

BACK-END

Prof. Bruno Kurzawe



Programação Orientada a Objetos

Classes Abstract

Uma classe abstrata é uma classe que não pode ser instanciada diretamente. Ou seja, você não pode criar um objeto a partir de uma classe abstrata diretamente. Em vez disso, classes abstratas são destinadas a serem herdadas por outras classes. A principal razão para usar uma classe abstrata é definir uma base comum para suas subclasses, estabelecendo atributos, métodos (funções) ou comportamentos que todas as subclasses herdam.

Olhando para estrutura atual do nosso projeto que classes poderiam ser abstratas?

Estrutura atual do nosso projeto

```
└─ org.example
   └─ model
      ├── Cliente
      ├── EntityId
      ├── FormaPagamento
      ├── Fornecedor
      ├── ItemVenda
      ├── ItemVendavel
      ├── Pessoa
      ├── Produto
      ├── Servico
      ├── Status
      ├── Venda
      └─ Application
```

Que classes aqui
nunca serão
instanciadas
diretamente?

```
public static void main(String[] args) {  
  
    Pessoa bruno = new Pessoa();  
    bruno.setNome("Bruno Kurzawe");  
    bruno.setEmail("bruno.kurzawe@betha.com.br");  
    bruno.setEndereco("Rua Almirante B");  
    bruno.setTelefone("48 99908-9410");  
  
}
```

No nosso negócio faz sentido instanciar uma Pessoa?



```
public abstract class Pessoa extends EntityId {  
    private String nome;  
    private String telefone;  
    private String endereco;  
    private String email;
```

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

```
    Pessoa bruno = new Pessoa();
```

```
    bruno.setNome("Bruno");
```

```
    bruno.setEmail("bruno@unisa.br");
```

```
    bruno.setEndereco("Rua da Liberdade, 123 - Centro - São Paulo - SP");
```

```
    bruno.setTelefone("48 99908-9410");
```

'Pessoa' is abstract; cannot be instantiated

org.example.model

```
public abstract class Pessoa
```

```
extends EntityId
```

Exemplo0001

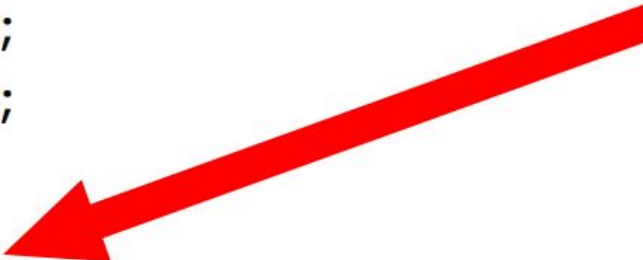
```
}
```

Métodos Abstratos

Classes abstratas podem conter métodos abstratos. Estes são métodos que são declarados na classe abstrata, mas não têm uma implementação definida. Qualquer subclasse que herda da classe abstrata deve fornecer uma implementação para esses métodos abstratos.

Vamos criar um método abstract para nossa classe pessoa

```
public abstract class Pessoa extends EntityId {  
    private String nome;  
    private String telefone;  
    private String endereco;  
    private String email;  
  
    public abstract String getDocumentoPrincipal();  
  
    public String getNome() {  
        return nome;  
    }  
}
```



```
public class Fornecedor extends Pessoa {
```

```
private String c
```

```
private String i
```

```
public String getCnpj() {  
    return cnpj;  
}
```

Class 'Fornecedor' must either be declared abstract or implement abstract method 'getDocumentoPrincipal()' in 'Pessoa'

