单例模式(Singleton Pattern)：确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，它提供全局访问的方法。单例模式是一种对象创建型模式。

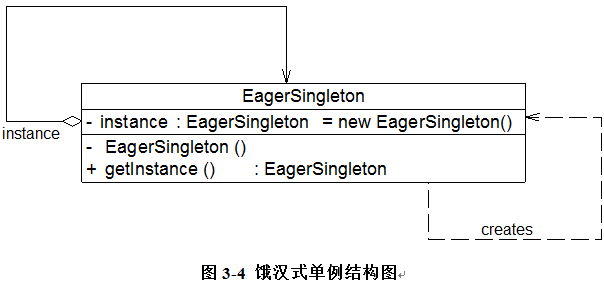
三个要点：

1. 是某个类只能有一个实例；

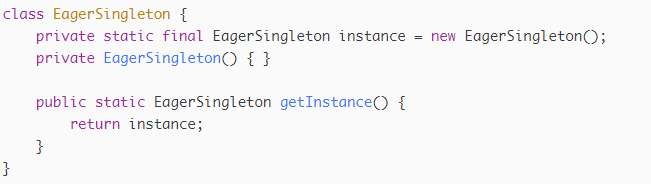
2. 必须自行创建这个实例；

3. 必须自行向整个系统提供这个实例。

单例类的两种不同实现方式，**饿汉式单例类**和**懒汉式单例类**。

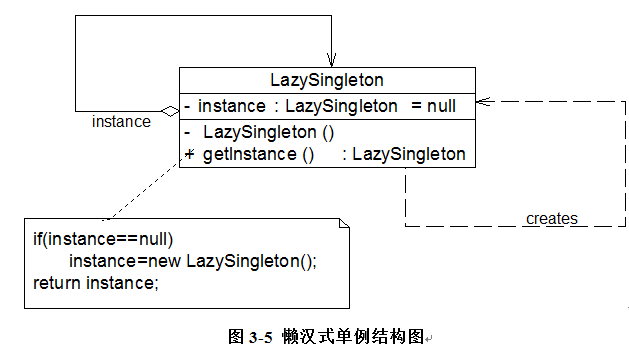
1. 饿汉式单例类是实现起来最简单的单例类。结构图如下。

从图中可以看出，由于在定义静态变量的时候实例化单例类，因此在类加载的时候就已经创建了单例对象，代码如下所示：

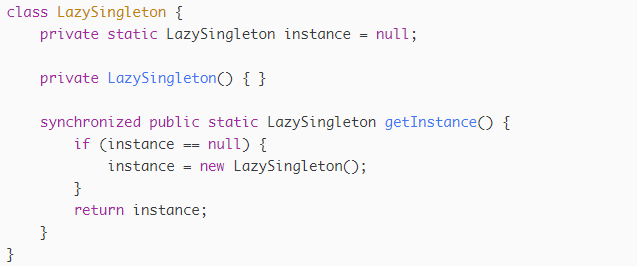


当类被加载时，静态变量instance会被初始化，此时类的私有构造函数会被调用，单例类的唯一实例将被创建。如果使用饿汉式单例，则不会出现创建多个单例对象的情况，可确保单例对象的唯一性。

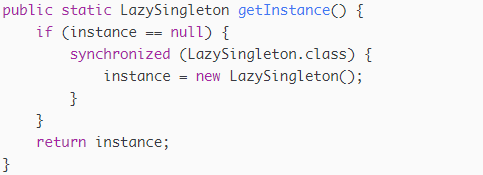
2. 懒汉式单例类与线程锁定



从图中可以看出，懒汉式单例在第一次调用getInstance()方法时实例化，在类加载时并不自行实例化，这种技术又称为**延迟加载**(Lazy Load)技术，即需要的时候再加载实例，为了避免多个线程同时调用getInstance()方法，我们可以使用关键字**synchronized**，代码如下所示：



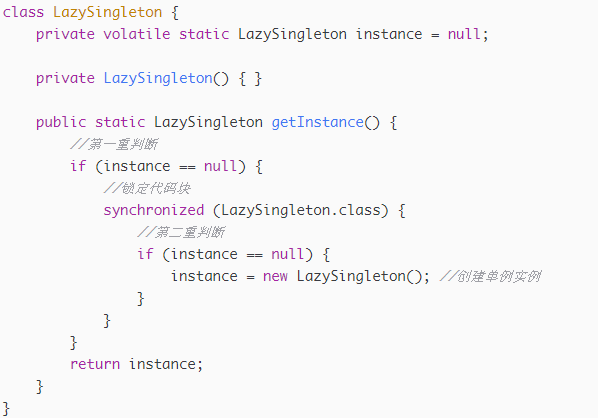
上述代码虽然解决了线程安全问题，但是性能大大降低。我们可以直接锁定生成实例的代码而非整块代码。如



这也暴露一个问题，如果线程A、线程B都在调用getInstance()方法，线程A进行synchronized锁定的代码，实例也生成了，但是新值并没有刷新到主内存中，所以线程B执行synchronized锁定的代码时，会认为实例并没有创建。

