**对象**

对象是独立存在的客观事物，由属性和操作构成

属性是静态特征，操作是动态特征

属性一般只能通过执行对象的操作来改变

操作描述对象执行的功能

**消息和方法**

接收器接收消息，响应消息，执行方法

**类**

根据抽象的原则对客观事物进行归纳和划分，把具有相同特征的事物归为一个类。类是具有相同属性和相同操作的对象的集合

**继承**

类被组织成有单个根结点的树状结构，称为继承层次结构

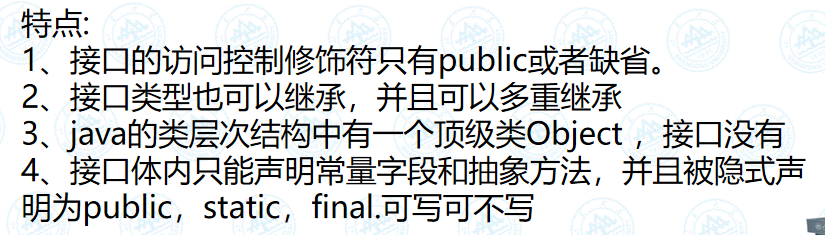
表达了对象一般与特殊的关系

**接口**

接口（interface）是向客户保证类的行为方式的一种协定（contract）。

当类实现某个接口时，相当于告诉客户：“我保证支持这个接口中提出的所有方法、属性、事件”

优点：可插拔，易扩展



**动/静态类型语言：**

动态类型语言(Dynamically Typed Language ):变量看作名称标识，类型和数值联系在一起

·类型的检查是在运行时做的;

·一般在变量使用之前不需要声明变量类型，而变量的类型通常是由被赋的值的类型决定。

·优点是方便阅读，不需要写非常多的类型相关的代码

·缺点是不方便调试，命名不规范时会造成读不懂，不利于理解等

·如php、Python和Ruby。

静态类型语言(Statically Typed Language ):类型和变量联系在一起

·类型的检查是在编译时做的.

