**Singletons**

#### **Intent**

**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.

#### **Problems**

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.

| ***Notes:*** *Clients may not even realize that they’re working with the same object all the time.* |
| --- |

1. **Provide a global access point to that instance.**

Remember those global variables that are 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.

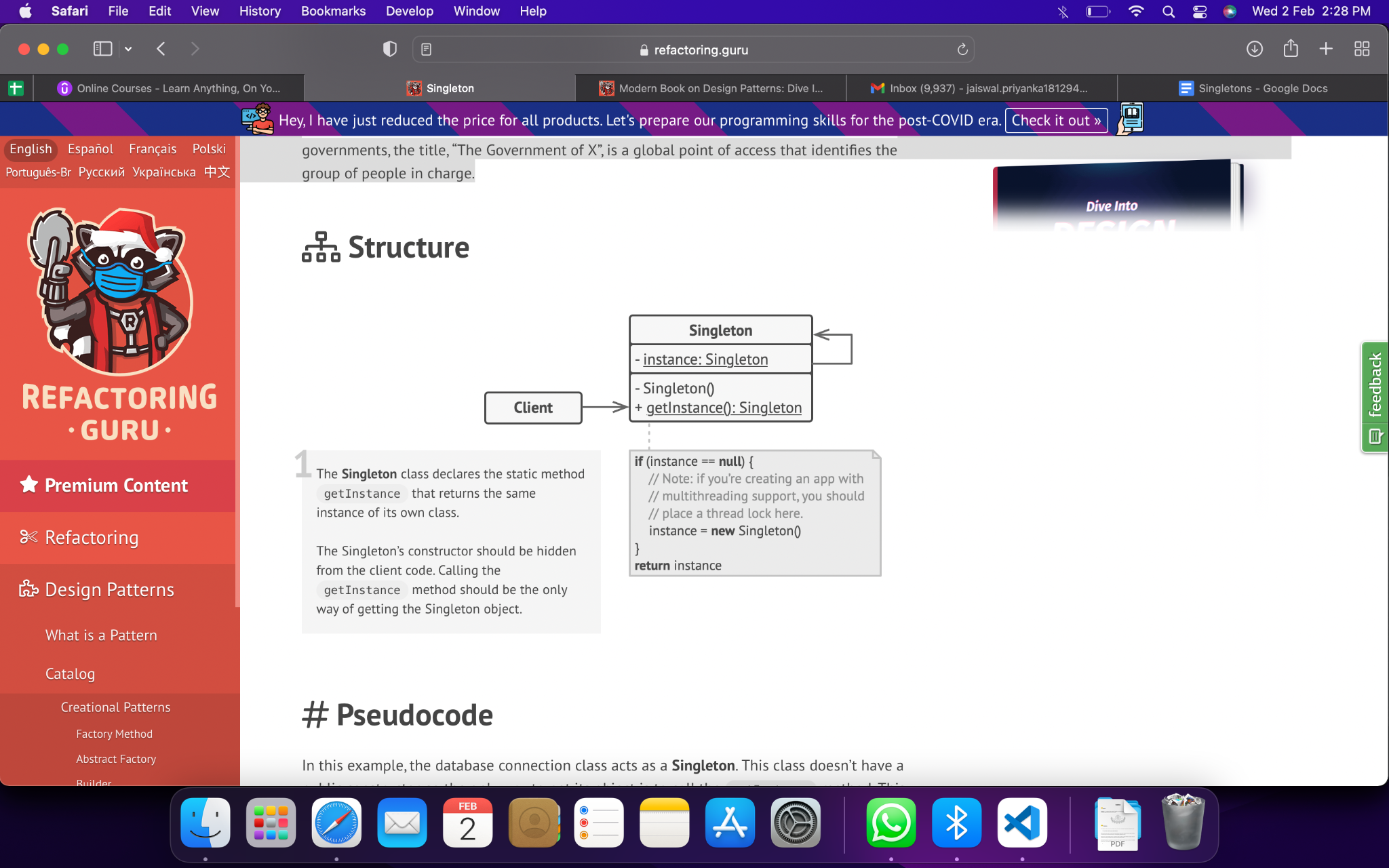
#### **Solution**

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’s able to call the Singleton’s static method. So whenever that method is called, the same object is always returned.

#### **Diagrammatic Representation**



#### **Checklist**

1. Add a private static field to the class for storing the singleton instance.
2. Declare a public static creation method for getting the singleton instance.
3. Implement “lazy initialization” inside the static method. It should create a new object on its first call and put it into the static field. The method should always return that instance on all subsequent calls.
4. Make the constructor of the class private. The static method of the class will still be able to call the constructor, but not the other objects.
5. In the client code, replace all direct calls to the singleton’s constructor with calls to its static creation method.

#### **Pros & Cons**

| You can be sure that a class has only a single instance. | Violates the *Single Responsibility Principle*. |
| --- | --- |
| You gain a global access point to that instance. | The Singleton pattern can mask bad design, for instance, when the components of the program know too much about each other. |
| The singleton object is initialized only when it’s requested for the first time. | The pattern requires special treatment in a multithreaded environment so that multiple threads won’t 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 don’t write the tests. Or don’t use the Singleton pattern. |

#### **Applicability**

1. 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.
2. Use the Singleton pattern when you need stricter control over global variables.

#### **Relation with Other Patterns**

1. **Facade** class can often be transformed into a **Singleton** since a single facade object is sufficient in most cases.
2. **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 (Objects whose value can change are said to be mutable). Flyweight objects are immutable (Objects whose value is unchangeable once they are created are called immutable).
3. **Abstract Factories**, **Builders** and **Prototypes** can all be implemented as **Singletons**.

#### **References**

1. “Singleton.” *Refactoring.Guru*, https://refactoring.guru/design-patterns/singleton.