Aula 15

Princípios SOLID

Rafael Geraldeli Rossi

Introdução

- **Definições**:
  - [Aniche, 2015]: conjunto de princípios de boas práticas de programação orientada a objetos que visa diminuir o acoplamento entre classes e separar responsabilidades como forma de melhorar o código da aplicação desenvolvida
  - [Wikipedia, 2020]: é um acrônimo para cinco postulados de design, destinados a facilitar a compreensão, o desenvolvimento e a manutenção de *software*
  - [Souza, 2017]: o padrão S.O.L.I.D. são cinco princípios que nos ajudam a desenvolver aplicações mais robustas e de fácil manutenção.

Introdução

- Os princípios SOLID visam aumentar a facilidade de manutenção e extensão do *software*

- São ao todo 5 princípios, cada um correspondente à uma letra do SOLID
  - **S***ingle responsibility principle*
  - **O***pen closed principle*
  - **L***iskov substitution principle*
  - **I***nterface segregation principle*
  - **D***ependency inversion principle*

Introdução

- **Não usar o princípios SOLID pode causar**:
  - Repetição de código
  - Simples alterações por causar várias alterações em diferentes pontos
  - Código sem estrutura coesa ou padronizada
  - Código com mais chances de erros mediante a alterações

Introdução

- **Usar o princípios SOLID pode causar**:
  - Fácil no entendimento e manutenção do código
  - Facilidade na aplicação de testes automatizados
  - Maior reaproveitamento do código
  - Fácil adaptação a mudanças

Single Responsibility Principle

Definição

Uma classe deve ter um único, e somente um, motivo para que possa ser modificada

Single Responsibility Principle

- Vamos tomar como base um classe cuja responsabilidade é calcular o desconto no salário do funcionário com base no cargo que ocupa e no valor do salário

```
8   public class CalculadoraDeSalario {
9
10
11      public double calcula(Funcionario funcionario) {
12          if(Cargo.DESENVOLVEDOR.equals(funcionario.getCargo())) {
13              return dezOuVintePorcento(funcionario);
14          }
15
16          if(Cargo.DESENVOLVEDOR.equals(funcionario.getCargo()) ||
17             Cargo.TESTER.equals(funcionario.getCargo())) {
18              return quinzeOuVinteCincoPorcento(funcionario);
19          }
20
21          throw new RuntimeException("funcionario invalido");
22      }
23
24      private double dezOuVintePorcento(Funcionario funcionario) {
25          if(funcionario.getSalarioBase() > 3000.0) {
26              return funcionario.getSalarioBase() * 0.8;
27          }
28          else {
29              return funcionario.getSalarioBase() * 0.9;
30          }
31      }
32
33      private double quinzeOuVinteCincoPorcento(Funcionario funcionario) {
34          if(funcionario.getSalarioBase() > 2000.0) {
35              return funcionario.getSalarioBase() * 0.75;
36          }
37          else {
38              return funcionario.getSalarioBase() * 0.85;
39          }
40      }
41
42  }
```

Single Responsibility Principle

- **Classe coesa**:
  - Se uma classe é coesa, ela tem uma única responsabilidade, ou seja, ela cuida de apenas uma única funcionalidade do sistema
  - Se essa classe é coesa, ela tem que parar de crescer um dia, uma vez que as responsabilidades de uma entidade um dia param de aparecer
  - Classes não coesas geram códigos complicados $\rightarrow$ difícil dar manutenção, mais suscetível a bugs, diminuição do reuso

Single Responsibility Principle

- Vamos deixar o código anterior coeso:
  - Colocar cada uma das funções das regras (`DezOuVintePorCento` e `QuinzeOuVintePorCento`) e colocá-las cada uma em uma classe $\rightarrow$ por sí só, cada uma dessas classes será coesa
  - Cada uma dessas classes só vai mudar se a regra mudar (um bom e único motivo)
  - Além disso, para padronizarmos o uso das regras de cálculo, vamos definir uma interface para a regra de cálculo e fazer a interface implementá-las

Single Responsibility Principle

```
public interface RegraDeCalculo {

    public double calcula(Funcionario funcionario);

}
```

