目录

[封装 1](#_Toc114656350)

[继承 1](#_Toc114656351)

[多态 2](#_Toc114656352)

[重写与重载的区别？ 2](#_Toc114656353)

[对this和super的认识？ 3](#_Toc114656354)

[谈谈权限修饰符的特性？ 3](#_Toc114656355)

[对java多态的理解 /Java中实现多态的机制是什么？ 3](#_Toc114656356)

[多态是什么？ 3](#_Toc114656357)

[编译时多态 4](#_Toc114656358)

[运行时多态 4](#_Toc114656359)

[多态动态绑定机制？ 4](#_Toc114656360)

[静态属性和静态方法能被继承吗？静态方法又是否能被重写呢？ 5](#_Toc114656361)

[静态绑定和动态绑定的区别 6](#_Toc114656362)

面向对象的三大特性包括：封装、继承、多态。

# 封装

封装通过修饰符赋予对象不同的访问权限，给对象提供了隐藏内部属性和行为的能力，来保护对象内部的状态，防止对象之间的不良交互，提高安全性。

# 继承

继承就是子类继承父类的特征和行为，使得子类对象具有父类的属性和方法，从而避免存在重复的代码，更加简洁，并提高代码的可维护性、复用性。Java中使用extends和implements实现继承。

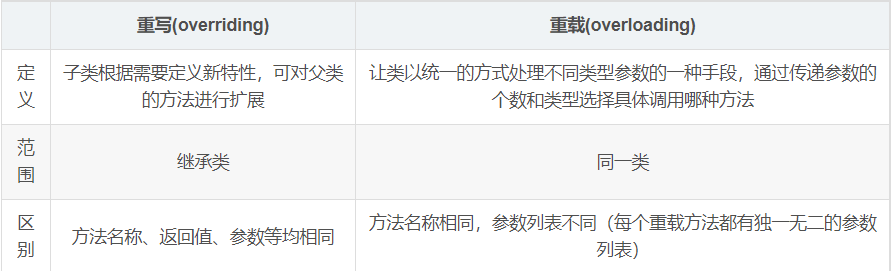
# 多态

多态是同一个行为具有多个不同表现形式或形态的能力。多态就是同一个接口，使用不同的对象而执行不同操作。多态分为编译时多态和运行时多态：编译时多态指方法的重载；运行时多态指程序中定义的对象引用所指向的具体类型在运行期间才确定。运行时多态的三个必要条件：（1）继承（2）重写（3）父类引用指向子类对象。多态可以使程序有良好的扩展，并可以对所有类的对象进行通用处理。

例：声明父类对象指向子类对象，在使用时直接使用父类对象。

# 重写与重载的区别？

方法的重写(Overriding)和重载(Overloading)是java多态性的不同表现，重写是父类与子类之间多态性的一种表现，重载可以理解成多态的具体表现形式。

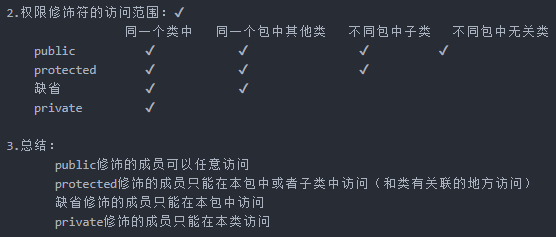


# 对this和super的认识？

this：就是类中指向对象本身的一个特殊引用。

super：指向离自己最近的父类对象的一个指针。

# 谈谈权限修饰符的特性？



# 对java多态的理解 /Java中实现多态的机制是什么？

# 多态是什么？

多态是同一个行为具有多个不同表现形式或形态的能力。多态就是同一个接口，使用不同的对象而执行不同操作。

多态分类？

本质上多态分为两种：

编译时多态（静态绑定），如重载方法。

运行时多态（动态绑定），如重写方法。

# 编译时多态

即在编译阶段方法已经被绑定，由编译器实现，绑定的是类信息。java中的方法，只有final，static，private，重载方法和构造方法还有所有的变量是静态绑定。

# 运行时多态

是运行阶段根据传入的对象信息，才进行绑定相应的方法，绑定的是对象信息。运行时多态通常有两种实现方法：1、子类继承父类（extends）2、类实现接口（implements）。运行时多态的必要条件有：继承、重写、父类引用指向子类对象。

