**Java – aplicando a Orientação a Objetos**

**Classes e objetos**

Uma classe é uma estrutura que define um tipo de objeto. A classe é como um molde, que define quais são as características (atributos) e comportamentos (métodos) que os objetos desse tipo vão possuir. Por exemplo, podemos definir uma classe chamada Pessoa que tenha os atributos nome e idade, além do método fazAniversario(). A sintaxe para definir essa classe em Java é a seguinte:

**public** **class** **Pessoa** {

String nome;

int idade;

**void** **fazAniversario**() {

idade++;

}

}

Já um objeto, é uma instância de uma classe, sendo por meio dele que conseguimos representar informações na aplicação, pois a classe serve apenas para padronizar os objetos, mas não para representar um objeto em si. Para criar um objeto em Java, precisamos utilizar a palavra reservada new seguida do nome da classe e de parênteses vazios. Por exemplo, para criar um objeto do tipo Pessoa, podemos escrever o seguinte código:

**Pessoa** pessoa1 = **new** **Pessoa**();

Nesse exemplo, criamos um objeto a partir da classe Pessoa, e guardamos na variável pessoa1 uma referência a ele. Como esse objeto foi criado como sendo do tipo Pessoa, ele possui os atributos nome e idade, além do método fazAniversario(). É a partir do objeto que podemos de fato representar uma pessoa, ou seja, preencher os atributos e chamar os métodos. Por exemplo:

Pessoa pessoa1 = **new** Pessoa();

pessoa1.nome = "Ana";

pessoa1.idade = 20;

System.**out**.println(pessoa1.nome + " tem " + pessoa1.idade + " anos");

pessoa1.fazAniversario();

System.**out**.println("A idade agora é: " +pessoa1.idade);

Podemos criar vários objetos do tipo Pessoa, sendo que todos eles terão os mesmos atributos e métodos, já que são da mesma classe, mas cada um pode possuir informações distintas. Por exemplo:

Pessoa pessoa1 = **new** Pessoa();

pessoa1.nome = "Ana";

pessoa1.idade = 20;

System.**out**.println(pessoa1.nome + " tem " + pessoa1.idade + " anos");

pessoa1.fazAniversario();

System.**out**.println("A idade agora é: " +pessoa1.idade);

Pessoa pessoa2 = **new** Pessoa();

pessoa2.nome = "Carlos";

pessoa2.idade = 20;

System.**out**.println(pessoa2.nome + " tem " + pessoa2.idade + " anos");

pessoa2.fazAniversario();

System.**out**.println("A idade agora é: " +pessoa2.idade);

**Manipulando objetos**

Alguém, que está iniciando os estudos em Java, criou a seguinte classe para representar uma **Venda**:

**public** **class** **Venda** {

String produto;

**int** quantidade;

**double** valorUnitario;

**double** valorTotal;

}

No entanto, ao criar uma instância dessa classe em um método **main**, você percebeu que ela escreveu o seguinte código contendo alguns erros:

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

**Venda** venda1 = **Venda**();

venda1.produto = "Camiseta";

venda1.quantidade = "2";

venda1.valorUnitario = 25.50;

**System**.out.**println**("Valor total da venda: " + venda1.quantidade \* venda1.valorUnitario);

}

Quais os erros cometidos no código acima?

* O atributo quantidade foi preenchido de maneira incorreta.

O atributo quantidade foi definido como sendo do tipo **int** na classe **Venda**, não sendo permitido então passar a **String** “2” ao atribuí-lo no objeto **venda1**.

* Existe um erro ao instanciar o objeto venda1.

Faltou a palavra reservada **new** para criar a instância do objeto, causando com isso um erro de compilação.

