

## O conceito de encapsulamento no JAVA.

Encapsulamento é uma técnica da O.O aplicada a P.O.O que permite ocultar os dados (atributos) e as funcionalidades (métodos) de uma classe em nível de objeto, ou seja, é possível ocultar certos atributos e métodos dos objetos das classes tornando assim o sistema mais seguro.

Para aplicar tal conceito precisamos lançar mão de um outro conceito chamado de **visibilidade**, também conhecido como **modificador de acesso**, **visão de método** ou **visão de atributo** que é aplicado no momento da criação (declaração) de atributos e métodos. Existem três níveis de visibilidade no JAVA, são eles:

**public** – define um nível de visibilidade para um atributo ou método que permite que qualquer classe/objeto tenha acesso aos mesmos.

**private** – define um nível de visibilidade onde os atributos ou métodos só podem ser acessados por métodos da própria classe.

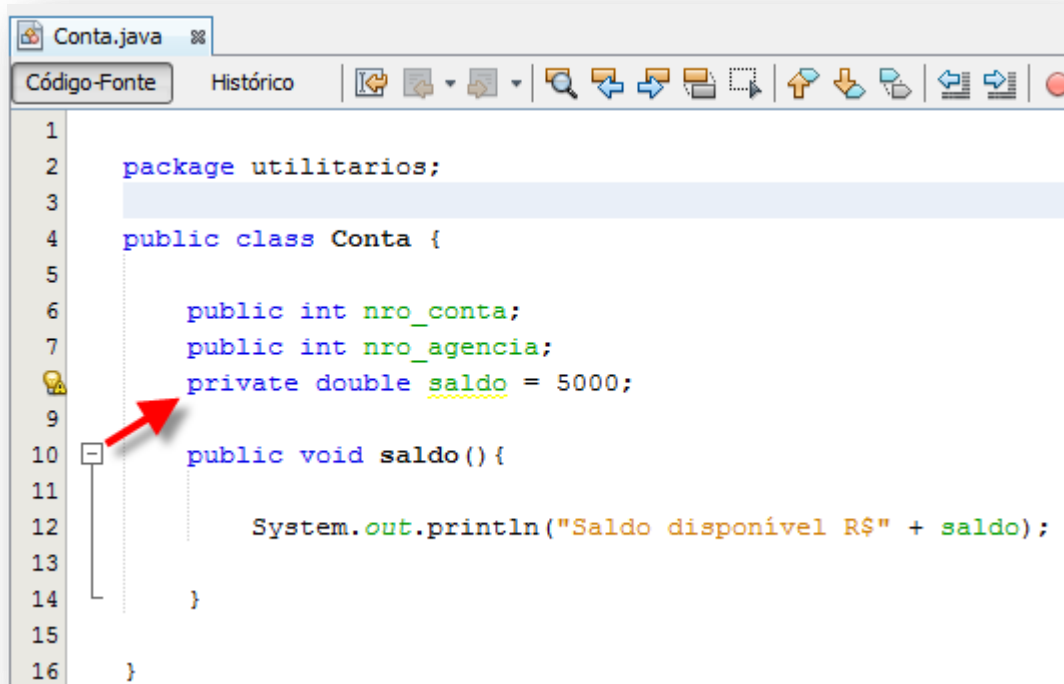
**protected** – define um nível de acesso onde os atributos e métodos podem ser acessados apenas pela própria classe que os criou ou classes que herdam a classe “dona” do atributo ou método que implementou o nível de acesso protected.

### “Acessadores” ou métodos “getters” e métodos “setters”.

A forma mais comum de implementar o conceito de encapsulamento é através de “**acessadores**” ou métodos de “**getter**” e “**setter**”, onde os métodos de “**get**” são responsáveis por retornar (recuperar) os valores de um atributo com a visibilidade definida como “**private**”, por exemplo e os métodos de “**set**” são responsáveis por configurar (atribuir) um valor aos atributos com a visibilidade definida como “**private**”.

Para exemplificar melhor vamos aplicar o conceito de encapsulamento no atributo “**saldo**” da classe “Conta”.