·在编译时，便需要确定类型的语言。即写程序时需要明确声明变量类型。

·如C/C++、Java、C#等。

·优点在于其结构非常规范，便于调试，方便类型安全；

·缺点是为此需要写更多的类型相关代码，导致不便于阅读、不清晰明了

**继承的作用**

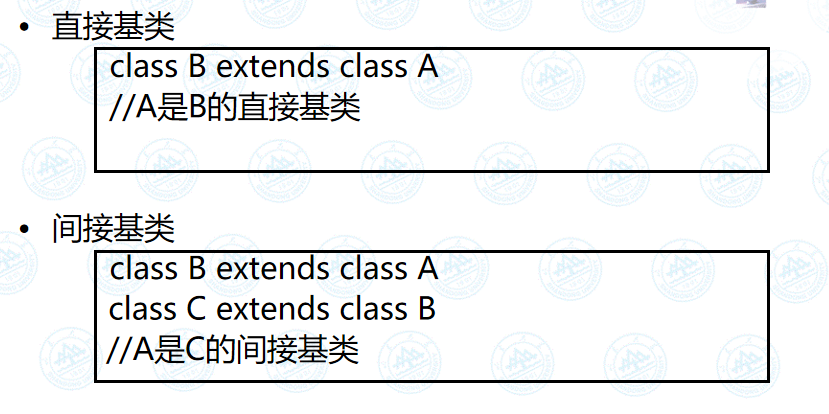
①使代码具有可重用性－通过继承类库拥有类库的能力

②可以重定义基类的成员函数，实现新的功能

③向派生类添加新成员，实现功能发展和扩展

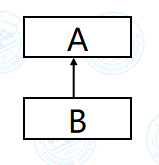
④不需要了解核心技术的细节，就能拥有别人的能力

**直接基类和间接基类**

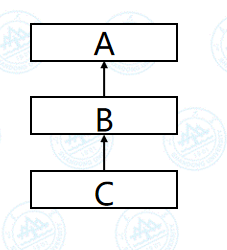


**继承的层次**

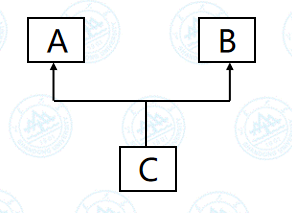
1. 单一继承：只拥有一个父类的新类的创建过程，从一个现有基类创建新类的过程



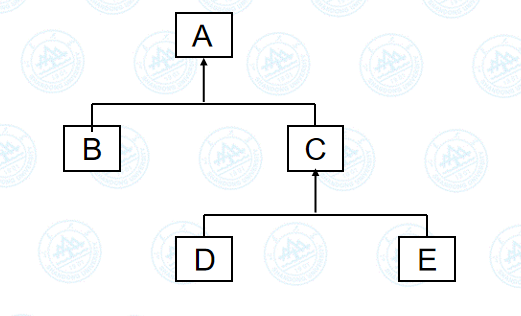
1. 多级继承



1. 多重继承

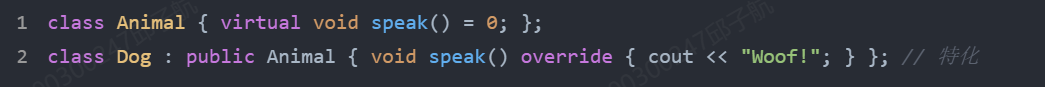


1. 层次继承

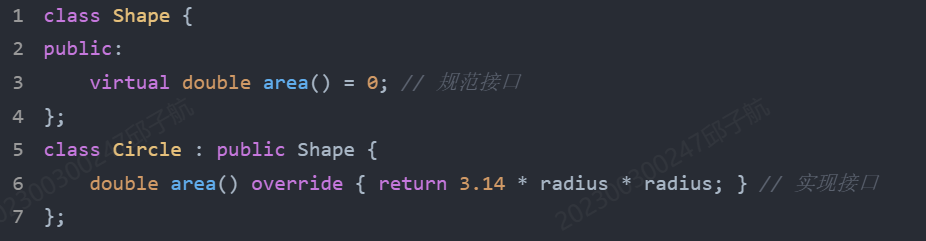


