**01Construtores**

**Transcrição**

Chegamos ao ponto em que temos uma conta bem encapsulada, de forma que não é mais possível inserir um saldo negativo, pois ele está velado pelos métodos de manipulação da conta, como saca(), transfere() e deposita().

Caso encontremos algum *bug*, como a inserção de um depósito negativo, basta adicionar um if no método deposita(). As soluções para possíveis problemas do nosso código estão bem localizadas, ou seja, alterado o código em um único ponto podemos realizar a manutenção do nosso sistema.

O próximo ponto que iremos nos voltar é se as nossas contas conseguem ser criadas de uma maneira que sempre possuam dados consistentes.

Quando pensamos em objetos consistentes, queremos dizer que seus atributos funcionam de acordo com as regras de negócios estipuladas por uma empresa, chefe ou algo do gênero.

Para testarmos a consistência dos objetos do nosso banco, criaremos mais uma classe chamada TestaValores. Sabemos que não é possível criar uma conta e deixá-la com um valor negativo, pois um saque não seria permitido neste caso e nem outros métodos de alteração de saldo seriam eficientes neste sentido. Entretanto podemos criar uma conta e atribuir ao número de agencia valores negativos.

public class TestaValores {

public static void main(String[] args) {

Conta conta = new Conta();

conta.setAgencia(-50);

conta.setNumero(-330);

}

} Copiar código

Na classe Conta, precisaremos adicionar ifs nos métodos setAgencia e setNumero, afirmando que caso seja postulado um valor menor ou igual a zero, ocorrerá uma mensagem de erro e a execução será interrompida.

public class Conta {

// atributos

// método deposita

// método saca

// método tranfere

// método pegaSaldo

public int getNumero() {...}

public void setNumero(int numero) {

if (numero <= 0) {

System.out.println("não pode valor <= 0");

return;

}

this.numero = numero;

}

public int getAgencia() {...}

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) {...}

public Cliente getTitular() {...}

}Copiar código

Com isso, ao tentarmos executar o programa da classe TestaValores, que está com números negativos para o atributo agencia, será impressa a mensagem nao pode valor menor igual a 0.

Porém, não estamos totalmente protegidos desses valores negativos. Percebam o problema caso escrevamos a seguinte linha de código:

public class TestaValores {

public static void main(String[] args) {

conta.setAgencia(-50);

conta.setNumero(-330);

System.out.println(conta.getAgencia());

}

} Copiar código

Veremos como resultado da aplicação o valor impresso 0. Portanto, não solucionamos o problema da numeração, porque estamos trabalhando com o valor *default*, lembrem-se que no momento em que acionamos o new em um objeto, os atributos são zerados, e os atributos de tipo referência recebem valor null.

Muitas vezes encontramos objetos que nascem em um estado inconsistente com relação à regra de negócio. Existe uma forma de restringirmos dados: toda a vez que criamos um objeto somos obrigados a passar informações específicas, fazemos isso através de um **construtor**.

Os parênteses () fechados ao lado do objeto Conta estão invocando um construtor.

public class TestaValores {

public static void main(String[] args) {

Conta conta = new Conta();

// ...

Copiar código

O construtor é um trecho de código caracterizado pela seguinte conformação:

public Conta() {

} Copiar código

Não escrevemos esse código ao longo do curso. O Java automaticamente insere esse código, é o que chamamos de **construtor padrão**.

Iremos escrever esse código em nossa classe Conta e acionaremos o sysout.

public class Conta {

private double saldo;

private int agencia

private int numero

private Cliente titular;

public Conta() {

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

// ...

} Copiar código

Feito isso, executaremos a aplicação da classe TestaValores. Lembrando que o construtor está executando a terceira linha do código da classe.

public class TestaValores {

public static void main(String[] args) {

Conta conta = new Conta();

// ...Copiar código

Veremos que o resultado impresso será estou criando uma conta.

Os parênteses estão passando um construtor. Não se trata de um método, o construtor não possui um retorno void, double, ele é uma rotina de inicialização.

O construtor é executado apenas uma vez no momento em que construímos um objeto. Não há como executar duas vezes o construtor para um mesmo objeto.

O construtor nos oferece a possibilidade de inicializar alguns dados, como por exemplo, podemos estipular que o saldo inicial de uma conta vale 100 reais. Mas o mais interessante é que o construtor pode receber parâmetros.