org.example.model

```
public class Fornecedor  
extends Pessoa
```

Exemplo0001

 `public class Fornecedor extends Pessoa {` **ALT + ENTER**

`private String cnpj;`

`private String incricao;`

`public String getCnpj() {
 return cnpj;`

`}`

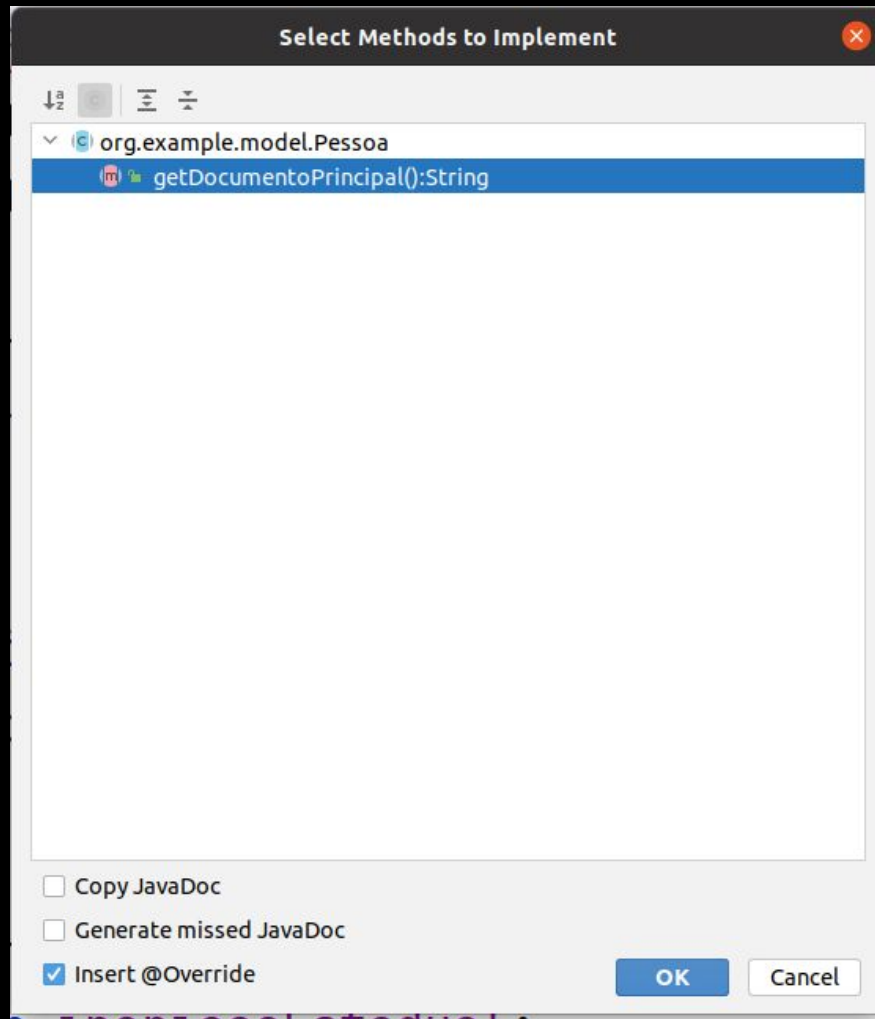
- Implement methods
 - Make 'Fornecedor' abstract >
 - Create Subclass >
 - Create Test >
 - Seal class >
- Press Ctrl+Q to toggle preview

22 @Override

23 public String
 getDocumentoPrincipal() {

24 return null;

25 }



```
@Override  
public String getDocumentoPrincipal() {  
    return null;  
}
```

Qual seria a implementação para **getDocumentoPrincipal()** em fornecedor?

```
@Override  
public String getDocumentoPrincipal() {  
    return this.getCnpj();  
}
```



Outra classe que quebrou também foi a classe Cliente

```
public class Cliente extends Pessoa {
```

```
    private String cpf;
```

```
    private String rg;
```



```
public String getCpf() { return cpf; }
```

```
@Override  
public String getDocumentoPrincipal() {  
    return this.getCpf();  
}
```



Agora vamos entender o que esse **@Override** em cima do método significa.


Sobrescrita

"Sobrescrita" (ou "overriding", em inglês) é um conceito da programação orientada a objetos (OOP) que se refere à capacidade de uma subclasse modificar ou aprimorar uma implementação de um método que é herdado de uma superclasse.

Vamos abrir a classe **ItemVendavel**



```
public class ItemVendavel extends EntityId {  
    private String descricao;  
    private Double valorUnitario;  
  
    public String getDescricao() {  
        return descricao;  
    }  
}
```

```
public class ItemVendavel extends EntityId {  
    private String descricao;  
    private Double valorUnitario;  
    private Boolean estocavel;  
  
    public String getDescricao() {  
        return descricao;  
    }  
}
```



Vamos criar os getters e setters de **estocavel**

```
public class ItemVendavel extends EntityId {  
    private String descricao;  
    private Double valorUnitario;  
    private Boolean estocavel;  
  
    public Boolean getEstocavel() {  
        return estocavel;  
    }  
  
    public void setEstocavel(Boolean estocavel) {  
        this.estocavel = estocavel;  
    }  
}
```

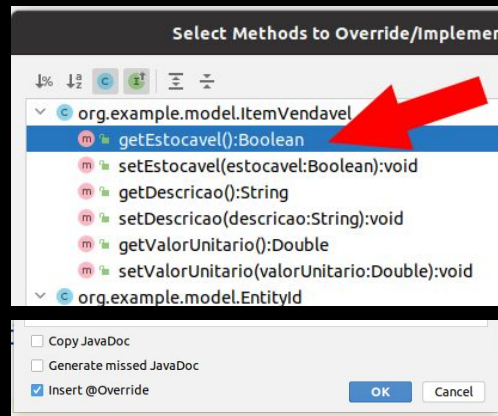
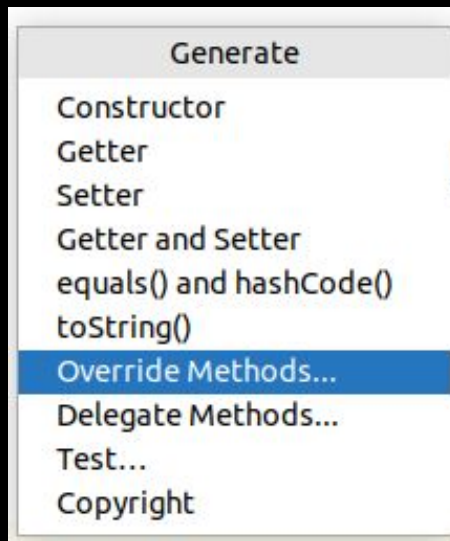


Servico em algum momento vai ser estocável?

