## 一句话概括：

**顾名思义，一个类只有一个实例**。

## 补充介绍：

单例模式（Singleton Pattern）是 Java 中最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

**注意：**

单例模式只能有一个实例；

单例类必须自己创建自己的唯一实例；

单例类必须给所有其他对象提供这个实例

## 参与角色:

1）单例类（自己创建单一示例，并且提供一个共有方法返回该实例）

**优点：**

节省资源，避免不必要的开销，避免重复的创建销毁对象。

## 缺点：

很开心，面向对象的特性： 抽象 继承 多态， 它几乎没有。

## 使用案例或场景：

太多太多啦，几乎任何一个框架里面都会用到，比如Hibernate的SessionFactory，它就是单例的，再比如Spring MVC里面的DispatcherServlet，它也是的，噢不，不止，Spring 里面的bean默认都是单例的。

使用场景：那些使用频率高并且类的变量和方法几乎不会改变，这样的类可以创建成单例的

## 示例程序

需要源码的朋友可以前往github下载：

<https://github.com/aharddreamer/chendong/tree/master/design-patterns/demo-code/design-patterns>

**程序简介**：

在下面这些类中，我们创建了各式各样的单例类，他们的共同点是在单例类中有私有的构造函数，自己创建一个单一实例，并且有一个公共方法对外提供该实例。请看类注释来理解他们的优缺点。

**代码：**

**package** org.cd.designpatterns.singleton;  
  
*/\*\*  
 \* 懒汉式，线程不安全  
 \* 这种方式是最基本的实现方式，这种实现最大的问题就是不支持多线程。  
 \* 因为没有加锁 synchronized，所以严格意义上它并不算单例模式。  
 \*  
 \*/***public class** LazySingletonObject {  
  
 **private static** LazySingletonObject *instance*;  
  
 *//是该对象不能被外部实例化* **private** LazySingletonObject() {  
 }  
  
 *//返回单一实例， 如果为空则创建实例，线程不安全* **public static** LazySingletonObject getInstance() {  
 **if** (*instance* == **null**) {  
 *instance* = **new** LazySingletonObject();  
 }  
 **return** *instance*;  
 }  
}

**package** org.cd.designpatterns.singleton;  
  
*/\*\*  
 \* 懒汉式，线程安全  
 \* 这种方式具备很好的 lazy loading，能够在多线程中很好的工作，但是，效率很低，99% 情况下不需要同步  
 \* 优点：第一次调用才初始化，避免内存浪费。  
 \* 缺点：必须加锁 synchronized 才能保证单例，但加锁会影响效率。  
 \*/***public class** LazyButSecuritySingletonObject {  
  
 **private static** LazyButSecuritySingletonObject *instance*;  
  
 *//是该对象不能被外部实例化* **private** LazyButSecuritySingletonObject() {  
 }  
  
 *//返回单一实例， 如果为空则创建实例，线程不安全* **public static synchronized** LazyButSecuritySingletonObject getInstance() {  
 **if** (*instance* == **null**) {  
 *instance* = **new** LazyButSecuritySingletonObject();  
 }  
 **return** *instance*;  
 }  
}

**package** org.cd.designpatterns.singleton;  
  
*/\*\*  
 \* 饿汉式单例模式  
 \* 这种方式比较常用，但容易产生垃圾对象。  
 \* 优点：没有加锁，执行效率会提高。  
 \* 缺点：类加载时就初始化，浪费内存。  
 \* 它基于 classloader 机制避免了多线程的同步问题，不过，instance 在类装载时就实例化，虽然导致类装载的原因有很多种，在单例模式中大多数都是调用 getInstance 方法， 但是也不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化 instance 显然没有达到 lazy loading 的效果。  
 \*/***public class** HungrySingletonObject {  
  
 **private static** HungrySingletonObject *instance* = **new** HungrySingletonObject();  
  
 *//是该对象不能被外部实例化* **private** HungrySingletonObject() {  
 }  
  
 *//返回单一实例* **public static** HungrySingletonObject getInstance() {  
 **return** *instance*;  
 }  
}

**package** org.cd.designpatterns.singleton;  
  
*/\*\*  
 \* 双重校验锁  
 \* 优点：这种方式采用双锁机制，安全且在多线程情况下能保持高性能。  
 \* 缺点：较复杂，你很可能现在看了明天就忘了  
 \*  
 \*/***public class** DoubleLockSingletonObject {  
  
 **private volatile static** DoubleLockSingletonObject *instance*;  
  
 **private** DoubleLockSingletonObject() {  
 }  
  
 **public static** DoubleLockSingletonObject getInstance() {  
 **if** (*instance* == **null**) {  
 **synchronized** (DoubleLockSingletonObject.**class**) {  
 **if** (*instance* == **null**) {  
 *instance* = **new** DoubleLockSingletonObject();  
 }  
 }  
 }  
 **return** *instance*;  
 }  
}

**package** org.cd.designpatterns.singleton;  
  
*/\*\*  
 \* 内部类方式  
 \* 这种方式能达到双检锁方式一样的功效，但实现更简单。对静态域使用延迟初始化，应使用这种方式而不是双检锁方式。这种方式只适用于静态域的情况，双检锁方式可在实例域需要延迟初始化时使用。  
 \* 这种方式能达到双检锁方式一样的功效，但实现更简单。对静态域使用延迟初始化，应使用这种方式而不是双检锁方式。这种方式只适用于静态域的情况，双检锁方式可在实例域需要延迟初始化时使用。  
 \* 这种方式同样利用了 classloader 机制来保证初始化 instance 时只有一个线程，它跟第 3 种方式不同的是：第 3 种方式只要 Singleton 类被装载了，那么 instance 就会被实例化（没有达到 lazy loading 效果），而这种方式是 Singleton 类被装载了，instance 不一定被初始化。因为 SingletonHolder 类没有被主动使用，只有通过显式调用 getInstance 方法时，才会显式装载 SingletonHolder 类，从而实例化 instance。想象一下，如果实例化 instance 很消耗资源，所以想让它延迟加载，另外一方面，又不希望在 Singleton 类加载时就实例化，因为不能确保 Singleton 类还可能在其他的地方被主动使用从而被加载，那么这个时候实例化 instance 显然是不合适的。这个时候，这种方式相比第 3 种方式就显得很合理。  
 \*  
 \*/***public class** InnerClassSingletonObject {  
  
 **private static class** SingletonHolder {  
 **private static final** InnerClassSingletonObject ***INSTANCE*** = **new** InnerClassSingletonObject();  
 }  
  
 **private** InnerClassSingletonObject() {}  
  
 **public static final** InnerClassSingletonObject getInstance() {  
 **return** SingletonHolder.***INSTANCE***;  
 }  
}

参考：

《单例模式》菜鸟教程网站