**01Atributos privados e encapsulamento**

**Transcrição**

Neste ponto do curso, já sabemos como trabalhar com métodos e atributos, inclusive, atributos que servem como referência para outros objetos. Veremos qual é o nosso novo fator limitante para o progresso do projeto **ByteBank**.

Criaremos uma nova classe de teste chamada TesteSacaNegativo.

Nosso objetivo é ficar com uma quantidade negativa de dinheiro. Usaremos o nome da variável conta, mas poderia ser qualquer outro de sua escolha, diferente do **tipo** Conta, que, obrigatoriamente, precisa ser o nome de uma classe que exista no sistema.

Lembrando que é muito comum criar uma variável que tenham o mesmo nome da classe. Depositaremos 100 reais na conta.

public class TesteSacaNegativo {

public static void main(String[] args) {

Conta conta = new Conta();

conta.deposita(100);

}

}Copiar código

Feito isso, tentaremos sacar 200 reais, e verificaremos o que acontece quando executamos a aplicação.

public class TesteSacaNegativo {

public static void main(String[] args) {

Conta conta = new Conta();

conta.deposita(100);

conta.saca(200);

System.out.println(conta.saldo);

}

}Copiar código

O resultado da aplicação foi a impressão do saldo 100. Isso ocorreu, pois o saque não foi efetivado, já que o valor disponível na conta não era o suficiente.

Quando mantemos o "Ctrl" pressionado e passamos o mouse sobre a código, vemos que o método retornou o valor false. Podemos passar o valor booleano diretamente para o sysout, ou seja, não precisamos sempre guardar o retorno de um método dentro de uma variável.

public class TesteSacaNegativo {

public static void main(String[] args) {

Conta conta = new Conta();

conta.deposita(100);

System.out.println(conta.saca(200));

System.out.println(conta.saldo);

}

}Copiar código

Ao executarmos novamente aplicação, veremos o valor false impresso. Conseguimos estipular o critério de que *nenhuma conta pode ter valores negativos*.

Porém, existe um truque que tentaremos fazer. Ao tentarmos sacar 101 reais da nossa conta, cujo saldo é 100, o procedimento não será efetivado. A não ser que, diretamente no atributo saldo, estipulamos que o seu valor é o saldo - 101.

public class TesteSacaNegativo {

public static void main(String[] args) {

Conta conta = new Conta();

conta.deposita(100);

System.out.println(conta.saca(101));

System.out.println(conta.saldo);

conta.saldo = conta.saldo - 101;

System.out.println(conta.saldo);

}

}Copiar código

No término da execução do programa, veremos que o valor impresso será -1.0. Supondo que seja um requisito do **ByteBank** que não haja valores negativos nas contas, temos um problema.

Uma solução é avisar para os funcionários do nosso banco para que nunca acessem o atributo saldo diretamente, e sim, invocando o método adequado.

O ideal é que sempre trabalhemos utilizando os **métodos**, nunca diretamente os atributos. Podemos utilizar a analogia do funcionamento de um carro: utilizamos o pedal de aceleração para que o carro se locomova, não precisamos abrir o capo e reviramos mecanismos de injeção de gasolina de forma manual para gerar a aceleração necessária.

A ideia é utilizar a interface adequada para a melhor funcionalidade de um sistema. A forma mecânica de funcionamento do carro está escondida, ou seja, *encapsulada*.

O que precisamos saber operar é a metodologia de funcionamento do carro, o mesmo vale para a conta bancária.