**继承的形式**

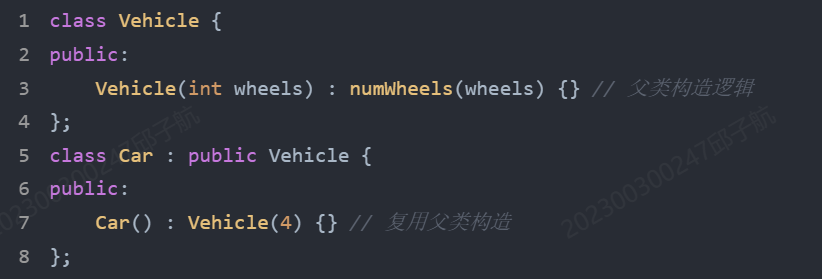
特殊化继承：从一个通用的父类派生出更具体的子类，子类扩展或重写父类的功能



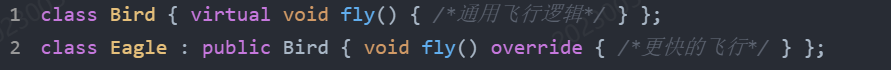
规范化继承：将类中不属于其核心职责的部分提取为独立的类或父类



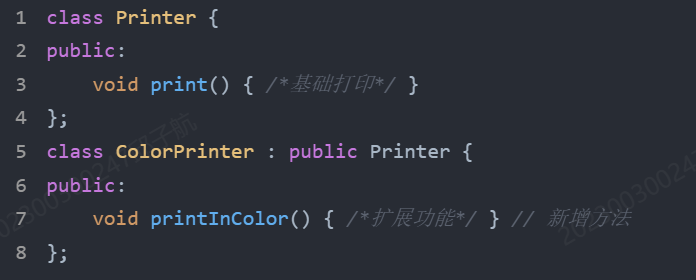
构造继承：子类通过继承复用父类的构造逻辑



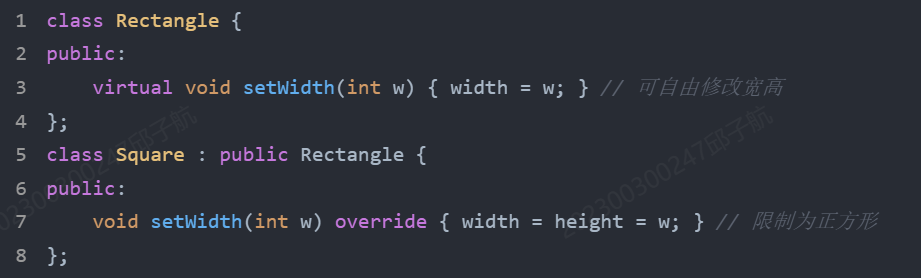
泛化继承：将多个具有相似属性和行为的子类抽象成一个共同的父类



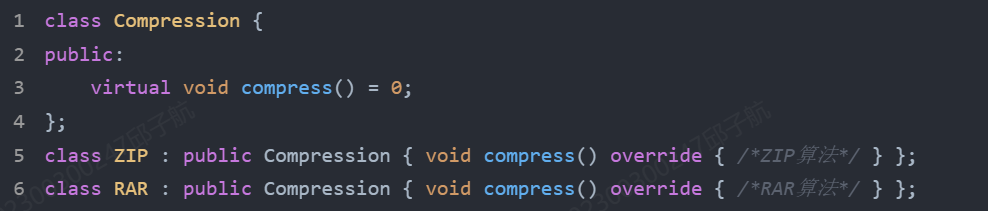
扩展继承：子类只是往父类中添加新行为，并不修改从父类继承来的任何属性



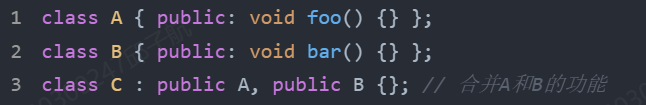
限制继承：子类约束父类的行为



变体继承：子类通过继承实现同一接口的不同变体



合并继承：子类同时继承多个父类



**重载、重写、重定义**

①重载Overload

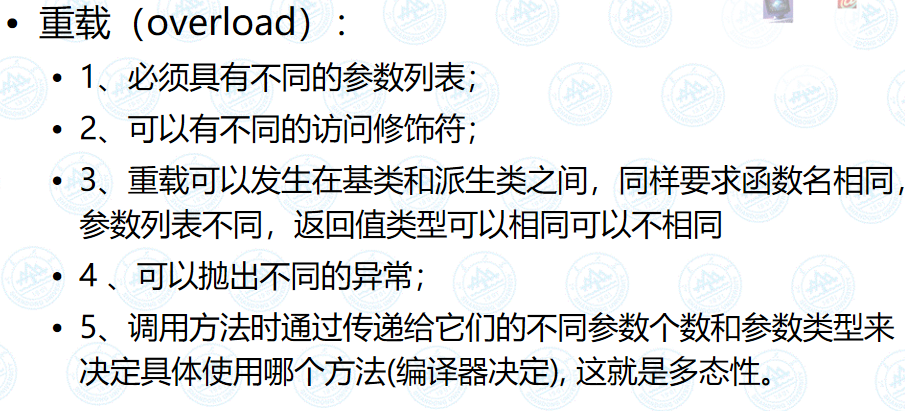
同一个类定义中有多个同名的方法，但有不同的形参，而且每个方法有不同的方法体，调用时根据形参的个数和类型来决定调用的是哪个方法。重载是编译时检查。

