工厂模式：

工厂模式属于创建型模式，提供了一种创建对象的最佳方式。在该模式中，创建对象时不会对客户端暴露创建逻辑。

当需要在不同条件下创建不同的对象时，可以使用工厂模式，其主要解决的是接口选择的问题。

作为一种创建型模式，在任何需要生成复杂对象的地方，都可以使用工厂模式。而简单对象，特别是只需要通过new就可以完成创建的对象，无需使用工厂模式，否则会引入一个工厂类，会增加系统的复杂度。

优点：调用者创建一个对象，只需要知道其名称就行，屏蔽了产品的具体实现；扩展性高，

　　　如果增加一个产品，只需要扩展一个工厂类

缺点：每次增加一个产品，都需要增加一个具体工厂类，使得系统中类的个数增加。增加系 　统复杂度

实现：

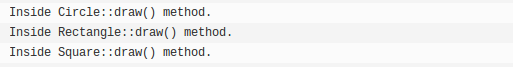








输出为：



单例模式：

单例模式属于创建型模式，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。即：

1. 单例类只能有一个实例
2. 单例类必须自己创建自己的该唯一实例
3. 单例类必须向外提供此唯一实例

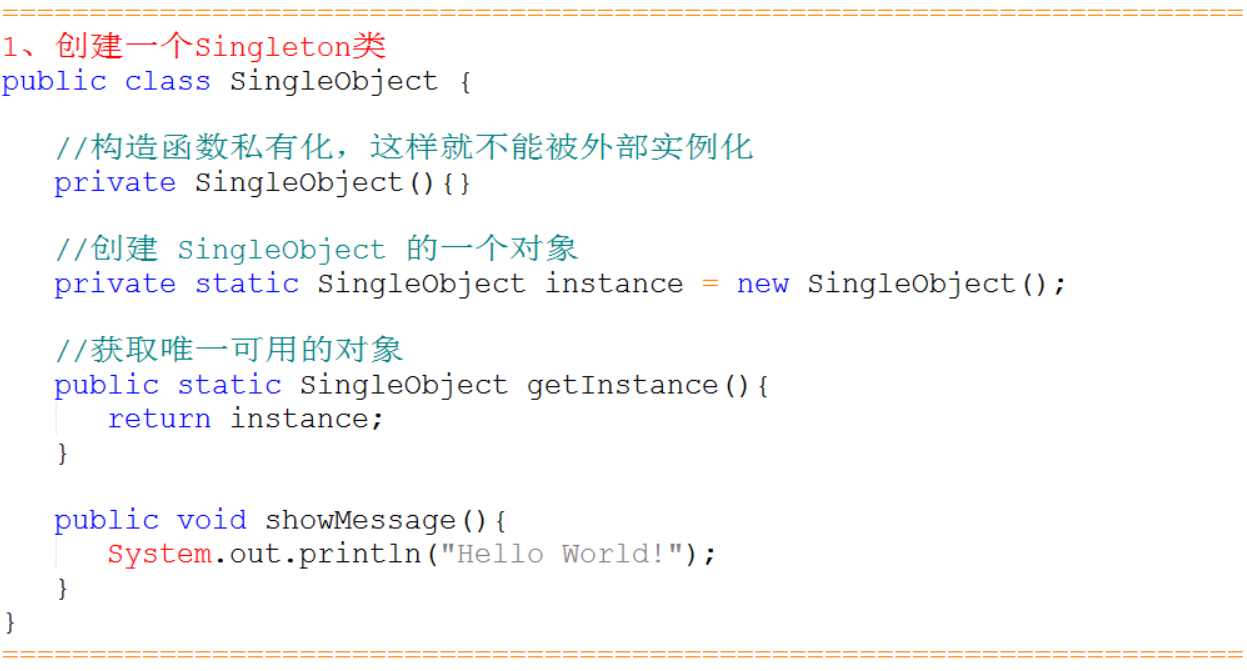
**实现单例模式的关键代码是将构造函数私有化**

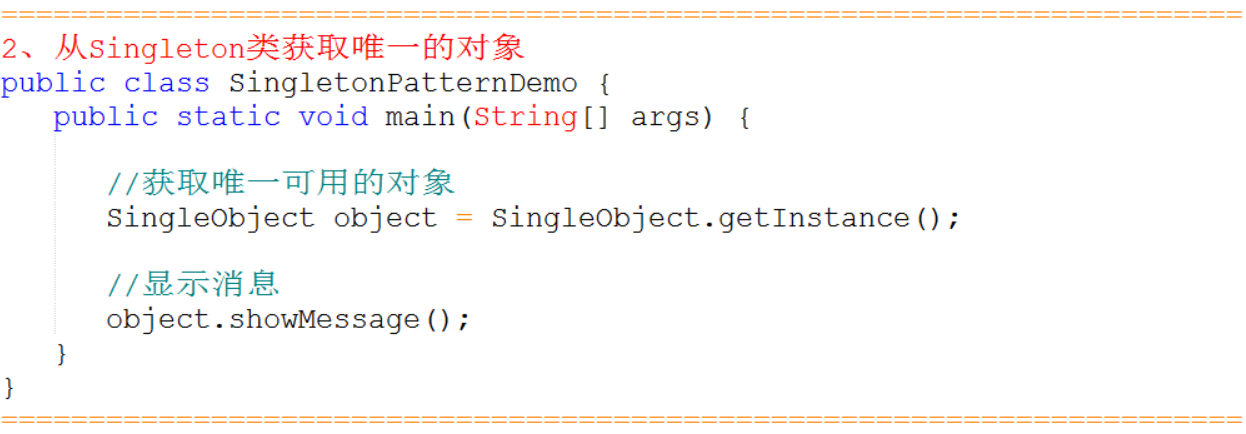
优点：

在内存中只有一个实例，减少了内存的开销；避免对资源的多重占用(比如写文件操作)

