

BACK-END

Prof. Bruno Kurzawe



Programação Orientada a Objetos

Herança

Herança é um conceito fundamental na programação orientada a objetos (POO) que permite que uma classe (chamada de classe derivada ou subclasse) herde os atributos e métodos de outra classe (chamada de classe base ou superclasse). Com a herança, a classe derivada pode reutilizar e estender as funcionalidades da classe base, evitando a duplicação de código e permitindo uma melhor organização e estruturação do código.

Herança Simples (Single Inheritance): Nesse tipo de herança, uma classe filha herda atributos e métodos de uma única classe pai. Isso significa que cada classe filha possui apenas uma classe pai.

Herança Múltipla (Multiple Inheritance): Nesse tipo de herança, uma classe filha pode herdar atributos e métodos de múltiplas classes pai. Isso pode resultar em uma hierarquia mais complexa, mas também pode levar a problemas de ambiguidade quando duas classes pai têm métodos ou atributos com o mesmo nome.

Em Java, não é possível realizar herança múltipla diretamente, ou seja, uma classe não pode herdar diretamente de várias classes pai. Isso ocorre porque a herança múltipla pode levar a problemas de ambiguidade e complexidade no sistema de tipos.

E agora, pensando no nosso projeto, onde poderíamos incluir herança?

E agora, pensando no nosso projeto, onde poderíamos incluir herança?

- Produto
- Cliente
- Venda
- Locação
- Fornecedor
- Compra
- Estoque

Alguma classe aqui
poderia herdar algo uma
da outra?

A princípio não, elas tem negócio específico

Então vamos desenvolver as seguintes classes do nosso projeto juntos...

```
public class Cliente {  
    private Long id;  
    private String nome;  
    private String cpf;  
    private String rg;  
    private String telefone;  
    private String endereco;  
    private String email;  
  
    // Criem os getters e setters  
}
```

```
public class Fornecedor {  
    private Long id;  
    private String nome;  
    private String cnpj;  
    private String incricaoEstadual;  
    private String telefone;  
    private String enderecao;  
    private String email;  
  
    // Criem os getters e setters  
}
```

Se olharmos essas classes de perto, temos algumas coisas em comum, certo?

```
public class Cliente {  
    private Long id;  
    private String nome;  
    private String cpf;  
    private String rg;  
    private String telefone;  
    private String endereco;  
    private String email;  
  
    // Criem os getters e setters  
}
```

```
public class Fornecedor {  
    private Long id;  
    private String nome;  
    private String cnpj;  
    private String inscricaoEstadual;  
    private String telefone;  
    private String endereco;  
    private String email;  
  
    // Criem os getters e setters  
}
```


Aqui faremos nosso primeiro **refactoring!**


Podemos resolver esse problema desses atributos duplicados usando herança!

Poderíamos generalizar essas informações em uma classe **Pessoa**


```
public class Pessoa {  
    private Long id;  
    private String nome;  
    private String telefone;  
    private String endereco;  
    private String email;  
  
    // Criem os getters e setters  
}
```

Aplicando herança em classes JAVA

```
public class Cliente extends Pessoa {  
    private String cpf;  
    private String rg;  
  
    // Criem os getters e setters  
}
```



```
public class Fornecedor extends Pessoa {  
    private String cnpj;  
    private String inscricaoEstadual;  
  
    // Criem os getters e setters  
}
```



Agora vamos criar um fornecedor no nosso Main

de Pai para Filho =D


```
public class Application {  
    public static void main(String[] args) {  
  
        Fornecedor dell = new Fornecedor();  
        dell.setNome("Dell LTDA");  
        dell.setCnpj("045802000188");  
        dell.setInscricaoEstadual("456411321");  
        dell.setEmail("comercial@dell.com.br");  
        dell.setTelefone("48 9999998585");  
        dell.setEndereco("Rua Joaquim XXII");  
    }  
}
```

de Pai para Filho =D

```
Cliente bruno = new Cliente();  
bruno.setNome("Bruno Kurzawe");  
bruno.setCpf("04685825233");  
bruno.setRg("5229814");  
bruno.setEmail("bruno.kurzawe@betha.com.br");  
bruno.setTelefone("48 999089410");  
bruno.setEndereco("Rua almirante b");
```

Aqui faremos um novo **refactoring!**

```
public class Pessoa {  
    private Long id;  
    private String nome;  
    private String telefone;  
    private String endereco;  
    private String email;
```




```
public class Produto {  
    private Integer id;  
    private String nome;  
    private String descricao;  
    private Double precoVenda;  
    private Double precoCompra;  
    private LocalDate dataValidade;  
    private LocalDate dataPrazo;  
    private Status status;
```




Vamos criar uma classe EntityId

```
public class EntityId {  
  
    private Long id;  
  
    public Long getId() {  
        return id;  
    }  
  
    public void setId(Long id) {  
        this.id = id;  
    }  
}
```

