

SINGLETON **ABUSE**

або чому одинак може створити не **одну** проблему

Що таке сінглтон(одинак)?

породжувальний патерн проектування, який гарантує, що клас має лише один екземпляр, та надає глобальну точку доступу до нього

Вирішує дві проблеми:

- **Гарантує наявність єдиного екземпляра класу**

Уявіть собі, що ви створили об'єкт, а через деякий час намагаєтесь створити ще один. У цьому випадку хотілося б отримати старий об'єкт замість створення нового.

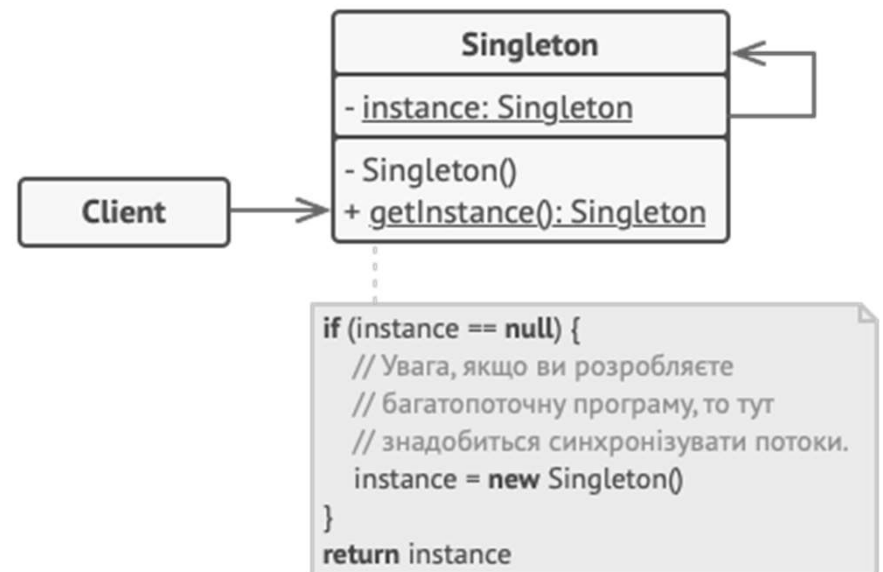
- **Надає глобальну точку доступу**

Це не просто глобальна змінна, через яку можна дістатися до певного об'єкта. Глобальні змінні не захищені від запису, тому будь-який код може підмінити їхнє значення без вашого відома.

Рішення:

Всі реалізації Одинака зводяться до того, аби приховати типовий конструктор та створити публічний статичний метод, який і контролюватиме життєвий цикл об'єкта-одинака.

Якщо у є доступ до класу одинака, отже, буде й доступ до цього статичного методу. З якої точки коду б його не викликали, він завжди віддаватиме один і той самий об'єкт.



Приклад з життя:

Суть Одиная це єдиний ресурс, який спільно використовується кількома користувачами:

- спільне використання однієї пральної машини для всіх мешканців готелю
- спільне використання одного холодильника для всіх членів сім'ї
- Пес Патрон





ПРИКЛАДИ ІНІЦІАЛІЗАЦІЇ

Дуже примітивний приклад eager ініціалізації:

```
public class PatronEagerInit {  
    private static final PatronEagerInit instance = new PatronEagerInit();  
    private PatronEagerInit(){}  
    public static PatronEagerInit getInstance() {  
        return instance;  
    }  
    public void searchBomb(){  
        System.out.println("Hav-hav");  
    }  
}
```

Що не так?

- Під час швидкої ініціалізації екземпляр класу singleton створюється під час завантаження класу.
- Недоліком швидкої ініціалізації є те, що метод створюється, навіть якщо клієнтська програма його не використовує.
- Якщо ваш клас не використовує багато ресурсів, це підхід для використання. Але в більшості сценаріїв singleton класи створюються для таких ресурсів, як файлова система, підключення до бази даних тощо. Нам слід уникати створення екземплярів, якщо клієнт не викликає метод `getInstance`. Крім того, цей метод не надає жодних опцій для обробки винятків.

Менш примітивний приклад:

```
public class PatronStaticBlockInit {
    private static PatronStaticBlockInit instance;
    private PatronStaticBlockInit(){}
    // static block initialization дозволяє додати обробку помилок
    static {
        try {
            instance = new PatronStaticBlockInit();
        } catch (Exception e) {
            throw new RuntimeException("Problem occurred");
        }
    }

    public static PatronStaticBlockInit getInstance() { return instance; }

    public void searchBomb() { System.out.println("Hav-hav"); }
}

class Test3 {
    public static void main(String[] args){
        PatronStaticBlockInit sobaka = PatronStaticBlockInit.getInstance();
        sobaka.searchBomb();
    }
}
```

Реалізація ініціалізації статичного блоку подібна до попередньої ініціалізації, за винятком того, що екземпляр класу створюється в статичному блоці, який надає можливість обробки винятків.

