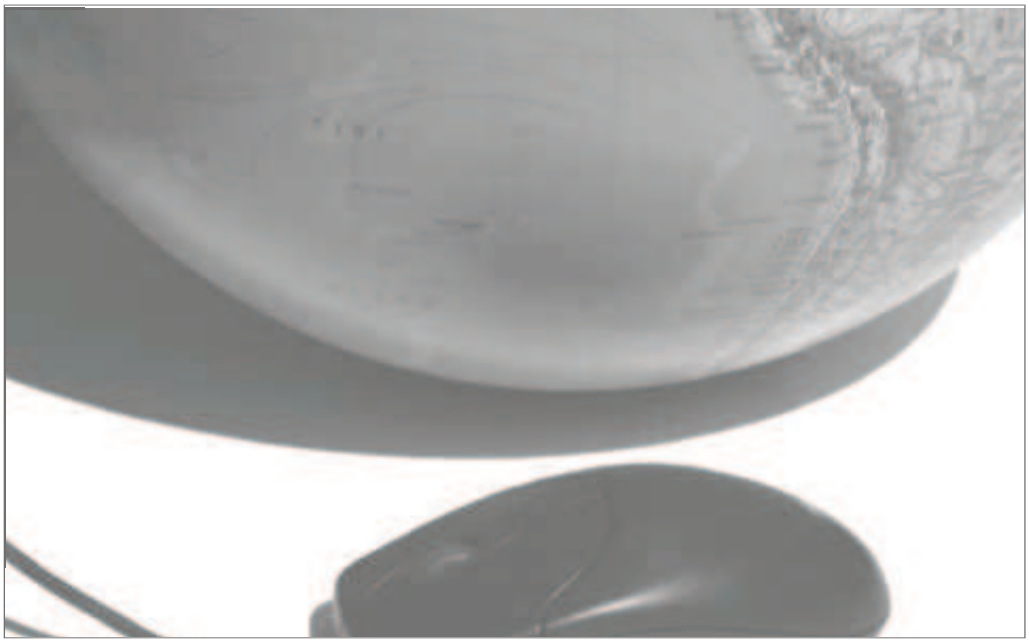
5

Orientação a Objetos: Métodos e Encapsulamento



Meta

Consolidar os conhecimentos referentes a Orientação a Objetos, além de introduzir os conceitos referentes a métodos e encapsulamento em Java.

Objetivos

Ao final desta aula, você deverá ser capaz de:

✓ Utilizar os conceitos no dia a dia do desenvolvimento de sistemas em Java.

1 – Orientação a Objetos: Métodos e Encapsulamento

Vamos relembrar alguns conceitos importantes da programação orientada a objetos aprendidos no capítulo anterior:

• **Classe**: É um tipo de objeto que pode ser definido pelo programador para descrever uma entidade real ou abstrata.

• **Objeto**: É uma instância de uma classe. Por exemplo, vamos imaginar uma classe chamada **Moto**. Cada moto existente é uma instância (ou um objeto) dessa classe. Apesar de parecidas, cada moto possui

características próprias, como cor, chassi, arranhões, etc.

• **Métodos** são comportamentos que os objetos de uma classe podem possuir. Eles definem as operações que podemos realizar com os objetos de uma classe.

Construtores:

Quando usamos a palavra "*new*", como fizemos no código da classe "Programa", estamos construindo um novo objeto. Sempre que isso é feito, o **construtor** da classe é executado. O construtor da classe é um bloco declarado com o mesmo nome que a classe. Veja um exemplo no código a seguir:

```
class Conta {
  int numero;
  String nome;
  double saldo;
  double limite;

  Conta () { // Aqui estamos declarando o construtor
    System.out.println("Nova conta criada.");
  }
```

No caso do exemplo acima, sempre que um novo objeto da classe Conta for criado, a mensagem "Nova conta criada" aparecerá na tela. É como uma rotina de inicialização que é chamada sempre que um novo objeto é criado. Apesar de parecer, um construtor não é um método. Até agora, as nossas classes não possuíam nenhum construtor. Então, o que acontecia, se o construtor da classe é sempre executado quando um novo objeto é criado? Quando você não declara algum construtor na sua classe, o Java cria um para você. Esse construtor é o **construtor padrão**, que não recebe nenhum argumento e cujo corpo é vazio. A partir do momento em que você declara um construtor, o construtor padrão não existe mais.