Single Responsibility Principle

```
public class DezOuVintePorcento implements RegraDeCalculo{

    @Override
    public double calcula(Funcionario funcionario) {
        if(funcionario.getSalarioBase() > 3000.0) {
            return funcionario.getSalarioBase() * 0.8;
        }
        else {
            return funcionario.getSalarioBase() * 0.9;
        }
    }

}
public class QuinzeOuVintePorcento implements RegraDeCalculo{

    @Override
    public double calcula(Funcionario funcionario) {

        if (funcionario.getSalarioBase() > 2000.0) {
            return funcionario.getSalarioBase() * 0.75;
        } else {
            return funcionario.getSalarioBase() * 0.85;
        }

    }

}
```

Single Responsibility Principle

Com isso, a classe de cálculo de regras ficaria da seguinte forma:

```
public class CalculadoraDeSalario {


    public double calcula(Funcionario funcionario) {
        if(Cargo.DESENVOLVEDOR.equals(funcionario.getCargo())) {
            return new DezOuVintePorcento().calcula(funcionario);
        }

        if(Cargo.DESENVOLVEDOR.equals(funcionario.getCargo()) ||
           Cargo.TESTER.equals(funcionario.getCargo())) {
            return new QuinzeOuVintePorcento().calcula(funcionario);
        }

        throw new RuntimeException("funcionario invalido");
    }

}
```

Single Responsibility Principle

Apesar de mais coesa.. se surgirem novas regras de calculo, além de gerar novas classes, a classe para calcular o pagamento ainda terá que ser modificada

Single Responsibility Principle

```
public enum Cargo {
    DESENVOLVEDOR(new DezOuVintePorCento()),
    DBA(new QuinzeOuVintePorCento()),
    TESTER(new QuinzeOuVintePorCento());

    private RegraDeCalculo regra;

    Cargo(RegraDeCalculo regra){
            this.regra = regra;
    }

    public RegraDeCalculo getRegra() {
            return regra;
    }
}
```

Single Responsibility Principle

```java
public class CalculadoraDeSalario {


    public double calcula(Funcionario funcionario) {
        return funcionario.calcularSalario();
    }

}
```

```java
public class Funcionario {
        private int id;
        private String nome;
        private Cargo cargo;
        private Calendar dataDeAdmissao;
        private double salarioBase;

        public int getId() {...3 linhas }
        public void setId(int id) {...3 linhas }

        public String getNome() {...3 linhas }

        public void setNome(String nome) {...3 linhas }

        public Cargo getCargo() {...3 linhas }

        public void setCargo(Cargo cargo) {...3 linhas }

        public Calendar getDataDeAdmissao() {...3 linhas }

        public void setDataDeAdmissao(Calendar dataDeAdmissao) {...3 linhas }

        public double getSalarioBase() {...3 linhas }
        public void setSalarioBase(double salarioBase) {
            this.salarioBase = salarioBase;
        }

        public double calcularSalario() {
                return cargo.getRegra().calcula(this);
        }

}
```

Open Closed Principle

Definição

Objetos ou entidades devem estar abertos para extensão, mas fechados para modificação

Open Closed Principle

- **Motivação do OCP**: alterar uma classe já existente para adicionar um novo comportamento, corremos um sério risco de introduzir bugs em algo que já estava funcionando.

- **Como fazer**: separar o comportamento "extensível" por trás de uma interface

Open Closed Principle

```java
public class CalculadoraDePrecos {

    public double calcula(Compra produto) {
        TabelaDePrecoPadrao tabela = new TabelaDePrecoPadrao();
        Frete correios = new Frete();

        double desconto = tabela.descontoPara(produto.getValor());
        double frete = correios.para(produto.getCidade());

        return produto.getValor() * (1-desconto) + frete;
    }

}
```

```java
public class TabelaDePrecoPadrao {

    public double descontoPara(double valor) {
        if(valor>5000) return 0.03;
        if(valor>1000) return 0.05;
        return 0;

    }
}
```

```java
public class Frete {

    public double para(String cidade) {
        if("SAO PAULO".equals(cidade.toUpperCase())) {
            return 15;
        }
        return 30;
    }

}
```

```java
public class Compra {

    private double valor;
    private String cidade;

    public double getValor() {
        return valor;
    }
    public String getCidade() {
        return cidade;
    }
}
```

Open Closed Principle

- No exemplo anterior, tem-se várias classes, porém coesas (pequenas e com responsabilidades bem definidas)