所以再改改。

先在synchronized中再进行一次(instance == null)判断，这种方式称为**双重检查锁定**(Double-Check Locking)。



可以看出，instance变量使用了volatile关键字，那么它的作用就是，单一个线程修改了这个变量的值，volatile保证了新值能立即同步到主内存，以及每次使用前立即从主内存刷新。但普通变量做不到这点，普通变量的值在线程间传递均需要通过主内存（详见：Java内存模型）来完成。

volatile 的读性能消耗与普通变量几乎相同，但是写操作稍慢，因为它需要在本地代码中插入许多内存屏障指令来保证处理器不发生乱序执行。

由于volatile关键字会屏蔽Java虚拟机所做的一些代码优化，可能会导致系统运行效率降低，因此即使使用双重检查锁定来实现单例模式也不是一种完美的实现方式。

[Java中Volatile关键字详解](https://www.cnblogs.com/zhengbin/p/5654805.html)

两种实现方式的比较

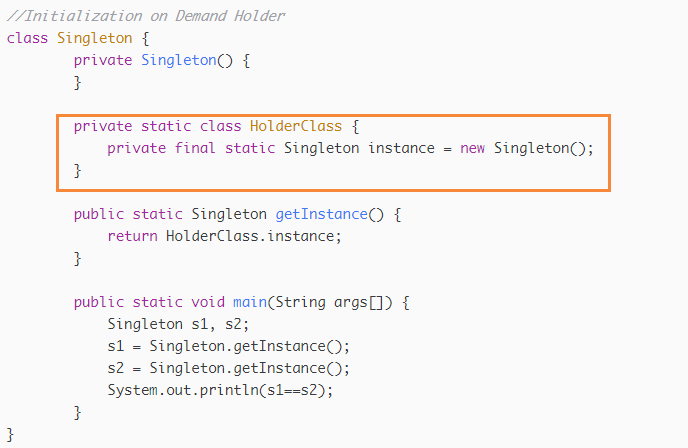
**饿汉式单例类**在类被加载时就将自己实例化，它的优点在于无须考虑多线程访问问题，可以确保实例的唯一性；从调用速度和反应时间角度来讲，由于单例对象一开始就得以创建，因此要优于懒汉式单例。但是无论系统在运行时是否需要使用该单例对象，由于在类加载时该对象就需要创建，因此从资源利用效率角度来讲，饿汉式单例不及懒汉式单例，而且在系统加载时由于需要创建饿汉式单例对象，加载时间可能会比较长。

**懒汉式单例类**在第一次使用时创建，无须一直占用系统资源，实现了延迟加载，但是必须处理好多个线程同时访问的问题，特别是当单例类作为资源控制器，在实例化时必然涉及资源初始化，而资源初始化很有可能耗费大量时间，这意味着出现多线程同时首次引用此类的机率变得较大，需要通过双重检查锁定等机制进行控制，这将导致系统性能受到一定影响。

一种更好的单例是实现方法，IoDH技术

IoDH(initialization on demand holder)： **一种延迟加载且线程安全的单例模式实现方式。这种方式的实现依赖于JVM对类加载过程中初始化阶段的执行。**

在IoDH中，我们在单例类中增加一个静态(static)内部类，在该内部类中创建单例对象，再将该单例对象通过getInstance()方法返回给外部使用，实现代码如下所示：



通过使用IoDH，我们既可以实现延迟加载，又可以保证线程安全，不影响系统性能，不失为一种最好的Java语言单例模式实现方式（其缺点是与编程语言本身的特性相关，很多面向对象语言不支持IoDH）。

单例模式总结：

优点：

      (1) 单例模式提供了对唯一实例的受控访问。因为单例类封装了它的唯一实例，所以它可以严格控制客户怎样以及何时访问它。

      (2) 由于在系统内存中只存在一个对象，因此可以节约系统资源，对于一些需要频繁创建和销毁的对象单例模式无疑可以提高系统的性能。

      (3) 允许可变数目的实例。基于单例模式我们可以进行扩展，使用与单例控制相似的方法来获得指定个数的对象实例，既节省系统资源，又解决了单例单例对象共享过多有损性能的问题。

缺点：

      (1) 由于单例模式中没有抽象层，因此单例类的扩展有很大的困难。

      (2) 单例类的职责过重，在一定程度上违背了“单一职责原则”。因为单例类既充当了工厂角色，提供了工厂方法，同时又充当了产品角色，包含一些业务方法，将产品的创建和产品的本身的功能融合到一起。

适用场景：

(1) 系统只需要一个实例对象，如系统要求提供一个唯一的序列号生成器或资源管理器，或者需要考虑资源消耗太大而只允许创建一个对象。

(2) 客户调用类的单个实例只允许使用一个公共访问点，除了该公共访问点，不能通过其他途径访问该实例。

**[单例模式](https://blog.csdn.net/LoveLion/article/details/17517213)**

[确保对象的唯一性——单例模式 （一）：单例模式的动机，单例模式概述](https://blog.csdn.net/lovelion/article/details/7420883)

[确保对象的唯一性——单例模式 （二）：负载均衡器的设计与实现](https://blog.csdn.net/lovelion/article/details/7420885)

[确保对象的唯一性——单例模式 （三）：饿汉式单例与懒汉式单例的讨论](https://blog.csdn.net/lovelion/article/details/7420886)

[确保对象的唯一性——单例模式 （四）：一种更好的单例实现方法（静态内部类）](https://blog.csdn.net/lovelion/article/details/7420888)

[确保对象的唯一性——单例模式 （五）：单例模式总结](https://blog.csdn.net/lovelion/article/details/7420889)