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

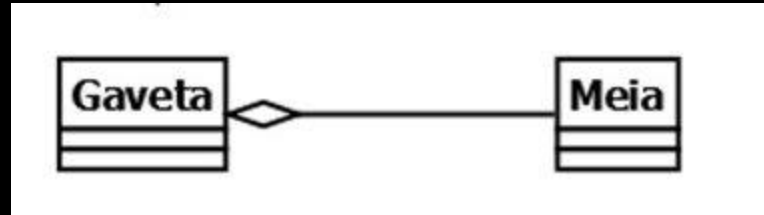


```
public class Produto extends EntityId {  
    private String nome;  
    private String descricao;  
    private Double precoVenda;  
    private Double precoCompra;
```



Agregação e Composição

Agregação:: É quando um objeto possui outros objetos, ele não depende desses objetos para existir.



Composição:: É quando um objeto é formado por outros objetos.
Ou seja, suas partes o compõem, sem elas o objeto não existe.



A principal diferença entre composição e agregação em Java está na força do vínculo entre as classes e no controle do ciclo de vida dos objetos relacionados. Composição é uma relação mais forte e controlada, enquanto agregação é uma relação mais fraca e independente.

Vamos pensar na nossa classe VENDA...

ID

DataVenda

Cliente

FormaPagamento

Observações

```
public class Venda extends EntityId {  
  
    private LocalDate dataVenda;  
    private FormaPagamento formaPagamento;  
    private String observacao;  
  
}
```

Agora como podemos adicionar nosso cliente a venda?


```
public class Venda extends EntityId {  
  
    private LocalDate dataVenda;  
    private Cliente cliente; ←  
    private FormaPagamento formaPagamento;  
    private String observacao;  
  
}
```

Vamos a implementação disso no main

```
public static void main(String[] args) {  
    Cliente bruno = new Cliente();  
    bruno.setNome("Bruno Kurzawe");  
    bruno.setCpf("04685825233");  
  
    Venda venda01 = new Venda();  
    venda01.setId(10L);  
    venda01.setDataVenda(LocalDate.of(2023,1,1));  
    venda01.setCliente(bruno);  
    venda01.setFormaPagamento(FormaPagamento.A_VISTA);  
    venda01.setObservacao("Observação 01");  
}
```

Agregação ou composição?

Polimorfismo


Polimorfismo é um conceito na programação orientada a objetos (POO) que permite que objetos de diferentes classes sejam tratados de forma uniforme, mesmo que essas classes possuem comportamentos diferentes. É uma das características fundamentais da POO que visa aprimorar a flexibilidade e extensibilidade do código.

Para entender esse conceito, vamos dar uma modificada na nossa classe Venda, se vamos vender... como relacionar os produtos?

```
public class ItemVenda {  
    private Produto produto;  
    private Double valorUnitario;  
    private Double quantidade;  
    private Double desconto;  
  
    //getters e setter  
}
```


Beleza, como eu posso adicionar esse **itens da venda** na **venda**?

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



Esse **List** vem de uma interface que permite a manipulação de coleções.

Podemos criar getters e setter para coleções? Sim! mas preferimos uma abordagem diferente para adicionar dados.

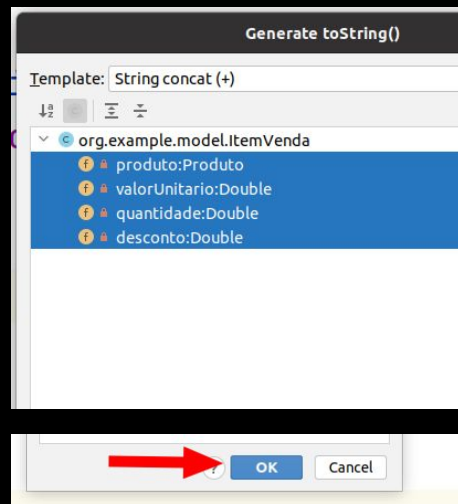
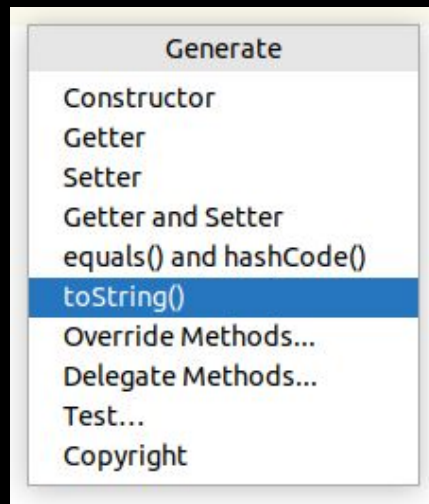
```
public void addItemVenda(ItemVenda item) {  
    this.itens.add(item);  
}  
  
public void delItemVenda(ItemVenda item) {  
    this.itens.remove(item);  
}  
  
public List<ItemVenda> getItens() {  
    return itens;  
}
```

