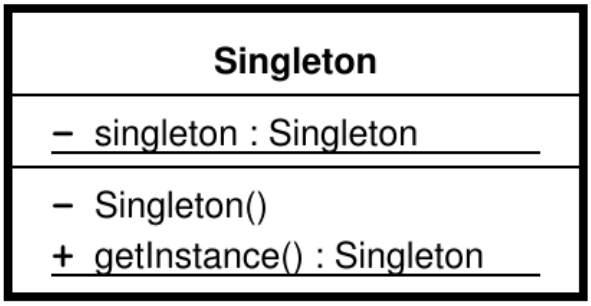
**UNIFACISA – Centro Universitário** **Data: 29/08/2017**
**Disciplina: Programação III** **Professor: Daniel Abella**
**Curso: Sistemas de Informação** **Semestre: 2017.2 (1ª Unidade)**
**Aluno: Ayrton Carlos Araújo de Andrade**

# Singleton

**Singleton** é um padrão de projeto de software (do inglês ***Design Pattern***). O termo vem do significado em inglês para um conjunto (entidade matemática) que contenha apenas um elemento.




Este padrão garante a existência de apenas uma instância de uma classe, mantendo um ponto global de acesso ao seu objeto.

Alguns projetos necessitam que algumas classes tenham apenas uma instância. Por exemplo, em uma aplicação que precisa de uma infraestrutura de log de dados, pode-se implementar uma classe no padrão singleton. Desta forma existe apenas um objeto responsável pelo log em toda a aplicação que é acessível unicamente através da classe singleton.

- **Onde Utilizar?**

  Quando você necessita de somente uma instância da classe, por exemplo, a conexão com banco de dados, vamos supor que você terá que chamar diversas vezes a conexão com o banco de dados em um código na mesma execução, se você instanciar toda vez a classe de banco, haverá grande perda de desempenho, assim usando o padrão singleton, é garantida que nesta execução será instanciada a classe somente uma vez. Lembrando que este pattern é considerado por muitos desenvolvedores um antipattern, então, cuidado onde for utilizá-lo.

- **Como fazer a implementação?**

  1. Deixar o construtor privado, pois assim ninguém deve conseguir instanciar a classe, apenas o próprio **Singleton**.
  2. Criar um atributo privado e estático do mesmo tipo da classe **(instance)**. Algumas linguagens não tipadas não irão precisar do tipo, caso do **PHP**, por exemplo.
  3. Método **getInstance()** é o principal ponto da classe. Ele verifica se a variável **instance** já foi iniciada, caso não tenha sido, ele faz sua criação pela primeira e única vez.
  4. Para fazer a conexão, devemos chamar o **getInstance** da seguinte forma: **ClasseSingleton.getInstance()**.

Agora que sabemos como fazer a implementação, devemos focar agora em como aplicar este padrão de forma correta, vamos ver alguns exemplos:

- **Singleton sem concorrência:**

  A implementação do singleton pattern mais simples é desta forma:

```
private static PEHandlerService instancia;
public static PEHandlerService getInstancia() {
    if (instancia == null) instancia = new PEHandlerService();
    return instancia;
}
```

Como você já sabe, esta versão poderia gerar duas instâncias num cenário um tanto incomum, isto é, se duas threads executassem o **getInstancia** ao mesmo tempo na primeira chamada ao método.

- **Singleton concorrente:**

  Para resolver isso, a solução mais fácil é sincronizar o método:

```
private static PEHandlerService instancia;
synchronized public static PEHandlerService getInstancia() {
    if (instancia == null) instancia = new PEHandlerService();
    return instancia;
}
```

Isso evita problemas de concorrências, mas gera um pequeno atraso em cada chamada ao método para gerenciar a concorrência, além do que se houverem múltiplas threads somente uma poderá chamar o método por vez, possivelmente gerando gargalos num sistema altamente concorrente.

- **Singleton concorrente com sincronização mínima:**

Para melhorar um pouco a versão acima, alguns autores propõem a seguinte construção:

```
private volatile static PEHandlerService instancia;
public static PEHandlerService getInstancia() {
    if (instancia == null) {
        synchronized (PEHandlerService.class) {
            if (instancia == null) instancia = new PEHandlerService();
        }
    }
    return instancia;
}
```

Isso faz com que a sincronização ocorra somente na inicialização e não nas demais chamadas.