- Entretanto, supondo que o *software* irá crescer para considerar outras tabelas de preço (diferenciada por cliente), ou ainda, outras tabelas para calcular o frete

Open Closed Principle

- **Solução 1**: colocar um `if` na classe `CalculadoradePrecos` para definir se será utilizada a TabelaDePrecoPadroa ou `TabelaDePrecoDiferenciada`, ou ainda se será utilizada o frete dos correios ou de uma outra empresa

- **Consequência 1**:
  - Código começa a crescer com muitos `ifs` (problemas já apresentados no SRP)
  - Começa a depender de muitas classes $\rightarrow$ alterações nas classes das tabelas de preços ou fretes podem ter consequência na classe `CalculadoraDePrecos`

Open Closed Principle

- **Solução 2**:
  - "Abrir" para extensão: conseguir estender as funcionalidades de maneira fácil
  - "Fechada para alteração: se tem que mexer na classe adicionar as novas funcionalidades
  - Abstrair os comportamentos padrões e definir interfaces

- **Consequência 2**: podemos estender as funcionalidades de uma classe criando novas classes que implementam as interfaces definidas no item anterior

Open Closed Principle

Interfaces para a continuidade do exemplo do OCP

```
public interface TabelaDePreco {

    double descontoPara(double valor);

}


public interface ServicoDeEntrega {

    double para(String cidade);

}
```

Open Closed Principle

```
public class CalculadoraDePrecos {

    private TabelaDePreco tabela;
    private ServicoDeEntrega entrega;

    public CalculadoraDePrecos(TabelaDePreco tabela, ServicoDeEntrega entrega) {
            this.tabela = tabela;
            this.entrega = entrega;
    }

    public double calcula(Compra produto) {
        double desconto = tabela.descontoPara(produto.getValor());
        double frete = entrega.para(produto.getCidade());

        return produto.getValor() * (1-desconto) + frete;
    }

}
```

Liskov Substitution Principle

Definição

As classes derivadas devem ser substituíveis por suas classes bases

Liskov Substitution Principle

- O *Liskov Substitution Principle* (LSP) ou Princípio de Substituição de Liskov está diretamente ligado ao *Open Closed Principle*

- A definição apresentada anteriormente pode ser representada pela seguinte premissa: *"Se para cada objeto o_1 do tipo S há um objeto o_2 do tipo T de forma que, para todos os programas P definidos em termos de T, o comportamento de P é inalterado quando o_1 é substituído por o_2, então S é um subtipo de T"*

Liskov Substitution Principle

- Definição alternativa da Wikipedia: *"Se S é um subtipo de T, então os objetos do tipo T, em um programa, podem ser substituídos pelos objetos de tipo S sem que seja necessário alterar as propriedades deste programa"*

- Além disso, não só tem que ser possível usar tanto uma classe pai quando uma classe filha, quanto a classe filha não deve "afrouxar" as condições da classe pai

Liskov Substitution Principle

- Por "afrouxar" entende-se:
  - Valores retornados não devem estar além do escopo da classe pai
  - Classes filhas não deve ter menos atributos ou menos funcionalidades que as classes pai

Liskov Substitution Principle

- Considere um exemplo parecido com o nosso Projeto Branco, em que há uma `ContaComum`, a qual tem rendimentos, e uma `ContaEstudante`, a qual tem todas as características de uma conta comum, porém, não rende

```java
public class ContaComum {

    protected double saldo;

    public ContaComum() {
        this.saldo = 0;
    }

    public void deposita(double valor) {
        this.saldo += valor;
    }

    public double getSaldo() {
        return saldo;
    }

    public void rende()      {
        this.saldo*= 1.1;
    }

}
```

```java
public class ContaEstudante extends ContaComum{

    public void rende() throws ContaNaoRendeException{
        throw new ContaNaoRendeException();
    }

}
```

Liskov Substitution Principle

- Ao executar o programa anterior, se o banco só tiver objetos do tipo `ContaComum`, o programa será executado sem problemas
- Entretanto, na presença de um objetos do tipo `ContaEstudante`, será lançada uma exceção
- Em uma situação em que o sistema já existia somente com contas comuns, e foi adicionada a ContaEstudante como extensão do sistema, os laços que processavam as contas comuns irão para de funcionar, isso por conta da criação de uma classe filho

