实世界中客观存在的事物（即对象）出发来构造软件系统，并且在系统构造中尽可能运用人类的自然思维方式。面向对象是为了解决系统的可维护性，可扩展性，可重用性....

（2）轻松理解

首先,面向对象是一种设计程序的方式,是一种设计思想.

其次,面向对象节省代码,思路清晰,便于扩展.你所创建出的对象就是给你提供服务的.

## 二、面向对象的三大特征

**1 封装，封装是实现面向对象程序设计的第一步，封装就是将属性和方法放在一个类中。**对外提供服务时尽可能的隐藏对象内部细节，只保留有限的接口和方法;

**（1）封装的意义：**

　　封装的意义在于保护或者防止代码（数据）被我们无意中破坏。在面向对象程序设计中属性被看作是一个中心的元素并且和使用它的方法结合的很密切，从而保护　　它不被其它的方法意外的修改。

可以通过对类的成员设置一定的访问权限，实现类中成员的信息隐藏。

**（2）访问权限控制符**

**private：**类中限定为private的成员，只能被这个类本身访问。如果一个类的构造方法声明为private,则其它类不能生成该类的一个实例。

**default：**类中不加任何访问权限限定的成员属于缺省的（default）访问状态，可以被这个类本身和同一个包中的类所访问。

**protected**：类中限定为protected的成员，可以被这个类本身、它的子类（包括同一个包中以及不同包中的子类）和同一个包中的所有其他的类访问。

**public：**类中限定为public的成员，可以被所有的类访问。

**2 继承**

（1）Java的类可以分为三类

类：使用class定义，没有抽象方法

抽象类：使用abstract class定义，可以有也可以没有抽象方法

接口：使用inerface定义，只能有抽象方法,(也可以没有),接口里的变量默认使用public static final修饰

（2）在这三个类型之间存在如下关系

类可以extends：类、抽象类（必须实现所有抽象方法），但只能extends一个，可以implements多个接口（必须实现所有接口方法）--->单继承多实现

抽象类可以extends：类，抽象类（可全部、部分、或者完全不实现父类抽象方法），可以implements多个接口（可全部、部分、或者完全不实现接口方法）

一个接口可以继承多个接口

（3）继承以后子类可以得到什么：

子类直接拥有父类非private的属性和方法

子类可以添加自己的属性和方法，即对父类进行扩展

子类可以重写父类的方法

（4）关于构造方法：

构造方法不能被继承，子类可以通过super()显式调用父类的构造方法

创建子类对象时，编译器会自动调用父类的无参构造方法

如果父类没有定义无参构造方法，子类必须在构造方法的第一行代码使用super()显式调用

类默认拥有无参构造方法，如果定义了其他有参构造方法，则无参构造方法失效，所以父类没有定义无参构造方法，不是指父类没有写无参构造方法。

（5）Java中通过super来实现对父类成员的访问，super用来引用当前对象的父类。super 的使用有三种情况：

访问父类被隐藏的成员变量

调用父类中被重写的方法

调用父类的构造函数

（6）继承的意义

代码重用是一点，最重要的还是向上转型，即父类的引用变量可以指向子类对象，这是Java面向对象最重要特性多态的基础

**3 多态**

（1）什么是相同的方法：

一个方法可以由：修饰符+返回值+方法名+参数+throw的异常+方法体6部分组成

其中只有方法名和参数是唯一性标识，意即只要方法名和参数相同那他们就是相同的方法

所谓参数相同，是指参数的个数，类型，顺序一致，其中任何一项不同都是不同的方法

（2）什么是重载：

重载是指一个类里面（包括父类的方法）存在方法名相同，但是参数不一样的方法，参数不同可以是不同的参数个数、类型或顺序

如果仅仅是修饰符、返回值、throw的异常不同，那这是2个相同的方法，编译都通不过，更不要说重载了

（3）什么是重写:

子类中存在和父类相同的方法即为重写(有特殊情况,请注意下面重写的规则)，何谓相同方法请牢记前面的描述，方法名和参数相同，包括参数个数、类型、顺序

（4）重写的规则：

子类不能覆盖父类private的方法，private对子类不可见，如果子类定义了一个和父类private方法相同的方法，实为新增方法

重写方法的修饰符权限一定要大于或等于被重写方法的修饰符权限(public > protected > default )

