**Java设计模式**

# UML类图

* **统一建模语言，**
* **非专利的第三代建模和规约语言**

## UML特点

* **UML是一种开放方法**
* **用于说明、可视化、构建和编写一个正在开发的面向对象的、软件密集系统的制品的开放方法。**

## UML分类

**UML2.2一共定义了14种图示：**

* 结构式图形：强调的是系统示建模
* 行为式模型：强调系统模型中触发的事件
* 交互式模型：属于行为式模型的子集合，强调系统模式中资料流程

### 结构式模型

* + **静态图 : 类图、对象图、包图**
  + **实现图： 组件图、部署图**
  + **剖面图**
  + **复合结构图**

### 行为式模型

* + **活动图**
  + **状态图**
  + **用例图**

### 交互式模型

* + **通信图**
  + **交互概述图**
  + **时序图**
  + **时间图**

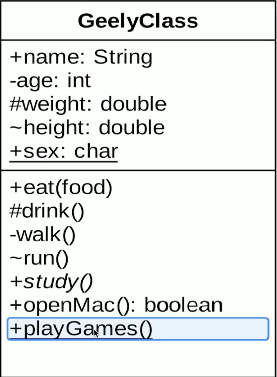
## UML图

### 类图

**用于表示类、接口、实例等之间相互的静态关系**

**注意：**

* + - * **箭头一定是子类指向父类**
      * **实现接口是虚线、继承类是实线**
      * **实现是关联、虚线是依赖**
      * **空心菱形是聚合、实心菱形是组合**



**+ ： 代表public**

**- : 代表private**

**# ： 代表protected**

**~ : 代表默认权限**

**下划线： 代表static**

**斜体：代表抽象方法**

# 设计原则

### 开闭原则

**定义：**

**一个软件实体如类、模块和函数应该对扩展开放，对修改关闭**

**用抽象构建框架，用实现扩展细节**

**优点：**

**提高软件系统的可复用性和可维护性**

### 单一职责原则

**定义：**

**不要存在多余一个导致类变更的原因**

**一个类/接口/方法只负责一项职责**

**优点：**

**降低类的复杂度、提高类的可读性，提高系统的可维护性、降低变更**

**引发的风险**

### 依赖倒置原则

**定义：**

**高层模块不应该依赖低层模块，二者都应该依赖其抽象**

**抽象不应该依赖细节，细节应该依赖于抽象；针对接口编程，不要针对实现编程。**

### 接口隔离原则

**定义：**

* + **用多个专门的接口，而不使用单一的总接口，客户端不应该依赖它不需要的接口；**
  + **一个类对一个类的依赖应该建立在最小的接口上；**
  + **建立单一接口，不要建立庞大臃肿的接口；**
  + **尽量细化接口，接口中的方法尽量少**
  + **注意适度原则，一定要适度**

**优点：**

**符合我们常说的高类聚低耦合的设计思想，从而使得类具有很好的可读性、可扩展性和可维护性。**

### 迪米特原则

**定义：**

* + **一个对象应该对其他对象保持最少的了解。又叫最少知道原则**
  + **尽量降低类与类之间的耦合**

**优点：**

**降低类之间的耦合**

**只和朋友说话，不和陌生人交流**

**朋友：**

**出现在成员变量、方法的输入、输出参数中的类称为成员朋友类，而出现在方法体内部的类不属于朋友类。**

# 设计模式

### 简单工厂

**定义：**

**由一个工厂对象决定创建出哪一种产品实例**

**类型：**

**创建型；但不属于GOF23种设计模式当中**

**适应场景：**

**工厂类负责创建的对象较少**

**客户端(应用层)只知道传入工厂类的参数，对于如何创建对象(逻辑)不关心**

**缺点：**

**工厂类的职责相对过重，增加新的产品需要修改工厂类的判断逻辑，违背开闭原则**

#### 代码

**public class** SimpleFactory {  
 */\*  
 V1版本  
 public Video getVideo(String type){  
 if ("java".equalsIgnoreCase(type)){  
 return new JavaVideo();  
 }else if ("Python".equalsIgnoreCase(type)){  
 return new PythonVideo();  
 }  
 return null;  
 }\*/* **public** Video getVideo(Class<? **extends** Video> clz){  
 Video video = **null**;  
 **try** {  
 video = clz.newInstance();  
 } **catch** (InstantiationException e) {  
 e.printStackTrace();  
 } **catch** (IllegalAccessException e) {  
 e.printStackTrace();  
 }  
 **return** video;  
 }  
}

### 工厂方法模式

**定义：**

定义一个创建对象的接口，但让实现这个接口的类来决定实例化哪个类

工厂方法让类的实例化推迟到子类中进行.