Queremos, portanto, que a manipulação direta de atributos seja proibida.

public class TesteSacaNegativo {

public static void main(String[] args) {

Conta conta = new Conta();

conta.deposita(100);

System.out.println(conta.saca(200));

System.out.println(conta.saldo);

//proibido

conta.saldo = conta.saldo - 101;

System.out.println(conta.saldo);

}

}Copiar código

No Java, existe a possibilidade de ocultarmos um atributo, deixá-lo privado. Na classe Conta, escreveremos ao lado do atributo saldo que queremos o encapsular. Para isso, utilizamos a palavra-chave private

public class Conta {

private double saldo;

int agencia;

int numero;

Cliente titular;

// ...Copiar código

A partir do momento em que um atributo se torna **privado**, isso quer dizer que ele não pode ser lido ou modificado, a não ser na própria classe. Esse é o conceito principal de encapsulamento.

Ainda existe um problema: na classe TesteSacaNegativo, não podemos mais imprimir o valor de saldo através do sysout. É preciso utilizar um novo método para acessar o atributo saldo.

Na classe Conta, criaremos um método que devolve/informe o saldo. Chamaremos esse novo método de pegaSaldo, que não precisará receber parâmetro, mesmo assim, os parênteses são obrigatórios.

Ao lado esquerdo do método, colocaremos o seu retorno: um double. Escreveremos, também, o public. Dentro do método, diremos que ele simplesmente retorna o valor de saldo, utilizando a palavra-chavereturn.

public class Conta {

private double saldo;

int agencia;

int numero;

Cliente titular;

public void deposita(double valor) {...}

public boolean saca(double valor) {...}

public boolean transfere(double valor, Conta destino, Conta origem) {...}

public double pegaSaldo() {

return this.saldo;

}Copiar código

É interessante que as funções sejam bem localizadas no código, por isso, o objetivo desse método é simplificar os processos de manutenção do sistema sem que precisemos fazer alterações em múltiplos trechos do código.

De volta a classe TesteNegativo, iremos invocar o método pegaSaldo:

public class TesteSacaNegativo {

public static void main(String[] args) {

Conta conta = new Conta();

conta.deposita(100);

System.out.println(conta.saca(101));

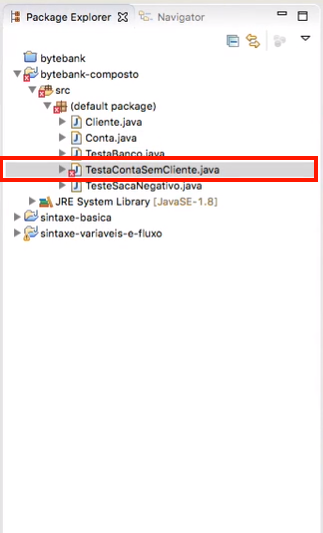
conta.saca(101);

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

}

}Copiar código

Há uma classe no nosso projeto que ainda está acessando o atributo saldo da maneira antiga.



O TestaContaSemCliente está dessa forma:

public static void main(String[] args) {

Conta contaDaMarcela = new Conta();

System.out.println(contaDaMarcela.saldo);

// ...Copiar código

Vamos corrigir o código :

public class TestaContaSemCliente {

public static void main(String[] args) {

Conta contaDaMarcela = new Conta();

System.out.println(contaDaMarcela.pegaSaldo());

// ...Copiar código

**02Público x Privado**

Luan resolveu criar um modelo para treinar orientação a objeto e encapsulamento:

public class Cliente {

String nome;

private String cpf;

int idade;

}Copiar código

E está criando um objeto Cliente na outra classe:

public class Banco {

public static void main(String[] args) {

Cliente cliente = new Cliente();

cliente.nome = "José";

cliente.cpf = "12312312312";

cliente.idade = 49;

}

}Copiar código

O que podemos afirmar sobre o código?

Parte superior do formulário

* Não compila pois o cpf é privado.

Correto, não podemos acessar o atributo cpf na classe Banco (o atributo cpf é visível apenas na classe Cliente ).

* Alternativa incorreta

As atribuições estão sendo feitas do jeito correto. O código compila e executa.

* Alternativa incorreta

Não compila pois a classe Cliente não foi criada do jeito correto.

Parte inferior do formulário

Parabéns, você acertou!