重写抛出的异常需与父类相同或是父类异常的子类，或者重写方法干脆不写throws

重写方法的返回值必须与被重写方法一致，否则编译报错

静态方法不能被重写

（5）什么是多态?同一操作作用于不同的对象，可以产生不同的执行结果，这就是多态性。

（6）Java的多态性体现在两个方面：由方法重载实现的静态多态性（编译时多态）和方法重写实现的动态多态性（运行时多态）。

编译时多态：在编译阶段，具体调用哪个被重载的方法，编译器会根据参数的不同来调用相应的方法。

运行时多态：当父类引用引用指向子类对象时，被引用的对象的类型决定了调用谁的方法，这个被调用的方法必须是在父类中定义过的，也就是说被子类重写的方法。

（7）实例讲解

1 class Animal {

2 public void speak() {

3 System.out.println("我是动物");

4 }

5 }

6

7 class Dog extends Animal {

8 public void speak() {

9 System.out.println("汪汪...");

10 }

11 }

12

13 class Cat extends Animal {

14 public void speak() {

15 System.out.println("喵喵...");

16 }

17 }

18

19 public class Test {

20 public static void main(String args[]) {

21 Animal animal = new Dog();

22 animal.speak(); // 汪汪...

23 animal = new Cat();

24 animal.speak();// 喵喵...

25 }

26 }

（8）稍微复杂一点的案例:

1 class Animal {

2 public void speak1() {

3 System.out.println("我是动物..");

4 speak2();

5 }

6 private void speak2() {

7 System.out.println("我是一只凶猛的动物");

8 }

9 }

10

11 class Dog extends Animal {

12 public void speak1(String name) {

13 System.out.println("汪汪...");

14 }

15 public void speak2() {

16 System.out.println("我是一只泰迪");

17 }

18 }

19

20 public class Test {

21 public static void main(String args[]) {

22 Animal animal = new Dog();

23 animal.speak1();

24 // 我是动物..

25 // 我是一只凶猛的动物

26 }

27 }

## 三、对象和类的概念

**1 对象是用于计算机语言对问题域中事物的描述**，对象通过“属性（attribute）”和“方法（method）”来分别对应事物所具有的静态属性和动态属性。

**2 类是用于描述同一类的对象的一个抽象的概念**，类中定义了这一类对象所具有的静态属性和动态属性类可以看成一类对象的模板，对象可以看成该类的一个具体实例。

**3 什么叫瓶子？**

瓶子的定义：具有某些类特征的东西就是瓶子，比分说什么样的形状，比方说有个口，能倒水，能装水，一般有个盖等等。给瓶子下定义的过程，其实就是把瓶子里的某些东西抽象出来了，所以瓶子在这里是叫做一类事物的一个抽象，在你脑子里有瓶子的概念，可瓶子的概念在你脑子里到底是什么呢？瓶子的概念在你脑子里叫做一类事物的一个抽象。怎么抽象的呢？你往往抽象的是这两个方面：第一个方面我们叫它静态的属性，瓶子应该具有哪些特征，比分说瓶子应有个口，这是它的具有的一个静态属性，瓶子一般有一个盖，这也是它的具有一个静态属性，除此之外，你还可能给它总结动态的属性，什么动态的属性呢？比放说瓶子能倒水，这是它的动态属性。瓶子这个概念在你脑子里如果你细细的思维的话，其实你给它做了两方面的总结，一方面是静态的，一方面是动态的。反映到JAVA的类上，一个就是成员变量（静态属性），一个就是方法（动态属性）方法是可以执行的，可以动的。成员变量是某一个类的静态属性。所以你脑子里瓶子的概念实际上是一类事物的一个抽象，这种东西我们叫它类，椅子是类，桌子是类，学生是类。什么是对象呢？这一类事物的具体的某个实例就叫做对象。所以一类事物的具体的某一个东西，符合这类事物具体的特征的某个东西就叫做对象。瓶子是一个类，某个瓶子就是瓶子这个类里面的一个对象。

