POO Java Alura

**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();

}

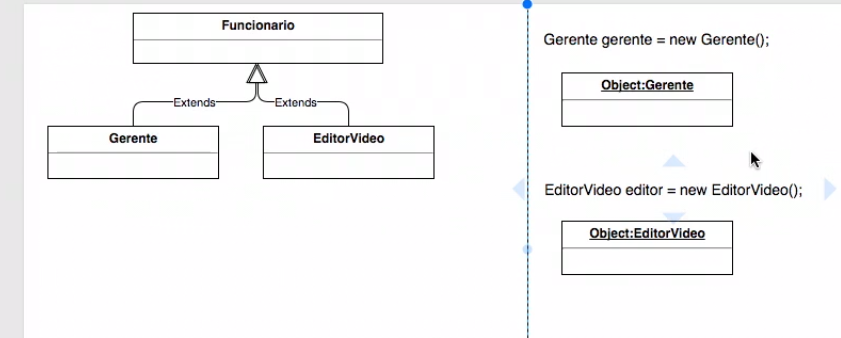
}

**COPIAR CÓDIGO**

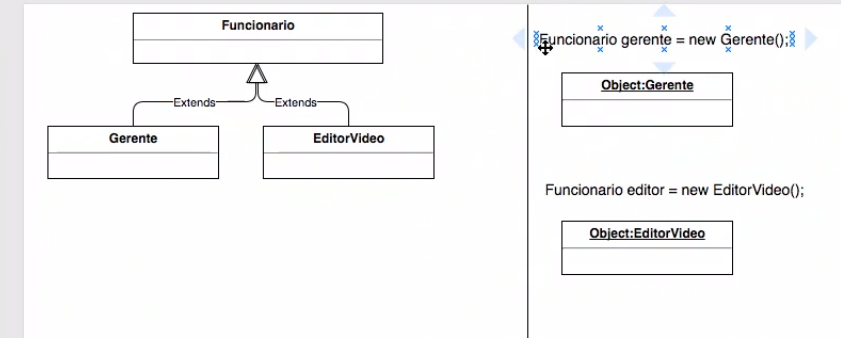
O que podemos inserir no lugar de **????** para compilar o código sem erros?

* 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());