**Definição de métodos**

Suponha que você esteja trabalhando em uma aplicação Java de uma loja, que tenha a seguinte classe:

**public** **class** **Produto** {

String nome;

**double** preco;

**double** descontoParaPix;

**double** **pegaPrecoFinal**(**boolean** pagamentoViaPix) {

**if** (pagamentoViaPix == **true**) {

**double** precoFinal = preco - descontoParaPix;

} **else** {

**double** precoFinal = preco;

}

}

}

O método **pegaPrecoFinal** realiza o cálculo do preço final do produto, levando em consideração se o pagamento será realizado via pix. No entanto, esse método foi definido de maneira incorreta. Qual o problema do método **pegaPrecoFinal**?

* Falta a linha após o if-else que retorna o preco final: **return precoFinal;**

**Modificadores de acesso**

Em Java, os modificadores de acesso são palavras-chave que definem o nível de visibilidade de classes, atributos e métodos, sendo que eles ajudam a garantir a **segurança** e **encapsulamento do código**. Existem quatro tipos de modificadores de acesso em Java: **public**, **protected**, **private** e **default** (também conhecido como **package-private**).

* **Public:** O modificador de acesso **public** é o mais permissivo de todos. Uma classe, atributo ou método declarado como public ***pode ser acessado por qualquer classe em qualquer pacote***. Ou seja, ele possui visibilidade pública e pode ser utilizado livremente. Por exemplo:

**public** **class** **Conta** {

**public** **double** saldo;

**public** **void** **sacar**(**double** valor) {

// lógica de saque...

}

}

**public** **class** **Principal** {

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

**Conta** c1 = **new** **Conta**();

c1.saldo = 300;

c1.**sacar**(100);

}

}

* **Default (Package-private):** O modificador de acesso **default** é aquele que não especifica nenhum modificador de acesso. Quando nenhum modificador de acesso é especificado, a classe, atributo ou método ***pode ser acessado apenas pelas classes que estão no mesmo pacote***. Por exemplo:

**package** br.com.alura.conta;

**public** **class** **Conta** {

**default** **double** saldo;

**default** **void** **sacar**(**double** valor) {

// lógica de saque...

}

}

**package** br.com.alura.testes;

**public** **class** **Principal** {

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

**Conta** c1 = **new** **Conta**();

c1.saldo = 300;

c1.sacar(100);

}

}

No código anterior, a classe **Conta** está em um pacote e a classe **Principal** em outro pacote distinto. A classe **Conta** pode ser instanciada dentro da classe **Principal**, pois ela possui o modificador de acesso **public**, entretanto, o atributo **saldo** e o método **sacar()** tem o modificador **default** e, ***portanto, não podem ser acessados de dentro da classe Principal***, o que vai causar um erro de compilação no código anterior.

* **Private:** O modificador de acesso **private** é o mais restritivo de todos. Uma classe, atributo ou método declarado como private ***só pode ser acessado dentro da própria classe***. Ou seja, ele possui visibilidade restrita e não pode ser utilizado por outras classes. Por exemplo:

**public** **class** **Conta** {

**private** **double** saldo;

**private** **void** **sacar**(**double** valor) {

// lógica de saque...

}

}

**public** **class** **Principal** {

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

**Conta** c1 = **new** **Conta**();

c1.saldo = 300;

c1.**sacar**(100);

}

}

No código anterior, vai ocorrer erro de compilação na classe **Principal**, pois o atributo **saldo** e o método **sacar()** foram declarados como **private**, ***não podendo com isso serem acessados de fora da própria classe Conta.***

Existe ainda um último modificador de acesso, que é o **protected**, mas falaremos dele mais adiante no curso, após ser apresentado ***o conceito de herança de classes***.

**JavaDoc - A documentação do Java**

A documentação do Java, conhecida como **JavaDoc**, é uma referência completa para todas as classes, interfaces e métodos disponíveis na plataforma Java. Ela está disponível online e pode ser acessada neste [site da Oracle](https://docs.oracle.com/en/java/javase/17/docs/api/index.html).

