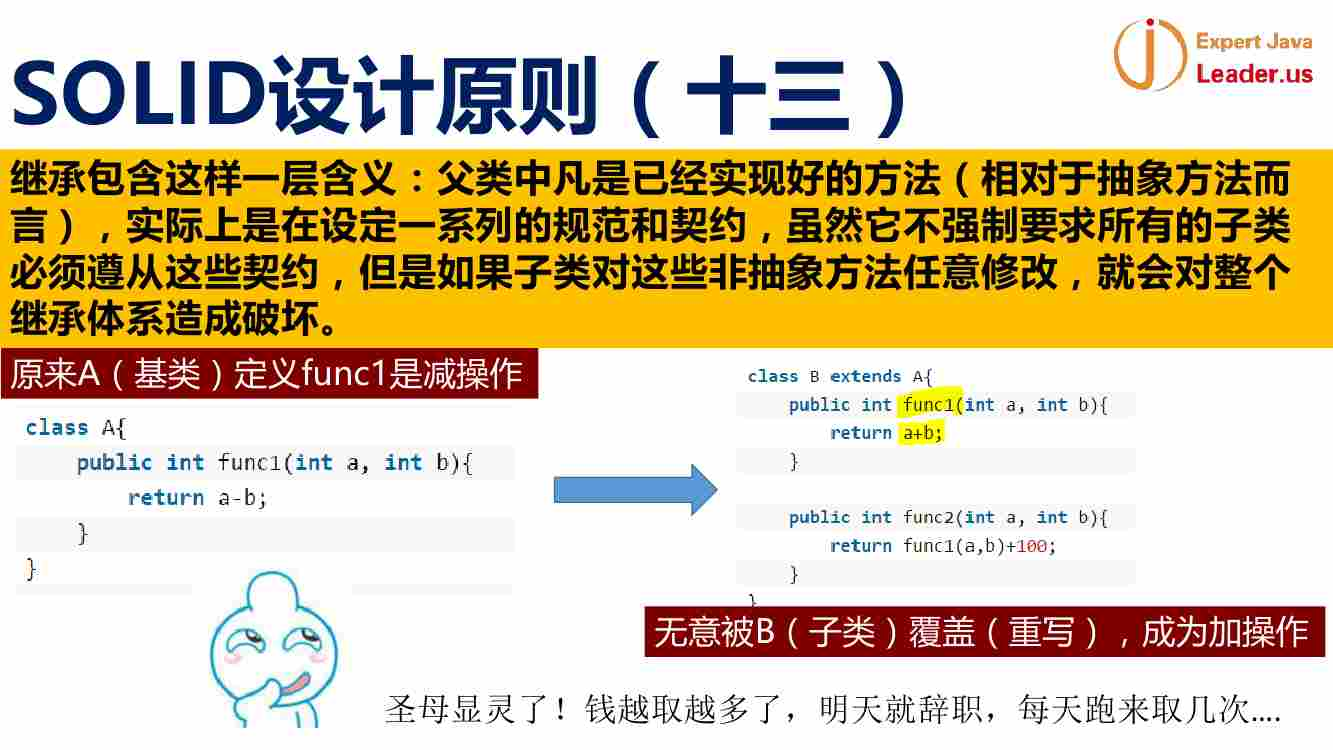


开始看 都是鸭子 小黄鸭是玩具 有电池 小黄鸭不能替代真的鸭子



父类中实现好的方法 不强制子类必须遵守 但是 如果操作不当 没有理解父类中的契约 就把继承搞坏了 ------ 这样 **里氏替换原则就是对继承的规定和约束** --- 可以去继承 要明白父类中的契约再去继承 不能随意破坏

比如繁殖的过程中，可以有基因突变 但是 这个基因突变不能让鸭子编程一头牛

子类还是父类的一部分 而不是完全编程一个新的东西

图中的父类func1是减法操作 但是子类覆盖之后 就变成了加操作 破坏了

这就是为什么有final方法？ ----- 【final的意思就是 最后的 最终的 想想都已经到最终了 怎么可能还有继承 或者重写方法呢 如果这个方法可以被重写 那么这个重写之后的方法就被看成了是一个新的 被覆盖的方法就不是final 类不能被继承 final也是这个意思 变量前面加上final 表示 这个数量值到头了 是最终的了

所以 final的方法到头了 最后的了 ---- 不能被覆盖

Final的类 到头了 最后的了 ---- 不能被继承

Final的变量 到头了 最后的了 ---- 不能被重新赋值 ----- final修饰之后的 就是常量】

比如原来是银行转账的问题 ----- 现在变成了价钱的问题 ----- 理解了里氏替换的方式 就能理解Java中的final的真正的含义 ------ 还有就是 不理解父类中的方法的时候 不要轻易的覆盖 这也就是 Java中为什么要经常使用super.xxx

Super.xxx调用完成之后 就表示先调用父类的 再实现自己的 这就遵循了契约和规范



里氏替换原则 ---- 子类是可以扩展父类的功能 但是不能改变父类的原有功能 ---- 我们在射击子类和父类的时候 子类在扩展的时候 一定要保证 不能改变父类原有的功能！！！这里面引申出来final的方法 ---- 父类的方法是final修饰 保证了语法层面不能被修改

我们面对一个陌生的父类的时候 一定要搞清楚父类中的代码是做什么的

真的要覆盖的时候 要么使用super.xxxx调用父类的方法 要么看懂父类的代码做什么 把父类的代码抄过来

**这样遵循好了以后 可以发现子类也是父类** =====> 子类X是父类T的实现类 这样可以使用子类X来替换父类T【也就是父类出现的地方 子类是一定可以出现的】 === 可以用任何一个子类替换父类 ------ 如果违反了里氏替换原则 发现X和T是不一样的

====》 使用了里氏替换原则之后 定义的时候 可以使用父类的类型去定义 使用的时候 可以使用子类的类型（也就是赋值的时候 使用子类来赋值）----- 这样的地方在Java中是非常常见的 ====》里氏替换原则是**继承原则复用**的基石

也是开闭原则的补充 ---- 这个理解就是 继承父类生成子类之后，子类的扩展（开闭原则）是没有问题的，但是不能产生基因突变（里氏替换原则限制了扩展可以量变 但是不能质变 也就是 不能扩展的太大了 适当的 遵守父类契约的扩展是没有问题的）

**开闭+里氏替换 是软件开发最重要的【我认为考虑设计模式 度量设计规范 首先从这两个原则入手】**

ArrayList是Collection的实现 定义的时候使用的是Collection 传值的时候 使用的是ArrayList

比如Collection是可以遍历的 但是 如果ArrayList不能被遍历 那么 这种替换就违反了里氏替换原则 就会出问题

也就是 继承扩展的时候 不能改变父类的功能



上面是里氏替换原则的具体条款

(1). 子类可以实现父类的抽象方法，但是不能覆盖父类的非抽象方法 ---- 就是父类的非抽象方法不能被随意替换 ---- 如果要替换 也是先调用父类的功能super.xxx 然后再扩展自己的 覆盖是危险的 为了安全起见 父类的方法必要的时候 需要加上final

(2). 子类中可以增加自己特有的方法（这个没有悬念）

(3). 当**子类的方法重载**父类的方法的时候，**形参要比父类的方法的形参类型更宽泛** ---- **也就是父类能够接受的这些参数，你子类一定要能继承下来** ---- 不能是父类能接受的 子类无法接受（**接受不是指的是输入参数么 你的类型宽泛了 父类能接受的 你就都能接收了 就是这个意思 这样才能满足 父类出现的地方 一定能被子类替换 也就是子类可以接受父类的 也可以接受不能接受的 也就是说 父类会的 子类会 父类不会的 子类还会 那么 父类就能被子类替换掉 也就是 父类出现的地方一定能使用子类替换**） === 比如父债子还，不能是老爸欠下的钱 儿子不认账 要认账 所以就反映出 子类的方法 形参比父类宽泛【这个地方要重新理解父类 子类 重写 重载】

