



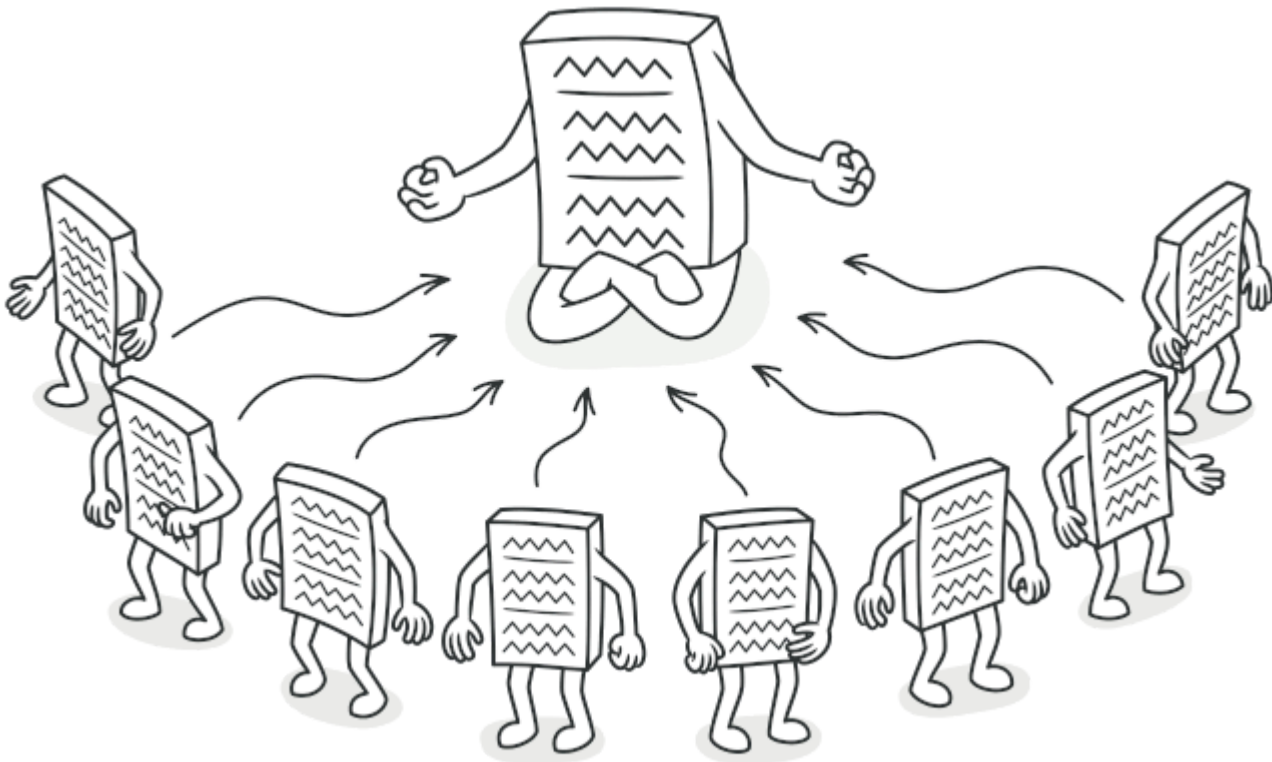
🏠 / Паттерны проектирования / Порождающие паттерны

Одиночка

Также известен как: Singleton

💬 Суть паттерна

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



😞 Проблема

Одиночка решает сразу две проблемы, нарушая *принцип единственной ответственности* класса.

доступа к какому-то общему ресурсу, например, базе данных.