Porém, note o modificador **volatile** no atributo da classe. Ele é necessário mesmo com a sincronização, pois devido ao modelo de memória do **Java**, principalmente antes do **Java 5**, ainda poderiam ocorrer erros causados por um tipo de cache onde outras thread poderia ainda ver o valor **null** na variável, mesmo após a atribuição por outra thread de modo atômico.

- **Cuidado com inicializações em múltiplos comandos:**

Um cuidado muito importante é não atribuir o objeto à variável estática antes dele estar completamente inicializado. Considere o seguinte código:

```
private volatile static PEHandlerService instancia;
public static PEHandlerService getInstancia() {
    if (instancia == null) {
        synchronized (PEHandlerService.class) {
            if (instancia == null) {
                instancia = new PEHandlerService();
                instancia.setAlgumaDependencia(new Dependencia());
            }
        }
    }
    return instancia;
}
```

O código acima atribui uma nova instância de PEHandlerService à instancia e depois passa algum objeto para ela. O problema é que como **instancia != null**, outra thread pode chamar o método **getInstancia** e recuperar o objeto antes dele receber a dependência. Neste caso você poderia ter um **NullPointerException**.

- **Singleton pré-carregado sem sincronização:**

Para evitar todos esses problemas acima, a solução mais simples apontada é simplesmente inicializar o seu objeto **singleton** fora do método **getInstancia**, exatamente como no seu primeiro exemplo:

```
private static PEHandlerService instancia = new PEHandlerService();
public static PEHandlerService getInstancia() {
    return instancia;
}
```

Se for necessário algum tipo de inicialização, é possível usar um bloco de inicialização estático:

```
private static PEHandlerService instancia;
static {
    instancia = new PEHandlerService();
    instancia.setDependencia(new Dependencia());
}
public static PEHandlerService getInstancia() {
    return instancia;
}
```

A maior diferença desta abordagem é que o objeto não será mais inicializado por demanda **(lazy initialization)**, mas logo que a classe for usada pela primeira vez **(eager initialization)**. Isso pode ser bom ou ruim, dependendo do caso.

- **Singleton concorrente sem sincronização:**

  Para tentar juntar tudo, isto é, evitar sincronização e carregar o **singleton** em modo **lazy**, existem algumas alternativas.

  Uma delas consiste em usar uma terceira classe para carregar a variável estática somente quando a mesma for acessada. Exemplo:

```
private static class SingletonLoader {
     private static PEHandlerService instancia = new PEHandlerService();
}
public static PEHandlerService getInstancia() {
    return SingletonLoader.instancia;
}
```

- **Alternativa: use um Enum:**

  Outra alternativa ao Single é simplesmente declarar a sua classe como um **Enum** de um valor só. Exemplo:

```
public enum PEHandlerServiceSingleton {
    INSTANCE;

    //métodos aqui

}
```

  E aí você pode acessar isso da seguinte forma:

```
PEHandlerServiceSingleton.INSTANCE
```

## **Benefícios**:

1. Permite o controle sobre como e quando os clientes acessam a instância.
2. Várias classes **singleton** podem obedecer uma mesma interface, permitindo assim que um singleton em particular seja escolhido para trabalhar com uma determinada aplicação em tempo de execução.
3. Com apenas uma implementação interna do singleton pode-se fazer com que o singleton crie um número controlado de instâncias.
4. É mais flexível que métodos estáticos por permitir o polimorfismo.

## Contras:

1. Acoplamento: Usando **Singleton** você estará acoplando o seu código em uma implementação estática e específica. Isso faz o seu código dependente dessa classe e impede, por exemplo, criar ***mocks*** em testes unitários.
2. Escopo: Se você por alguma razão decidir que para determinado componente da aplicação você precisa de outra implementação terá que alterar manualmente todas as classes.
3. Falsa segurança: No **Java**, por exemplo, não existe uma classe apenas por **JVM**. O conceito de carregamento de classes em Java é feito por ***ClassLoader***.

## Considerações:

Existem muitas formas diferentes de usar um padrão como o **Singleton**. Cada uma pode ser boa ou ruim para determinadas situações e algumas escondem certos problemas.

Entretanto, um vez que se compreenda um pouco a diferença entre as implementações, não é difícil escolher uma que se encaixe melhor na sua solução.

# Referências Bibliográficas


- https://pt.wikipedia.org/wiki/Singleton
- https://pt.stackoverflow.com/questions/56010/como-aplicar-o-padr%C3%A3o-singleton-corretamente