The Singleton Design Pattern is a software design pattern that addresses the problem of ensuring that a class has **only one instance** and provides a global point of access to that instance throughout an application.

Here are the key notes and concepts related to the Singleton Design Pattern:

### . ****Structure (Class Diagram)****

lua

Copy code

+----------------+

| Singleton |

+----------------+

| - instance | <-- static

| - Singleton() | <-- private constructor

+----------------+

| + getInstance()| <-- static method

+----------------+

**Use Cases in Real World**

| **Use Case** | **Why Singleton?** |
| --- | --- |
| **Logger** | Only one logger to manage logs consistently |
| **Configuration Manager** | Global access to app settings |
| **Database Connection Pool** | Reuse a single connection factory |
| **Cache Manager** | Maintain centralized cache |

* **Problem Solved by Singleton Design Pattern**
  + It addresses the need for **only one instance of a class or a shared resource** across an entire application.
  + This single instance needs to be accessible from all parts of the application, but new instances should not be created every time it's needed.
  + Examples of such shared resources include a **database connection**, a **logger instance**, or the Runtime class in Java.
* **Why Global Variables Are Not Sufficient**
  + One might consider using a global variable to store the instance and allow access.
  + However, if a global variable (like a logger instance) is used, different application modules can read it, but there's a risk of **one module overwriting or modifying its value**.
  + If the value changes, it's no longer the consistent shared instance needed by all resources, which is why global variables alone are not suitable for this problem.
* **How Singleton Protects the Instance**
  + To protect the variable, the instance is initialized using a **private constructor**, which prevents external classes from directly creating new instances.
  + Access to the instance is *only* allowed through a **public getter function**.
  + **Class Diagram/Core Principles**:
    - A Singleton class contains just **one, private instance** of itself.
    - The **constructor** that initializes this instance is also **private**.
    - The only **public** component is a **getter function**, which returns the single instance.
    - This structure ensures that all application modules access the *same* instance and cannot modify it directly, only retrieve it.
* **Analogy: Prime Minister's Calendar**
  + A good real-world analogy is a **Prime Minister's calendar**.
  + There's only one Prime Minister and one calendar.
  + Everyone needs access to schedule meetings, but giving direct access would allow people to override it.
  + Instead, the Prime Minister's **personal assistant (PA)** has direct access to the calendar, and everyone else interacts with the PA (who acts like the **getter function**) to schedule slots.
  + This way, the calendar is a shared global resource that cannot be directly modified by others; they can only read from it or request changes via the PA.
* **Important Violation/Avoidance**
  + A singleton class **should never accept a parameter**.
  + If it does, it **violates the singleton pattern** and essentially becomes a **factory pattern**. This is considered an "empty pattern".
* **Eager vs. Lazy Loading**
  + **Eager Loading**:
    - Means the singleton instance is **initialized immediately** when the application starts.
    - The instance is always ready to be returned.
    - *Use Case*: If your application has just **one singleton class**, eager loading might be acceptable.
    - *Drawback*: If the instance is initialized but **never used**, it will still consume memory unnecessarily.
    - **Code Example (Conceptual - Eager Loading)**:
    - public class LoggerSingleton {
    - // Instance gets initialized when the class is loaded
    - private static LoggerSingleton instance = new LoggerSingleton(); // Eagerly initialized
    - private LoggerSingleton() {
    - // Private constructor
    - }
    - public static LoggerSingleton getInstance() {
    - return instance; // Returns the pre-initialized instance
    - }
    - }

*When running the demo, loggerSingleton.getInstance() calls would*

*return the same object, confirming it's the same instance.*

* + **Lazy Loading**:
    - Means the singleton instance is **not initialized** when the application starts.
    - It is initialized **only when someone first requests or invokes it**.
    - *Benefit*: **Saves memory** if the instance is not immediately needed or if there are multiple singleton classes.
    - *Mechanism*: When the getInstance() method is called for the first time, it checks if the instance is null. If it is, the instance is created; otherwise, the existing instance is returned.
    - **Code Example (Conceptual - Lazy Loading)**:
    - public class LoggerSingleton {
    - // Instance is initially null
    - private static LoggerSingleton instance = null;
    - private LoggerSingleton() {
    - // Private constructor
    - }
    - public static LoggerSingleton getInstance() {
    - if (instance == null) { // Check if instance is null
    - instance = new LoggerSingleton(); // Initialize only if null
    - }
    - return instance; // Return the instance
    - }
    - }