O **JavaDoc** é organizado em pacotes, que contêm diversas classes e interfaces relacionadas a alguma funcionalidade específica da linguagem. Cada classe e interface é documentada individualmente, com informações detalhadas sobre sua funcionalidade, uso e exemplos de código.

Essa documentação deve ser uma **referência essencial** para qualquer pessoa que queira programar na linguagem Java, pois fornece informações detalhadas sobre como usar as classes e interfaces fornecidas pela linguagem. Tendo acesso à documentação, podemos também descobrir novas classes e métodos que podem ser úteis em nossos projetos, para resolver problemas comuns e recorrentes do dia a dia.

**Padrão de nomes de pacotes**

Em Java, pacotes são usados para agrupar classes que estão relacionadas a alguma funcionalidade. Eles ajudam a organizar o código, facilitando o seu gerenciamento e evitando que centenas de classes distintas fiquem todas misturadas num único diretório.

A organização de classes dentro de um pacote também pode ser feita usando subpacotes ou colocando as classes diretamente dentro do pacote, dependendo da complexidade do projeto. Além disso, é importante seguir a convenção de nomeação de pacotes para garantir que os pacotes sejam facilmente identificados.

Em relação à nomenclatura de pacotes, outro aspecto importante é o uso do nome de domínio reverso da empresa ou organização como parte do nome do pacote. Por exemplo, se o nome de domínio da empresa fosse "minhaempresa.com.br" e o projeto Java fosse chamado de "meuprojeto", o nome do pacote de domínio reverso seria br.com.minhaempresa.meuprojeto, sendo que dentro desse pacote principal podemos ter diversos subpacotes, para melhor organização do código do projeto.

Isso ajuda a garantir que o nome do pacote seja exclusivo e evita conflitos de nome com outros pacotes em outros projetos. Além disso, o uso do nome de domínio reverso como parte do nome do pacote também ajuda a identificar facilmente o proprietário do projeto Java.

**A palavra-chave ''this''**

“This”, traduzindo para o português (Isto/este/esta), é usado para fazer referência aos atributos da classe, especialmente em métodos que têm parâmetros com o mesmo nome do atributo da classe em que estamos trabalhando. Para compreendermos essa ideia na prática, vamos ver um exemplo:

**public** **class** **Conta** {

**private** **double** saldo;

**private** **int** numero;

**public** **void** **deposita**(**double** valor) {

saldo = saldo + valor;

}

}

Repare que nesse primeiro exemplo, dentro da classe conta, nós temos os atributos **saldo** e **numero**, e em seguida o método deposita, que recebe como parâmetro um double **valor**. Nesse caso, não temos nenhum atributo com o mesmo nome do parâmetro e estamos dentro da classe, o que nos permite utilizar o atributo saldo a vontade, sem a necessidade do “this”.

Algo diferente ocorre quando temos um atributo da classe com o mesmo nome de um parâmetro de algum método dessa mesma classe:

**public** **class** **Lampada** {

**private** boolean ligada;

**private** **String** modelo;

**public** void **acendeLampada**(boolean ligada) {

ligada = ligada;

}

}

Perceba que nesse exemplo, temos um atributo da classe chamado **ligada**, e o método **acendeLampada** que define um novo valor para esse atributo. O método recebe como parâmetro um boolean que tem o mesmo nome do atributo da classe. ***É aí que está o problema! O que faríamos para saber quem é o atributo da classe e quem é o parâmetro do método?*** Nessa situação vamos fazer o uso da palavra-chave “this”:

**public** **class** **Lampada** {

**private** boolean ligada;

**private** **String** modelo;

**public** void **acendeLampada**(boolean ligada) {

this.ligada = ligada;

}

}

Podemos concluir então que “this” se refere ao objeto atual e não ao parâmetro do método. É comum usarmos o this para eliminar essa confusão entre os atributos e parâmetros, sendo que ele não é uma exclusividade do Java, pois outras linguagens de programação orientadas a objetos também possuem esse recurso.

