多态通过分离做什么和怎么做，从另一角度将接口和实现分离开来。

多态能够改善代码的组织结构和可读性，能够创建可扩展的程序

与封装的关系

封装通过合并特征和行为来创建新的数据类型。”实现隐藏”则通过将细节”私有化”把接口和实现分离开来

多态的作用则是消除类型之间的耦合关系。多态方法调用允许一种类型表现出和其他相似类型之间的区别，只要它们都是从同一基类导出而来的。这种区别是根据方法行为的不同而表示出来的，虽然这些方法都可以通过同一个基类来调用

对象既可以作为它自己本身的类型使用，也可以作为它的基类型使用。而这种把对某个对象的引用视为对其基类型的引用的做法被称为向上转型

将一个方法调用同一个方法主体关联起来称作绑定。

若在程序执行前进行绑定，叫做前期绑定

后期绑定，它的含义是在运行时根据对象的类型进行绑定。后期绑定也叫做动态绑定或运行时绑定。

Java中除了static方法和final方法(private方法属于final方法)之外,其他所有的方法都是后期绑定。这意味着通常情况下，我们不必判定是否应该进行后期绑定—它会自动发生

为什么要为某个方法声明为final呢？

第一：它可以防止其他人覆盖该方法

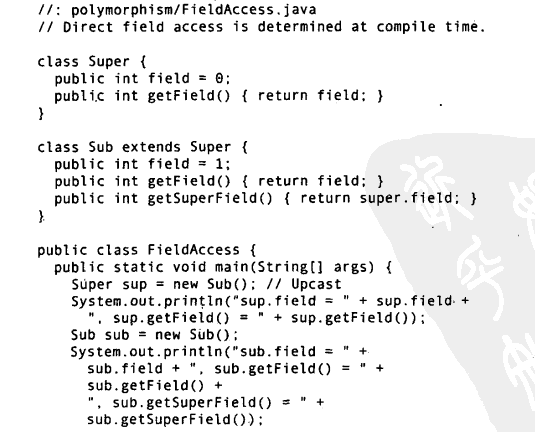
第二：可以有效的”关闭”动态绑定，或者说告诉编译器不需要对其进行动态绑定。这样编译器就可以为final方法调用生成更有效的代码。然而这样做对程序的整体性能不会有什么改观。所以根据设计来决定是否使用final,而不是出于试图提高性能的目的来使用final。

多态是让程序员”将改变的事物与未变的事物分离开来”

缺陷 “覆盖”私有方法

只有非private方法才可以被覆盖,但是还需要密切注意覆盖private方法的现象,这时虽然编译器不会报错，但是也不会按照我们所期望的来执行。确切的说，在导出类中，对于基类中的private方法，最好采用不同的名字。

缺陷 域与静态方法



当Sub对象转型为Super引用时,任何域访问操作都将由编译器解析,因此不是多态的。在本例中,为Super.field和Sub.field分配了不同的存储空间.这样,Sub实际上包含两个称为field的域：它自己的和它从Super处得到的.然而,在引用Sub中的field时所产生的默认域并非Super版本的field域。因此，为了得到Super.field，必须显示的指明super.field

尽管这看起来好像会成为一个容易令人混淆的问题,但是在实践中,它实际上从来不会发生。受限，你通常会将所有的域都设置成private，因此不能直接访问它们,其副作用是只能调用方法来访问.另外,你可能不会对基类中的域和导出类中的域赋予相同的名字，因为这种做法容易令人混淆。

构造器和多态

构造器的调用顺序：

基类的构造器总是在导出类的构造过程中被调用，而且按照继承层次逐渐向上链接，已使每个基类的构造器都能得到调用。

构造器具有一项特殊任务：检查对象是否被正确地构造。导出类只能访问它自己的成员，不能访问基类中的成员(基类成员通常是private类型).只有基类的构造器才具有恰当的知识和权限来对自己的元素进行初始化。因此，必须令所有构造器都得到调用，否则就不可能正确构造完整对象。这正是编译器为什么要强制每个导出类部分都必须调用构造器的原因。如果没有明确指定调用某个基类构造器，它就会”默默”地调用默认构造器。如果不存在默认构造器，编译器就会报错

总结

多态意味着”不同形式”。在面向对象的程序设计中，我们持有从基类继承而来的相同接口，以及使用该接口的不同形式：不同版本的动态绑定方法