缺点：

没有接口，不能继承；一个类应该只关心内部逻辑，而不关心外部怎么将其实例化

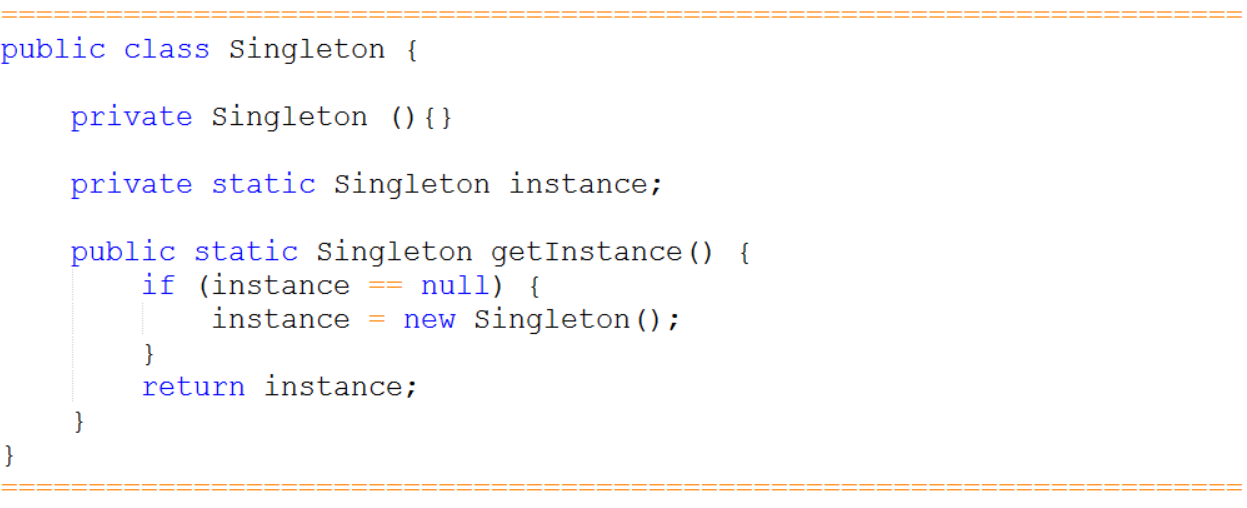




**单例模式的实现方式：**

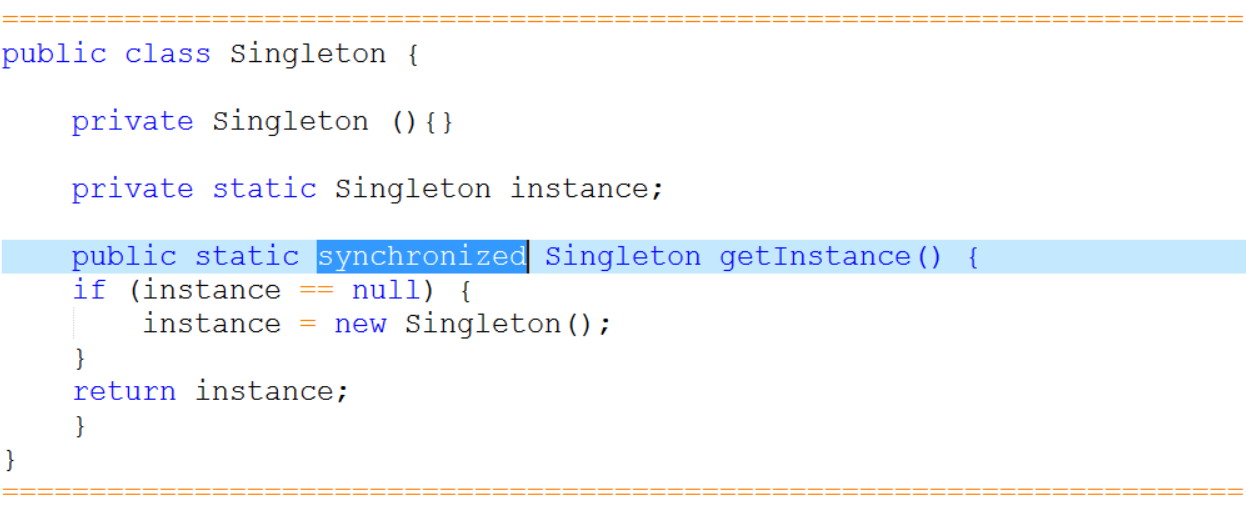
1. **懒汉式，线程不安全**

最基本的实现方式，不支持多线程，因为没有加锁synchronized，严格意义上讲并不算单例模式



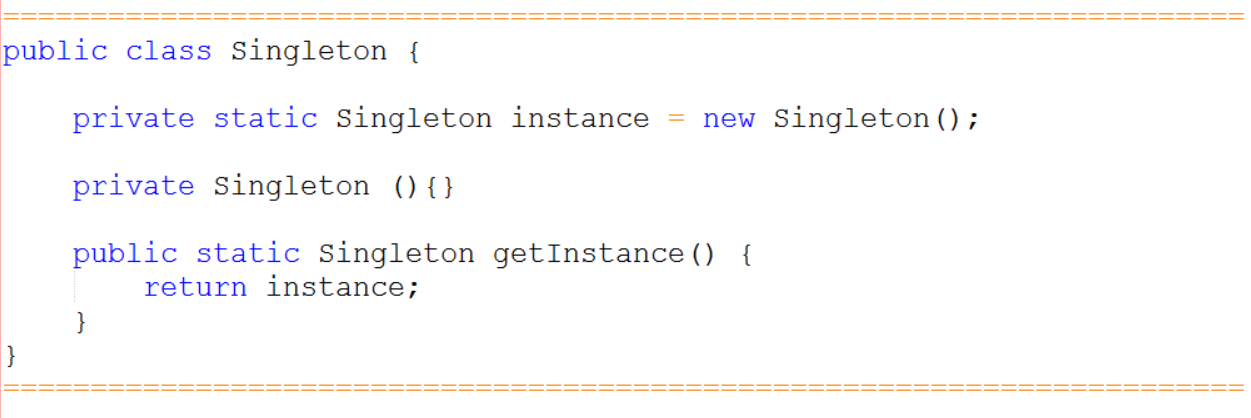
