**策略模式属于对象的行为模式。其用意是针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。策略模式使得算法可以在不影响到客户端的情况下发生变化。**

策略模式的结构

　　策略模式是**对算法的包装**，是把**使用算法的责任和算法本身分割开来，委派给不同的对象管理**（各算法自成一类（**具体策略(ConcreteStrategy)角色**），实现抽象算法接口（**抽象策略(Strategy)角色**），再提供一个构造器（**环境(Context)角色**），供外部调用）。策略模式通常把一个系列的算法包装到一系列的策略类里面，作为一个抽象策略类的子类。用一句话来说，就是：“**准备一组算法，并将每一个算法封装起来，使得它们可以互换**”。下面就以一个示意性的实现讲解策略模式实例的结构。

　　这个模式涉及到三个角色：

　　●　　**环境(Context)角色：**持有一个Strategy的引用。

　　●　　**抽象策略(Strategy)角色：**这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需的接口。

　　●　　**具体策略(ConcreteStrategy)角色：**包装了相关的算法或行为。

# 认识策略模式

**策略模式的重心**

　　策略模式的重心不是如何实现算法，而是如何组织、调用这些算法，从而让程序结构更灵活，具有更好的维护性和扩展性。

**算法的平等性**

　　策略模式一个很大的特点就是各个策略算法的平等性。对于一系列具体的策略算法，大家的地位是完全一样的，正因为这个平等性，才能实现算法之间可以相互替换。所有的策略算法在实现上也是相互独立的，相互之间是没有依赖的。

　　所以可以这样描述这一系列策略算法：策略算法是相同行为的不同实现。

**运行时策略的唯一性**

运行期间，**策略模式在每一个时刻只能使用一个具体的策略实现对象**，虽然可以动态地在不同的策略实现中切换，但是同时只能使用一个。

**公有的行为**

经常见到的是，所有的具体策略类都有一些公有的行为。这时候，就应当把这些**公有的行为放到共同的抽象策略角色Strategy类里面**（比如算法都是要先降9折的情况下，再各自减去他们的算法。那么可以打9折在Strategy抽象类实现）。当然这时候抽象策略角色必须要用Java抽象类实现，而不能使用接口。