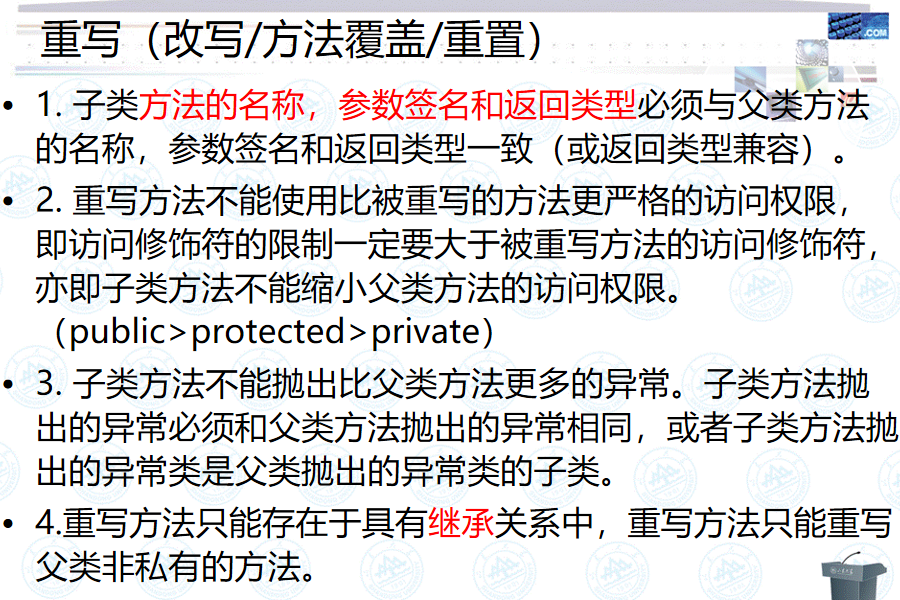
②重写Override

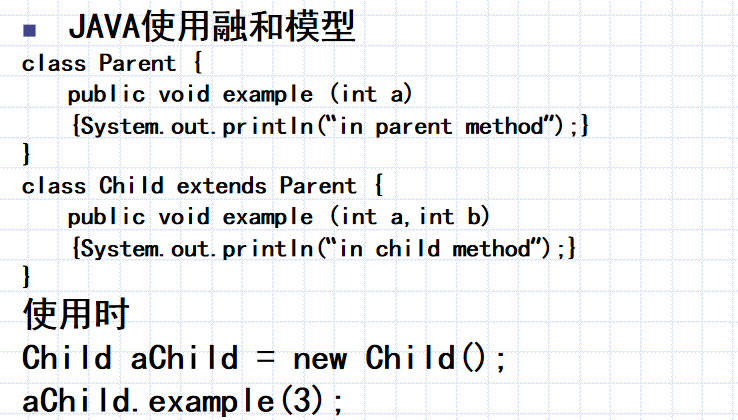
在继承情况下，子类中定义了与其基类中方法具有相同名称、相同类型签名（返回类型或兼容类型和相同参数）的方法，但重新编写了方法体。在运行时根据对象实际类型决定调用哪个方法。重置是运行时检查

③重定义Overwrite

子类定义了一个与父类同名但类型签名不同的方法，父类的同名方法被隐藏。编译时检查。调用时优先匹配子类方法，除非显式指定父类方法







**简述多态的四种形式**

**改写/覆盖/重写/重置/覆写：**

在继承情况下，子类中定义了与其基类中方法具有相同名称、相同类型签名（相同返回值类型或兼容类型和相同参数类型）的方法，但重新编写了方法体

**重载（Overload）：**

同一个类定义中有多个同名的方法，但有不同的形参，而且每个方法有不同的方法体，调用时根据形参的个数、顺序和类型来决定调用的是哪个方法

**多态变量：**

可以引用多种对象类型的变量

Parent variable=new Child();

**泛型：**

兼容多种数据类型的操作。即编写的代码可以在不同的数据类型上重用

public class Generic<T> { public T Field;}

**多态变量形式：**

简单变量

接收器变量

向下转型

纯多态

**纯多态**

**纯多态** 是指程序在**编译时**只依赖于**基类或接口的类型**，而在**运行时**根据对象的实际类型调用合适的方法（即动态绑定）（类似Adapter模式）



**强制、转换和造型：**

强制是一种隐式的类型转换，它发生在无需显式引用的程序中

转换表示程序员所进行的显式类型转换。在许多语言里这种转换操作称为“造型”

**向上转型向下转型：**

把子类对象直接赋给父类引用叫向上转型（upcasting），向上转型不用显式转型。

如Father father = new Son();

