Compartilhar o seu link com: luciorocha @ professores.utfpr.edu.br
Mabily < Cópia de Aula 27 - POCO4A - Exercícios propostos >
Vitor Viana< Cópia de Aula 27 - POCO4A - Exercícios propostos >
Angélica < Cópia de Aula 27 - POCO4A - Exercícios propostos >
Séfora< Aula 27 - POO >
Isabella < Cópia de Aula 27 - POCO4A - Exercícios propostos >
Rafael Kendy < Cópia de Aula 27 - POCO4A - Exercícios propostos >
Gabriel Reis Macedo Alexandre Aparecido <
Cópia de Aula 27 - POCO4A - Exercícios propostos >
Raphael Uematsu/Deivid< Cópia de Aula 27 - POCO4A - Exercícios propostos >
João Vitor N. Yoshida < Cópia de Aula 27 - POCO4A - Exercícios propostos >
Thales Kohki Hasegawa, Fernando < Cópia de Aula 27 - POCO4A - Exercícios propostos >
Plinio < Cópia de Aula 27 - POCO4A - Exercícios propostos >
Matheus Hirata & Thiago Cristovão< Cópia de Aula 27 - POCO4A - Exercícios propostos >
Gabriel Takeshi < Cópia de Aula 27 - POCO4A - Exercícios propostos >
Vitor Hugo Leite A. de Oliveira < Aula 27 - Exercícios >
Felipe F B Lorusso < Aula 27 - POCO4A - Exercícios propostos >
Filipe Augusto < Cópia de Aula 27 - POCO4A - Exercícios propostos >
Daniel Martins de Carvalho < Cópia de Aula 27 - POCO4A - Exercícios propostos >
Lucas Viana e Victor Ramos < Cópia de Aula 27 - POCO4A - Exercícios propostos >
Guilherme Conceição Ramalho < Cópia de Aula 27 - POCO4A - Exercícios propostos >
Wesley Santos Figueiredo Zimmer< Cópia de Aula 27 - POCO4A - Exercícios propostos >
Roberto Furlani Neto< Cópia de Aula 27 - POCO4A - Exercícios propostos >
Gustavo Nunes Cópia de Aula 27 - POCO4A - Exercícios propostos
João Pedro de Paula:< Cópia de Aula 27 - POCO4A - Exercícios propostos >.
Matheus Maeda Cópia de Aula 27 - POCO4A - Exercícios propostos
Henrique Cois < Cópia de Aula 27 - POCO4A - Exercícios propostos >

1) Padrão de Projeto: Singleton.

Motivação: manter uma única instância ativa de um objeto de determinada classe.

Passo 1: Criar classe com construtor private e variável de classe private.

```
public class Prateleira {

        private static Prateleira estoque;
```

```
        private Prateleira (){
                System.out.println("Singleton iniciado.");
        }

        public static Prateleira iniciar(){
                if(estoque==null)as2
                        estoque = new Prateleira();
                return estoque;
        }

}
```

Passo 2: Instanciar o singleton em outra classe.

```
public class Principal {

        public void iniciar(){

            Prateleira adm = Prateleira.iniciar();
            Prateleira adm2 = Prateleira.iniciar();

        }

        public static void main(String[] args) {
                new Principal().iniciar();

        }
}
```

1.1) (Online) Exercício: Acesse o link da atividade (Aula25prog5): https://codeboard.io/projects/359841

1.2) (Online) Exercício: Acesse o link da atividade (Aula25prog6): https://codeboard.io/projects/359856

2) Padrão de Projeto Visitor:

Motivação: filtrar de maneiras diferentes objetos de classes diferentes mantendo as mesmas assinaturas.

Exemplo1:

```
package aula26;

public class Aula26 {

   public interface IFiltro {
      public void filtrar(IDados dados); //SetorCompras/SetorAgua
   }

   public class FiltroNome implements IFiltro {
      public void filtrar(IDados dados) {
         dados.getNome();
      }
   }
   public class FiltroLocal implements IFiltro {
      public void filtrar(IDados dados) {
         dados.getLocal();
      }
   }

   public interface IDados {
      public void getNome();
      public void getLocal();
   }

 public class SetorAtendimento implements IDados {
      public void getNome(){
         System.out.println("NOME: SETOR ATENDIMENTO");
      }
      public void getLocal(){
         System.out.println("LOCAL: SETOR ATENDIMENTO");
      }
```

```
  }

public class SetorCompras implements IDados {
  public void getNome(){
     System.out.println("NOME: SETOR COMPRAS");
  }
  public void getLocal(){
     System.out.println("LOCAL: SETOR COMPRAS");
  }
}

public class SetorAgua implements IDados {
   public void getNome(){
     System.out.println("NOME: SETOR AGUA");
  }
  public void getLocal(){
     System.out.println("LOCAL: SETOR AGUA");
  }
}


public Aula26(){

   IFiltro fn = new FiltroNome();

   IDados objeto = new SetorCompras();

   fn.filtrar(objeto); //nome do SETOR COMPRAS

   objeto = new SetorAgua();

   fn.filtrar( objeto ); //nome do SETOR AGUA
   //..........

   fn = new FiltroLocal();
   fn.filtrar(objeto); //local do SETOR AGUA

   objeto = new SetorCompras();
   fn.filtrar(objeto); //local do SETOR COMPRAS
   //.......
   objeto = new SetorAtendimento();
   fn.filtrar(objeto); //Local do SETOR ATENDIMENTO
   fn = new FiltroNome();
   fn.filtrar(objeto); //Nome do SETOR ATENDIMENTO

}
```

```
    public static void main(String[] args) {
        new Aula26();

    }

}
```

