Design pattern

Hadis Ghafouri, Hoora Mahmoudian, Mehrana Kalagari

What's a design pattern?

- Typical Solutions To Commonly Occurring Problems In Software Design.
- They Are Like Pre-made Blueprints That You Can Customize To Solve A Recurring Design Problem In Your Code.
- You Can't Just Find A Pattern And Copy It Into Your Program, The Way You Can With Off-the-shelf Functions Or Libraries.
- The Pattern Is Not A Specific Piece Of Code, But A General Concept For Solving A Particular Problem.
- You Can Follow The Pattern Details And Implement A Solution That Suits The Realities Of Your Own Program.
- Patterns Are Often Confused With Algorithms, Because Both Concepts Describe Typical Solutions To Some Known Problems.
- An Algorithm Always Defines A Clear Set Of Actions That Can Achieve Some Goal.
- A Pattern Is A More High-level Description Of A Solution.
- The Code Of The Same Pattern Applied To Two Different Programs May Be Different.
- An Analogy To An Algorithm Is A Cooking Recipe: Both Have Clear Steps To Achieve A Goal.
- A Pattern Is More Like A Blueprint: You Can See What The Result And Its Features Are, But The Exact Order Of Implementation Is Up To You.

What does the pattern consist of?

- Most patterns are described very formally so people can reproduce them in many contexts.
- Here are the sections that are usually present in a pattern description:
- **Intent of the pattern** briefly describes both the problem and the solution.
- **Motivation** further explains the problem and the solution the pattern makes possible.
- **Structure of classes** shows each part of the pattern and how they are related.
- **Code example** in one of the popular programming languages makes it easier to grasp the idea behind the pattern.

History of patterns

- **Who invented patterns?**
- Patterns are typical solutions to common problems in object-oriented design.
- When a solution gets repeated over and over in various projects, someone eventually puts a name to it and describes the solution in detail. That's basically how a pattern gets discovered.
- The idea was picked up by four authors: Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm.
- In 1994, they published Design Patterns: Elements of Reusable Object-Oriented Software, in which they applied the concept of design patterns to programming.
- The book featured 23 patterns solving various problems of object-oriented design and became a best-seller very quickly.
- Due to its lengthy name, people started to call it "the book by the gang of four" which was soon shortened to simply "the GoF book".
- Since then, dozens of other object-oriented patterns have been discovered. The "pattern approach" became very popular in other programming fields, so lots of other patterns now exist outside of object-oriented design as well.

Why should I learn patterns?

- Design patterns are a toolkit of **tried and tested solutions** to common problems in software design.
- Even if you never encounter these problems, knowing patterns is still useful because it teaches you how to solve all sorts of problems using principles of object-oriented design.
- Design patterns define a common language that you and your teammates can use to communicate more efficiently.
- You can say, "Oh, just use a Singleton for that," and everyone will understand the idea behind your suggestion.
- No need to explain what a singleton is if you know the pattern and its name.

