Padrões de Projeto Criacionais

Padrões Criacionais

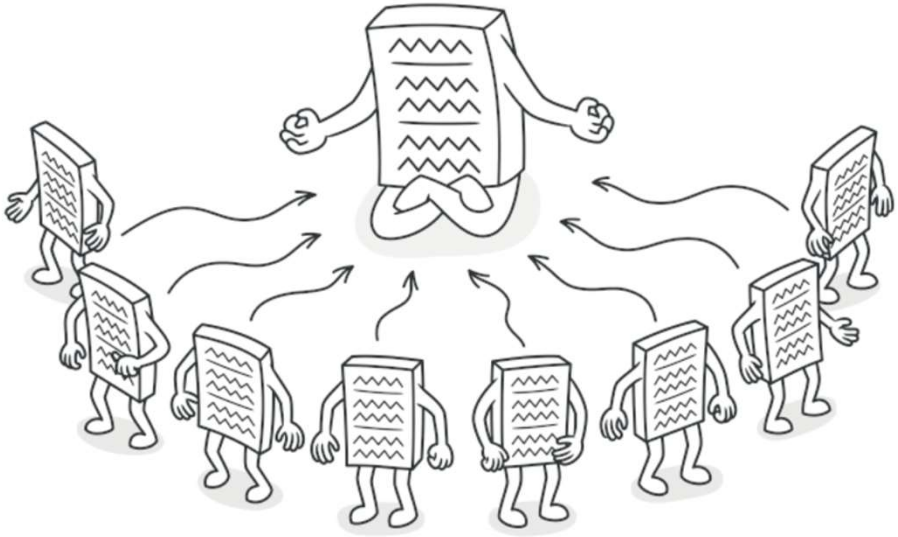
- Os padrões **criacionais** fornecem vários **mecanismos de criação de objetos**, que aumentam a flexibilidade e reutilização de código já existente.

Padrões de projeto criacionais

- Factroy Method
- Abstract Factory
- Builder
- Prototype
- Singleton

Singleton

- O **Singleton** é um padrão de projeto que garante que uma **classe tenha apenas uma instância**, enquanto provê um **ponto de acesso global para essa instância**.



Por que alguém iria querer controlar quantas instâncias uma classe tem?

- A razão mais comum para isso é para **controlar o acesso a algum recurso compartilhado**.
- **Por exemplo**: uma base de dados, um arquivo de configurações.
- E **evitar o uso de variáveis globais** passadas como parâmetro para os objetos.
- Oferecendo **mais flexibilidade e segurança**.

Como que funciona?

- Imagine que você **criou um objeto** e depois de um tempo você decidiu criar um novo.
- Ao invés de receber um novo objeto, você **obterá o mesmo objeto** que foi criado anteriormente.
- Uma vez que esse objeto possui uma **única instância para toda a aplicação.**

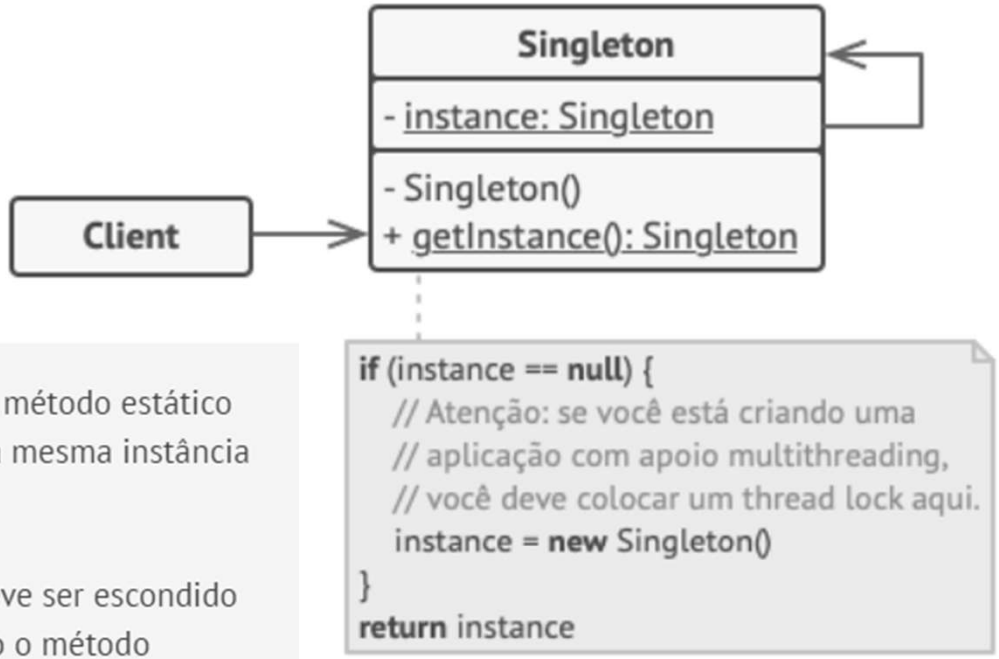
Como que funciona?