Обидві ініціалізації створюють екземпляр ще до його використання, і це не найкраща практика для використання.

Відкладена ініціалізація:

```
public class PatronLazyInit {  
    // відкладена ініціалізація  
    private static PatronLazyInit INSTANCE;  
  
    private PatronLazyInit() {}  
  
    // статичний метод виклику собаки  
    public static PatronLazyInit getInstance() {  
        // якщо собаки не існує - створити одну єдину собаку  
        if(INSTANCE == null) {  
            INSTANCE = new PatronLazyInit();  
        }  
        return INSTANCE;  
    }  
  
    public void searchBomb() { System.out.println("Hav-hav"); }  
}
```

А тут що не так?

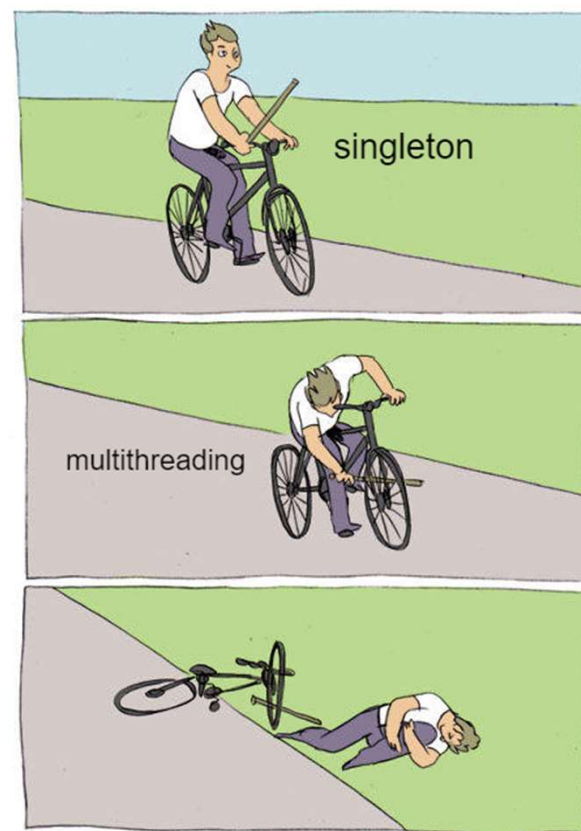
- Ця реалізація чудово працює у випадку однопоточового середовища, але коли мова йде про багатопотокові системи, це може спричинити проблеми, якщо кілька потоків знаходяться в умові if одночасно.
- Це знищить шаблон singleton, і обидва потоки отримають різні екземпляри класу singleton.



ЯК
ЗЛАМАТИ
ОДИНАКА?

Зламати варіант #1 або «Багатопоточність»

Як вже сказано відкладена ініціалізація за допомогою `if` блоку може зламати одинака в багатопоточному середовищі, якщо декілька потоків одночасно зайдуть в блок `if`. Тоді два потоки отримають різні екземпляри класу.



Полагодити варіант #1

Простий спосіб створити потокобезпечний одиночний клас полягає в тому, щоб синхронізувати метод глобального доступу, щоб тільки один потік міг виконувати цей метод одночасно.

```
public class PatronThreadSafe {  
    private static PatronThreadSafe instance;  
  
    private PatronThreadSafe(){}  
  
    public static synchronized PatronThreadSafe getInstance() {  
        if (instance == null) {  
            instance = new PatronThreadSafe();  
        }  
        return instance;  
    }  
  
    public void searchBomb() { System.out.println("Hav-hav"); }  
}
```

Що тут не так?

Ця реалізація працює нормально та забезпечує безпеку потоків, але вона знижує продуктивність через вартість, пов'язану з синхронізованим методом, хоча нам вона потрібна лише для кількох перших потоків, які можуть створювати окремі екземпляри.

Чи можна краще?

Щоб уникнути цих додаткових витрат кожного разу, використовується принцип подвійної перевірки блокування. У цьому підході синхронізований блок використовується всередині умови if з додатковою перевіркою, щоб гарантувати, що створено лише один екземпляр класу singleton.

```
public class PatronThreadSafeEfficient {  
    private static PatronThreadSafeEfficient instance;  
  
    private PatronThreadSafeEfficient(){}  
  
    public static PatronThreadSafeEfficient getInstance() {  
        if (instance == null) {  
            synchronized (PatronThreadSafeEfficient.class) {  
                if (instance == null) {  
                    instance = new PatronThreadSafeEfficient();  
                }  
            }  
        }  
        return instance;  
    }  
    public void searchBomb() { System.out.println("Hav-hav"); }  
}
```


Зламати варіант #2 або «Рефлексія»