**Código encapsulado**

Imagine que você esteja analisando o código de uma aplicação Java e encontra a seguinte classe:

**public** **class** **Conta** {

**private** int numero;

**private** int agencia;

**private** double saldo;

**public** **void** **deposita**(double valor) {

saldo += valor;

}

**public** **void** **saca**(double valor) {

**if** (valor <= saldo) {

saldo -= valor;

} **else** {

System.**out**.println("Saldo insuficiente.");

}

}

**public** **void** **setNumero**(int numero) {

**this**.numero = numero;

}

**public** **void** **setAgencia**(int agencia) {

**this**.agencia = agencia;

}

**public** **void** **setSaldo**(double saldo) {

**this**.saldo = saldo;

}

**public** int **getNumero**() {

**return** numero;

}

**public** int **getAgencia**() {

**return** agencia;

}

**public** double **getSaldo**() {

**return** saldo;

}

}

Em relação ao conceito de encapsulamento, existe um problema do código, qual?

* A classe não deveria ter o método setSaldo.

O atributo saldo somente deveria ser modificado pela própria classe, por meio dos métodos **sacar** e **depositar**, mas ao declarar um método setter para ele o encapsulamento será quebrado, pois o seu valor poderá ser modificado arbitrariamente de fora da classe.

**Entendendo melhor a herança**

A herança é um conceito fundamental da orientação a objetos, sendo implementada em Java através da relação é um entre classes. Isso significa que uma classe pode herdar atributos e métodos de outra classe, tornando com isso o código mais reutilizável. No Java, a herança é realizada através da palavra-chave **extends**. A classe que herda é chamada de **subclasse**, e a classe que é herdada é chamada de **superclasse**. A subclasse pode acessar todos os atributos e métodos públicos e protegidos da superclasse, além de poder sobrescrever os métodos da superclasse para criar comportamentos específicos. Por exemplo:

**public** **class** **Conta** {

**private** String titular;

**private** double saldo;

**public** **void** **sacar**(double valor) {

**if** (valor <= 0) {

System.**out**.println("Valor deve ser maior do que zero!");

} **else** **if** (saldo >= valor) {

saldo -= valor;

System.**out**.println("Saque realizado com sucesso. Saldo atual: " +saldo);

} **else** {

System.**out**.println("Saldo insuficiente.");

}

}

**public** **void** **depositar**(double valor) {

**if** (valor > 0) {

saldo += valor;

System.**out**.println("Depósito realizado com sucesso. Saldo atual: " +saldo);

} **else** {

System.**out**.println("Valor deve ser maior do que zero!");

}

}

//getters e setters

}

**public** **class** **ContaPoupanca** **extends** **Conta** {

**private** double taxaDeJuros;

**public** **void** **calcularJuros**() {

double juros = **this**.getSaldo() \* taxaDeJuros;

System.**out**.println("Juros atual: " +juros);

}

**public** **void** **sacar**(double valor) {

double taxaSaque = 0.01;

super.sacar(valor + taxaSaque);

}

//getters e setters

}

No código anterior, a classe **Conta** é a superclasse e a classe **ContaPoupanca** é a subclasse. A classe **ContaPoupanca** herda os atributos e métodos da classe **Conta**, e adiciona um novo atributo **taxaDeJuros** e um novo método **calcularJuros**. Embora os atributos sejam herdados, como eles foram declarados como private na superclasse, não poderão ser acessados diretamente na subclasse, devendo então serem utilizados os métodos getters/setter, que são públicos. Repare também no código anterior que a subclasse sobrescreveu o método **sacar**, para que seja descontada a taxa de saque, além de utilizar a palavra-chave **super** para chamar o método da superclasse, evitando com isso duplicar um código já existente. Essa é a grande vantagem da herança: reaproveitamento de código com flexibilidade para sobrescrever comportamentos.