Na classe Servico


```
public class Servico extends ItemVendavel {  
    private Double quantidadeHoras;  
  
    public Servico(String descricao, Double quantidadeHoras) {  
        super.setDescricao(descricao);  
        this.quantidadeHoras = quantidadeHoras;  
        super.setValorUnitario(valor);  
    }  
  
    public Double getQuantidadeHoras() { return quantidadeHoras; }  
  
    public void setQuantidadeHoras(Double quantidadeHoras) {  
        this.quantidadeHoras = quantidadeHoras;  
    }  
  
    @Override  
    public String toString() {  
        return "Servico [descricao=" + descricao + ", quantidadeHoras=" + quantidadeHoras + ", valorUnitario=" + valorUnitario + "]";  
    }  
}
```

1. Clique com o botão direito
2. Clique em generate (alt + insert)




Na classe Servico

```
@Override  
public Boolean getEstocavel() {  
    return super.getEstocavel();  
}
```



Na classe Servico

```
@Override  
public Boolean getEstocavel() {  
    return false;  
}
```



Na classe Main

```
public class Application {  
    public static void main(String[] args) {  
        Produto produto = new Produto("Computador", "I5 8gb");  
        System.out.println("Prod: "+produto.getEstocavel());  
  
        Servico servico = new Servico("Instalação Office", 2.0, 100.00);  
        System.out.println("Serv: "+servico.getEstocavel());  
    }  
}
```

```
Application x
.jar=43489:/snap/intellij-idea-community
/home/bruno.kurzawe/Documentos/senac/Exe
.Application
Prod: null
Serv: false

Process finished with exit code 0
```

Interfaces

"Interfaces" são um conceito chave na programação orientada a objetos, especialmente em linguagens como Java, C# e TypeScript. Uma interface define um contrato que uma classe pode optar por implementar. Esse contrato estipula um conjunto de métodos (sem implementação) que a classe deve fornecer.

Para trabalharmos com Interfaces vamos criar um cenário.

Vamos criar a classe **ItemCompra**

```
public class ItemCompra {  
    private Produto produto;  
    private Double valorUnitario;  
    private Double quantidade;  
    private Double desconto;  
  
    //Criem os getters e setters  
}
```

Vamos criar a classe **Compra**

```
public class Compra extends EntityId {  
  
    private LocalDate dataCompra;  
    private Fornecedor fornecedor;  
    private String observacao;  
    private List<ItemCompra> itens = new ArrayList<>();  
  
    //criem os getters e setters
```

Vamos criar a classe **ItemLocacao**

```
public class ItemLocacao {  
    private Produto produto;  
    private Double valorUnitario;  
    private Double quantidade;  
    private Double desconto;  
  
    //Criem os getters e setters  
}
```

Vamos criar a classe **Locacao**

```
public class Locacao extends EntityId {  
    private LocalDate dataLocacao;  
    private LocalDate dataDevolucao;  
    private Cliente cliente;  
    private String endereco;  
    private String observacao;  
    private List<ItemLocacao> itens = new ArrayList<>();  
}
```

Beleza, criamos agora um cenário, agora imaginem o seguinte, vendemos, compramos e alugamos. Dessas operações retiraremos o nosso balanço, entradas e saídas financeiras.

Vamos criar a classe **Balanco**

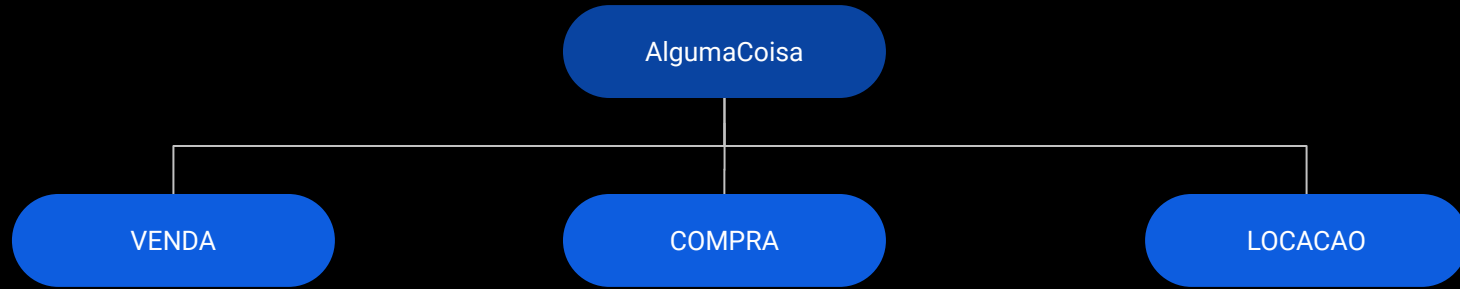
```
public class Balanco extends EntityId {  
    private LocalDate dataBalanco;  
    private String responsavel;  
  
    //Itens de entrada/saida financeira  
  
}
```

