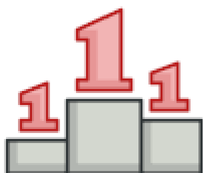
Содержание

Singleton ... 3
Проблема ... 3
Структура ... 4
Примеры ... 5

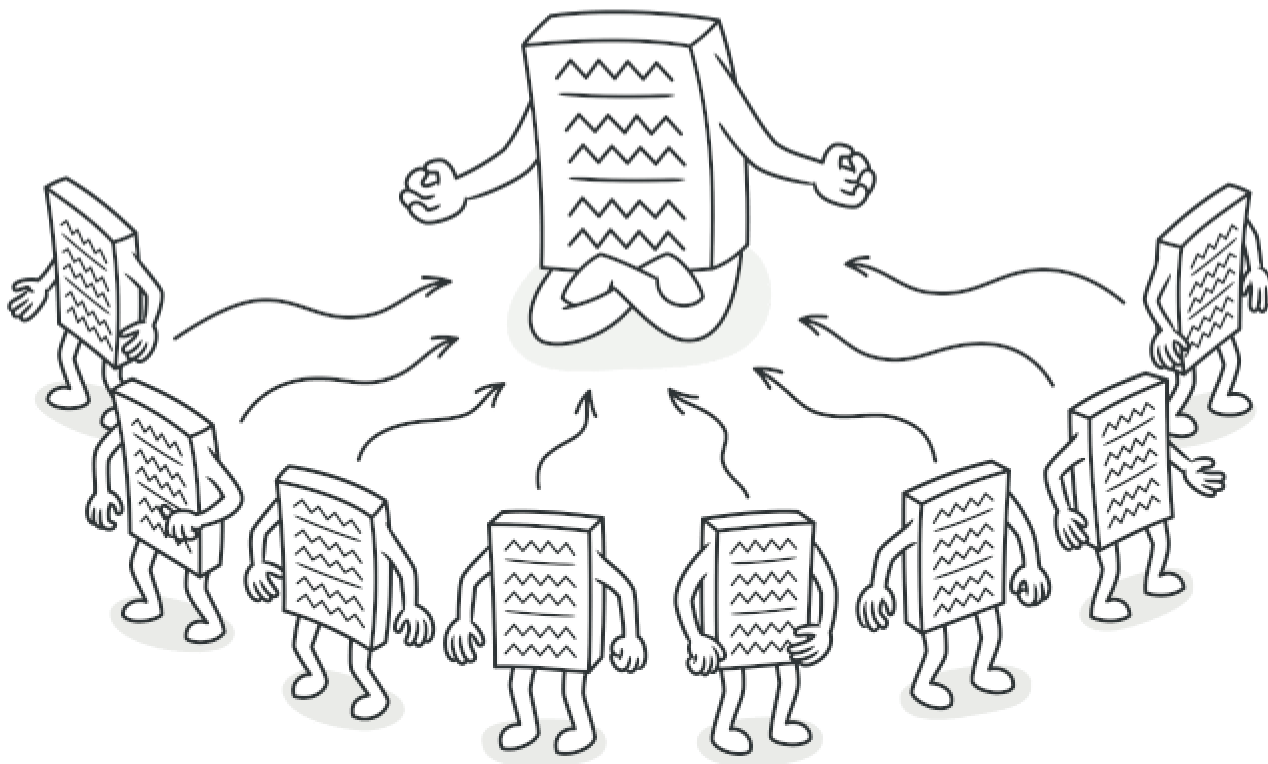
Singleton

Также известен как: *Одиночка*

(https://refactoring.guru/ru/design-patterns/singleton)



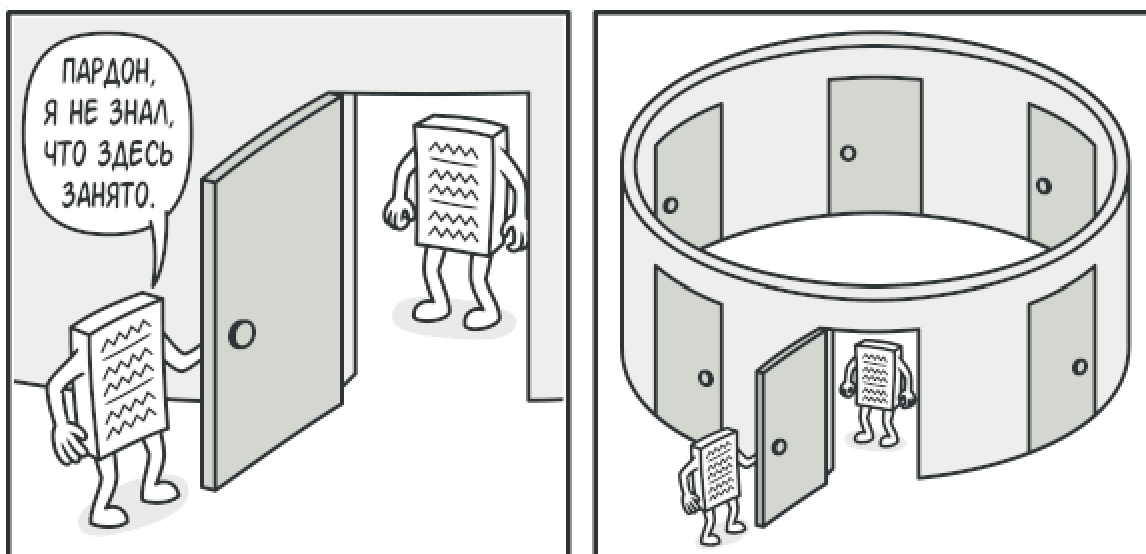
Одиночка — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.