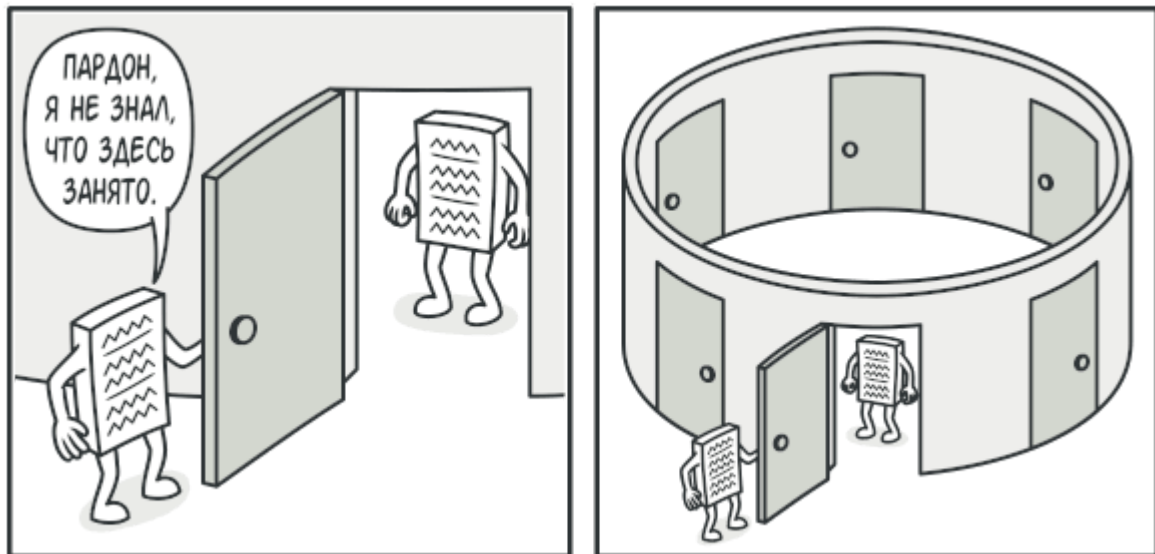


НОВОГОДНЯЯ РАСПРОДАЖА!



Представьте, что вы создали объект, а через некоторое время пробуете создать ещё один. В этом случае хотелось бы получить старый объект, вместо создания нового.

Такое поведение невозможно реализовать с помощью обычного конструктора, так как конструктор класса **всегда** возвращает новый объект.



Клиенты могут не подозревать, что работают с одним и тем же объектом.

2. **Предоставляет глобальную точку доступа.** Это не просто глобальная переменная, через которую можно достигаться к определённому объекту. Глобальные переменные не защищены от записи, поэтому любой код может подменять их значения без вашего ведома.

Но есть и другой нюанс. Неплохо бы хранить в одном месте и код, который решает проблему №1, а также иметь к нему простой и доступный интерфейс.

Интересно, что в наше время паттерн стал настолько известен, что теперь люди называют «одиночками» даже те классы, которые решают лишь одну из проблем, перечисленных выше.

😊 Решение

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



НОВОГОДНЯЯ РАСПРОДАЖА!

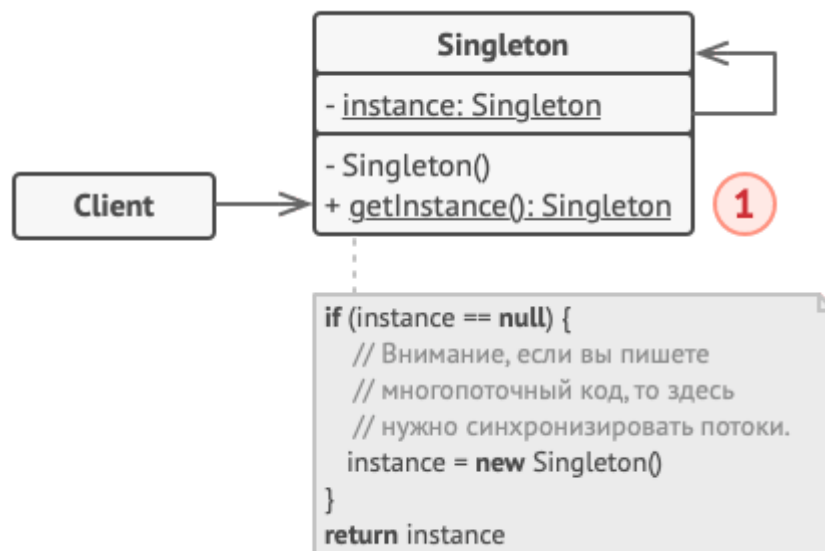


Если у вас есть доступ к классу одиночки, значит, будет доступ и к этому статическому методу. Из какой точки кода вы бы его ни вызвали, он всегда будет отдавать один и тот же объект.

