**设计模式简介**

**一、本节目标**

我们本节课程主要介绍【设计模式】的相关知识，将会涉及以下内容：

* 什么是设计模式
* 设计模式原则
* 设计模式分类
* 常用设计模式

**二、什么是设计模式**

在软件工程中，【设计模式】是对软件设计中普遍存在的各种问题，所提出的 **解决方案**。

换句话说，设计模式是一套被反复使用、多数人知晓的、经过分类的、代码设计的 **经验的总结**。使用设计模式是为了可重用代码，让代码更容易被他人理解，保证代码可靠性。

**三、 设计模式原则**

**1、开闭原则（Open Close Principle）**

开闭原则的意思是：对扩展开放，对修改封闭。在程序需要进行扩展的时候，不能去修改或影响原有的代码，实现一个热插拔的效果。简言之，是为了使程序的扩展性更好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类。

**2、里氏替换原则（Liskov Substitution Principle）**

里氏替换原则是面向对象设计的基本原则之一。 里氏替换原则中说，任何基类可以出现的地方，子类一定可以出现。里氏替换原则是继承复用的基石，只有当子类可以替换掉基类，且软件单位的功能不受到影响时，基类才能真正被复用，而且子类也能够在基类的基础上增加新的行为。里氏代换原则是对开闭原则的补充。实现开闭原则的关键步骤就是抽象化，而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

**3、依赖倒置原则（Dependence Inversion Principle）**

这个原则是开闭原则的基础，核心内容：针对接口编程，高层模块不应该依赖底层模块，二者都应该依赖抽象。

**4、接口隔离原则（Interface Segregation Principle）**

这个原则的意思是：使用多个隔离的接口，比使用单个庞大的接口要好。其目的在于降低耦合度。由此可见，其实设计模式就是从大型软件架构出发，便于升级和维护的软件设计思想。它强调低依赖、低耦合。

**5、单一职责原则（Single Responsibility Principle）**

一个实体应尽量少地与其他实体之间发生相互作用，应该使得系统功能模块相对独立。

可能有的人会觉得单一职责原则和前面的接口隔离原则很相似，其实不然。其一，单一职责原则原注重的是职责；而接口隔离原则注重对接口依赖的隔离。其二，单一职责原则主要约束的是类，其次才是接口和方法，它针对的是程序中的实现和细节；而接口隔离原则主要约束接口，主要针对抽象，针对程序整体框架的构建。

**6、最少知识原则（Demeter Principle）**

一个对象应该对其他对象保持最少的了解。类与类之间的关系越密切，耦合度越大，当一个类发生改变时，对另一个类的影响也越大。如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用。如果其中一个类需要调用另一个类的某一个方法的话，可以通过第三者转发这个调用。所以在类的设计上，每一个类都应当尽量降低成员的访问权限。

**7、合成复用原则（Composite Reuse Principle）**

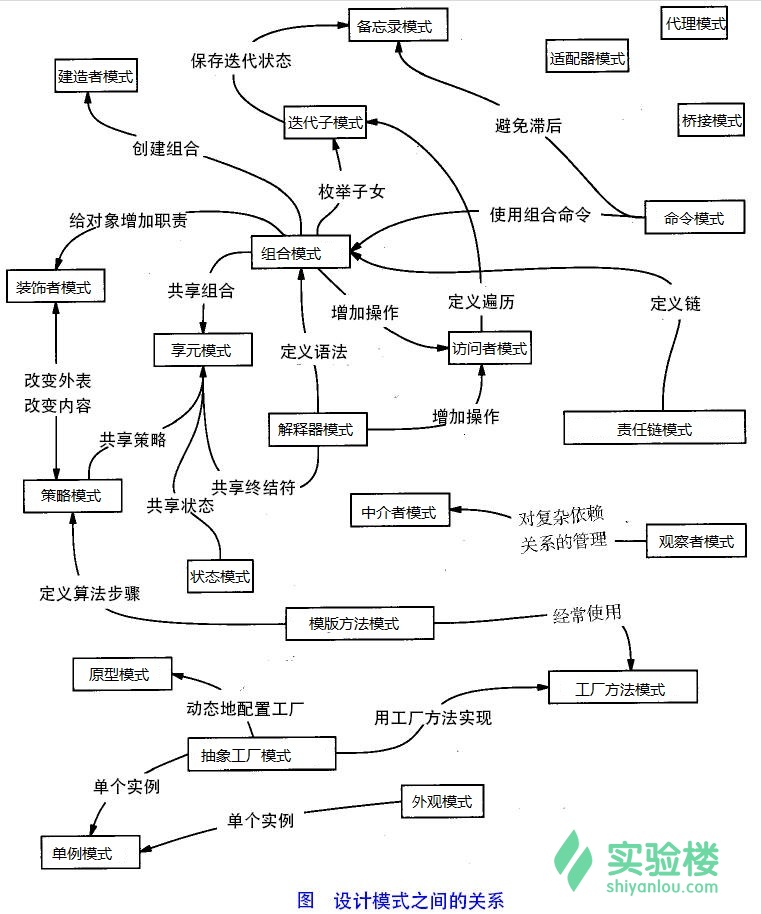
合成复用原则就是在一个新的对象里通过关联关系（组合关系、聚合关系）来使用一些已有的对象，使之成为新对象的一部分；新对象通过委派调用已有对象的方法达到复用功能的目的。简而言之，尽量使用 组合/聚合 的方式，而不是使用继承。

**四、 设计模式分类**

通常来说设计模式分为三大类：

* **创建型模式** ：工厂模式、抽象工厂模式、单例模式、建造者模式、原型模式。
* **结构型模式** ：适配器模式、装饰者模式、代理模式、外观模式、桥接模式、组合模式、享元模式。
* **行为型模式** ：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