O interessante é que o construtor pode receber parâmetros, assim como um método. Para aprimorar o construtor da classe Conta, por exemplo, poderíamos receber o valor do "nome" do titular da conta como parâmetro.

Veja no trecho de código a seguir:

```
Conta (String nome) {
  this.nome = nome;
}
```

Agora não precisamos mais atribuir os valores a esses parâmetros após instanciarmos o objeto "minhaConta" da classe "Programa". Ao invés disso, passamos os valores como parâmetro para o construtor da classe Conta ao criarmos o objeto. Veja a nova classe "Programa" no código a seguir:

```
class Programa {
  public static void main (String[] args) {
    Conta minhaConta;
    minhaConta = new Conta("Hilário");

    minhaConta.saldo = 1000.0;
    /* Deixe aqui as demais operações que foram inseridas
no programa */
}
```

Se tudo estava funcionando até agora, para que utilizamos um construtor?

A idéia é bem simples: se toda conta precisa de um usuário titular, como obrigar todos os objetos que forem criados a ter um valor desse tipo? Basta criar um construtor que recebe essa String!

O construtor se resume a isso: dar possibilidades ou obrigar o usuário de uma classe a passar argumentos para o objeto durante o processo de criação do mesmo.

Por exemplo, não podemos abrir um arquivo para leitura sem dizer qual é o nome do arquivo que desejamos ler! Portanto, nada mais natural que passar uma String representando o nome de um arquivo

na hora de criar um objeto do tipo de leitura de arquivo, e que isso seja obrigatório.

Além disso, se criamos um construtor com vários parâmetros, o usuário não precisará criar várias sentenças ao longo do programa para atribuir os valores de cada um desses parâmetros.

Também é possível que criemos mais de um construtor em uma mesma classe, cada um com um número diferente de parâmetros. Quando o objeto for criado, o construtor apropriado será escolhido. Veja o exemplo:

```
class Conta {
  int numero;
  String nome;
  double saldo;
  double limite;

  Conta (String nome) {
    this.nome = nome;
  }

  Conta (String nome, double saldo) {
    this.nome = nome;
    this.saldo = saldo;
  }
}
```

A seguir, temos o exemplo de um trecho de código que cria dois objetos à classe **Conta**. O primeiro passa uma String e um *float* como parâmetros na criação do objeto, utilizando o segundo método construtor definido na classe acima. O segundo passa apenas uma String como parâmetro, utilizando o primeiro método construtor definido na classe:

```
Conta minhaConta = new Conta("Hilário", 1000.0);
Conta outraConta = new Conta("Joãozinho");
System.out.print("Usuário: " + minhaConta.nome);
System.out.println(" / Saldo: " + minhaConta.saldo);
System.out.print("Usuário: " + outraConta.nome);
System.out.println(" / Saldo: " + outraConta.saldo);
```

A saída do trecho de código acima é:

```
Usuário: Hilário / Saldo: 1000.0
Usuário: Joãozinho / Saldo: 0
```

Exercício 41 - Crie um construtor para a classe **Empresa**, que recebe como parâmetros um "nome" e um "cnpj" e o tamanho do *array* de funcionários. O atributo "qtde_de_funcionario" deve começar com o valor zero. Crie também um construtor para a classe **Empresa** que não receba o tamanho do *array* de funcionários. Neste caso, defina que o padrão é criá-lo com tamanho 100.

Exercício 42 - Crie um construtor para a classe **Funcionario** que receba os atributos "nome", "departamento", "salario" e RG. No atributo booleano, considere que o funcionário está empregado.

Crie também um construtor para a classe **Funcionario** que receba apenas nome e RG, e outro que não receba parâmetros.

