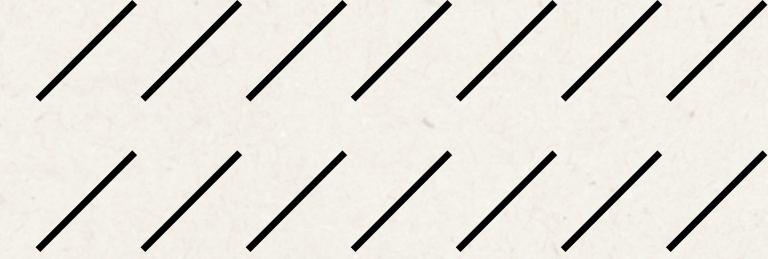


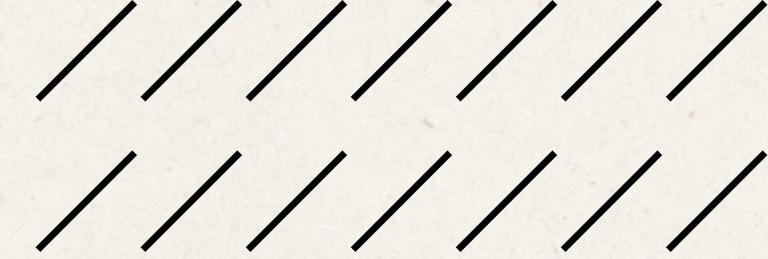
SINGLETON

Підготував Прокоф'єв Андрій



Що таке **Singleton**?

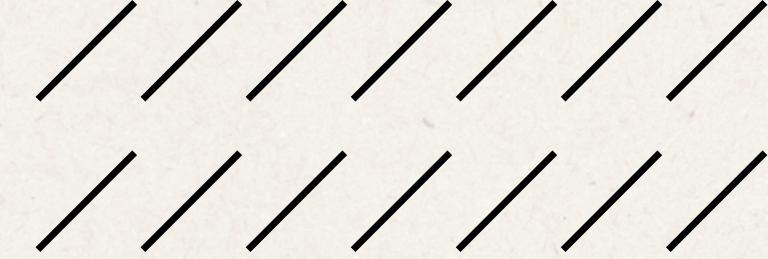
Породжувальний патерн проектування, який гарантує, що клас має лише один екземпляр, та надає глобальну точку доступу до нього.



Вирішує такі задачі :

Гарантує наявність
єдиного екземпляра
класу

Надає глобальну
точку доступу



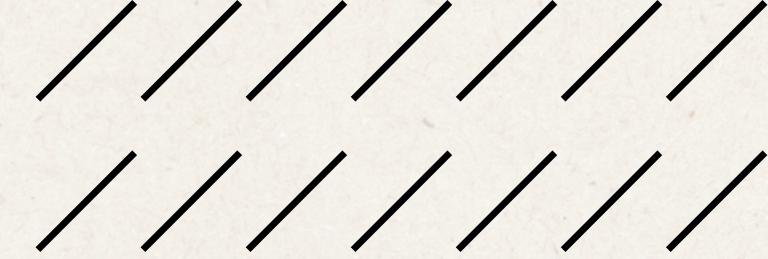
Приклад з життя

Принтер в офісі:

Черга друку має бути одна на весь
офіс, щоб уникнути конфліктів.

Мікрохвильовка в гуртожитку:

Усі студенти використовують
одну спільну піч на кухні.