(4). 子类的方法实现父类的抽象方法时候，返回值要严格

如果子类做了更宽泛的返回值的时候 ----- 如果更加宽泛，父类**没办法**认可

【怎么理解呢

比如 public class father{

public ArrayList doXxx(){…}

}

使用的时候：ArrayList a = father.doXxx();

当子类的返回值类型是Vector的时候

Public class son extends father{

Public Vector doXxx(){…}

}

里氏替换原则：父类出现的地方 子类一定可以出现 这样 我们直接替换

ArrayList a = son.doXxx()

这样的话 返回的是Vector 是ArrayList的子类 ----Java中允许子类对象赋值给父类引用 所以 子类返回的类型如果比父类的方法返回的严格 仍然满足在父类出现的地方 子类可以替换掉 ---- 就是里氏替换原则

但是 如果

Public class son extends father{

Public List doXxx(){…}

}

子类替换父类

ArrayList a = son.doXxx() 错了 因为 父类的对象不能复制给子类的引用 所以 违反了里氏替换原则

------ 再使用这个推到第三个准则

public class father{

public void doXxx(ArrayList x){…}

}

public class son extends father{

Public void doXxx(Collection x){…}

}

父类使用的地方

Father.doXxx(new ArrayList()) ----》 替换成子类 Son.doXxx(new ArrayList()) 语法正确 没有任何问题

public class son extends father{

Public void doXxx(AttibuteList x){…} [AttibuteList是ArrayList的子类]

}

Father.doXxx(new ArrayList()) ----》 替换成子类 Son.doXxx(new ArrayList()) 语法不正确 因为 还是 父类的对象不能赋值给子类的应用

所以 从参数 和 返回值两个地方 体现了要是父类出现的地方 子类一定可以替换掉 这才是真正的里氏替换原则 ----- 最最根本的就是 子类的对象可以赋值给父类的引用 但是 父类的对象不能赋值给子类的引用 ------ 满足了这个 里氏替换原则就满足了

这个父类的对象不能赋值给子类的引用 分别用在了参数 和返回值上面 就出现了第三条和第四条准则}

】

也就无法完成里氏替换原则了【就是 父类出现的地方 子类一定可以出现 但是你现在父类都不认可 就违反了历史替换原则】

上面的public class A ----- public collection sort(ArrayList data) ----》

Public class B extends A

----- public List getResult(List data)

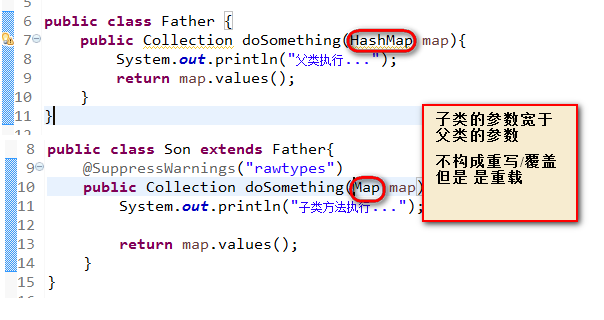
返回值子类比父类严格 但是 参数子类比父类宽泛 ---- 这两点是里氏替换原则的重要标准（问问老师 为什么B类的方法叫getResult呢？老师的PPT是不是写错了？）

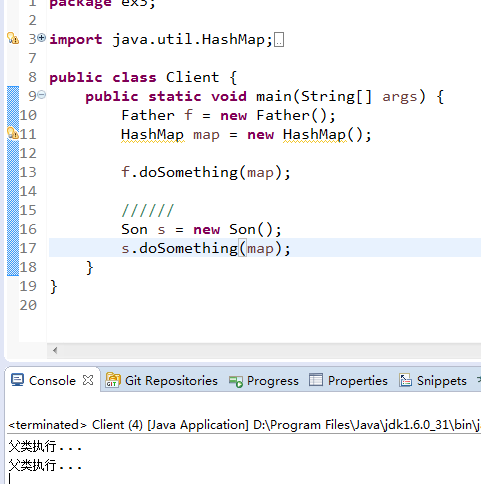
继承的很多都是里氏替换原则的体现

【eclipse上面验证哪一种是重写 哪一种是重载

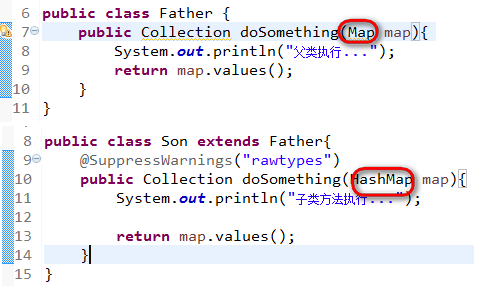
----参数方面

子类(Map)>父类(HashMap) ----- 里氏替换原则

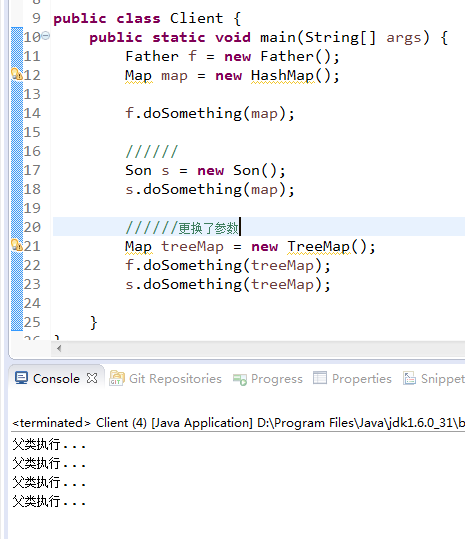


 ----- 重写的方法没有执行 一直都是父类的方法执行 ===== 子类替换之后 执行的还是父类的那一套 遵守了父类的契约 没问题 ----- 因为这个是重载 不是重写 所以 不算是扩展父类的功能 所以 这个子类没有扩展父类的功能

子类(Map)<父类(HashMap)



还是没有构成重写 也不是里氏替换原则



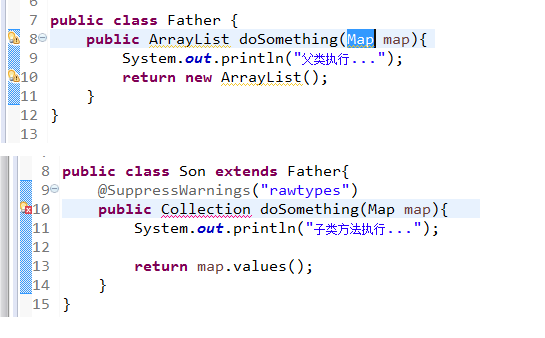
全部都是执行父类的 但是 不是里氏替换原则 最后一种情况 就是 treeMap的时候 直接调用子类继承的父类的方法