2.1) (Online) Exercício: Acesse o link da atividade (Aula27prog1):

https://codeboard.io/projects/

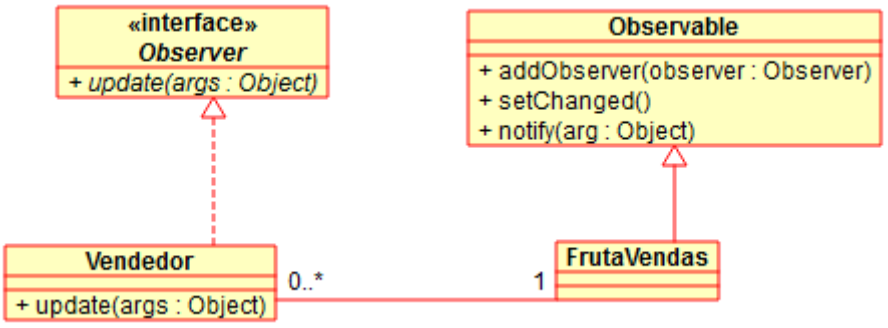
```

```

360577

2.2) (Online) Exercício: Acesse o link da atividade (Aula27prog2): https://codeboard.io/projects/360580

3) Padrão de Projeto: Observer

Motivação: receber notificação de alteração de instâncias.



Passo 1: A classe observadora deve implementar a interface 'java.util.Observer':

```
public class Vendedor implements java.util.Observer {

                private String nome;
                public Vendedor(String nome){
```

```
                this.nome=nome;
        }

        @Override
        public void update(Observable remoto, Object atributoRemoto) {
                System.out.println(this +": Recebi atualizacao de [" + remoto + "]" +
                                " que mudou o seguinte atributo: " +
atributoRemoto);
        }
        public String toString(){
                return this.nome;
        }

    }
```

Passo 2: A classe observável deve ser subclasse de 'java.util.Observable':

```
public class FrutaVendas extends java.util.Observable {
        private String nome;
        private int estoque;
        public FrutaVendas(String nome, int estoque){
                this.nome=nome;
                this.estoque=estoque;
        }
        public void venda(int qtde){
                if(estoque>0){
                        estoque -= qtde;
                        this.setChanged(); //Mudou o estado do objeto
                        this.notifyObservers("estoque: " + this.estoque); //Notifique
todos os Observers
                }
        }//fim venda
        public String toString(){
                return this.getClass().getSimpleName();
        }
    }//fim classe
```

Passo 3: A instância da classe observável deve adicionar um observador:

```
public void iniciar(){
        FrutaVendas fruta = new FrutaVendas("Abacate",100); //nome,estoque

        Vendedor joao = new Vendedor("joao");
        Vendedor maria = new Vendedor("MARIA");
```

```
                fruta.addObserver(joao);
                fruta.addObserver(maria);

                fruta.venda(10);
        }
```

//Exemplo de sala

```
package aula26;

import java.util.Observable;

public class Aula26 {

   public class Funcionario implements java.util.Observer {

      public void update(Observable remoto, Object atributoRemoto) {
         System.out.println(this +
            ": Recebi atualizacao de [" + remoto + "]" +
                  " que mudou o seguinte atributo: " + atributoRemoto);
      }
      public String toString(){
         return this.getClass().getSimpleName();
      }
   }

   public class Pagamento extends java.util.Observable {

      private float pagamento;
      public void pagar(){
         this.pagamento = 1000.0f;
         this.setChanged(); //Mudou o estado do objeto
         this.notifyObservers("pagamento: " + this.pagamento); //Notifique todos os
Observers
      }
      public float getPagamento(){
         return this.pagamento;
      }
   }

   public Aula26(){

      Funcionario f1 = new Funcionario();
```

```
        Funcionario f2 = new Funcionario();

        Pagamento pagamento = new Pagamento();
        pagamento.addObserver(f1);
        pagamento.addObserver(f2);

        f1.update(
              pagamento,
              pagamento.getPagamento());
        f2.update(pagamento,
              pagamento.getPagamento());

        pagamento.pagar();
    }



    public static void main(String[] args) {
        new Aula26();

    }

}
```

3.1) (Online) Exercício: Acesse o link da atividade (Aula27prog3): https://codeboard.io/projects/360597

3.2) (Online) Exercício: Acesse o link da atividade (Aula27prog4): https://codeboard.io/projects/360600

4) Padrão de Projeto: Template

Motivação: Invocar na superclasse métodos da subclasse.

```
public class Principalb {

	public interface ISuperClasse {
		//Visivel no package + subclasses
		public abstract String imprimir();
	}

	public abstract class SuperClasse implements ISuperClasse {

		public SuperClasse(){
			imprimirSubClasse();
		}

		public void imprimirSubClasse(){
			System.out.println("Estou na superclasse: " +
					this.imprimir()); //Invoca informacoes da subclasse
		}
	}

	public class SubClasse extends SuperClasse {

		public SubClasse(){
		}

		public final String imprimir(){
			return this.getClass().getSimpleName()+"";
		}
	}

	public Principalb(){
		SubClasse s = new SubClasse();
	}

	public static void main (String [] args){

		new Principalb();

	}//fim main

}//fim classe
```