- Використовуючи рефлексію, ми все одно можемо створити кілька екземплярів класу, змінивши область видимості конструктора.
- Ось як ви бачите, ми створили ще один об'єкт класу Singleton.

```
BreakPatronReflection
C:\Users\daria\.jdk\corretto-11.0.10\bin\java.exe "-javaagent:C:\Users\daria\Files\JetBrains\IntelliJ IDEA Community Edition 2020.3.3\bin" -
.BreakPatronReflection
2088051243
1277181601
```

```
public class BreakPatronReflection {
    public static void main(String[] args) throws InstantiationException,
        IllegalAccessException,
        IllegalArgumentException,
        InvocationTargetException{
        Patron a = Patron.getSobaka();
        Patron b = null;

        Constructor<?>[] constructors = Patron.class.getDeclaredConstructors();

        for (Constructor constructor : constructors) {
            //зробити приватний конструктор публічним
            constructor.setAccessible(true);
            b = (Patron) constructor.newInstance();
            break;
        }

        System.out.println(a.hashCode());
        System.out.println(b.hashCode());
    }
}
```



Полагодити варіант #2

- Щоб подолати цю ситуацію з Reflection, Джошуа Блох пропонує використовувати enum для реалізації singleton, оскільки Java гарантує, що будь-яке значення enum створюється лише один раз у програмі.

```
.FixPatronReflection  
2088051243  
2088051243
```

```
public class FixPatronReflection {  
    public static void main(String[] args) {  
        PatronEnum a = PatronEnum.INSTANCE;  
        PatronEnum b = PatronEnum.INSTANCE;  
        System.out.println(a.hashCode());  
        System.out.println(b.hashCode());  
    }  
}  
  
enum PatronEnum {  
    INSTANCE;  
}
```

Переваги:

- Java гарантує, що будь-яке значення enum створюється лише один раз у програмі.
- Оскільки значення Java Enum доступні глобально, сінглтон також доступний

Недоліки:

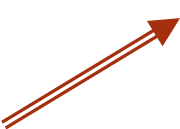
- Тип enum дещо негнучкий (наприклад, він не допускає відкладену ініціалізацію)

Зламати варіант #3 або «Сереалізація»

Іноді в розподілених системах нам потрібно реалізувати інтерфейс `Serializable` в класі `singleton`, щоб ми могли зберігати його стан у файловій системі та отримати його пізніше.

Проблема з серіалізованим Одинаком полягає в тому, що щоразу, коли ми його десеріалізуємо, він створить новий екземпляр класу. Таким чином, це руйнує шаблон одного елемента.

```
class SingletonSerializedTest {  
  
    public static void main(String[] args) throws FileNotFoundException, IOException, ClassNotFoundException {  
        BreakPatronSerialization instanceOne = BreakPatronSerialization.getInstance();  
        ObjectOutput out = new ObjectOutputStream(new FileOutputStream("filename.ser"));  
        out.writeObject(instanceOne);  
        out.close();  
  
        // десереалізація файлу в об'єкти  
        ObjectInput in = new ObjectInputStream(new FileInputStream("filename.ser"));  
        BreakPatronSerialization instanceTwo = (BreakPatronSerialization) in.readObject();  
        in.close();  
  
        System.out.println("instanceOne hashCode="+instanceOne.hashCode());  
        System.out.println("instanceTwo hashCode="+instanceTwo.hashCode());  
    }  
}
```



```
.SingletonSerializedTest  
instanceOne hashCode=1641808846  
instanceTwo hashCode=2140832232
```

Полагодити варіант #3

- Щоб подолати цей сценарій, все, що нам потрібно зробити, це реалізувати метод `readResolve()`.
- Проблема `readResolve` полягає в тому, що він замінює зчитуваний з файлу об'єкт тим, що зараз знаходиться в пам'яті. Це означає, що якщо після запису у файл у класі відбулись зміни, десереалізувати попередню версію класу вже не вийде.

```
Files\JetBrains\IntelliJ IDEA Commu
 SingletonSerializedTestFix
instanceOne hashCode=1641808846
instanceTwo hashCode=1641808846
```

```
public class FixPatronSerialization implements Serializable {
    private static FixPatronSerialization instance;
    private FixPatronSerialization(){}
    public static FixPatronSerialization getInstance() {
        if (instance == null) {
            synchronized (FixPatronSerialization.class) {
                if (instance == null) {
                    instance = new FixPatronSerialization();
                }
            }
        }
        return instance;
    }
    protected Object readResolve() {
        return getInstance();
    }

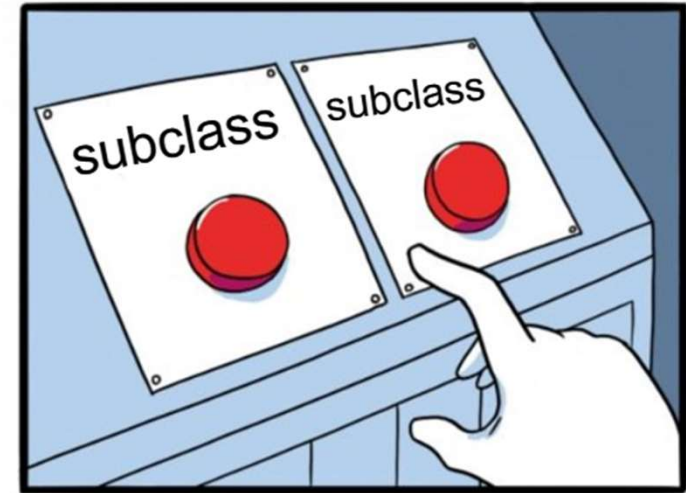
    public void searchBomb() { System.out.println("Hav-hav"); }
}
```