从eclipse来看 父子的类型不一样 就是重载 不是重写

再看返回结果

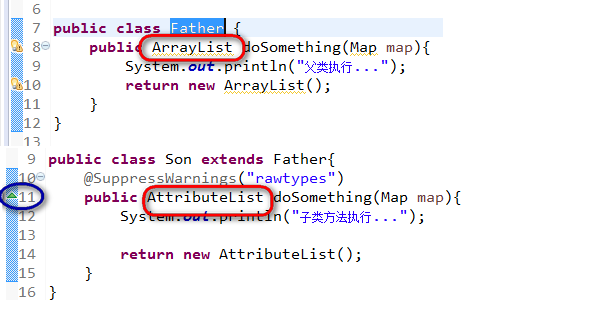
此时保证参数一样 这时候 编译器就认为应该是重写了

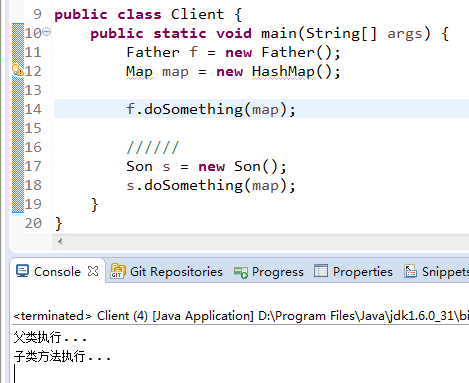
子类返回类型>父类返回类型



这样直接报错

子类返回类型<父类返回类型



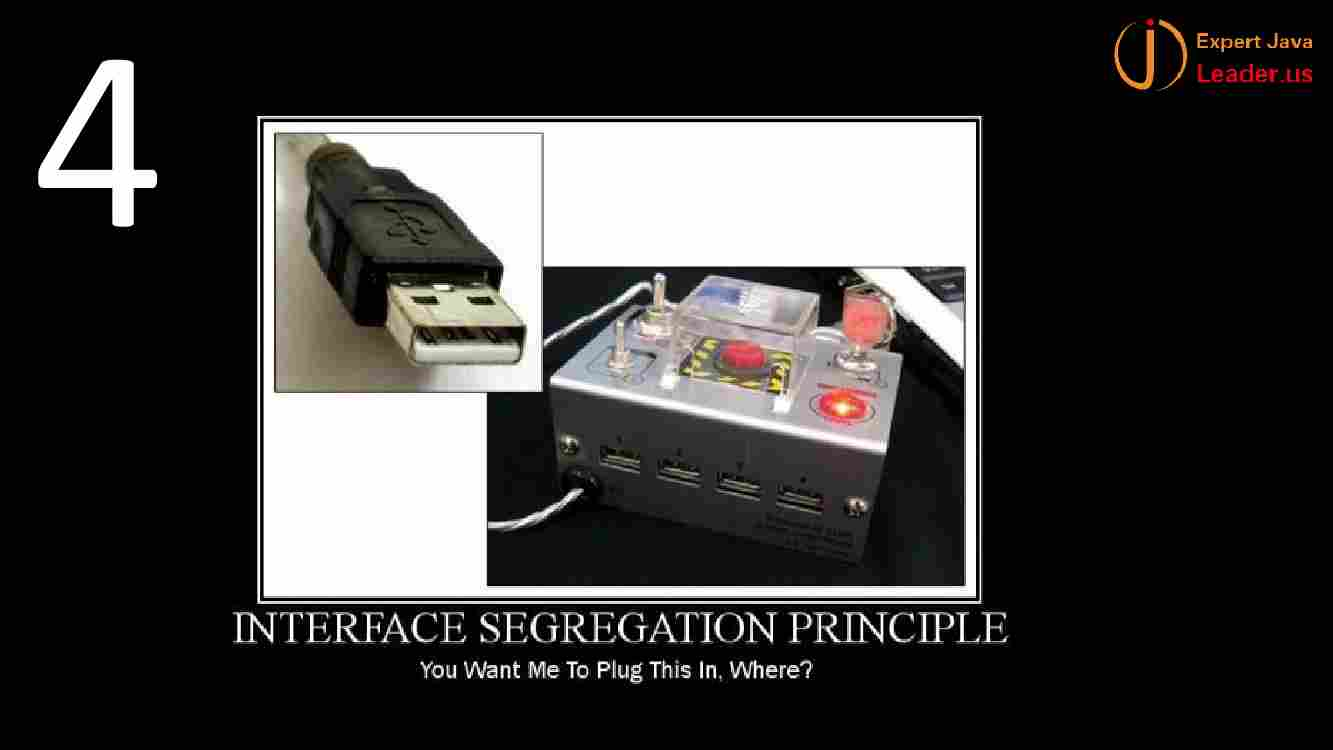


这个时候 是重写 是子类对父类的扩展 所以 子类方法被执行

所以 重载一定是 方法名 参数 一致的情况下 **编译器会判定为重写** 此时 返回值类型 <=父类返回值类型 就完全满足重写的条件 否则直接报错

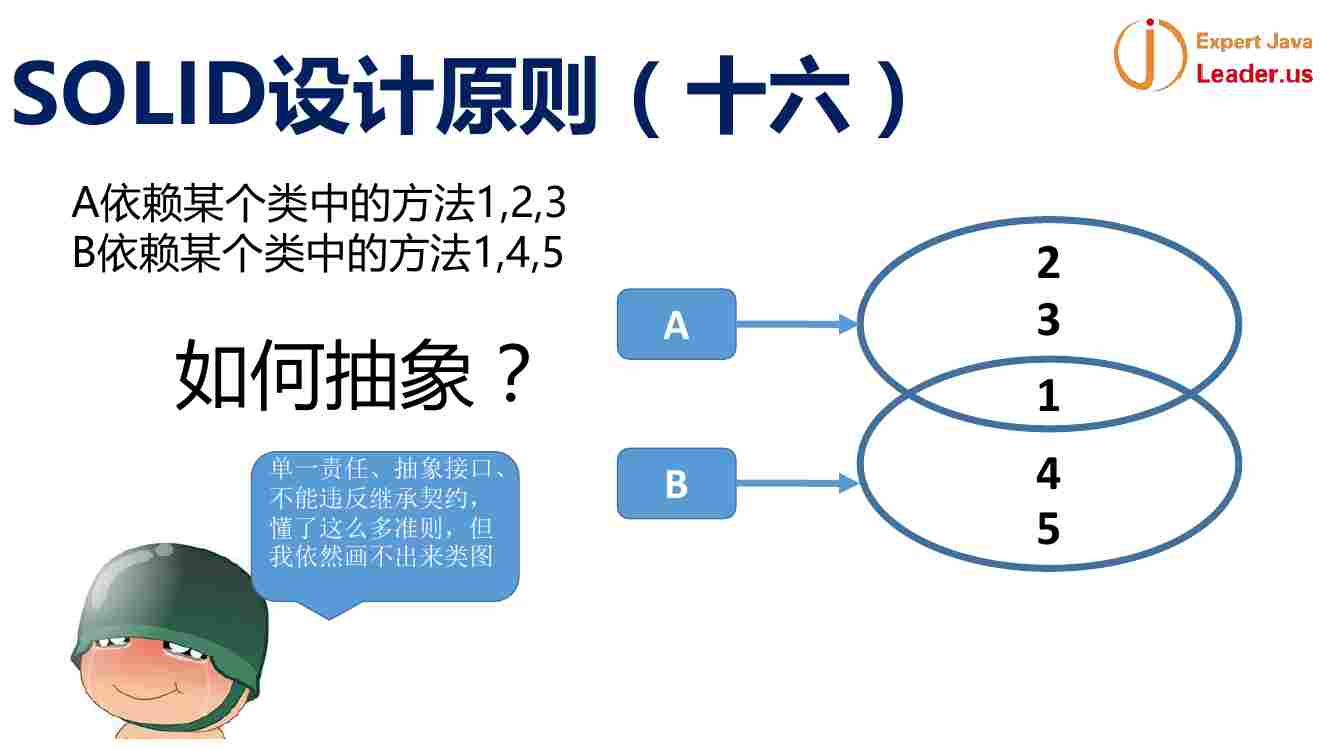
如果方法名一致 参数类型有一点不一样 不会报错 但是 仅仅是重载

】

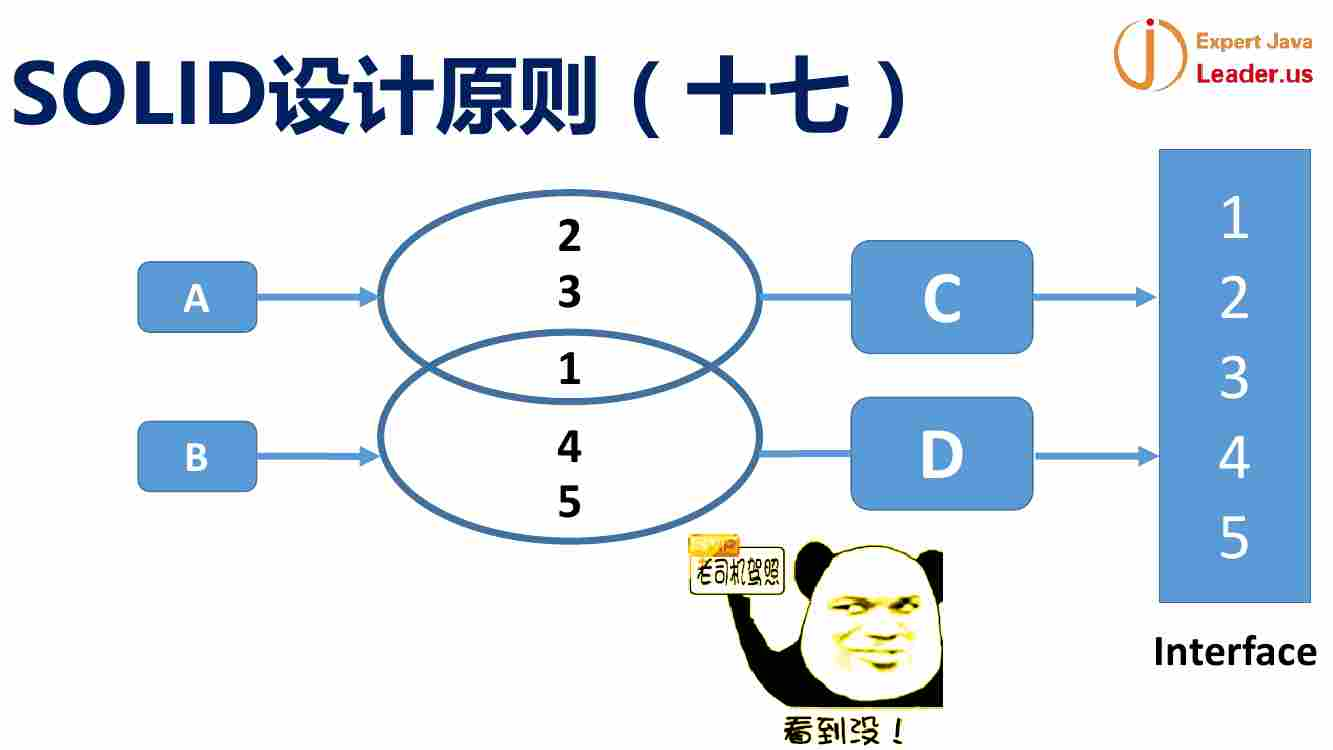


这个里面做了很多插座 到底插到哪一个

很多选择的时候 选择不了



有一个交集1 这种情况如何抽象？ 不能写两份代码



很多人设计如下

直接出一个大的接口 规定了1 2 3 4 5这些接口

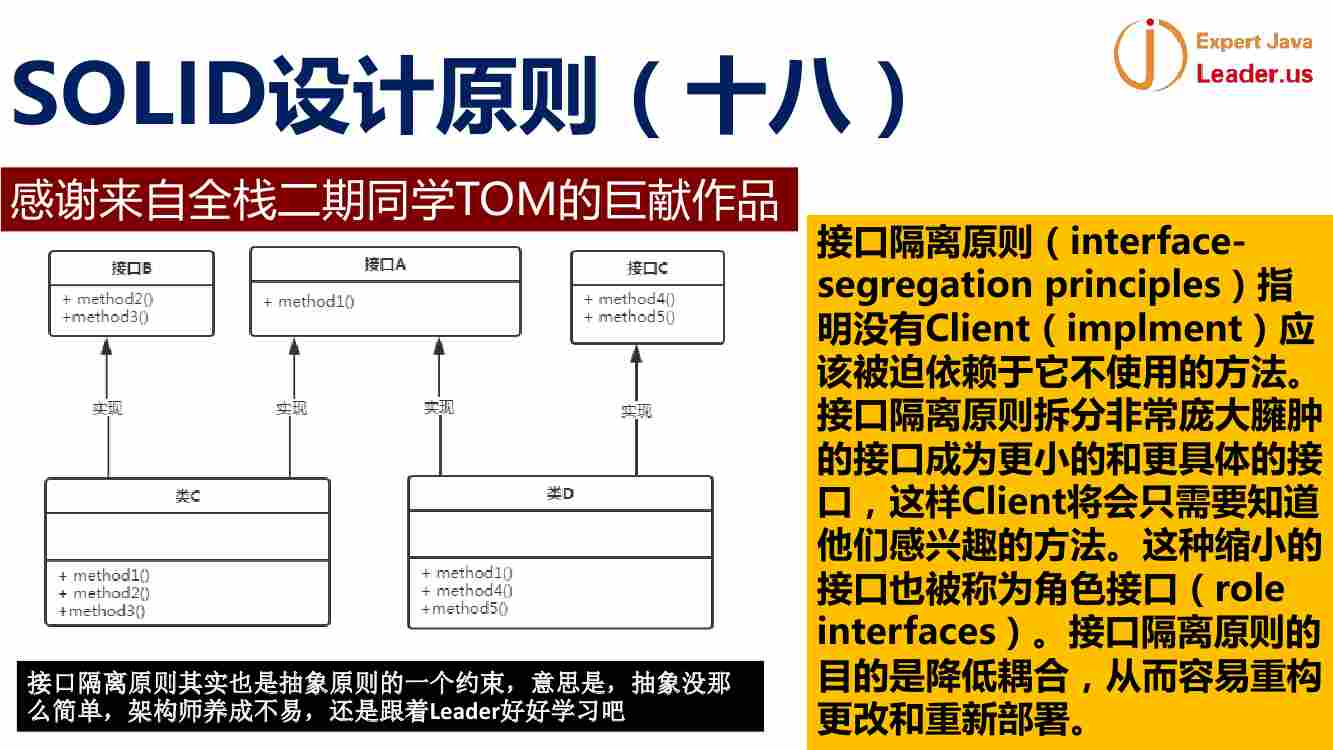