Liskov Substitution Principle

```
public class ContaEstudante extends ContaComum{

    public void rende() throws ContaNaoRendeException{
        throw new ContaNaoRendeException();
    }

}
```

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

     for (ContaComum conta : contasDoBanco()) {

         conta.rende();
         System.out.println("Novo Saldo:");
         System.out.println(conta.getSaldo());

     }

 }
```

```
public class ContaComum {

    protected double saldo;

    public ContaComum() {
        this.saldo = 0;
    }

    public void deposita(double valor) {
        this.saldo += valor;
    }

    public double getSaldo() {
        return saldo;
    }

    public void rende()      {
        this.saldo*= 1.1;
    }

}
```

Liskov Substitution Principle

- De acordo com o princípio LSP, nunca pode-se criar uma pré-condição que seja mais restrita do que da classe pai, ou seja, nunca uma classe filha pode ter menos funcionalidades que a classe pai (**as classes filhas devem respeitar os contratos das classes pai)**

- Muitos autores dizem que muitas vezes é interessante fazer a composição ao invés da herança, isto é:
  - Criar uma classe com as funcionalidades em comum
  - Utilizar objetos da referida classe como campos em outras classes

Liskov Substitution Principle

```java
public class ManipuladorDeSaldo {

    private double saldo;

    public void deposita(double valor) {
        this.saldo += valor;
    }

    public void saca(double valor) {
        if (valor <= this.saldo) {
                this.saldo -= valor;
        } else {
                throw new IllegalArgumentException();
        }
    }

    public double getSaldo() {
        return saldo;
    }

    public void rende(double taxa) {
        this.saldo *= taxa;
    }

}
```

```java
public class ContaComum {

    private ManipuladorDeSaldo manipulador;

    public ContaComum() {
            this.manipulador = new ManipuladorDeSaldo();
    }

    public void saca(double valor) {
            manipulador.saca(valor);
    }

    public void deposita(double valor) {
            manipulador.deposita(valor);
    }

    public void rende() {
            manipulador.rende(1.1);
    }

    public double getSaldo() {
            return manipulador.getSaldo();
    }

}
```

```java
public class ContaEstudante{

    private ManipuladorDeSaldo manipulador
    private int milhas;

    public ContaEstudante() {
            this.manipulador = new Manipul
    }

    public void deposita(double valor) {
        manipulador.deposita(valor);
        this.milhas += (int) valor;
    }

    public int getMilhas() {
        return milhas;
    }
}
```

Liskov Substitution Principle

- **Precauções ao utilizar a herança**
  - Considere uma classe `Gerente`, filha de `Funcionario`, cujo bônus é 30%, enquanto que de um Funcionário é 20%
  - Suponha que no método bonus() é chamado o método `super.bonus()`, para obter os 20% e depois o `super.bonus()/2` para obter os demais 10%
  - Se por um acaso alterar o cálculo do método super.bonus() na classe pai, isso se refletirá na classe filho, gerando um comportamento errôneo na classe `Gerente`

Liskov Substitution Principle

- **Favoreça a composição**
  - Muito desenvolvedores sugerem o uso da composição ao invés da herança
  - Não utilizar a herança apenas para ganhar métodos (ex: classe `Math`)
  - É fácil trocar a implementação passada e obter um novo comportamento.

Liskov Substitution Principle

- **Problemas com a composição**
  - Considere a classe `ManipuladorDeSaldo`
  - As classes que fazer uso por meio de composição da classe `ManipuladorDeSaldo`, devem implementar uma grande quantidade de métodos que apenas repassam a chamada de métodos
  - O polimorfismo, principalmente e linguagens, tipadas torna-se mais complicado

Interface Segregation Principle

Definição

Muitas interfaces específicas são melhores do que uma interface única geral

Interface Segregation Principle

- Clientes não devem ser forçados a depender de interfaces que eles não usam
- Crie interfaces granulares a específicas para os seus clientes
- Interfaces que tem muitos comportamentos podem forçar implementações não necessárias e permitir comportamentos indevidos do sistema

Interface Segregation Principle

```java
public interface Imposto {

    public abstract NotaFiscal geraNota();
    public abstract double imposto(double valorBruto);

}
```

```java
public class ISS implements Imposto{


    public double imposto(double valorBruto){
        return valorBruto * 0.1;
    }

    public NotaFiscal geraNota(){
        return new NotaFiscal("XXXX","YYYY",12312);
    }