**类型**：创建型

#### 适用场景

* 创建对象需要大量重复的代码
* 客户端(应用层)不依赖于产品类实例如何被创建、实现等细节
* 一个类通过其子类来指定创建哪个对象

#### 优点

* 用户只需要关心所需要的产品对应的工厂类，无需关心创建细节
* 加入新产品符合开闭原则，提高可扩展性

#### 缺点

* 类的个数容易过多
* 增加了系统的抽象性和理解难度

### 抽象工厂模式

**定义：**

**抽象工厂模式提供一个创建一系列相关或相互依赖对象的接口**

**无需指定他们具体的类**

**类型：创建型**

#### 适用场景

* **客户端(应用层)不依赖于产品类实例如何被创建、实现等细节**
* **强调一系列相关的产品对象(属于同一产品族)一起使用创建对象需要大量重复的代码**
* **提供一个产品类的库，所有的产品以同样的接口实现，从而使客户端不依赖于具体实现**

#### 优点

* **具体产品在应用层代码隔离, 无需关心创建细节**
* **将一个系列的产品族统一到一起创建**

#### 缺点

* + - * + **规定了所有可能被创建的产品集合，产品族中扩展新的产品困难，**

**需要修改抽象工厂的接口**

* + - * + **增加了系统的抽象性和理解难度**

#### 与工厂方法的区别

工厂方法主要注重的是**产品等级**

抽象工厂主要注重的是**产品族**

### 建造者模式

**定义：**

将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

用户只需要指定需要建造的类型就可以得到他们，建造过程及细节不需要知道

**类型：**创建型

#### 适用场景

* 如果一个对象非常复杂的内部结构
* 想把复杂对象的创建和使用分离

#### 优点

* 封装性好，创建和使用分离
* 扩展性好、建造类之间独立、一定程度上解耦

#### 缺点

* 产生多余的builder对象
* 产品内部发生变化，建造者都需要修改，成功较大

### 单例模式

**定义：**

**保证一个类仅有一个实例，并提供一个全局访问点**

#### 适用场景

**想确保任何情况下都绝对只有一个实例**

#### 优点

**在内存里只有一个实例，减少了内存开销**

**可以避免对资源的多重占用**

#### 缺点

**没有接口，扩展困难**

#### 重点

* + **私有构造器**
  + **线程安全**
  + **延迟加载**
  + **序列化和反序列化安全问题**
  + **防止反射攻击**

#### 代码

##### 懒汉式

*/\*\*  
 \* 懒汉式  
 \* 延迟加载对象  
 \*/***public class** LazySingleton {  
  
 **private static** LazySingleton *instance* = **null**;  
  
 **private** LazySingleton(){}  
  
 **public synchronized static** LazySingleton getInstance(){  
 **if** (*instance* == **null**){  
 *instance* = **new** LazySingleton();  
 }  
 **return** *instance*;  
 }  
}

**懒汉式在饿汉式的基础上增加了延迟加载的特性，但增加了锁的开销**

**优化后： (双重锁检查)**

*/\*\*  
 \* 懒汉式  
 \* 延迟加载对象  
 \*/***public class** LazySingleton2 {  
  
 **private static** LazySingleton2 *instance* = **null**;  
  
 **private** LazySingleton2(){}  
  
 */\*\*  
 \* 在方法上添加锁，属于类锁, 我们可以优化下该方法  
 \** ***@return*** *\*/* **public static** LazySingleton2 getInstance(){  
 *// 首先判断实例是否为空，如果不为空则直接返回实例对象，这样就不用进入同步锁,减少线程等待的时间* **if** (*instance* == **null**){  
 **synchronized** (LazySingleton2.**class**){  
 *// 因为在上面判断实例是否为空时，没有在锁，可以有多个线程进入该代码，所以需要重新判断实例是否为空* **if** (*instance* == **null**){  
 *instance* = **new** LazySingleton2();  
 }  
 }  
 }  
 **return** *instance*;  
 }  
}