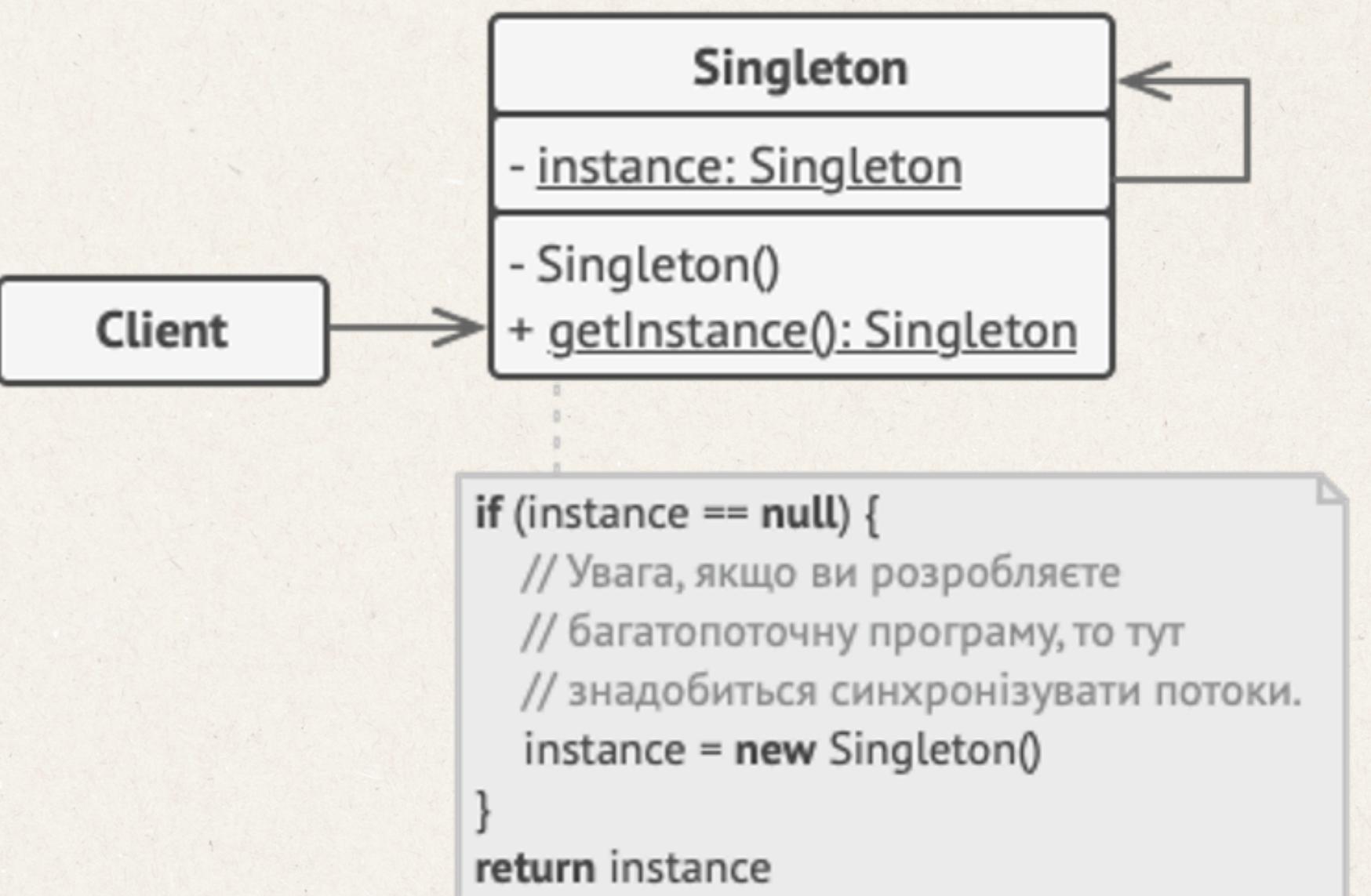
Проблеми для нас :

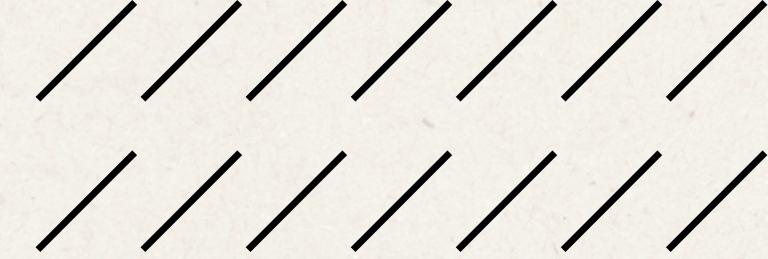
Існують ситуації, коли наявність більше ніж одного екземпляра класу є небезпечною для логіки або стабільності застосунку.

