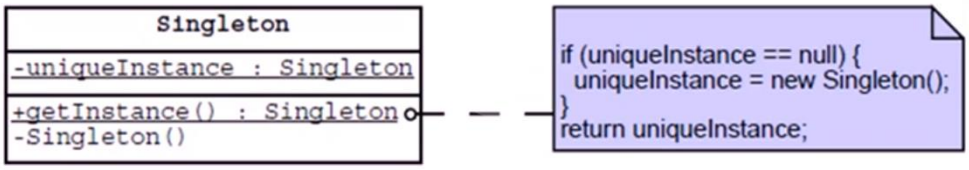
Roteiro 05

Seguindo a sequência com mais um roteiro no nosso projeto, utilizaremos na evolução deste roteiro o padrão Singleton.

Padrão Singleton – Assegura que uma classe tenha apenas uma instância e fornece o ponto de acesso global a esta classe. Podemos ter 2 variações deste padrão chamados de **Lazy Singleton** e **Eager Singleton**.

Inicialmente trabalharemos com o modelo formal do padrão **Lazy Singleton**



No modelo acima os métodos estáticos estão sublinhados. Na classe Singleton temos que destacar :

- uma variável (atributo) estática que vai garantir uma instância única;
- um método estático para acesso a esta variável;
- um construtor privado para impedir que mais de uma instância seja criada.

Algumas aplicações : Pool de conexões, pool de threads, Gerenciador de impressão, objetos que manipulam preferências, Objetos de Log, Objetos que atuam com drivers de impressora ou placas de vídeo. Em alguns destes casos, se tivermos se tivermos mais de uma instância podemos ter vários problemas como : comportamento incorreto do programa, uso exagerado dos recursos, ou resultados inconsistentes.

Cenário:

Um sistema de controle de decolagem e aterrissagem no aeroporto. Neste exemplo suponha que apenas uma permissão de decolagem ou aterrissagem possa ser dada por vez. Ou seja, caso concedida uma permissão de aterrissagem, somente podemos conceder permissão de decolagem na sequência.

Início do projeto – Pacote : roteiro5.parte1

1 – Dentro do projeto criar um pacote chamado **roteiro5.parte1**

2 – Inicialmente crie a classe **ControladorAereo** dentro do pacote **roteiro5.parte1.**

```
package roteiro5.parte1;

public class ControladorAereo {
   private boolean permitidoAterrissar;
   private boolean permitidoDecolar;

   public ControladorAereo() {
      this.permitidoAterrissar = false;
      this.permitidoDecolar = true;
   }

   public void solicitarAterrissagem(){
      if (this.permitidoAterrissar){
        System.out.println("Permissão para aterrissagem concedida ");
        this.permitidoAterrissar = false;
        this.permitidoDecolar = true;
      }
      else{
         System.out.println("Permissão para aterrissagem negada ");
      }
   }

   public void solicitarDecolagem(){
      if (this.permitidoDecolar){
        System.out.println("Permissão para decolar concedida ");
        this.permitidoAterrissar = true;
        this.permitidoDecolar = false;
      }
      else{
         System.out.println("Permissão para decolagem negada ");
      }
   }
}
```

3 – Vamos criar a classe Teste agora para validar esta lógica.

```
package roteiro5.parte1;

public class Teste {

   public static void main(String[] args) {
      ControladorAereo c1 = new ControladorAereo();
      ControladorAereo c2 = new ControladorAereo();

      c1.solicitarDecolagem();
      c2.solicitarDecolagem();

      System.out.println(" ");

      c1.solicitarAterrissagem();
      c2.solicitarAterrissagem();
   }
}
```

4 – O resultado saiu como esperado conforme descrito no cenário? Sim/Não e porquê?

Pacote : roteiro5.parte2

Notem que na parte1 do roteiro não implementamos nenhum padrão. Vamos agora tentar usar os requisitos do padrão Singleton.

1 – No mesmo projeto crie o pacote roteiro5.parte2

2 – Copie todas as classes criadas na parte1 para o novo pacote.