把指向子类对象的父类引用赋给子类引用叫向下转型（downcasting），要显式转型。

如father就是一个指向子类对象的父类引用，把father赋给子类引用son 即Son son =（Son）father；

静态：编译时绑定；动态：运行时绑定

动态：动态类和静态类的类型不一致

**静态方法绑定和动态方法绑定**

①静态方法绑定（静态联编）：在程序编译时进行的联编, 又称为早期联编; 因为这种联编是在程序运行之前完成的

②动态方法绑定（动态联编）：在程序运行时进行的联编，又称为晚期联编；响应消息时对哪个方法进行绑定是由接收器当前所包含的动态数值来决定的

静态绑定效率高，动态绑定灵活性高

**静态属性的特征：**

1. 属于类本身，而非对象实例
2. 可以通过类名访问
3. 生命周期长

**静态方法不能重写的原因：**

1. 静态方法属于类，而非实例
2. 静态方法绑定是编译期决定的
3. 语义上不符合重写的目的

**代替与改进**

代替（replacement）：在程序执行时，实现代替的方法完全覆盖父类的方法。即，当操作子类实例时，父类的代码完全不会执行。

改进(refinement)：实现改进的方法将继承自父类的方法的执行作为其行为的一部分。这样父类的行为得以保留且扩充。

**简述面向对象中可能会采用的内存分配方法**

**①最小静态空间分配：**只分配基类所需的存储空间。

**适用场景**：基类对象独立使用，不涉及派生类

**优点：**内存占用小，节省空间

**缺点：**可能无法容纳所有派生类对象

**②最大静态空间分配：**无论基类还是派生类，都分配可用于所有合法的数值的最大的存储空间。

**适用场景：**基类可能存储任何派生类对象

**优点：**兼容所有派生类，避免内存不足

**缺点：**浪费内存

**③动态内存分配：**只分配用于保存一个指针所需的存储空间。在运行时通过堆来分配数值所需的存储空间，同时将指针设为相应的合适值

**适用场景：**需要运行时灵活调整大小

**优点：**灵活调整大小，按需分配

**缺点：**手动管理，速度慢

#### **2.1 静态存储分配**

****静态存储分配****是指在编译时就能确定每个数据目标在运行时刻的存储空间需求,因而****在编译时就可以给他们分配固定的内存空间****。

#### **2.2 动态存储分配**

也被称为****栈式存储分配****，它是由一个类似于堆栈的运行栈来实现的。在栈式存储方案中，程序对数据区的需求在编译时是****完全未知的****，只有到运行的时候才能够知道，但是规定在运行中进入一个程序模块时，必须知道该程序模块所需的数据区大小才能够为其分配内存。

#### **2.3堆式存储分配**

堆式存储分配专门负责在编译时或运行时模块入口处都无法确定存储要求的数据结构的内存分配，比如可变长度串和对象实例。

**类的结合与分离**

分离：将系统功能分解到不同的类中，每个类专注于单一职责

结合：通过明确的关联关系（组合、关联、接口等）将这些分离的类连接起来，共同完成复杂功能

**复制和克隆**

**复制（copy）**：通常指创建一个对象的副本，可能是浅拷贝也可能是深拷贝，强调数据的复制过程。

手动 new Person(p.name)  
**克隆（clone）**：更强调"整个对象"的自我复制行为，往往与对象的生命周期和结构紧密相关，通常由对象自己完成。



**浅复制和深复制：**

浅复制(shallow copy)：共享实例变量。原有变量和复制产生的变量引用相同的变量值

深复制(deep copy)：建立实例变量的新的副本。

// 自定义拷贝构造函数（深复制）

DeepCopyExample(const DeepCopyExample& other) {

data = new int(\*other.data); // 分配新内存并拷贝值

}

//Main中：

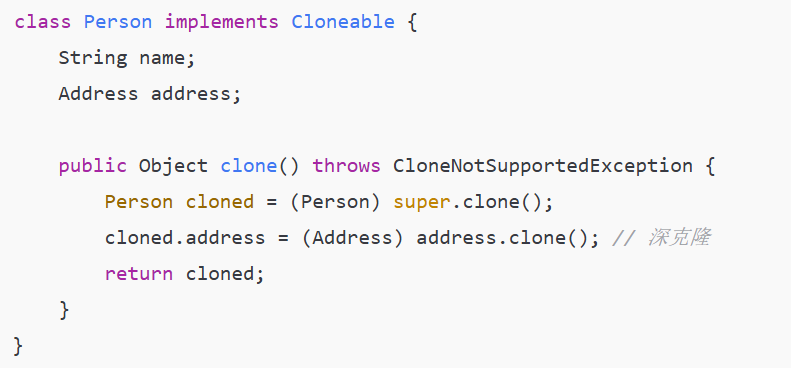
DeepCopyExample obj1(10);