然后 C和D分别实现了1 2 3和1 4 5

最后 A 和 B分别对C和D依赖

A只要用1 2 3就看到了4 5 就会产生疑惑 要不要使用4 5

这样就是给了A一个用不到的方法

实现上，只实现1 2 3 会不会出现问题？



这个类图 就是二期的一个设计 是接口隔离原则的体现

**不要把没用的都柔和在一起 做一个胖接口 ---- 分离出来**

微服务的接口 超过了10个方法 ----- 臃肿的接口类 第一点就违反了单一职责原则

接口隔离原则就是对单一职责原则设计上的一个补充准则

也就是客户端不能被迫实现它用不到的接口 ---- 如果 客户端真的实现了不需要的接口 那么就违反了接口隔离原则 这个接口就应该被分离出去

【接口隔离原则很好的限制了 或者 保证了单一职责原则的执行】

这也是Java很多复杂控件就有很多很多接口 一个类实现很多不同的接口 这就是一个接口仅仅对某些类有用

这个图中 接口C对**类C**没用 B对D没用

降低了耦合 ---- 对D来说 完全不依赖于接口B 对C来说 完全不依赖于接口C

【从前面的图中可以看出来 两个不同的接口 都有同一个或者同几个功能 这个相同的就应该聚类成新的接口 和其他的接口分来 因为学生要吃饭 学习 喝水 考试 司机要吃饭 开车 喝水 驾驶 这样看来 喝水和吃饭是人人都要做的 ----- 一个通用的功能 所以 分离出来】