Classification of patterns

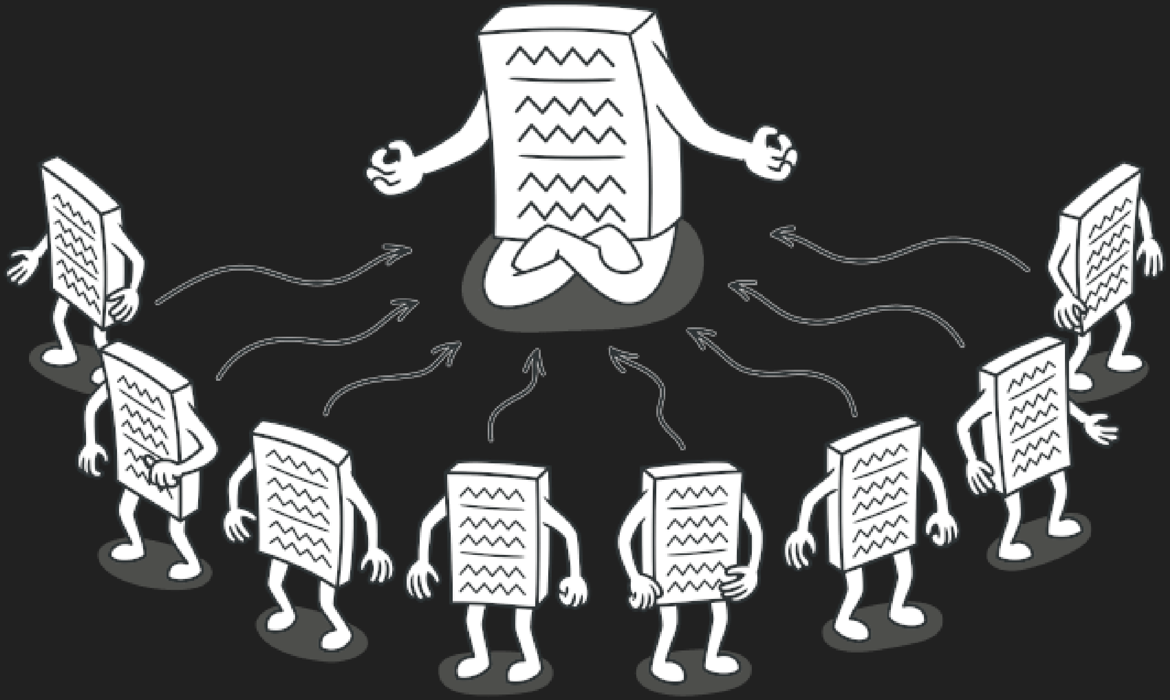
- Design patterns differ by their complexity, level of detail and scale of applicability to the entire system being designed.
- The most basic and low-level patterns are often called *idioms*. They usually apply only to a single programming language.
- The most universal and high-level patterns are *architectural patterns*. Developers can implement these patterns in virtually any language. Unlike other patterns, they can be used to design the architecture of an entire application.
- Three main groups of patterns:
- **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
- **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.
- **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

Singleton Pattern

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.
2. Provide a global access point to that instance.



Ensure that a class has just a single instance.

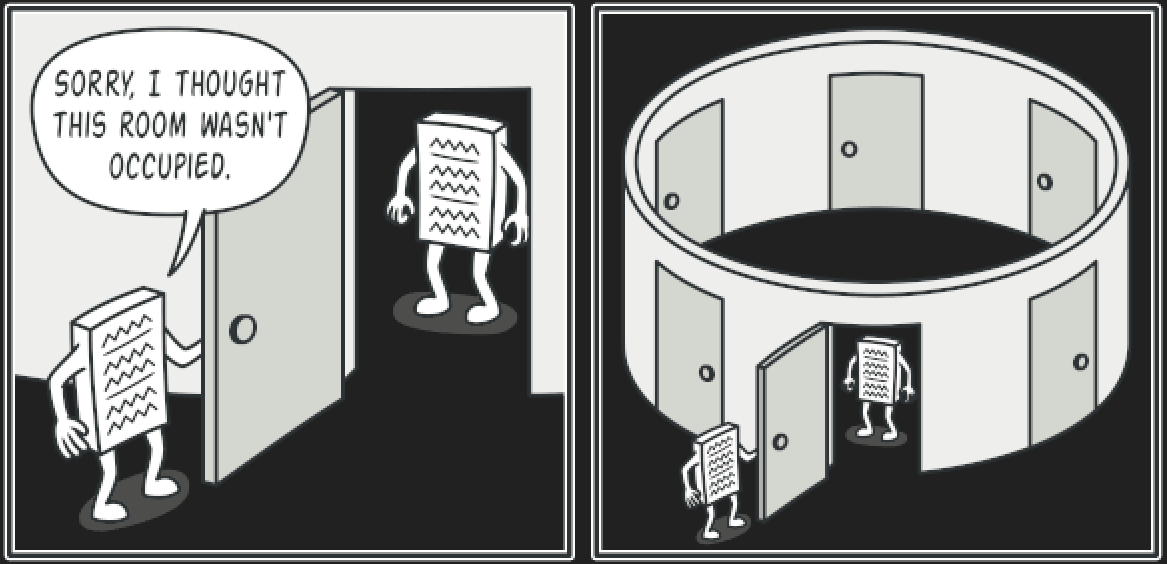
Why would anyone want to control how many instances a class has?