3 – Vamos aplicar agora os requisitos do padrão Singleton na classe **ControladorAereo**.

```
package roteiro5.parte2;

public class ControladorAereo {
   private boolean permitidoAterrissar;
   private boolean permitidoDecolar;
   private static ControladorAereo instance = new ControladorAereo();

   private ControladorAereo() {
      this.permitidoAterrissar = false;
      this.permitidoDecolar = true;
   }

   public static ControladorAereo getInstace(){
      return instance;
   }

   public void solicitarAterrissagem(){
      if (this.permitidoAterrissar){
        System.out.println("Permissão para aterrissagem concedida ");
        this.permitidoAterrissar = false;
        this.permitidoDecolar = true;
      }
      else{
         System.out.println("Permissão para aterrissagem negada ");
      }
   }

   public void solicitarDecolagem(){
      if (this.permitidoDecolar){
        System.out.println("Permissão para decolar concedida ");
        this.permitidoAterrissar = true;
        this.permitidoDecolar = false;
      }
      else{
         System.out.println("Permissão para decolagem negada ");
      }
   }
}
```

Lembrando os requisitos:

- uma variável (atributo) estática que vai garantir uma instância única; instance

- um método estático para acesso a esta variável; getInstance

- um construtor privado para impedir que mais de uma instância seja criada.

4 – O resultado saiu como esperado conforme descrito no cenário? Sim/Não e porquê?

5 – Utilize uma ferramenta de software qualquer para geração do diagrama de classes para esta etapa do projeto (Sugestão : Astah Community). Obs.: Adicione aqui o diagrama para que seja disponibilizado no teams

Pacote : roteiro5.parte3

Neste pacote vamos fazer algumas experiências com as variações do Padrão Singleton

1 – Dentro do projeto criar um pacote chamado **roteiro5.parte3**

2 – Crie agora classe **InocenteSingleton** dentro do pacote **roteiro5.parte3.**

```
package roteiro5.parte3;

public class InocenteSingleton {

   private InocenteSingleton() {
   }

   public static InocenteSingleton getInstance(){
      return new InocenteSingleton();
   }
}
```

Nesta classe aparentemente foi implementado o padrão Sigleton, pois apresenta alguns requisitos do padrão.

3 – Vamos agora criar a classe **TesteSingleton** para testar esta lógica

```
package roteiro5.parte3;

public class TesteSingleton {

   public static void main(String[] args) {
      InocenteSingleton n1 = InocenteSingleton.getInstance();
      InocenteSingleton n2 = InocenteSingleton.getInstance();

      System.out.println(n1 == n2 ? "Instâncias iguais" : "Instâncias diferentes");

   }
}
```

4 – O resultado saiu como esperado conforme o Padrão Singleton descreve? Sim/Não e porquê?

5 – Vamos agora criar a classe **LazySingleton** como uma variação do Padrão Singleton.

```
package roteiro5.parte3;

public class LazySingleton {

   private static LazySingleton instance;

   private LazySingleton() {

   }

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

Relembrando os requisitos:

- uma variável (atributo) estática que vai garantir uma instância única; instance

- um método estático para acesso a esta variável; getInstance

- um construtor privado para impedir que mais de uma instância seja criada.

6 – Voltando para a classe **TesteSingleton** para testar esta lógica

```
package roteiro5.parte3;

public class TesteSingleton {

   public static void main(String[] args) {
      InocenteSingleton n1 = InocenteSingleton.getInstance();
      InocenteSingleton n2 = InocenteSingleton.getInstance();

      System.out.println(n1 == n2 ? "Instâncias iguais" : "Instâncias diferentes");

      LazySingleton n3 = LazySingleton.getInstance();
      LazySingleton n4 = LazySingleton.getInstance();

      System.out.println(n3 == n4 ? "Instâncias iguais" : "Instâncias diferentes");

   }
}
```

7 – Qual é a sua analise sobre os resultados apresentados para as variáveis n1, n2, n3 e n4 ?

A seguir faremos algumas observações sobre o padrão **Lazy Singleton** e continuaremos o roteiro logo depois !

Observação : O padrão Lazy Singleton funciona bem apenas em um ambiente onde temos apenas uma Thread de processamento. Em ambientes Multithread podemos ter problemas e possivelmente correremos o risco de ter mais de uma instância rodando ao mesmo tempo.

Suponha um cenário onde temos 2 Threads e ambas tem um tempo de CPU para executar fatias de ciclos de cpu para executar o mesmo método getInstance de uma implementação em Lazy Singleton(). As Threads rodarão simultaneamente.

Observe a simulação abaixo.

Thread 01	Thread 02	Referência objeto Singleton
public static Singleton getInstance(){		Null
	public static Singleton getInstance(){	Null
instance = new Singleton() Return instance		Objeto Singleton 01
	instance = new Singleton() Return instance	Objeto Singleton 02

Uma solução relativamente simples para utilizar o LazySingleton em ambiente MultiThread é de alguma forma bloquear a Thread até que ela conclua o método. O recurso interessante usado em Java é utilizar o comando "synchonized" na assinatura do método "getInstance". Desta forma, a execução das Threads irão ocorrer de forma sincronizada. Vale lembrar que este recurso normalmente acarreta **perda de performance em ambiente de produção na execução da aplicação**.