1. **懒汉式，线程安全**

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



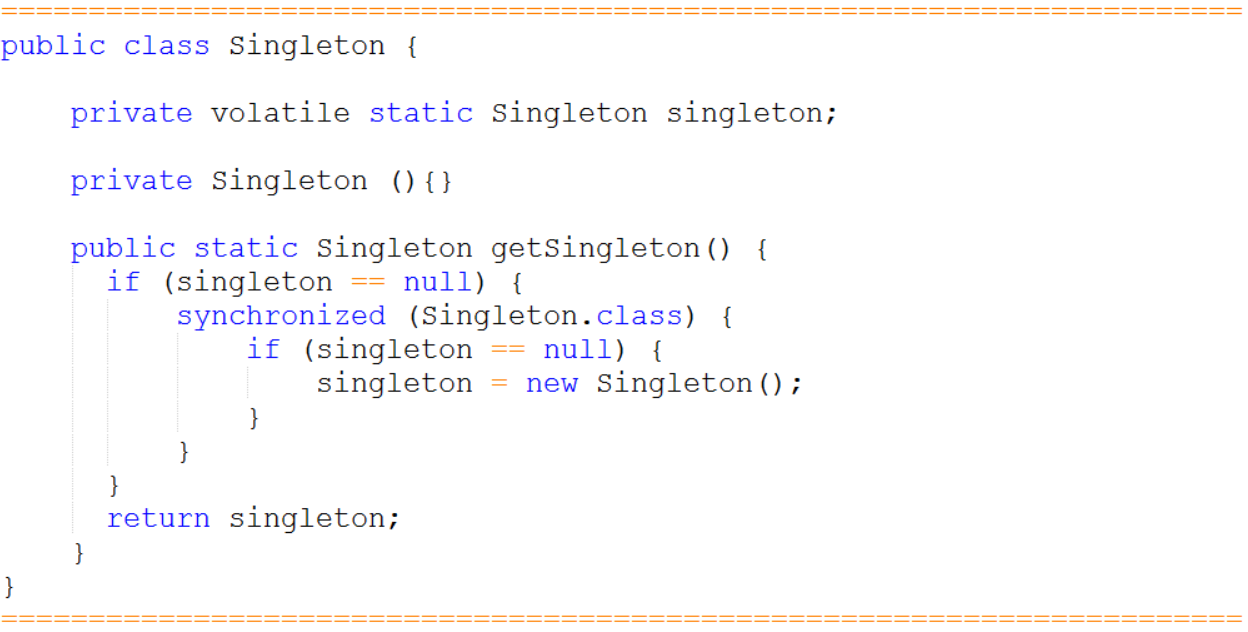
1. **饿汉式**

比较常用，但容易产生垃圾对象，优点是没有枷锁，执行效率会提高，缺点是在类加载时就初始化，会浪费内存。它基于 classloder 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。