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.



Provide a global access point to that instance.

- Remember those global variables that you 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.

Solution

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

1. Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.
2. 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.

Real-World Analogy

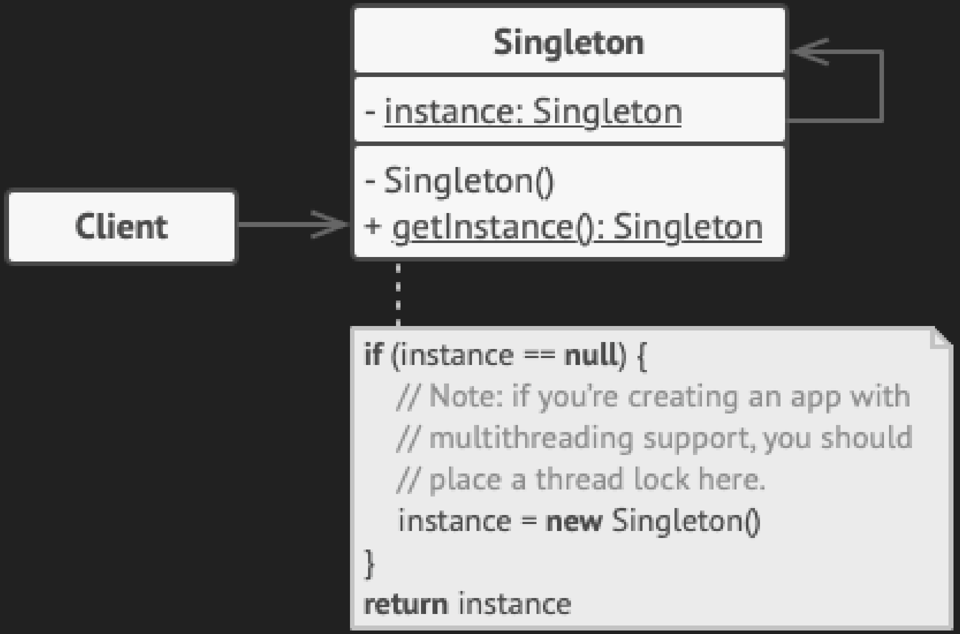
The government is an excellent example of the Singleton pattern.

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.

Structure

- The Singleton class declares the static method getInstance that returns the same instance of its own class.
- The Singleton's constructor should be hidden from the client code.
- Calling the getInstance method should be the only way of getting the Singleton object.