- **Пул з'єднань з БД:** Одинак використовується для обмеження кількості активних підключень, що дозволяє уникнути перевантаження сервера бази даних через надмірну кількість одночасних запитів.
- **Логування подій:** Використання патерна гарантує, що весь застосунок пише повідомлення в один спільний файл, що дозволяє зберегти чітку хронологію подій та уникнути проблем із доступом до файлу.
- Окрім цього ще **керування налаштуваннями** пристрою чи додатку, **керування станом користувача** в додатку і далі по аналогії.

Рішення

Якщо коротко, це прихований типовий конструктор та створений публічний статичний метод, який повернатиме інстанс об'єкта-одинака.





Способів ініціалізації та реалізації багато

Eager initialization

- 01 Екземпляр класу Singleton створюється автоматично, безпосередньо в момент завантаження самого класу системою..
- 02 Об'єкт ініціалізується та займає місце в пам'яті навіть у тому випадку, якщо клієнтська частина програми жодного разу його не використає.
- 03 Даня стратегія допустима лише для легких об'єктів, які не потребують значних системних потужностей.

```
package initialization;

4 usages  ↗ Prokofiev Andrii
public class ServiceEagerInit {
    1 usage
    private static final ServiceEagerInit instance = new ServiceEagerInit();
    1 usage  ↗ Prokofiev Andrii
    private ServiceEagerInit() {}
    ↗ Prokofiev Andrii
    public static ServiceEagerInit getInstance() { return instance; }
    ⚡ no usages  ↗ Prokofiev Andrii
    public void execute() { System.out.println("Eager Service: done."); }
}
```

ServiceStaticBlock initialization

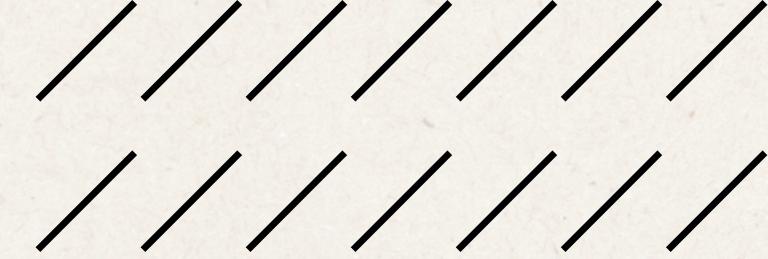
- 01 Схожа на швидку ініціалізацію, але створення екземпляра відбувається всередині статичного блоку. Цей статичний блок надає можливість оброблювати ПОМИЛКИ.
- 02 Обидві ініціалізації створюють екземпляр ще до його використання, і це не найкраща практика для використання.

```
4 usages  ↗ Prokofiev Andrii
public class ServiceStaticBlock {
    2 usages
    private static ServiceStaticBlock instance;
    1 usage  ↗ Prokofiev Andrii
    private ServiceStaticBlock() {}
    static {
        try {
            instance = new ServiceStaticBlock();
        } catch (Exception e) {
            throw new RuntimeException("Помилка при створенні статичного блоку");
        }
    }
    ↗ Prokofiev Andrii
    public static ServiceStaticBlock getInstance() { return instance; }
}
```

Як наслідок, відкладена ініціалізація

Lazy initialization

```
10 usages  ↳ Prokofiev Andrii
public class ServiceLazyInit {
    3 usages
    private static ServiceLazyInit instance;
    1 usage  ↳ Prokofiev Andrii
    private ServiceLazyInit() {}
    ↳ Prokofiev Andrii
    public static ServiceLazyInit getInstance() {
        if (instance == null) {
            instance = new ServiceLazyInit();
        }
        return instance;
    }
    no usages  ↳ Prokofiev Andrii
    public void performAction() { System.out.println("Lazy initialization service: done."); }
}
```



Але

Дана реалізація ефективна в однопотоковому середовищі, проте в багатопотокових системах виникає ризик порушення унікальності об'єкта.