Проблема

Решает две проблемы:

1. **Гарантирует наличие единственного экземпляра класса**. Чаще всего это полезно для доступа к какому-то общему ресурсу, например, базе данных. Представьте, что вы создали объект, а через некоторое время пробуете создать ещё один. В этом случае хотелось бы получить старый объект, вместо создания нового.
2. **Предоставляет глобальную точку доступа**. Это не просто глобальная переменная, а защищенная от записи глобальная переменная.



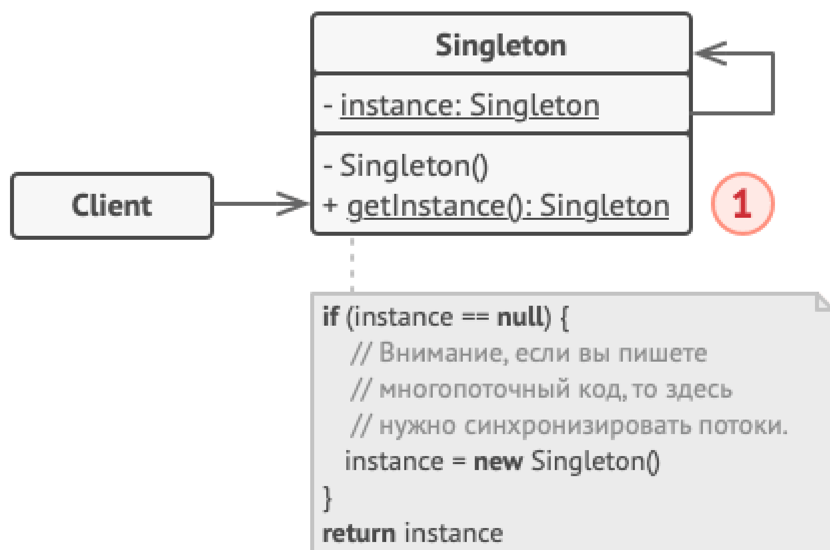
Применим:

- **Когда в программе должен быть единственный экземпляр какого-то класса**: Одиночка скрывает от клиентов все способы создания нового объекта, кроме специального метода. Этот метод либо создаёт объект, либо отдаёт существующий объект, если он уже был создан.
- **Когда вам хочется иметь больше контроля над глобальными переменными**: В отличие от глобальных переменных, Одиночка гарантирует, что никакой другой код не заменит созданный экземпляр класса, поэтому вы всегда уверены в наличии лишь одного объекта-одиночки.

Преимущества и недостатки

+	**-**
Гарантирует наличие единственного экземпляра класса	Нарушает *принцип единственной ответственности класса*
Предоставляет к нему глобальную точку доступа	Маскирует плохой дизайн
Реализует отложенную инициализацию объекта-одиночки	Проблемы мультипоточности
	Требует постоянного создания Mock-объектов при юнит-тестировании

Структура



Одиночка определяет статический метод `getInstance`, который возвращает единственный экземпляр своего класса.

Все реализации одиночки сводятся к тому, чтобы скрыть конструктор по умолчанию и создать публичный статический метод, который и будет контролировать жизненный цикл объекта-одиночки.

Примеры

Наивный Одиночка

Топорно реализовать Одиночку очень просто — достаточно скрыть конструктор и предоставить статический создающий метод.