Аналогия из жизни

Правительство государства — хороший пример одиночки. В государстве может быть только одно официальное правительство. Вне зависимости от того, кто конкретно заседает в правительстве, оно имеет глобальную точку доступа «Правительство страны N».

Структура



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

Конструктор одиночки должен быть скрыт от клиентов. Вызов метода `getInstance` должен стать единственным способом получить объект этого класса.

Псевдокод

В этом примере роль **Одиночки** отыгрывает класс подключения к базе данных.



НОВОГОДНЯЯ РАСПРОДАЖА!



Объект — это вызвать метод `getInstance`. Этот метод сохранит первый созданный объект и будет возвращать его при всех последующих вызовах.

```
// Класс одиночки определяет статический метод `getInstance`,
// который позволяет клиентам повторно использовать одно и то же
// подключение к базе данных по всей программе.
```

```
class Database is
```

```
    // Поле для хранения объекта-одиночки должно быть объявлено
    // статичным.
```

```
    private static field instance: Database
```

```
    // Конструктор одиночки всегда должен оставаться приватным,
    // чтобы клиенты не могли самостоятельно создавать
    // экземпляры этого класса через оператор `new`.
```

```
    private constructor Database() is
```

```
        // Здесь может жить код инициализации подключения к
        // серверу баз данных.
        // ...
```

```
    // Основной статический метод одиночки служит альтернативой
    // конструктору и является точкой доступа к экземпляру этого
    // класса.
```

```
    public static method getInstance() is
```

```
        if (Database.instance == null) then
            acquireThreadLock() and then
                // На всякий случай ещё раз проверим, не был ли
                // объект создан другим потоком, пока текущий
                // ждал освобождения блокировки.
                if (Database.instance == null) then
                    Database.instance = new Database()
            return Database.instance
```

```
    // Наконец, любой класс одиночки должен иметь какую-то
    // полезную функциональность, которую клиенты будут
    // запускать через полученный объект одиночки.
```

```
    public method query(sql) is
```

```
        // Все запросы к базе данных будут проходить через этот
        // метод. Поэтому имеет смысл поместить сюда какую-то
        // логику кеширования.
        // ...
```

```
class Application is
```

```
    method main() is
```

```
        Database foo = Database.getInstance()
        foo.query("SELECT ...")
        // ...
        Database bar = Database.getInstance()
        bar.query("SELECT ...")
```



💡 Применимость

🔒 Когда в программе должен быть единственный экземпляр какого-то класса, доступный всем клиентам (например, общий доступ к базе данных из разных частей программы).

⚡ 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.

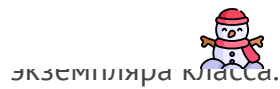
⚡ В отличие от глобальных переменных, Одиночка гарантирует, что никакой другой код не заменит созданный экземпляр класса, поэтому вы всегда уверены в наличии лишь одного объекта-одиночки.

Тем не менее, в любой момент вы можете расширить это ограничение и позволить любое количество объектов-одиночек, поменяв код в одном месте (метод `getInstance`).

📋 Шаги реализации

1. Добавьте в класс приватное статическое поле, которое будет содержать одиночный объект.
2. Объявите статический создающий метод, который будет использоваться для получения одиночки.
3. Добавьте «ленивую инициализацию» (создание объекта при первом вызове метода) в создающий метод одиночки.
4. Сделайте конструктор класса приватным.
5. В клиентском коде замените вызовы конструктора одиночка вызовами его создающего метода.