Este conteúdo foi útil para o seu aprendizado?

Sim

Não muito

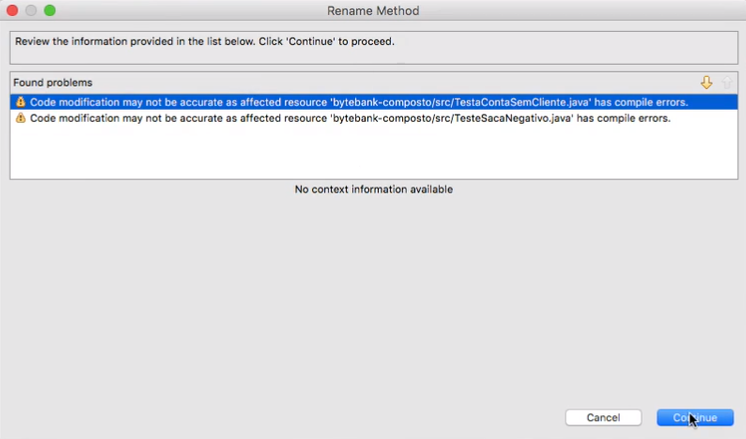
**03Getters e Setters**

**Transcrição**

O nome do método pegaSaldo() que criamos, poderia ter qualquer outro nome. Por uma questão de convenção, alteraremos o nome para getSaldo().

Devido à essa alteração, os outros arquivos não serão compilados, porque estarão com o nome antigo pegaSaldo(). Para resolvermos esse problema, na classe Conta, selecionaremos o método e daremos um clique duplo sobre ele. Feito isso, escolheremos ao opção "Refactor > Rename".

Surgirá uma caixa de diálogo e pressionamos "Continue". Com isso, o nome do método foi alterado em todos os arquivos.



O nome desse tipo de método que simplesmente *exibe uma informação* é **getter**. Não se trata de um elemento que compõe a sintaxe do Java, **get** não é uma palavra-chave do Java.

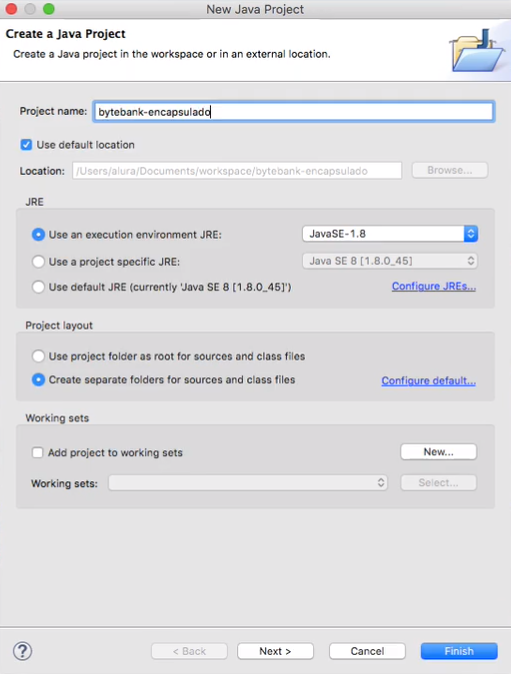
Dizemos que temos um método *getter* para saldo, pois este atributo é privado e sentimos a necessidade ao longo do projeto de acessar a informação contida em saldo.

Percebam que não há a necessidade de criarmos novo método equivalente - algo como "setSaldo" - para *modificar o atributo saldo*, essas modificações serão feitas pelos já conhecidos saca(), deposita() e transfere(). Queremos que as alterações de saldo em nossas contas do **ByteBank** sejam sempre feitas através de **saques**, **depósitos** ou **transferências**.

