POO Java Alura

O que são comportamento e Métodos? <TODO>

Comportamento e Métodos

Nessa aula falamos sobre o comportamento que são os métodos.

Vimos:

* como definir métodos com parâmetros e retorno
* como retornar algo usando a palavra chave return
* como usar a referência this para acessar um atributo
* que podemos passar uma referência como parâmetro do método
* métodos são chamadas a partir da referência usando o operador .

Se você tiver com dúvidas ainda sobre referências, this e uso dos métodos, fique tranquilo, pois nas próximas aulas (e cursos) vamos revisar os conceitos e praticar muito mais!! Bora continuar?

**Extraindo o que é comum**

Juarez criou as seguintes classes:

public class Pessoa {

String nome;

String cpf;

int idade;

String logradouro;

String complemento;

String numero;

String bairro;

String cidade;

String cep;

}

public class Empresa {

String razaoSocial;

String cnpj;

String logradouro;

String complemento;

String numero;

String bairro;

String cidade;

String cep;

}

Podemos perceber que os atributos logradouro, complemento, numero, bairro, cidade e cep são os mesmos nas duas classes. Essas informações são fortes candidatas para serem externalizadas na classe Endereco e associadas às classes Pessoa e Empresa através de composição.

public class Endereco {

String logradouro;

String complemento;

String numero;

String bairro;

String cidade;

String cep;

}

Atividade

Marque a única alternativa verdadeira que modifica corretamente as classes Pessoa e Empresa para utilizarem a classe Endereco.

a)

public class Pessoa {

String nome;

String cpf;

int idade;

Endereco endereco;

}

public class Empresa {

String razaoSocial;

String cnpj;

Endereco;

}

b)

public class Pessoa {

String nome;

String cpf;

int idade;

}

public class Empresa {

String razaoSocial;

String cnpj;

Endereco endereco;}

c)

public class Pessoa {

String nome;

String cpf;

int idade;

Endereco endereco;

}

public class Empresa {

String razaoSocial;

String cnpj;

Endereco endereco;

}

**Mão na massa: Referências**

1) Caso queira, crie um novo projeto e copie a classe Conta, ou apenas crie uma nova classe Cliente para compor nosso bytebank, nossa classe Cliente deve ter a seguinte cara:

public class Cliente {

String nome;

String cpf;

String profissao;

}

2) Defina os atributos da classe Cliente como preferir, no curso, usamos nome, cpf e profissão.

3) Modifique a classe Conta adicionando um atributo do tipo Cliente.

public class Conta{

double saldo;

int agencia;

int numero;

Cliente titular;

}

4) Crie um classe de teste e instancie um cliente e uma conta! Lembre-se de que o atributo Titular de uma conta guarda um Cliente

5) Repare que podemos fazer de uma forma mais direta!

Conta contaDaMarcela = new Conta();

contaDaMarcela.titular = new Cliente();

Podemos definir o nome de um cliente fazendo:

contaDaMarcela.titular.nome = "Marcela";

**Mão na massa: Criando Getters e Setters**

Vamos incrementar nosso código, modificando-o de acordo com as aulas.

1) Vamos proteger os atributos da classe Conta, para isso adicione a palavra reservada private:

private double saldo;

private int agencia;

private int numero;

private Cliente titular;

Repare que quando fazemos isso, não conseguiremos mais acessar os valores.

2) Precisamos criar métodos que nos possibilite receber os valores dos atributos, ou até alterá-los, caso seja interessante. Para isso criaremos os Getters e os Setters, siga os modelos abaixo e modifique para que retorne/altere o atributo correto!

Não esqueça de alterar o tipo de retorno do método! Nesse caso é double

//Getter

public double getSaldo(){

return this.saldo;

}

//Setter

public void setNumero(int numero){

this.numero = numero;

}

No curso criamos getSaldo, getNumero, setNumero, getAgencia, setAgencia, setTitular, getTitular.

3) Na classe Cliente, escreva os métodos getters e setters e modifique os atributos adicionando private

Faça uma classe de teste e veja as possibilidades, dessa forma, esse conhecimento será consolidado!

Nosso código está abaixo, apenas clique para ver a opinião, mas recomendamos que tente fazer antes de ver o código completo!

Nossa classe Conta está da seguinte forma:

public class Conta {

private double saldo;

private int agencia;

private int numero;

private Cliente titular;

public void deposita(double valor) {

this.saldo = this.saldo + valor;

}

public boolean saca(double valor) {

if(this.saldo >= valor) {

this.saldo -= valor;

return true;

} else {

return false;

}

}

public boolean transfere(double valor, Conta destino) {

if(this.saldo >= valor) {

this.saldo -= valor;

destino.deposita(valor);

return true;

}

return false;

}

public double getSaldo(){

return this.saldo;

}

public int getNumero(){

return this.numero;

}

public void setNumero(int numero){

this.numero = numero;

}

public int getAgencia(){

return this.agencia;

}

public void setAgencia(int agencia){

this.agencia = agencia;

}

public void setTitular(Cliente titular){

this.titular = titular;

}

public Cliente getTitular(){

return this.titular;

}

}

Nossa classe **Cliente**:

public class Cliente {

private String nome;

private String cpf;

private String profissao;

public String getNome() {

return nome;

}

public void setNome(String nome) {

this.nome = nome;

}

public String getCpf() {

return cpf;

}

public void setCpf(String cpf) {

this.cpf = cpf;

}

public String getProfissao() {

return profissao;

}

public void setProfissao(String profissao) {

this.profissao = profissao;

}

}

public void setSaldo(double saldo){

this.saldo = saldo;

}

public double getSaldo(){

return saldo;

}

}

Continuamos usando atributos privados e nosso modelo parece seguir perfeitamente a proposta do encapsulamento onde a própria classe gerencia o seus estados(atributos). Uma utilização clássica dessa Conta nos levaria ao seguinte cenário:

Conta conta = new Conta();

conta.setTitular("Fábio")

conta.setSaldo(100.0);

Tudo parece perfeito, agora imagine que seja necessário sacar 50.0 dessa conta. Essa operação vai exigir que o saldo seja suficiente. Uma simples verificação como a seguir asseguraria que o saldo não tenha ficado negativo. Nesse nosso exemplo não há limite além do saldo :)

Conta conta = new Conta();

conta.setTitular("Fábio")

conta.setSaldo(100.0);

double valorSaque = 50.0

if(conta.getSaldo() >= valorSaque){

double novoSaldo = conta.getSaldo() - valorSaque;

conta.setSaldo(novoSaldo)

}

Funcionou! Mas um problema é que essa lógica de restringir o saque à quantidade de saldo vai ter que ser refeita toda vez que for necessária uma operação de saque na nossa conta.

Além do problema de duplicações desse trecho, um problemão para encapsulamento é que quem está de fato controlando as regras do saldo da conta é quem está usando a Conta.

Em outras palavras nada impede que alguém implemente um limite extra para isso e tenha uma regra completamente diferente dos demais objetos do tipo Conta:

Conta conta = new Conta();

conta.setTitular("Fábio");

conta.setSaldo(100.0);

double valorSaque = 50.0;

if(conta.getSaldo() + 1000.0 >= valorSaque){

double novoSaldo = conta.getSaldo() - valorSaque;

conta.setSaldo(novoSaldo)

}

Quando construímos classes que se limitam a ter atributos privados com os setters e getters normalmente dizemos que são classes que só carregam valor, por isso são comumente chamados de classes fantoches ou Value Objects.

Uma classe fantoche é a que não possui responsabilidade alguma, a não ser carregar um punhado de atributos! Definitivamente isso não é a Orientação a Objetos! Esse modelo embora usado em alguns momentos não deve ser prática comum ao desenvolver o domínio do nosso projeto com risco de se cair no Modelo Anêmico que é exatamente o que a Conta hoje é. Uma classe onde os dados e comportamentos/lógicas não estão juntos.

Voltando ao nosso exemplo da Conta, percebe-se que no mundo real as operações poderiam ser representadas com métodos como saca( ) e deposita( ) em vez de só termos setSaldo( ):

class Conta{

private String titular;

private double saldo;

public void setTitular(String titular){

this.titular = titular;

}

public String getTitular(){

return titular;

}

public void saca(double valor){

if(valor > 0 && saldo >= valor){

saldo -= valor;

}

}

public void deposita(double valor){

if(valor>0){

saldo += valor;

}

}

public double getSaldo(){

return saldo;

}

}

Perceba que as lógicas de saque e depósito estão representados dentro da classe e além disso nosso setSaldo() deixa de fazer sentido para o usuário da Conta. A maneira de interagir com o saldo é sempre via uma das operações anteriores:

Conta conta = new Conta();

conta.setTitular("Fábio");

conta.deposita(100.0);

double valorSaque = 50.0;

conta.saca(valorSaque);

double valorDeposito = 70.0;

conta.deposita(valorDeposito)

Muito melhor não é mesmo? Nada de duplicações de código por aí e muito menos outras classes controlando o estado da nossa Conta como tínhamos anteriormente.

**Conclusão**

Setters e Getters devem ser usados com cautela e nem todos os atributos privados precisam ser expostos por meio desses dois métodos com riscos de cairmos em um modelo anêmico que tem os seus comportamentos controlados por outras classes.

Aprendemos nesta aula:

* Atributos privados, restringindo o acesso aos atributos
* Encapsulamento de código
* Métodos de leitura dos atributos, os getters
* Métodos de modificação dos atributos, os setters
* Getters e Setters de referência

Mãos na massa: Criando construtores e variáveis estáticas

Vamos começar a escrever nossos métodos.

1) Adicione o construtor Conta, de mesmo nome da classe:

public Conta(){

System.out.println("Criando uma conta");

}

2) Faça a inicialização dos atributos da classe Conta dentro do construtor:

public Conta(int agencia, int numero){

this.agencia = agencia;

this.numero = numero;

this.saldo = 100; //isso significa que toda conta começa com 100 de saldo.

System.out.println("Estou criando uma conta");

}

3) Para que seu construtor receba os parâmetros, devemos adicionar o que queremos passar na hora que construímos nosso objeto no método main:

public static void main(String args[]){

Conta conta = new Conta(123,456);

4) Agora podemos utilizar nos getters e setters para pegarmos/alterarmos informações desejadas.

5) Crie a variável private static int total = 0; dentro da classe Conta (não dentro do construtor):

public class Conta{

//outros atributos omitidos

private static int total = 0;

//outros métodos omitidos

}

6) Dentro do construtor adicione total++;

7) Crie o método estático getter getTotal():

public static int getTotal(){

return Conta.total;

}

8) Teste o getter em seu método main:

System.out.println(Conta.getTotal());

9) Rode o método main e fique atento no console.

Nosso código será apresentado abaixo, mas é muito importante tentar fazer sem olhar o resultado final!

Mãos na massa: Criando construtores e variáveis estáticas

Vamos começar a escrever nossos métodos.

1) Adicione o construtor Conta, de mesmo nome da classe:

public Conta(){

System.out.println("Criando uma conta");

}

2) Faça a inicialização dos atributos da classe Conta dentro do construtor:

public Conta(int agencia, int numero){

this.agencia = agencia;

this.numero = numero;

this.saldo = 100; //isso significa que toda conta começa com 100 de saldo.

System.out.println("Estou criando uma conta");

}

3) Para que seu construtor receba os parâmetros, devemos adicionar o que queremos passar na hora que construímos nosso objeto no método main:

public static void main(String args[]){

Conta conta = new Conta(123,456);

4) Agora podemos utilizar nos getters e setters para pegarmos/alterarmos informações desejadas.