Como poderíamos incluir esses itens?

Poderia ser assim?

```
public class Balanco {  
    private LocalDate dataBalanco;  
    private String responsavel;  
    private List<Compra> compras;  
    private List<Venda> vendas;  
    private List<Locacao> locacoes;  
  
}
```


Poderia usar herança?



Para resolver isso podemos usar **interface**.

Vamos criar a ENUM **TipoOperacao**

```
public enum TipoOperacao {  
    DEBITO,  
    CREDITO  
}
```

Vamos criar a ENUM **OperacaoFinaceira**

```
public interface OperacaoFinanceira {  
  
    public LocalDate getDataOperacao();  
  
    public Double getValorTotalOperacao();  
  
    public TipoOperacao getTipoOperacao();  
  
}
```

Essa interface será nosso contrato, vai definir que qualquer um que a use tenha que seguir sua implementação base.

Aqui faremos um novo **refactoring!**

Vamos utilizar nosso contrato

```
public class Compra implements OperacaoFinanceira{
```

```
    private LocalDate data;  
    private Fornecedor fornecedor;  
    private String observacao;  
    private List<ItemCompra> itens = new ArrayList<>();
```

Class 'Compra' must either be declared abstract or implement abstract method 'getDataOperacao()' in 'OperacaoFinanceira'


org.example.model

public interface OperacaoFinanceira

Exemplo0001



Vamos utilizar nosso contrato



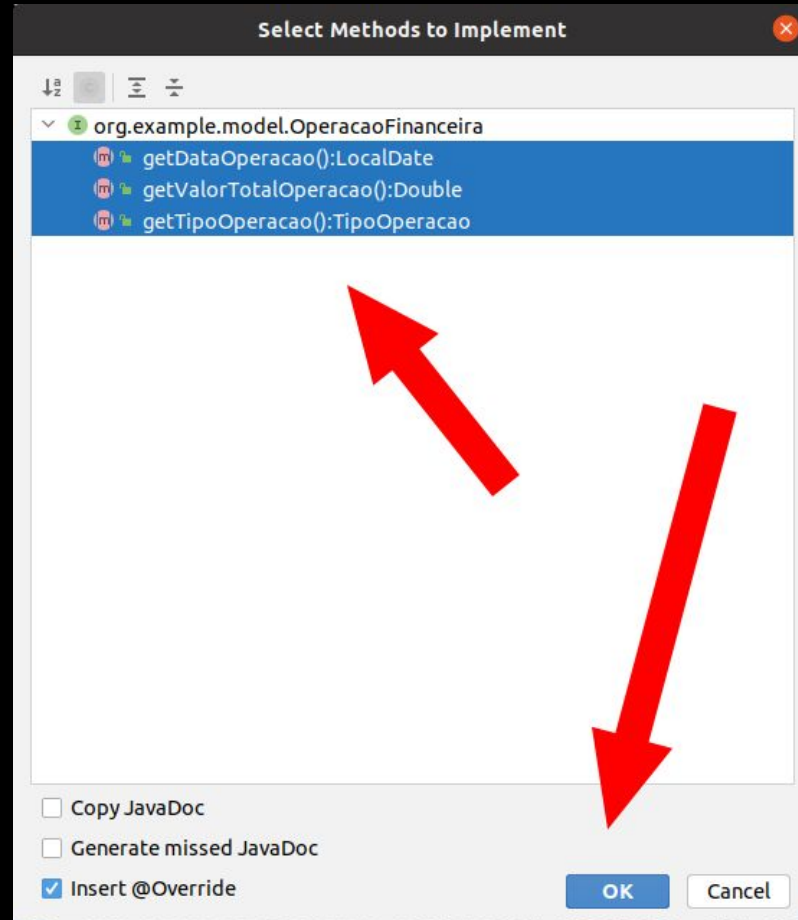
```
public class Compra implements OperacaoFina
```

```
private LocalDate dataCom
```

```
private Fornecedor fornec
```

- Implement methods
- Make 'Compra' abstract >
- Create Subclass >
- Create Test >
- Seal class >

Press Ctrl+Q to toggle preview



Nossos métodos são implementados

```
@Override
public LocalDate getDataOperacao() {
    return null;
}

@Override
public Double getValorTotalOperacao() {
    return null;
}

@Override
public TipoOperacao getTipoOperacao() {
    return null;
}
```

Nossos métodos são implementados

```
@Override
public LocalDate getDataOperacao() {
    return this.getDataCompra();
}


@Override
public Double getValorTotalOperacao() {
    return this.getItems().stream()
        .mapToDouble(ItemCompra::getValorUnitario)
        .sum();
}

@Override
public TipoOperacao getTipoOperacao() {
    return TipoOperacao.DEBITO;
}
```

Agora nossa classe **Compra** segue o nosso contrato de
OperacaoFinanceira.