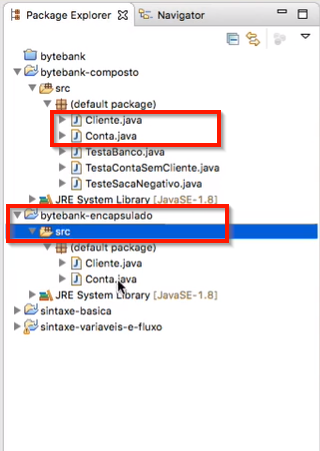
A ideia é diferente para agencia, numero e titular. Não temos métodos para alterar esses atributos, e por enquanto, só conseguimos fazer modificações diretas, o que quer dizer que podemos inserir números negativos como valores, por exemplo.

Com o tempo, entenderemos que o ideal é que todos os atributos sejam privados. Existem casos raros de atributos públicos, mas são realmente exceções.

Criaremos um novo projeto Java que chamaremos de bytebank-encapsulado. "Encapsulado" é o termo utilizado para elementos ocultos, escondidos.



Na área *Package Explorer*, copiaremos as classes Cliente e Conta e as colaremos no default package do novo projeto bytebank-encapsulado.



Utilizando o atalho "Ctrl + W", fecharemos todas as abas do Eclipse. Também fecharemos o projeto bytebank-composto para não confundirmos as classes dos dois projetos, tornado o ambiente de trabalho mais claro e limpo.

Na classe Conta, transformaremos todos os atributos em private.

public class Conta {

private double saldo;

private int agencia;

private int numero;

private Cliente titular;

public void deposita(double valor) {

this.saldo += valor;

}

}Copiar código

O fato de tornarmos um atributo privado facilita a modificação e atualização do código. Com uma classe sendo responsável por seus próprios atributos, a manutenção do sistema se torna localizada, por conseguinte, mais simples.

Para que os atributos sejam acessados fora da classe, utilizaremos os *getters*. Na classe Conta criaremos um método público denominado getNumero() que devolve um int e retorna numero.

public class Conta {

private double saldo;

private int agencia;

private int numero;

private Cliente titular;

public void deposita(double valor) {...}

public boolean saca (double valor) {...}

public boolean transfere(double valor, Conta destino, Conta origem) {...}

public double pegaSaldo() {...}

public int getNumero() {

return this.numero;

}

}Copiar código

Além de termos um método que devolve qual é o numero de uma conta, queremos também um método que *altere* esse mesmo numero. Esse tipo de método é chamado de **set**.

Diferente do **get**, há um parâmetro a ser passado para o método, afinal queremos modificar o número e precisamos informar qual será essa modificação. Usaremos uma variável novoNumero e o método não retornará nada.

public class Conta {

private double saldo;

private int agencia;

private int numero;

private Cliente titular;

public void deposita(double valor) {...}

public boolean saca(double valor) {...}

public boolean transfere(double valor, Conta destino, Conta origem) {...}

public double pegaSaldo() {...}

public int getNumero() {

return this.numero;

}

public void setNumero(int novoNumero) {

this.numero = novoNumero;

}

}Copiar código

Criaremos numa nova classe para utilizar tanto o *getter* como o *setter*. Essa nova classe será chamada de TestaGetESet. Na nova classe, criaremos uma nova conta chamada conta e a instanciaremos através da palavra-chave new. Feito isso, escreveremos o numero desta nova conta como sendo 1337.

public class TestaGetESet {

public static void main(String[] args) {

Conta conta = new Conta();

conta.numero = 1337

}

}Copiar código

Da forma como o nosso código está escrito ele não será compilado, afinal, numero é um atributo privado e *precisa* ser acessado através de um método. Utilizaremos o método setNumero()

public class TestaGetESet {

public static void main(String[] args) {

Conta conta = new Conta();

conta.setNumero(1337);

}

}Copiar código

Uma das vantagens de utilizar os métodos, é que dentro do próprio método setNumero() podemos adicionar ifs, gerando especificações, mensagens de erro, e assim por diante.