Beleza, para nosso retorno ficar mais interessante vamos aprender sobre o `toString()`

Na classe ItemVenda

```
public class ItemVenda {  
    private Produto produto;  
    private Double valorUnitario;  
    private Double quantidade;  
    private Double desconto;  
  
    public ItemVenda(Produto produto, D  
        this.produto = produto;  
        this.valorUnitario = valorUnitario;  
        this.quantidade = quantidade;  
        this.desconto = desconto;  
}
```

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



@Override

```
public String toString() {  
    return "ItemVenda{" +  
        "produto=" + produto +  
        ", valorUnitario=" + valorUnitario +  
        ", quantidade=" + quantidade +  
        ", desconto=" + desconto +  
        '}';  
}
```


Agora vamos ao uso disso no Main...



```
Venda venda01 = new Venda();  
venda01.setId(10L);  
venda01.setDataVenda(LocalDate.of(2023, 1, 1));  
venda01.setCliente(bruno);  
venda01.setFormaPagamento(FormaPagamento.A_VISTA);  
venda01.setObservacao("Observação 01");  
  
Produto produto = new Produto("Computador", "I5 8gb");  
ItemVenda itemVenda = new ItemVenda(produto, 100.00, 1.0, 10.0);  
venda01.addItemVenda(itemVenda);  
  
Produto produto2 = new Produto("Computador", "I7 16gb");  
ItemVenda itemVenda2 = new ItemVenda(produto2, 50.00, 1.0, 10.0);  
venda01.addItemVenda(itemVenda2);  
  
System.out.println(venda01.getItems());
```


Run: Application

.Application

```
[ItemVenda{produto=org.example.model.Produto@30f39991, valorUnitario=100.0, quantidade=1.0, desconto=10.0}, ItemVenda{produto=org.example.model.Produto@452b3a41, valorUnitario=50.0, quantidade=1.0, desconto=10.0}]
```

Process finished with exit code 0

Podemos fazer o `toString()` para a classe `Produto`? Podemos, por padrão vamos criar `toString` para todas as nossas classes!



```
Run: Application x
[ItemVenda{produto=Produto{nome='Computador', descricao='I5 8gb'}, valorUnitario=100.0,
quantidade=1.0, desconto=10.0}, ItemVenda{produto=Produto{nome='Computador', descricao='I7
16gb'}, valorUnitario=50.0, quantidade=1.0, desconto=10.0}]
```

Mas ainda não vimos o Polimorfismo de fato... criamos um cenário possível de aplicá-lo.


Digamos que agora nossa Loja não vai apenas vender e alugar equipamentos, ela vai prestar serviços.

```
public class Servico extends EntityId {  
    private String descricao;  
    private Double quantidadeHoras;  
    private Double valor;  
  
    //Construtor  
    //Getters e Setter  
}
```


Agora temos que vender produtos e serviços, como colocar os dois na classe **ItemVenda**.

Poderia ser assim?

```
public class ItemVenda {  
    private Servico servico;  
    private Produto produto;  
    private Double valorUnitario;  
    private Double quantidade;  
    private Double desconto;
```



Poderia, mas não é usual...

Para resolver isso podemos usar **polimorfismo**

Vamos criar a classe ItemVendavel

```
public class ItemVendavel extends EntityId {  
    private String descricao;  
    private Double valorUnitario;  
  
    //Getters e Setters  
}
```

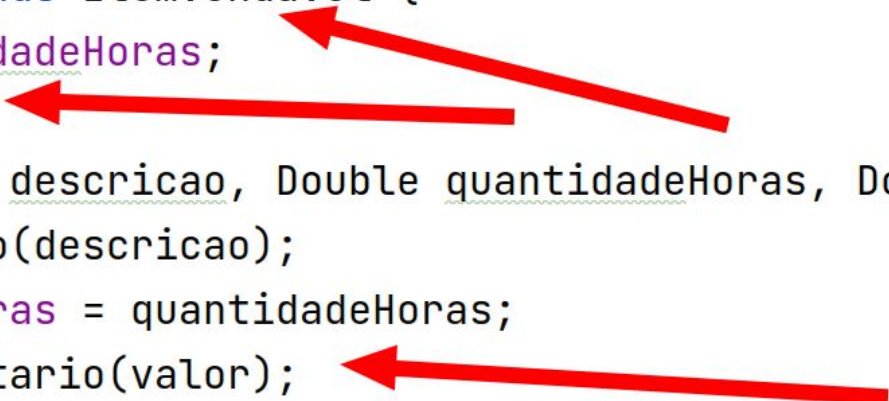
Aqui faremos um novo **refactoring!**

Vamos refatorar a classe Servico

```
public class Servico extends EntityId {  
  
    private String descricao;  
    private Double quantidadeHoras;  
    private Double valor;  
  
    public Servico(String descricao, Double quantidadeHo  
        this.descricao = descricao;  
        this.quantidadeHoras = quantidadeHoras;  
        this.valor = valor;  
}
```