}
```

```java
public class IXMX implements Imposto{

    @Override
    public double imposto(double valorBruto) {
        return 0.2 * valorBruto;
    }

    @Override
    public NotaFiscal geraNota() {
        throw new UnsupportedOperationException("Not supported yet.");
    }

}
```

Interface Segregation Principle

```java
public interface CalculadorImposto {

    public abstract double imposto(double valorBruto);

}
```

```java
public interface GeradorNota {

    public abstract NotaFiscal geraNota();

}
```

```java
public class ISS implements CalculadorImposto, GeradorNota{


    public double imposto(double valorBruto){
        return valorBruto * 0.1;
    }

    public NotaFiscal geraNota(){
        return new NotaFiscal("XXXX","YYYY",12312);
    }

}
```

```java
public class IXMX implements CalculadorImposto{

    @Override
    public double imposto(double valorBruto) {
        return 0.2 * valorBruto;
    }

}
```

Dependency Inversion Principle

Definição

Dependa de abstrações e não de implementações.

Dependency Inversion Principle

- Se no LSP o cuidado era com as heranças, no DIP o cuidado é com o acoplamento
- Ao acoplar com muitas classes específicas, uma alteração ou mal funcionamento nas classes acopladas podem causa mal funcionamento ou grandes alterações de códigos em classes que fazer uso delas
- A classe, quando possui muitas dependências, torna-se muito frágil, fácil de quebrar

Dependency Inversion Principle

```
public class GeradorDeNotaFiscal {

    private EnviadorDeEmail email;
    private NotaFiscalDAO dao;

    public GeradorDeNotaFiscal(EnviadorDeEmail email, NotaFiscalDAO dao) {
        this.email = email;
        this.dao = dao;
    }

    /*...*/

}
```

Dependency Inversion Principle

- **Problemas**:
  - O grande problema do acoplamento é que uma mudança em qualquer uma das classes pode impactar em mudanças na classe
  - Uma mudança de métodos nas classes também irá causar mal funcionamento
  - O reúso dessas classes também fica cada vez mais difícil, uma vez que se quisermos reutilizar uma determinada classe em outro lugar, precisaremos levar junto todas suas dependências
  - Lembre-se também que as dependências de uma classe podem ter suas próprias dependências, gerando uma grande árvore de classes que devem ser levadas junto

Dependency Inversion Principle

- Por outro lado, é **IMPOSSÍVEL** nos livrarmos do acomplamento
- Já que não é possível eliminar os acoplamentos, é necessário diferenciá-los
- A ideia é fugir de acomplamentos "perigosos"
- **SOLUÇÃO**: acoplar-se com coisas (classes e interfaces) estáveis

Dependency Inversion Principle

- Por exemplo, depender de uma interface List ou da classe String é uma dependencia estável
  - Muitas outras classes dependes dessas classes
  - Exemplos de dependência da classe `List`: `AbstractList`, `AbstractSequentialList`, `ArrayList`, `AttributeList`, `CopyOnWriteArrayList`, `LinkedList`, `RoleList`, `RoleUnresolvedList`, `Stack`, `Vector`

- Classes estáveis tendem a mudar pouco, e, portanto, a dependência delas é menos problemática

Dependency Inversion Principle

- Interfaces são um bom começo ou caminho pra a estabilidade
- Interface não têm código que pode forçar uma mudança, e geralmente tem implementações dela, e isso faz com que o desenvolvedor pense duas vezes antes de mudar o contrato (como mostrado anteriormente).
- Considerando o exemplo anterior, poderíamos fazer uma interface `AcaoAposGerarNotaFiscal`
  - `EnviadorDeEmail` e `NotaFiscalDAO` dependeriam dessa interface
  - Poderiamos acrescer outras ações, como EnviadorDeSMS, etc., de maneira mais fácil no sistema

Dependency Inversion Principle

```java
public interface AcaoAposGerarNota {

    public abstract void executar(NotaFiscal nf);

}
```

```java
public class GeradorDeNotaFiscal {

    private List<AcaoAposGerarNota> acoes;


    public GeradorDeNotaFiscal(List<AcaoAposGerarNota> acoes) {
        this.acoes = acoes;
    }

    public NotaFiscal gera(Fatura fatura){
        NotaFiscal nf = null; // Null só para fins de exmeplo

