sãojudas universidade	FTCE Faculdade de Tecnologia e Ciências Exatas

USJT – 2019 - ARQSW – Arquitetura de Software

Professores: Bossini e Hamilton

Aula: 7
Assunto: Design Patterns - Singleton

Passo 1 (Sobre o Design Patterns**)** São elementos da programação orientada a objetos (no nosso caso em Java, mas você pode se sentir à vontade para procurar em outras tantas linguagens como , C#, Delphi, entre outras...), pelo(s) qual(is) nos permite abstrair de maneira mais eficiente e porque não eficaz a recuperação de dados através de uma simples codificação própria para cada elemento de um Design Pattern.

Isso quer dizer que temos alguns Design Patterns para determinados tipos de ações e soluções.

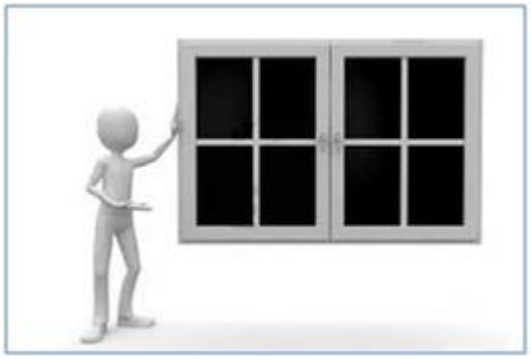
O nome de Design Pattern vem da lógica que temos que ter "as melhores práticas para resolver problemas conhecidos".

Todo o conhecimento da P.O.O é exigido, ou seja, Informações sobre seu conceito e criação é totalmente obrigatório os conhecimentos de: Polimorfismo, Encapsulamento, Abstração, Herança e Agregação.

Passo 2 (Sobre o Singleton**)** A idéia do cerce da questão em se falando de Singleton, é de que tenhamos uma classe-objeto capaz de ser instanciada(criada em memória pronta para seu uso) uma única vez e com visibilidade e acessibilidade global dessa instância em um determinado escopo de projeto, conforme a Figura 1 abaixo.

Figura 1 - Classe Singleton

Imagine que se tenha uma Janela, essa seria a classe, chamada Janela.



Que por sua vez, teria duas funções(Métodos), abrir() e fechar(), conforme é mostrado na Listagem 1 abaixo.

Listagem 1

```
public class Janela {
  //Artributos
     public Janela( )
     {
     }

     public void abrir( )
     {
     }

     public void fechar( )
     {
     }
  //getters/setters
}
```

A questão é que para acessar uma classe e seus métodos/atributos tem que instanciá-la, conforme é mostrado na Listagem 2 abaixo:

Listagem 2

```
public class TesteJanela {
     public static void main(String args[])
     {
          Janela janela = new Janela();
          janela.abrir();
          janela.fechar();
     }
}
```

Mas há um problema, toda vez que tivermos que abrir ou fechar essa janela, vamos ter que instanciar. O que consome linhas de código, tempo, e até mesmo os recurso de seu projeto/sistema. Nesses casos onde temos que usar várias vezes um código semelhante, para tarefas semelhantes, em todo (global) do nosso projeto/sistema.

Para resolver este problema, pode-se utilizar o Design Patterns – Singleton, conforme será mostrado nos exemplos a seguir.

Passo 3 (Design Patterns Singleton, Construtores e variáveis públicos) Este é o jeito mais fácil de se construir um sigleton: usando uma variável de classe (static) pública e instanciando-a logo de início, conforme Listagem 3 abaixo.

Listagem 3

```
public class Janela {
    public static Janela janela = new Janela( );
    public Janela( )
    {
    }
    public static void abrir( )
    {
    }
    public static void fechar( )
    {
    }
  //getters/setters
}
```

Mas este código tem um problema. Um objeto pode, em vez de usar a variável pública janela, instanciar sua própria versão da classe Janela e aí o pattern perdeu o sentido.

Passo 4 (Design Patterns Singleton, Construtor private e variável pública) Para resolver o problema listado acima, iremos utilizar o construtor private e a variável public, conforme Listagem 4 abaixo.

Listagem 4

```
public class Janela {
    public static Janela janela = new Janela( );
    private Janela( )
    {
    }
    public static void abrir( )
    {
    }
    public static void fechar( )
    {
    }
  //getters/setters
}
```

Com o construtor private outros objetos não podem instanciar seu próprio singleton. Entretanto, não é uma boa prática dar acesso direto às variáveis da classe, para resolver o problema.

Passo 5 (Design Patterns Singleton, Construtor e variável private) Uma outra solução seria, além do construtor private, agora a variável estática janela também é private, e também será criado um método getInstance que retorna a instância do singleton, conforme Listagem 5 abaixo.