　　这其实也是典型的将代码向继承等级结构的上方集中的标准做法。

# 策略模式的优点

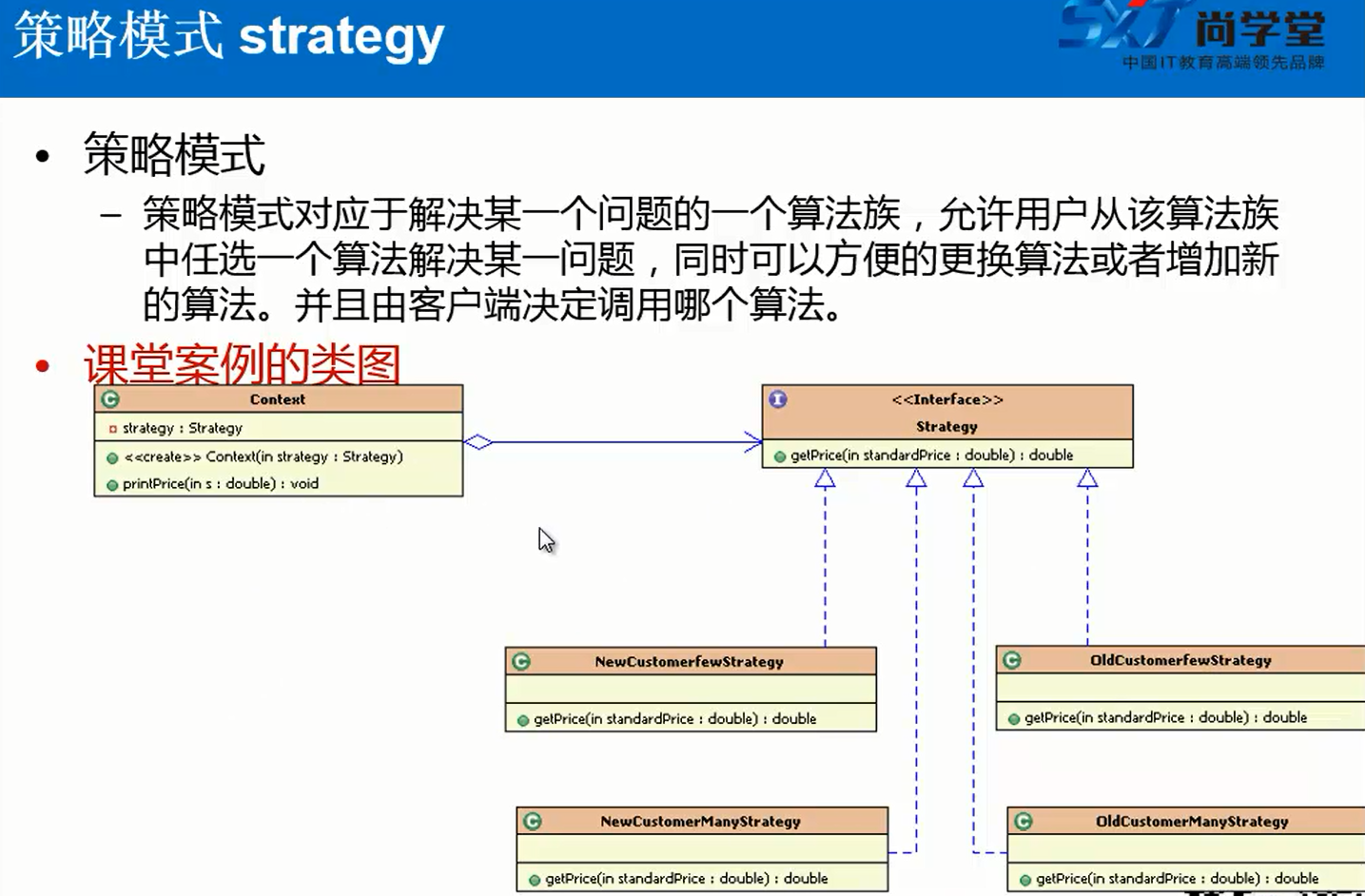
　　（1）策略模式提供了管理相关的算法族的办法。策略类的等级结构定义了一个算法或行为族。恰当使用继承可以把公共的代码移到父类里面，从而避免代码重复。

　　（2）使用策略模式可以避免使用多重条件(if-else)语句。多重条件语句不易维护，它把采取哪一种算法或采取哪一种行为的逻辑与算法或行为的逻辑混合在一起，统统列在一个多重条件语句里面，比使用继承的办法还要原始和落后。

# 策略模式的缺点

　　（1）客户端必须知道所有的策略类，并自行决定使用哪一个策略类。这就意味着客户端必须理解这些算法的区别，以便适时选择恰当的算法类。换言之，**策略模式只适用于客户端知道算法或行为的情况。**

　　（2）由于策略模式把每个具体的策略实现都单独封装成为类，如果备选的策略很多的话，那么对象的数目就会很可观。



实现代码如下：

package design.strategy;  
  
public interface Strategy {  
    double getPrice(double price);  
}  
package design.strategy;  
  
public class OldUser implements Strategy {  
  
    @Override  
    public double getPrice(double price) {  
        System.out.println("老用户打8折");  
//        System.out.println(price \* 0.8 + "元");  
        return price\*0.8;  
    }  
}  
package design.strategy;  
  
public class NewUser implements Strategy {  
  
    @Override  
    public double getPrice(double price) {  
        System.out.println("新用户不打折");  
//        System.out.println(price + "元");  
        return price;  
    }  
}  
package design.strategy;  
  
public class Context {  
    private Strategy strategy;  
  
    public Context() {  
    }  
  
    public Context(Strategy strategy) {  
        this.strategy = strategy;  
    }  
  
    public void setStrategy(Strategy strategy) {  
        this.strategy = strategy;  
    }  
  
    public void getPrice(double price) {  
        System.out.println(strategy.getPrice(price));  
    }  
}  
package design.strategy;  
  
import javax.print.DocFlavor;  
  
public class text {  
  
    public static void main(String[] args) {  
  
        Context context = new Context( );  
        context.setStrategy(new OldUser());  
  
        context.getPrice(100);  
    }  
  
  
}