        /* ... */

        for(AcaoAposGerarNota acao : acoes){
            acao.executar(nf);
        }

        /* ... */

        return null;
    }

    /*...*/

}
```

Dependency Inversion Principle

- Com a solução anterior, o `GeradorDeNotaFiscal` depende apenas de `List` e de `AcaoAposGerarNotaFiscal`
- Porém, podemos agora ter inúmeras funcinoalidaes dentro do `GeradorDeNotaFiscal`
- Se tivermos várias classes dependendo da interface `AcaoAposGerarNotaFiscal`, dificilmente essa classe será alterada

Dependency Inversion Principle

- **Voltando ao DIP**
  - Sempre que uma classe for depender de outra, ela deve depender sempre de outro módulo mais estável do que ela mesma
  - Se A depende de B, a ideia é que B seja mais estável que A, ou ainda, se possível, tente sempre depender da abstração
  - As classes devem sempre andar em direção à estabilidade, depender de módulos mais estáveis que ela
    - `AcaoAposGerarNota` é uma abstração, estável, e não conhece detalhes de implementação
    - `GeradorDeNotaFiscal` é uma implementação, o que faz dela um módulo mais instável, mas que só depende de abstrações

Extra: Enum

- Enum no Java
  - São tipos de campos que consistem em um conjunto fixo de constantes (static final)
  - Podemos considerar como uma lista de valores pré-definidos
  - Esses valores são tipos primitivos (que serão transformados em wrappers) ou por referência

Declaração

A declaração de um enum é semelhante a declaração de uma classe

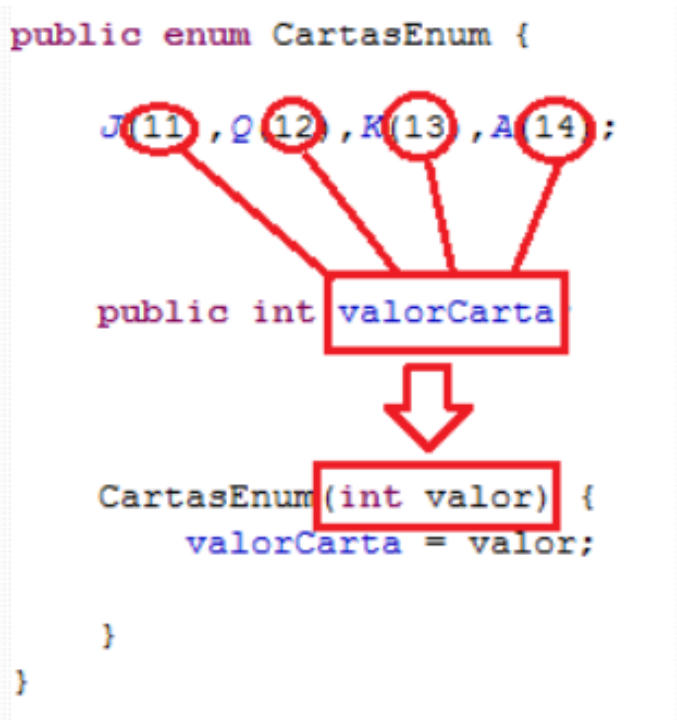
```
public enum OpcoesMenu {

    //Corpo do enum

}
```

Campos e Valores

- Para definir os campos e valores, deve-se especificar o identificador do campo e em seguida o valor entre parênteses
- A sequência de campos e valores deve ser separada por “,”
- Por convensão, campos estáticos e finais devem ser declarados utilizando letras maiúsculas

Campos e Valores

- É também necessário implementar um construtor com o intuito de inicializar os valores
- É como se cada campo do tipo enum, chamasse esse construtor



Campos e Valores

```
public enum OpcoesMenu {

    CADASTRAR(1), APAGAR(2), PESQUISAR(3), SAIR(4);

    public int valorOpcao;

    OpcoesMenu(int valorOpcao){
        this.valorOpcao = valorOpcao;
    }

    public int getValorOpcao() {
        return valorOpcao;
    }

}
```

Percorrendo campos e valores de um `enum`

- O enum herda métodos e campos os quais podemos utilizá-los para:
  - Percorrer os elementos
  - Fazer comparações
  - ...

Percorrendo campos e valores de um enum