下面用图片来整体描述一下设计模式之间的关系：



**五、常用设计模式**

在以后的课程中，我们会逐步讲解常用的设计模式，包括 工厂模式、单例模式、适配器模式、观察者模式等等。

**六、小结**

本次课程介绍了【设计模式】相关知识，为后续的各种设计模式课程打下基础。此外，我非常推荐大家读一本关于设计模式的书，[《Head First 设计模式》](http://book.douban.com/subject/2243615/)，书中的讲解非常有趣，清晰易懂。

**七、参考文档**

* [设计模式简介](http://www.w3cschool.cc/design-pattern/design-pattern-intro.html)
* [设计模式 (计算机)](http://zh.wikipedia.org/zh/%E8%AE%BE%E8%AE%A1%E6%A8%A1%E5%BC%8F_(%E8%AE%A1%E7%AE%97%E6%9C%BA))

**工厂模式**

**一、本节目标**

我们本节课程主要介绍【工厂模式】的相关知识，将会涉及以下内容：

* 什么是工厂模式
* 工厂模式类图
* 工厂模式代码
* 工厂模式应用
* PS：学习 Java 设计模式之前最好有一定的 Java 语言基础。由于讲这种东西其实比较枯燥，为了让大家学习起来轻松愉快一点，我会尽量使用一些生动、形象和幽默的例子。

**二、什么是工厂模式**

工厂模式（Factory Pattern）的意义就跟它的名字一样，在面向对象程序设计中，工厂通常是一个用来创建其他对象的对象。工厂模式根据不同的参数来实现不同的分配方案和创建对象。例如用工厂来创建 人 这个对象，如果我们需要一个男人对象，工厂就会为我们创建一个男人；如果我们需要一个女人，工厂就会为我们生产一个女人（女娲娘娘的既视感啊）。

工厂模式通常分为：

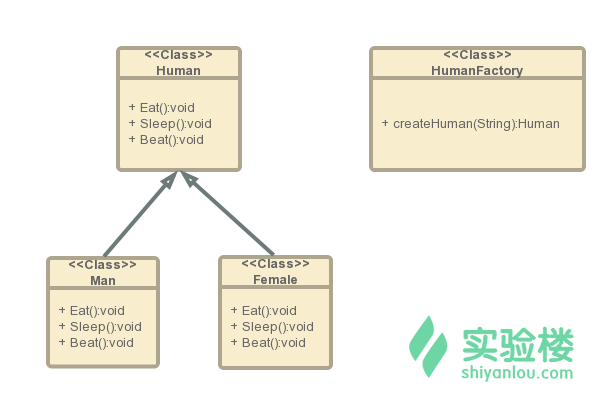
* 简单工厂模式
* 工厂方法模式
* 抽象工厂模式

本节课讲解 简单工厂模式、工厂方法模式，抽象工厂模式将会在下节课中讲到。

**三、工厂模式类图**

刚刚我们说到，用工厂模式来创建人。现在我们来当回 “女娲娘娘”，先创建一个男人，他每天都 “吃饭、睡觉、打豆豆”，然后我们再创建一个女人，她每天也 “吃饭、睡觉、打豆豆”。

我们以简单工厂模式为例。



**四、 工厂模式示例代码**

Talk is cheap, show me the code.

// 抽象产品

abstract class Human {

public abstract void eat();

public abstract void sleep();

public abstract void beat();

}

// 具体产品-男人

class Man extends Human{

public void eat() {

System.out.println("Man can eat.");

}

public void sleep() {

System.out.println("Man can sleep.");

}

public void beat() {

System.out.println("Man can beat doudou.");

}

}

// 具体产品-女人

class Female extends Human{

public void eat() {

System.out.println("Female can eat.");

}

public void sleep() {

System.out.println("Female can sleep.");

}

public void beat() {

System.out.println("Female can beat doudou.");

}

}

// 简单工厂

public class HumanFactory {

public static Human createHuman(String gender){

Human human = null;

if( gender.equals("man") ){

human = new Man();

}else if( gender.equals("female")){

human = new Female();

}

return human;

}

}

// 女娲造人

public class Goddess {

public static void main(String[] args) throws IOException {

// 造个男人

Human human = HumanFactory.createHuman("man");

human.eat();

human.sleep();

human.beat();

}

}

简单工厂模式就是上面那样子了，那么工厂方法模式又有什么不同呢？工厂方法模式在简单工厂模式的基础之上，把简单工厂抽象化了。

// 抽象产品

abstract class Human {

public abstract void eat();

public abstract void sleep();

public abstract void beat();

}

// 具体产品-男人

class Man extends Human {

public void eat() {

System.out.println("Man can eat.");

}

public void sleep() {

System.out.println("Man can sleep.");

}

public void beat() {

System.out.println("Man can beat doudou.");

}

}

// 具体产品-女人

class Female extends Human{

public void eat() {

System.out.println("Female can eat.");

}

public void sleep() {

System.out.println("Female can sleep.");

}

public void beat() {

System.out.println("Female can beat doudou.");

}

}

// 简单工厂变为了抽象工厂

abstract class HumanFactory {

public abstract Human createHuman(String gender) throws IOException;

}

// 具体工厂（每个具体工厂负责一个具体产品）

class ManFactory extends HumanFactory{

public Human createHuman(String gender) throws IOException {

return new Man();

}

}

class FemaleFactory extends HumanFactory{

public Human createHuman(String gender) throws IOException {

return new Female();