5) Crie a variável private static int total = 0; dentro da classe Conta (não dentro do construtor):

public class Conta{

//outros atributos omitidos

private static int total = 0;

//outros métodos omitidos

}

6) Dentro do construtor adicione total++;

7) Crie o método estático getter getTotal():

public static int getTotal(){

return Conta.total;

}

8) Teste o getter em seu método main:

System.out.println(Conta.getTotal());

9) Rode o método main e fique atento no console.

Nosso código será apresentado abaixo, mas é muito importante tentar fazer sem olhar o resultado final!

Nossa classe Conta:

public class Conta {

private double saldo;

private int agencia;

private int numero;

private Cliente titular;

private static int total = 0;

public Conta(int agencia, int numero){

Conta.total++;

System.out.println("O total de contas é " + Conta.total);

this.agencia = agencia;

this.numero = numero;

this.saldo = 100;

System.out.println("Estou criando uma conta" + this.numero);

}

public void deposita(double valor) {

this.saldo = this.saldo + valor;

}

public boolean saca(double valor) {

if(this.saldo >= valor) {

this.saldo -= valor;

return true;

} else {

return false;

}

}

public boolean transfere(double valor, Conta destino) {

if(this.saldo >= valor) {

this.saldo -= valor;

destino.deposita(valor);

return true;

}

return false;

}

public double getSaldo(){

return this.saldo;

}

public int getNumero(){

return this.numero;

}

public void setNumero(int numero){

if(numero <= 0) {

System.out.println("Nao pode valor menor igual a 0");

return;

}

this.numero = numero;

}

public int getAgencia(){

return this.agencia;

}

public void setAgencia(int agencia){

if(agencia <= 0) {

System.out.println("Nao pode valor menor igual a 0");

return;

}

this.agencia = agencia;

}

public void setTitular(Cliente titular){

this.titular = titular;

}

public Cliente getTitular(){

return this.titular;

}

public static int getTotal(){

return Conta.total;

}

}

Nossa classe Cliente:

public class Cliente {

private String nome;

private String cpf;

private String profissao;

public String getNome() {

return nome;

}

public void setNome(String nome) {

this.nome = nome;

}

public String getCpf() {

return cpf;

}

public void setCpf(String cpf) {

this.cpf = cpf;

}

public String getProfissao() {

return profissao;

}

public void setProfissao(String profissao) {

this.profissao = profissao;

}

}

Qualquer dúvida nos pergunte no fórum!

Para saber mais: reaproveitamento entre construtores

Nesse capítulo o nosso aprendizado foi focado nos construtores. Eles são elaborados visando que os objetos tenham seus atributos inicializados na própria construção. Essa estratégia evita estados inconsistentes no nosso objeto. Veja essa classe:

public class Carro{

private int ano;

private String modelo;

private double preco;

//getters e setters omitidos

}

Como já se sabe, quando o construtor não está declarado na classe usa-se o padrão, que não recebe parâmetro algum. Logo, uma utilização da classe poderia ser como a seguir:

Carro carro = new Carro();

carro.setAno(2013);

carro.setPreco(35000.0);

Ficou faltando uma informação preciosa! Qual o modelo dele? Para evitar esse tipo de problema devemos exigir os dados que fazem sentido o Carro ter logo na criação. Algo como:

public class Carro{

private int ano;

private String modelo;

private double preco;

public Carro(int ano, String modelo, double preco){

this.ano = ano;

this.modelo = modelo;

this.preco = preco;

}

//getters e setters omitidos

}

Agora a utilização exige a presença dos 3 parâmetros definidos.

Carro carro = new Carro(2013, "Gol", 35000.0);

Tudo funciona bem! Até que um dia é pedido que o nosso sistema aceite a criação com a passagem somente do modelo e valor. Nessa situação deve-se encarar o ano como sendo 2017. Uma solução seria:

public class Carro{

private int ano;

private String modelo;

private double preco;

public Carro(int ano, String modelo, double preco){

this.ano = ano;

this.modelo = modelo;

this.preco = preco;

}

//Novo construtor AQUI!

public Carro(String modelo, double preco){

this.ano = 2017;

this.modelo = modelo;

this.preco = preco;

}

//getters e setters omitidos

}

E dessa forma pode-se construir carros com qualquer um dos dois construtores:

Carro carro = new Carro(2013, "Gol", 35000.0);

Carro outroCarro = new Carro("Civic", 95000.0);

Só que na empresa onde esse sistema está sendo codificado existe uma equipe de testes que verificou que o nosso sistema permite a criação de um Carro com datas anteriores ao primeiro automóvel que chegou ao Brasil, um Peugeot trazido por Santos Dumont em 1891. (Alura também é história!) Além de também permitir que o modelo não seja passado(null) e o preço inválido.

O desenvolvedor logo tratou de implementar essa regra em um dos construtores:

public class Carro{

private int ano;

private String modelo;

private double preco;

public Carro(int ano, String modelo, double preco){

if(ano >= 1891){

this.ano = ano;

}else{

System.out.println("O ano informado está inválido. Por isso usaremos 2017!");

this.ano = 2017;

}

if( modelo != null){

this.modelo = modelo;

}else{

System.out.println("O modelo não foi informado. Por isso usaremos Gol!");

this.modelo = "Gol";

}

if(preco > 0){

this.preco = preco;

}else{

System.out.println("O preço não é válido. Por isso usaremos 40000.0!");

this.preco = 40000.0;

}

}

//....

}

Perceba que como temos dois construtores a regra também deveria valer para o outro:

public class Carro{

private int ano;

private String modelo;

private double preco;

public Carro(int ano, String modelo, double preco){

if(ano >= 1891){

this.ano = ano;

}else{

System.out.println("O ano informado está inválido. Por isso usaremos 2017!");

this.ano = 2017;

}

if( modelo != null){

this.modelo = modelo;

}else{

System.out.println("O modelo não foi informado. Por isso usaremos Gol!");

this.modelo = "Gol";

}

if(preco > 0){

this.preco = preco;

}else{

System.out.println("O preço não é válido. Por isso usaremos 40000.0!");

this.preco = 40000.0;

}

}

//Novo construtor AQUI!

public Carro(String modelo, double preco){

this.ano = 2017;

if( modelo != null){

this.modelo = modelo;

}else{

System.out.println("O modelo não foi informado. Por isso usaremos Gol!");

this.modelo = "Gol";

}

if(preco > 0){

this.preco = preco;

}else{

System.out.println("O preço não é válido. Por isso usaremos 40000.0!");

this.preco = 40000.0;

}

//getters e setters omitidos

}

Funcionou mas o código está duplicado e nossa classe começa a cheirar mal! Códigos duplicados exigem manutenção em dobro no futuro e em grande parte das vezes um futuro nem tão distante. Seria ótimo se fosse possível reaproveitar a lógica de validação do primeiro construtor declarado não é mesmo? Reaproveitaríamos todo ele e qualquer mudança também traria o impacto direto. No Java podemos chamar a implementação de um construtor através de outro usando simplesmente this( ) com os parâmetros exigidos pelo construtor.

Observe como ficaria o segundo construtor da nossa classe:

public Carro(String modelo, double preco){

//chamando o construtor que recebe int, String e double. Nosso primeiro!

this(2017, modelo, preco);

}

Muito mais simples de manter não é mesmo? Nossa classe, Carro, ficaria portanto assim:

public class Carro{

private int ano;

private String modelo;

private double preco;

public Carro(int ano, String modelo, double preco){

if(ano >= 1891){

this.ano = ano;

}else{

System.out.println("O ano informado está inválido. Por isso usaremos 2017!");

this.ano = 2017;

}

if( modelo != null){

this.modelo = modelo;

}else{

System.out.println("O modelo não foi informado. Por isso usaremos Gol!");

this.modelo = "Gol";

}

if(preco > 0){

this.preco = preco;

}else{

System.out.println("O preço não é válido. Por isso usaremos 40000.0!");

this.preco = 40000.0;

}

}

//Novo construtor AQUI!

public Carro(String modelo, double preco){

this(2017, modelo, preco);

}

//getters e setters omitidos

}

**Conclusão**

No Java é possível fazer a chamada de um construtor dentro de outro e faz-se isso para evitar duplicações de códigos e regras. Afinal uma regra aplicada em um construtor normalmente será a mesma para o outro caso. Para isso usa-se o this( ) passando os parâmetros correspondentes ao construtor que se queira chamar.

O que aprendemos?

Aprendemos nesta aula:

Construtor da classe, possibilitando receber argumentos e inicializar atributos a partir da criação de um objeto

Com isso, a inicialização dos atributos recebidos no construtor torna-se obrigatória

Atributos da classe, os atributos estáticos

Métodos da classe, os métodos estáticos

Ausência de referência, do this, dentro de métodos estáticos

Formação Aprenda Java com Orientação a Objetos

Programação orientada a objetos e programação estruturada

Como a maioria das atividades que fazemos no dia a dia, programar também possui modos diferentes de se fazer. Esses modos são chamados de paradigmas de programação e, entre eles, estão a programação orientada a objetos (POO) e a programação estruturada.

Quando começamos a utilizar linguagens como Java, C#, Python e outras que possibilitam o paradigma orientado a objetos, é comum errarmos e aplicarmos a programação estruturada achando que estamos usando recursos da orientação a objetos.

Na programação estruturada, um programa é composto por três tipos básicos de estruturas:

Confira neste artigo:

Programação orientada a objetos e programação estruturada

O que são classes e objetos?

Encapsulamento, herança e polimorfismo: as principais características da POO

Design Patterns

Clean code e SOLID

sequências: são os comandos a serem executados

condições: sequências que só devem ser executadas se uma condição for satisfeita (exemplos: if-else, switch e comandos parecidos)

repetições: sequências que devem ser executadas repetidamente até uma condição for satisfeita (for, while, do-while etc)

Essas estruturas são usadas para processar a entrada do programa, alterando os dados até que a saída esperada seja gerada. Até aí, nada que a programação orientada a objetos não faça, também, certo?

A diferença principal é que na programação estruturada, um programa é tipicamente escrito em uma única rotina (ou função) podendo, é claro, ser quebrado em subrotinas. Mas o fluxo do programa continua o mesmo, como se pudéssemos copiar e colar o código das subrotinas diretamente nas rotinas que as chamam, de tal forma que, no final, só haja uma grande rotina que execute todo o programa.

Fluxograma de um programa estruturado. Depois de começar, o programa lê dois números A e B, verifica se A é divisível por B e, se for, imprime "A é divisvel por B", senão imprime "A no é divisível por B".

Além disso, o acesso às variáveis não possuem muitas restrições na programação estruturada. Em linguagens fortemente baseadas nesse paradigma, restringir o acesso à uma variável se limita a dizer se ela é visível ou não dentro de uma função (ou módulo, como no uso da palavra-chave static, na linguagem C), mas não se consegue dizer de forma nativa que uma variável pode ser acessada por apenas algumas rotinas do programa.

O contorno para situações como essas envolve práticas de programação danosas ao desenvolvimento do sistema, como o uso excessivo de variáveis globais. Vale lembrar que variáveis globais são usadas tipicamente para manter estados no programa, marcando em qual parte dele a execução se encontra.

A programação orientada a objetos surgiu como uma alternativa a essas características da programação estruturada. O intuito da sua criação também foi o de aproximar o manuseio das estruturas de um programa ao manuseio das coisas do mundo real, daí o nome "objeto" como uma algo genérico, que pode representar qualquer coisa tangível.

Esse novo paradigma se baseia principalmente em dois conceitos chave: classes e objetos. Todos os outros conceitos, igualmente importantes, são construídos em cima desses dois.

Banner da Imersão IA da Alura, com um ciborgue no centro da imagem com os olhos brilhando uma luz branca, enquanto no lado esquerdo está o texto "Mergulhe em Inteligência Artificial aplicada na sua carreira" e no lado direito o texto "Alura Imersão IA", com a data de realização logo abaixo: de 19 a 23 de junho de 2023.