Exercício 43 - Crie um objeto da classe Empresa. Em seguida, crie vários objetos da classe **Funcionario** e inclua-os na **Empresa.** Demita alguns e depois imprima a quantidade de funcionários da empresa utilizando o método "numeroDeFuncionarios".

Os **destrutores** são chamados logo antes do "coletor de lixo" (*garbage collector*, descrito no capítulo anterior) passar e sua memória ser liberada. O método destrutor é chamado de "finalize()". Cada objeto possui um método destrutor padrão, que não faz nada. Para criar um método destrutor para suas próprias classes, basta sobrepor o método "finalize()" com o seguinte cabeçalho:

```
protected void finalize () {
  // código necessário para limpeza do objeto;
}
```

Utilizando corretamente o método, devolvemos corretamente ao sistema o espaço de memória que sabemos que não será mais utilizado. Caso o destrutor não seja explicitamente chamado, o gerenciador automático

do Java irá destruir os objetos perdidos apenas quando o programa estiver ocioso (por exemplo, esperando uma por uma entrada de dados).

Para forçar a coleta de lixo em determinado momento, basta chamar o método "System.gc()".

Encapsulamento:

Em Java, existem atributos que modificam o nível de acesso às variáveis definidas em uma classe, aos métodos, aos construtores e às próprias classes. São eles: ***protected, private e public***.

- **Public** indica que uma classe, variável, método ou construtor podem ser usados externamente ao pacote em que foram definidos. Sem ele, o uso só pode ser feito internamente ao pacote no qual ocorre a definição.

- **Private** indica que uma variável, método ou construtor só pode ser utilizado internamente à classe em que foi declarado.

- **Protected** indica que o uso só pode ser feito na classe ou em uma subclasse.

Esses modificadores permitem a implementação do encapsulamento no Java. O **encapsulamento** faz com que não seja permitido acessarmos diretamente as propriedades de um objeto. Neste caso, precisamos operar sempre por meio dos métodos pertencentes a ele. A complexidade de um objeto é escondida, portanto, através da abstração de dados que estão "por trás" de suas operações.

ENCAPSULAMENTO consiste na separação entre os aspectos externos de um objeto, acessíveis por outros objetos.

Não entendeu?

É simples: Você não precisa, por exemplo, conhecer os detalhes dos circuitos internos do seu computador para utilizá-lo. A carcaça do computador ***encapsula*** *esses detalhes, provendo a você uma interface*

mais amigável (monitor, teclado, mouse etc) para utilizá-lo.

Um problema da classe **Conta** criada no capítulo anterior é que não há qualquer impedimento com relação a um saque que deixe a conta com saldo menor que o limite permitido. Na verdade, o método "sacar" já foi alterado de forma que essa verificação seja feita. Entretanto, nada garante que o usuário da classe **Conta** irá utilizar esse método **sempre** que precisar o valor do atributo "saldo". O código a seguir ultrapassa o limite diretamente:

```
Conta minhaConta = new Conta();
minhaConta.limite = 1000.0;
minhaConta.saldo = -3000.0; // saldo abaixo do limite!
```

A melhor forma de impedir que isso aconteça é forçar que o desenvolvedor seja sempre obrigado, por exemplo, a utilizar o método "sacar". Para fazer isso no Java, basta declararmos que os atributos não podem ser acessados fora da classe, utilizando o modificador de acesso *private*, da seguinte forma:

```
class Conta {
  int numero;
  String nome;
  private double saldo;
  private double limite;
  // métodos da classe...
```

Dessa forma, o exemplo anterior (que acessa diretamente o valor do atributo "saldo") não iria compilar. Cada classe é responsável por controlar seus atributos, portanto ela deve julgar se aquele novo valor é válido ou não. Essa validação não deve ser controlada por quem está utilizando a classe, e sim por ela mesma (centralizando essa responsabilidade e facilitando futuras mudanças no sistema).