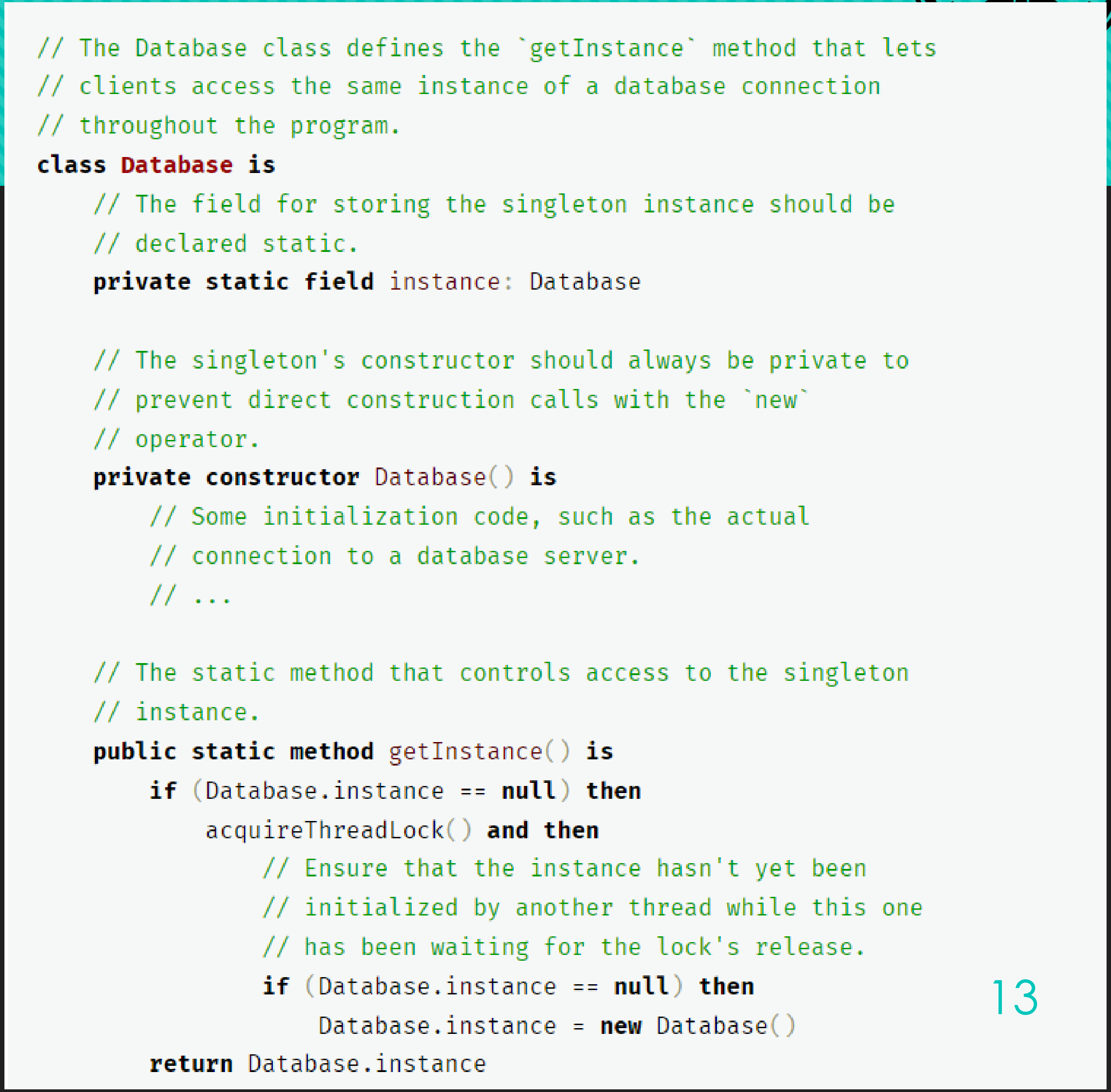
```
if (instance == null) {
  // Note: if you're creating an app with
  // multithreading support, you should
  // place a thread lock here.
  instance = new Singleton()
}
return instance
```

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's 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.

Pseudocode

- In this example, the database connection class acts as a Singleton.
- This class doesn’t 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 lets
// clients access the same instance of a database connection
// throughout the program.
class Database is
    // The field for storing the singleton instance should be
    // declared static.
    private static field instance: Database

