The Singleton Pattern is one of the simplest design patterns in Java. This type of design pattern is a creational pattern that provides an optimal way to create objects

**Notice:**

1. A singleton class can only have one instance.
2. A singleton class must create its own unique instance by itself.
3. A singleton class must provide this instance to all other objects.

**introduce**

**Intent**: To ensure that there is only one instance of a class, and to provide a global access point to it.

**The main solution**: a globally used class is frequently created and destroyed.

**When to use**: When you want to control the number of instances and save system resources.

**How to solve**: Determine whether the system already has this singleton, if so, return it, if not, create it.

**Keycode**: The constructor is private.

**advantage:**

1. There is only one instance in the memory, which reduces the memory overhead, especially the frequent creation and destruction of instances (such as the home page cache of the School of Management).
2. Avoid multiple occupations of resources (such as write file operations).

**Disadvantages**: No interface, no inheritance, conflicts with the principle of single responsibility, a class should only care about the internal logic, not how to instantiate it outside.

**scenes to be used**:

1. It is required to produce a unique serial number.
2. The counter in the WEB does not need to be added to the database every time it is refreshed but is cached first with a singleton.
3. An object created needs to consume too many resources, such as the connection between I/O and the database.

**Note**: The synchronization lock synchronized (Singleton. class) needs to be used in the getInstance() method to prevent multiple threads from entering at the same time and causing instances to be instantiated multiple times.

**accomplish**

We will create a single object class. The SingleObject class has its private constructor and a static instance of itself.

Step 2

Get a unique object from the singleton class.

Step 3

Execute the program and output the result:

Hello World!

**Several implementations of the singleton pattern**

There are several ways to implement the singleton pattern, as follows:

1. **Lazy, thread-unsafe**

**Whether to initialize Lazy**: yes

**Is it multi-thread safe**: no

**Difficulty to achieve**: easy

**Description**: This method is the most basic implementation. The biggest problem with this implementation is that it does not support multithreading. Because there is no lock synchronization, it is not strictly a singleton pattern.

This method of lazy loading is obvious, does not require thread safety, and does not work well in multi-threading.

The following implementations all support multi-threading but differ in performance.

**2. Lazy, thread-safe**

**Whether to initialize Lazy**: yes

**Is it multi-thread safe**: yes

**Difficulty to achieve**: easy

**Description**: This method has good lazy loading and can work well in multi-threading, but it is very inefficient and does not require synchronization in 99% of cases.  
Advantages: It is initialized only after the first call, avoiding memory waste.

**Disadvantage**: You must lock synchronized to ensure a singleton, but locking will affect efficiency.  
The performance of getInstance() is not critical to the application (this method is used less frequently).

**3. Hungry Chinese**

**Whether to initialize Lazy**: No

**Is it multi-thread safe**: yes

**Difficulty to achieve**: easy

**Description**: This method is more commonly used, but it is easy to generate garbage objects.

Advantages: There is no lock, and the execution efficiency will be improved.

Disadvantages: The class is initialized when the class is loaded, which wastes memory.

It avoids the synchronization problem of multiple threads based on the classloader mechanism. However, the instance is instantiated when the class is loaded. Although there are many reasons for class loading, most of them call the getInstance method in the singleton mode, but it is not certain. There are other ways (or other static methods) to cause class loading. At this time, the initialization instance obviously does not achieve the effect of lazy loading.

**4. Double-checked lock/double-checked lock (DCL, ie double-checked locking)**

**JDK version**: from JDK1.5

**Whether to initialize Lazy**: yes

**Is it multi-thread safe**: yes

**Difficulty to implement**: more complicated

**Description**: This method uses a double lock mechanism, which is safe and maintains high performance in multi-threaded situations.  
The performance of getInstance() is critical to the application.

**5.registration / static inner class**

**Whether to initialize Lazy**: yes

**Is it multi-thread safe**: yes

**Difficulty to achieve**: average

**Description**: This method can achieve the same effect as the double-check lock method, but the implementation is simpler. Use lazy initialization for static fields, which should be used instead of double-checked locking. This method is only applicable to the case of static fields, and the double-check lock method can be used when the instance field needs to be initialized lazily.  
This method also uses the classloader mechanism to ensure that there is only one thread when the instance is initialized. It is different from the third method: in the third method, as long as the Singleton class is loaded, the instance will be instantiated (without reaching lazy loading). effect), and in this way, the Singleton class is loaded, and the instance is not necessarily initialized. Because the SingletonHolder class is not actively used, only by explicitly calling the getInstance method, the SingletonHolder class will be explicitly loaded to instantiate the instance. Imagine if instantiating instance is very resource-intensive, so you want it to be loaded lazily. On the other hand, you don’t want to instantiate when the Singleton class is loaded, because you can’t ensure that the Singleton class may also be actively used in other places. Loading, then it is obviously inappropriate to instantiate instance at this time. At this time, this method is very reasonable compared to the third method.

**6. Enumeration**

**JDK version**: from JDK1.5

**Whether to initialize Lazy**: No

**Is it multi-thread safe**: yes

**Difficulty to achieve**: easy

**Description**: This implementation is not yet widely adopted, but it is the best way to implement the singleton pattern. It is more concise, automatically supports serialization mechanism, and definitely prevents multiple instantiations.  
This approach is advocated by Effective Java author Josh Bloch, which not only avoids multi-thread synchronization issues, but also automatically supports serialization mechanisms, preventing deserialization from recreating new objects, and absolutely preventing multiple instantiations. However, since the enum feature was added after JDK1.5, writing in this way makes people feel unfamiliar, and it is rarely used in practical work.  
Private constructors cannot be invoked through reflection attack.

**A word of experience**: In general, the 1st and 2nd lazy ways are not recommended, and the 3rd way of hungry is recommended. The 5th registration method is only used when the lazy loading effect is to be implemented explicitly. If it involves deserialization to create objects, you can try the sixth enumeration method. If there are other special requirements, you can consider using the fourth double-check locking method.

Singleton — this pattern is one of the most used patterns and its goal is to prevent instantiating the same class multiple times allowing only single instance across the given context. Please see my other [article](https://azeynalli1990.medium.com/singleton-design-pattern-in-java-and-in-spring-9fe3eb007af1) for more information about this pattern. Examples are:

* [java.lang.Runtime](https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/Runtime.html)
* [java.awt.Desktop](https://docs.oracle.com/en/java/javase/15/docs/api/java.desktop/java/awt/Desktop.html)