DeepCopyExample obj2 = obj1;

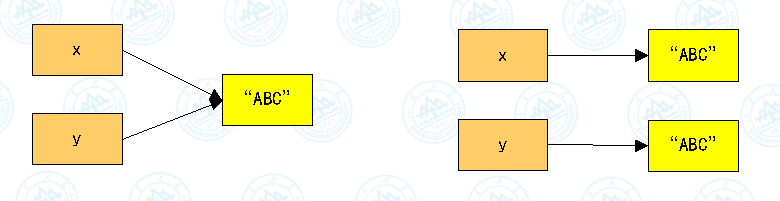
**浅克隆和深克隆**

浅克隆：只复制对象本身，对象内部的引用类型成员仍指向原先的对象

深克隆：不仅复制对象本身，还递归复制它内部所有引用的对象



**相同(equality)和同一( identity )：**



（地址相同和内容相同的区别）

**消除多重继承的二义性：**

1. 使用相应的类名进行标识

device.Device1::showPower();

1. 在派生类中重定义有名称冲突的成员

void showPower()

{

Device1::showPower();

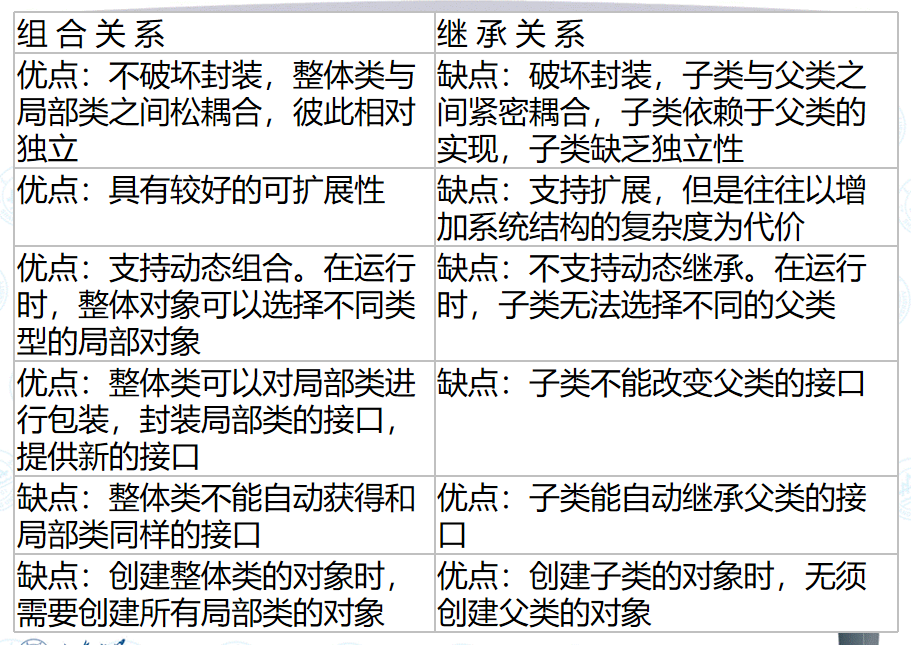
Device2::showPower();