- Assim como uma **variável global**, o padrão ***Singleton*** permite que você **acesse o mesmo objeto** de **qualquer lugar no programa**.
- Além de protege aquela instância de ser **sobrescrita** por outro código. Sendo por isso **mais seguro** que o uso de **variáveis globais**.

Implementação

- Todas as implementações do ***Singleton*** tem esses **dois passos em comum**:
  - Fazer o **construtor padrão privado**, para prevenir que outros objetos usem o operador ***new*** com a classe ***singleton***.
  - Criar um **método estático de criação** que age como um construtor. Esse método **chama o construtor privado** por debaixo dos panos para criar um objeto e o **salva em um campo estático**. Todas as chamadas seguintes para esse método **retornam o objeto em cache.**
- Se o seu código tem acesso à classe ***singleton***, então ele será capaz de chamar o método estático da ***singleton***. Então sempre que aquele método é chamado, **o mesmo objeto é retornado.**

Estrutura



1 A classe **Singleton** declara o método estático `getInstance` que retorna a mesma instância de sua própria classe.

O construtor da singleton deve ser escondido do código cliente. Chamando o método `getInstance` deve ser o único modo de obter o objeto singleton.

```java
public class Singleton {

    private static Singleton uniqueInstance;

    private Singleton() {
    }

    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null)
            uniqueInstance = new Singleton();

        return uniqueInstance;
    }
}
```

Utilizando **Synchronized** temos a certeza que o método nunca será acessado por duas threads ao mesmo tempo.

getInstance(), poderia receber parâmetros para instanciar o objeto ***Singleton*.**

```java
public class Singleton {

    private static Singleton uniqueInstance = new Singleton();

    private Singleton() {
    }

    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

Se uma determinada classe ***Singleton*** sempre **é criada e usada** e **não recebe parâmetros**. Prefira essa abordagem de implementação.

Vantagens do Padrão Singleton

- O **Padrão *Singleton*** pode ser instanciada e usada quando necessário, diferentemente de uma variável global em que o objeto é sempre criado quando o aplicativo é inicializado e poderá estar usando recursos que não são necessários neste momento.
- **Ficando mais fácil de criar e gerenciar a utilização da instância**.

Exemplos de Utilização

- Classes de **acesso a banco de dados**.
- Classes de **autenticação**.
- Classes de **criptografia**.
- Em projetos que necessitem de uma única instância de um objeto sendo compartilhado entre vários outros objetos.

Exemplos de Utilização

```java
public static void main(String[] args) throws InterruptedException {
    Configuracao conf1=Configuracao.getInstance();
    System.out.println("Data Config1: " + conf1.getData().toString());
    System.out.println("Data Sistema: " + Calendar.getInstance().toString());
    Thread.sleep(1000);

    Configuracao conf2=Configuracao.getInstance();
    System.out.println("Data Config2:" + conf2.getData().toString());
    System.out.println("Data Sistema: " + Calendar.getInstance().toString());
    Thread.sleep(1000);

    Configuracao conf3=Configuracao.getInstance();
    System.out.println("Data Config3: " + conf3.getData().toString());
    System.out.println("Data Sistema: " + Calendar.getInstance().toString());

}
```

```
run:
Data Config1: java.util.GregorianCalendar[time=1714603270638
Data Sistema: java.util.GregorianCalendar[time=1714603270654
Data Config2:java.util.GregorianCalendar[time=1714603270638,
Data Sistema: java.util.GregorianCalendar[time=1714603271662
Data Config3: java.util.GregorianCalendar[time=1714603270638
Data Sistema: java.util.GregorianCalendar[time=1714603272672
BUILD SUCCESSFUL (total time: 2 seconds)
```

```java
public class Configuracao {

    private static Configuracao instance;
    private final Calendar data;

    public static Configuracao getInstance(){
        if(instance==null)
            instance=new Configuracao();
        return instance;
    }

    private Configuracao(){
        data = Calendar.getInstance();
    }

    public Calendar getData() {
        return data;
    }
```

Diferentes objetos (conf1, conf2 e conf3) com a mesma instância de data.