**4 如何抽象出一个类？**

有两个方面，一方面是它的静态属性，另一方面是它的动态属性。反映到JAVA里面的类怎么包装它呢？一方面成员变量，另一方面是方法。

eg.职员这个类该怎么抽象出来？也是从两个方面，一方面是它的静态属性，另一方面它的动态属性职员有哪些属性呢？有姓名，年龄，目前工资数额等属性，他有哪些方法呢？让这个职员来显示姓名，显示年龄，修改姓名，领取工资。当然显示姓名，显示年龄，修改姓名，领取工资这些也可以让别人来做，但面向对象的设计思维是最合适的方法应该出现在最合适的类里面。显示姓名，显示年龄，修改姓名，领取工资由谁来做更合适呢，那就是职员自己最合适。所以这些方法应该出现在职员这个类里面。

**5 对于类来说，它有一些属性或者称为成员变量**，以后说属性或者成员变量指的是同一回事。具体的对象他有没有相关的一些属性或者叫成员变量呢？有，每一个人都有一份，只不过是取值不同而已。如从职员这个类实例化出来的两个职员：职员A和职员B，他们都有姓名，年龄，目前工资数额这些属性，但他们的名字，年龄，领取的工资数额都不一样。这样就能把职员A和职员B区分开来了，正是因为他们的属性值不一样，所以这个对象才能和另外的对象区分开来，所以通过属性是可以区分两个对象的。猫是一个类，这只猫是一个对象，这只猫和另外一只猫该怎么区分开来呢？那就得看你的猫这个类是怎么定义的了，猫有猫毛，毛有颜色。OK，这只猫是黑猫，另一只猫是白猫，这样通过猫毛的颜色区分开来了。如果只定义一个，如捉老鼠，白猫也能捉，黑猫也能捉，这样就没办法区分出黑猫和白猫了，所以根据方法是没办法区分两个对象的。所以每个对象都有自己的属性，属性值和另外一个对象一般是不一样的。

**6 一定要区分类和对象，什么叫做类？**什么叫做对象？类是一类事物的一个抽象，具有共同特征的一类事物的一个抽象。对象是这个类具体的某一个实例，所以以后说实例（instance）或者说对象（object）指的是同一回事。

## 四、类（对象）之间的关系

**1 关联关系(最弱的一种关系)**

类和类之间是有关系的，如学生和老师这两个类，老师可以教学生，学生可以向老师学习。这就是他们之间的关系。关系和关系之间是不同的，你和你老婆的关系和你和你其他女朋友的关系是不能混为一谈的。关系最弱的一种关系叫关联关系。关联关系反应到代码上往往是一个类的方法里面的参数是另一个类的具体的某一个对象，比如教授教研究生，教哪个研究生，教是教授这个类里面的一个方法，某个研究生是研究生这个类里面的一个具体的对象。关联关系是最弱的一种关系，咱们两个类之间有关系，或者两个对象之间有关系，但关系不是很紧密。

**2 继承关系（比较强的一种关系）**

继承关系封装了这样一种逻辑：“XX是一种XX”，只要这东西能说通了，就可以考虑用继承关系来封装它。如：游泳运动员从运动员继承，游泳运动员是一种运动员，这能说得通，所以游泳运动员就是从运动员继承过来的，游泳运动员和运动员就是一种继承关系。学生是一个人，学生从人继承，老师是一个人，老师也从人继承，学生是一种老师，这说不通，所以学生和老师就不存在继承关系。所以将来做设计的时候要分清继承关系很简单，你只要说通这么一句话：“XX是一种XX”。OK，那他们之间就是继承关系。篮球运动员是一种球类运动员，这说得通，所以篮球运动员从球类运动员继承，这样继承很有可能会产生一棵继承树，运动员派生出来，派生出来的意思是游泳运动员这个类、球类运动员这个类、射击运动员类从它继承，相当于运动员派生出来了这些个不同的运动员，包括游泳的，球类的，射击的。球类的再派生足球的，篮球的，排球的。这就是一棵继承树，不过这棵树是比较理想化的情况，只有一个根节点。但实际当中，我们真实世界当中的继承关系不一定只从一个类继承，一个类可能从多个类继承过来，比如说：金丝猴从动物这个类继承，这很正常，但我还有另外一个专门的类：“应该受到保护的东西”，这也是一个类，金丝猴应该受到保护，所以金丝猴是一种应该受到保护的东西。所以金丝猴从“应该受到保护的东西”这个类继承过来。所以在现实情况当中，一个类完完全全有可能从多个不同的类继承，C++正是因为想封装这种继承关系，所以C++存在多重继承。