**1º passo** – Se você olhar o nível de visibilidade do atributo “**saldo**” na classe “**Conta**” vai notar que esse está como “**public**” vamos mudar esse nível para “**private**”, observe a imagem abaixo:

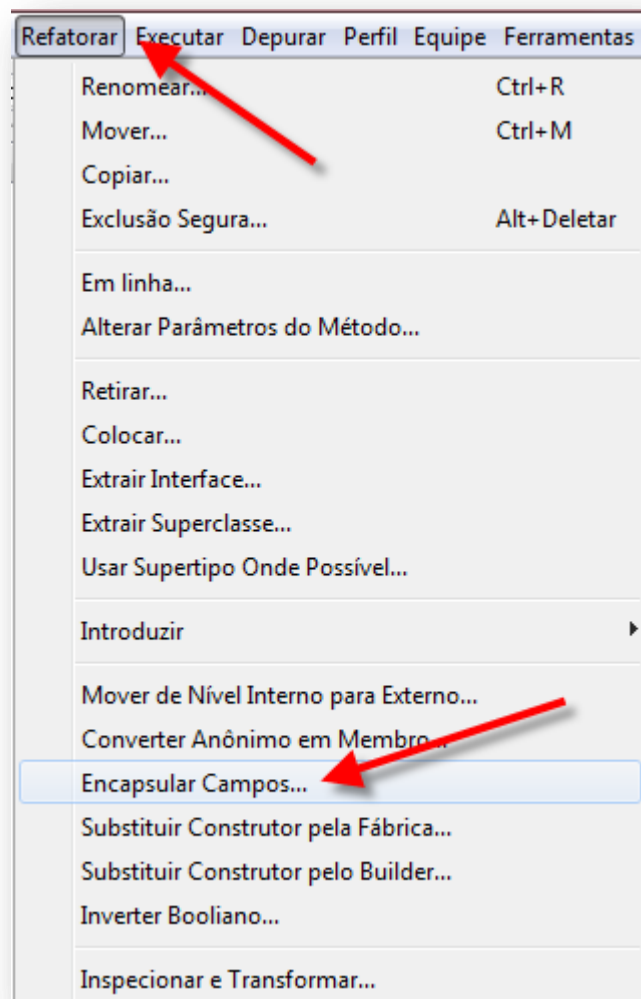


```
1 package utilitarios;
2
3
4 public class Conta {
5
6     public int nro_conta;
7     public int nro_agencia;
8     private double saldo = 5000;
9
10    public void saldo() {
11
12        System.out.println("Saldo disponível R$" + saldo);
13
14    }
15
16 }
```