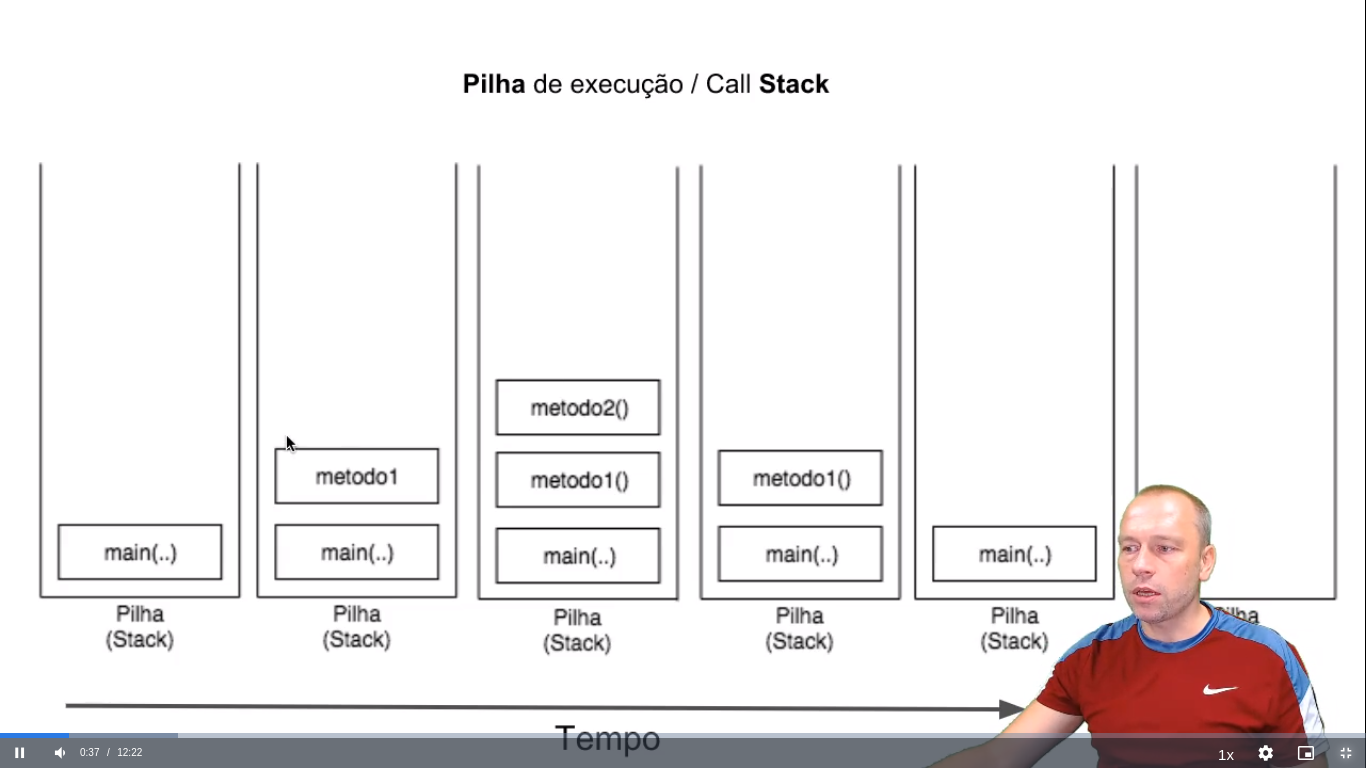
}

}

**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");

}

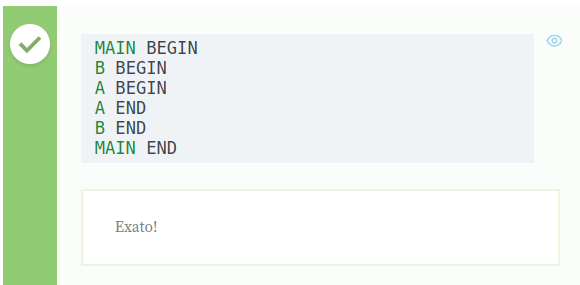
public static void m2() {

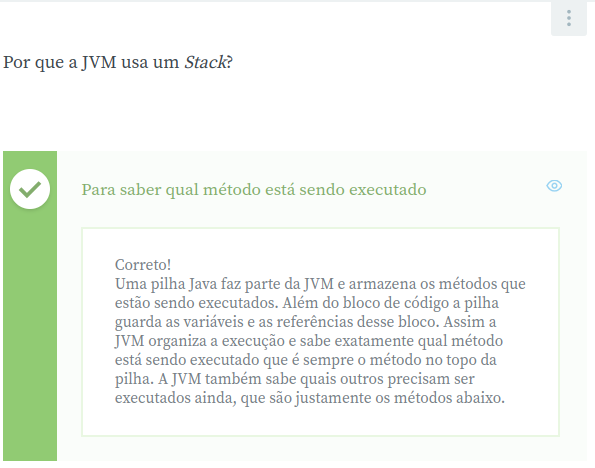
System.out.println("A BEGIN");

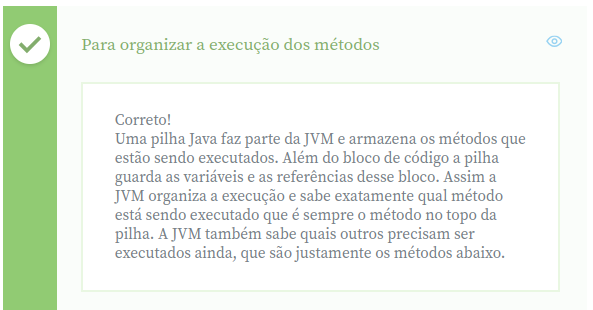
System.out.println("A END");

}

}







Debug

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

package alura;

public class Main {

public static void main(String[] args) {

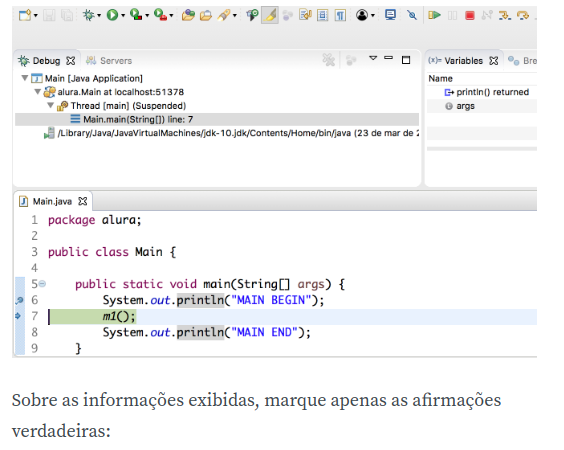
System.out.println("MAIN BEGIN");

m1();

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

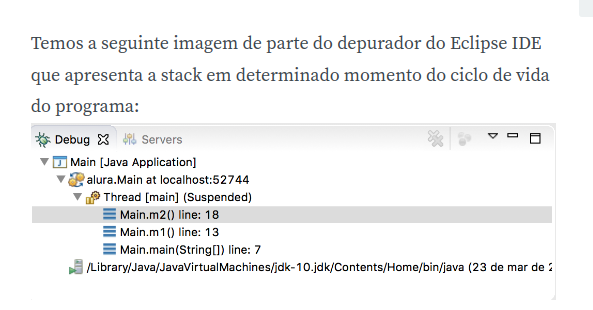
}

}



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.