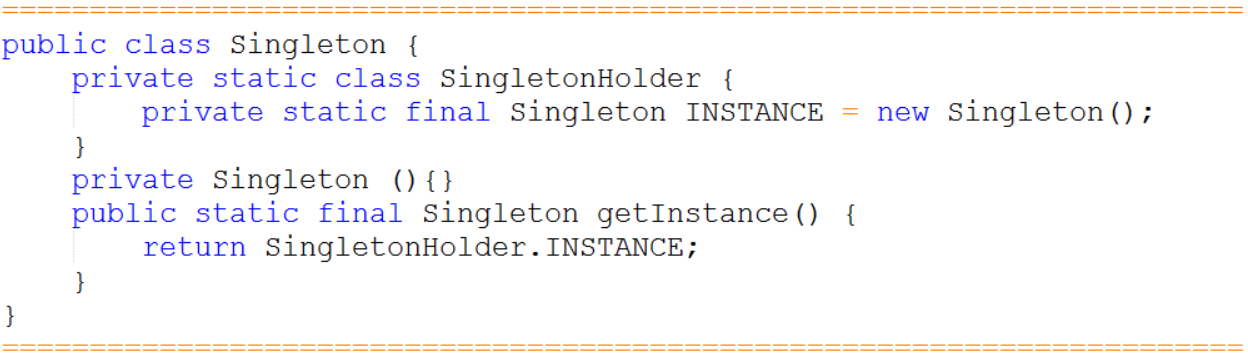
1. **双检锁 / 双重校验锁**

采用双锁机制，安全且在多线程情况下能保持高性能



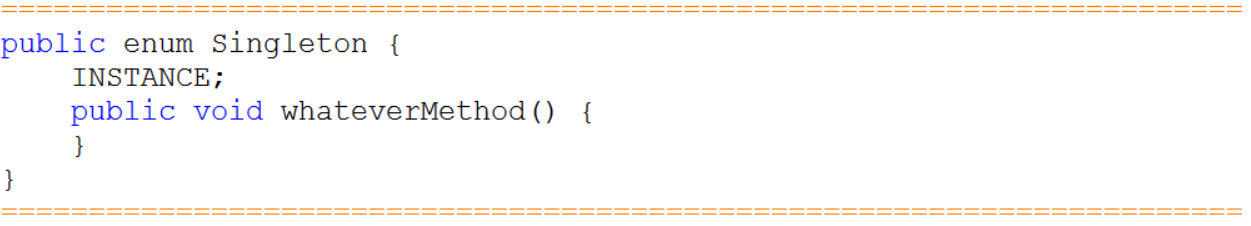
1. **登记式 / 静态内部类**

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 classloder 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，只有显示通过调用 getInstance 方法时，才会显示装载 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。



1. **枚举**

这种实现方式还没有被广泛采用，但这是实现单例模式的最佳方法。它更简洁，自动支持序列化机制，绝对防止多次实例化。这种方式是 Effective Java 作者 Josh Bloch 提倡的方式，它不仅能避免多线程同步问题，而且还自动支持序列化机制，防止反序列化重新创建新的对象，绝对防止多次实例化。不过，由于 JDK1.5 之后才加入 enum 特性，用这种方式写不免让人感觉生疏，在实际工作中，也很少用。不能通过 reflection attack 来调用私有构造方法。



一般情况下，不建议使用第 1 种和第 2 种懒汉方式，建议使用第 3 种饿汉方式。只有在要明确实现 lazy loading 效果时，才会使用第 5 种登记方式。如果涉及到反序列化创建对象时，可以尝试使用第 6 种枚举方式。如果有其他特殊的需求，可以考虑使用第 4 种双检锁方式。

观察者模式：

当对象间存在一对多关系时，可以使用观察者模式。比如，当一个对象被修改时，需要自动通知它的依赖对象，观察者模式属于行为型模式。

观察者模式定义了对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

优点：

观察者和被观察者是抽象耦合；建立了一套触发机制

缺点：

如果在观察者和被观察者之间有循环依赖，则会进行循环调用，可能导致系统崩溃；观察者模式没有相应的机制让观察者知道被观察者是怎么发生变化的，仅仅只是知道观察者发生了变化。

被观察者通知观察者调用的方法是在观察者中定义的。

观察者抽象基类：

