Singleton

Montag, 26. August 2024 10:30

Author: Damir Ljubic
e-mail: damirlj@yahoo.com

Introduction

Singleton belongs to **the *creational patterns***.

It's quite notorious one - I would dare to say: almost as the goto statement
(although, the old enough readers probably remember the Basic - procedural language - where the only way
to control the program flow was via goto statements).

@Disclaimer For those who are already screaming "Man, you are so retro - this is from GoF time: *It's ANTI pattern now!*" -
just because they heard that on radio - I'm hearing you.
Could we still agree on one thing - before we label <Something> as ANTI, shouldn't we first try to understand what <Something> is?
I'm not advocate here anything - I'll try to bring some light into this topic, and describe the concept and mechanics behind.

Essentially, it's similar to using the global static variable which is strongly discouraged, for various reasons.
It's actually initializing the static variable within local - block scope **once**, in a thread-safe way, and
use it "*everywhere*" - across the entire program.
But - these are implementation details.

So, Singleton is not about having the single instance of the class in a program - but rather being prohibited by design
to produce multiple instances of the same type.
For example, **Inversion Control Containers (ICC)** - is the way to hold either a single instances of the dependency modules,
or to provide the API for obtaining the new instances from container on each and every request.
Holding the singletons wouldn't be able to satisfy such requirement.

<**More on that**>: https://github.com/damirlj/modern_cpp_tutorials/tree/main?tab=readme-ov-file#tut5
@note This is not chosen by accident - If you rethink - this is a valid alternative for Singleton design (anti) pattern

In this article, I've briefly mentioned Dependency Injection Frameworks (like *Dogger* - Java, or *Kotlin injection* in Kotlin), without
going into details.
These two use annotations, unlike the **Koin** - which is DSL based.
What we described as ICC containers, we can easily model that with Koin as

```
interface DatabaseEngine {...}
class SQLDatabaseEngine : DatabaseEngine {...}

class DatabaseService(private val databaseEngine: DatabaseEngine, private val database : String) {
 fun connect() : Int {...}
 ...
}

val databaseServiceDependencies = module {
    single { SQLDatabaseEngine() }
    factory { (database :String) -> DatabaseService(get(), database)}
}
```

We specify the dependency as **singleton** - by replacing the interface with concreate implementation at single place (customization point).
For dependent object *DatabaseService*- we specify the *factory* method: on each instantiation, a new instance
of the *DatabaseService* will be created

```
class A {
    private val ordersService: DatabaseService("ordersDB") by inject()
    private val usersService: DatabaseService("usersDB") by inject()
    ...
}
```

If this required by design, we can specify for the instantiation of dependent object - to be singleton as well

```
val parserDependencies = module {
    single { JSONFormat() }
    single { Parser(get()) }
}
```

Where to use singletons?

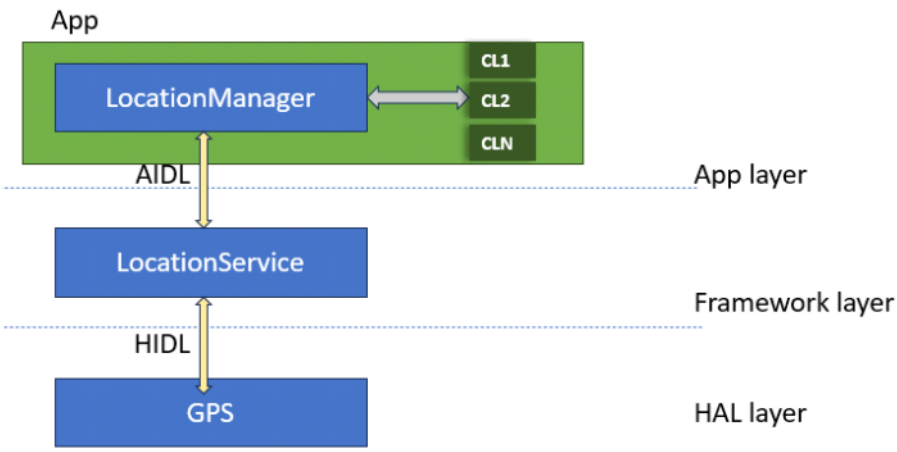
Whenever there is a need for a single - centralized component for managing certain resources: for performing a particular job (SRP - Single Responsibility Principle) and be accessible across the entire code - as static storage class which lifetime is bounded to the lifetime of the program.

Ok - I admit, this sounds a little bit foggy.

For those who work on Android, it's common knowledge that Android framework is layered one.
The lowest layer is known as HAL (Hardware Abstraction Layer) - and it's mainly about vendor-specific driver implementation in native (C/C++) language.
On top of it is Middleware - which consists mainly on the system services - that directly speaks with the HAL (HIDL/AIDL interfaces).
For each system service - there is a matching (1:1) Manager: a through client to the service.
The Manager (like LocationManager, CarPropertyManager, etc.) is actually internally implemented as a singleton (per process: not system-wide) - that can be obtained at the topmost Application layer as

```
(LocationManager) getSystemService(Context.LOCATION_SERVICE);
```

In the context of the process - it always returns the very same instance of the Manager.
Manager offers the listener: callbacks interface - for all different clients within the same application, to subscribe (1:N) to the very same Manager - and indirectly, to the single Service instance that directly speaks with HAL layer.
These are internal details for the Publisher-Subscriber mechanism that is used for this inter-layers communication.



Implementation

The Singleton design pattern relies on two premises:

- Having a **private** constructor: to control the fabrication of the type
- Providing the **single instance** - **with static storage duration**

This becomes quite obvious in Java implementation

```
private enum Singleton implements Supplier<T> {
    INSTANCE;

    private final T instance;

    Singleton() {
        instance = new T();
    }

    @Override
    public T get() {
        return instance;
    }
}

private T() {}
public static T get() { return Singleton.INSTANCE.get(); }
```

where T is enclosing type - that will be turned into singleton.