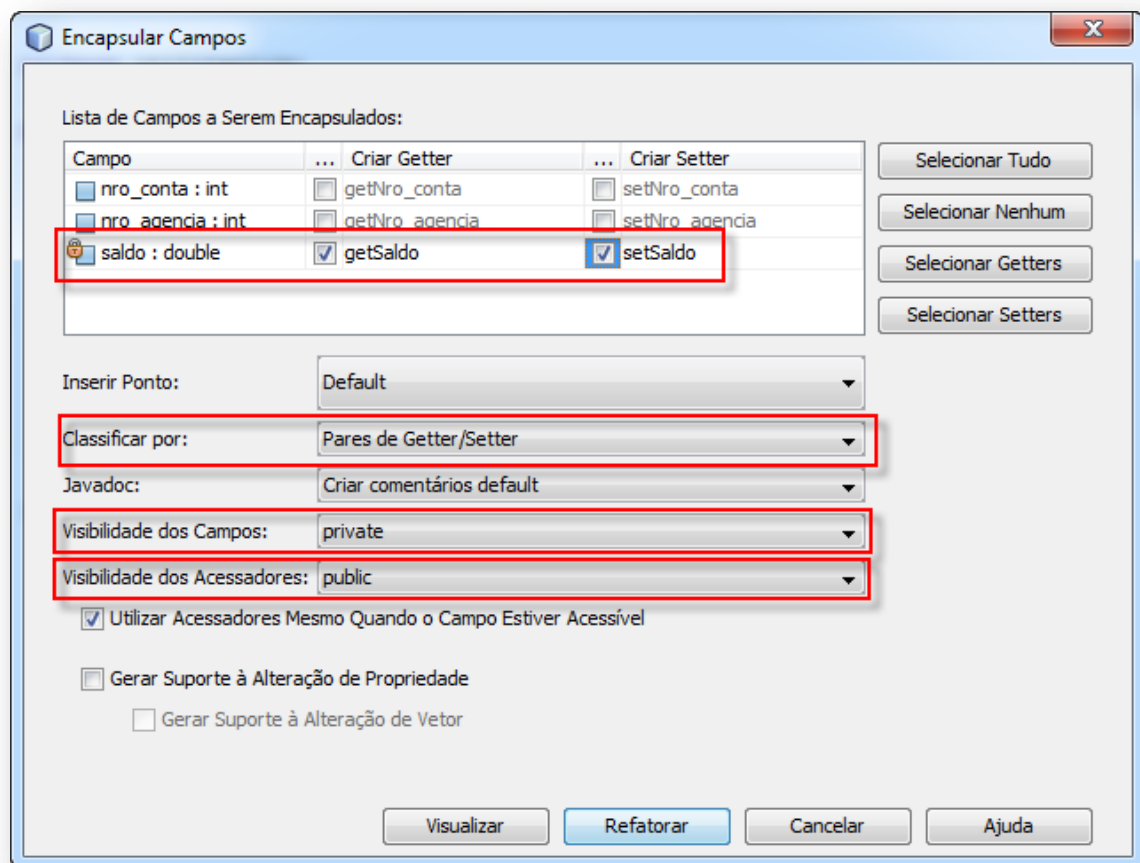
A partir de agora nenhum objeto pode acessar o atributo “saldo” diretamente, para encapsular o atributo devemos então criar um método de “set” e outro de “get” que vão permitir os acessos para configurar (“**settar**”) um valor e recuperar (“**gettar**”) um valor respectivamente.

Podemos criar tais métodos manualmente ou usar um assistente do “**NetBeans**” para tal.

**2º passo** – Acesse o menu “**Refatorar**” e em seguida clique sobre a opção “**Encapsular Campos...**”, observe a imagem abaixo:



Na janela seguinte temos o assistente para encapsular os campos, observe a imagem abaixo:

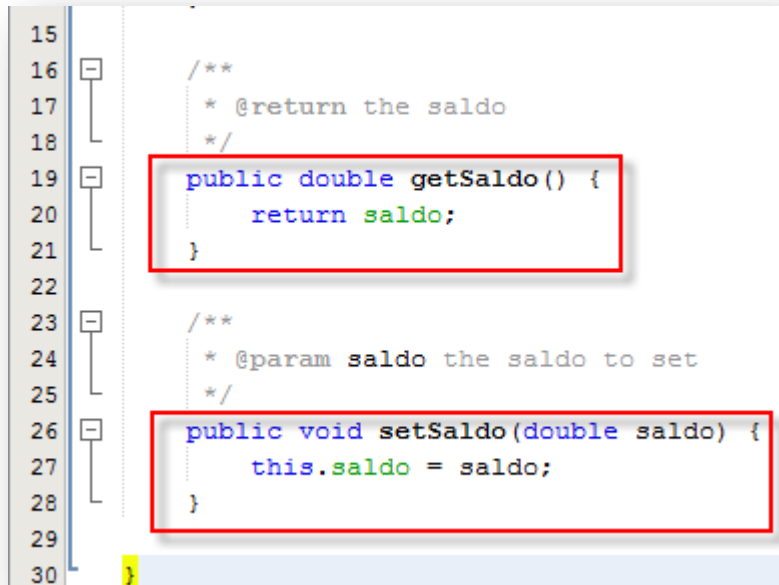


Observe as áreas em destaque:

- Primeiro temos uma listagem com os três atributos onde os dois primeiros “nro\_conta” e “nro\_agencia” são de acesso público e o terceiro em destaque é de acesso privado, marque as caixas de seleção referentes aos métodos “get” e “set”, observe que o NetBeans vai criar um método “getSaldo” e outro “setSaldo” as palavras “get” e “set” são apenas uma convenção, poderíamos criar esses métodos manualmente e nomear da forma que quiséssemos;
- O próximo controle em destaque na imagem é “Classificar por:” que indica que os métodos serão criados em pares, ou seja um “get” seguido por um “set” para cada atributo marcado na lista;
- Na sequência temos em destaque o controle “Visibilidade dos campos:” que nesse caso deve estar marcado como “private” pois estamos encapsulando os campos;