控制反转原则【也就是spring中所说的DI】

控制反转 一个插头想要一个插座 但是 他仅仅需要这些 而不是需要底层的电烙铁告诉你的插座怎么做的 只想要一个成果

这张图就是 插座是高层的结果 底层的结果是电烙铁等 所以 就是表示我们要关心的操作是高层的操作 不是底层的操作



比如我们要做做一个网关 后面有支付宝 微信支付 在线支付

客户端调用的时候 给支付网关发起请求 而不是对后面的各种具体支付打交道

所以 客户端依赖的是PayGateWay 而不是后面具体的实现

不依赖于支付宝 微信 和 在线支付

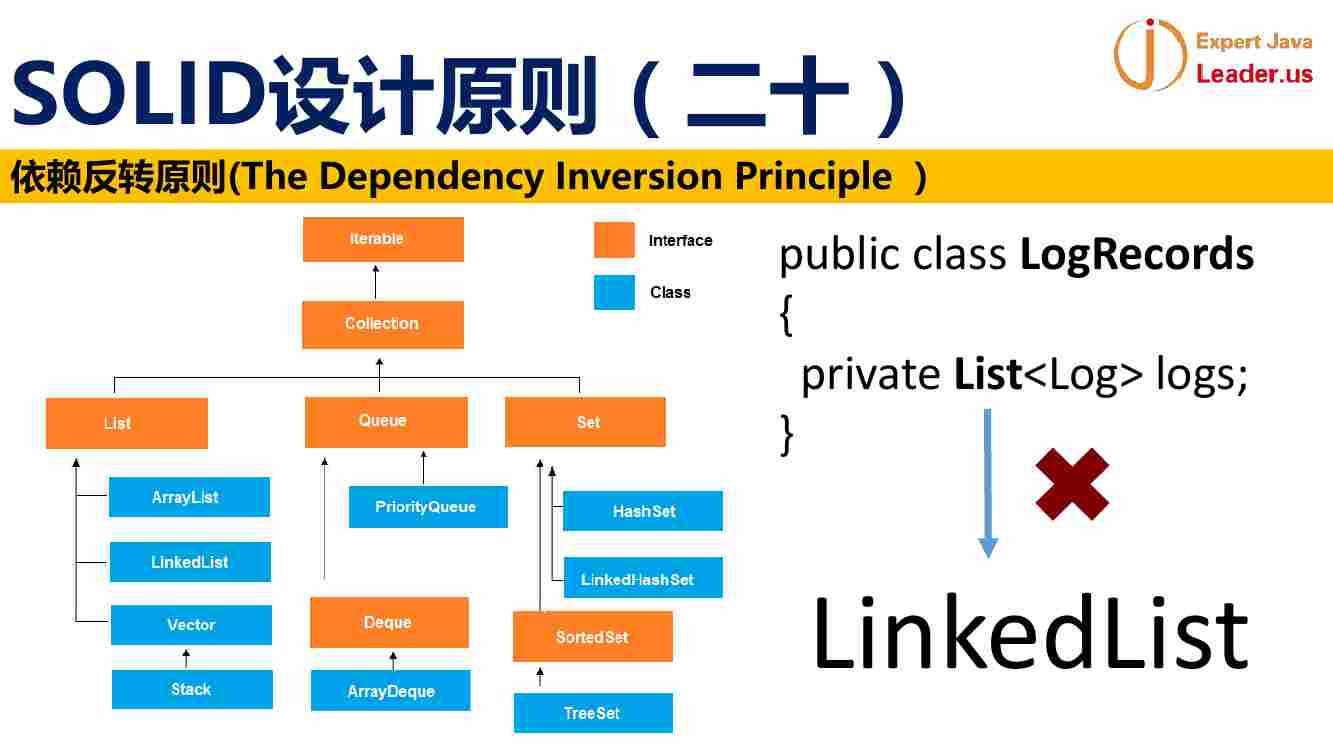
依赖的是高层次的 而不是低层次

客户端依赖的是高层的抽象

客户端具体采用哪种方式 是看不到的

另外某一个支付方式被取消 也是看不到的

也就是必须依赖抽象类或者接口 不依赖具体实现 这就是依赖反转 用的非常多



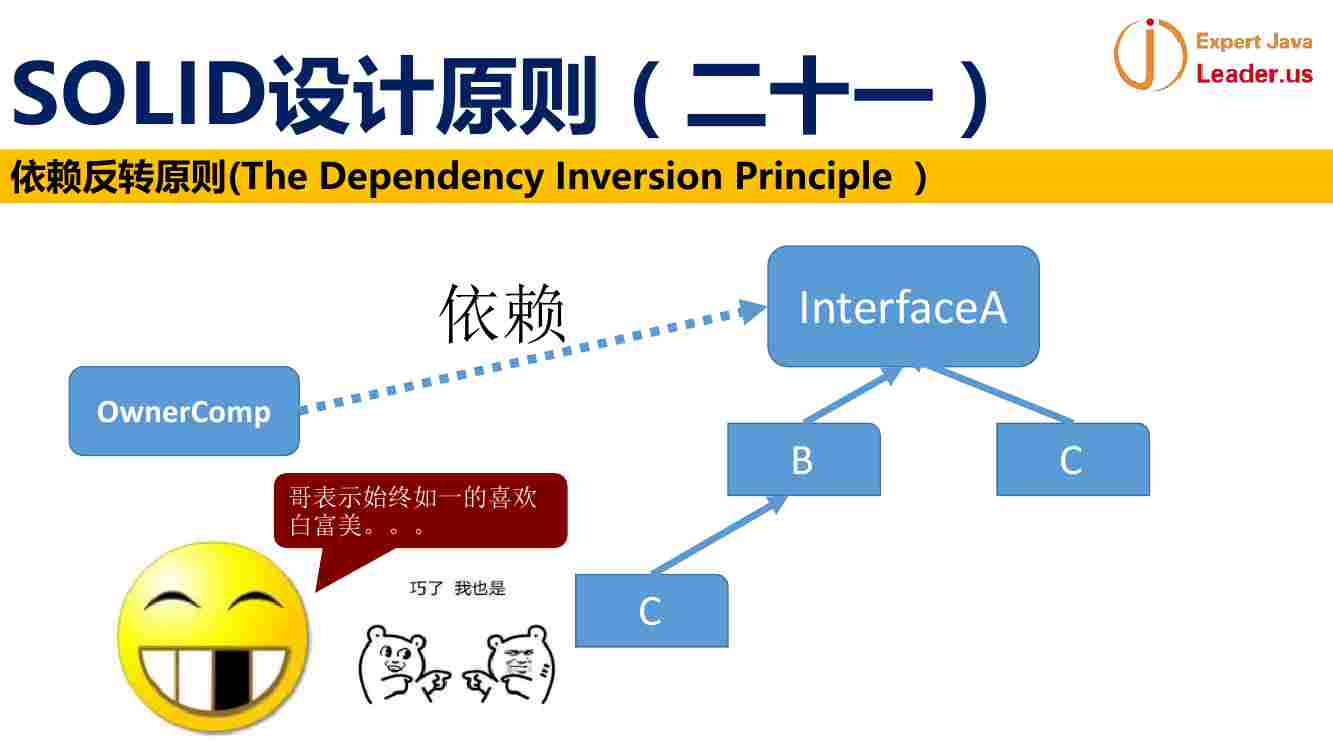
面向接口编程 Collection List ---- 这就是DI的一个准则

如果接口里面定义了LinkedList 之后 如果改成ArrayList 客户端就改了

所以 依赖了高层 客户端不用修改代码 ----- 这还是对开闭原则的进一步设计指导

也就是客户端要依赖抽象的 高层的 不要依赖底层的 ---- 这样一旦底层实现扩充了 修改了 不影响我的客户端 ---- 客户端的代码不修改 也就是对修改封闭

New出来是最底层的 但是 说出去 是我们要的是Collection 而不是ArrayList



依赖反转的方式

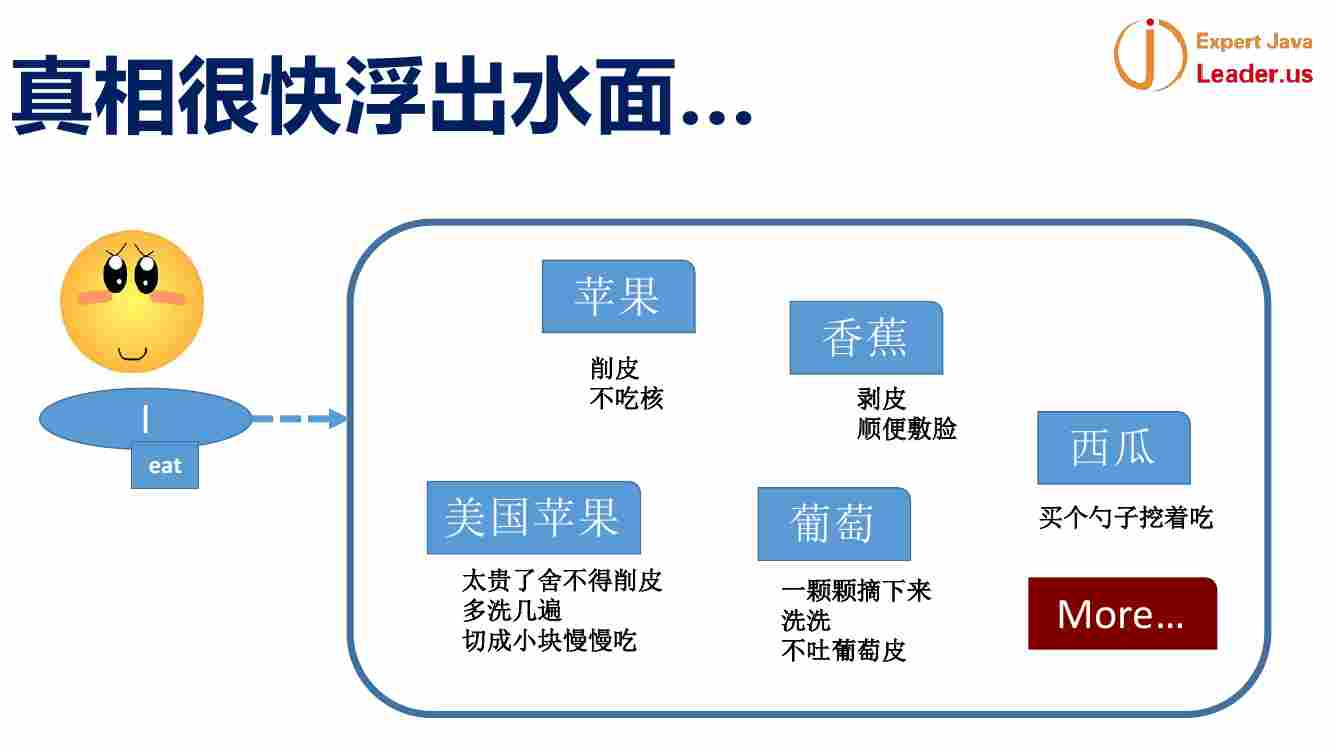
OwnerComp依赖的是高层的InterfaceA 生命的是依赖A 但是 最后注入的是具体的实现C B