O que são classes e objetos?

Imagine que você comprou um carro recentemente e decide modelar esse carro usando programação orientada a objetos. O seu carro tem as características que você estava procurando: um motor 2.0 híbrido, azul escuro, quatro portas, câmbio automático etc. Ele também possui comportamentos que, provavelmente, foram o motivo de sua compra, como acelerar, desacelerar, acender os faróis, buzinar e tocar música. Podemos dizer que o carro novo é um objeto, onde suas características são seus atributos (dados atrelados ao objeto) e seus comportamentos são ações ou métodos.

Seu carro é um objeto seu mas na loja onde você o comprou existiam vários outros, muito similares, com quatro rodas, volante, câmbio, retrovisores, faróis, dentre outras partes. Observe que, apesar do seu carro ser único (por exemplo, possui um registro único no Departamento de Trânsito), podem existir outros com exatamente os mesmos atributos, ou parecidos, ou mesmo totalmente diferentes, mas que ainda são considerados carros.

Podemos dizer então que seu objeto pode ser classificado (isto é, seu objeto pertence à uma classe) como um carro, e que seu carro nada mais é que uma instância dessa classe chamada "carro".

Imagem com dois retângulos lado a lado. O primeiro representa uma classe, apenas contorno do desenho de um carro; o segundo representa um objeto, com três carros desenhados.

Assim, abstraindo um pouco a analogia, uma classe é um conjunto de características e comportamentos que definem o conjunto de objetos pertencentes à essa classe. Repare que a classe em si é um conceito abstrato, como um molde, que se torna concreto e palpável através da criação de um objeto. Chamamos essa criação de instanciação da classe, como se estivéssemos usando esse molde (classe) para criar um objeto.

Exemplo em Java

public class Carro {

Double velocidade;

String modelo;

public Carro(String modelo) {

this.modelo = modelo;

this.velocidade = 0;

}

public void acelerar() {

/\* código do carro para acelerar \*/

}

public void frear() {

/\* código do carro para frear \*/

}

public void acenderFarol() {

/\* código do carro para acender o farol \*/

}

}

Exemplo em Python

class Carro:

def \_\_init\_\_(self, modelo):

self.modelo = modelo;

self.velocidade = 0

def acelerar(self):

# Codigo para acelerar o carro

def frear(self):

# Codigo para frear o carro

def acenderFarol(self):

# Codigo para acender o farol do carro

Encapsulamento, herança e polimorfismo: as principais características da POO

As duas bases da POO são os conceitos de classe e objeto. Desses conceitos, derivam alguns outros conceitos extremamente importantes ao paradigma, que não só o definem como são as soluções de alguns problemas da programação estruturada. Os conceitos em questão são o encapsulamento, a herança, as interfaces e o polimorfismo.

Encapsulamento

Ainda usando a analogia do carro, sabemos que ele possui atributos e métodos, ou seja, características e comportamentos. Os métodos do carro, como acelerar, podem usar atributos e outros métodos do carro como o tanque de gasolina e o mecanismo de injeção de combustível, respectivamente, uma vez que acelerar gasta combustível.

Desenho de um carro, representando a classe "carro", com os métodos e atributos da classe escritos dentro do desenho

No entanto, se alguns desses atributos ou métodos forem facilmente visíveis e modificáveis, como o mecanismo de aceleração do carro, isso pode dar liberdade para que alterações sejam feitas, resultando em efeitos colaterais imprevisíveis. Nessa analogia, uma pessoa pode não estar satisfeita com a aceleração do carro e modifica a forma como ela ocorre, criando efeitos colaterais que podem fazer o carro nem andar, por exemplo.

Dizemos, nesse caso, que o método de aceleração do seu carro não é visível por fora do próprio carro. Na POO, um atributo ou método que não é visível de fora do próprio objeto é chamado de "privado" e quando é visível, é chamado de "público".

Desenho de um diagrama de Venn, representando um círculo mais interno com os métodos e atributos privados, contido em outro círculo maior com os métodos e atributos públicos. O círculo mais interno é inacessível de fora e acessível aos componentes públicos do objeto, enquanto estes são acessíveis de dentro e de fora. Imagem encontrada no Google Imagens.

Mas então, como sabemos como o nosso carro acelera? É simples: não sabemos. Nós só sabemos que para acelerar, devemos pisar no acelerador e de resto o objeto sabe como executar essa ação sem expor como o faz. Dizemos que a aceleração do carro está encapsulada, pois sabemos o que ele vai fazer ao executarmos esse método, mas não sabemos como - e na verdade, não importa para o programa como o objeto o faz, só que ele o faça.

O mesmo vale para atributos. Por exemplo: não sabemos como o carro sabe qual velocidade mostrar no velocímetro ou como ele calcula sua velocidade, mas não precisamos saber como isso é feito. Só precisamos saber que ele vai nos dar a velocidade certa. Ler ou alterar um atributo encapsulado pode ser feito a partir de getters e setters (colocar referência).

Esse encapsulamento de atributos e métodos impede o chamado vazamento de escopo, onde um atributo ou método é visível por alguém que não deveria vê-lo, como outro objeto ou classe. Isso evita a confusão do uso de variáveis globais no programa, deixando mais fácil de identificar em qual estado cada variável vai estar a cada momento do programa, já que a restrição de acesso nos permite identificar quem consegue modificá-la.

Exemplo em Java

public class Carro {

private Double velocidade;

private String modelo;

private MecanismoAceleracao mecanismoAceleracao;

private String cor;

/\* Repare que o mecanismo de aceleração é inserido no carro ao ser construído, e

não o vemos nem podemos modificá-lo, isto é, não tem getter nem setter.

Já o "modelo" pode ser visto, mas não alterado. \*/

public Carro(String modelo, MecanismoAceleracao mecanismoAceleracao) {

this.modelo = modelo;

this.mecanismoAceleracao = mecanismoAceleracao;

this.velocidade = 0;

}

public void acelerar() {

this.mecanismoAceleracao.acelerar();

}

public void frear() {

/\* código do carro para frear \*/

}

public void acenderFarol() {

/\* código do carro para acender o farol \*/

}

public Double getVelocidade() {

return this.velocidade

}

private void setVelocidade() {

/\* código para alterar a velocidade do carro \*/

/\* Como só o próprio carro deve calcular a velocidade,

esse método não pode ser chamado de fora, por isso é "private" \*/

}

public String getModelo() {

return this.modelo;

}

public String getCor() {

return this.cor;

}

/\* podemos mudar a cor do carro quando quisermos \*/

public void setCor(String cor) {

this.cor = cor;

}

}

Exemplo em Python

# Exemplo da classe Carro em Python

class Carro:

def \_\_init\_\_(self, modelo, mecanismoAceleracao):

self.\_\_modelo = modelo;

self.\_\_velocidade = 0

self.\_\_mecanismoAceleracao = mecanismoAceleracao

def acelerar(self):

mecanismoAceleracao.acelerar()

def frear(self):

# Codigo para frear o carro

def acenderFarol(self):

# Codigo para acender o farol do carro

def getVelocidade(self):

return self.velocidade

def \_\_setVelocidade(self):

# Codigo para alterar a velocidade por dentro do objeto

def getModelo(self):

return self.modelo

def getCor(self):

return self.cor

def setCor(self, cor):

self.cor = cor

Herança

No nosso exemplo, você acabou de comprar um carro com os atributos que procurava. Apesar de ser único, existem carros com exatamente os mesmos atributos ou formas modificadas. Digamos que você tenha comprado o modelo Fit, da Honda. Esse modelo possui uma outra versão, chamada WR-V (ou "Honda Fit Cross Style"), que possui muitos dos atributos da versão clássica, mas com algumas diferenças bem grandes para transitar em estradas de terra: o motor é híbrido (aceita álcool e gasolina), possui um sistema de suspensão diferente, e vamos supor que além disso ele tenha um sistema de tração diferente (tração nas quatro rodas, por exemplo). Vemos então que não só alguns atributos como tambm alguns mecanismos (ou métodos, traduzindo para POO) mudam, mas essa versão "cross" ainda é do modelo Honda Fit, ou melhor, é um tipo do modelo.

Imagem representando a taxonomia de animais. Nele, uma hierarquia com a classe "Animal", que é o primeiro nível da hierarquia, com as classes "Mamífero" e "Ave" que herdam de "Animal" e, no último nível, as classes "Cachorro" e "Homem" herdando de "Mamífero" e "Beija-flor" herdando de "Ave". Imagem encontrada no Google Imagens.

Quando dizemos que uma classe A é um tipo de classe B, dizemos que a classe A herda as características da classe B e que a classe B é mãe da classe A, estabelecendo então uma relação de herança entre elas. No caso do carro, dizemos então que um Honda Fit "Cross" é um tipo de Honda Fit, e o que muda são alguns atributos (paralama reforçado, altura da suspensão etc), e um dos métodos da classe (acelerar, pois agora há tração nas quatro rodas), mas todo o resto permanece o mesmo, e o novo modelo recebe os mesmos atributos e métodos do modelo clássico.

Exemplo em Java

// "extends" estabelece a relação de herança dom a classe Carro

public class HondaFit extends Carro {

public HondaFit(MecanismoAceleracao mecanismoAceleracao) {

String modelo = "Honda Fit";

// chama o construtor da classe mãe, ou seja, da classe "Carro"

super(modelo, mecanismoAceleracao);

}

}

Exemplo em Python

# As classes dentro do parênteses são as classes mãe da classe sendo definida

class HondaFit(Carro):

def \_\_init\_\_(self, mecanismoAceleracao):

modelo = "Honda Fit"

# chama o construtor da classe mãe, ou seja, da classe "Carro"

super().\_\_init\_\_(self, modelo, mecanismoAceleracao)

Interface

Muitos dos métodos dos carros são comuns em vários automóveis. Tanto um carro quanto uma motocicleta são classes cujos objetos podem acelerar, parar, acender o farol etc, pois são coisas comuns a automóveis. Podemos dizer, então, que ambas as classes "carro" e "motocicleta" são "automóveis".

Quando duas (ou mais) classes possuem comportamentos comuns que podem ser separados em uma outra classe, dizemos que a "classe comum" é uma interface, que pode ser "herdada" pelas outras classes. Note que colocamos a interface como "classe comum", que pode ser "herdada" (com aspas), porque uma interface não é exatamente um classe, mas sim um conjunto de métodos que todas as classes que herdarem dela devem possuir (implementar) - portanto, uma interface não é "herdada" por uma classe, mas sim implementada.

No mundo do desenvolvimento de software, dizemos que uma interface é um "contrato": uma classe que implementa uma interface deve fornecer uma implementação a todos os métodos que a interface define, e em compensação, a classe implementadora pode dizer que ela é do tipo da interface. No nosso exemplo, "carro" e "motocicleta" são classes que implementam os métodos da interface "automóvel", logo podemos dizer que qualquer objeto dessas duas primeiras classes, como um Honda Fit ou uma motocicleta da Yamaha, são automóveis.

Um pequeno detalhe: uma interface não pode ser herdada por uma classe, mas sim implementada. No entanto, uma interface pode herdar de outra interface, criando uma hierarquia de interfaces. Usando um exemplo completo com carros, dizemos que a classe "Honda Fit Cross" herda da classe "Honda Fit", que por sua vez herda da classe "Carro". A classe "Carro" implementa a interface "Automóvel" que, por sua vez, pode herdar (por exemplo) uma interface chamada "MeioDeTransporte", uma vez que tanto um "automóvel" quanto uma "carroça" são meios de transporte, ainda que uma carroça não seja um automóvel.