**Herança múltipla**

**Em Java, é importante notar que a herança múltipla não é permitida**. A herança múltipla ocorre quando uma subclasse herda de duas ou mais superclasses. Por exemplo:

**public** **class** **ContaPoupanca** **extends** **Conta**, **Pagamento** {

//codigo da classe omitido

}

O código anterior não compila, pois o **extends** aceita apenas uma única classe, ou seja, uma classe pode ter apenas uma superclasse. **Entretanto, é possível criar uma hierarquia de classes utilizando herança, simulando com isso uma herança múltipla.** Por exemplo:

**public** **class** **Conta** {

//codigo da classe omitido

}

**public** **class** **ContaCorrente** **extends** **Conta** {

//codigo da classe omitido

}

**public** **class** **ContaCorrentePessoaFisica** **extends** **ContaCorrente** {

//codigo da classe omitido

}

No código anterior, a classe **ContaCorrentePessoaFisica** está herdando de **ContaCorrente**, que por sua vez herda da classe **Conta**, ou seja, indiretamente a classe **ContaCorrentePessoaFisica** vai herdar de **Conta**, pois sua superclasse herda dela.

**Anotações no Java**

As anotações, também conhecidas como **annotations**, são uma forma de adicionar configurações ao código Java de uma maneira bem simples. Elas são usadas para fornecer informações adicionais sobre o código, como o significado de uma classe, um método ou um atributo. No Java, as anotações são definidas com o uso do símbolo "**@**" seguido do nome da anotação. Por exemplo, a anotação **@Deprecated** é usada para indicar que um método ou classe está obsoleto e não deve ser mais utilizado. Outra anotação muito útil é a **@Override** que indica que um método está sobrescrevendo o mesmo método definido na sua classe mãe.

Diversas especificações e frameworks Java, como Hibernate, Bean Validation e Spring, utilizam anotações. Por exemplo, no Bean Validation a anotação **@NotNull** é usada para validar que um atributo não seja nulo.

As anotações são muito úteis e comuns em aplicações Java, sendo importante estar familiarizado com esse recurso, pois certamente será utilizado bastante nos projetos.

**O modificador protected**

Ao usar herança no Java, temos a possibilidade de utilizar o modificador de acesso **protected**, que ***permite que os atributos de uma classe sejam acessados por outras classes do mesmo pacote e por suas subclasses***, independentemente do pacote em que se encontram.

O modificador **protected** é útil em situações em que uma classe precisa permitir que suas subclasses acessem diretamente seus atributos, sem a necessidade de restringir o acesso apenas pelos métodos **getters** e **setters**. Por exemplo, suponha que temos as seguintes classes em um projeto:

**public** **class** **Conta** {

**private** String titular;

**private** **double** saldo;

**public** **void** **sacar**(**double** valor) {

//implementacao do metodo omitida

}

**public** **void** **depositar**(**double** valor) {

//implementacao do metodo omitida

}

//getters e setters

}

**public** **class** **ContaPoupanca** **extends** **Conta** {

**private** double taxaDeJuros;

**public** **void** **calcularJuros**() {

double juros = **this**.getSaldo() \* taxaDeJuros;

System.**out**.println("Juros atual: " +juros);

}

//getters e setters

}

No código anterior, repare que no método **calcularJuros**, da classe ContaPoupanca, o atributo saldo não é acessado diretamente, pois ele foi declarado como **private** na classe **Conta**, devendo então seu acesso ser feito pelo método **getSaldo()**.

Podemos declarar o atributo saldo como **protected**, para evitar essa situação e liberar o acesso direto a ele pelas classes que herdam da classe **Conta**:

**public** **class** **Conta** {

**private** String titular;

**protected** **double** saldo;

**public** **void** **sacar**(**double** valor) {

//implementacao do metodo omitida

}

**public** **void** **depositar**(**double** valor) {

//implementacao do metodo omitida

}

//getters e setters

}

**public** **class** **ContaPoupanca** **extends** **Conta** {

**private** double taxaDeJuros;

**public** **void** **calcularJuros**() {

double juros = **this**.saldo \* taxaDeJuros;

System.**out**.println("Juros atual: " +juros);

}

//getters e setters

}

Repare que agora o atributo **saldo** foi acessado diretamente pela classe **ContaPoupanca**.