也就是用的是底层的实现 但是 声明的是高层的抽象

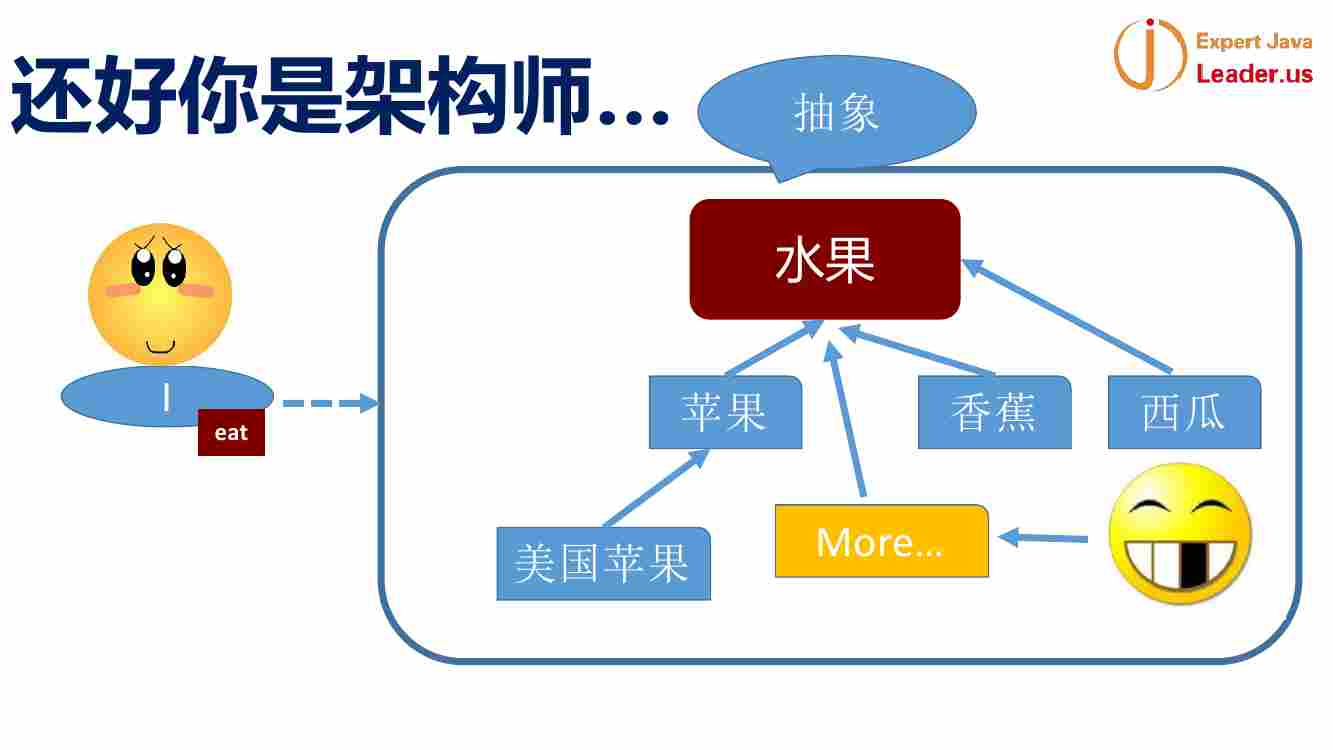




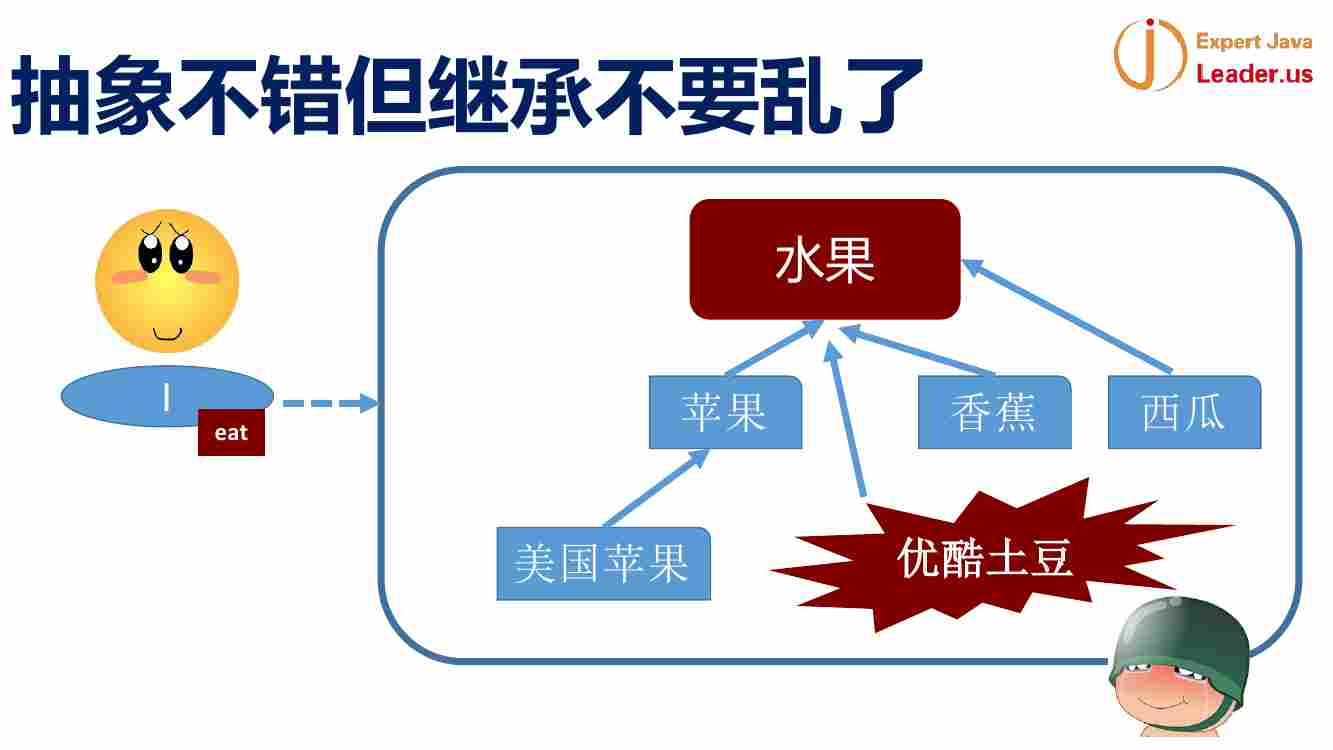
女朋友刚才是不太熟悉 吃的很少 仅仅吃一个苹果 削皮 不吃核 ----也就是开始的需求很少