Якщо кілька потоків одночасно пройдуть перевірку умови `if`, це призведе до створення декількох екземплярів `Singleton`

Double-Checked Locking

- 01 Найпростіший метод забезпечення потокобезпеки полягає в синхронізації методу `getInstance()`. Це гарантує, що в будь-який момент часу лише один потік зможе виконувати код ініціалізації об'єкта.
- 02 Але, якщо не вдосконалити наш клас зокрема простою синхронизації, мі зіткнемось зі зниженою продуктивністю через вартість, пов'язану з синхронізованим методом. Тому мій код має деякі покращення.
- 03 Ми оптимізували багатопотоковий доступ за допомогою механізму подвійної перевірки, де первинна перевірка виконується без блокування. Додатково ми використали ключове слово `volatile`, яке гарантує коректну видимість об'єкта в пам'яті (Java Memory Model)

```
6 usages • Prokofiev Andrii *
public class ThreadSafeDoubleCheck {
    4 usages
    private static volatile ThreadSafeDoubleCheck instance;

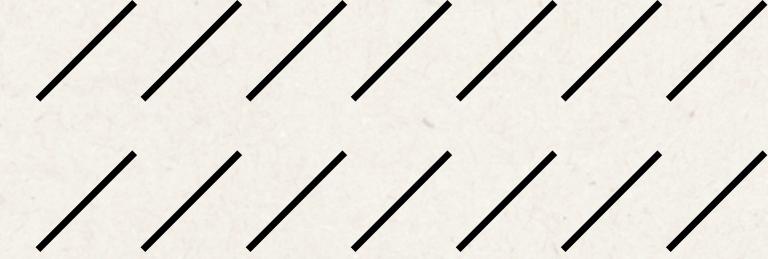
    • Prokofiev Andrii
    public static ThreadSafeDoubleCheck getInstance() {
        if (instance == null) {
            synchronized (ThreadSafeDoubleCheck.class) {
                if (instance == null) {
                    instance = new ThreadSafeDoubleCheck();
                }
            }
        }
        return instance;
    }
}
```

Ще один підхід до багатопоточності,
використовуючи особливість
завантаження класів у **Java**

Bill Pugh Singleton

```
5 usages  ↗ Prokofiev Andrii
public class BillPughSingleton {
    1 usage  ↗ Prokofiev Andrii
    private BillPughSingleton() {}
    1 usage  ↗ Prokofiev Andrii
    private static class SingletonHelper {
        1 usage
        private static final BillPughSingleton INSTANCE = new BillPughSingleton();
    }
    ↗ Prokofiev Andrii
    public static BillPughSingleton getInstance() { return SingletonHelper.INSTANCE; }
}
```

Це рішення є потокобезпечним за замовчуванням на рівні **JVM**, воно не
потребує **synchronized** або **volatile**, що робить код чистішим та
швидшим.



Додаткові способи поламати **Singleton**

Нажаль, разом з елегантним вирішенням проблеми багатопоточності Java також надає спосіб зламати сінглтон через **Reflexion**

Але, на щастя, з цим можна боротись!

Використання **Enum** як протидія

SecurityEnum

```
1 usage  ↗ Prokofiev Andrii
public enum SecurityEnum {
    1 usage
    INSTANCE;
    1 usage  ↗ Prokofiev Andrii
    public void processData() { System.out.println("Enum processData called."); }
}
```

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

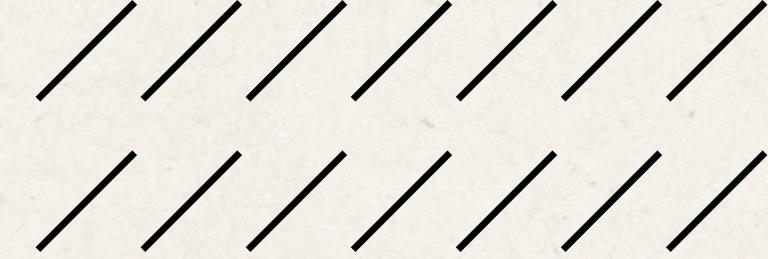
З мінусів, тип enum
дещо негнучкий,
наприклад, він не
допускає відкладену
ініціалізацію

Комбо захисту

- 01 Якщо ж Enum нам не підходить, іншим способом захиститись від рефлексії є викристання конструкції з внутрішнім класом **Loader**, яка перевіре що єдиний екземпляр уже ініціалізований цим внутрішнім класом, і негайно викине виняток.
- 02 На прикладі цього ж коду можна побачити засіб боротьби з **сереалізацією**.
- 03 Ми реалізуємо спеціальний метод **readResolve()**, який автоматично викликається JVM одразу після десеріалізації. Замість того, щоб дозволити створити новий об'єкт, цей метод примусово повертає вже існуючий інстанс.