Vamos aplicar o mesmo contrato a **Venda** e **Locacao**.

```
public class Venda extends EntityId implements OperacaoFinanceira {  
    private LocalDate dataVenda;  
    private Cliente cliente;  
    private FormaPagamento formaPagamento;  
    private String observacao;  
    private List<ItemVenda> itens = new ArrayList<>();  
}
```



@Override

```
public LocalDate getDataOperacao() {  
    return this.getDataVenda();  
}
```

@Override

```
public Double getValorTotalOperacao() {  
    return this.getItems().stream()  
        .mapToDouble(ItemVenda::getValorUnitario)  
        .sum();  
}
```

@Override

```
public TipoOperacao getTipoOperacao() {  
    return TipoOperacao.CREDITO;  
}
```

```
public class Locacao extends EntityId implements OperacaoFinanceira{  
    private LocalDate dataLocacao;  
    private LocalDate dataDevolucao;  
    private Cliente cliente;  
    private String endereco;  
    private String observacao;  
    private List<ItemLocacao> itens = new ArrayList<>();  
}
```



```
@Override
public LocalDate getDataOperacao() {
    return this.getDataLocacao();
}

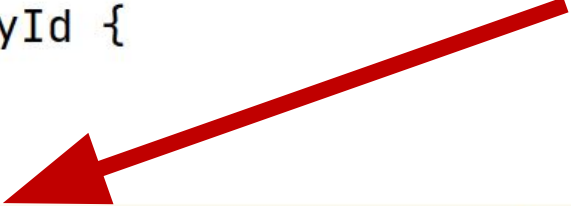
@Override
public Double getValorTotalOperacao() {
    return this.getItems().stream()
        .mapToDouble(ItemLocacao::getValorUnitario)
        .sum();
}

@Override
public TipoOperacao getTipoOperacao() {
    return TipoOperacao.CREDITO;
}
```

Agora vamos a grande sacada das interfaces =D

Conseguimos aplicar polimorfismo em interfaces

```
public class Balanco extends EntityId {  
    private LocalDate dataBalanco;  
    private String responsavel;  
    private List<OperacaoFinanceira> operacoes = new ArrayList<>();  
}
```



Vamos criar alguns métodos getters e setter, métodos de Add e Del das operações.

```
public void addOperacoes(OperacaoFinanceira operacao) {  
    this.operacoes.add(operacao);  
}
```

```
public void delOperacoes(OperacaoFinanceira operacao) {  
    this.operacoes.add(operacao);  
}
```

Vamos criar um método para imprimirmos nosso balanço

```
public void imprimirBalanco() {  
    System.out.println("-----");  
    System.out.println("Balanco numero: " + this.getId());  
    System.out.println("Data: " + this.getDataBalanco());  
    System.out.println("Responsavel: " + this.getResponsavel());  
    System.out.println("-----");  
    System.out.println("Itens: ");  
    for (OperacaoFinanceira op : this.getOperacoes()) {  
        System.out.println("----- ");  
        System.out.println("Data operação: " + op.getDataOperacao());  
        System.out.println("Tipo operação: " + op.getTipoOperacao());  
        System.out.println("Valor operação: " + op.getValorTotalOperacao());  
        System.out.println("----- ");  
    }  
    System.out.println("-----");  
}
```

Vamos para nosso método MAIN

Criem

- Um fornecedor
- Um cliente
- Um produto
- Um serviço
- Uma compras
- Duas vendas
- Uma Locacao

e agora vamos criar um balanço (Main aula 04)

Agora vamos criar nosso balanço

```
Balanco balanco = new Balanco();  
balanco.setId(352578L);  
balanco.setDataBalanco(LocalDate.now());  
balanco.setResponsavel("Maria Luiza");  
balanco.addOperacoes(venda);  
balanco.addOperacoes(venda2);  
balanco.addOperacoes(compra);  
balanco.addOperacoes(locacao);  
  
balanco.imprimirBalanco();
```

Balanço numero: 352578

Data: 2023-08-20

Responsavel: Maria Luiza

Itens:

Data operação: 2023-08-20

Tipo operação: CREDITO

Valor operação: 1500.0

Data operação: 2023-08-20

Tipo operação: CREDITO

Valor operação: 1650.0

Data operação: 2023-08-20

Tipo operação: DEBITO

Valor operação: 1000.0

Data operação: 2023-08-20

Tipo operação: CREDITO

Valor operação: 150.0

Aqui faremos um novo **refactoring!**

```
public void imprimirBalanco() {  
    System.out.println("-----");  
    System.out.println("Balanco numero: " + this.getId());  
    System.out.println("Data: " + this.getDataBalanco());  
    System.out.println("Responsavel: " + this.getResponsavel());  
    System.out.println("-----");  
    System.out.println("Itens: ");  
    for (OperacaoFinanceira op : this.getOperacoes()) {  
        System.out.println("Data operação: " + op.getDataOperacao() +  
            " Tipo operação: " + op.getTipoOperacao() +  
            " Valor operação: " + op.getValorTotalOperacao());  
    }  
    System.out.println("-----");  
}
```

Balanco numero: 352578

Data: 2023-08-20

Responsavel: Maria Luiza

Itens:

Data operação: 2023-08-20 Tipo operação: CREDITO Valor operação: 1500.0

Data operação: 2023-08-20 Tipo operação: CREDITO Valor operação: 1650.0

Data operação: 2023-08-20 Tipo operação: DEBITO Valor operação: 1000.0

Data operação: 2023-08-20 Tipo operação: CREDITO Valor operação: 150.0

Tem algo errado nesses valores?

Balanço numero: 352578

Data: 2023-08-20

Responsavel: Maria Luiza

Itens:

Data operação:	2023-08-20	Tipo operação:	CREDITO	Valor operação:	1500.0
Data operação:	2023-08-20	Tipo operação:	CREDITO	Valor operação:	1650.0
Data operação:	2023-08-20	Tipo operação:	DEBITO	Valor operação:	1000.0
Data operação:	2023-08-20	Tipo operação:	CREDITO	Valor operação:	150.0

Aqui faremos um novo **refactoring!**

Vamos alterar essa lógica

```
@Override
public Double getValorTotalOperacao() {
    return this.getItems().stream()
        .mapToDouble(ItemCompra::getValorUnitario)
        .sum();
}
```

Em ItemCompra vamos criar o seguinte método

```
public Double getValorCalculado() {  
    double valorTotal = this.getValorUnitario() * this.getQuantidade();  
    double descontoCalculado = valorTotal * (this.getDesconto() / 100);  
    return valorTotal - descontoCalculado;  
}
```

Agora vamos usá lo na compra

```
@Override
public Double getValorTotalOperacao() {
    return this.getItems().stream()
        .mapToDouble(ItemCompra::getValorCalculado)
        .sum();
}
```

Agora nossa compra está calculando certo

Balanco numero: 352578

Data: 2023-08-20

Responsavel: Maria Luiza

Itens:

Data operação: 2023-08-20 Tipo operação: CREDITO Valor operação: 1500.0

Data operação: 2023-08-20 Tipo operação: CREDITO Valor operação: 1650.0

Data operação: 2023-08-20 Tipo operação: DEBITO Valor operação: 9000.0

Data operação: 2023-08-20 Tipo operação: CREDITO Valor operação: 150.0



Vamos aplicar essa mesma lógica em Venda e Locação

Balanço numero: 352578

Data: 2023-08-20

Responsavel: Maria Luiza

Itens:

Data operação: 2023-08-20 Tipo operação: CREDITO Valor operação: 1350.0

Data operação: 2023-08-20 Tipo operação: CREDITO Valor operação: 1485.0

Data operação: 2023-08-20 Tipo operação: DEBITO Valor operação: 9000.0

Data operação: 2023-08-20 Tipo operação: CREDITO Valor operação: 1500.0

Agora vamos trabalhar nos totalizadores


```
public Double getValorTotalDebitos() {  
    return this.getOperacoes().stream().filter(op -> op.getTipoOperacao().equals(TipoOperacao.DEBITO))  
        .mapToDouble(OperacaoFinanceira::getValorTotalOperacao)  
        .sum();  
}  
  
public Double getValorTotalCreditos() {  
    return this.getOperacoes().stream().filter(op -> op.getTipoOperacao().equals(TipoOperacao.CREDITO))  
        .mapToDouble(OperacaoFinanceira::getValorTotalOperacao)  
        .sum();  
}
```

