怎么理解OO呢？

　　首先得搞明白Object的含义。所谓对象，在开发者眼里万物皆为对象。

　　比如早上起床：

　　睁开眼（Object:Eye）看见天（Object:Sky）亮了，那是光（Object:Light），抬头（Object:Head）望望窗（Object:Window）外，太阳（Object:Sun）出来了...

对象不仅能表示具体的事物，还能表示抽象的规则、计划或事件。

　　对象具有状态，一个对象用数据值来描述它的状态。

　　对象还有行为，用于改变对象的状态。

　　具有相同或相似性质的对象的抽象就是类。

　　类具有属性（Property），它是对象的状态的抽象，通过数据结构来描述类。

类具有方法（Method），它是对象的行为的抽象，通过方法名和实现该方法的操作来描述。

面向对象是把待解决的问题事务分解成各个对象（分而治之），建立对象的目的不是为了完成一个步骤，而是为了描叙某个事物在整个解决问题的步骤中的行为。

听起来很拗口，我们来看一个例子。

例如五子棋，面向过程的设计思路就是首先分析问题的步骤：1、开始游戏；2、黑子先走；3、绘制画面；4、判断输赢；5、轮到白子；6、绘制画面；7、判断输赢；8、返回步骤2；9、输出最后结果。大概想一想，数据结构 + 算法，五子棋的棋盘是15×15的布局，可以采用二维列表来表示棋盘的状态，假定黑子为1，白子为-1，通过获取以每个棋子为中心的横、纵、左斜、右斜四个方向的棋子颜色（1或-1），来判断是否满足五子连珠的条件。每个步骤之间环环相扣，耦合度太高，一旦出现错误，很难修改，例如列表下标的变化，黑白棋子代码冗余……

而面向对象的设计则是从另外的思路来解决问题。整个五子棋可以分为：1、黑白双方（棋子类）；2、棋盘系统（棋盘类）；3、规则系统（规则类）。第一类对象（棋子对象）仅负责接受用户输入，第二类对象（棋盘对象）仅负责在屏幕上面显示棋子布局的变化，第三类对象（规则对象）仅负责对棋局进行判定。顺便提一下，规则类如何实现？二维列表获取一条线上的数据，可以直接调用祁天轩老师的DoubleListHelper，如果用面向过程的思想就得改。连续5个值，相加为5则黑胜，-5则白胜。导入DoubleListHelper，无须知其内部逻辑，只用知道实现的功能，调用即可。

面向对象：将实现部分封装起来，仅为使用者提供调用接口，使得程序易于重用和维护，同时也加强了内部数据的保护。

可以明显地看出，面向对象是以功能来划分问题，而不是步骤。同样是绘制棋局，这样的行为在面向过程的设计中分散在了多个步骤中，很可能出现不同的绘制版本，因为通常设计人员会考虑到实际情况进行各种各样的简化。而面向对象的设计中，绘图只可能在棋盘对象中出现，从而保证了绘图的统一。

功能上的统一保证了面向对象设计的可扩展性。比如高手与新手下棋，黑白落子规则设定区别，如果要改动面向过程的设计，那么从输入到判断到显示这一连串的步骤都要改动，甚至步骤之间的循序都要进行大规模调整。如果是面向对象的话，黑白棋子分别继承棋子类，对棋子类的实例方法进行重写，但棋盘类只需调用父类对象（隔离客户端代码与功能的实现方式），体现多态性。同时整个对象功能的调用顺序都没有变化，改动只是局部的。

再比如要把这个五子棋游戏改为围棋游戏，如果是面向过程设计，那么五子棋的规则就分布在了程序的每一个角落，要改动还不如重写。但是如果采用面向对象的设计，那么只用改动棋盘对象（15 🡪 19）和规则对象（连线 🡪 吃子）就可以了，五子棋和围棋的区别不就是这些吗？而下棋的大致步骤从面向对象的角度来看没有任何变化。

面向对象的三大特征分别是：封装、继承、多态，这三者是面向对象编程的基本要素

**封装(Encapsulation)**

通过对象隐藏程序的具体实现细节，将数据与操作包装在一起，对象与对象之间通过消息传递机制实现互相通信（方法调用），具体的表现就是通过提供访问接口实现消息的传入传出。

封装的意义：由于封装隐藏了具体的实现，如果实现的改变或升级对于使用方而言是无感知的，提高程序的可维护性；而且封装鼓励程序员把特定数据与对数据操作的功能打包在一起，有利于应用程序的去耦。

**继承(Inheritance)**

通过继承得到的类称为子类，被继承的类为父类，子类相对于父类更加具体化。

子类具有自己特有的属性和方法，并且子类使用父类的方法也可以覆盖(重写)父类方法。

继承的意义：隔离客户端与后台代码。

**多态(Polymorphism)**

多态发生在运行期间，即子类型多态，指的是子类型是一种多态的形式，不同类型的对象实体有统一接口，相同的消息给予不同的对象会引发不同的动作。

多态的意义：提供了编程的灵活性，简化了类层次结构外部的代码，使编程更加注重关注点分离(Separation of concerns，SoC)

**设计原则**

**类的单一职责（Single responsibility principle）**

一个类或则一个模块应当只有一种职责，其提供的服务应该与其责任保持一致，如果存在多种责任则应考虑对其拆分。

**开闭原则（Open/closed principle, OCP, Open for extension, close for modification）**

软件中的对象（类，模块，函数等等）应该对于扩展是开放的，但是对于修改是封闭的

开闭原则主要思想就是对于扩展的包含，对于修改的限制，新增功能的同时避免修改已有的实现，尽量做到对外提供的功能不变

**里氏替换原则（Liskov substitution principle）**

程序中的对象应该是可以在不改变程序正确性的前提下被它的子类所替换的

里氏替换原则认为子类的功能应该可以完全替换父类并且不会影响程序的正确性，简单理解就是子类在继承父类的同时不能改变父类已有的功能，加上开闭原则子类只能对父类进行扩展而不能对父类的功能进行修改。

**组合复用原则（Composite Reuse Principle, CRP）**

如果仅仅为了代码复用优先选择组合复用，而非继承复用。

组合的耦合性相对继承低。

**依赖倒置原则（Dependency inversion principle, DIP）**

1.高层次的模块不应该依赖于低层次的模块，两者都应该依赖于抽象接口</br>

2.抽象接口不应该依赖于具体实现。而具体实现则应该依赖于抽象接口

依赖反转原则是指一种特定的解耦（传统的依赖关系创建在高层次上，而具体的策略设置则应用在低层次的模块上）形式，使得高层次的模块不依赖于低层次的模块的实现细节，依赖关系被颠倒（反转），从而使得低层次模块依赖于高层次模块的需求抽象。

**迪米特法则（Law of Demeter，LoD）**

一个对象应当对其他对象尽可能少的了解。不和陌生人说话。