Exemplo em Java

public interface Automovel {

void acelerar();

void frear();

void acenderFarol();

}

public class Carro implements Automovel {

/\* ... \*/

@Override

public void acelerar() {

this.mecanismoAceleracao.acelerar();

}

@Override

public void frear() {

/\* código do carro para frear \*/

}

@Override

public void acenderFarol() {

/\* código do carro para acender o farol \*/

}

/\* ... \*/

}

public class Moto implements Automovel {

/\* ... \*/

@Override

public void acelerar() {

/\* código específico da moto para acelerar \*/

}

@Override

public void frear() {

/\* código específico da moto para frear \*/

}

@Override

public void acenderFarol() {

/\* código específico da moto para acender o farol \*/

}

/\* ... \*/

}

Exemplo em Python

class Automovel():

def acelerar(self):

raise NotImplementedError()

def frear(self):

raise NotImplementedError()

def acenderFarol(self):

raise NotImplementedError()

class Carro(Automovel):

# ...

def acelerar(self):

# Codigo para acelerar o carro

def frear(self):

# Codigo para frear o carro

def acenderFarol(self):

# Codigo para acender o farol do carro

# ...

class Moto(Automovel):

# ...

def acelerar(self):

# Codigo para acelerar a moto

def frear(self):

# Codigo para frear a moto

def acenderFarol(self):

# Codigo para acender a moto

# ...

Nota: criar um erro do tipo NotImplementedError é apenas uma conveção para que, caso uma classe filha tente executar um método da classe mãe sem tê-la implementado, ocorra o erro. Em Python, as interfaces são criadas como classes normais que são herdadas pelas classes filhas. Existem formas de forçar a implementação por parte das classes filhas, mas para nosso exemplo essa abordagem é suficiente.

Polimorfismo

Vamos dizer que um dos motivos de você ter comprado um carro foi a qualidade do sistema de som dele. Mas, no seu caso, digamos que a reprodução só pode ser feita via rádio ou bluetooth, enquanto que no seu antigo carro, podia ser feita apenas via cartão SD e pendrive. Em ambos os carros está presente o método "tocar música" mas, como o sistema de som deles é diferente, a forma como o carro toca as músicas é diferente. Dizemos que o método "tocar música" é uma forma de polimorfismo, pois dois objetos, de duas classes diferentes, têm um mesmo método que é implementado de formas diferentes, ou seja, um método possui várias formas, várias implementações diferentes em classes diferentes, mas que possuem o mesmo efeito ("polimorfismo" vem do grego poli = muitas, morphos = forma).

Exemplo em Java

public class Main {

public static void main(String[] args) {

Automovel moto = new Moto("Yamaha XPTO-100", new MecanismoDeAceleracaoDeMotos())

Automovel carro = new Carro("Honda Fit", new MecanismoDeAceleracaoDeCarros())

List<Automovel> listaAutomoveis = Arrays.asList(moto, carro);

for (Automovel automovel : listaAutomoveis) {

automovel.acelerar();

automovel.acenderFarol();

}

}

}

Exemplo em Python

def main():

moto = Moto("Yahama XPTO-100", MecanismoDeAceleracaoDeMotos())

carro = Carro("Honda Fit", MecanismoDeAceleracaoDeCarros())

listaAutomoveis = [moto, carro]

for automovel in listaAutomoveis:

automovel.acelerar()

automovel.acenderFarol()

Repare que apesar de serem objetos diferentes, moto e carro possuem os mesmos métodos acelerar e acenderFarol, que são chamados da mesma forma, apesar de serem implementados de maneira diferente.

Design Patterns

Alguns problemas aparecem com tanta frequência em POO que suas soluções se tornaram padrões de design de sistemas e modelagem de código orientado a objeto, a fim de resolvê-los. Esses padrões de projeto, (ou design patterns) nada mais são do que formas padronizadas de resolver problemas comuns em linguagens orientadas a objetos. O livro "Design Patterns", conhecido como Gof:Gang of Four, é a principal referência nesse assunto, contendo os principais padrões usados em grandes projetos. A Alura também oferece cursos de Design Patterns em linguagens de programação como Java, Python e C#.

Clean code e SOLID

Em projetos desenvolvidos com POO, assim como em qualquer outro, o cdigo pode se tornar desorganizado e dificil de manter a médio e longo prazo. Em vista dessa situação, alguns princípios de boas práticas de programação e código limpo foram desenvolvidos como, por exemplo:

KISS (Keep It Simple, Stupid, "Mantenha as coisas simples"): Sempre que um código for escrito, ele deve ser escrito da forma mais simples possível, para manter o código mais legível. Códigos complexos demais são mais difíceis de se manter, j que é mais difícil entender o que ele faz e como ele faz.

DRY (Don't Repeat Yourself, "Não se repita"): Todo código escrito para resolver um problema deve ser escrito apenas uma vez, a fim de evitar repetição de código. É quase uma variação do KISS, dado que a repetição de código o torna mais confuso e difícil de manter e corrigir, se necessário.

Alguns cursos da Alura abordam esses assuntos, como o curso de SOLID com Java, SOLID com PHP, e também nosso artigo sobre o que é Clean Code.

Além dos design patterns e dos princípios de código limpo existe um conjunto de técnicas, mais generalizadas que os design patterns, que ajudam a criar código orientado a objeto de forma a deixá-lo mais maleável, possibilitando uma manutenção e expansão mais suave e descomplicada do código ao longo do tempo.

No blog da Caelum, você também pode ver sobre os grandes erros da orientação a objetos, como o uso de herança de maneira excessiva e a criação de getters e setters sem necessidade.

Nesse video eu converso com a Roberta Arcoverde sobre OO e mais tópicos importantes em Design:

# **Mãos na massa: Utilizando herança**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/128964/next)

Chegou a hora de você executar o que foi visto na aula! Para isso, execute os passos listados abaixo.

1) No Eclipse, crie um projeto Java, chamado **bytebank-herdado**.

2) Dentro deste projeto, crie a classe **Funcionario** com os atributos privados **nome** e **cpf**, do tipo **String**, e **salario**, do tipo **double**.

3) Gere os *getters* e *setters* para todos os atributos da classe **Funcionario**.

4) O funcionário pode receber uma bonificação, de 10% do salário, então crie o método **getBonificacao()**, que faça esse cálculo:

public double getBonificacao() {

return this.salario \* 0.1;

}

**COPIAR CÓDIGO**

5) Para testar a classe que você acabou de implementar, crie a classe **TesteFuncionario**, com o método **main**. Nela, instancie um **Funcionario**, dê um nome, CPF e salário a ele, e imprima-os:

public class TesteFuncionario {

public static void main(String[] args) {

Funcionario nico = new Funcionario();

nico.setNome("Nico Steppat");

nico.setCpf("223355646-9");

nico.setSalario(2590.80);

System.out.println(nico.getNome());

System.out.println(nico.getBonificacao());

}

}

**COPIAR CÓDIGO**

6) Agora, crie a classe **Gerente**, que **é um funcionário**, logo faça com que essa classe herde tudo da classe **Funcionario**.

7) Além disso, um gerente tem uma senha, então crie esse atributo, inteiro e privado, juntamente com o seu *setter*, e um método para autenticar essa senha no sistema:

public boolean autentica(int senha) {

if (this.senha == senha) {

return true;

} else {

return false;

}

}

**COPIAR CÓDIGO**

8) Por fim, para testar a classe **Gerente**, crie a classe **TesteGerente**, com o método **main**. Nela, instancie um **Gerente**, dê um nome, CPF, salário e senha a ele, e imprima-os. Além disso, teste a autenticação da senha:

public class TesteGerente {

public static void main(String[] args) {

Gerente g1 = new Gerente();

g1.setNome("Marco");

g1.setCpf("235568413");

g1.setSalario(5000.0);

System.out.println(g1.getNome());

System.out.println(g1.getCpf());

System.out.println(g1.getSalario());

g1.setSenha(2222);

boolean autenticou = g1.autentica(2222);

System.out.println(autenticou);

}

}

**COPIAR CÓDIGO**

**VER OPINIÃO DO INSTRUTOR**

### **Opinião do instrutor**

A classe **Funcionario** ficará assim:

public class Funcionario {

private String nome;

private String cpf;

private double salario;

public double getBonificacao() {

return this.salario \* 0.1;

}

public String getNome() {

return nome;

}

public void setNome(String nome) {

this.nome = nome;

}

public String getCpf() {

return cpf;

}

public void setCpf(String cpf) {

this.cpf = cpf;

}

public double getSalario() {

return salario;

}

public void setSalario(double salario) {

this.salario = salario;

}

}

**COPIAR CÓDIGO**

E a classe **Gerente** ficará assim:

public class Gerente extends Funcionario {

private int senha;

public void setSenha(int senha) {

this.senha = senha;

}

public boolean autentica(int senha) {

if (this.senha == senha) {

return true;

} else {

return false;

}

}

}

**COPIAR CÓDIGO**

Continue com os seus estudos, e se houver dúvidas, não hesite em recorrer ao nosso fórum!

Nessa aula começamos a falar sobre a herança e aprendemos:

* quais problemas a herança pode resolver
* como usar herança no Java através de palavra chave extends
* ao herdar, a classe filha ganha todas as características (atributos) e todas as funcionalidades (métodos) da classe mãe
* conhecemos o primeiro benefício da herança: *Reutilização do código*

Na próxima aula veremos mais detalhes sobre herança como as palavras chaves super, protected e a sobrescrita de métodos!

# **Visibilidade**

Em relação ao que você aprendeu até agora, qual é a ordem correta dos modificadores de visibilidade, da menor visibilidade para a maior?

* Alternativa correta
* **public < protected < private**
* Alternativa correta
* **private < public < protected**
* Alternativa correta
* **private < protected < public**
* Correto! A palavra chave com a menor visibilidade é private, depois vem o protected e depois public.
* private - apenas visível dentro da classe
* protected - visível dentro da classe e também para as filhas
* public - visível em todo lugar
* Repare também que protected é relacionado com a herança.
* Alternativa correta
* **protected < private < public**

Parabéns, você acertou!

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/128969/next)

# **Sobrescrita**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/128970/next)

Vimos que a sobrescrita é um conceito importante na herança, pois permite redefinir um comportamento previsto na classe mãe através da classe filha.

Agora veja a classe Veiculo abaixo:

class Veiculo {

public void liga() {

// alguma implementação

}

}

**COPIAR CÓDIGO**

E a classe filha Carro:

class Carro extends Veiculo {

// ????

}

**COPIAR CÓDIGO**

No que aprendemos até agora, qual dos métodos abaixo inserido no lugar de **// ????** **sobrescreve corretamente** o método liga?

* Alternativa incorreta

public void liga(int tentativas) {

// implementação

* }
* Errado, pois introduzimos um novo parâmetro (tentativas). O que fizemos aqui não é *sobrescrita* e sim *sobrecarga*.
* Falaremos mais sobre a sobrecarga na outra atividade.
* Alternativa incorreta

private void liga() {

// implementação

* }
* Alternativa incorreta

public int liga() {

// implementação

* }
* Alternativa incorreta

public void liga() {

// implementação

* }

Não foi desta vez! Tente novamente.

# **Sobrescrita**

Vimos que a sobrescrita é um conceito importante na herança, pois permite redefinir um comportamento previsto na classe mãe através da classe filha.

Agora veja a classe Veiculo abaixo:

class Veiculo {

public void liga() {

// alguma implementação

}

}

**COPIAR CÓDIGO**

E a classe filha Carro:

class Carro extends Veiculo {

// ????

}

**COPIAR CÓDIGO**

No que aprendemos até agora, qual dos métodos abaixo inserido no lugar de **// ????** **sobrescreve corretamente** o método liga?