**3 聚合关系（整体和部分）（比较强的一种关系）**

什么是聚合？聚合就是一个整体与部分的关系。我们说“XX是XX的一部分”，只要说得通，那么他们之间的关系就是聚合关系，队长是球队的一部分，队员是球队的一部分。所以队长和球队是聚合关系，队员和球队也是聚合关系。脑袋是人的以部分，身体和胳膊也是人的一部分，因此脑袋，身体和胳膊与人都是聚合关系。聚合关系分得再细一点的话就可以分成聚集关系和组合关系，比如球队、队长，队员，这三者是聚集关系，假如这个队长既是足球的队长，同时也是篮球的队长，一个人分属两个不同的球队，这是可以的，球队与队长之间没有我离不了你，你离不了我这种情况，所以如果分得更细的话，这种就叫做聚集关系。还有一种情况叫组合，组合说的是咱们俩密不可分，我是你必不可少的一部分。一个人的脑袋不可能既属于你又属于别人，身体也一样，不可能既属于你又属于别人。所以你的身体，脑袋和你是密不可分的，这是一种更加严格的聚合关系，专门给它取了个名字叫组合。

**4 实现关系**

作为父类来说，我觉得我应该具有这样一个方法，但我不知道怎么去实现，谁去实现，我的子类去实现，这就是实现关系。和实现关系息息相关的还有一种关系叫多态。

## 五、Java与面向对象

**1 对象和类是分不开的，必须首先定义类才能有对象。**首先定义方法才能调用。对象是JAVA里面的核心，做任何东西你首先都得给我造出一个对象才能做。静态属性简称属性，

也叫成员变量，以后说属性或者说成员变量它们指的都是同一回事。整个类可以看作是静态的属性还有方法他们之间的一个综合。怎么抽象出一个类的概念，还是那句话，你必

须抽象出两个方面来，第一方面是看他们的静态属性，他们有哪些成员变量，第二方面是看他们的有哪些方法。写JAVA程序时，我们一上来写的就是public class（声明一个类

），在这个class里面我们写的是成员变量和方法。每一个JAVA里面的class（类）都对应了我们现实生活中某一类事物的一个抽象。比如说要在JAVA里面封装一只狗，具体怎么封

装，如何写代码，代码如下：

1 package cn.javastudy.summary;

2 /\*\*

3 \* 一类事物封装到JAVA里面首先得写class，定义这个类，类名是什么可以自己取。

4 \* 这里把类名叫做Dog

5 \*/

6 public class Dog {

7 /\*\*

8 \* 接下来就是写这个狗这个类的属性或者叫成员变量，

9 \* 比如说狗这个类的毛的颜色，怎么定义这个属性呢，

10 \* 首先得定义毛的一个类型,如使用int来定义毛的颜色类型

11 \*/

12 int furcolor; //定义属性：毛的颜色

13 float height; //定义属性：狗的高度

14 float weight; //定义属性：狗的体重

15

16 /\*\*

17 \* 狗的颜色，高度，体重这些属性定义完了，接下来要定义的就是方法了。

18 \* 如写一个CatchMouse（）方法，捉老鼠的方法。

19 \* CatchMouse这个方法里面有一个对象类型的参数，捉哪一只老鼠，这个对象参数是属于Mouse这个类的

20 \* @param m

21 \*/

22 void CatchMouse(Mouse m){

23 //在方法体内写捉老鼠这个过程，怎么捉，跑着捉，走着捉

24 System.out.println("我捉到老鼠了，汪汪！，老鼠要尖叫了！");

25 /\*\*

26 \* 老鼠尖叫一声，表示被狗咬到了，咬到了能不叫吗，很自然而然地想到，

27 \* 尖叫（scream()）这个方法是属于Mouse这个类里面的某一个方法。

28 \* 老鼠自己调用它，让它自己尖叫。这就是面向对象的思维。

29 \*/

30 m.scream();

31 }

32

33 public static void main(String[] args) {

34 Dog d = new Dog();//首先用new关键字创建一只狗

35 Mouse m=new Mouse();//造出一只老鼠。

36 d.CatchMouse(m);//然后用这只狗去抓老鼠，让狗调用CatchMouse()方法去捉某只老鼠。

37 }

38 }

1 package cn.javastudy.summary;