}

}

// 女娲造人

public class Goddess {

public static void main(String[] args) throws IOException {

// 造个男人

HumanFactory hf = new ManFactory();

Human h = hf.createHuman("man");

h.eat();

h.sleep();

h.beat();

}

}

**五、工厂模式应用**

你可能就问了，工厂模式工厂模式的，我咋没见过哪儿用过啊？这这这儿，在 Java 库里面。根据不同的参数，getInstance() 方法会返回不同的 Calendar 对象。

java.util.Calendar - getInstance()

java.util.Calendar - getInstance(TimeZone zone)

java.util.Calendar - getInstance(Locale aLocale)

java.util.Calendar - getInstance(TimeZone zone, Locale aLocale)

java.text.NumberFormat - getInstance()

java.text.NumberFormat - getInstance(Locale inLocale)

写 Android 的童鞋们，应该还知道 BitmapFactory , 这也是工厂模式的一种应用。

**六、实验总结**

总结一下，本次课程我们站在 “女娲娘娘” 的角度，学习了工厂模式的相关知识。

**七、参考文档**

* [工厂方法](http://zh.wikipedia.org/zh/%E5%B7%A5%E5%8E%82%E6%96%B9%E6%B3%95)
* [Java Design Pattern: Factory](http://www.programcreek.com/2013/02/java-design-pattern-factory/)

**抽象工厂模式**

**一、本节目标**

我们本节课程主要介绍【抽象工厂模式】的相关知识，将会涉及以下内容：

* 什么是抽象工厂模式
* 抽象工厂模式类图
* 抽象工厂模式代码
* 抽象工厂模式应用
* 工厂方法模式、抽象工厂模式区别

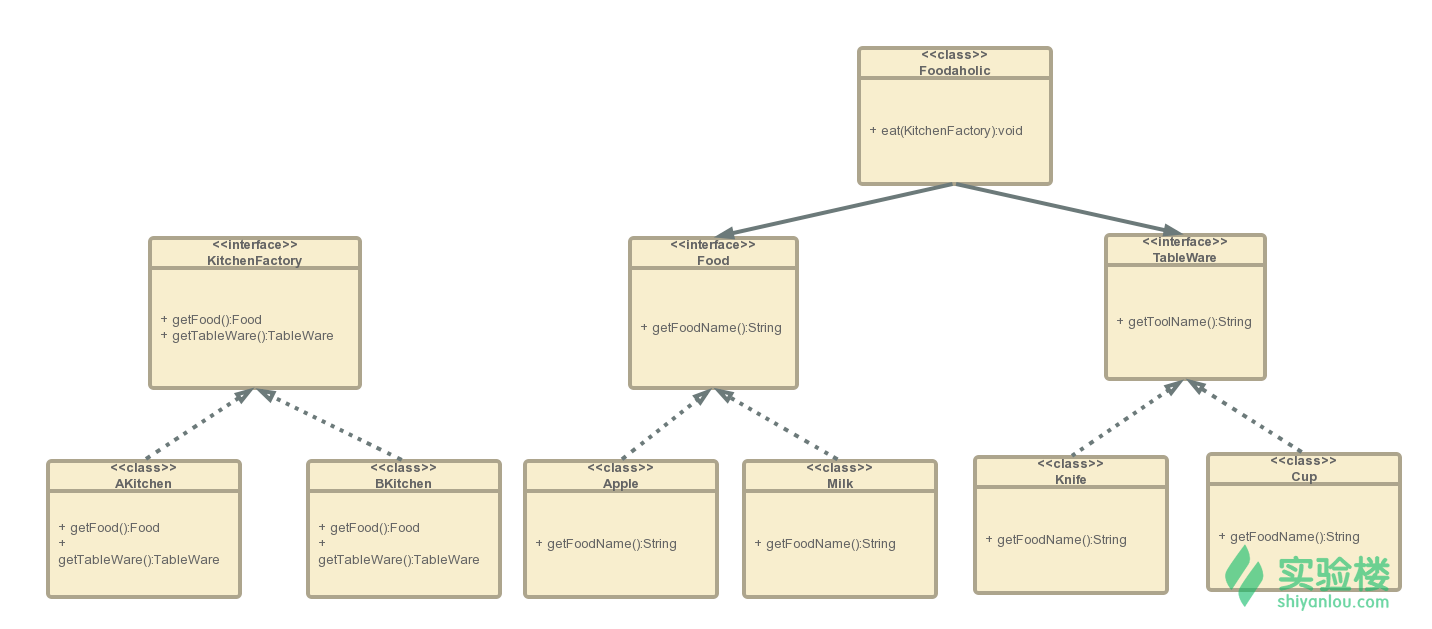
**二、什么是 抽象工厂模式**

抽象工厂模式（Abstract Factory Pattern）是一种软件开发设计模式。抽象工厂模式提供了一种方式，可以将一组具有同一主题的单独的工厂封装起来。如果比较抽象工厂模式和工厂模式，我们不难发现前者只是在工厂模式之上增加了一层抽象的概念。抽象工厂是一个父类工厂，可以创建其它工厂类。所以我们也叫它 “工厂的工厂”。（想想上节课的 “女娲娘娘”，这简直就是 “女娲娘娘的亲娘” 啊...）

**三、 抽象工厂模式类图**

“女娲娘娘”只有一个，而我们的工厂却可以有多个，因此在这里用作例子就不合适了。作为“女娲娘娘”生产出来的男人女人们，那就让我们来当一次吃货吧。（吃的东西总可以任性多来一点...）

现在，假设我们有 A、B 两个厨房。每个厨房拥有的餐具和食品都不一样，但是用户搭配使用的方式，比如刀子和苹果、杯子和牛奶等等，我们假设是一致的。



**四、 抽象工厂模式示例代码**

Talk is cheap, show me the code.