Podemos trocar essas duas por essa?

```
public Double getValorTotal(TipoOperacao tipo) {  
    return this.getOperacoes().stream().filter(op -> op.getTipoOperacao().equals(tipo))  
        .mapToDouble(OperacaoFinanceira::getValorTotalOperacao)  
        .sum();  
}
```

```
public void imprimirBalanco() {  
    System.out.println("-----");  
    System.out.println("Balanco numero: " + this.getId());  
    System.out.println("Data: " + this.getDataBalanco());  
    System.out.println("Responsavel: " + this.getResponsavel());  
    System.out.println("-----");  
    System.out.println("Itens: ");  
    for (OperacaoFinanceira op : this.getOperacoes()) {  
        System.out.println("Data operação: " + op.getDataOperacao() +  
            " Tipo operação: " + op.getTipoOperacao() +  
            " Valor operação: " + op.getValorTotalOperacao());  
    }  
    System.out.println("-----");  
    System.out.println("Total Debitos: " + this.getValorTotal(TipoOperacao.DEBITO));  
    System.out.println("Total Creditos: " + this.getValorTotal(TipoOperacao.CREDITO));  
    System.out.println("Total: " + (this.getValorTotal(TipoOperacao.CREDITO) -  
        this.getValorTotal(TipoOperacao.DEBITO)));  
}
```

Balanço numero: 352578

Data: 2023-08-20

Responsavel: Maria Luiza

Itens:

Data operação: 2023-08-20 Tipo operação: CREDITO Valor operação: 1350.0

Data operação: 2023-08-20 Tipo operação: CREDITO Valor operação: 1485.0

Data operação: 2023-08-20 Tipo operação: DEBITO Valor operação: 9000.0

Data operação: 2023-08-20 Tipo operação: CREDITO Valor operação: 1500.0

Total Debitos: 9000.0

Total Creditos: 4335.0

Total: -4665.0

Aqui faremos um novo **refactoring!**

Na classe **Balanco**

```
public String getTipoOperacao(OperacaoFinanceira operacao) {  
    if (operacao instanceof Compra) {  
        return "Compra";  
    }  
    if (operacao instanceof Venda) {  
        return "Venda";  
    }  
    return "Locação";  
}
```

Na classe **Balanco**

```
public void imprimirBalanco() {  
    System.out.println("-----");  
    System.out.println("Balanco numero: " + this.getId());  
    System.out.println("Data: " + this.getDataBalanco());  
    System.out.println("Responsavel: " + this.getResponsavel());  
    System.out.println("-----");  
    System.out.println("Itens: ");  
    for (OperacaoFinanceira op : this.getOperacoes()) {  
        System.out.println("Data operação: " + op.getDataOperacao() +  
            " Tipo operação: " + op.getTipoOperacao() +  
            " Valor operação: " + op.getValorTotalOperacao() +  
            " - (" + getTipoOperacao(op)+"");  
    }  
    System.out.println("-----");  
    System.out.println("Total Debitos: " + this.getValorTotal(TipoOperacao.DEBITO));  
    System.out.println("Total Creditos: " + this.getValorTotal(TipoOperacao.CREDITO));  
    System.out.println("Total: " + (this.getValorTotal(TipoOperacao.CREDITO) -  
        this.getValorTotal(TipoOperacao.DEBITO)));  
}
```


Na classe **Balanco**

Balanco numero: 352578

Data: 2023-08-20

Responsavel: Maria Luiza

Itens:

Data operação:	2023-08-20	Tipo operação:	CREDITO	Valor operação:	1350.0	(Venda)
Data operação:	2023-08-20	Tipo operação:	CREDITO	Valor operação:	1485.0	(Venda)
Data operação:	2023-08-20	Tipo operação:	DEBITO	Valor operação:	9000.0	(Compra)
Data operação:	2023-08-20	Tipo operação:	CREDITO	Valor operação:	1500.0	(Locação)

Total Debitos: 9000.0

Total Creditos: 4335.0

Total: -4665.0

Exceções

Em Java, uma exceção é um evento que ocorre durante a execução de um programa e que interrompe o fluxo normal das instruções. Exceções são usadas para indicar que algo inesperado ou anormal aconteceu e precisa ser tratado. Em Java, exceções são representadas por classes, e o sistema de exceções é integrado ao núcleo da linguagem.

Vamos ao seguinte código na Main

```
//Declaração de cliente
```

```
Cliente cliente = new Cliente();  
cliente.setNome("Bruno Kurzawe");
```

```
cliente.getCpf().toUpperCase();
```

Application x

```
/home/bruno.kurzawe/.jdk/corretto-17.0.7/bin/java -javaagent:/snap/intellij-idea-community/451/lib/idea_rt  
.jar=35301:/snap/intellij-idea-community/451/bin -Dfile.encoding=UTF-8 -classpath /home/bruno  
.kurzawe/Documentos/senac/ExemploAula/Exemplo0001/target/classes org.example.Application
```

```
Exception in thread "main" java.lang.NullPointerException Create breakpoint : Cannot invoke "String.toUpperCase()" because the return value of "org  
.example.model.Cliente.getCpf()" is null  
at org.example.Application.main(Application.java:17)
```



//Declaração de cliente

```
Cliente cliente = new Cliente();
```

```
cliente.setNome("Bruno Kurzawe");
```

```
try {
```

```
    cliente.getCpf().toUpperCase();
```

```
} catch (Exception e) {
```

```
    System.out.println("CPF não informado!");
```

```
}
```

Application x

```
/home/bruno.kurzawe/.jdk/corretto-17.0.7/bin/java -javaagent:/snap/intellij-idea-community/451/lib/idea_rt  
.jar=33135:/snap/intellij-idea-community/451/bin -Dfile.encoding=UTF-8 -classpath /home/bruno  
.kurzawe/Documentos/senac/ExemploAula/Exemplo0001/target/classes org.example.Application
```

CPF não informado!