}

面向对象系统中功能复用最常用的两种技术是组合和继承

**继承与组合的比较**

1. 继承描述的是一种一般性和特殊性的关系，使用继承可创建已存在类的特殊版本。
2. 组合描述的是一种组成关系，使用组合可用已存在的类组装新的类。



**白盒复用与黑盒复用**

**类继承**允许我们根据自己的实现来覆盖重写父类的实现细节，父类的实现对于子类是**可见**的，所以我们一般称之为白盒复用。

**对象**持有（其实就是组合）的整体类和部分类之间不会去关心各自的实现细节，即它们之间的实现细节是**不可见**的，故称为黑盒复用。

**反射和内省：**

**反射（Reflection）**：允许你在运行时检查、修改类的结构和行为（字段、方法、构造器等），甚至创建和调用它们。

**内省（Introspection）**：只允许你在运行时**检查（inspect）对象的信息**，但**不改变**或调用行为，主要用于读取属性、方法等信息。

**什么叫面向接口编程？在通过接口的形式调用对象时有什么优势？**

面向接口编程：不将变量声明为某个特定的具体类的实例对象，而让其遵从抽象类定义的接口。实现类仅实现接口，不添加方法。

优势：

①Client不必知道其使用对象的具体所属类。

②一个对象可以很容易地被(实现了相同接口的)的另一个对象所替换。

③对象间的连接不必硬绑定(hardwire)到一个具体类的对象上，因此增加了灵活性。

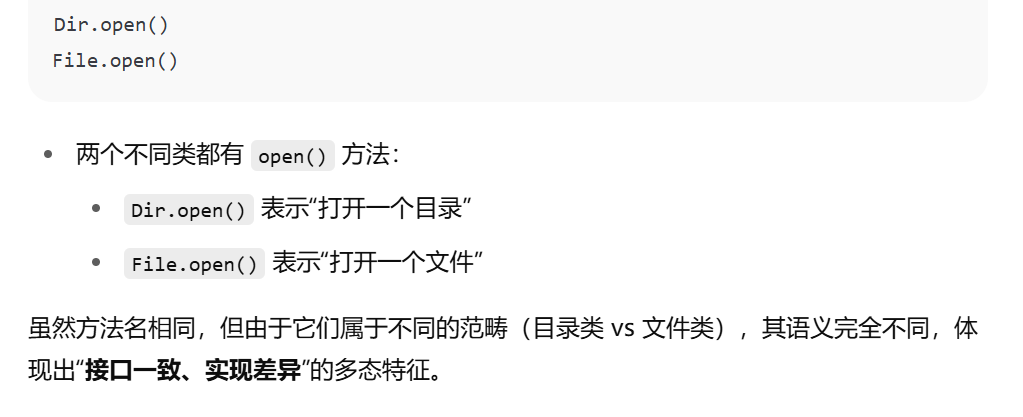
④松散耦合(loosens coupling)。

⑤增加了重用的可能性。

⑥提高了(对象)组合的机率，因为被包含对象可以是任何实现了一个指定接口的类。

**范畴与示例**

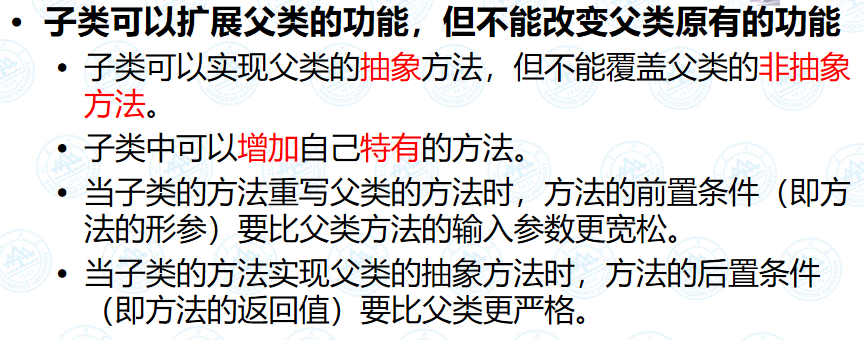
“范畴”可以理解为**一组具有某种语义或职责相似的类的集合。**在编程中，**相同的方法名在不同类中表示不同语义上的操作**，这就是一种“**语义范畴上的多态**”——即“**范畴多态**