**Herdando classes**

Considere a seguinte hierarquia de classes em Java:

**public** **class** **Animal** {

**private** **String** nome;

**private** double peso;

**public** void **fazerBarulho**() {

**System**.out.**println**("fazendo barulho!");

}

**public** **String** **getNome**() {

**return** nome;

}

**public** void **setNome**(String nome) {

this.nome = nome;

}

**public** double **getPeso**() {

**return** peso;

}

**public** void **setPeso**(double peso) {

this.peso = peso;

}

}

**public** **class** **Cachorro** **extends** **Animal** {

**public** **void** **fazerBarulho**() {

System.**out**.println("Au au au!");

}

**public** **void** **buscarOsso**() {

System.**out**.println("Buscando o osso!");

}

}

**public** **class** **Gato** **extends** **Animal** {

**public** **void** **brincarComLaser**() {

System.**out**.println("Brincando com o laser!");

}

}

Em relação ao conceito de herança, podemos constatar nos códigos acima que:

* A classe **Gato** não pode herdar da classe **Cachorro**.

Como a classe **Gato** já está herdando de **Animal**, não pode herdar também características de **Cachorro**, pois no Java não é permitido herança múltipla.

* Um objeto do tipo **Animal** pode ser instanciado e fazer barulho, mas não é possível chamar o método **buscarOsso()**.

O método **buscarOsso()** é exclusivo da classe **Cachorro**, portanto não é possível chamá-lo em um objeto do tipo **Animal**.

**Entendendo melhor interfaces**

Em Java, interfaces são **uma forma de definir um contrato que as classes devem seguir**, sendo que ele define quais métodos devem ser implementados pelas classes que o implementarem. Interfaces permitem que diferentes classes possam ser tratadas de maneira padronizada, via **polimorfismo**, tornando assim o código fácil de estender com novos comportamentos. No Java, uma interface é definida usando a palavra-chave interface. Por exemplo:

**public** **interface** **Tributavel** {

double **getValorImposto**();

}

No exemplo de código anterior, estamos definindo uma interface chamada Tributavel, sendo que ela possui apenas um método chamado **getValorImposto()** que retorna um valor do tipo **double**. Essa interface pode ser implementada por qualquer classe que queira ser tributável no projeto.

Para implementar uma interface, usamos a palavra-chave **implements** após a definição da classe. A classe que implementa a interface deve implementar **todos** os métodos definidos na interface. Por exemplo:

**public** **class** **Produto** **implements** **Tributavel** {

**private** String nome;

**private** **double** valor;

@Override

**public** **double** **getValorImposto**() {

**return** this.valor \* 0.1;

}

//getters e setters

}

No exemplo anterior, estamos criando uma classe chamada **Produto** que implementa a interface **Tributavel**. Essa classe implementa o método **getValorImposto()**, que está definido na interface **Tributavel**, com uma lógica de que o imposto do produto é igual a 10% do seu valor. Também poderíamos ter uma classe de serviços, conforme abaixo

**public** **class** **Servico** **implements** **Tributavel** {

**private** String descricao;

**private** **double** valor;

**private** **double** aliquotaISS;

@Override

**public** **double** **getValorImposto**() {

**return** this.valor \* this.aliquotaISS / 100;

}

//getters e setters

}

No exemplo acima, estamos criando uma classe chamada **Servico** que implementa a interface **Tributavel**. Essa classe implementa o método **getValorImposto()**, que está definido na interface **Tributavel**, com uma lógica de que o imposto do serviço é igual ao seu valor vezes a alíquota de ISS definida e dividido por 100. Então para um serviço de R$ 1.200,00 e alíquota de 5%, o método retornará: 1200 \* 5 / 100, cujo valor do imposto fica R$ 60,00.