Para que o construtor seja invocado na abertura de uma nova conta, é necessário obrigatoriamente que seja passada a agência dessa conta e seu número. Feito isso, os atributos podem ser populados. Na execução iremos exibir o número da conta, assim conseguimos ver a atuação do construtor.

public class Conta {

private double saldo;

private int agencia

private int numero

private Cliente titular;

public Conta( int agencia, int numero) {

this.agencia = agencia;

this.numero = numero;

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

}

// ...

} Copiar código

O construtor padrão que o Java estipula caso não tenhamos escrito nenhum outro construtor, deixa de existir.

Através do construtor podemos definir restrições e exigir informações específicas do objeto. O que será exigido pelo construtor varia de acordo com as regras de negócio, no caso do nosso banco é interessante que o saldo inicie com zero, por exemplo.

Definimos alguns parâmetros para o construtor, então, para cada objeto criado, precisaremos primeiramente comunicar a agencia (1337) e o numero da conta (24226).

public class TestaValores {

public static void main(String[] args) {

Conta conta = new Conta(1337, 24226);

// ...

Copiar código

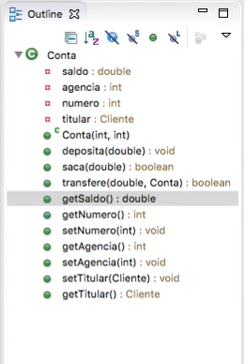
Ao executarmos a aplicação, o resultado impresso será estou criando uma conta 24226.

Podemos adicionar ifs no nosso código para gerar as condições ideais de criação do objeto de acordo com as regras de negócio estipuladas. Por exemplo, que o número de agencia deve ser maior ou igual a zero.

Com a presença do construtor e os parâmetros que a ele foram passados, não é mais necessário que haja no nosso código os métodos setAgencia() e setNumero(). Se for estipulado como regra de negócio que uma conta sempre terá o mesmo número e a mesma agência, não há necessidade de evoca métodos para alterá-la.

Trata-se de um **atributo imutável**, e há vários casos em que essa é uma opção interessante.

Uma dica de navegação no Eclipse: a nossa classe Conta está grande. Para facilitar o acesso ao conteúdo da classe, no cabeçalho do Eclipse, selecionaremos a opção "Window > Show View > Outline" (atalho "Ctrl + O"). Ao lado esquerdo da tela, surgirá um resumo de todos os métodos e atributos presentes na classe, podemos clicar em cada um deles para termos um acesso rápido.



**02Utilizando construtores**

Abaixo temos algumas afirmações a respeito da utilização de construtores, qual delas é verdadeira?

Parte superior do formulário

* Construtores são utilizados para inicialização dos atributos.

Correto, a inicialização de atributos é a principal responsabilidade do construtor.

* Alternativa incorreta

Não podemos ter mais de um construtor

* Alternativa incorreta

Construtores não podem receber parâmetros.

* Alternativa incorreta

Construtores não têm utilidade real, podemos deixar os atributos públicos e defini-los manualmente.

Parte inferior do formulário

Parabéns, você acertou!

Este conteúdo foi útil para o seu aprendizado?

Sim

Não muito

**03Aonde está o erro?**

Ainda sobre o jogo de Luan, temos outro trecho do código:

public class Jogo {

//Código omitido

private Componente comp;

public Jogo(Usuario usuario){

this.comp = usuario;

}

}Copiar código

Porém o código acima sequer compila. Qual dos motivos abaixo explica a razão desse acontecimento?

Parte superior do formulário

* Não podemos atribuir porque é um elemento privado.
* Alternativa incorreta

O uso da palavra reservada this está errado.

* Alternativa incorreta

Os atributos não podem ter o mesmo nome.

* Alternativa incorreta

Está sendo feita uma atribuição de objetos de tipos diferentes.

Correto!

Parte inferior do formulário

Parabéns, você acertou!

Este conteúdo foi útil para o seu aprendizado?

Sim

Não muito

**04Static**

**Transcrição**

Mais um desafio: precisaremos saber quantas contas foram abertas no sistema. Na linguagem Java, saberemos quantas contas foram instanciadas.

Na classe TestaValores, criaremos uma variável chamada total, que começa com o valor 0.