* Alternativa correta

public void liga(int tentativas) {

// implementação

* }
* Errado, pois introduzimos um novo parâmetro (tentativas). O que fizemos aqui não é *sobrescrita* e sim *sobrecarga*.
* Falaremos mais sobre a sobrecarga na outra atividade.
* Alternativa correta

private void liga() {

// implementação

* }
* Alternativa correta

public int liga() {

// implementação

* }
* Errado, pois alteramos o tipo do retorno do método. Repare que o método na classe Veiculo tem o retorno void.
* Alternativa correta

public void liga() {

// implementação

* }
* Correto! Repare que o método possui a mesma *assinatura*. Isto é, a mesma *visibilidade*, mesmo *retorno*, mesmo *nome* e os mesmos *parâmetros*.

Parabéns, você acertou!

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/128970/next)

**Dominando herança**

**Sobre herança em Java, julgue as seguintes afirmativas:**

**1) Uma classe pode ter várias filhas, mas apenas uma mãe.**

**2) A partir de uma instância de uma classe filha, podemos chamar qualquer método público que tenha sido declarado na classe mãe.**

**3) Na classe filha, podemos escolher o que herdar da classe mãe.**

**4) No exemplo abaixo, Cachorro também herda tudo da classe Animal:**

**class Animal {**

**// atributos e métodos**

**}**

**class Mamifero extends Animal {**

**// atributos e métodos**

**}**

**class Cachorro extends Mamifero {**

**// atributos e métodos**

**class Animal {**

**// atributos e métodos**

**}**

**class Mamifero extends Animal {**

**// atributos e métodos**

**}**

**class Cachorro extends Mamifero {**

**// atributos e métodos**

**}}**

**Quais afirmativas estão corretas?**

**Alternativa incorreta**

**Apenas as afirmativas 1, 2 e 4 estão corretas.**

**Alternativa incorreta**

**Apenas as afirmativas 2, 3 e 4 estão corretas.**

**Alternativa incorreta**

**Apenas as afirmativas 1 e 2 estão corretas.**

**A afirmativa 4 também está correta, pois a classe Cachorro herda sim também de Animal, indiretamente.**

**Não foi desta vez! Tente novamente.**

**PRÓXIMA ATIVIDADE**

**Mãos na massa: Reescrita de método**

**Chegou a hora de você executar o que foi visto na aula! Para isso, execute os passos listados abaixo.**

**1) O gerente tem uma bonificação diferente, ele tem a bonificação de um funcionário comum mais um salário. Então, na classe Gerente, reescreva o método getBonificacao().**

**2) Nesse método, reaproveite o método getBonificacao() da superclasse:**

**public double getBonificacao() {**

**return super.getBonificacao();**

**}**

**3) Além disso, some um salário à bonificação atual. Para tal, acesse o método getSalario(), também pertencente à superclasse:**

**public double getBonificacao() {**

**return super.getBonificacao() + super.getSalario();**

**}**

**4) Na classe TesteGerente, imprima ao final a bonificação do gerente e veja o resultado:**

**public class TesteGerente {**

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

**Gerente g1 = new Gerente();**

**g1.setNome("Marco");**

**g1.setCpf("235568413");**

**g1.setSalario(5000.0);**

**System.out.println(g1.getNome());**

**System.out.println(g1.getCpf());**

**System.out.println(g1.getSalario());**

**g1.setSenha(2222);**

**boolean autenticou = g1.autentica(2222);**

**System.out.println(autenticou);**

**System.out.println(g1.getBonificacao());**

**}**

**}**

**VER OPINIÃO DO INSTRUTOR**

**Opinião do instrutor**

**Continue com os seus estudos, e se houver dúvidas, não hesite em recorrer ao nosso fórum!**

**PRÓXIMA ATIVIDADE**

**Para saber mais: Sobrecarga**

**Existe um outro conceito nas linguagens OO que se chama de sobrecarga que é muito mais simples do que a sobrescrita e nem dependente da herança.**

**Por exemplo, na nossa classe Gerente, imagine um outro novo método autentica que recebe além da senha também o login:**

**public class Gerente extends Funcionario {**

**private int senha;**

**public void setSenha(int senha) {**

**this.senha = senha;**

**}**

**public boolean autentica(int senha) {**

**if(this.senha == senha) {**

**return true;**

**} else {**

**return false;**

**}**

**}**

**//novo método, recebendo dois parametros**

**public boolean autentica(String login, int senha) {**

**//implementacao omitida**

**}**

**//outros métodos omitidos**

**}**

**Repare que criamos uma nova versão do método autentica. Agora temos dois métodos autentica na mesma classe que variam na quantidade ou tipos de parâmetros. Isso se chama sobrecarga de métodos.**

**A sobrecarga não leva em conta a visibilidade ou retorno do método, apenas os parâmetros e não depende da herança.**

**PRÓXIMA ATIVIDADE**

**O que aprendemos?**

**O que aprendemos?**

**Nessa aula já entramos mais a fundo na herança. Aprendemos:**

**que classe mãe é chamada de super ou base class**

**que a classe filha também é chamada de sub class**

**como aumentar a visibilidade de um membro (atributo, método) através do protected**

**como acessar ou chamar um membro (atributo, método) através do super**

**como redefinir um método através da sobrescrita**

**Na próxima aula veremos um novo benefício da herança, o Polimorfismo. Segura!**

**Polimorfismo**

**Atividade**

# **Qual é a saída?**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/128980/next)

Dada a classe Veiculo:

public class Veiculo {

public void liga() {

System.out.println("Ligando Veiculo");

}

}

**COPIAR CÓDIGO**

A classe Carro:

class Carro extends Veiculo {

public void liga() {

System.out.println("Ligando Carro");

}

}

**COPIAR CÓDIGO**

E a classe Moto:

class Moto extends Veiculo {

public void liga() {

System.out.println("Ligando Moto");

}

}

**COPIAR CÓDIGO**

Veja o código com o método main:

public class Teste {

public static void main(String[] args) {

Veiculo m = new Moto();

m.liga();

Veiculo c = new Carro();

c.liga();

}

}

**COPIAR CÓDIGO**

Ao executar, o que será impresso no console?

* Alternativa correta

Ligando Veiculo

* Ligando Veiculo
* Errado! Sempre será chamado o método mais específico. A saída correta é:

Ligando Moto

* Ligando Carro
* **COPIAR CÓDIGO**
* Alternativa correta
* O código não compila, pois o Moto não é um Veiculo.
* Alternativa correta

Ligando Veiculo

* Ligando Moto
* Alternativa correta

Ligando Moto

* Ligando Carro
* Correto! Sempre será chamado o método mais específico, primeiro o método de Moto, depois de Carro.

atividade

Veja o código abaixo, que deve estar dentro do método main:

Funcionario f = new Gerente();

f.autentica(1234);

**COPIAR CÓDIGO**

Baseado no que você aprendeu na aula, por que o código não compilou?

* Alternativa correta
* Porque a classe Gerente não possui um método autentica.
* Alternativa correta
* Porque a referência f é do tipo Funcionario e a classe Funcionario não tem o método autentica.
* Correto! Quem define o que podemos chamar é a referência, que é do tipo Funcionario, e a classe Funcionario realmente não tem esse método.
* Alternativa correta
* Porque a classe Gerente não sobrescreveu o método autentica.
* Alternativa correta
* Porque a referência f sempre precisa ser do mesmo tipo do objeto.

Parabéns, você acertou!

# **Tipo da referência**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/128982/next)

Continuando com o exemplo Veiculo, Moto e Carro:

public class Veiculo {

public void liga() {

System.out.println("Ligando Veiculo");

}

}

public class Carro extends Veiculo {

public void liga() {

System.out.println("Ligando Carro");

}

}

public class Moto extends Veiculo {

public void liga() {

System.out.println("Ligando Moto");

}

}

**COPIAR CÓDIGO**

E veja o código quase completo:

public class Teste {

public static void main(String[] args) {

???? v = new Carro();

}

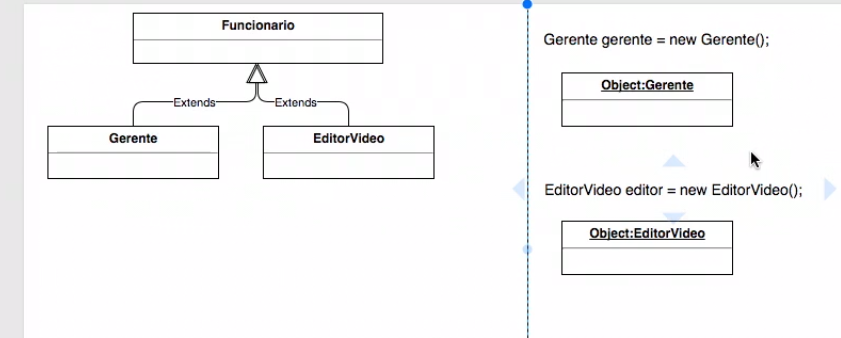
}

**COPIAR CÓDIGO**

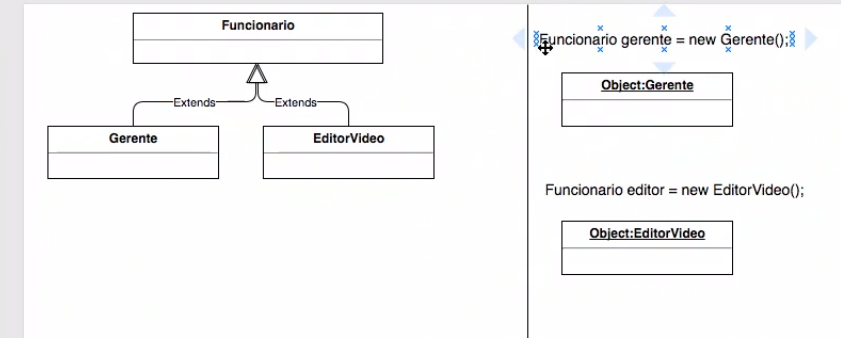
O que podemos inserir no lugar de **????** para compilar o código sem erros?

* Alternativa correta
* Veiculo
* Correto, pois o Carro é um Veiculo.
* Alternativa correta
* Moto
* Alternativa correta
* Carro
* Correto, podemos sempre usar o mesmo tipo na referência e objeto.

polimorfismo



polimorfismo



# **O que é polimorfismo?**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/128984/next)

No mundo orientado a objetos, o polimorfismo permite que ...

* Alternativa correta
* Objetos mais concretos referenciem tipos do objetos mais genéricos.
* Alternativa correta
* Referências usem outras referências mais concretas ou genéricas.
* Alternativa correta
* Referências de tipos de classes mais genéricas referenciem objetos mais específicos.
* Correto e vimos isso no código, através do exemplo:
* Funcionario f = new Gerente();

# **Mãos na massa: Polimorfismo**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/128985/next)

Chegou a hora de você executar o que foi visto na aula! Para isso, execute os passos listados abaixo.

1) Para controlar a bonificação dos funcionários, crie a classe **ControleBonificacao** com o atributo privado **soma**, do tipo **double**, e seu *getter*.

2) Nessa classe, crie o método **registra**, que recebe um **Funcionario** por parâmetro, pega a sua bonificação, e em seguida a soma com as outras bonificações. E, crie também o método **getSoma** para retornar o valor do atributo **soma**:

public void registra(Funcionario f) {

double boni = f.getBonificacao();

this.soma = this.soma + boni;

}

public double getSoma() {

return this.soma;

}

**COPIAR CÓDIGO**

3) Para testar, crie dois novos funcionários, ou seja, duas novas classes, com bonificações diferentes. Por exemplo, a classe **EditorVideo** com bonificação de R$150 e a classe **Designer**, com bonificação de R$200.