ValidatedSingleton

```
8 usages  ↗ Prokofiev Andrii
public class ValidatedSingleton implements Serializable {
    no usages
    @Serial
    private static final long serialVersionUID = 1L;
    2 usages  ↗ Prokofiev Andrii
    private static class Loader {
        2 usages
        private static final ValidatedSingleton INSTANCE = new ValidatedSingleton();
    }
    1 usage  ↗ Prokofiev Andrii
    private ValidatedSingleton() {
        if (Loader.INSTANCE != null) {
            throw new RuntimeException("Вже існує");
        }
    }
    ↗ Prokofiev Andrii
    public static ValidatedSingleton getInstance() { return Loader.INSTANCE; }
    no usages  ↗ Prokofiev Andrii
    @Serial
    protected Object readResolve() { return getInstance(); }
}
```



На які підвиди варто
звернути увагу?
На ті що дозволяють
звертатись до групи
об'єктів.

Multiton

Набір одинаків

- 01 Реалізація базується на використанні колекції Мар як реєстру екземплярів, де ключем виступає унікальний ідентифікатор (наприклад, TenantKey або ContextType). Програма ініціалізує та зберігає конкретний об'єкт для кожного ключа, забезпечуючи централізований доступ до іменованих ресурсів через метод getInstance

```
7 usages • Prokofiev Andrii *
public class TenantMultiton {
    1 usage
    private static final Map<TenantKey, TenantMultiton> instances = new ConcurrentHashMap<>();
    1 usage new *
    private TenantMultiton(TenantKey key) {}

    • Prokofiev Andrii *
    public static TenantMultiton getInstance(TenantKey key) {
        return instances.computeIfAbsent(key, TenantMultiton::new);
    }
}
```

- 02 Для ідентичних ключів система завжди повертає посилання на один і той самий об'єкт, гарантуючи його унікальність у межах заданої ролі. Використання різних ключів дозволяє ізолювати контексти даних або налаштувань, зберігаючи при цьому переваги патерна Singleton для кожного окремого випадку.

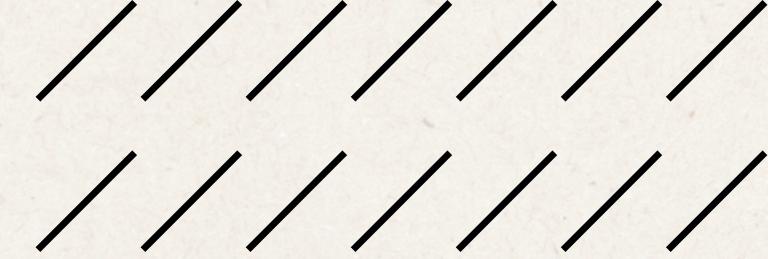
Реєстр компонентів

Component Registry

- 01 Реєстр використовує колекцію Мар для динамічного зберігання об'єктів різних типів, де ключем є повне ім'я класу (`Class<T>`), а значенням – створений екземпляр. Це дозволяє перетворити будь-який клас на `Singleton` у межах цього реєстру, не змінюючи його вихідний код.

```
2 usages  ↗ Prokofiev Andrii
public class ComponentRegistry {
    3 usages
    private static final Map<String, Object> components = new HashMap<>();
    2 usages  ↗ Prokofiev Andrii
    public static synchronized <T> T getComponent(Class<T> cls) {
        if (!components.containsKey(cls.getName())) {
            try {
                components.put(cls.getName(), cls.getDeclaredConstructor().newInstance());
            } catch (Exception e) {
                throw new RuntimeException("Registration error");
            }
        }
        return cls.cast(components.get(cls.getName()));
    }
}
```

- 02 Для кожного окремого класу система гарантує повернення ідентичного екземпляра при повторних викликах завдяки синхронізації методу `getComponet`. Такий підхід забезпечує централізовану точку доступу до різномірних сервісів застосунку, запобігаючи дублюванню об'єктів та надмірному використанню пам'яті.



А чи завжди доцльний **Singletone**?