2 /\*\*

3 \* 封装的老鼠类

4 \*/

5 public class Mouse {

6 /\*\*

7 \* 老鼠自己有一个发出尖叫的方法

8 \* 当被狗咬到时就会发出尖叫

9 \*/

10 public void scream() {

11 System.out.println("我被狗咬到了，好痛啊！");

12 }

13

14 }

**2 从这个意义上来讲，JAVA里面的每定义一个类实际上就相当于一种新的数据类型。**就跟int ，float， String等一样，不过是一种新定义的类型而已。

**3 为什么使用面向对象编程？**

面向对象编程：一组对象互相配合通过沟通完成特定功能。做软件苦苦追求的一种境界是可重用性（reusable），可扩展性。如果是面向过程，一般情况是属性和方法它们是分开的，他们不是聚合的关系，不是合在一起的，这样要复用起来比较麻烦，复用的层次只是局限于方法这个层次上，而面向对象则不同，它是把属性和方法综合在一个里面。综合在一起复用的时候是整个对象进行复用。所以面向对象和面向过程相比，前者更加容易让我们达到可重用性。

## 六、Java中的面向接口编程

**1 面向接口编程是很多软件架构设计理论都倡导的编程方式。**

**2 接口体现的是一种规范和实现分离的设计哲学，**充分利用接口可以极好地降低程序各模块之间的耦合，从而提高系统的可扩展性和可维护性。基于这种原则，

通常推荐“面向接口”编程，而不是面向实现类编程，希望通过面向接口编程来降低程序的耦合。下面分两种常用场景来示范“面向接口”编程的优势。

**3 简单工厂模式**

有一个场景，假设程序中有个Comupter类需要组合一个输出设备，现在有两个选择：直接让Comupter该类组合一个Printer属性，或者让Comupter组合一个Output属性，那么到底采用哪种方式更好呢？假设让Computer组合一个Printer属性，如果有一天系统需要重构，需要使用BetterPrinter来代替Printer，于是我们需要打开Computer类源代码进行修改。如果系统中只有一个Computer类组合了Printer属性还好，如果系统中有100个类组合了Printer属性，甚至1000个，10000个……将意味着我们要打开100个、1000个、10000类进行修改，这是多么大的工作量！为了避免这个问题，我们让Comupter组合一个Output属性，将Comupter类与Printer类完全分离。Computer对象实际组合的是Printer对象，还是BetterPrinter对象，对Computer而言完全透明。当Printer对象切换到BetterPrinter对象时，系统完全不受影响。下面是这个Computer类定义的代码。

public class Computer

{

private Output out;

public Computer(Output out)

{

this.out = out;

}

//定义一个模拟获取字符串输入的方法

public void keyIn(String msg)

{

out.getData(msg);

}

//定义一个模拟打印的方法

public void print()

{

out.out();

}

}

**4 上面的Computer类已经完全与Printer类分离开**，只是与Output接口耦合。Computer不再负责创建Output对象，系统提供一个Output工厂来负责生成Output对象。这个OutputFactory工厂类代码如下：

public class OutputFactory

{

public Output getOutput()

{

return new Printer();

}

public static void main(String[] args)

{

OutputFactory of = new OutputFactory();

Computer c = new Computer(of.getOutput());

c.keyIn("疯狂Java讲义");

c.keyIn("轻量级J2EE企业应用实战");

c.print();

}

}

**5 在该OutputFactory类中包含了一个getOutput方法**，该方法返回一个Output实现类的实例，该方法负责创建Output实例，具体创建哪一个实现类的对象由该方法决定

（具体由该方法中粗体部分控制，当然也可以增加更复杂的控制逻辑）。如果系统需将Printer改为BetterPrinter实现类，只需要让BetterPrinter实现Output接口，并改变OutputFactory类中的getOutput方法即可。

**6 下面是BetterPrinter实现类的代码**，BetterPrinter只是对原有的Printer进行简单修改，以模拟系统重构后的改进。

public class BetterPrinter implements Output

{

private String[] printData = new String[MAX\_CACHE\_LINE \* 2];

//用以记录当前需打印的作业数

private int dataNum = 0;

public void out()

{

//只要还有作业，继续打印

while(dataNum > 0)

{

System.out.println("高速打印机正在打印：" + printData[0]);