// 抽象工厂

public interface KitchenFactory {

public Food getFood();

public TableWare getTableWare();

}

// 抽象食物

public interface Food {

public String getFoodName();

}

// 抽象餐具

public interface TableWare {

public String getToolName();

}

// 以具体工厂 AKitchen 为例

public class AKitchen implements KitchenFactory {

public Food getFood() {

return new Apple();

}

public TableWare getTableWare() {

return new Knife();

}

}

// 具体食物 Apple 的定义如下

public class Apple implements Food{

public String getFoodName() {

return "apple";

}

}

// 具体餐具 Knife 的定义如下

public class Knife implements TableWare {

public String getToolName() {

return "knife";

}

}

// 吃货要开吃了

public class Foodaholic {

public void eat(KitchenFactory k) {

System.out.println("A foodaholic is eating "+ k.getFood().getFoodName()

+ " with " + k.getTableWare().getToolName() );

}

public static void main(String[] args) {

Foodaholic fh = new Foodaholic();

KitchenFactory kf = new AKitchen();

fh.eat(kf);

}

}

**五、抽象工厂模式应用**

抽象工厂模式特别适合于这样的一种产品结构：产品分为几个系列，在每个系列中，产品的布局都是类似的，在一个系列中某个位置的产品，在另一个系列中一定有一个对应的产品。这样的产品结构是存在的，这几个系列中同一位置的产品可能是互斥的，它们是针对不同客户的解决方案，每个客户都只选择其一。

**六、工厂方法模式、抽象工厂模式区别**

工厂方法模式、抽象工厂模式，傻傻分不清楚...

为了解释得更清楚，先介绍两个概念：

* **产品等级结构**：比如一个抽象类是食物，其子类有苹果、牛奶等等，则抽象食物与具体食物名称之间构成了一个产品等级结构。食物是抽象的父类，而具体的食物名称是其子类。
* **产品族**：在抽象工厂模式中，产品族是指由同一个工厂生产的，位于不同产品等级结构中的一组产品。如 AKitchen 生产的苹果、刀子，苹果属于食物产品等级结构中，而刀子则属于餐具产品等级结构中。

因此工厂方法模式、抽象工厂模式最大的区别在于：

工厂方法模式：针对的是 **一个产品等级结构**。

抽象工厂模式：针对 **多个产品等级结构**。

吃货们，懂了吧？

**七、小结**

国际惯例，总个结。本次课程我们做了一回吃货，学习了【抽象工厂模式】的相关知识。

**八、参考文档**

* [Java Design Pattern: Abstract Factory](http://www.programcreek.com/2013/02/java-design-pattern-abstract-factory/)
* [抽象工厂模式解析例子](http://tianli.blog.51cto.com/190322/32545)

**适配器模式**

**一、本节目标**

我们本节课程主要介绍【适配器模式】的相关知识，将会涉及以下内容：

* 什么是适配器模式
* 适配器模式类图
* 适配器模式代码
* 适配器模式的应用

**二、什么是适配器模式**

顾名思义，适配器模式（Adapter Pattern）当然是用来适配的啦。当你想使用一个已有的类，但是这个类的接口跟你的又不一样，不能拿来直接用，这个时候你就需要一个适配器来帮你了。

这就好像你兴冲冲地跑去香港，买了个港版的 iPhone6，充电器插头拿回家一看，不能用啊。这时候你多么需要买一个转接头适配器...