ПІДКЛАСИ

Проблема

- Основна проблема полягає не стільки у визначенні підкласів, скільки в визначенні його унікального екземпляра, щоб клієнти могли ним користуватись.
- По суті, змінна, яка посилається на наш сінглтон, повинна бути ініціалізована екземпляром підкласу. Потрібно визначити, який саме з підкласів сінглтону ви хочете використовувати.



JAKE-CLARK.TUMBLR

Варіант #1: нехай метод MazeFactory instance() визначає підклас для створення екземпляра

```
public abstract class MazeFactory1 {  
  
    private static MazeFactory1 uniqueInstance = null;  
  
    protected MazeFactory1() {}  
  
    public static MazeFactory1 instance() {  
        if (uniqueInstance == null)  
            return instance("enchanted");  
        else return uniqueInstance;  
    }  
  
    public static MazeFactory1 instance(String name) {  
        if(uniqueInstance == null)  
            if (name.equals("enchanted"))  
                uniqueInstance = new EnchantedMazeFactory1();  
            else if (name.equals("agent"))  
                uniqueInstance = new AgentMazeFactory1();  
        return uniqueInstance;  
    }  
}
```

```
class Test1{  
    public static void main(String [] args){  
        // створити обрану фабрику  
        MazeFactory1 factory = MazeFactory1.instance("enchanted");  
        // доступитись до екземпляру фабрики  
        MazeFactory1 factory1 = MazeFactory1.instance();  
        System.out.println("instanceOne hashCode="+factory.hashCode());  
        System.out.println("instanceTwo hashCode="+factory1.hashCode());  
    }  
}
```


Варіант #1: недоліки

- Конструктори EnchantedMazeFactory та AgentMazeFactory не можуть бути приватним, оскільки MazeFactory повинен бути в змозі створювати їх екземпляри.
- Потенційно клієнти можуть створити екземпляри інших екземплярів цих підкласів.
- Методи instance(String) порушують принцип «open-closed», оскільки його потрібно змінювати для кожного нового підкласу MazeFactory.
- Функцію Instance потрібно змінювати щоразу, коли ви визначаєте новий підклас MazeFactory. Це може не бути проблемою в цій програмі, але це може бути для абстрактних фабрик, визначених у фреймворку.

Варіант #2: нехай кожен підклас надає статичний метод instance()

```
public abstract class MazeFactory2 {
    // protected посилання на єдиний екземпляр.
    protected static MazeFactory2 uniqueInstance = null;
    // default constructor, не може бути приватним тут
    protected MazeFactory2() {}
    public static MazeFactory2 instance() {return uniqueInstance;}
}

// Клас EnchantedMazeFactory є реалізацією класу що дозволяє лише один екземпляр.
class EnchantedMazeFactory2 extends MazeFactory2 {
    public static MazeFactory2 instance() {
        if(uniqueInstance == null)
            uniqueInstance = new EnchantedMazeFactory2();
        return uniqueInstance;
    }
    // Private конструктор підкласу!!
    private EnchantedMazeFactory2() {}
}
```

```
class Test2{
    public static void main(String [] args){
        // створити обрану фабрику
        MazeFactory2 factory = EnchantedMazeFactory2.instance();
        // доступитись до екземпляру фабрики
        MazeFactory2 factory1 = MazeFactory2.instance();
        System.out.println("instanceOne hashCode="+factory.hashCode());
        System.out.println("instanceTwo hashCode="+factory1.hashCode());
    }
}
```

Варіант #2

- Клас MazeFactory є реалізацією класу, який дозволяє лише один екземпляр підкласу. Ця версія вимагає, щоб його підкласи забезпечували реалізацію статичний метод instance().
- Це дозволяє програмісту обирати клас singleton під час зв'язування, але зберігає це прихованим від клієнтів.
- Підхід зв'язку фіксує вибір класу singleton під час зв'язування, що ускладнює вибір класу singleton під час виконання.
- Використання умовних операторів для визначення підкласу є більш гнучким, але воно жорстко об'єднує набір можливих класів Singleton.
- **Жоден** підхід не є достатньо гнучким у всіх випадках.