Para imprimirmos o valor de numero, utilizamos o sysout e o método getNumero(). Ao executarmos a aplicação, teremos o resultado de 1337.

public class TestaGetESet {

public static void main(String[] args) {

Conta conta = new Conta();

conta.setNumero(1337);

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

}

}Copiar código

Como já foi dito, no caso do atributo saldo, não criaremos um método setter, pois trabalharemos com os métodos deposita(), saca() e transfere() para modificar seu valor.

Criaremos métodos **get** e **set** para o atributo agencia, mas dessa vez, de uma maneira mais simples.

Antes, voltemos ao método setNumero() na classe Conta.

public void setNumero(int novoNumero) {

this.numero = novoNumero;

}Copiar código

Em muitos casos, em vez de escrever o nome da variável novoNumero, utiliza-se o mesmo nome do atributo, no caso, numero.

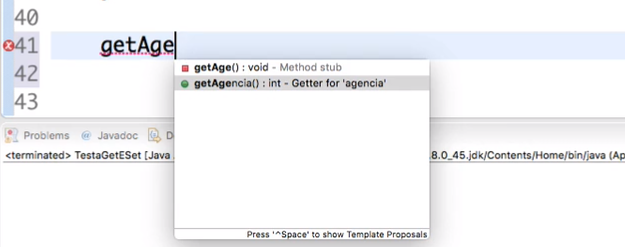
A princípio, podemos nos confundir, mas basta nos atentarmos para o uso da palavra-chave this, ao lado esquerdo do código, que marca a referência ao atributo.

public void setNumero(int numero) {

this.numero = numero;

}Copiar código

Escreveremos o método get para o atributo agencia. Percebam que basta iniciarmos a escrever o código e pressionarmos o atalho "Ctrl + Space", será sugerido a criação automática de getAgencia(). Escolheremos esta opção e pressionaremos "Enter".



Feito isso, o código será automaticamente escrito na classe Conta.

public class Conta {

// atributos

// método deposita

// método saca

// método transfere

// método pegaSaldo

public int getNumero() {

return this.numero;

}

public void setNumero(int numero) {

this.numero = numero;

}

public int getAgencia() {

return agencia;

}

}Copiar código

Adicionaremos o this para marcarmos bem o que é atributo e variável temporária.

public int getAgencia() {

return this.agencia;

}Copiar código

Podemos realizar o mesmo procedimento com o método **set** do atributo agencia. O Eclipse possui teclas de atalho que facilitam muito a construção do código.

No cabeçalho de ferramentas existe opção de gerar *getters* e *setters*, basta selecionarmos em "Source > Generate Getters and Setters", mas não há necessidade de acessá-los desta forma, já que o "Ctrl + Space" disponibiliza a mesma função, neste caso.

**04Criando Getters e Setters**

Paulo está criando getters e setters para seguinte classe:

public class Aluno {

private String nome;

private String matricula;

}Copiar código

Como ficaria, seguindo a convenção explicada no vídeo anterior, a declaração dos getters e setters para os dois atributos da classe?

Parte superior do formulário

* public void getNome(){
* return this.nome;
* }
* public void getMatricula(){
* return this.matricula;
* }
* public String setMatricula(String matricula){
* this.matricula = matricula;
* }
* public String setNome(String nome){
* this.nome = nome;
* }
* Alternativa incorreta
* public int pegaNome(){
* return this.nome;
* }
* public String pegaMatricula(){
* return this.matricula;
* }
* public void atribuiMatricula(String matricula){
* this.matricula = matricula;
* }
* public void atribuiNome(String nome){
* this.nome = nome;

}

* Alternativa incorreta
* public String getNome(){
* return this.nome;
* }
* public String getMatricula(){
* return this.matricula;
* }
* public void setMatricula(String matricula){
* this.matricula = matricula;
* }
* public void setNome(String nome){
* this.nome = nome;

}

Correto!

