JAVA类谜题48：我所得到的都是静态的 PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/138/2021_2022_JAVA_E7_B1_BB_E8_B0_9C_c97_138490.htm 下面的程序对巴辛吉小鬣狗和其它狗之间的行为差异进行了建模。如果你不知道什么是巴辛吉小鬣狗，那么我告诉你，这是一种产自非洲的小型卷尾狗，它们从来都不叫唤。那么，这个程序将打印出什么呢？ class Dog { public static void bark() { System.out.print("woof "). } } class Basenji extends Dog { public static void bark() { } } public class Bark { public static void main(String args[]) { Dog woofer = new Dog(). Dog nipper = new Basenji(). woofer.bark(). nipper.bark(). } } 随意地看一看，好像该程序应该只打印一个woof。毕竟，Basenji扩展自Dog，并且它的bark方法定义为什么也不做。main方法调用了bark方法，第一次是在Dog类型的woofer上调用，第二次是在Basenji类型的nipper上调用。巴辛吉小鬣狗并不会叫唤，但是很显然，这一只会。如果你运行该程序，就会发现它打印的是woof woof。这只可怜的小家伙到底出什么问题了？ 问题在于bark是一个静态方法，而对静态方法的调用不存在任何动态的分派机制[JLS 15.12.4.4]。当一个程序调用了一个静态方法时，要被调用的方法都是在编译时刻被选定的，而这种选定是基于修饰符的编译期类型而做出的，修饰符的编译期类型就是我们给出的方法调用表达式中圆点左边部分的名字。在本案中，两个方法调用的修饰符分别是变量woofer和nipper，它们都被声明为Dog类型。因为它们具有相同的编译期类型，所以编译器使得它们调用的是相同的方法：Dog.bark。这也就解释了为什么程序打印出woof woof

。尽管nipper的运行期类型是Basenji，但是编译器只会考虑其编译器类型。 要订正这个程序，直接从两个bark方法定义中移除掉static修饰符即可。这样，Basenji中的bark方法将覆写而不是隐藏Dog中的bark方法，而该程序也将会打印出woof，而不是woof woof。通过覆写，你可以获得动态的分派；而通过隐藏，你却得不到这种特性。 当你调用了一个静态方法时，通常都是用一个类而不是表达式来标识它：例如，Dog.bark或Basenji.bark。当你在阅读一个Java程序时，你会期望类被用作为静态方法的修饰符，这些静态方法都是被静态分派的，而表达式被用作为实例方法的修饰符，这些实例方法都是被动态分派的。通过耦合类和变量的不同的命名规范，我们可以提供一个很强的可视化线索，用来表明一个给定的方法调用是动态的还是静态的。本谜题的程序使用了一个表达式作为静态方法调用的修饰符，这就误导了我们。千万不要用一个表达式来标识一个静态方法调用。 覆写的使用与上述的混乱局面搅到了一起。Basenji中的bark方法与Dog中的bark方法具有相同的方法签名，这正是覆写的惯用方式，预示着要进行动态的分派。然而在本案中，该方法被声明为是static的，而静态方法是不能被覆写的；它们只能被隐藏，而这仅仅是因为你没有表达出你应该表达的意思。为了避免这样的混乱，千万不要隐藏静态方法。即便在子类中重用了超类中的静态方法的名称，也不会给你带来任何新的东西，但是却会丧失很多东西。 对语言设计者的教训是：对类和实例方法的调用彼此之间看起来应该具有明显的差异。第一种实现此目标的方式是不允许使用表达式作为静态方法的修饰符；第二种区分静态方法和实例方法调用的方式是使用不同的操作符，

就像C 那样；第三种方式是通过完全抛弃静态方法这一概念来解决此问题，就像Smalltalk那样。 总之，要用类名来修饰静态方法的调用，或者当你在静态方法所属的类中去调用它们时，压根不去修饰这些方法，但是千万不要用一个表达式去修饰它们。还有就是要避免隐藏静态方法。所有这些原则合起来就可以帮助我们去消除那些容易令人误解的覆写，这些覆写需要对静态方法进行动态分派。 