**Utilização de interfaces**

Interfaces podem ser utilizadas para definir comportamentos que podem ser aplicados a várias classes diferentes, tornando assim o código mais modular e fácil de manter. Por exemplo, suponha que temos um sistema de vendas que precisa calcular o imposto de diferentes tipos de produtos. Podemos criar a interface **Tributavel**, para definir o comportamento de calcular imposto, e criar várias classes diferentes que implementam essa interface para calcular o imposto de diferentes produtos. Por exemplo:

**public** **class** **CalculadoraImposto** {

**private** double totalImposto = 0;

**public** **void** **calcularImposto**(Tributavel item) {

**this**.totalImposto += item.getValorImposto();

}

**public** double **getTotalImposto**() {

**return** **this**.totalImposto;

}

}

Nesse exemplo, estamos criando uma classe chamada **CalculadoraImposto**, que tem um atributo privado chamado **totalImposto**, que armazena o valor total dos impostos. Repare que o método **calcularImposto** recebe um parâmetro do tipo **Tributavel**. ***Ao declarar uma variável com o tipo de uma interface, como é feito nesse método, podemos atribuir a essa variável qualquer objeto que implemente essa interface***, ou seja, tanto um objeto do tipo **Servico** quanto **Produto**. Para ambos os casos, a **CalculadoraImposto** irá chamar o método implementado na classe específica. Ou seja, para um produto, irá chamar o método **getTotalImposto** implementado na classe **Produto**. E para um serviço, irá chamar o método getTotalImposto implementado na classe **Servico**. Isso é muito útil quando queremos tratar vários objetos de classes diferentes de forma semelhante, permitindo que diferentes classes possam ser tratadas de maneira padronizada, facilitando a manutenção e extensão do código. Esse é mais um exemplo de aplicação do polimorfismo em Java, mas agora com a utilização de interfaces.

**Herança e interface**

Alguém está trabalhando numa aplicação Java de uma loja e precisa implementar uma nova funcionalidade de tributação de impostos. Até então, a aplicação possuía as seguintes classes:

**public** **class** **Item** {

**private** String nome;

**private** **double** preco;

// métodos getters e setters omitidos

}

**public** **class** **Produto** **extends** **Item** {

**private** **double** peso;

// métodos getters e setters omitidos

}

**public** **class** **Servico** **extends** **Item** {

**private** **int** quantidadeHoras;

// outros métodos da classe Servico

}

**public** **class** **Brinde** **extends** **Item** {

**private** String justificativa;

// métodos getters e setters omitidos

}

E para implementar a nova funcionalidade, foi criada uma interface chamada **Tributavel** com o seguinte código:

**public** **interface** **Tributavel** {

double **calculaImposto**();

}

E foi criada uma classe para calcular o total de impostos:

**public** **class** **CalculadoraImposto** {

**private** **double** totalImposto = 0;

**public** **void** **calculaImposto**(Tributavel tributavel) {

totalImposto += tributavel.calculaImposto();

}

}

E para que os **Produtos** e **Serviços** da loja sejam tributados, foi necessário alterar o código dessas respectivas classes para que implementem a interface **Tributavel**:

**public** **class** **Produto** **extends** **Item** **implements** **Tributavel** {

**private** **double** peso;

**public** **double** **calculaImposto**() {

//produto tem 10% de imposto:

**return** getPreco() \* 0.1;

}

// métodos getters e setters omitidos

}

**public** **class** **Servico** **extends** **Item** **implements** **Tributavel** {

**private** **int** quantidadeHoras;

**public** **double** **calculaImposto**() {

//serviço tem 12% de imposto:

**return** getPreco() \* 0.12;

}

// outros métodos da classe Servico

}

Levando em consideração os conceitos de **interface** e **polimorfismo**, o que se pode declarar em relação à implementação da funcionalidade:

* Nem todas as classes filhas de Item precisam implementar a interface **Tributavel**.