Servico vai ficar assim!

```
public class Servico extends ItemVendavel {  
    private Double quantidadeHoras;  
  
    public Servico(String descricao, Double quantidadeHoras, Double valor)  
    {  
        super.setDescricao(descricao);  
        this.quantidadeHoras = quantidadeHoras;  
        super.setValorUnitario(valor);  
    }  
  
    public Double getQuantidadeHoras() { return quantidadeHoras; }
```

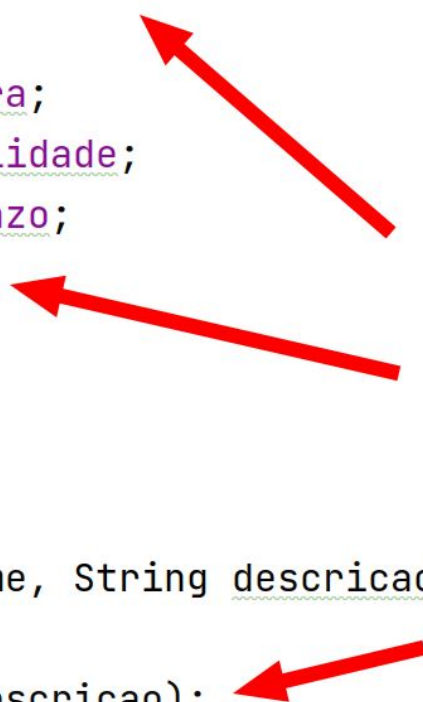


Vamos refatorar a classe Produto

```
public class Produto extends EntityId {  
    private String nome;  
    private String descricao;  
    private Double precoVenda;  
    private Double precoCompra;  
    private LocalDate dataValidade;  
    private LocalDate dataPrazo;  
    private Status status;  
  
    public Produto() {  
    }  
  
    public Produto(String nome, String descricao) {  
        this.nome = nome;  
        this.descricao = descricao;  
    }  
}
```


Vamos refatorar a classe Produto

```
public class Produto extends ItemVendavel {  
    private String nome;  
    private Double precoCompra;  
    private LocalDate dataValidade;  
    private LocalDate dataPrazo;  
    private Status status;  
  
    public Produto() {  
    }  
  
    public Produto(String nome, String descricao) {  
        this.nome = nome;  
        super.setDescricao(descricao);  
    }  
}
```



Vamos refatorar a classe Produto

```
public Double calculaMargemDeLucro() {  
    double lucro = super.getValorUnitario() - precoCompra;  
    double margemLucro = (lucro / super.getValorUnitario()) * 100;  
    return margemLucro;  
}
```



Beleza! Aqui temos a seguinte situação, Produto e Serviço herdam as propriedades de ItemVendavel.

```
public class ItemVenda {  
    private Servico servico;  
    private Produto produto;  
    private Double valorUnitario;  
    private Double quantidade;  
    private Double desconto;
```



```
public class ItemVenda {  
    private ItemVendavel produtoServico;  
    private Double valorUnitario;  
    private Double quantidade;  
    private Double desconto;
```

ItemVendavel se torna polimórfico neste cenário, podendo receber um Produto ou Serviço.

```
Venda venda01 = new Venda();
venda01.setId(10L);
venda01.setDataVenda(LocalDate.of(2023, 1, 1));
venda01.setCliente(bruno);
venda01.setFormaPagamento(FormaPagamento.A_VISTA);
venda01.setObservacao("Observação 01");

Produto produto = new Produto("Computador", "I5 8gb");
ItemVenda itemVenda = new ItemVenda(produto, 100.00, 1.0, 10.0);
venda01.addItemVenda(itemVenda);

Servico servico = new Servico("Instalação Office", 2.0, 100.00);
ItemVenda itemVenda2 = new ItemVenda(servico, 100.00, 1.0, 10.0);
venda01.addItemVenda(itemVenda2);

System.out.println(venda01.getItems());
```

```
Application x
/home/bruno.kurzawe/documentos/senac/ExemploAula/Exemplo00001/target/classes org.example
.Application
[ItemVenda{produto=Produto{nome='Computador', descricao='I5 8gb'}, valorUnitario=100.0,
quantidade=1.0, desconto=10.0}, ItemVenda{produto=Servico{descricao=Instalação Office,
quantidadeHoras=2.0}, valorUnitario=100.0, quantidade=1.0, desconto=10.0}]

Process finished with exit code 0
```


Fim da aula 03...