```
package roteiro5.parte3;

public class LazySingleton {

   private static LazySingleton instance;

   private LazySingleton() {

   }

   public static synchronized LazySingleton getInstance(){
      if (instance == null){
         instance = new LazySingleton();
      }
      return instance;
   }
}
```

O padrão **LazySingleton** tem limitações para ambientes Muitithreads, mas pode ser resolvido com a sincronização do método getInstance. Ainda assim, podemos ter problemas. Agora não mais com 2 instâncias rodando simultaneamente, mas com uma perda substancial na performance de execução da aplicação. Se a precisamos garantir que o padrão Singleton funcione e a perda de performance for um grande problema para a aplicação é recomendado o padrão **Eager Singleton**.

8 – Ainda no pacote roteiro5.parte3 vamos criar agora a classe **EagerSingleton**.

```
package roteiro5.parte3;

public class EagerSingleton {

   private static EagerSingleton instance = new
EagerSingleton();

   public EagerSingleton() {

   }

   public static EagerSingleton getInstance(){
      return instance;
   }
}
```

9 – Devemos agora acrescentar um novo teste na classe **TesteSingleton**

```
package roteiro5.parte3;

public class TesteSingleton {

   public static void main(String[] args) {
      InocenteSingleton n1 = InocenteSingleton.getInstance();
      InocenteSingleton n2 = InocenteSingleton.getInstance();

      System.out.println(n1 == n2 ? "Instâncias iguais" : "Instâncias diferentes");

      LazySingleton n3 = LazySingleton.getInstance();
      LazySingleton n4 = LazySingleton.getInstance();

      System.out.println(n3 == n4 ? "Instâncias iguais" : "Instâncias diferentes");

      EagerSingleton n5 = EagerSingleton.getInstance();
      EagerSingleton n6 = EagerSingleton.getInstance();

      System.out.println(n5 == n6 ? "Instâncias iguais" : "Instâncias diferentes");

   }
}
```

10 – Qual dos padrões (Lazy Singleton ou Eager Singleton) foi utilizado no cenário do controlador de vôo do pacote 2 ?

11 – Criaremos agora uma outra classe como forma de implementação do padrão Singleton e chamaremos de **StaticSingleton** conforme segue abaixo

```
package roteiro5.parte3;

public class StaticSingleton {
   public static final StaticSingleton instance = new StaticSingleton();

}
```

12 – Novamente devemos acrescentar este teste na classe **TesteSingleton**

```
package roteiro5.parte3;

public class TesteSingleton {

   public static void main(String[] args) {
      InocenteSingleton n1 = InocenteSingleton.getInstance();
      InocenteSingleton n2 = InocenteSingleton.getInstance();

      System.out.println(n1 == n2 ? "Instâncias iguais" : "Instâncias diferentes");

      LazySingleton n3 = LazySingleton.getInstance();
      LazySingleton n4 = LazySingleton.getInstance();

      System.out.println(n3 == n4 ? "Instâncias iguais" : "Instâncias diferentes");

      EagerSingleton n5 = EagerSingleton.getInstance();
      EagerSingleton n6 = EagerSingleton.getInstance();

      System.out.println(n5 == n6 ? "Instâncias iguais" : "Instâncias diferentes");

      StaticSingleton n7 = StaticSingleton.instance;
      StaticSingleton n8 = StaticSingleton.instance;

      System.out.println(n7 == n8 ? "Instâncias iguais" : "Instâncias diferentes");

   }
}
```

OBSERVAÇÃO : Criamos uma classe chamada **StaticSingleton** apenas com uma variável constante e pública que guarda a instância do objeto. Observe que esta implementação abre mão do encapsulamento feito nas implementações **EagerSingleton** e **LazySingleton**. Vejam que nos padrões sugeridos pelo GOF, o construtor das classes "Singleton" são privados, e assim nenhuma outra classe pode instanciar um singleton diretamente. Outras classes podem apenas obter o objeto singleton a partir do método getIntance, que por sua vez foi instanciado dentro da própria classe Singleton.

O exemplo de implementação usado na classe **StaticSingleton** como um Padrão Singleton que apesar de funcionar, fere as boas práticas de programação e o padrão formal sugerido pelo GOF.