4) E crie a classe **TesteReferencias**, onde você instancia funcionários diferentes, registra suas bonificações e visualiza a sua soma:

public class TesteReferencias {

public static void main(String[] args) {

Gerente g1 = new Gerente();

g1.setNome("Marcos");

g1.setSalario(5000.0);

EditorVideo ev = new EditorVideo();

ev.setSalario(2500.0);

Designer d = new Designer();

d.setSalario(2000.0);

ControleBonificacao controle = new ControleBonificacao();

controle.registra(g1);

controle.registra(ev);

controle.registra(d);

System.out.println(controle.getSoma());

}

}

**COPIAR CÓDIGO**

**VER OPINIÃO DO INSTRUTOR**

### **Opinião do instrutor**

A classe **ControleBonificacao** ficará assim:

public class ControleBonificacao {

private double soma;

public void registra(Funcionario f) {

this.soma += f.getBonificacao();

}

public double getSoma() {

return soma;

}

}

**COPIAR CÓDIGO**

Continue com os seus estudos, e se houver dúvidas, não hesite em recorrer ao nosso fórum!

# **O que aprendemos?**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/128986/next)

Nessa aula aprendemos que:

* objetos não mudam de tipo;
* a referência pode mudar, e aí entra o polimorfismo;
* o polimorfismo permite usar referências mais genéricas para a comunicação com um objeto;
* o uso de referências mais genéricas permite desacoplar sistemas.

No próximo vídeo vamos falar sobre como se comportam os construtores na herança.

# **Herança de classes**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/128989/next)

Sobre a herança de classes, todas as afirmativas abaixo são verdadeiras, exceto:

* Alternativa correta
* Quando uma classe herda de outra, ela recebe também seus atributos.
* Alternativa correta
* Quando uma classe herda de outra, ela recebe também seus métodos.
* Alternativa correta
* Quando uma classe herda de outra, ela recebe também seus construtores automaticamente.
* Não é verdade, pois recebe apenas seus métodos e atributos. Lembra-se não tem herança de construtores

Aprendemos que a construção de um objeto é baseada em seu(s) construtor(es).

Qual das alternativas abaixo é a correta?

* Alternativa correta
* Só podemos declarar um construtor por classe, essa é uma limitação do próprio Java, que deixaremos de ter a partir do Java 8.
* Alternativa correta
* O construtor *default* do Java deixa de existir a partir do momento que algum outro é declarado na classe.
* Correto, assim que criarmos o nosso próprio construtor, o construtor *default* (sem parâmetros) deixa de existir, No entanto, nada impede adicionar o construtor *default* explicitamente.
* Alternativa correta
* Todas as classes que possuem atributos, recebem sempre esses valores via construtores.
* Alternativa correta
* Todas os objetos podem ser instanciados, sem a passagem de parâmetros no construtor. Independentemente se a classe declara algum outro.

# **A anotação @Override**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/128992/next)

Na última aula vimos sobre a anotação @Override. Qual a finalidade dela?

* Alternativa correta
* É usada para sobrescrever o método da classe mãe, indicando que o método original foi alterado.
* Alternativa correta!
* Alternativa correta
* É apenas preciosismo no código.
* Alternativa correta
* É usada para a classe atual se deserdar da classe mãe.

# **Mãos na massa: Focando em herança e polimorfismo**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/128993/next)

Chegou o momento de implementarmos o que foi visto em aula

1) Crie um novo projeto, continuaremos com o nome **bytebank-herdado-conta**.

2) Copie as duas classes (Cliente e Conta) do projeto anterior e coloque na pasta **src** do novo projeto.

3) Clique com o botão direito no projeto atual e selecione a opção **"Close unrelated projects"** para fecharmos tudo que não for relacionado ao projeto atual.

4) Crie uma nova classe, no curso utilizamos o nome de **ContaCorrente**, repare que na hora da criação, logo abaixo do campo do nome, podemos definir a **super classe**, clique em **Browse**, escreva e selecione a classe **Conta**.

5) Escreva o construtor da classe **ContaCorrente**, repassando os parâmetros para a classe mãe através do **super();**.

public ContaCorrente(int agencia, int numero){

super(agencia, numero);

}

**COPIAR CÓDIGO**

6) Crie agora a classe **ContaPoupanca**, já definindo a classe **Conta** como mãe.

7) Crie um construtor da mesma forma que foi feito na classe **ContaCorrente**:

public ContaPoupanca(int agencia, int numero){

super(agencia, numero);

}

**COPIAR CÓDIGO**

8) Crie uma classe de teste, no curso, demos o nome de **TesteContas**, essa classe deve ter o método **main**!

9) Dentro do método **main**, instancie uma conta corrente e uma poupança e então realize algumas transações, veja o código abaixo:

//Dentro do método main

ContaCorrente cc = new ContaCorrente(111, 111);

cc.deposita(100.0);

ContaPoupanca cp = new ContaPoupanca(222, 222);

cp.deposita(100.0);

**COPIAR CÓDIGO**

10) Utilize o método transfere para transferir valores de uma conta para a outra:

//Dentro do método main

ContaCorrente cc = new ContaCorrente(111, 111);

cc.deposita(100.0);

ContaPoupanca cp = new ContaPoupanca(222, 222);

cp.deposita(100.0);

cc.transfere(10.0, cp);

System.out.println("CC: " + cc.getSaldo());

System.out.println("CP: " + cp.getSaldo());

**COPIAR CÓDIGO**

11) Comente os **System.out.prinln** do construtor execute o nosso **TesteContas** e veja se as transações foram feitas com sucesso.

OBS: Na classe **Conta**, iniciamos as contas com o saldo de 100 (no construtor), comente essa linha para não confundir!

12) Na classe **ContaCorrente**, sobreescreva o método saca, para isso, escreva **saca** e utilize o atalho do eclipse **CTRL + ESPAÇO** e selecione a opção que contenha **Override method**.

13) Implemente a nova regra de negócio, o código deve ficar conforme abaixo:

@Override

public boolean saca(double valor){

double valorASacar = valor + 0.2;

return super.saca(valorASacar);

}

**COPIAR CÓDIGO**

14) Execute novamente e veja se tudo saiu conforme o planejado!

**VER OPINIÃO DO INSTRUTOR**

# **O que aprendemos?**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/128994/next)

**Nessa aula, vimos:**

* **Conceitos de herança, construtores e polimorfismo**
* **A utilização da anotação @Override**
* **Construtores não são herdados**
* **Um construtor da classe mãe pode ser chamado através do super()**

**No próximo video vamos falar como se comportam classes e métodos abstratos! Aguarde :)**

# **obre Classes Abstratas**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/128997/next)

**Qual das afirmativas abaixo é VERDADEIRA sobre classes abstratas?**

* **Alternativa correta**
* **Podem ser instanciadas normalmente.**
* **Alternativa correta**
* **Classes abstratas não podem ter métodos concretos.**
* **Alternativa correta**
* **Não podem ser instanciadas. Para instanciar, devemos criar primeiro uma classe filha não abstrata.**
* **Alternativa correta! Uma classe abstrata representa um conceito, algo abstrato, e o compilador não permite instanciar um objeto dessa classe. Para instanciar é preciso criar primeiro uma classe filha não abstrata.**

**Parabéns, você acertou!**

# **Sobre métodos abstratos**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/128999/next)

**Qual das afirmativas abaixo é verdadeira sobre os *métodos abstratos*?**

* **Alternativa correta**
* **Não possuem corpo (implementação), apenas definem a assinatura.**
* **Correto, um método abstrato define apenas a assinatura (visibilidade, retorno, nome do método e parâmetros).**
* **Alternativa correta**
* **Métodos abstratos podem ou não ter uma implementação. Caso não tenham implementação, a classe filha precisa implementar.**
* **Alternativa correta**
* **Podemos adicionar um método abstrato em qualquer classe, sem restrição.**

# **Classes e Métodos Abstratos**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/129000/next)

**Sobre classes e métodos abstratos, das afirmativas abaixo, qual delas é FALSA?**

* **Alternativa correta**
* **Classes abstratas são úteis quando queremos utilizar comportamentos e atributos base em classes com comportamentos em comum.**
* **Alternativa correta**
* **Classes e métodos abstratos consomem menos memória e por conta disso melhoram o desempenho do nosso programa.**
* **Essa afirmação realmente é errada. Classes e métodos abstratos não tem relação direta com consumo de memória.**
* **Alternativa correta**
* **Usamos métodos abstratos quando queremos "forçar" que um filho concreto (classe concreta) implemente um método.**

**Parabéns, você acertou!**

# **Mãos na massa: Classe e método abstrato**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/129001/next)

**1) Volte ao projeto bytebank-herdado das últimas aulas, onde foi trabalhado a questão de funcionários**

**2) Transforme a classe Funcionario em abstrata, utilizando a palavra reservada abstract, veja abaixo:**

**public abstract class Funcionario{**

**//conteúdo da classe**

**}**

**COPIAR CÓDIGO**

**3) Repare que agora não conseguimos mais criar um objeto do tipo Funcionário, que agora é abstrato! Corrija o código onde for preciso e crie um Gerente ou outro funcionário concreto.**

**4) Na classe Funcionario declare o método getBonificacao como abstrato para garantir que ele seja implementado pelas classes filhas. O código deve ficar assim:**

**//método sem corpo, não há implementação**

**public abstract double getBonificacao();**

**COPIAR CÓDIGO**

**5) Agora cada classe filha deve implementar seu próprio método getBonificacao. Veja se tudo está compilando e teste seu código!**

**6) Rode suas classes de teste, fique atento ao erros de compilação.**

**O que é verdade sobre classes abstratas? Selecione todas as afirmações verdadeiras:**

* **Alternativa correta**
* **Podem ter métodos concretos (com implementação)**
* **Correto, como podem ter atributos também podem ter métodos concretos!**
* **Alternativa correta**
* **Não podem ser instanciadas**
* **Correto, pois o que é abstrato (a classe) não pode se tornar concreto (objeto). Por isso não podemos instanciar objetos de uma classe abstrata.**
* **Alternativa correta**
* **Podem ter atributos**
* **Correto, podemos sim ter atributos! Uma classes abstrata é uma classe normal, só *não pode instanciar* e pode ter *métodos abstratos*. O resto continua valendo!**
* **Alternativa correta**
* **Podem ter métodos abstratos (sem implementação)**
* **Correto, como vimos uma classe abstrata pode ter métodos sem implementação. Dessa forma obrigamos um filho implementar o método.**
* **Alternativa correta**
* **Não podem ter construtores**

# **Mãos na massa: Contas**

* [**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/129004/next)
* **1) Antes de mexer com o projeto de *contas*, feche o projeto sobre Funcionarios. Depois abra o projeto de contas.**
* **2) Transforme a classe Conta em abstrata, já que não devemos poder instanciar apenas uma conta, mas sim uma conta corrente ou poupança. Fique atento aos erros de compilação e corrija-os.**
* **3) Transforme o método deposita em abstrato, veja o código abaixo:**
* **public abstract void deposita(double valor);**
* **COPIAR CÓDIGO**
* **4) Repare que nossas outras classes deixaram de compilar. Isso acontece devido ao fato de que agora essas classes filhas devem implementar o método deposita.**
* **Implemente o método deposita nas classes filhas. Lembre-se de antes transformar o atributo saldo da classe Conta para protected, permitindo que as classes filhas o vejam.**
* **O método deve ficar assim:**
* **@Override**
* **public void deposita(double valor){**
* **super.saldo += valor;**
* **}**
* **COPIAR CÓDIGO**
* **5) Rode o teste e veja se tudo continua funcionando normalmente!**

**Nessa aula aprendemos:**

* **O que são classes abstratas**
* **Para que servem classes abstratas**
* **O que são métodos abstratos**
* **Para que servem métodos abstratos**