Lembre-se que variável local precisa ser zerada.

Feito isso, escreveremos que a cada vez que surgir uma nova conta através do new, isso seja contabilizado na variável total++.

public class TestaValores {

public static void main(String[] args) {

int total = 0;

Conta conta = new Conta(1337, 24226);

total++;

System.out.println(conta.getAgencia());

conta.setAgencia(1232123);

}

}Copiar código

Essa saída é funcional, mas apresenta problemas de ordem prática, por exemplo, toda a vez que uma nova conta for aberta, o desenvolvedor deve escrever a variável total++.

Uma forma melhor de contabilizar as contas que foram criadas no nosso banco é acionar o **construtor**. Na classe Conta, adicionaremos no construtor a requisição da variável total++.

public class Conta {

private double saldo;

private int agencia;

private int numero;

private Cliente titular;

public Conta(int agencia, int numero) {

total ++;

this.agencia = agencia;

this.numero = numero;

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

}Copiar código

Para que o nosso código compile, iremos declarar total como um atributo do tipo int privado. Feito isso, iremos imprimir o total de contas adicionando o texto o total de contas é mais a variável.

public class Conta {

private double saldo;

private int agencia;

private int numero;

private Cliente titular;

private int total;

public Conta(int agencia, int numero) {

total ++;

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

this.agencia = agencia;

this.numero = numero;

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

}Copiar código

A variável total++ que escrevemos na classe TestaValores não tem qualquer ligação com o atributo total que criamos. Portanto, podemos excluí-la, já que descobrimos uma forma mais prática de contabilizar o total de contas.

public class TestaValores {

public static void main(String[] args) {

Conta conta = new Conta(1337, 24226);

System.out.println(conta.getAgencia());

conta.setAgencia(1232123);

}

} Copiar código

Ao executarmos nosso programa, veremos que o valor impresso será o total de contas é 1. O resultado, portanto, está correto. Adicionaremos mais duas novas contas para testarmos se o programa está operando corretamente.

public class TestaValores {

public static void main(String[] args) {

Conta conta = new Conta(1337, 24226);

System.out.println(conta.getAgencia());

conta.setAgencia(1232123);

Conta conta2 = new Conta(1337, 16549);

Conta conta3 = new Conta(2112, 14660);

}

} Copiar código

Veremos que o resultado do programa é o total de contas é 1. Ou seja, não houve a atualização quanto ao número de contas. Percebemos que houve um erro na criação do atributo total na classe Conta. É como se cada objeto conta tivesse um saldo, agencia, numero, titular e total.

O que queremos é que total fosse uma variável que não ficasse em cada instância, mas em algum lugar da classe Conta, algo como um atributo compartilhado e não de um objeto especificamente.

Para isso, existe a palavra-chave static. O static faz com que o atributo seja da classe, e não mais do objeto. Com isso, todo o objeto conta possui acesso a um único total.

Ao lado esquerdo da variável total++ podemos adicionar a classe Conta. Afinal, não estamos mais fazendo referência à uma conta - como o this fazia - e sim à classe Conta.

public class Conta {

private double saldo;

private int agencia;

private int numero;

private Cliente titular;

private static int total;

public Conta(int agencia, int numero) {

Conta.total ++;

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

this.agencia = agencia;

this.numero = numero;

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

}Copiar código

Ao executarmos a aplicação, veremos que o valor impresso será o total de contas é 3.

Caso retirássemos os sysout da classe Conta, o programa não imprimiria mais o total de contas.

public Conta(int agencia,int numero) {

Conta.total ++;

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

this.agencia = agencia;

this.numero = numero;

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

} Copiar código

Tendo isso em vista, poderíamos imprimir o total de contas na classe TestaValor fazendo um sysout no atributo total. Isso funcionaria se o atributo não fosse privado. A questão é que não há mais necessidade de marcar conta2 ou conta3, pois o total é compartilhado entre as instâncias.

public class TestaValores {

public static void main(String[] args) {

Conta conta = new Conta(1337, 24226);

System.out.println(conta.getAgencia());

conta.setAgencia(1232123);

Conta conta2 = new Conta(1337, 16549);

Conta conta3 = new Conta(2112, 14660);

System.out.println(Conta.total);