Варіант #3: реєстр сінглтонів

```
public class FlexibleExtendableSingleton {  
    //instance - лише один із них може існувати в системі (наразі потоки не враховуються).  
    private static FlexibleExtendableSingleton instance;  
  
    // Це має бути перед thisSingleton, тому що конструктору потрібен реєстр.  
    protected static Map<String, FlexibleExtendableSingleton> register = new HashMap<String, FlexibleExtendableSingleton>();  
  
    // кожен клас в ієрархії потребує thisSingleton. Це запустить створення (а отже, реєстрацію).  
    private static FlexibleExtendableSingleton thisSingleton = new FlexibleExtendableSingleton();  
  
    protected static void register(String name, FlexibleExtendableSingleton singletonClass) {  
        register.put(name, singletonClass);  
    }  
  
    protected static FlexibleExtendableSingleton lookup(String name) {  
        return register.get(name);  
    }  
  
    public static FlexibleExtendableSingleton getInstance() {  
        if (instance == null) {  
            String singletonName = System.getProperty("classname");  
            instance = lookup(singletonName);  
        }  
        return instance;  
    }  
  
    protected FlexibleExtendableSingleton() {  
        register(this.getClass().getSimpleName(), this);  
    }  
}
```

Варіант #3:

```
class FlexibleSubclassSingleton extends FlexibleExtendableSingleton {  
    // кожен клас в ієрархії потребує thisSingleton. Це запустить створення (а отже, реєстрацію).  
    private static FlexibleSubclassSingleton thisSingleton = new FlexibleSubclassSingleton();  
  
    protected FlexibleSubclassSingleton() {  
        //Наступний рядок зареєструє клас.  
        super();  
    }  
}
```

Варіант #3

- Більш гнучкий підхід використовує реєстр синглтонів. Замість того, щоб Instance визначав набір можливих класів Singleton, класи Singleton можуть зареєструвати свій екземпляр Singleton за назвою у загальновідомому реєстрі.
- Реєстр відповідає іменам рядків і синглтонам. Коли Instance потрібен синглтон, він звертається до реєстру, запитуючи синглтон за назвою.
- Register реєструє екземпляр Singleton під заданим іменем. Операція lookup знаходить синглтон із його назвою.
- Де реєструються класи Singleton? Один з варіантів - в їх конструкторі.
- Клас FlexibleExtendableSingleton більше не несе відповідальності за створення singleton. Натомість його головна відповідальність полягає в тому, щоб зробити єдиний об'єкт вибору доступним у системі.

Варіант #3

- Звичайно, конструктор не буде викликаний, якщо хтось не створить екземпляр класу, що повторює проблему, яку намагається вирішити шаблон Singleton.
- Ми можемо вирішити цю проблему, визначивши статичний екземпляр MySingleton.
- В наведеному прикладі на Java використан просто метод eager initialization (що, звичайно потребує змін для багатопоточного використання)

```
MySingleton::MySingleton() {  
    // ...  
    Singleton::Register("MySingleton", this);  
}
```

```
static MySingleton theSingleton;
```

```
// кожен клас в ієрархії потребує thisSingleton. Це запустить створення (а отже, реєстрацію).  
private static FlexibleExtendableSingleton thisSingleton = new FlexibleExtendableSingleton();
```



ІНШІ СІНГЛТОНИ

Сінглтон Маєрса

- Цей спосіб дуже лаконічно використовує можливості мови C++. В даному випадку функція `getInstance()` повертає посилання на об'єкт класу, а не вказівник, як у класичному синглтоні.
- Коротко і просто. Статичні дані можна ініціалізувати тільки один раз, чим Майєрс і скористався. При першому виклику функції `getInstance()` створиться об'єкт `instance` і повернеться посилання на нього. При другому і наступних викликах об'єкт `instance` вже не створюватиметься, оскільки він статичний. Тоді просто повертатиметься посилання на об'єкт, який вже створено.

звичайний сінглтон

```
class ClassicSingleton
{
public:
    static ClassicSingleton* getInstance();

private:
    ClassicSingleton();
    ClassicSingleton(const ClassicSingleton& cs) = delete;
    ClassicSingleton& operator=(const ClassicSingleton& cs) = delete;
    ~ClassicSingleton();

    static ClassicSingleton* m_pInstance;
};
```

```
ClassicSingleton* ClassicSingleton::m_pInstance = 0;
ClassicSingleton* ClassicSingleton::getInstance()
{
    if (m_pInstance == nullptr)
        m_pInstance = new ClassicSingleton();
    return m_pInstance;
}
```