Parte inferior do formulário

Parabéns, você acertou!

Este conteúdo foi útil para o seu aprendizado?

Sim

Não muito

**05Getters e Setters de referência**

**Transcrição**

Elevaremos o nível de dificuldade na prática de métodos **getters** e **setters** considerando o tipo Cliente.

Neste ponto, já trabalhamos com os atributos saldo, agencia e numero.

Lembrando que para o caso de saldo, não precisamos utilizar um setter, porque outros métodos como deposita() e saca() já servem para modificar o atributo.

É possível que haja atributos em que não exista a necessidade tanto de um método get quanto de um set, como por exemplo, uma conexão para o banco de dados que só possui um uso interno com relação à própria classe Conta.

No caso do atributo titular, é de nosso interesse que possamos acessar o titular de uma determinada conta e modificá-lo.

Na classe TestaGetESet, criaremos um novo Cliente e faremos a atribuição do titular cujo valor é null, para um novo titular identificado como paulo.

public class TestaGetESet {

public static void main(String[] args) {

Conta conta = new Conta();

conta.setNumero(1337);

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

Cliente paulo = new Cliente();

conta.titular = paulo;

}

}Copiar código

O código desta forma não compila, pois titular é um atributo privado. Precisaremos invocar algum método para finalizar nossa operação.

Poderíamos acionar um método hipotético, como setTitular(). Esse código também não compila, pois ainda não criamos seu **setter**.

public class TestaGetESet {

public static void main(String[] args) {

Conta conta = new Conta();

conta.setNumero(1337);

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

Cliente paulo = new Cliente();

conta.setTitular(paulo);

}

}Copiar código

Na classe Conta, criamos no novo método setTitular(). Feito isso, criaremos, também, o método getTitular().

public class Conta {

// atributos

// método deposita

// método saca

// método transfere

// método pegaSaldo

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 titular;

}

}Copiar código

Feito isso, veremos que o nosso código na classe TestaGetESet já está sendo compilado e podemos fazer alterações no atributo titular. Mudaremos o nome do titular para paulo silveira e queremos incluir cpf e profissao.

Poderíamos modificar o atributo nome de forma direta:

public class TestaGetESet {

public static void main(String[] args) {

Conta conta = new Conta();

conta.setNumero(1337);

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

Cliente paulo = new Cliente();

paulo.nome = "paulo silveira";

conta.setTitular(paulo);

}

}Copiar código

Essa modificação direta foi possível, pois na classe Cliente todos os atributos são não-privados, diferentemente da classe Conta.

Vamos transformar os atributos da classe Cliente em *privados*, pois queremos colocar padrões específicos em cada atributo, como não permitir números em nome, por exemplo.

public class Cliente {

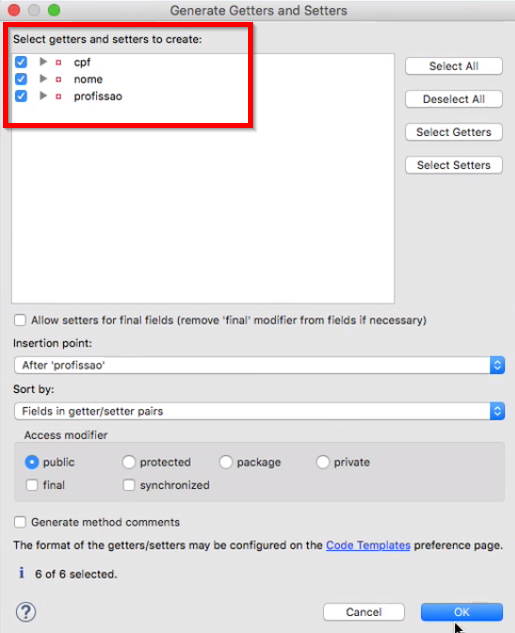
private String nome;

private String cpf;

private String profissao;

} Copiar código

Para que você não precise ficar escrevendo todos os getters e setters nececessários, no cabeçalho do Eclipse acione a opção "Source > Generate Getters and Setters..." Em "Select getters and setters to create" selecionaremos todas as opções de atributos.