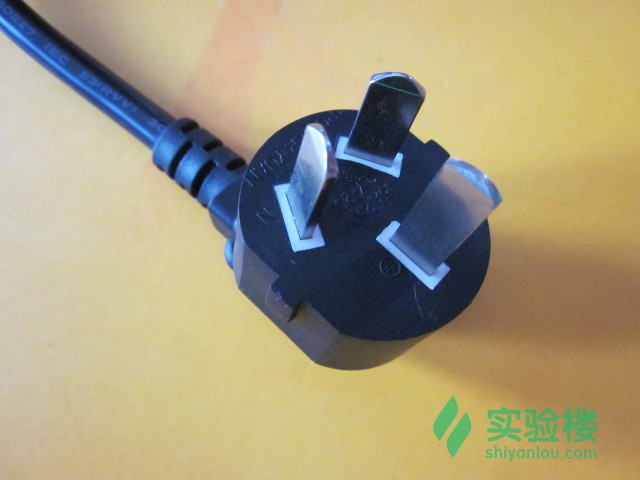


**三、适配器模式类图**

你去香港旅游，买的 iPhone6 的充电器插头是英标的，它是那种三脚是方形的插头。



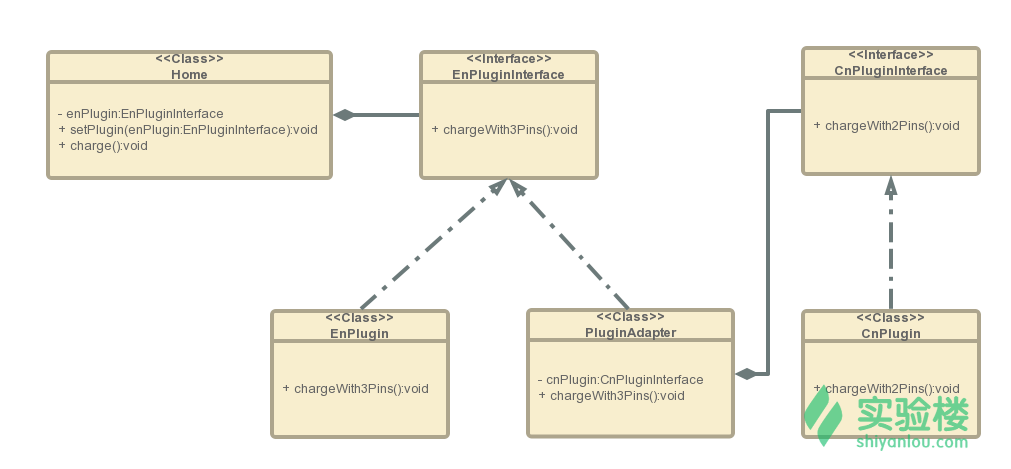
而咱们国标的插头是两只脚，即使是三只脚的插头也和英标不一样。



为了方便，这里我们就假设国标插头就只是两只脚的插头吧。（其实我是懒。。）



好的，目标明确，英标三只脚插头充电，国标两只脚插头充电。你家很富，有很多插座可以充电。



**四、 适配器模式示例代码**

Talk is cheap, show me the code.

// 英标接口

public interface EnPluginInterface {

void chargeWith3Pins();

}

// 实现英标接口的充电方法

public class EnPlugin implements EnPluginInterface {

public void chargeWith3Pins() {

System.out.println("charge with EnPlugin")

}

}

// 在室内充电

public class Home {

private EnPluginInterface enPlugin;

public Home() { }

public Home(EnPluginInterface enPlugin) {

this.enPlugin = enPlugin;

}

public void setPlugin(EnPluginInterface enPlugin) {

this.enPlugin = enPlugin;

}

// 充电

public void charge() {

enPlugin.chargeWith3Pins();

}

}

// 英标测试类

public class Test {

public static void main(String[] args) {

EnPluginInterface enPlugin = new EnPlugin();

Home home = new Home(enPlugin);

// 会输出 “charge with EnPlugin”

home.charge();

}

}

// 现在你回内陆了，只能用国标充电

// 国标接口

public interface CnPluginInterface {

void chargeWith2Pins();

}

// 实现国标接口的充电方法

public class CnPlugin implements CnPluginInterface {

public void chargeWith2Pins() {

System.out.println("charge with CnPlugin");

}

}

// 适配器

public class PluginAdapter implements EnPluginInterface {

private CnPluginInterface cnPlugin;

public PluginAdapter(CnPluginInterface cnPlugin) {

this.cnPlugin = cnPlugin;

}

// 这是重点，适配器实现了英标的接口，然后重载英标的充电方法为国标的方法

@Override

public void chargeWith3Pins() {

cnPlugin.chargeWith2Pins();

}

}

// 适配器测试类

public class AdapterTest {

public static void main(String[] args) {

CnPluginInterface cnPlugin = new CnPlugin();

Home home = new Home();

PluginAdapter pluginAdapter = new PluginAdapter(cnPlugin);

home.setPlugin(pluginAdapter);

// 会输出 “charge with CnPlugin”

home.charge();

}

}

**五、适配器模式的应用**

前面已经说了，当你想使用一个已有的类，但是这个类的接口跟你的又不一样，不能拿来直接用，这个时候你就需要一个适配器来帮你了，其主要作用就是在旧的接口、新的接口之间完成适配。

比如说，做过 Android 的同学肯定写 ListView 的适配器都写吐了吧...

**六、小结**

国际惯例，总个结。本次课程我们去香港买了个 iPhone6 回家... 顺便学习了【适配器模式】的相关知识。

**七、参考文档**

* [一个示例让你明白适配器模式](http://blog.csdn.net/zhangjg_blog/article/details/18735243)

**装饰者模式**

**一、本节目标**

我们本节课程主要介绍【装饰者模式】的相关知识，将会涉及以下内容：

* 什么是装饰者模式
* 装饰者模式类图
* 装饰者模式代码
* 装饰者模式应用
* 装饰者模式、适配器模式区别
* PS：剧情预告，本节课程会十分酸爽...

**二、什么是 装饰者模式**

装饰者模式（Decorator Pattern，有的也用 Wrapper Pattern）就是动态地把职责附加到已有对象上去，实现功能扩展。这种特性，使得装饰者模式提供了比继承更具有弹性的解决方案。