A interface **Tributavel** deve ser implementada apenas pelas classes que forem tributáveis, independente do uso da herança.

* O uso da interface **Tributavel** permite que a classe **CalculadoraImposto** possa calcular os impostos de qualquer classe que a implemente, sem precisar conhecer a implementação específica de cada uma delas.

Isso é possível graças ao polimorfismo, que permite que objetos de classes diferentes sejam tratados de forma uniforme, simplificando o código e evitando duplicações.

**Referências**

1. [TechGuide: Orientação a Objetos – Hipsters Ponto Tech #350](https://www.hipsters.tech/praticas-de-orientacao-a-objetos-hipsters-129/) (gratuito, português, podcast): Podcast que discute a orientação a objetos, paradigma essencial em muitas linguagens de programação. O episódio abrange tópicos como encapsulamento, abstração, polimorfismo e muito mais.

2. [Orientação a Objetos: aprenda seus conceitos e suas aplicabilidades de forma efetiva](https://www.casadocodigo.com.br/products/livro-oo-conceitos) (pago, português, livro): O livro apresenta conceitos de orientação a objetos para utilizar da forma mais eficiente. Por meio de exemplos, visa as boas práticas do uso para tornar o processo de desenvolvimento mais produtivo.

3. [Orientação a objetos com Roberta Arcoverde](https://www.youtube.com/watch?v=jpuJ1qrluoU) (gratuito, português, vídeo): Vídeo que apresenta os conceitos fundamentais sobre sobre a utilização de orientação a objeto e suas aplicações.

4. [Java e orientação a objetos](https://www.alura.com.br/apostila-java-orientacao-objetos?_gl=1*ffa1jk*_ga*NjUzNTI4NjEzLjE3MTIwODAxNDE.*_ga_1EPWSW3PCS*MTcxNTM3ODAyOC42Ny4xLjE3MTUzNzgwNzIuMC4wLjA.*_fplc*MHFkVkxETXU1bmtjYUtNT0w3RVRxMXhNNktOSVZucjhOc01RdDBYVHIlMkZEMkZBb2hPVSUyRndReWVRczNybEs2d3VsdVBsOSUyQnhPRG0lMkJWMGkzZCUyRnB1dHdMYUtZWXRkbVJwbTJPZHoxZElFTkhTTUVQazl6d204SVl5S0lKWXUwQSUzRCUzRA..) (gratuito, português, apostila): A apostila oferece uma exploração detalhada dos conceitos de orientação a objetos em Java, começando desde os primeiros passos e incluindo diversos exercícios práticos para reforçar o aprendizado.

5. [POO: O que é programação orientada a objetos?](https://www.alura.com.br/artigos/poo-programacao-orientada-a-objetos?_gl=1*roxqqx*_ga*NjUzNTI4NjEzLjE3MTIwODAxNDE.*_ga_1EPWSW3PCS*MTcxNTM3ODAyOC42Ny4xLjE3MTUzNzgwNzIuMC4wLjA.*_fplc*MHFkVkxETXU1bmtjYUtNT0w3RVRxMXhNNktOSVZucjhOc01RdDBYVHIlMkZEMkZBb2hPVSUyRndReWVRczNybEs2d3VsdVBsOSUyQnhPRG0lMkJWMGkzZCUyRnB1dHdMYUtZWXRkbVJwbTJPZHoxZElFTkhTTUVQazl6d204SVl5S0lKWXUwQSUzRCUzRA..) (gratuito, português, artigo): Este artigo faz uma introdução à Programação Orientada a Objetos (POO) e a compara com a programação estruturada, utilizando exemplos de código em linguagens como Python e Java.