# 使用多态应该遵循的原则是：

定义方法参数时总是优先使用父类类型或接口类型，创建的是实际类型。这样做可以减少代码的耦合，提高代码的可维护性。

# 多态动态绑定机制？

类对象方法的调用必须在运行过程中采用动态绑定机制。动态绑定的解题思路：

编译阶段：

1. 首先，根据对象的声明类型(对象引用的类型)找到“合适”的方法。具体步骤如下：

编译器检查对象的声明类型和方法名称（p.method()）

* 若p的声明类型为Child类，则编译器会列举出Child类中所有名为method方法和父类继承的method方法（方法表），如果有重写，则覆盖；
* 若p的声明类型为Parent类，并指向子类对象，则编译器会列举出Parent类中所有名为method方法，如果有重写，则覆盖；

如果能在实际类型中匹配到方法签名完全一样(参数类型一致)的方法，那么这个方法是最合适的。

1. 在上一条不能满足的情况下，寻找可以“凑合”的方法（最佳匹配法）。标准就是通过将参数类型进行自动转型之后再进行匹配。

自动转型会匹配最精确的一个参数列表，当一个方法的任何参数都可以传递给另一个方法（可接受范围更小），则这个方法更加精确。

1. 如果仍然在声明类型中找不到“合适”的方法，则编译阶段就无法通过。

运行阶段：

编译阶段在声明对象类型的方法表中查找方法，只是为了安全地通过编译（也为了检验方法是否是存在的）。而在实际运行这条语句时，在执行Father ft=new Son();这一句时创建了一个Son实例对象，然后在ft.say()调用方法时，JVM会把刚才的son对象压入操作数栈，用它来进行调用。而用实例对象进行方法调用的过程就是动态绑定：根据实例对象所属的类型去查找它的方法表，找到匹配的方法进行调用。我们知道，子类中如果重写了父类的方法，则方法表中同名表项会指向子类的方法代码；若无重写，则按照父类中的方法表顺序保存在子类方法表中。故此：动态绑定根据对象的类型的方法表查找方法是一定会匹配（因为编译时在父类方法表中已经查找并匹配成功了，说明方法是存在的。这也解释了为何向上转型时父类引用不能调用子类新增的方法：在父类方法表中必须先对这个方法的存在性进行检验，如果在运行时才检验就容易出危险——可能子类中也没有这个方法）。

总结：编译时用父类方法表匹配后，运行时用子类方法表调用。

补充：

动态绑定（后期绑定）是指：在程序运行过程中，根据具体的实例对象才能具体确定是哪个方法。

动态绑定是多态性得以实现的重要因素，它通过方法表来实现：每个类被加载到虚拟机时，在方法区保存元数据，其中，包括一个叫做方法表（methodtable）的东西，表中记录了这个类定义的方法的指针，每个表项指向一个具体的方法代码。如果这个类重写了父类中的某个方法，则对应表项指向新的代码实现处。从父类继承来的方法位于子类定义的方法的前面。

我们假设Father ft=new Son();ft.say();Son继承自Father，重写了say()。

编译

我们知道，向上转型时，用父类引用执行子类对象，并可以用父类引用调用子类中重写了的同名方法。但是不能调用子类中新增的方法。

在代码的编译阶段，编译器通过声明对象的类型（即引用本身的类型）在方法区中该类型的方法表中查找匹配的方法（最佳匹配法：参数类型最接近的被调用），如果有则编译通过。（这里是根据声明的对象类型来查找的，所以此处是查找Father类的方法表，而Father类方法表中是没有子类新增的方法的，所以不能调用。）

编译阶段是确保方法的存在性，保证程序能顺利、安全运行。

运行

ft.say()调用的是Son中的say()，这里就是动态绑定机制的真正体现。

# 静态属性和静态方法能被继承吗？静态方法又是否能被重写呢？

静态属性、静态方法和非静态的属性都可以被继承和隐藏而不能被重写，因此不能实现多态，不能实现父类的引用可以指向不同子类的对象。

非静态方法可以被继承和重写，因此可以实现多态。

# 静态绑定和动态绑定的区别

A.静态绑定是绑定类，动态绑定是绑定对象

B.静态绑定在编译期绑定，动态绑定在运行期绑定

C.静态绑定继承后不能被重写

重载时若没有完全相同的参数，则会向上转型寻找最相近的参数（往继承的根结点找，向上找）