提到装饰者模式，总让我想起卞之琳的《断章》来：

你站在桥上看风景，

看风景的人在楼上看你。

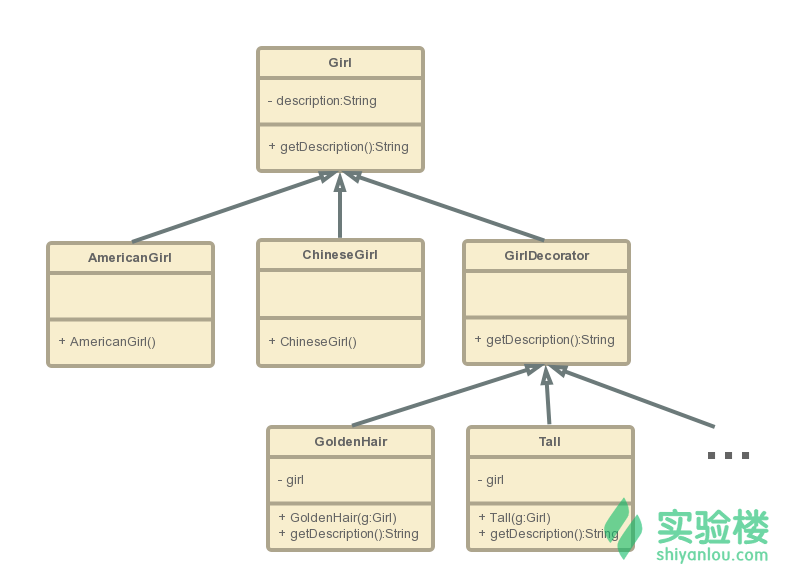
明月装饰了你的窗子，

你装饰了别人的梦。

多么有意境啊，单身狗读完会不会觉得心好累... 没关系，我来拯救你！

**三、 装饰者模式类图**

送你一个女朋友怎么样！想她是美国金发大妞？浪漫的法国女郎？国产的萌萌哒妹子？OK，没问题！你想她是哪个国家的就是哪个国家的。她们有不同的爱好或者习惯，每一个这样的女孩，都可以看作是一个 Java 类。我知道此刻你一定在想，这一个、那一个...那岂不是有很多类？这种方式没有扩展性，每当有一个新类型的女孩，就得又新建一个类，这简直就是类爆炸啊！



**四、 装饰者模式示例代码**

Talk is cheap, show me the code.

// 抽象类 Girl

public abstract class Girl {

String description = "no particular";

public String getDescription(){

return description;

}

}

// 美国女孩

public class AmericanGirl extends Girl {

public AmericanGirl() {

description = "+AmericanGirl";

}

}

// 国产妹子

public class ChineseGirl extends Girl {

public ChineseGirl() {

description = "+ChineseGirl";

}

}

// 装饰者

public abstract class GirlDecorator extends Girl {

public abstract String getDescription();

}

// 下面以美国女孩示例

// 给美国女孩加上金发

public class GoldenHair extends GirlDecorator {

private Girl girl;

public GoldenHair(Girl g) {

girl = g;

}

@Override

public String getDescription() {

return girl.getDescription() + "+with golden hair";

}

}

// 加上身材高大的特性

public class Tall extends GirlDecorator {

private Girl girl;

public Tall(Girl g) {

girl = g;

}

@Override

public String getDescription() {

return girl.getDescription() + "+is very tall";

}

}

// 检验一下

public class Main {

public static void main(String[] args) {

Girl g1 = new AmericanGirl();

System.out.println(g1.getDescription());

GoldenHair g2 = new GoldenHair(g1);

System.out.println(g2.getDescription());

Tall g3 = new Tall(g2);

System.out.println(g3.getDescription());

// 你也可以一步到位

// Girl g = new Tall(new GoldenHair(new AmericanGirl()));

}

}

最后美国金发大妞奉上（输出内容为）：

+AmericanGirl

+AmericanGirl+with golden hair

+AmericanGirl+with golden hair+is very tall

**五、装饰者模式应用**

当你需要动态地给一个对象添加功能，实现功能扩展的时候，就可以使用装饰者模式。

Java IO 类中有一个经典的装饰者模式应用， BufferedReader 装饰了 InputStreamReader.

BufferedReader input = new BufferedReader(new InputStreamReader(System.in));

* **InputStreamReader(InputStream in)** - InputSteamReader 读取 bytes 字节内容，然后转换成 characters 流 输出。
* **BufferedReader(Reader in)** - 从 characters 流 中读取内容并缓存。

**六、装饰者模式、适配器模式区别**

* 1.关于新职责：适配器也可以在转换时增加新的职责，但其主要目的并不在此；而装饰者模式主要目的，就是给被装饰者增加新职责用的。
* 2.关于原接口：适配器模式是用新接口来调用原接口，原接口对新系统来说是不可见或者说不可用的；而装饰者模式原封不动的使用原接口，系统对装饰的对象也通过原接口来完成使用。
* 3.关于其包裹的对象：适配器是知道被适配者的详细情况的（就是那个类或那个接口）；而装饰者只知道其接口是什么，至于其具体类型（是基类还是其他派生类）只有在运行期间才知道。

**七、小结**

国际惯例，总个结。本次课程我们赚大了，各种女朋友...顺便学习了【装饰者模式】的相关知识。

**八、参考文档**

* [Java Design Pattern: Decorator](http://www.programcreek.com/2012/05/java-design-pattern-decorator-decorate-your-girlfriend/)
* [装饰者模式（Decorator Pattern）](http://blog.csdn.net/lcl_data/article/details/8830455)

**观察者模式**

**一、本节目标**

我们本节课程主要介绍【观察者模式】的相关知识，将会涉及以下内容：

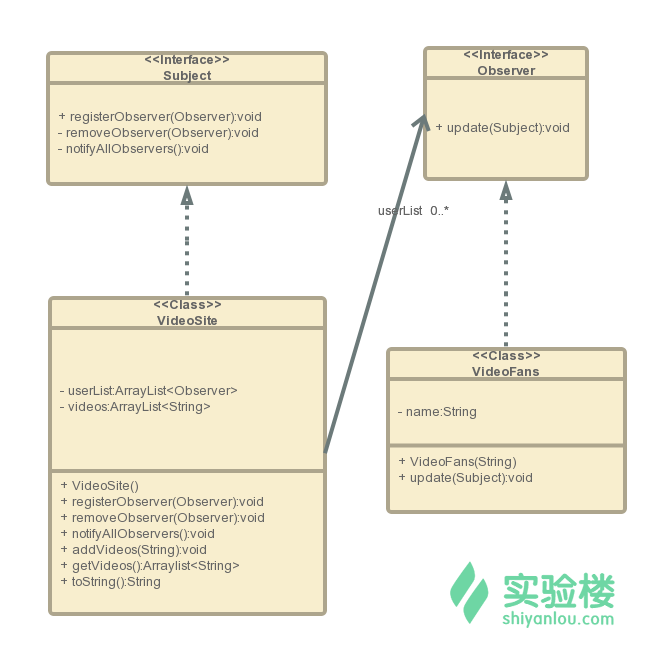
* 什么是观察者模式
* 观察者模式类图
* 观察者模式代码
* 观察者模式的应用