//把作业队列整体前移一位，并将剩下的作业数减1

System.arraycopy(printData , 1, printData, 0, --dataNum);

}

}

public void getData(String msg)

{

if (dataNum >= MAX\_CACHE\_LINE \* 2)

{

System.out.println("输出队列已满，添加失败");

}

else

{

//把打印数据添加到队列里，已保存数据的数量加1。

printData[dataNum++] = msg;

}

}

}

上面的BetterPrinter类也实现了 Output接口，因此也可当成Output对象使用，于是我们只要把OutputFactory工厂类的getOutput方法中粗体部分改为如下代码：

return new BetterPrinter();再次运行前面的OutputFactory.java程序，发现系统运行时已经改为BetterPrinter对象，而不再是原来的Printer对象。通过这种方式，我们把所有生成Output对象的逻辑集中在OutputFactory工厂类中管理，而所有需要使用Output对象的类只需与Output接口耦合，而不是与具体的实现类耦合。即使系统中有很多类使用了Printer对象，只要OutputFactory类的getOutput方法来生成Output对象是BetterPrinter对象，则它们全部都会改为使用BetterPrinter对象，而所有程序无需修改，只需要修改OutputFactory工厂的getOutput的方法实现即可。

## 七、深入理解OOP-java版本

**1 面向对象程序设计，不应该拘泥于具体到语言层面的继**承规则，而应该单纯的去分析OOP的概念。每种语言对于OOP的理念，都有不同的处理，具体到继承，具体到访问控制。以目前而言相对经典的两种OOP语言C++和java来讲，它们在面向对象的问题上，如果在你眼里是相去甚远的，只能说你对面向对象的理解还是停留在具体的语法和语义上。

**2 面向对象程序设计（OOP）的精髓在于：对象驱动，向上转型，后期绑定。**

所谓面向对象，是对具体业务的层层抽象，形成概念，以概念为核心，增加对外接口，以接口为通道，联系业务。面向对象的理念和面向过程的理念的区别是，后者认为程序是一系列动作的集合，而前者认为程序是一系列业务的集合。

**3 java提供了基本类型的包装类**，在java中，所有的事物都是对象或者对象的一部分。并且，所有的类都直接或者间接地继承自Object，Object是一个根类。这也是java彻底的对向驱动的根据。相对于面向过程的程序设计，面向对象更安全，更容易设定访问权限的边界。

**4 java用于指定权限边界的关键字有三个public private 和protected。public是最广泛的访问权限**，允许无条件的访问对应的域和方法；private是最严格的访问权限，只允许当前类中对该域和方法的访问；protect是一种折衷的选择，允许当前类和被继承之后的类对当前域和方法的访问。除此之外，还有一种缺省的访问权限，即不使用任何关键字指定访问权限级别，这个时候是包访问权限。

**5 然而，在很多时候，理解访问权限不能只根据public、private和protected关键字，很多缺省的权限也许更重要。**

对于普通类而言，只存在两种访问权限，缺省的包访问权限和public访问权限。这是很容易理解的，如果是protected或者private，以为这类是不可被外界访问的，而这个类自然失去了存在的意义。只有一种情况下是例外，那就是类属于内部类，这个时候因为有了相应的外部类包装，从而可以有对应的继承机制，可以有相应的private 和protected权限控制。在继承机制里面，任何基类的方法必须是public的。这也很容易理解，因为派生类必须实现基类的所有方法，即使使用基类版本，如果不是public的，那么会产生编译错误。

**6对于单例模式而言，我们需要该类不允许被自由创建，**这个时候可以使用指定所有的构造方法为private，并且用static在类中初始化一个构造方法。在这里，需要介绍static和final两个关键词，因为对于static和final而言，经常有一些和权限控制交叉的意义。static的含义是，当声明一个事物是static时，就意味着这个域或者方法不会与包含它的那个类的任何实例关联在一起。