*When running the demo, even with lazy loading, subsequent calls to getInstance() would return the same object.*

* **Thread Safety in Singleton Implementation**
  + A basic singleton implementation (like the eager or lazy loading examples above) is **not thread safe**.
  + If multiple threads try to access and initialize the singleton concurrently, it can lead to issues, especially in lazy loading where multiple instances might be created.
  + **Mechanisms for Thread Safety**:
    - **volatile keyword**: The instance variable is often declared volatile. This ensures that changes to the variable are immediately visible across different threads.
    - public class LoggerSingleton {
    - // Declared as volatile
    - private static volatile LoggerSingleton instance;
    - private LoggerSingleton() {
    - // Private constructor
    - }
    - // ... rest of the class
    - }
    - **Preventing Reflection Attacks**: The private constructor can be vulnerable to instantiation via **reflection**. To prevent this, a check can be added inside the constructor to throw a RuntimeException if an instance already exists.
    - public class LoggerSingleton {
    - // ... volatile instance declaration
    - private LoggerSingleton() {
    - // Check to prevent instantiation via reflection
    - if (instance != null) {
    - throw new RuntimeException("Use getInstance() method to get the single instance.");
    - }
    - }
    - // ... rest of the class
    - }
    - **synchronized keyword (Double-Checked Locking)**: To ensure only one thread initializes the instance, the synchronized keyword is used. It's typically applied at the **block level** (inside the getInstance() method) rather than the class level to avoid blocking unnecessary operations. A **double check** for nullability is performed:
      1. First check: if (instance == null) (outside the synchronized block) for performance.
      2. Second check: if (instance == null) (inside the synchronized block) to ensure only one thread initializes.
    - public class LoggerSingleton {
    - private static volatile LoggerSingleton instance;
    - private LoggerSingleton() {
    - if (instance != null) {
    - throw new RuntimeException("Use getInstance() method to get the single instance.");
    - }
    - }
    - public static LoggerSingleton getInstance() {
    - if (instance == null) { // First null check (outside synchronized block)
    - // Synchronize the block for initialization
    - synchronized (LoggerSingleton.class) { // Using class lock
    - if (instance == null) { // Second null check (inside synchronized block)
    - instance = new LoggerSingleton();
    - }
    - }
    - }
    - return instance;
    - }
    - }

*This double-checked locking mechanism ensures that only one thread initializes the instance while others wait, and once initialized, subsequent calls don't incur the overhead of the synchronized block. When tested, this still returns the same object for all calls.*

* **Pros of Singleton Design Pattern**
  + **Neat and Clean**: Provides a structured way to manage access to shared global resources.
  + **Easy to Implement**: Relatively straightforward to code.
  + **Guaranteed Single Instance**: Ensures that only one instance of the class exists.
  + **Solves a Well-Defined Problem**: Addresses the specific need for a unique, globally accessible instance.
* **Cons of Singleton Design Pattern**
  + **Overuse/Abuse**: Developers sometimes overuse or misapply the pattern, even when not strictly necessary. It's crucial to evaluate if the resource *truly* needs to be a global singleton.
  + **Confusing with Factory Pattern**: Never use it with parameters; doing so makes it a factory and violates the pattern.
  + **Hard to Test**: Can be challenging to write unit tests for singleton classes.
  + **Danger without Thread Safety**: If thread safety mechanisms are forgotten, multiple threads accessing the singleton can lead to **inconsistent behavior** or multiple instances, making it dangerous for the application. This point is very important to remember.

**Anti-Patterns / Gotchas**

* Breaking Singleton with **Reflection** or **Cloning**
* Overuse = **Global State Hell**
* Hidden **tight coupling** and **testing difficulties**