- Por último temos em destaque o controle “Visibilidade dos Acessadores:” que nada mais é do que a visibilidade dos métodos de “get” e “set”

Estando tudo devidamente configurado basta clicar em refatorar. A imagem abaixo ilustra o resultado:



```
15
16 /**
17  * @return the saldo
18  */
19 public double getSaldo() {
20     return saldo;
21 }
22
23 /**
24  * @param saldo the saldo to set
25  */
26 public void setSaldo(double saldo) {
27     this.saldo = saldo;
28 }
29
30
```

Note antes de qualquer coisa que eu retirei apenas o código que foi gerado o restante é igual ao código visto anteriormente, vamos dissecar os métodos começando por “**getSaldo**”, observe que após a definição do nível de acesso nesse caso “**public**” temos o comando “**double**” que é um tipo de dado, isso indica que o método “**getSaldo**” retorna um valor do tipo “**double**” quando chamado em seguida temos o nome do método seguido por parênteses vazios “**getSaldo()**” os parênteses servem para declararmos possíveis parâmetros (dados necessários para um método executar sua tarefa) que nesse caso, como estão vazios indicam que o método em questão executa sua tarefa sem a necessidade de dados externos. As chaves indicam a abertura do “corpo” do método e em seu interior temos uma linha de código “**return saldo**” indicando que o método retorna o valor contido no atributo “**saldo**”

Observe agora o método **“setSaldo”**, logo após a definição de seu nível de acesso **“public”** temos o comando **“void”** que indica que o método executa sua tarefa e não retorna valor algum no final, seguido pelo nome do método e nesse caso entre os parêntese temos a declaração de uma variável **“(double saldo)”** que indica que o método em questão requer um valor do tipo **“double”** para executar sua tarefa, note também que a variável dentro do parâmetro se chama **“saldo”**, ou seja, possui o mesmo nome do atributo, isso é possível graças ao conceito de escopo de variável.

O atributo **“saldo”** existe no escopo da classe e pode ser manipulado por todos os métodos da classe, já o parâmetro referenciado no método **“setSaldo”** dentro dos parênteses **“setSaldo(double saldo)”** existe apenas dentro do método, por essa razão não existe conflito entre os dois nomes.

Observando agora a linha de código dentro do corpo do método **“setSaldo”** temos o seguinte **“this.saldo = saldo”**, dentro do método temos a variável local de parâmetro **“saldo”** e precisamos atribuir o valor dessa para o atributo **“saldo”** de escopo de classe para que o JAVA possa fazer isso dentro do corpo do método é preciso sinalizar quem é o atributo e quem é a variável local, usamos então o comando **“this”** seguido de um ponto e do nome do atributo, isso indica que a palavra **“saldo”** a direita do ponto é o atributo e a palavra **“saldo”** a direita do sinal de atribuição **“=”** é a variável local.

### **Criando os métodos de saque e deposito.**

Para que fique mais claro a aplicação do conceito de encapsulamento com o uso de métodos **“get”** e **“set”** vamos criar dois métodos fundamentais em nossa classe **“Conta”** que vão se valer do conceito de encapsulamento para realizar suas tarefas com a aplicação correta dos conceitos de O.O no JAVA.

Porem essas duas classes vão executar operações aritméticas, então vamos a uma rápida apresentação dos operadores aritméticos do JAVA.

Assim como toda linguagem de programação o JAVA possui operadores aritméticos que se dividem em dois grupo:

**Operadores binários:** possuem valores numericos a esquerda e a direita.

**Operadores unários:** possuem valores numericos somente a direita.

As tabelas abaixo ilustram esse dois grupos:

Operadores binários da linguagem JAVA	
Operador	Descrição
=	Atribuição
+	Soma
-	Subtração
/	Divisão
%	Modulo (resto de uma divisão)
Operadores unários da linguagem JAVA	
Operador	Descrição
+	Inversão de sinal (positivo)
-	Inversão de sinal (negativo)
++	Incremento
--	Decremento

### **Precedência dos operadores aritméticos da linguagem JAVA.**

A precedência define a ordem em que as operações matemáticas serão realizadas quando temos uma expressão plana com múltiplos operadores aritméticos, ou seja, a expressão não possui parenteses separando as operações umas das outras.