Balanco numero: 352578

Data: 2023-08-20

Responsavel: Maria Luiza

Vamos ao seguinte código na Main



```
Integer calculo = 10/0;  
System.out.println(calculo);
```

Application x

```
/home/bruno.kurzawe/.jdk/corretto-17.0.7/bin/java -javaagent:/snap/intellij-idea-community/451/bin/idea-rt.jar=34797:/snap/intellij-idea-community/451/bin/idea-rt.jar -Dfile.encoding=UTF-8 -classpath /home/bruno.kurzawe/Documents/senac/ExemploAula/Exemplo0001/target/classes org.example.Application
```

CPF não informado!

Exception in thread "main" java.lang.ArithmeticException: / by zero
at org.example.Application.main(Application.java:24)



```
Integer calculo;  
try {  
    calculo = 10 / 0;  
} catch (ArithmeticException aex) {  
    calculo = 0;  
}  
System.out.println(calculo);
```

Application x

```
/home/bruno.kurzawe/.jdk/corretto-17.0.7/bin/java -javaagent:/snap/intellij-idea-community/451/lib/idea_rt  
.jar=43747:/snap/intellij-idea-community/451/bin -Dfile.encoding=UTF-8 -classpath /home/bruno  
.kurzawe/Documentos/senac/ExemploAula/Exemplo0001/target/classes org.example.Application
```

CPF não informado!

0



Vamos ao seguinte código na Main

```
Integer[] array = {10, 14, 22, 33};  
System.out.println(array[4]);
```



Application x

```
.jar=35535:/snap/intellij-idea-community/451/bin -Dfile.encoding=UTF-8 -classpath /home/bruno  
.kurzawe/Documentos/senac/ExemploAula/Exemplo0001/target/classes org.example.Application  
CPF não informado!  
0  
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Create breakpoint : Index 4 out of bounds for length 4  
at org.example.Application.main(Application.java:34)
```

```
try {  
    Integer[] array = {10, 14, 22, 33};  
    System.out.println(array[4]);  
} catch (ArrayIndexOutOfBoundsException aix) {  
    System.out.println("Posição não existe!");  
}
```

Application x

```
/home/bruno.kurzawe/.jdk/corretto-17.0.7/bin/java -javaagent:/snap/intellij-idea-community/451/1  
.jar=42335:/snap/intellij-idea-community/451/bin -Dfile.encoding=UTF-8 -classpath /home/bruno  
.kurzawe/Documentos/senac/ExemploAula/Exemplo0001/target/classes org.example.Application
```

CPF não informado!

0

Posição não existe!



Exceções personalizadas

Exceções personalizadas, também conhecidas como exceções definidas pelo usuário, são exceções criadas por desenvolvedores para representar situações de erro específicas em suas aplicações. Ao criar suas próprias exceções, os desenvolvedores podem definir tipos de erro que são específicos para o domínio ou as regras de negócio de suas aplicações.

```
public void setPrecoVenda(Double precoVenda) {  
    super.setValorUnitario(precoVenda);  
    if (this.calculaMargemDeLucro() < 20.0) {  
        System.out.println("A Margem de lucro deve ser sempre maior ou igual a 20%");  
    }  
}
```


Criando uma exceção personalizada

```
public class MargemLucroException extends Exception {  
  
    public MargemLucroException() {  
        super("A Margem de lucro deve ser sempre maior ou igual a 20%!");  
    }  
}
```

Utilizando uma exceção personalizada

```
public void setPrecoVenda(Double precoVenda) throws MargemLucroException {  
    super.setValorUnitario(precoVenda);  
    if (this.calculaMargemDeLucro() < 20.0) {  
        throw new MargemLucroException();  
    }  
}
```

Utilizando no Main

//Declaração de Produto

```
Produto produto = new Produto("Computador", "I5 8gb");  
produto.setPrecoCompra(1200.00);  
produto.setPrecoVenda(1400.00);
```

Application x

CPF não informado!

0

Posição não existe!

Exception in thread "main" org.example.model.MargemLucroException Create breakpoint : A Margem de lucro deve ser sempre maior ou igual a 20%!
at org.example.model.Produto.setPrecoVenda(Produto.java:50)
at org.example.Application.main(Application.java:44)

Fim da aula 04...