**Singleton**

Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

The Singleton pattern solves two problems at the same time, violating the Single Responsibility Principle:

1. Ensure that a class has just a single instance. Why would anyone want to control how many instances a class has? The most common reason for this is to control access to some shared resource - for example, a database or a file.

Here’s how it works: imagine that you created an object, but after a while decided to create a new one. Instead of receiving a fresh object, you’ll get the one you already created.

Note that this behavior is impossible to implement with a regular constructor since a constructor call must always return a new object by design.

2. Provide a global access point to that instance. Remember those global variables that you (all right, me) used to store some essential objects? While they’re very handy, they’re also very unsafe since any code can potentially overwrite the contents of those variables and crash the app.

Just like a global variable, the Singleton pattern lets you access some object from anywhere in the program. However, it also protects that instance from being overwritten by other code.

There’s another side to this problem: you don’t want the code that solves problem #1 to be scattered all over your program. It’s much better to have it within one class, especially if the rest of your code already depends on it.

Nowadays, the Singleton pattern has become so popular that people may call something a singleton even if it solves just one of the listed problems.

**Solution**

All implementations of the Singleton have these two steps in common:

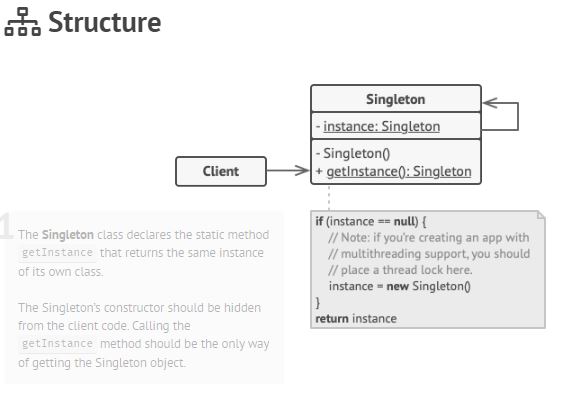
Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.

Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.

If your code has access to the Singleton class, then it is able to call the Singleton’s static method. So whenever that method is called, the same object is always returned.

**Real-World Analogy**

The government is an excellent example of the Singleton pattern. A country can have only one official government. Regardless of the personal identities of the individuals who form governments, the title, “The Government of X”, is a global point of access that identifies the group of people in charge.



**Pseudocode**

In this example, the database connection class acts as a Singleton. This class does not have a public constructor, so the only way to get its object is to call the getInstance method. This method caches the first created object and returns it in all subsequent calls.

// The Database class defines the `getInstance` method that allows clients access

// the same instance of a database connection throughout the program.

**class** **Database** **is**

// The field for storing the Singleton instance is declared static.

**private** **static** **field** instance: Database

// The Singleton's constructor is always declared private to

// prevent direct construction calls with the `new` operator.

**private** **constructor** Database() **is**

// Some initialization code, such as the actual

// connection to a database server.

// ...

// The static method that controls access to the Singleton instance.

**public** **static** **method** getInstance() **is**

**if** (Database.instance == **null**) **then**

acquireThreadLock() **and** **then**

// Ensure that the instance has not yet been initialized by another

// thread while this one has been waiting for the lock's release.

**if** (Database.instance == **null**) **then**

Database.instance = **new** Database()

**return** Database.instance

// Finally, any Singleton should define some business logic

// which can be executed on its instance.

**public** **method** query(sql) **is**

// For instance, all database queries of an app go through this method.

// Therefore, you can place throttling or caching logic here.

// ...

**class** **Application** **is**

**method** main() **is**

Database foo = Database.getInstance()

foo.query("SELECT ...")

// ...

// The variable `bar` will contain the same object as the variable `foo`

Database bar = Database.getInstance()

bar.query("SELECT ...")

**Applicability**

Use the Singleton pattern when a class in your program should have just a single instance available to all clients. For example, a single database object shared by different parts of the program.

The Singleton pattern disables all other means of creating objects of a class except for the special creation method. This method either creates a new object or returns an existing one if it has already been created.

Use the Singleton pattern when you need stricter control over global variables.

Unlike global variables, the Singleton pattern guarantees that there is just one instance of a class. Nothing, except for the Singleton class itself, can replace the cached instance.

Note that you can always adjust this limitation and allow creating any number of Singleton instances. The only piece of code that needs changing is the body of the getInstance method.

**Pros**

You can be sure that a class has only a single instance.

You gain a global access point to that instance.

The singleton object is initialized only when it is requested for the first time.

**Cons**

Violates the Single Responsibility Principle. The pattern solves two problems at the time.

The Singleton pattern can mask bad design, for instance, when the components of the program know too much about each other.

The pattern requires special treatment in a multithreaded environment so that multiple threads will not create a singleton object several times.

It may be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance when producing mock objects. Since the constructor of the singleton class is private and overriding static methods is impossible in most languages, you will need to think of a creative way to mock the singleton. Or just do not write the tests. Or do not use the Singleton pattern.

**Relations with Other Patterns**

A **Facade** class can often be transformed into a **Singleton** since a single facade object is sufficient in most cases.

**Flyweight** would resemble **Singleton** if you somehow managed to reduce all shared states of the objects to just one flyweight object. But there are two fundamental differences between these patterns:

1. There should be only one Singleton instance, whereas a Flyweight class can have multiple instances with different intrinsic states.

2. The Singleton object can be mutable. Flyweight objects are immutable.

**Abstract Factories**, **Builders** and **Prototypes** can all be implemented as **Singletons**.

**Singleton in C++**

**Singleton** is a creational design pattern, which ensures that only one object of its kind exists and provides a single point of access to it for any other code.

Singleton has almost the same pros and cons as global variables. Although they are super-handy, they break the modularity of your code.

You cannot just use a class that depends on a Singleton in some other context, without carrying over the Singleton to the other context. Most of the time, this limitation comes up during the creation of unit tests.

**Usage examples**: A lot of developers consider the Singleton pattern an antipattern. That is why its usage is on the decline in C++ code.

**Identification**: Singleton can be recognized by a static creation method, which returns the same cached object.

**Naïve Singleton**

It is pretty easy to implement a sloppy Singleton. You just need to hide the constructor and implement a static creation method.

The same class behaves incorrectly in a multithreaded environment. Multiple threads can call the creation method simultaneously and get several instances of Singleton class.

<https://refactoring.guru/design-patterns/singleton>