Case 1

01 Основна проблема Singleton в цьому прикладі полягає в тому, що він суперечить принципу впровадження залежностей і, таким чином, перешкоджає тестуванню. Поки, він діє як глобальна константа, і його важко замінити тестовим дублікатом, коли це необхідно.

```
4 usages new *
public class SystemGatewaySingleton {
    1 usage
    private static final SystemGatewaySingleton instance = new SystemGatewaySingleton();
    1 usage new *
    private SystemGatewaySingleton() {}
    new *
    public static SystemGatewaySingleton getInstance() { return instance; }
    1 usage new *
    public void send(String data) { System.out.println(STR."Gateway: \{data\}"); }
}

2 usages new *
class BusinessProcessorSingleton {
    1 usage new *
    public void run(String data) { SystemGatewaySingleton.getInstance().send(data); }
}
```

02 Замість глобального доступу до Singleton, залежність можна передавати безпосередньо як параметр методу. Це робить зв'язки в коді явними та прозорими, що значно спрощує процес тестування, дозволяючи легко замінювати реальні об'єкти на тестові подвійники під час виконання.

Case 2

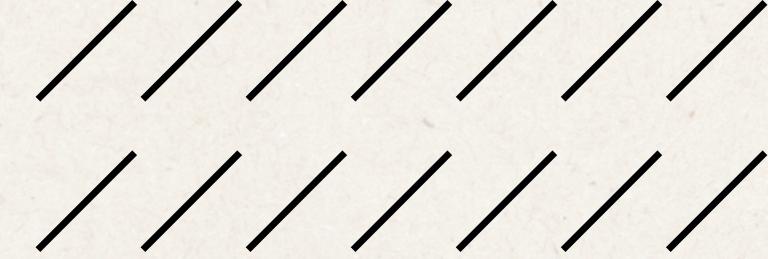
- 01 Така архітектура робить код повністю придатним для модульного тестування, дозволяючи замінити реальний банківський шлюз на підроблений об'єкт. Це дає змогу перевірити коректність логіки процесора без виконання справжніх транзакцій, гарантуючи, що під час тестів не здійснюються жодні зовнішні виклики.
- 02 Слабкий зв'язок у підході DI досягається завдяки тому, що клас BusinessProcessorDI залежить лише від абстрактного інтерфейсу Gateway, а не від його конкретної реалізації. Це забезпечує виняткову гнучкість: ми можемо миттєво створювати «фейкові» шлюзи через Lambda-вирази, що технічно неможливо в Singleton через приватний конструктор та жорстко зашитий статичний доступ.

```
package singletonVsDI;

5 usages 1 implementation new *
interface Gateway {
    1 usage 1 implementation new *
    void send(String data);
}

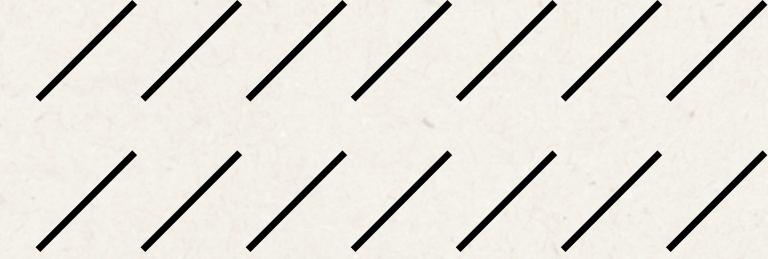
1 usage new *
public class SystemGatewayDI implements Gateway {
    1 usage new *
    public void send(String data) { System.out.println(STR."Gateway: \{data\}"); }
}

4 usages new *
class BusinessProcessorDI {
    2 usages
    private final Gateway gateway;
    2 usages new *
    public BusinessProcessorDI(Gateway gateway) { this.gateway = gateway; }
    2 usages new *
    public void run(String data) { gateway.send(data); }
}
```



Висновки

- Наявність лише одного екземпляра в системі можна забезпечити іншими архітектурними способами, не вдаючись до цього патерна.
- Використовувати Singleton варто лише тоді, коли створення другого об'єкта критично загрожує стабільності системи; у більшості інших випадків існують гнучкіші рішення.
- Попри критику, існують специфічні сценарії, де Singleton залишається найбільш доцільним вибором.
- Необхідно враховувати всі технічні ризики та обмеження патерна, застосовуючи його лише там, де переваги переважають недоліки.



Джерела

- <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/>