**二、什么是 观察者模式**

一句话，观察者模式（Observer Pattern）就是一种 “发布者-订阅者” 的模式。有时也被称为 “模型-视图”模式、“源-监听者”模式等。在这种模式中，由一个目标对象来管理所有依赖与它的观察者对象，并且当这个目标对象自身发生改变时，会主动向它的观察者们发出通知。

**三、 观察者模式类图**

比如你最近在追一个美剧《生活大爆炸》，假设著名在线视频网站某狐买下独家版权，在线更新与播放。于是你天天等啊等啊，等它的更新通知一来，你就去看那些最新的视频。



**四、 观察者模式示例代码**

Talk is cheap, show me the code.

// Subject 主题接口

public interface Subject {

public void registerObserver(Observer o);

public void removeObserver(Observer o);

public void notifyAllObservers();

}

// 观察者接口

public interface Observer {

public void update(Subject s);

}

// 视频网站某狐 实现 Subject 接口

public class VideoSite implements Subject{

// 观察者列表 以及 更新了的视频列表

private ArrayList<Observer> userList;

private ArrayList<String> videos;

public VideoSite(){

userList = new ArrayList<Observer>();

videos = new ArrayList<String>();

}

@Override

public void registerObserver(Observer o) {

userList.add(o);

}

@Override

public void removeObserver(Observer o) {

userList.remove(o);

}

@Override

public void notifyAllObservers() {

for (Observer o: userList) {

o.update(this);

}

}

public void addVideos(String video) {

this.videos.add(video);

notifyAllObservers();

}

public ArrayList<String> getVideos() {

return videos;

}

public String toString(){

return videos.toString();

}

}

// 实现观察者，即看视频的美剧迷们

public class VideoFans implements Observer {

private String name;

public VideoFans(String name){

this.name = name;

}

@Override

public void update(Subject s) {

System.out.println(this.name + ", new videos are available! ");

// print video list

System.out.println(s);

}

}

// 测试一下

public class Main {

public static void main(String[] args) {

VideoSite vs = new VideoSite();

vs.registerObserver(new VideoFans("LiLei"));

vs.registerObserver(new VideoFans("HanMeimei"));

vs.registerObserver(new VideoFans("XiaoMing"));

// add videos

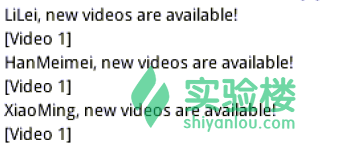
vs.addVideos("Video 1");

//vs.addVideos("Video 2");

}

}

运行结果为：



**五、观察者模式的应用**

前面已经说了，观察者模式也可以理解为 “源-监听者” 模式，这种应用就太多了。举个简单的例子就是各种 listener，比如当你有一个按键，你肯定要给这个按键添加监听事件（listener）来完成指定动作吧，这就是一种应用。

**六、小结**

国际惯例，总个结。本次课程我们看美剧去了... 顺便学习了【观察者模式】的相关知识。

**七、参考文档**

* [Java Design Pattern: Observer](http://www.programcreek.com/2011/01/an-java-example-of-observer-pattern/)

**单例模式**

**一、本节目标**

我们本节课程主要介绍【单例模式】的相关知识，将会涉及以下内容：

* 什么是单例模式
* 单例模式类图
* 单例模式代码
* 单例模式的应用

PS：Java 设计模式系列课程，到这节课就是最后一节啦。相信大家一路学下来还是收获颇丰的，现在回想一下学过的那些设计模式吧，我相信你很快就会有类似于这样的联想片段：

* 工厂模式：女娲娘娘...
* 抽象工厂模式：厨房、食物、吃货...
* 适配器模式：港版 iphone6、充电头、转接头...
* 装饰者模式：好多各式各样的女朋友...
* 观察者模式：视频网站、美剧...

发现没有？通过这样的记忆方式，把知识点都串联起来了，你会记得更牢固更清楚，这也正是我们用心良苦的地方。（不谦虚了...）所以希望大家好好学习、天天向上！

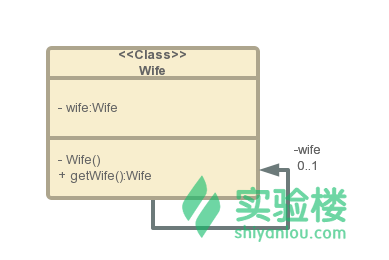
虽然编故事很累，但为了造福广大童鞋们，故事还得继续。Let's go!

**二、什么是单例模式**

单例模式（Singleton Pattern），顾名思义，就是被单例的对象只能有一个实例存在。单例模式的实现方式是，一个类能返回对象的一个引用（永远是同一个）和一个获得该唯一实例的方法（必须是静态方法）。通过单例模式，我们可以保证系统中只有一个实例，从而在某些特定的场合下达到节约或者控制系统资源的目的。

**三、 单例模式类图**

在 【装饰者模式】中，我们体验了拥有各种不同特性的女朋友的 “酸爽”... 不过梦想很丰满，现实很骨感，最后你只能拥有一个老婆。



**四、 单例模式示例代码**

Talk is cheap, show me the code.

在学习本课之前，你也许已经听说或者了解过【单例模式】了。但你知道单例模式有多少种方式实现吗？这个看起来最为简单的设计模式，其实有很多坑...