сінглтон Маєрса

```
class MeyersSingleton
{
public:
    static MeyersSingleton& getInstance();

private:
    MeyersSingleton();
    MeyersSingleton(const MeyersSingleton& ms) = delete;
    MeyersSingleton& operator=(const MeyersSingleton& ms) = delete;
    ~MeyersSingleton();
};
```

```
MeyersSingleton& MeyersSingleton::getInstance()
{
    static MeyersSingleton instance;
    return instance;
}
```

Сінглтон Білла П'ю (або ініціалізація вкладеним класом)

```
public class BillPughSingleton {  
  
    private BillPughSingleton(){}  
  
    private static class SingletonHelper {  
        private static final BillPughSingleton INSTANCE = new BillPughSingleton();  
    }  
  
    public static BillPughSingleton getInstance() {  
        return SingletonHelper.INSTANCE;  
    }  
}
```

Сінглтон Білла П'ю

- До Java 5 модель пам'яті Java мала багато проблем, і попередні підходи зазнавали помилок у певних сценаріях, коли забагато потоків намагалися отримати примірник класу `singleton` одночасно. Тож Білл П'ю запропонував інший підхід до створення класу `singleton` за допомогою внутрішнього статичного допоміжного класу.
- Коли клас `Singleton` завантажується, клас `SingletonHelper` не завантажується в пам'ять. Тільки коли хтось викликає метод `getInstance()`, цей клас завантажується та створює екземпляр класу `Singleton`.
- Це найбільш широко використовуваний підхід для класу `singleton`, оскільки він не потребує синхронізації.

Сінглтон та інші патерни

- **Фасад (Facade)** можна зробити сінглтоном, оскільки зазвичай потрібен тільки один об'єкт-фасад.
- Патерн **Легковаговик (Flyweight)** може нагадувати сінглтон, якщо для конкретного завдання ви змогли зменшити кількість об'єктів до одного. Але пам'ятайте, що між патернами є дві суттєві відмінності:
 - На відміну від сінглтона, ви можете мати безліч об'єктів-легковаговиків.
 - Об'єкти-легковаговики повинні бути незмінними, тоді як об'єкт-одинак допускає зміну свого стану.
- **Абстрактна фабрика (Abstract Factory), Будівельник (Builder) та Прототип(Prototype)** можуть реалізовуватися за допомогою сінглтона.
- **Мультитон**

Мультитон

- Шаблон проектування Multiton є розширенням шаблону singleton. Це забезпечує існування обмеженої кількості екземплярів класу, вказуючи ключ для кожного екземпляра та дозволяючи створювати лише один об'єкт для кожного з цих ключів.
- Цей шаблон узагальнює singleton. Тоді як сінглтон дозволяє створити лише один екземпляр класу, багатотонний шаблон дозволяє кероване створення кількох екземплярів, якими він керує за допомогою мапи.

Мультитон

«Назгули, яких також називають примарами перстня або Дев'ятьма вершниками, є найжахливішими слугами Саурона. За визначенням, їх завжди дев'ять»