**Na próxima aula veremos sobre o uso de Interfaces!**

# **Herdando de várias classes**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/129010/next)

**Vimos na última aula que não existe herança múltipla em Java. Como podemos contornar a falta disso?**

* **Alternativa correta**
* **Uma classe pode sim herdar de múltiplas outras classes.**
* **Alternativa correta**
* **Uma classe pode sim herdar de múltiplas outras classes.**
* **Alternativa correta**
* **Podemos contornar esta situação com o uso de interfaces.**
* **Correto! Utilizando interfaces temos uma outra forma de conseguir polimorfismo sem herança.**

# **Conceitos de Interfaces**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/129012/next)

**Sobre interfaces, qual das alternativas abaixo é VERDADEIRA?**

* **Alternativa correta**
* **Ela é um contrato onde quem assina se responsabiliza por implementar esses métodos (cumprir o contrato)**
* **Alternativa Correta!**
* **Alternativa correta**
* **Uma classe só pode implementar uma interface.**
* **Alternativa correta**
* **Interfaces podem conter atributos dentro de seu corpo.**

# **07Classes Abstratas x Interfaces**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/129013/next)

Sobre classes abstratas e interfaces, selecione todas as afirmativas verdadeiras:

* Alternativa correta
* Podemos estender apenas uma classe abstrata, mas podemos implementar várias interfaces.
* Correto! Existe apenas herança simples em Java, mas podemos implementar quantas interfaces que quisermos..
* Alternativa correta
* Todos os métodos de uma interface são abstratos, os de uma classe abstrata podem não ser.
* Correto, todos os métodos na interface são sempre abstratos e sempre públicos.
* Em uma classe abstratos podemos ter métodos concretos e abstratos.
* Alternativa correta
* Classes abstratas não podem ser instanciadas, interfaces sim.
* Errado! Na verdade nenhuma das duas podem ser implementadas.
* Não podemos dar new na classe abstrata, nem na interface!!

# **Sobre o polimorfismo**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/129014/next)

Quanto ao conceito do Polimorfismo marque as alternativas corretas:

* Alternativa correta
* É a capacidade de um objeto ser referenciado por vários tipos.
* Correta, podemos comunicar com um objeto através de tipos de variáveis diferentes.
* Por exemplo, se existir uma classe Gerente que seja filha de Funcionario, um objeto do tipo Gerente pode ser referenciado com o tipo Funcionario também.
* Alternativa correta
* É a capacidade de um objeto chamar métodos do seu pai usando super.
* Errado, através super deixamos claro que queremos acessar um membro da classe mãe.
* Alternativa correta
* Só temos polimorfismo quando uma classe extende de outra, ou seja, apenas via herança.
* Errado, vimos que as interfaces são uma alternativa também para ter polimorfismo.
* Alternativa correta
* Temos polimorfismo quando uma classe extende de outra ou também quando uma classe implementa uma interface.
* Correta, temos polimorfismo via herança ou interface.

# **Mãos na massa: Trabalhando com interfaces**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/129015/next)

1) Se não tiver aberto ainda, abra o projeto sobre **Funcionario**.

2) Crie uma classe chamada **SistemaInterno** que tenha como atributo uma senha e implemente o método autentica conforme abaixo:

public class SistemaInterno{

private int senha = 2222;

public void autentica(Gerente g){

boolean autenticou = g.autentica(this.senha);

if(autenticou){

System.out.println("Pode entrar no sistema");

} else {

System.out.println("Não pode entrar no sistema");

}

}

}

**COPIAR CÓDIGO**

3) Crie uma classe de testes chamada **TesteSistema** com o método **main** já gerado.

4) Instancie um gerente, utilize o método **setSenha** e coloque a senha correta, além disso, instancie o **SistemaInterno** e tente autenticar.

O código abaixo deve estar dentro do método **main**:

Gerente g = new Gerente();

g.setSenha(2222);

SistemaInterno si = new SistemaInterno();

si.autentica(g);

**COPIAR CÓDIGO**

5) Crie uma classe **Administrador** que deve herdar da classe **Funcionario**. Perceba que o Eclipse já nos entrega inclusive o método **getBonificacao**

6) Crie uma classe abstrata **FuncionarioAutenticavel** que herda de **Funcionario**

7) Complete o código de **FuncionarioAutenticavel** com o código abaixo:

public abstract class FuncionarioAutenticavel extends Funcionario{

private int senha;

public void setSenha(int senha){

this.senha = senha;

}

public boolean autentica(int senha){

if(this.senha == senha){

return true;

} else {

return false;

}

}

}

**COPIAR CÓDIGO**

8) Modifique o código de **Gerente** fazendo com que herde de **FuncionarioAutenticavel**

public class Gerente extends FuncionarioAutenticavel{

//resto do código

}

**COPIAR CÓDIGO**

9)Em **SistemaInterno** troque a referência do método **autentica** para que receba um **FuncionarioAutenticavel**, veja abaixo:

public class SistemaInterno{

private int senha = 2222;

public void autentica(FuncionarioAutenticavel fa){

boolean autenticou = fa .autentica(this.senha);

if(autenticou){

System.out.println("Pode entrar no sistema");

} else {

System.out.println("Não pode entrar no sistema");

}

}

}

**COPIAR CÓDIGO**

10) Perceba que se precisarmos que um cliente seja autenticavel, teremos um problema, já que a classe **Cliente** precisaria herdar de **FuncionarioAutenticavel** e isso acarretaria em um cliente ter bonificação já que a classe **FuncionarioAutenticavel** herda de **Funcionario**

11) Para resolver isso, faremos uso das **Interfaces**, transformando a antiga classe abstrata **FuncionarioAutenticavel** na interface **Autenticavel** como abaixo:

public abstract interface Autenticavel{

public abstract void setSenha(int senha);

public abstract boolean autentica(int senha);

}

**COPIAR CÓDIGO**

12) Na classe **Cliente** utilize a palavra reservada **implements** para utilizar a interface criada, veja abaixo:

public class Cliente implements Autenticavel{

}

**COPIAR CÓDIGO**

13) Complete o código da classe **Cliente**, implementando os métodos que faltam:

public class Cliente implements Autenticavel{

private int senha;

@Override

public void setSenha(int senha){

this.senha = senha;

}

@Override

public boolean autentica(int senha){

if(this.senha == senha){

return true;

} else {

return false;

}

}

}

**COPIAR CÓDIGO**

14) Faça com que as classes **Gerente** e **Administrador** implementem a interface **Autenticavel**, não se esqueça de implementar os métodos e declarar o atributo privado senha!

public class Gerente extends Funcionario implements Autenticavel {

private int senha;

public double getBonificacao(){

// implementação da bonificação do gerente omitida

}

@Override

public void setSenha(int senha){

this.senha = senha;

}

@Override

public boolean autentica(int senha){

if(this.senha == senha){

return true;

} else {

return false;

}

}

}

**COPIAR CÓDIGO**

public class Administrador extends Funcionario implements Autenticavel {

private int senha;

public double getBonificacao(){

// implementação da bonificação do administrador omitida

}

@Override

public void setSenha(int senha){

this.senha = senha;

}

@Override

public boolean autentica(int senha){

if(this.senha == senha){

return true;

} else {

return false;

}

}

}

**COPIAR CÓDIGO**

15) No **SistemaInterno** modifique o método autentica para que agora receba um **Autenticavel**

public class SistemaInterno{

private int senha = 2222;

public void autentica(Autenticavel fa){

boolean autenticou = fa .autentica(this.senha);

if(autenticou){

System.out.println("Pode entrar no sistema");

} else {

System.out.println("Não pode entrar no sistema");

}

}

}

**COPIAR CÓDIGO**

16) Repare que agora podemos usar **Autenticavel** como um tipo, veja o exemplo abaixo:

Autenticavel referencia = new Gerente();

Autenticavel referencia = new Cliente();

Autenticavel referencia = new Administrador();

**COPIAR CÓDIGO**

17) Teste tudo e veja se está tudo ok!

Nessa aula aprendemos que:

* Não existe herança múltipla em Java.
* Conceitos de interface.
* Diferenças entre classes abstratas e interfaces.
* interfaces são uma alternativa a herança referente ao polimorfismo

No próximo capítulo praticaremos mais um pouco sobre herança e interfaces!

# **Revisando conceitos de herança**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/129021/next)

Como vimos durante o curso e revisamos durante este capítulo, quais das afirmativas abaixo descreve uma vantagem do uso de herança?

* Alternativa correta
* A herança possui baixo acoplamento, logo é fácil mudar uma classe mãe sem causar problemas nas classes filhas.
* Errado, e justamente contrário. Há um acoplamento forte entre mãe e filho, por isso é preciso ter muito cautela na hora de usar a herança.
* Alternativa correta
* A herança captura o que é comum e isola aquilo que é diferente entre classes.
* Correto!
* Alternativa correta
* A herança apenas serve para definir um contrato entre classes.
* Errado! Na verdade a "criação de contratos" é feita pelas interfaces.

Parabéns, você acertou!

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/129021/next)

# **Revisando conceitos de interface**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/129022/next)

Como vimos durante o curso e revisamos durante este capítulo, quais das afirmativas abaixo descreve uma vantagem do uso de interfaces?

* Alternativa correta
* Ao serem estendidas, geram um contrato entre a interface e a classe que chamou.
* Errado. Na verdade interfaces são implementadas, não estendidas.
* Alternativa correta
* Permite atributos e com isso melhora a legibilidade do código.
* Alternativa correta
* Garante que todos os métodos de classes que implementam uma interface possam ser chamados com segurança.
* Correto! Isso é a ideia do contrato, garantindo que a classe tenha um comportamento, basta assinar o contrato (implementar a inteface).

# **Composição x Herança**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/129023/next)

Qual das afirmativas a seguir representa uma vantagem do uso de composição e interfaces sobre o uso de herança?

* Alternativa correta
* A herança é limitada ao polimorfismo, enquanto o uso de interfaces com composição possibilita polimorfismo E reutilização de código.
* Errado pois Herança possibilita sim polimorfismo E reutilização.
* Com interfaces temos uma alternativa ao polimorfismo, com a composição temos uma alternativa a reutilização de código.
* Alternativa correta
* Não há vantagem alguma pode escolher o que você julga melhor.
* Errado! Na verdade existem algumas vantagens. Normalmente o uso de interfaces (para definir o contrato e ter polimorfismo) e composição (para ter reutilização de código) é mais flexível comparado a herança.
* Alternativa correta
* Com composições e interfaces teremos mais flexibilidade com nosso código, já que não nos prenderemos ao acoplamento que a herança propõe.
* Correto!

Parabéns, você acertou!

# **Mãos na massa: Continuando o projeto**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/129024/next)

**1) Repare que ainda estamos repetindo código, o que não é uma boa prática.**

**2) Crie uma nova classe AutenticacaoUtil e coloque o código abaixo:**

**public class AutenticacaoUtil{**

**private int senha;**

**public void setSenha(int senha){**

**this.senha = senha;**

**}**

**public boolean autentica(int senha){**

**if(this.senha == senha){**

**return true;**

**} else {**

**return false;**

**}**

**}**

**}**

**COPIAR CÓDIGO**

**3) No código da classe Cliente, modifique o código conforme abaixo:**

**public class Cliente implements Autenticavel{**

**private AutenticacaoUtil autenticador;**

**public Cliente(){**

**this.autenticador = new AutenticacaoUtil();**

**}**

**@Override**

**public void setSenha(int senha){**

**this.autenticador.setSenha(senha);**

**}**

**@Override**

**public boolean autentica(int senha){**

**return this.autenticador.autentica(senha);**

**}**

**}**

**COPIAR CÓDIGO**

**4) Faça as mesmas modificações na classe Administrador e Gerente, não se esqueça de criar o construtor!**

**5) Teste tudo e veja se está tudo ok!**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/129023/next)

# **Mãos na massa (Opcional): Modificando o projeto de Conta**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-polimorfismo-heranca-interfaces/task/129025/next)

1) Volte para nosso projeto de conta.