**里氏替换原则LSP**

定义：所有引用基类的地方必须能透明地使用其子类的对象。子类可以扩展父类的功能，但不能改变父类原有的功能

类B继承类A时，除添加新的方法完成新增功能外，尽量不要重写父类A的方法，也尽量不要重载父类A的方法



**单例模式**

什么是单例模式：任何时间内只有一个类实例存在的模式

应用环境：需要一个从中进行全局访问和维护某种类型数据的区域的环境下使用单例模式

解决方案：为指示类是否已实例化的变量提供一个互斥锁，保证一个类只有一个实例存在

本质：实例共用同一块内存区域

**饿汉式和懒汉式单例模式**

**饿汉式**（Eager Initialization）类加载时就完成了实例化，避免了多线程同步问题。

优点：没有线程同步的开销；响应速度快。

缺点：提前占用资源。

**懒汉式**（Lazy Initialization）在第一次调用getInstance()方法时创建实例，并且会进行线程同步来确保只创建一个实例。

优点：延迟加载，只有在真正需要时才创建实例，节省资源。

缺点：线程安全需要额外处理；性能问题。

**工厂模式**

什么是工厂模式：定义一个用于创建对象的接口，让子类决定将哪一个类实例化

解决的问题：解决接口选择问题

应用环境：当一个类无法预测要创建哪种类的对象或一个类需要由子类来指定创建的对象时

解决方案：定义一个创建对性的接口，让子类来决定具体实例化哪一个类

本质：根据不同的情况选择不同的接口

**代理模式Proxy**

1) 远程代理（Remote  Proxy）为一个位于不同的地址空间的对象提供一个本地的代理对象。这个不同的地址空间可以是在同一台主机中，也可是在另一台主机中，远程代理又叫做大使(Ambassador)  
2) 虚拟代理（Virtual Proxy）：如果需要创建一个资源消耗较大的对象，先创建一个消耗相对较小的对象来表示，真实对象只在需要时才会被真正创建。   
3) 保护代理（Protection Proxy）控制对原始对象的访问。保护代理用于对象应该有不同的访问权限的时候。  
4) 智能指引（Smart Reference）取代了简单的指针，它在访问对象时执行一些附加操作。  
5) Copy-on-Write代理：它是虚拟代理的一种，把复制（克隆）操作延迟到只有在客户端真正需要时才执行。一般来说，对象的深克隆是一个开销较大的操作，Copy-on-Write代理可以让这个操作延迟，只有对象被用到的时候才被克隆。

**单一职责原则SRP：**

一个类只负责一项职责

优点：

提高代码可读性和可维护性。

降低修改风险（修改一个功能不会影响其他功能）。

**里氏替换原则LSP：**

子类能替换父类且不破坏程序正确性

优点：

保证继承关系的合理性。

避免运行时因类型替换导致的错误。

**依赖倒置原则DIP：**

高层模块不应依赖底层模块，抽象不应依赖细节

优点：

降低模块间的耦合。

提高代码可扩展性

**接口隔离原则ISP：**

客户端不应依赖不需要的接口方法

优点：

避免接口臃肿，减少无用方法的实现。

提高代码的灵活性。

**开闭原则OCP：**

软件实体应对扩展开放，对修改关闭

优点：

通过扩展（如继承、组合）增加功能，无需修改原有代码。

减少回归测试的风险。

**组合复用原则CRP：**

优先使用组合而非继承

优点：

避免继承带来的强耦合。

更灵活地动态替换组件。

**迪米特法则LOD：**

一个对象应尽可能少的和其它对象交互

优点：

降低类之间的耦合。

提高模块的独立性。

**什么是设计模式？设计模式的目标是什么？**

1. 设计模式是一套反复被使用，多数人知晓的，经过分类编写的代码设计经验的总结
2. 目标：为了可重用代码，让代码更容易被理解，保证代码可靠性

**设计模式**

单例模式：确保某个类只能创建一个实例，自行实例化并向整个系统提供这个实例

工厂模式：一个工厂类根据传入的参量决定创建出哪一种产品类的实例

适配器模式：将一个类的接口转换成客户希望的另一个接口，使得原本由于接口不兼容而不能一起的那些类可以一起工作

代理模式：创建占用大量内存或处理复杂的对象时，把创建这类对象推迟到使用它们的时候

桥接模式：将抽象部分和实现相分离，使它们都可以独立变化

装饰器模式：动态地给一个对象添加额外的职责

责任链模式：使请求的发送者和接受者解耦，请求沿责任链传递，直到有对象处理它

观察者模式：定义对象间的一种一对多的依赖关系，当一个对象的状态发生变化时，所有依赖于它的对象都得到通知并被自动更新

策略模式：对象的某个行为在不同场景有不同的实现方式