**7 final的含义是，无法改变的，这**个逻辑最初是为了设计和效率，但是随着软硬件技术的提升，后者的意义变得越来越不明显。相比于static，final更为复杂。

**final数据有两方面含义：1.一个永不改变的编译时常量 2.一个在运行时被初始化的值，你并且不希望它被改变。**

final方法也有两方面含义：第一个方面是要锁定方法，以防止任何继承类修改它的含义。第二方面的含义是出于效率的考虑，这涉及到java虚拟机原理方面的知识。

final和private在某些方面是类似的。是的，所有的private方法，都会被隐式地指定为final。

final类的含义你可以类比推测，即你不认为该类是应该被继承的。不同的是，final类中只有方法是隐式final的，每一个数据域，仍然有final和非final之分，如果你仔细思考，也能自己理解这个细微差别。

**8 因为static和final经常混用**，也许你并不知道static和final的区别所在。对于任何final域，static版本和非static版本是有很大区别的。简而言之，对于每一个对象，都会在内存中保存一份final的数据，而对于static final而言，所有的对象共用一份static final数据。例如下面代码：

1 package oop;

2 import java.util.Random;

3 public class oop {

4 private static Random rand=new Random(100);

5 private static final int onlyOne=rand.nextInt(20);

6 private final int mightMore=rand.nextInt(30);

7 public oop(){

8 }

9 public void getOnlyOne() {

10 System.out.println("onlyOne is :"+this.onlyOne);

11 }

12 public void getMightMore() {

13 System.out.println("mightMore is:"+this.mightMore);

14 }

15 public static void main(String []args){

16 oop op1=new oop();

17 oop op2=new oop();

18 op1.getOnlyOne();

19 op1.getMightMore();

20 op2.getOnlyOne();

21 op2.getMightMore();

22 }

23 }

24 //result:

25 onlyOne is :15

26 mightMore is:10

27 onlyOne is :15

28 mightMore is:4

**9 抽象类和接口是java中面向对象、概念与业务分离的关键环节。**抽象类必须使用abstract关键字修饰类名，意味着类中包含着没有定义的方法（只有类名），换句话说，包含抽象方法的类，一定是抽象类。对于接口，则做了更加严格的限制。接口使用interface而非class这个关键字，成为一个极度抽象的类，意味着接口内部所有的域都隐式地指定为static final，所有的方法都是抽象方法。由于抽象类和接口都包含没有定义的方法，所以不能初始化一个抽象类或者接口，他们从创建之初，目的就是被继承的。

**10 所谓后期绑定，是OOP中特别关键的概念**，即编译阶段并不能确定所要执行的代码，只有运行阶段才能确定。这在多态机制中，意味着运行过程中会根据对象的类型去执行对应的方法，而避免错误地执行基类版本，这是符合逻辑的。

阐述后期绑定的概念，我们应该先理清一个对象初始化的顺序，先看代码：

1 package oop;

2 public class Father {

3 public Father() {

4 System.out.println("基类版本构造函数");

5 }

6 }

7 package oop;

8 public class Child extends Father{

9 public Child() {

10 System.out.println("子类版本构造函数");

11 }

12 }

13 public static void main(String []args){

14 Father f=new Child();

15 Child c=new Child();

16 }

17 //result:

18 基类版本构造函数

19 子类版本构造函数

20 基类版本构造函数

21 子类版本构造函数

根据上面的代码，我相信你能理解最基本的构造方法的构造顺序，一定程度上也能理解后期绑定的原理。

构造方法的后期绑定相对简单，下面介绍普通方法的后期绑定机制：

1 package oop;

2 public class Father {

3 private static int mem=1;

4 public Father() {

5 System.out.println("基类版本构造函数");

6 }

7 public void Out() {

8 System.out.println("这里是父类的输出方法"+mem);

9 }

10 }

11

12 package oop;

13 public class Child extends Father{

14 private static int mem;

15 public Child() {

16 System.out.println("子类版本构造函数");

17 System.out.println("子类的mem初始值"+mem);

18 this.mem=2;

19 }

20 // public void Out() {

21 // System.out.println("这里是子类的输出方法"+this.mem);

22 // }

23 }

24

25 public static void main(String []args){

26 Father f=new Child();

27 Child c=new Child();

28 f.Out();

29 c.Out();