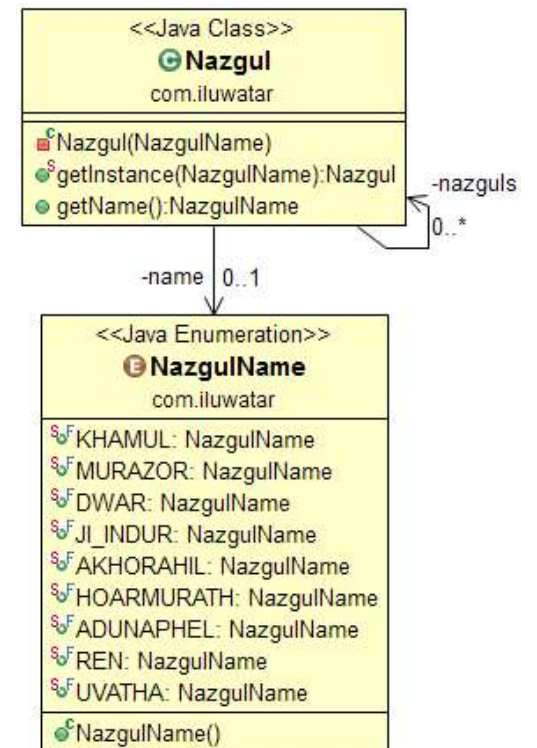


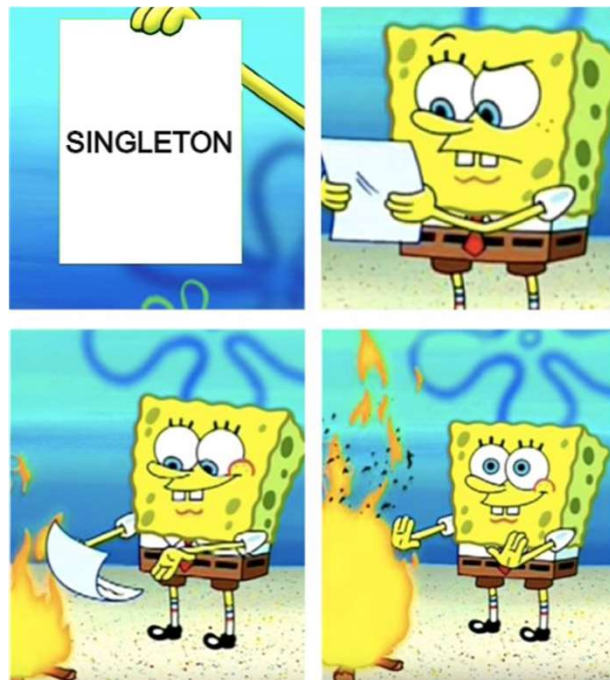
```

enum NazgulName {
    KHAMUL, MURAZOR, DWAR, JI_INDUR, AKHORAHL, HOARMURATH, ADUNAPHEL, REN, UVATHA
}

class Nazgul {
    private static final Map<NazgulName, Nazgul> nazguls;
    private final NazgulName name;
    static {
        nazguls = new ConcurrentHashMap<>();
        nazguls.put(NazgulName.KHAMUL, new Nazgul(NazgulName.KHAMUL));
        nazguls.put(NazgulName.MURAZOR, new Nazgul(NazgulName.MURAZOR));
        nazguls.put(NazgulName.DWAR, new Nazgul(NazgulName.DWAR));
        nazguls.put(NazgulName.JI_INDUR, new Nazgul(NazgulName.JI_INDUR));
        nazguls.put(NazgulName.AKHORAHIL, new Nazgul(NazgulName.AKHORAHIL));
        nazguls.put(NazgulName.HOARMURATH, new Nazgul(NazgulName.HOARMURATH));
        nazguls.put(NazgulName.ADUNAPHEL, new Nazgul(NazgulName.ADUNAPHEL));
        nazguls.put(NazgulName.REN, new Nazgul(NazgulName.REN));
        nazguls.put(NazgulName.UVATHA, new Nazgul(NazgulName.UVATHA));
    }
    private Nazgul(NazgulName name) {
        this.name = name;
    }
    public static Nazgul getInstance(NazgulName name) {
        return nazguls.get(name);
    }
    public NazgulName getName() {
        return name;
    }
}

```





Я вважаю, що спокусливо, якщо єдиним інструментом, який у вас є, є молоток, ставитися до всього, як до цвяха.

Абрахам Маслоу

ЧОМУ НЕ СІНГЛТОН?

- Використовуючи синглтони, ви майже завжди жертвуєте прозорістю заради зручності. Але скільки ви готові пожертвувати заради цієї маленької зручності?
- З часом ви втрачаєте об'єкти, які звертаються до об'єкта користувача, і, що більш важливо, об'єкти, які змінюють його властивості. Синглтони мають тенденцію поширюватися як вірус, тому що до них дуже легко отримати доступ. Важко відслідковувати, де вони використовуються, і позбавлення від синглтона може стати кошмаром рефакторинга у великих або складних проектах.

Чим замінити сінглтон?

- Якщо реалізація методу залежить від єдиного об'єкта, чому б не передати його як параметр? У цьому випадку ми явно показуємо, від чого залежить метод. Як наслідок, ми можемо легко відтворити ці залежності (якщо необхідно) під час виконання тестування.
- Тобто варто використати **Dependency Injection**



Singleton чи Dependency Injection?

- Основна проблема Singleton в цьому прикладі полягає в тому, що він суперечить принципу впровадження залежностей і, таким чином, перешкоджає тестуванню. По суті, він діє як глобальна константа, і його важко замінити тестовим дублікатом, коли це необхідно.
- Цей код було б важко протестувати, оскільки щоразу, коли ви запускаєте метод Process, він викликає bank gateway.

```
public class BankGateway
{
    public static BankGateway Instance = new BankGateway();

    private BankGateway()
    {
    }

    public void TransferMoney()
    {
        /* передаємо гроші */
    }
}

class Processor
{
    public void Process()
    {
        BankGateway.Instance.TransferMoney();
    }
}
```

Singleton чи Dependency Injection?