```
20      public static void main(String[] args) {
21
22          System.out.println("Imprimindo os campos ============== ");
23
24          System.out.println(OpcoesMenu.CADASTRAR);
25          System.out.println(OpcoesMenu.PESQUISAR);
26          System.out.println(OpcoesMenu.PESQUISAR);
27          System.out.println(OpcoesMenu.SAIR);
28
29          System.out.println("Listando os campos ================");
30          for(OpcoesMenu op : OpcoesMenu.values()){
31              System.out.println(op);
32          }
33
34          System.out.println("Listando os campos com os respectivos valores ================");
35          for(OpcoesMenu op : OpcoesMenu.values()){
36              System.out.println(op.name() + " - " + op.getValorOpcao());
37          }
38
39      }
40
```

Saída × Resultados da Pesquisa × String.java ×

Console do Depurador × Teste (run) ×

```
run:
Imprimindo os campos ==============
CADASTRAR
PESQUISAR
PESQUISAR
SAIR
Listando os campos ================
CADASTRAR
APAGAR
PESQUISAR
SAIR
Listando os campos com os respectivos valores ================
CADASTRAR - 1
APAGAR - 2
PESQUISAR - 3
SAIR - 4
```

Comparação de valores com enum

```java
public class Principal {

    public static void main(String[] args) {

        Scanner teclado = new Scanner(System.in);

        System.out.println("MENU =============");
        for(OpcoesMenu opcao : OpcoesMenu.values()){
            System.out.println(opcao.getValorOpcao() + " - " + opcao);
        }

        System.out.print("Digite uma opção: ");
        int op = teclado.nextInt();

        if(op == OpcoesMenu.CADASTRAR.getValorOpcao()){
            System.out.println("Acessou cadastrar");
        }else if(op == OpcoesMenu.PESQUISAR.getValorOpcao()){
            System.out.println("Acessou pesquisar");
        }else if(op == OpcoesMenu.REMOVER.getValorOpcao()){
            System.out.println("Acessou remover");
        }else if(op == OpcoesMenu.SAIR.getValorOpcao()){
            System.out.println("Acessou sair");
        }else{
            System.out.println("Opção inválida");
        }


    }
```

```
run:
MENU =============
1 - CADASTRAR
2 - REMOVER
3 - PESQUISAR
4 - AREA_RESTRITA
5 - SAIR
Digite uma opção: 2
Acessou remover
```

Exercício - Projeto Banco

- Criar `Enums` para o Menu Principal e Submenus e modificar o código para considerar os respectivos `Enums`
- Criar dependências de abstrações quando possível (DIP)
- Aplicar o SRP e OCP na validação de campos
- Aplicar o SRP e OCP na leitura e gravação das contas

Material Complementar

- Tipos Enum no Java

  `https://www.devmedia.com.br/tipos-enum-no-java/25729`

- Enums no Java

  `https://www.devmedia.com.br/enums-no-java/38764`

- Princípios S.O.L.I.D: o que são e porque projetos devem utilizá-los

  `https:`
  `//medium.com/@mari_azevedo/princ%C3%ADpios-s-o-l-i-d-o-que-s%C3%`
  `A3o-e-porque-projetos-devem-utiliz%C3%A1-los-bf496b82b299`

Material Complementar

- SOLID — Princípios da Programação Orientada a Objetos

  `https://medium.com/thiago-aragao/solid-princ%C3%ADpios-da-programa%C3%A7%C3%A3o-orientada-a-objetos-ba7e31d8fb25`

- SOLID (O básico para você programar melhor) // Dicionário do Programador

  `https://www.youtube.com/watch?v=mkx0CdWiPRA`

- Clean Code // Dicionário do Programador

  `https://www.youtube.com/watch?v=ln6t3uyTveQ`

Imagem do Dia

Programação Orientada a Objetos
http://lives.ufms.br/moodle/

Rafael Geraldeli Rossi
rafael.g.rossi@ufms.br

Slides baseados no *Curso de SOLID com Java: Orientação a Objetos com Java* da ALURA (https://cursos.alura.com.br/course/orientacao-a-objetos-avancada-e-principios-solid) e em [Aniche, 2015]

Referências Bibliográficas I


Aniche, M. (2015).
Orientação a Objetos e SOLID para Ninjas.
Série Caelum. Casa do Código, 1ª edition.


Souza, M. (2017).
S.o.l.i.d.


Wikipedia (2020).
Solid.