A tabela abaixo ilustra a precedencia dos operadores aritméticos em ordem decrescente:

Tabela de precedência dos operadores aritméticos do JAVA		
Prioridade	Operador	Descrição
1º	++, --	Da esquerda para direita na ordem em que aparecem.
2º	- (unário)	Da esquerda para direita na ordem em que aparece.
3º	*, /, %	Da esquerda para direita na ordem em que aparecem.
4º	+, -	Da esquerda para direita na ordem em que aparecem.

Na matemática temos as chaves, colchetes e parênteses que forçam a realização de uma operação de precedência menor ser realizada primeiro em relação a uma maior na programação essa ação é feita apenas pelos parênteses.

Exemplos:

$$10 * 10 + 50 - 10$$

O resultado da operação acima seria 140, primeiro seria resolvido a multiplicação pois essa tem prioridade sobre adição e multiplicação, logo,  $10 * 10$  é 100 em seguida temos uma operação de soma e outra de subtração, ambas tem a mesma prioridade então serão resolvidas da esquerda para a direita na ordem em que surgem,  $100 + 50$  que resulta em 150 e em seguida  $150 - 10$  resultando assim em um valor final de 140.

Porém poderíamos forçar operações de prioridade mais baixa em relação a de prioridades mais altas, observe:

$$10 * (10 + 50) - 10$$

No exemplo acima a soma entre 10 e cinquenta será realizada primeiro pois os parênteses forçam isso, observe:

$$10 * (10 + 50) - 10$$

$$10 * 60 - 10$$

$$600 - 10$$

$$590$$

A imagem abaixo ilustra a primeira versão dos métodos “sacar” e “depósito”:

```
10  [ ] public void saque(double valor){
11      [ ]
12      [ ]     setSaldo(getSaldo() - valor);
13      [ ]
14      [ ] }
15
16  [ ] public void deposito(double valor){
17      [ ]
18      [ ]     setSaldo(getSaldo() + valor);
19      [ ]
20      [ ] }
```