**1.饿汉模式**

最常见、最简单的单例模式写法之一。顾名思义，“饿汉模式” 就是很 “饥渴”，所以一上来就需要给它新建一个实例。但这种方法有一个明显的缺点，那就是不管有没有调用过获得实例的方法（本例中为 getWife() ），每次都会新建一个实例。

// 饿汉模式

public class Wife {

// 一开始就新建一个实例

private static final Wife wife = new Wife();

// 默认构造方法

private Wife() {}

// 获得实例的方法

public static Wife getWife() {

return wife;

}

}

**2.懒汉模式**

最常见、最简单的单例模式之二，跟 “饿汉模式” 是 “好基友”。再次顾名思义，“懒汉模式” 就是它很懒，一开始不新建实例，只有当它需要使用的时候，会先判断实例是否为空，如果为空才会新建一个实例来使用。

// 懒汉模式

public class Wife {

//一开始没有新建实例

private static Wife wife;

private Wife() { }

// 需要时再新建

public static Wife getWife() {

if (wife == null) {

wife = new Wife();

}

return wife;

}

}

**3.线程安全的懒汉模式**

是不是感觉很简单？但是上面的懒汉模式却存在一个严重的问题。那就是如果有多个线程（独享cpu和资源）并行调用 getWife() 方法的时候，还是会创建多个实例，单例模式就失效了。

Bug 来了，改改改！

简单，我们在基本的懒汉模式上，把它设为线程同步（synchronized）就好了。synchronized 的作用就是保证在同一时刻最多只有一个线程运行，这样就避免了多线程带来的问题。关于 synchronized 关键字，你可以 [点击这里](http://www.cnblogs.com/gnagwang/archive/2011/02/27/1966606.html) 了解更多。

// 懒汉模式（线程安全）

public class Wife {

private static Wife wife;

private Wife() { }

// 添加了 synchronized 关键字

public static synchronized Wife getWife() {

if (wife == null) {

wife = new Wife();

}

return wife;

}

}

**4.双重检验锁（double check）**

线程安全的懒汉模式解决了多线程的问题，看起来完美了。但是它的效率不高，每次调用获得实例的方法 getWife() 时都要进行同步，但是多数情况下并不需要同步操作（例如我的 wife 实例并不为空可以直接使用的时候，就不需要给 getWife() 加同步方法，直接返回 wife 实例就可以了）。所以只需要在第一次新建实例对象的时候，使用同步方法。

不怕，程序猿总是有办法的。于是，在前面的基础上，又有了 “双重检验锁” 的方法。

// 双重锁的 getWife() 方法

public static Wife getWife() {

// 第一个检验锁，如果不为空直接返回实例对象，为空才进入下一步

if (wife == null) {

synchronized (Wife.class) {

//第二个检验锁，因为可能有多个线程进入到 if 语句内

if (wife == null) {

wife = new Wife();

}

}

}

return wife ;

}

你以为这终于圆满了？NO...Too young, too naive! 主要问题在于 wife = new Wife() 这句代码，因为在 JVM（Java 虚拟机）执行这句代码的时候，要做好几件事情，而 JVM 为了优化代码，有可能造成做这几件事情的执行顺序是不固定的，从而造成错误。（为了不把问题更加复杂化，这里没有深入讲解在 JVM 中具体是怎么回事，有兴趣的同学可以点击 [这里](http://www.cnblogs.com/techyc/p/3529983.html) 自行了解下。）

这个时候，我们需要给实例加一个 volatile 关键字，它的作用就是防止编译器自行优化代码。最后，我们的 “双重检验锁” 版本终于出炉了...

// 双重检验锁

public class Wife {

private volatile static Wife wife;

private Wife() { }

public static Wife getWife() {

if (wife == null) {

synchronized(Wife.class) {

if (wife == null) {

wife = new Wife();

}

}

}

return wife;

}

}

**5.静态内部类**

上面的方法，修修补补，实在是太复杂了... 而且 volatile 关键字在某些老版本的 JDK 中无法正常工作。咱们得换一种方法，即 “静态内部类”。这种方式，利用了 JVM 自身的机制来保证线程安全，因为 WifeHolder 类是私有的，除了 getWife()之外没有其它方式可以访问实例对象，而且只有在调用 getWife() 时才会去真正创建实例对象。（这里类似于 “懒汉模式”）

// 静态内部类

public class Wife {

private static class WifeHolder {

private static final Wife wife = new Wife();

}

private Wife() { }

public static Wife getWife() {

return WifeHolder.wife;

}

}

**6.枚举**

还不懂什么是枚举的，先 [点这里](http://zh.wikipedia.org/zh/%E6%259%20E%259%20A%E4%B8%BE) 补补课。

如下，代码简直是简单得不能再简单了。我们可以通过 Wife.INSTANCE 来访问实例对象，这比 getWife() 要简单得多，而且创建枚举默认就是线程安全的，还可以防止反序列化带来的问题。这么 优（niu）雅（bi）的方法，来自于新版 [《Effective Java》](http://book.douban.com/subject/3360807/) 这本书。这种方式虽然不常用，但是最为推荐。

// 枚举

public enum Wife {

INSTANCE;

// 自定义的其他任意方法

public void whateverMethod() { }

}

**五、单例模式的应用**

当你只需要一个实例对象的时候，就可以考虑使用单例模式。比如在资源共享的情况下，避免由于多个资源操作导致的性能或损耗等就可以使用单例模式。

**六、实验总结**

总结一下，我们在最后的单例模式中“娶了个媳妇”。至此，Java 设计模式系列课程就结束了，希望同学们可以在本课程的基础上勤加练习。

**七、参考文档**

* [Java Design Pattern: Singleton](http://www.programcreek.com/2011/07/java-design-pattern-singleton/)