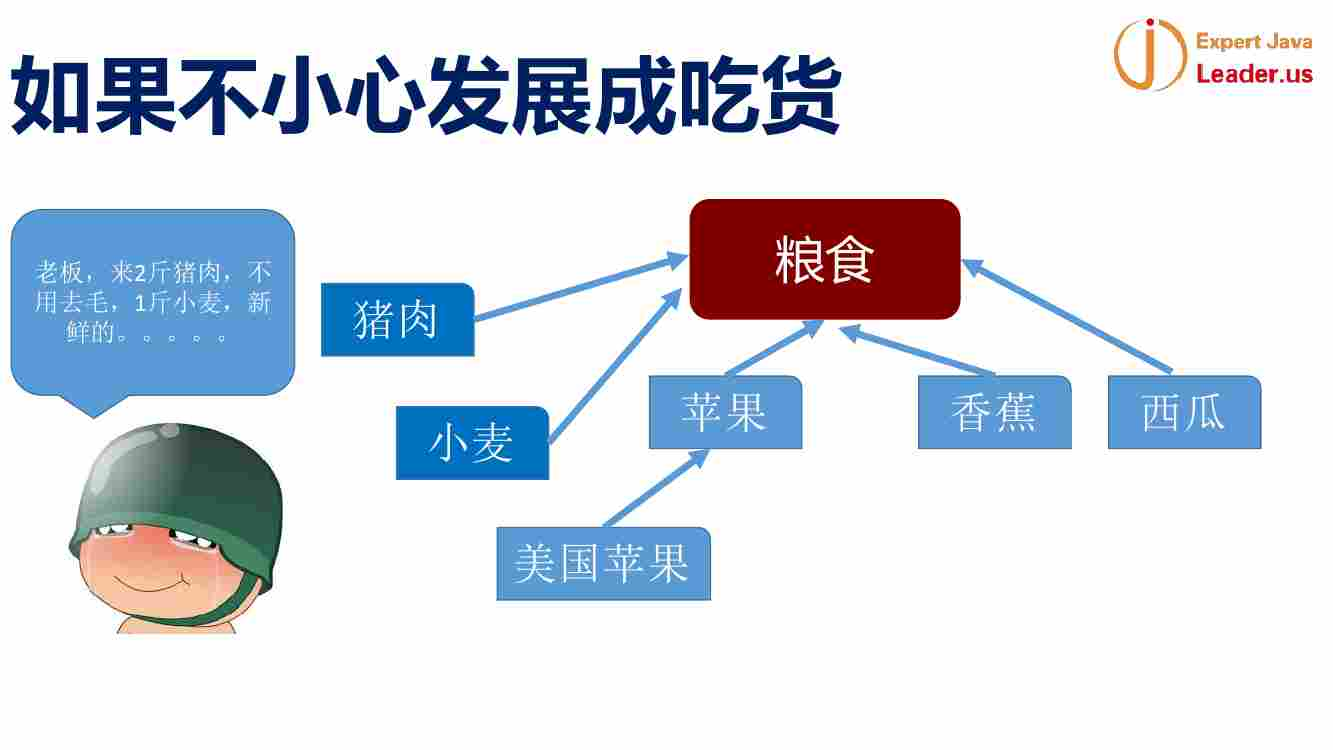
熟悉了 吃的越来越多 太多了 Client要写很多 很乱



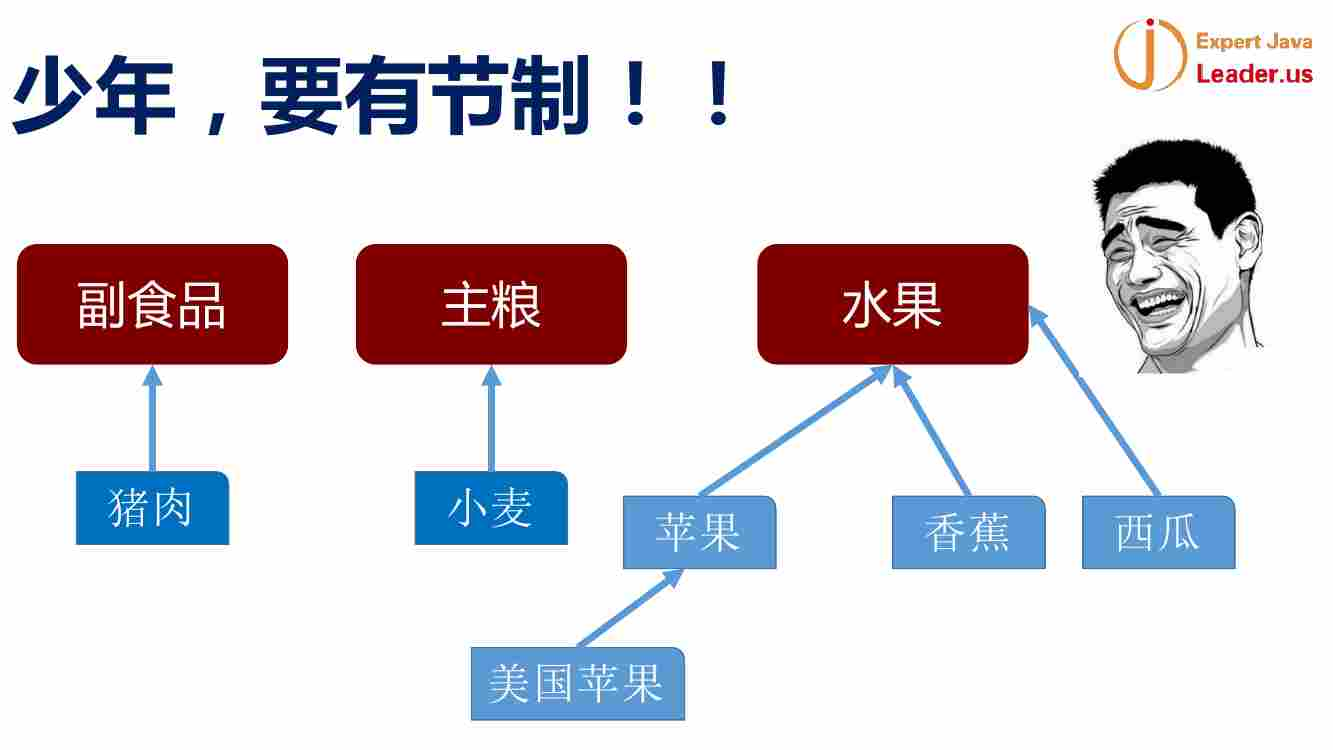
抽象出来一个水果 吃更多的没问题 还可以扩展 单一职责原则 和 开放封闭原则 来一个新的要吃的 就给出一个新的类