30 //加注释版本：

31 基类版本构造函数

32 子类版本构造函数

33 子类的mem初始值0

34 基类版本构造函数

35 子类版本构造函数

36 子类的mem初始值2

37 这里是父类的输出方法1

38 这里是父类的输出方法1

39 //不加注释版本：

40 基类版本构造函数

41 子类版本构造函数

42 子类的mem初始值0

43 基类版本构造函数

44 子类版本构造函数

45 子类的mem初始值2

46 这里是子类的输出方法2

47 这里是子类的输出方法2

**11 通过上面的运行结果可以提炼下面几条**

（1）子类对象会先寻找子类中的对应方法，没有的时候会向上访问基类的版本；

（2）基本类型静态变量没有指定初始值的时候为基本类型的初始值，对象引用静态变量的初始值为null；

（3）对于静态变量，基类版本和子类版本并不保持静态一致性，这意味着在内存中，静态变量只使用一份内存，是说每一个类的静态变量只使用一份内存；

（4）上面同时可以看出，静态域访问的是当前对象的实际类型的版本，但是对于静态方法而言，并非如此：

1 package oop;

2 public class Father {

3 private static int mem=1;

4 public Father() {

5 System.out.println("基类版本构造函数");

6 }

7 public void Out() {

8 System.out.println("这里是父类的输出方法"+mem);

9 }

10 public static void OutLn() {

11 System.out.println("这是父类的静态函数");

12 }

13 }

14

15

16 package oop;

17 public class Child extends Father{

18 private static int mem;

19 public Child() {

20 System.out.println("子类版本构造函数");

21 System.out.println("子类的mem初始值"+mem);

22 this.mem=2;

23 }

24 public void Out() {

25 System.out.println("这里是子类的输出方法"+this.mem);

26 }

27 public static void OutLn() {

28 System.out.println("这是子类的静态方法");

29 }

30 }

31 public static void main(String []args){

32 Father f=new Child();

33 Child c=new Child();

34 f.OutLn();

35 c.OutLn();

36 }

37 //result:

38 基类版本构造函数

39 子类版本构造函数

40 子类的mem初始值0

41 基类版本构造函数

42 子类版本构造函数

43 子类的mem初始值2

44 这是父类的静态函数

45 这是子类的静态方法

**12 上面这种情况被称为是java的多态陷阱！**

对于静态域和静态方法的访问机制是不同的。静态域的访问符合我们一般意义上的后期绑定，但是静态方法是前期绑定的，也就是说：

静态方法与类相关，而非与对象相关。

实际上，在java中只有两种方法是不是后期绑定。那就是static方法和final方法，final方法不需要多说，因为他不能被子类覆盖，所以子类执行final方法，只能回溯到父类。

当然，如果牵扯到另外一个问题，也许会推翻之前你所有的理解，事实上，只应该是你理解的不全面，而非逻辑的不完美，如下面代码：

1 package oop;

2 public class Father {

3 private static int mem=1;

4 public Father() {

5 System.out.println("基类版本构造函数");

6 this.Cg();

7 System.out.println("基类的mem值为"+mem);

8 }

9 public void Cg() {

10 System.out.println("基类的Cg方法"+mem);

11 this.mem=5;

12 }

13 }

14 package oop;

15 public class Child extends Father{

16 private static int mem;

17 public Child() {

18 System.out.println("子类版本构造函数");

19 System.out.println("子类的mem初始值"+mem);

20 this.mem=2;

21 System.out.println("子类的mem最终值"+mem);

22 }

23 public void Cg() {

24 this.mem=6;

25 System.out.println("子类的Cg方法"+mem);

26 }

27 }

28

29

30 public static void main(String []args){

31 Father f=new Child();

32 Child c=new Child();

33 }

34 //result：

35 基类版本构造函数

36 子类的Cg方法6

37 基类的mem值为1

38 子类版本构造函数

39 子类的mem初始值6

40 子类的mem最终值2

41 基类版本构造函数

42 子类的Cg方法6

43 基类的mem值为1

44 子类版本构造函数

45 子类的mem初始值6

46 子类的mem最终值2

**13 是的，在基类构造方法中绑定的是子类的普通方法，访问的是子类的静态域，这是值得思考的。**相对言，向上转型是比较好理解的，前面也使用了向上转型。假设有一个乐器的基类，吉他继承自乐器类，那么一个吉他对象就属于乐器类，这就是向上转型。向上转型是安全的，因为基类的方法和域一定会有对应的子类版本。而向下转型则是危险的，因为java虚拟机不能确定一个乐器一定是吉他。因此向下转型只能由程序员强制操作，程序员也必须确定对象确实是强制转型后的类型。