}

}Copiar código

Como já vimos, é interessante que este atributo seja privado para preservarmos características essenciais segundo a regra de negócios que rege o banco. Portanto, deveremos criar um getter para o atributo total na classe Conta.

public int getTotal() {

return Conta.total;

}Copiar código

Feito isso, acionaremos o getter na classe TestaValores usando o System.out. Não precisamos especificar conta1 ou conta2, porque estamos nos referindo à classe Conta.

public class TestaValores {

public static void main(String[] args) {

Conta conta = new Conta(1337, 24226);

System.out.println(conta.getAgencia());

conta.setAgencia(1232123);

Conta conta2 = new Conta(1337, 16549);

Conta conta3 = new Conta(2112, 14660);

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

}

} Copiar código

Veremos que o código não está sendo compilado, vamos entender o porquê: todos os métodos declarados eram da instância conta, ou seja, de uma conta específica. Neste caso, estamos nos comunicando com um atributo total. Para isso, precisaremos declarar que o método é static.

public static int getTotal() {

return Conta.total;

}Copiar código

O método static possui usos muito interessantes, mas possui suas limitações. Embora não seja do interesse do nosso projeto, não poderíamos acessar um atributo de instância via this, como saldo.

public static int getTotal() {

System.out.println(this.saldo);

return Conta.total; Copiar código

Os métodos estáticos acessam apenas atributos estáticos.

**05Por que não soma?**

Luan desenvolveu um jogo, e quer sempre manter o número de jogadores atualizados. Para isso Luan escreveu o código abaixo:

public class Jogador {

//Código omitido

private int total = 0;

public Jogador(//atributos){

total++;

}

}Copiar código

Porém o contador sempre apresenta 1 após inserir um novo jogador. Qual dos motivos abaixo explica a razão desse acontecimento?

Parte superior do formulário

* Luan deveria deixar o total como global, para que todos possam acessar.
* Alternativa incorreta

É impossível fazer isso.

* Alternativa incorreta

O total deveria estar como estático, assim sempre que fosse criado um novo objeto do tipo Jogador não seria criado um novo total, mantendo o valor correto.

Correto!

* Alternativa incorreta

Luan deveria deixar o total como público, para que possa ser somado sem problemas.

Parte inferior do formulário

Parabéns, você acertou!

Este conteúdo foi útil para o seu aprendizado?

Sim

Não muito

**06Mãos 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");

}Copiar código

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");

}Copiar código

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);Copiar código

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

}Copiar código

6) Dentro do construtor adicione total++;

7) Crie o método estático getter getTotal():

public static int getTotal(){

return Conta.total;

}Copiar código

8) Teste o getter em seu método main:

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

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!

**Opinião do instrutor**

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;

}

}Copiar código

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;

}

}Copiar código

Qualquer dúvida nos pergunte no fórum!

**07Para 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

}Copiar código

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);Copiar código

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

}Copiar código

Agora a utilização exige a presença dos 3 parâmetros definidos.

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

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

}Copiar código

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);Copiar código

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;

}

}

//....

}Copiar código

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

}Copiar código

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);

}Copiar código

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

}Copiar código

**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.

**08O 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

Este conteúdo foi útil para o seu aprendizado?

Sim

Não muito

**09E agora?**

Nesse curso apresentamos os fundamentos do paradigma *Orientação a Objetos*. Vimos como definir classes e quais são os membros delas como atributos, métodos e construtores. Aprendemos que as classes servem como planta usada para criar objetos e os métodos encapsulam a implementação. A notícia boa é que não chegamos ao fim desse paradigma poderoso e existem ainda tópicos mais avançados como **herança, interfaces, polimorfismo** entre outros assuntos da Orientação a Objetos.

Convido você a assistir o próximo curso que fala justamente sobre esses tópicos. Você aplicará esses novos recursos dentro de tudo que você aprendeu nesse curso e no anterior. Espero que você tenha gostado desse curso e peço para avaliar e dar um feedback para gente.

Muito obrigado, Paulo =)

**10Arquivos do projeto atual**

No link abaixo, você encontra o projeto até o momento atual do curso.

<https://github.com/alura-cursos/Curso-Java-parte-2-Introdu-o-Orienta-o-a-Objetos/archive/capitulo6.zip>

**Opinião do instrutor**

Lembre-se de usar o fórum em caso de dúvidas.