Наведений код можна протестувати. Ми можемо створити підроблений bank gateway, передати його процесору та переконатися, що він використовує його для переказу грошей. При цьому виклик справжнього bank gateway не здійснюється.

```
interface IBankGateway
{
    void TransferMoney();
}

public class BankGatewayDI implements IBankGateway{
    public void TransferMoney()
    {
        /* предаємо гроші */
    }
}

class ProcessorDI {
    private IBankGateway gateway;

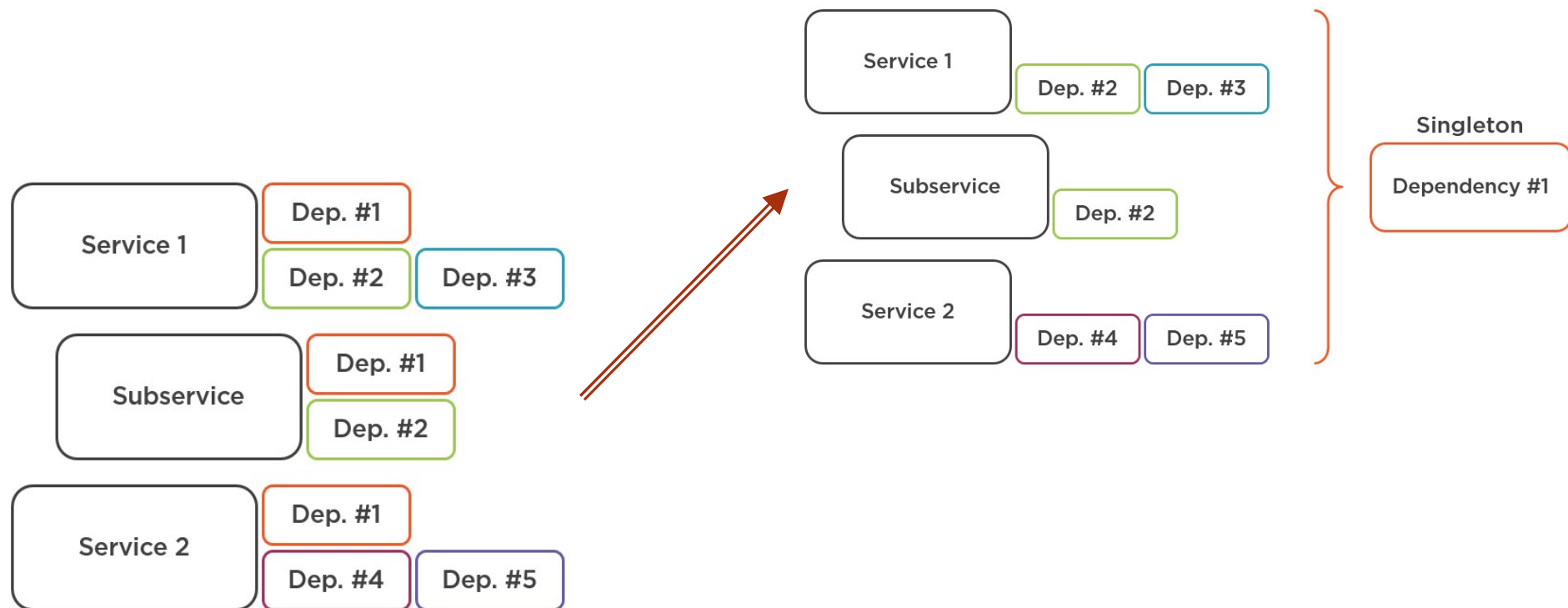
    public ProcessorDI(IBankGateway gateway)
    {
        gateway = gateway;
    }

    public void Process()
    {
        gateway.TransferMoney();
    }
}
```

Але...

- Однак все ще є залежності, які краще представити за допомогою Singleton.
- Вони залежать від середовища. Залежності навколишнього середовища — це залежності, які охоплюють кілька класів і часто кілька рівнів. Насправді немає сенсу застосовувати шаблон розробки Dependency Injection для таких залежностей, оскільки вони все одно будуть всюди

Залежність №1 присутня в кожному з трьох сервісів. Отже, це хороший кандидат для вилучення її в Singleton



Висновки

- Потреба в одному екземплярі не означає, що вам автоматично потрібен сінглтон.
- Хорошим стандартом, коли ви вирішуєте використовувати сінглтон чи ні, є те, що мати більше одного екземпляра небезпечно. Якщо ви уважніше дослідите свій дизайн, ви майже завжди зможете створити кращий дизайн.
- Тим не менш, все ще є крайні випадки, коли використання сінглтону має сенс.
- Потрібно знати всі підводні камені, що може створити сінглтон і використовувати його з розумом.

Перелік використаних джерел:

- <https://www.digitalocean.com/community/tutorials/java-singleton-design-pattern-best-practices-examples>
- <https://refactoring.guru/uk/design-patterns/singleton>
- <https://www.baeldung.com/java-singleton>
- <https://www.tutorialspoint.com/how-to-prevent-reflection-to-break-a-singleton-class-pattern>
- <https://java-design-patterns.com/patterns/multiton/>
- <https://enterprisecraftsmanship.com/posts/singleton-vs-dependency-injection/>
- <https://cocoacasts.com/are-singletons-bad>
- <https://codeguida.com/post/1053>
- <https://community.wvu.edu/~hhammar/rts/adv%20rts/design%20patterns%20tutorials/course%20slides%20on%20patterns%20and%20java/Singleton-2pp.pdf>