(небезопасный в многопоточной среде !!!)

```
/**
 * Класс Одиночка предоставляет метод `GetInstance`, который ведёт себя как
 * альтернативный конструктор и позволяет клиентам получать один и тот же
 * экземпляр класса при каждом вызове.
 */
class Singleton
{
    /**
   * Конструктор Одиночки всегда должен быть скрытым, чтобы предотвратить
   * создание объекта через оператор new.
     */
protected:
    Singleton(const std::string value): value_(value)
    {
    }

    static Singleton* singleton_;
```

```
    std::string value_;
public:
    /**
   * Одиночки не должны быть клонируемыми.
    */
    Singleton(Singleton &other) = delete;
    /**
   * Синглтоны не должны быть назначаемыми.
    */
    void operator=(const Singleton &) = delete;
    /**
   * Это статический метод, управляющий доступом к экземпляру одиночки. При
   * первом запуске, он создаёт экземпляр одиночки и помещает его в
   * статическое поле. При последующих запусках, он возвращает клиенту объект,
   * хранящийся в статическом поле.
    */
    static Singleton *GetInstance(const std::string& value);
    /**
   * Наконец, любой одиночка должен содержать некоторую бизнес-логику, которая
   * может быть выполнена на его экземпляре.
    */
    void SomeBusinessLogic()
    {
        // ...
    }

    std::string value() const{
        return value_;
    }
};

Singleton* Singleton::singleton_= nullptr;;

/**
* Статические методы должны быть определены вне класса.
*/
Singleton *Singleton::GetInstance(const std::string& value)
{
    /**
    * This is a safer way to create an instance. instance = new Singleton is
    * dangeruous in case two instance threads wants to access at the same
time
    */
    if(singleton_==nullptr){
        singleton_ = new Singleton(value);
    }
    return singleton_;
}
```

```
void ThreadFoo(){
    // Этот код эмулирует медленную инициализацию.
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    Singleton* singleton = Singleton::GetInstance("FOO");
    std::cout << singleton->value() << "\n";
}

void ThreadBar(){
    // Этот код эмулирует медленную инициализацию.
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    Singleton* singleton = Singleton::GetInstance("BAR");
    std::cout << singleton->value() << "\n";
}

int main()
{
    std::cout <<"If you see the same value, then singleton was reused
(yay!\n" <<
                "If you see different values, then 2 singletons were created
(booo!!)\n\n" <<
                "RESULT:\n";
    std::thread t1(ThreadFoo);
    std::thread t2(ThreadBar);
    t1.join();
    t2.join();

    return 0;
}
```
Вывод:
```
If you see the same value, then singleton was reused (yay!
If you see different values, then 2 singletons were created (booo!!)

RESULT:
BAR
FOO
```

Многопоточный Одиночка

```
/**
 * Класс Одиночка предоставляет метод `GetInstance`, который ведёт себя как
 * альтернативный конструктор и позволяет клиентам получать один и тот же
 * экземпляр класса при каждом вызове.
 */
class Singleton
{

    /**
   * Конструктор Одиночки всегда должен быть скрытым, чтобы предотвратить
```

```
   * создание объекта через оператор  new.
     */
private:
    static Singleton * pinstance_;
    static std::mutex mutex_;

protected:
    Singleton(const std::string value): value_(value)
    {
    }
    ~Singleton() {}
    std::string value_;

public:
    /**
   * Одиночки не должны быть клонируемыми.
     */
    Singleton(Singleton &other) = delete;
    /**
     * Singletons should not be assignable.
     */
    void operator=(const Singleton &) = delete;
    /**
   * Это статический метод, управляющий доступом к экземпляру одиночки. При
   * первом запуске, он создаёт экземпляр одиночки и помещает его в
   * статическое поле. При последующих запусках, он возвращает клиенту объект,
   * хранящийся в статическом поле.
     */

    static Singleton *GetInstance(const std::string& value);
    /**
   * Наконец, любой одиночка должен содержать некоторую бизнес-логику, которая
   * может быть выполнена на его экземпляре.
     */
    void SomeBusinessLogic()
    {
        // ...
    }

    std::string value() const{
        return value_;
    }
};

/**
 * Static methods should be defined outside the class.
 */

Singleton* Singleton::pinstance_{nullptr};
std::mutex Singleton::mutex_;
```

```
/**
 * The first time we call GetInstance we will lock the storage location
 *      and then we make sure again that the variable is null and then we
 *      set the value. RU:
 */
Singleton *Singleton::GetInstance(const std::string& value)
{
    std::lock_guard<std::mutex> lock(mutex_);
    if (pinstance_ == nullptr)
    {
        pinstance_ = new Singleton(value);
    }
    return pinstance_;
}

void ThreadFoo(){
    // Этот код эмулирует медленную инициализацию.
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    Singleton* singleton = Singleton::GetInstance("FOO");
    std::cout << singleton->value() << "\n";
}

void ThreadBar(){
    // Этот код эмулирует медленную инициализацию.
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    Singleton* singleton = Singleton::GetInstance("BAR");
    std::cout << singleton->value() << "\n";
}

int main()
{
    std::cout <<"If you see the same value, then singleton was reused
(yay!)\n" <<
                "If you see different values, then 2 singletons were created
(booo!!)\n\n" <<
                "RESULT:\n";
    std::thread t1(ThreadFoo);
    std::thread t2(ThreadBar);
    t1.join();
    t2.join();

    return 0;
}
```

Вывод:

```
If you see the same value, then singleton was reused (yay!)
If you see different values, then 2 singletons were created (booo!!)

RESULT:
```

```
FOO
FOO
```

From:
https://wiki.radi0.cc/ - **radi0wiki**

Permanent link:
https://wiki.radi0.cc/glossary:design:patterns:creational:singleton

Last update: **2025/11/09 12:07**