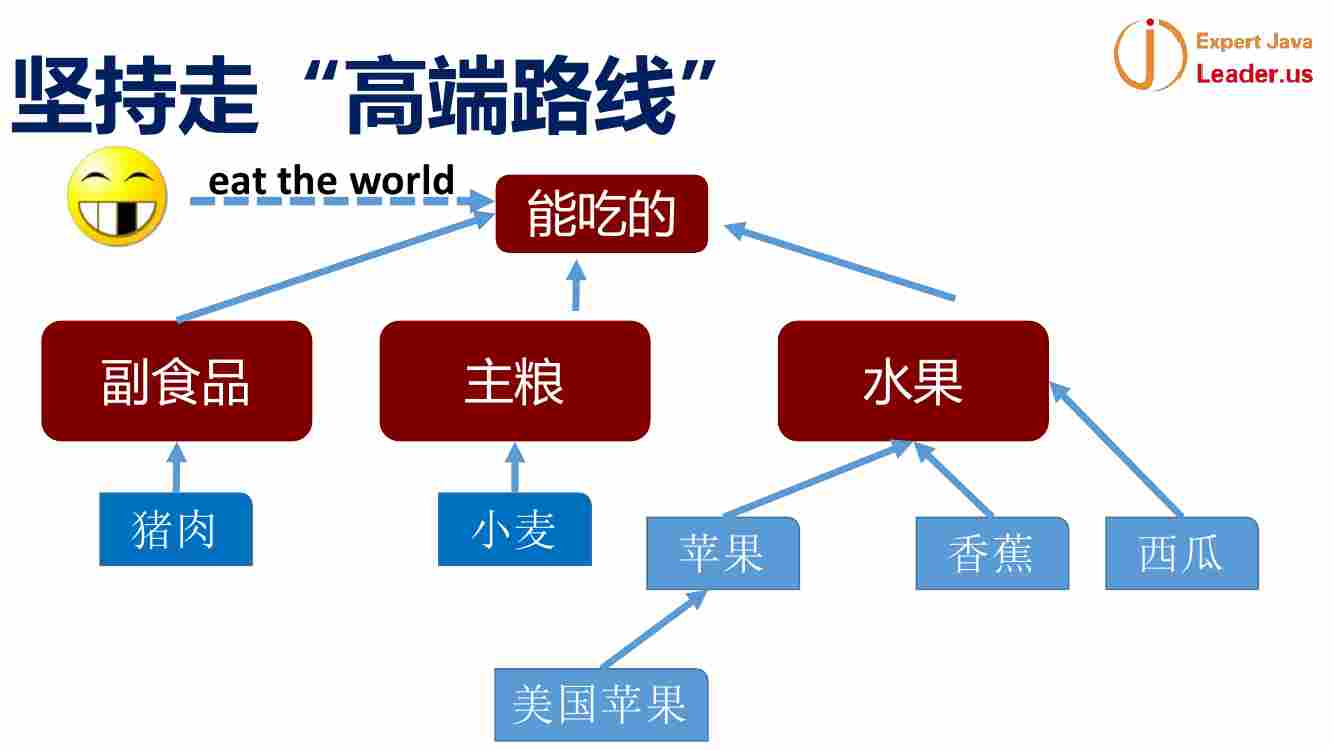
这个就是违反了里氏替换原则 ---- 子类已经变异过头了【继承不能违反里氏替换原则】



这个就是没有遵循接口隔离 一个大接口 ---- 接口隔离原则



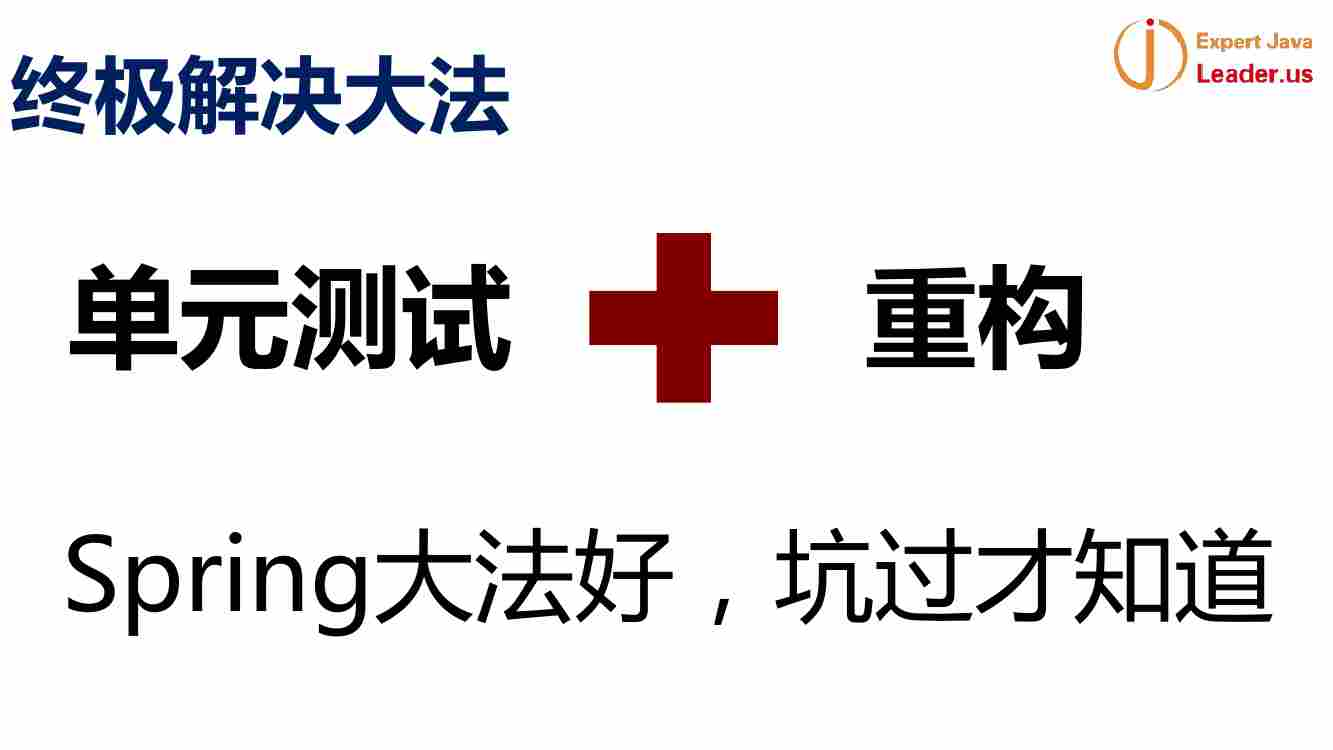
重新分大类 但是这样客户端要依赖更多的



统一到一个最高层

这个是依赖反转

把每一个设计模式都推推 遵循了什么 违反了什么



代码不断变化 如何支撑重构

Spring非常容易重构 非常容易进行单元测试 这就是Spring火起来的原因