Enum class in Java is a perfect choice since:

- Each enumeration constant is static instantiation of the enum class which is **thread-safe** (INSTANCE == static Singleton)
- The enum constructor is by design private

This way, calling *T.get()* will retrieve the single instance of T, by instantiate at the very first call - in a thread-safe manner, the static instance of enum class Singleton (INSTANCE) - calling the private constructor Singleton() - which will internally call the private constructor of the enclosing class T - for which we want to behave as actual singleton.

C++ implementation

The very same effect we can achieve in C++ using classical Scott Meyers' singleton implementation

```
// Singleton by design
class Singleton : non_copyable
{
    public:
        [[nodiscard]] static Singleton& get()
        {
            static Singleton instance{};// static variable within local scope: thread-safe instantiation
            return instance;
        }


    private:
        Singleton() = default;

};
```

But what if we want to be more deterministic on the singleton lifetime management, when there is a need for a proper resource release - where the order of destruction plays the role?
This is not the issue in Java/Kotlin - since there is a GC involved: there is no destructor and RAII concept

We need to extend the interface with additional API - to fulfill this requirement.
Possible implementation can be

```
// Using factory method - to create a singleton on the stack and return
// the (shared) pointer on it.
// This expensive approach would be justifiable only in case that we want to have a control
// over the singleton lifetime management - in case of custom-specific destruction,
// when acquired resources has to be released, or destruction has to be done in a specific order
friend auto factory::createOnStackPtr<Singleton>();// since constructor is private
[[nodiscard]] static std::shared_ptr<Singleton> getPtr()
{
    static std::shared_ptr<Singleton> instance = factory::createOnStack<Singleton>();
    return instance;
}
```

Where one possible implementation of stack allocation could be

```
namespace factory
{
    /**
    * Factory method
    *
    * Creating object as instance of a T on the stack.
    * It relies on the static instantiation that is thread-safe.
    * It will use pre-allocated memory on the stack (std::array),
    * using placement new for allocation
    */
    template <typename T, typename...Args>
    auto createOnStackPtr(Args&&...args)
    {
        using stack_memory = std::array<std::byte, sizeof(T)>;
        static stack_memory stack{};
        #if __cplusplus >= 201703L
            static std::pmr::monotonic_buffer_resource buf {stack.data(), stack.size()};
            return new(&buf)T {std::forward<Args>(args)...};
        #else
            return new(stack.data())T {std::forward<Args>(args)...};
        #endif
    }

    template <typename T>
    void deleteOnStackPtr(T*& ptr)
```

```
    {
        if (ptr)[[likely]]
        {
            ptr->~T();
            ptr = nullptr;
        }
    }

    // Control the life-time of the on the stack allocated pointer
    template <typename T, typename...Args>
    auto createOnStack(Args&&...args)
    {
        static auto* ptr = createOnStackPtr<T>(std::forward<Args>(args)...);

        auto deleter = [](T* p) { deleteOnStackPtr(p); }; // deleter embedded into type
        return std::unique_ptr<T, decltype(deleter)> {ptr, deleter};
    }
}
```

⇒ For more generic implementation of the **Factory Method**
https://github.com/damirlj/modern_cpp_tutorials/blob/main/docs/desing%20patterns/Factory%20method.pdf

Singleton helper

We can isolate the second premise - having single static instance of the parameterized type,
into separate helper utility class - this time using the *std::call_once*
(no advantage in compare with Scott Meyers' singleton - took as demonstration only)

```
// For turning any class into "singleton"" - ensuring it's called once

template <typename T>
requires std::is_default_constructible_v<T>
class SingletonHelper
{
    static inline std::once_flag once {};
    static inline T* instance = nullptr;
    public:
         static T& get()
         {
             std::call_once(once, []{ instance = factory::createOnStackPtr<T>();} );
             return *instance;
         }
};
```

Well, this will work with to limitations:
- T must have public constructor
- T must be default-constructible

We can overcome the second limitation by separating the initialization from getter method, which result in introducing another mutex, so that getter method is once again thread-safe.
For client code - this imposes the order in which methods can be invoked (*init*() call must precede the *get*() call)

```
template <typename T>
class SingletonHelper
{
    static inline std::once_flag once_ {};
    static inline T* instance_ = nullptr;
    static inline std::mutex lock_{};

    public:
        static T& get()
        {
            std::lock_guard lock {lock_};
            if (instance_ == nullptr) throw std::logic_error("<SingletonHelper> T is not instantiated");
            return *instance_;
        }

        // Separating the creation - instantiation from retrieving the instance (getter)
        // violates the thread-safeness
        // For that - we need to introduce additional mutex
        template<typename...Args>
        static void init(Args&&...args)
        {
            std::call_once(once_,
                [&]
```

```
                {
                    instance_ = factory::createOnStackPtr<T>(std::forward<Args>(args)...);
                });
        }
};
```

To overcome the first limitation, turning the parameterized type constructor into *private* - we need additionally to **couple** the factory method we used to produce a single instance on the stack- in SingletonHelper class - with parameterized type declaration

```
template <typename T>
requires std::is_arithmetic_v<T>
class A : non_copyable
{
    private:
       friend A* factory::createOnStackPtr<A>(value_type&&);
       explicit A(value_type val) noexcept: value_(val) {}
       ...
    private:
        value_type value_;
};
```

Turning A into singleton becomes now quite easy

```
using int_type = A<int>;
using singleton_type = SingletonHelper<int_type>;
```

<**Entire code**>: https://godbolt.org/z/W8r7dWaor