Com isso, todos os métodos foram automaticamente gerados.

public class Cliente {

private String nome;

private String cpf;

private String profissao;

public String getNome() {

return nome;

}

public String getCpf() {

return cpf;

}

public String getProfissao() {

return profissao;

}

public void setProfissao(String profissao) {

this.profissao = profissao

}

} Copiar código

Com isso, na classe TestaGetESet, não podemos mais alterar diretamente os atributos de Cliente, precisamos invocar métodos.

public class TestaGetESet {

public static void main(String[] args) {

Conta conta = new Conta();

conta.setNumero(1337);

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

Cliente paulo = new Cliente();

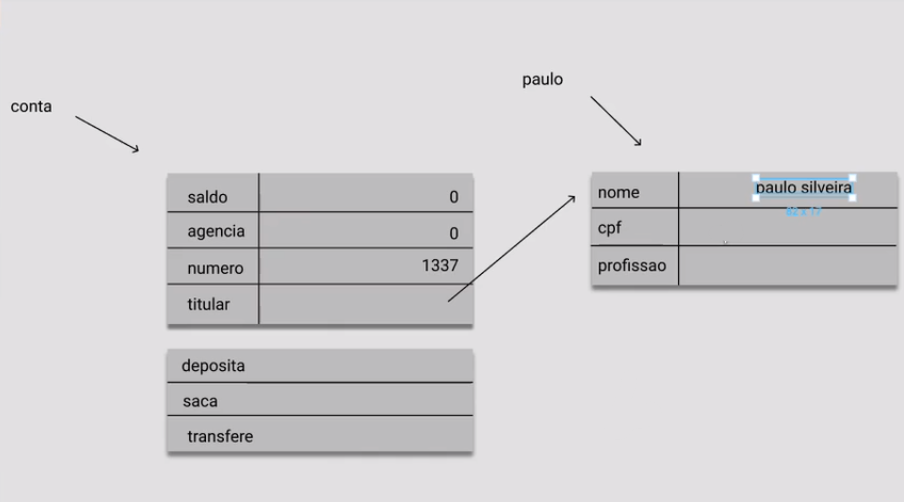
paulo.setNome("paulo silveira");

conta.setTitular(paulo);

}

}Copiar código

Nosso código no diagrama está representado da seguinte maneira: temos uma variável conta que faz referência a um objeto Conta. O atributo titular da conta bancária para o atributo nome do objeto Cliente, referenciado pela variável paulo.



Tentamos imprimir apenas o titular da conta para averiguarmos como o código se comporta.

public class TestaGetESet {

public static void main(String[] args) {

Conta conta = new Conta();

conta.setNumero(1337);

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

Cliente paulo = new Cliente();

paulo.setNome("paulo silveira");

conta.setTitular(paulo);

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

}

}Copiar código

Ao executarmos a aplicação, veremos que o foi impresso o valor de Cliente@15db9742. Não era esse o resultado que estávamos querendo, pois esse é o valor da referência ao objeto, e nós queremos imprimir o atributo nome do objeto.

public class TestaGetESet {

public static void main(String[] args) {

Conta conta = new Conta();

conta.setNumero(1337);

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

Cliente paulo = new Cliente();

paulo.setNome("paulo silveira");

conta.setTitular(paulo);

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

}

}Copiar código

Reconfiguraremos a linha, utilizando os métodos getters para chegarmos à informação do atributo nome do objeto.

public class TestaGetESet {

public static void main(String[] args) {

Conta conta = new Conta();

conta.setNumero(1337);

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

Cliente paulo = new Cliente();

paulo.setNome("paulo silveira");

conta.setTitular(paulo);

System.out.println(conta.getTitular().getNome());

}

}Copiar código

Ao rodarmos o programa, veremos que o resultado impresso é paulo silveira, como o esperado.