##### 静态内部类

*/\*\*  
 \* 使用内部类实现单例  
 \*/***public class** InnerSingleton {  
  
 **private** InnerSingleton(){}  
  
 **private static class** SingletonObj{  
 **private static** InnerSingleton *innerSingleton* = **new** InnerSingleton();  
 }  
  
 **public static** InnerSingleton getInstance(){  
 **return** SingletonObj.*innerSingleton*;  
 }  
  
}

**使用静态内部类的方式实现，既能保证线程安全，也能保持延迟加载的特性。其原理是**

**在类的初始化阶段，JVM会获取一个外部类Class的锁，除了构建线程其他线程都不能进入其内部，这样内部类构建对象时就能保证线程的安全。**

##### 饿汉式

*/\*\*  
 \* 饿汉式  
 \* - 实例初始化之前就创建对象  
 \*  
 \* 优点：  
 \* 不用考虑线程安全问题  
 \* 缺点：  
 \* 在不用的情况下, 增加了内存消耗  
 \*/***public class** HungrySingleton {  
  
 **private final** HungrySingleton **hungrySingleton** = **new** HungrySingleton();  
  
 **public** HungrySingleton getInstance(){  
 **return hungrySingleton**;  
 }  
  
}

##### 序列化攻击

**public class** HungryTest {  
  
 **public static void** main(String[] args) **throws** IOException, ClassNotFoundException {  
  
 HungrySingleton singleton = HungrySingleton.*getInstance*();  
 ObjectOutputStream out = **new** ObjectOutputStream(**new** FileOutputStream(**"ser\_file"**));  
 out.writeObject(singleton);  
  
 ObjectInputStream in = **new** ObjectInputStream(**new** FileInputStream(**"ser\_file"**));  
 HungrySingleton newSingleton = (HungrySingleton) in.readObject();  
  
 System.***out***.println(singleton == newSingleton);  
 }  
}

将单例对象序列化反序列化之后，不是同一个对象

加上下面这个方法, 在反序列化时会调用这个方法：

**private** HungrySingleton readResolve(){  
 **return *hungrySingleton***;  
}

##### 反射攻击

*// 反射攻击*Class clz = HungrySingleton.**class**;  
HungrySingleton singleton = HungrySingleton.*getInstance*();  
Constructor constructor = clz.getDeclaredConstructor();  
constructor.setAccessible(**true**);  
HungrySingleton newSingleton = (HungrySingleton) constructor.newInstance();  
  
System.***out***.println(singleton);  
System.***out***.println(newSingleton);  
System.***out***.println(newSingleton == singleton);

通过反射可以破坏掉其内部私有的构造方法，使其可以重新创建新的对象

##### 枚举方式创建单例

**public enum** Enum {  
  
 ***INSTANCE***;  
  
 **private static** Singleton *singleton* = **new** Singleton();  
  
 **public static** Singleton getSingleton() {  
 **return** *singleton*;  
 }  
  
 **public static void** setSingleton(Singleton singleton) {  
 Enum.*singleton* = singleton;  
 }  
  
}  
  
  
**class** Singleton{  
  
  
}

##### 基于容器上的单例

*/\*\*  
 \* 容器单例  
 \*  
 \*/***public class** ContainerSingleton {  
  
 **private** ContainerSingleton(){}  
  
 **private static** Map<String, Object> *container* = **new** HashMap<String, Object>();  
  
 **public void** pushInstance(String key, Object object){  
 **if** (key != **null** && key.trim() != **""** && object != **null** && !*container*.containsKey(key)){  
 *container*.put(key, object);  
 }  
 }  
  
 **public** Object getInstance(String key){  
 **if** (key != **null**){  
 **return** *container*.get(key);  
 }  
 **return null**;  
 }  
  
}

##### 线程单例

**public class** ThreadLocalSingleton {  
  
 **private static final** ThreadLocal<ThreadLocalSingleton> ***LOCAL*** = **new** ThreadLocal<ThreadLocalSingleton>(){  
 @Override  
 **protected** ThreadLocalSingleton initialValue() {  
 **return new** ThreadLocalSingleton();  
 }  
 };  
  