2) Crie uma interface com o nome de **Tributavel** veja o código abaixo:

public interface Tributavel{

double getValorImposto();

}

**COPIAR CÓDIGO**

3) Crie agora uma classe chamada **CalculadorDeImposto**, veja o conteúdo abaixo:

public class CalculadorDeImposto{

private double totalImposto;

public void registra(Tributavel t){

double valor = t.getValorImposto();

this.totalImposto += valor;

}

public double getTotalImposto(){

return totalImposto;

}

}

**COPIAR CÓDIGO**

4) Crie uma classe chamada **SeguroDeVida** e defina a interface **Tributavel**, defina também a **ContaCorrente** como **Tributavel** e complete o código, veja abaixo:

**ContaCorrente**

public class ContaCorrente implements Tributavel{

@Override

public double getValorImposto(){

return super.saldo \* 0.01;

}

}

**COPIAR CÓDIGO**

**SeguroDeVida** OBS: Repare que ainda falta implementar o método corretamente:

public class SeguroDeVida implements Tributavel{

@Override

public double getValorImposto(){

return 0;

}

}

**COPIAR CÓDIGO**

5) Crie uma classe **TesteTributaveis** com o método **main** veja o código abaixo:

public class TesteTributaveis{

public static void main(String[] args){

ContaCorrente cc = new ContaCorrente(222, 333);

cc.deposita(100.0);

SeguroDeVida seguro = new SeguroDeVida();

CalculadorDeImposto calc = new CalculadorDeImposto();

calc.registra(cc);

calc.registra(seguro);

System.out.println(calc.getTotalImposto());

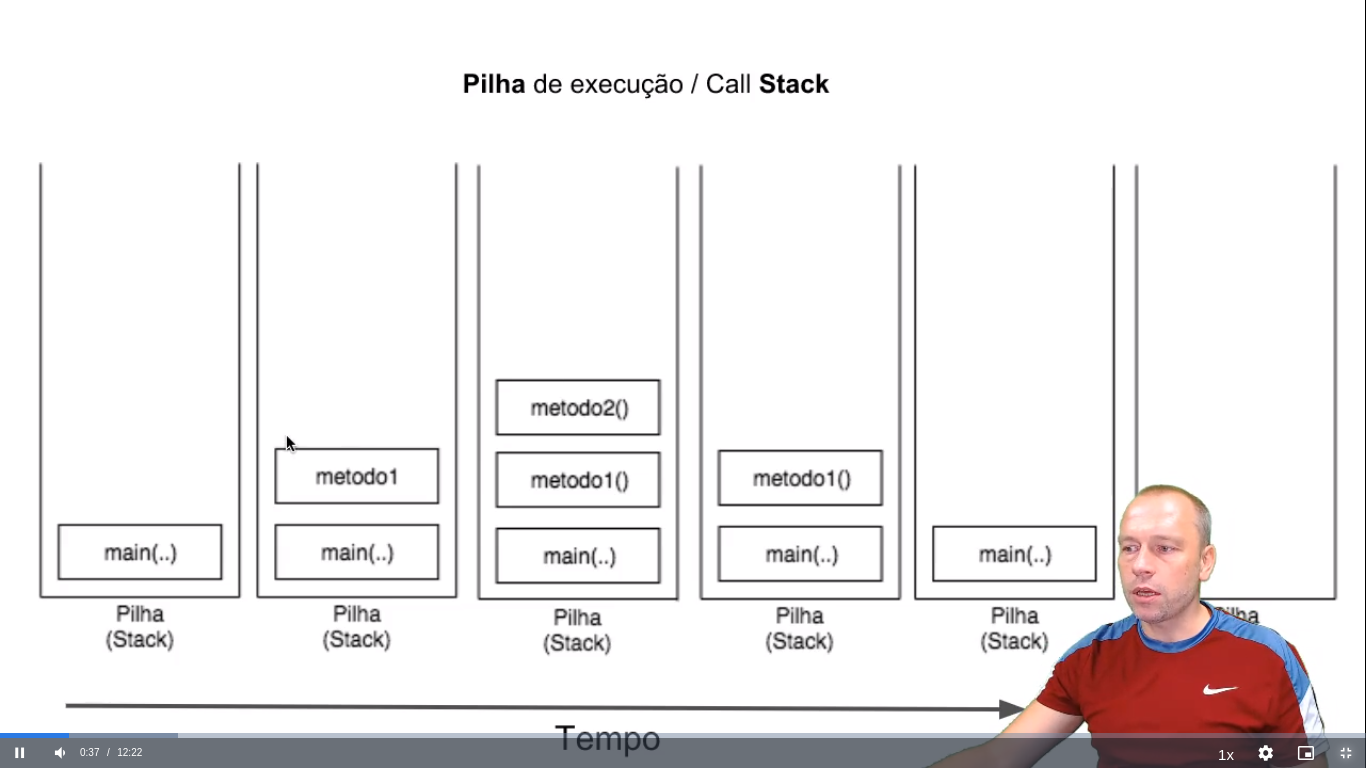
}

}

**COPIAR CÓDIGO**

6) Execute e veja se está tudo ok!

# **3Saída da stack (PILHA)**



public class Principal {

public static void main(String[] args) {

System.out.println("MAIN BEGIN");

m1();

System.out.println("MAIN END");

}

public static void m1() {

System.out.println("B BEGIN");

m2();

System.out.println("B END");

}

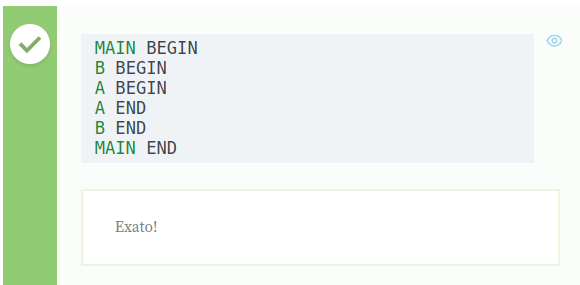
public static void m2() {

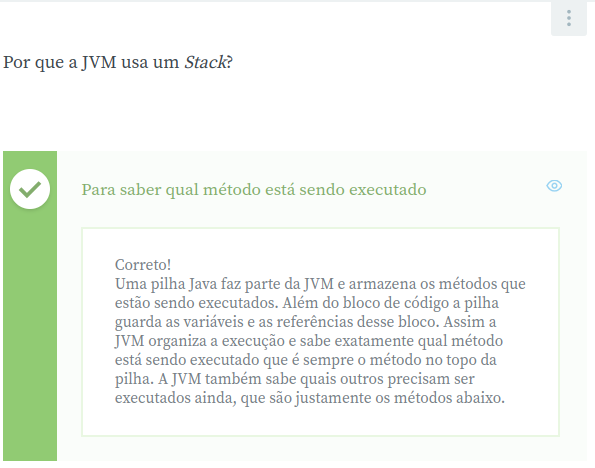
System.out.println("A BEGIN");

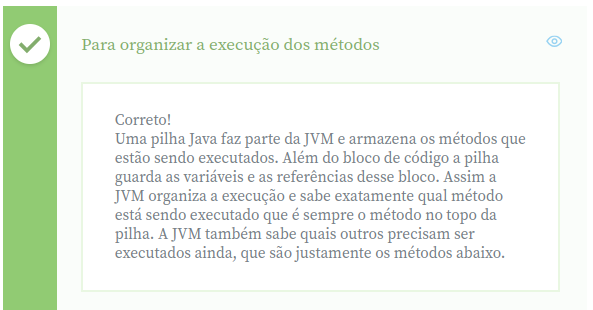
System.out.println("A END");

}

}







Debug

Temos o seguinte código sendo executado através do debugger do Eclipse.

package alura;

public class Main {

public static void main(String[] args) {

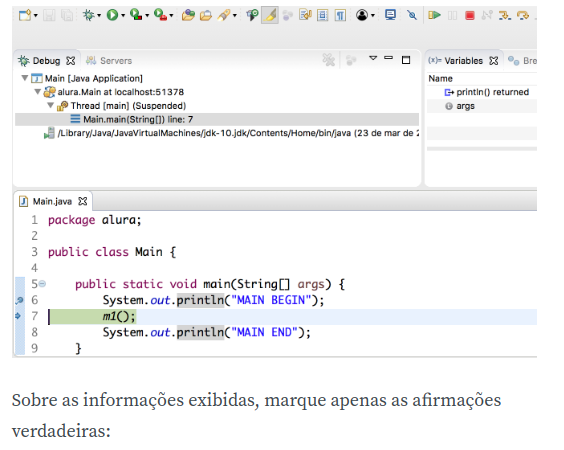
System.out.println("MAIN BEGIN");

m1();

System.out.println("MAIN END");

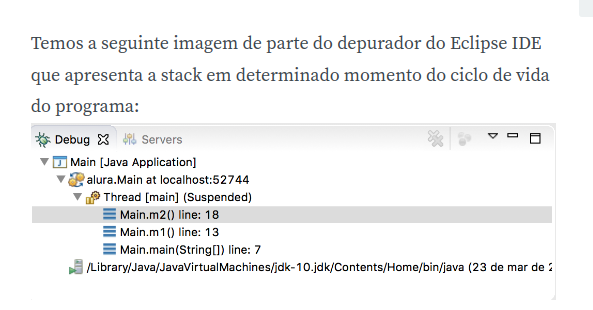
}

}



Sobre as informações exibidas, marque apenas as afirmações verdadeiras:

* Alternativa correta
* Nosso código esta parado na linha 7.
* Exato. Através do debugger podemos saber em qual linha do nosso programa estamos.
* Alternativa correta
* Se clicarmos no ícone do *Step Into (F5)* entraremos dentro do método m1 para que possamos acompanhar sua execução.
* Exato! Isso permite depurar o método também.
* Alternativa correta
* A instrução System.out.println("MAIN END"); já foi executada.
* Alternativa correta
* Se clicarmos no ícone do *Step Over (F6)* entraremos dentro do método m1 para que possamos acompanhar sua execução.



Marque a alternativa verdadeira sobre ele:

* Alternativa correta
* Temos dois métodos na pilha da execução.
* Alternativa correta
* O depurador esta parando no método m2, na linha 18.
* Exato.
* Alternativa correta
* O método m2 é o primeiro a ser chamado na pilha.

MÃO NA MASSA

Vamos começar nosso projeto!

1) Comece criando o projeto no eclipse chamado java-pilha.

2) Em seguida, crie a classe Fluxo e copie o seguinte código para a classe:

public class Fluxo {

public static void main(String[] args) {

System.out.println("Ini do main");

metodo1();

System.out.println("Fim do main");

}

private static void metodo1() {

System.out.println("Ini do metodo1");

metodo2();

System.out.println("Fim do metodo1");

}

private static void metodo2() {

System.out.println("Ini do metodo2");

for(int i = 1; i <= 5; i++) {

System.out.println(i);

}

System.out.println("Fim do metodo2");

}

}

3) Agora, analisaremos a pilha de execução. Como apresentado no vídeo, crie um ponto de parada (*Breakpoint*), por exemplo na linha que chama o metodo1(). Execute o programa no modo de depuração e use os comandos *Step into* e *Step Over*. Atenção à pilha de execução.

Nessa aula, aprendemos:

* O que é, para que serve e como funciona a pilha de execução.
* O que é depuração (debug) e para que serve.
* Como utilizar o Eclipse e sua perspectiva de debug.
* Como alternar entre perspectivas do Eclipse.