Listagem 5

```
public class Janela {
    private static Janela janela = new Janela( );
    private Janela( )
    {
    }
    public static Janela getInstance( )
    {
        return janela
```

```
    }
  //getters/setters e outros métodos
}
```

Passo 6 (Design Patterns Singleton, outra solução com Construtor e variável private) Similar ao código anterior, mas não instancia o objeto logo no início, mas espera que alguém peça uma instância. Aí ele instancia. Depois, para quem pedir, retorna o objeto que já está instanciado, conforme Listagem 6 abaixo.

Listagem 6

```
public class Janela {
    private static Janela janela = null;
    private Janela( )
    {
    }
    public static Janela getInstance( )
    {
        if(janela==null)
        {
            janela = new Janela();
        }
        return janela
    }
  //getters/setters e outros métodos
}
```

Passo 7 (Design Patterns Singleton, instanciando a classe Janela na classe TesteJanela) Para realizar a instancia será utilizado o método getInstance( ), conforme Listagem 7 abaixo.

Listagem 7

```
public class TesteJanela {
    public static void main(String args[])
    {
        Janela janela = Janela.getInstance();
        janela.abrir();
        janela.fechar();
    }
}
```

Passo 8 (Design Patterns Singleton, utilizando o synchronized) Se nos depararmos com um ambiente multithread, corre-se o risco de duas threads chamarem o método getInstance ao mesmo tempo, no momento em que a variável de instância seja null, e cada thread receber uma instância diferente. Para evitar este problema é só fazer o método getInstance ser synchronized, conforme Listagem 8 abaixo.

Listagem 8

```
public class Janela {
    private static Janela janela = null;
    private Janela( )
    {
    }
    public static synchronized Janela getInstance( )
    {
        if(janela==null)
        {
            janela = new Janela();
        }
        return janela
```

```
    }
  //getters/setters e outros métodos
}
```

Atividade para avaliação

1. Escreva, compile e execute o programa abaixo. Em seguida, troque sua implementação para que a classe Incremental seja Singleton. Execute novamente e veja os resultados.

```
public class Incremental {
	private static int count = 0;
	private int numero;

	public Incremental() {
		numero = ++count;
	}

	public String toString() {
		return "Incremental " + numero;
	}
}

public class TesteIncremental {
	public static void main(String[] args) {
		for (int i = 0; i < 10; i++) {
			Incremental inc = new Incremental();
			System.out.println(inc);
		}
	}
}
```

2. O código a seguir tem uma classe chamada Deck representando um conjunto de 52 cartas. Queremos que, durante um jogo qualquer, o mesmo Deck esteja disponível para todos os jogadores da mesa. Portanto, o mesmo objeto deck deve estar disponível em todo o programa. Este e uma clássica situação para o uso do singleton. Faca:

1. Mude a classe Deck para que ela implemente o pattern singleton.
2. Mude o metodo main para que obtenha uma instancia de Desck em vez de cria-la.
3. O programa deve imprimir todas as cartas em ordem aleatória.

```
import java.util.*;

enum Suit {
	SPADES, HEARTS, CLUBS, DIAMONDS
}

public class Card {
	public Card(Suit s, int n) {
		suit = s;
		if ((n < 2) || (n > 14)) {
```

```
                throw new IllegalArgumentException();
            }
            number = n;
        }

        public void print() {
            switch (number) {
            case 11:
                System.out.print("Jack");
                break;
            case 12:
                System.out.print("Queen");
                break;
            case 13:
                System.out.print("King");
                break;
            case 14:
                System.out.print("Ace");
                break;
            default:
                System.out.print(number);
                break;
            }
            System.out.print(" of ");
            switch (suit) {
            case SPADES:
                System.out.println("spades.");
                break;
            case HEARTS:
                System.out.println("hearts.");
                break;
            case CLUBS:
                System.out.println("clubs.");
                break;
            case DIAMONDS:
                System.out.println("diamonds.");
                break;
            }
        }

        private Suit suit;
        private int number;
}

public class Deck {
        public Deck() {
            cards = new ArrayList<Card>();
            // build the deck
            Suit[] suits = { Suit.SPADES, Suit.HEARTS, Suit.CLUBS,
```

```
Suit.DIAMONDS };
            for (Suit suit : suits) {
                  for (int i = 2; i <= 14; i++) {
                        cards.add(new Card(suit, i));
                  }
            }
            // shuffle it!
            Collections.shuffle(cards, new Random());
      }

      public void print() {
            for (Card card : cards) {
                  card.print();
            }
      }

      private List<Card> cards;
}

public class SingletonExercise {
      public static void main(String args[]) {
            Deck deck = new Deck();
            deck.print();
      }
}
```