    // The singleton's constructor should always be 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 hasn't 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
```

Disadvantages of Singleton

- Violates the *Single Responsibility Principle*.

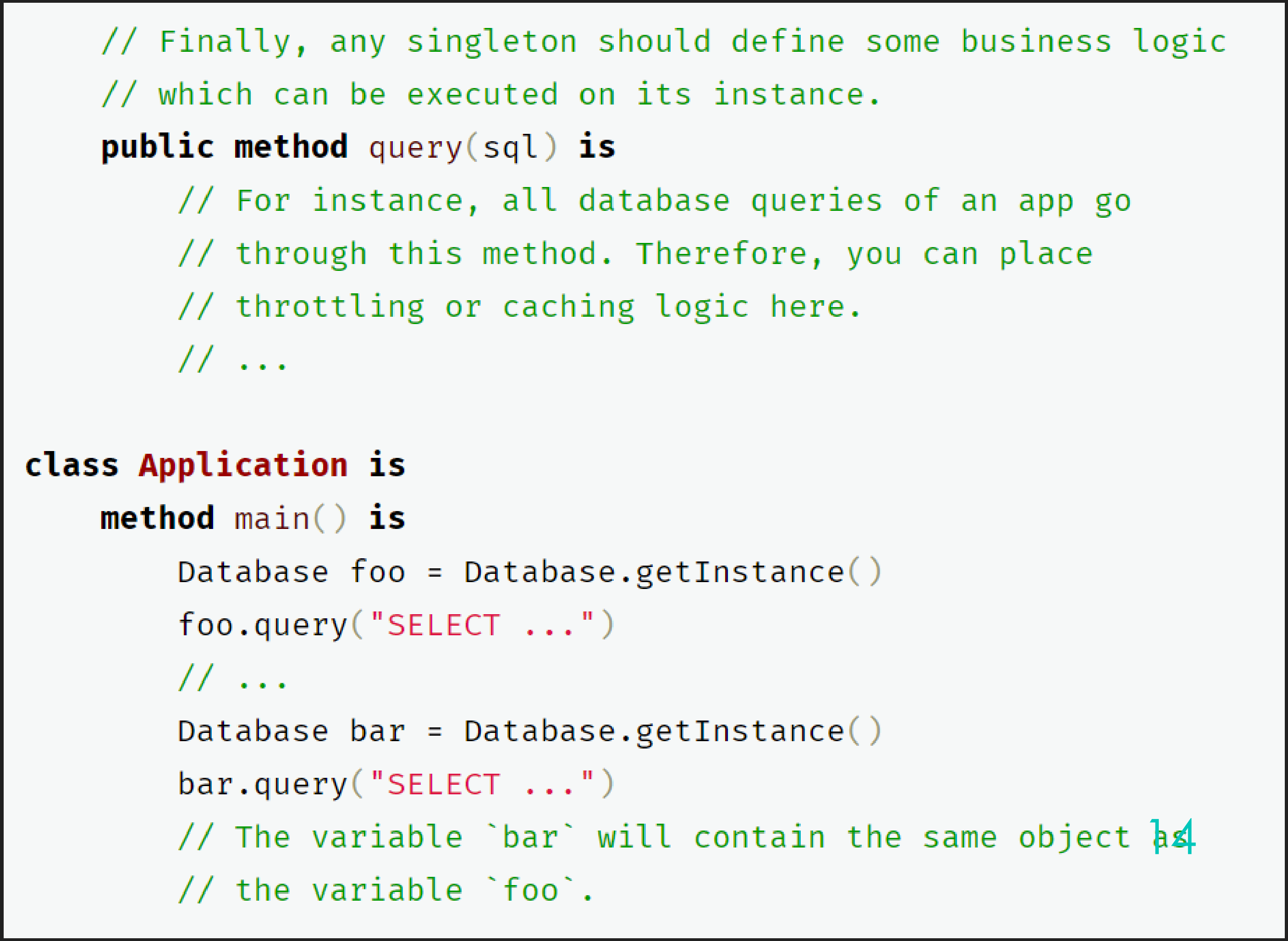
The pattern solves two problems at the time.

- 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.



```
    // 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 ...")
        // ...
        Database bar = Database.getInstance()
        bar.query("SELECT ...")
        // The variable `bar` will contain the same object as
        // the variable `foo`.
```

References

- https://refactoring.guru/design-patterns