⚖️ Преимущества и недостатки



НОВОГОДНЯЯ РАСПРОДАЖА!

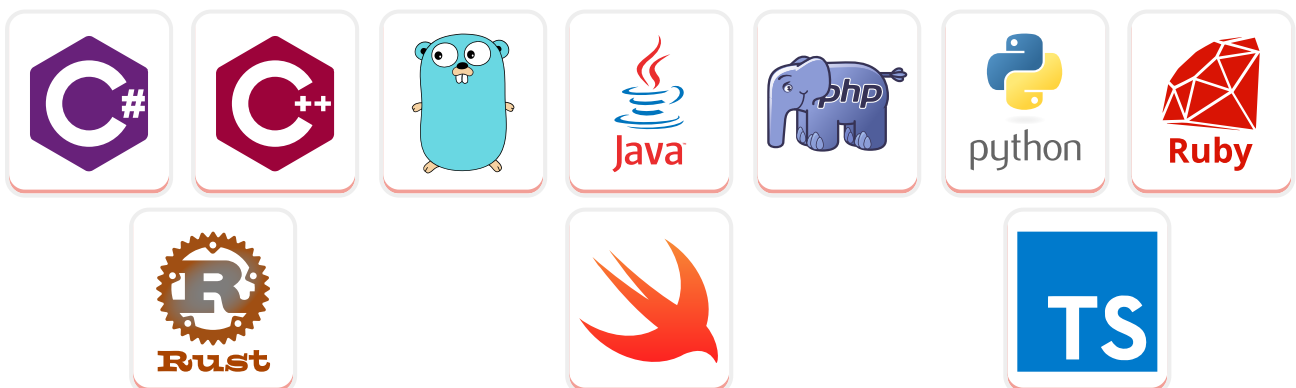


- | | |
|--|--|
| ✓ Предоставляет к нему глобальную точку доступа. | ✗ Маскирует плохой дизайн. |
| ✓ Реализует отложенную инициализацию объекта-одиночки. | ✗ Проблемы мультипоточности. |
| | ✗ Требуется постоянное создание Mock-объектов при юнит-тестировании. |

⇔ Отношения с другими паттернами

- Фасад можно сделать Одиночкой, так как обычно нужен только один объект-фасад.
- Паттерн Легковес может напоминать Одиночку, если для конкретной задачи у вас получилось свести количество объектов к одному. Но помните, что между паттернами есть два кардинальных отличия:
 1. В отличие от *Одиночки*, вы можете иметь множество объектов-легковесов.
 2. Объекты-легковесы должны быть неизменяемыми, тогда как объект-одиночка допускает изменение своего состояния.
- Абстрактная фабрика, Строитель и Прототип могут быть реализованы при помощи Одиночки.

</> Примеры реализации паттерна





НОВОГОДНЯЯ РАСПРОДАЖА!



Не втыкай в транспорте

Лучше почитай нашу книгу о паттернах проектирования.

Теперь это удобно делать даже во время поездок в общественном транспорте.

 Узнать больше...

[ВЕРНУТЬСЯ НАЗАД](#)[ЧИТАЕМ ДАЛЬШЕ](#)

[Главная](#) [Рефакторинг](#) [Паттерны](#) [Премиум контент](#)
[Курс по Git](#) [Форум](#) [Связаться](#)



© 2014-2025 Refactoring.Guru.

Все права защищены.

 Иллюстрации нарисовал Дмитрий Жарт


[Условия использования](#)

[Политика конфиденциальности](#)

[Использование контента](#) [About us](#)

Ukrainian office:

 FOP Olga Skobeleva

 Abolmasova 7

Kyiv, Ukraine, 02002

 Email:

support@refactoring.guru

Spanish office:

 Oleksandr Shvets

 Avda Pamplona 64

Pamplona, Spain, 31009

 Email:

support@refactoring.guru