Caso seja do nosso interesse alterar alguma informação do titular, o percurso é parecido. Iremos incluir a profissão programador: acionaremos getTitular() e depois o setProfissao().

System.out.println(conta.getTitular().getNome());

conta.getTitular().setProfissao("programador"); Copiar código

Podemos quebrar o código em duas linhas.

public class TestaGetESet {

public static void main(String[] args) {

Conta conta = new Conta();

conta.setNumero(1337);

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

Cliente paulo = new Cliente();

paulo.setNome("paulo silveira");

conta.setTitular(paulo);

System.out.println(conta.getTitular().getNome());

conta.getTitular().setProfissao("programador");

//agora em duas linhas:

Cliente titularDaConta = conta.getTitular();

titularDaConta.setProfissao("programador");

}

}Copiar código

O resultado é mesmo, a única diferença é que criamos uma variável temporária. Podemos quebrar a linha do código para facilitar a leitura nos momentos em que ela estiver muito grande.

A variável titularDaConta possui o mesmo endereço da variável paulo. como podemos verificar, realizando o sysout.

public class TestaGetESet {

public static void main(String[] args) {

Conta conta = new Conta();

conta.setNumero(1337);

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

Cliente paulo = new Cliente();

paulo.setNome("paulo silveira");

conta.setTitular(paulo);

System.out.println(conta.getTitular().getNome());

conta.getTitular().setProfissao("programador");

//agora em duas linhas:

Cliente titularDaConta = conta.getTitular();

titularDaConta.setProfissao("programador");

System.out.println(titularDaConta);

System.out.println(paulo);

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

}

}Copiar código

Ao executarmos a aplicação, veremos que todas as referências são para o mesmo endereço de memória Cliente@15db9742.

**06Vantagens de atributos privados**

Rômulo criou uma classe com diversos atributos privados, porém não sabe exatamente qual vantagem de utilizar esta abordagem. Qual das opções melhor define a vantagem do uso de atributos privados?

Parte superior do formulário

* A implementação interna pode ser modificada sem afetar nenhum código fora da própria classe.

Correto!

* Alternativa incorreta

Os atributos privados podem ser acessado em qualquer lugar que facilita a implementação da lógica de negócio.

* Alternativa incorreta

Usando atributos privados somos obrigados escrever getter e setters que é boa pratica.

Errado, você não precisa mas pode criar getters e setters.

Parte inferior do formulário

Parabéns, você acertou!

Este conteúdo foi útil para o seu aprendizado?

Sim

Não muito

**07Mão 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;Copiar código

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;

}Copiar código

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!

**Opinião do instrutor**

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;

}

}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

Dúvidas? Pergunte no fórum!

**08Para saber mais: Cuidado com o Modelo Anêmico**

Durante o aprendizado dos Getters e Setters é normal pensar sempre na necessidade deles para alterar algum estado dos nossos objetos.

Mas o uso dessa prática nem sempre é a mais indicada e expressa a realidade.

Observe a classe Conta representada abaixo que usa apenas getter e setters como métodos:

class Conta{

private String titular;

private double saldo;

public void setTitular(String titular){

this.titular = titular;

}

public String getTitular(){

return titular;

}

public void setSaldo(double saldo){

this.saldo = saldo;

}

public double getSaldo(){

return saldo;

}

}Copiar código

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);Copiar código

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)

}Copiar código

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)

}Copiar código

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;

}

}Copiar código

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)Copiar código

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 esses dois métodos com riscos de cairmos em um modelo anêmico que tem os seus comportamentos controlados por outras classes.

**09O que aprendemos?**

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

Este conteúdo foi útil para o seu aprendizado?

Sim

Não muito

**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/capitulo5.zip>

**Opinião do instrutor**

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