 **private** ThreadLocalSingleton(){}  
  
 **public static** ThreadLocalSingleton getInstance(){  
 **return *LOCAL***.get();  
 }  
  
}

基于ThreadLocal的单例模式，是实现了单个线程单例，同一个线程只存在同一个对象，但是不同的线程会有不同的实例

**public static void** main(String[] args) {  
 Runnable runnable = **new** Runnable() {  
 **public void** run() {  
 System.***out***.println(Thread.*currentThread*().getName() + **":"** + ThreadLocalSingleton.*getInstance*());  
 System.***out***.println(Thread.*currentThread*().getName() + **":"** + ThreadLocalSingleton.*getInstance*());  
 System.***out***.println(Thread.*currentThread*().getName() + **":"** + ThreadLocalSingleton.*getInstance*());  
 }  
 };  
 Thread t1 = **new** Thread(runnable);  
 Thread t2 = **new** Thread(runnable);  
 Thread t3 = **new** Thread(runnable);  
 t1.start();  
 t2.start();  
 t3.start();  
}

### 原型模式

**定义：**

**指原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象**

**不需要知道任何创建的细节，不调用构造函数**

**类型：创建型**

#### 适用场景

* **类初始化消耗较多资源**
* **New产生的一个对象需要非常复杂繁琐的过程(数据准备、访问权限等)**
* **构造函数比较复杂**
* **循环体中生产大量的对象时**

#### 优点

* **原型模式性能比直接new一个对象性能高**
* **简化创建过程**

#### 缺点

* **必须配备克隆方法**
* **对克隆复杂对象或对克隆出的对象进行复杂改造时，容易引入风险**
* **深拷贝、浅拷贝要运用得当**

#### 克隆破坏单例

**如果单例类继承了Closeable接口，那么类的实例就允许被克隆，**

**破坏了单例的**

### 外观模式

**定义：**

**又叫门面模式，提供了一个统一的接口，用来访问子系统中的一群接口**

**外观模式定义一个高层接口，让子系统更容易使用**

**类型：结构**型

#### 适用场景

* 子系统越来越复杂，增加外观模式提供简单调用接口
* 构建多层系统结构，利用外观对象作为每层的入口，简化调用

#### 优点

* 简化了调用过程, 无需了解深入子系统，防止带来风险
* 减少系统依赖，松散耦合
* 更好的划分访问层次

#### 缺点

* 增加子系统、扩展子系统行为容易引入风险
* 不符合开闭原则

### 装饰者模式

**定义：**

在不改变原有对象的基础上，将功能附加到对象上。

提供了比继承更有弹性的替代方案。

类型：结构型

#### 适用场景

扩展一个类的功能或者给一个类添加附加职责

动态的给一个对象添加功能，这些功能可以再动态的撤销

#### 优点

继承的有力补充，比继承灵活,不改变原有对象的情况下给一个对象扩展功能

通过使用不同装饰类以及这些装饰类的排列组合，可以实现不同效果

符合开闭原则

### 适配器模式

**定义：**

**将一个接口转成客户所期望的的另一个接口, 使原本不兼容的类可以一起工作**

**类型： 结构型**

#### 适用场景

* **已经存在的类，方法和需求不匹配时(方法结果相同或相似)**
* **不是软件设计阶段考虑的设计模式，是随着软件维护，由于不同产品、不同厂家造成功能类似而接口不相同情况下的解决方案。**

#### 优点

* **提高现有类的透明性和复用，现有的类复用但不需要改变**
* **目标类适配器类解耦，提高程序的扩展性**
* **符合开闭原则**

#### 缺点

* **在设计过程中需要全面考虑，可能会增加系统的复杂度**
* **增加代码的可读性**

#### 扩展

* **对象适配器**
* **类适配器**

### 享元模式

**定义：**

**提供了减少对象数量从而改善应用所需要的对象结构的方式**

**运用共享技术有效的支持大量细粒度的对象**

**类型：结构型**

#### 适用场景

* **常常应用于底层的开发，以便解决系统的性能问题**
* **系统有大量相似对象、需要缓冲池的场景**

#### 优点

* **减少对象的创建，降低内存中对象的数量，降低系统的内存，提高效率**
* **减少内存之外的其他资源占用**

#### 缺点

* **关注内/外部状态、关注线程安全问题**
* **使系统、程序的逻辑复杂化**