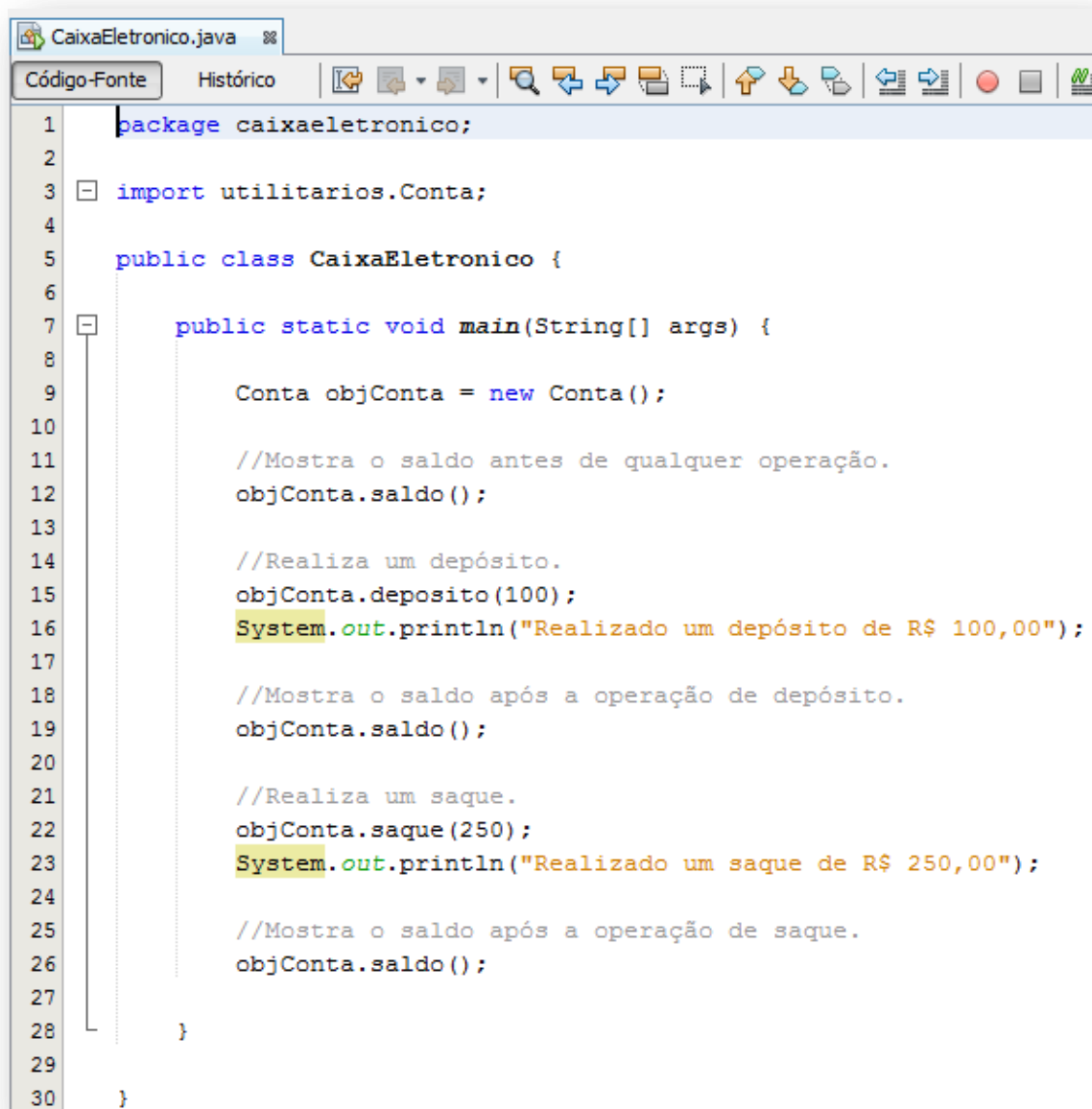


Mais uma vez vamos dissecar um dos métodos para entender suas funcionalidades, como podemos observar na linha dez (10) o método “saque” possui um modificador de acesso “**public**” seu tipo de retorno é “**void**” e um parâmetro do tipo “**double**” nomeador como valor é requerido.

Na linha 12 temos a implementação do método saque que consiste em subtrair a quantidade sacada pelo usuário do atributo “**saldo**”, lembre-se o atributo “**saldo**” tem sua visibilidade (modificador de acesso) “**private**” e só pode ser manipulado através de métodos, ou seja, o atributo está encapsulado. Como nós criamos um método chamado “**setSaldo**” que configura um valor no atributo saldo através de seu parâmetro requerido podemos passar um parâmetro que na verdade é o resultado de uma subtração do parâmetro do método “sacar” menos o valor do atributo “saldo” que deve ser recuperado através do método “**acessador**” “**getSlado**”, logo podemos codificar a seguinte linha “**setSaldo(getSaldo() - valor)**”, lembrando que nesse caso primeiro será realizada a operação de subtração e após isso o resultado será passado como parâmetro para o método “**setSaldo**”.

A mesma lógica é aplicada no método “**deposito**” o que muda é a operação que nesse caso é uma adição.

A imagem abaixo ilustra a utilização dos novos métodos na classe principal:



```
1 package caixaeletronico;
2
3 import utilitarios.Conta;
4
5 public class CaixaEletronico {
6
7     public static void main(String[] args) {
8
9         Conta objConta = new Conta();
10
11         //Mostra o saldo antes de qualquer operação.
12         objConta.saldo();
13
14         //Realiza um depósito.
15         objConta.deposito(100);
16         System.out.println("Realizado um depósito de R$ 100,00");
17
18         //Mostra o saldo após a operação de depósito.
19         objConta.saldo();
20
21         //Realiza um saque.
22         objConta.saque(250);
23         System.out.println("Realizado um saque de R$ 250,00");
24
25         //Mostra o saldo após a operação de saque.
26         objConta.saldo();
27
28     }
29
30 }
```