Repare que, agora, quem chama o método "sacar" não faz a menor idéia de que existe um limite que está sendo checado. Para quem for usar essa classe, basta saber **o que** o método faz e não exatamente **como** ele o faz (**"o que"** um método faz é sempre mais importante do que **"como"** ele faz: mudar a implementação é fácil, já mudar a assinatura de um método vai gerar problemas).

A palavra-chave *private* também pode ser usada para modificar o acesso a um método. Tal funcionalidade é normalmente utilizada quando existe um método apenas auxiliar para a própria classe e não queremos que outras pessoas o utilizem (ou apenas para seguir a boa prática de se expor ao mínimo). Sempre devemos expor o mínimo possível de funcionalidades, para criar um baixo **acoplamento** entre as nossas classes.

O que vimos acima é a idéia de encapsular, isto é, esconder todos os membros de uma classe, além de esconder como funcionam as rotinas (métodos) do nosso programa. Encapsular é fundamental para que seu sistema seja suscetível a mudanças: não precisaremos mudar uma regra de negócio em vários lugares, mas sim em apenas um único lugar, já que essa regra está encapsulada (veja o caso do método "sacar").

Quando você dirige um carro, o que importa são os pedais e o volante, e não o motor que você está usando. É claro que um motor diferente pode lhe dar melhores resultados, mas ***o que*** *ele faz é a mesma coisa que um motor menos potente, a diferença está em* ***como*** *ele faz. Para trocar um carro a álcool por um a gasolina você não precisa reaprender a dirigir!*

Todos os celulares fazem a mesma coisa: eles possuem maneiras (métodos) de discar, ligar, desligar, atender, etc. O que muda é como eles fazem cada funcionalidade, mas repare que para o usuário comum pouco importa se o celular é GSM ou CDMA, isso fica encapsulado.

Como vimos anteriormente, além do *private* existe o modificador *public*, que permite a todos acessarem um determinado atributo ou método. Quando não declaramos o modificador de acesso de um atributo ou método, ele é definido como *protected*, uma espécie de estado intermediário entre *private* e *public*. Um atributo ou método *protected* só pode ser acessado (visível) pela própria classe ou suas subclasses.

Exercício 44 - Adicione o modificador de visibilidade (private, se necessário) para cada atributo e método da classe **Funcionario**. Tente criar um **Funcionario** no "main" e modificar ou ler um de seus atributos privados. O que acontece?

Exercício 45 - Crie métodos que alteram e que retornam cada atributo da classe **Funcionario**. Exemplo:

```
void setSalario (double salario) {
    this.salario = salario;
}
void getSalario () {
    return this.salario;
}
```

Modifique os códigos que acessavam atributos da classe Funcionario e faça com que eles passem a utilizar as funções acima.

Exemplo:
f.salario = 100;
System.out.println(f.salario);
passa a ser:

f.setSalario(100);
System.out.println(f.getSalario());

Exemplo de Código

Segue abaixo o código de um programa que guarda os dados digitados por um usuário e retorna os valores.

- **Pessoa.java**

Nesta classe definimos os atributos como **privados** usando ***private***. Depois criamos os métodos *set* e *get* para cada atributo da classe, lembrando que o primeiro nos permite definir um valor para o atributo, enquanto o segundo retorna o valor do atributo.

Da mesma forma que nos métodos construtores mais simples usados nos códigos das aulas anteriores, o **método set** usa o *this* para diferenciar o atributo de classe do parâmetro que será passado. Já o **método get** usa o *return*, que serve justamente para **retornar o valor**.

Podemos também observar que no método *set* foi usado o *void*, já que o método *set* não retornará nenhuma valor, apenas servirá para definir um. O método *get* no exemplo abaixo foi construido usando String, já que o atributo usado é uma String e, por isso, retornará String.

```
public class Pessoa {
    private String nome;
    private String telefone;
    private String endereco;

    public void setNome(String nome) { this.nome = nome; }
    public String getNome() { return nome; }

    public void setTelefone(String telefone) { this.telefone = telefone; }
    public String getTelefone() { return telefone; }

    public void setEndereco(String endereco) { this.endereco = endereco; }
    public String getEndereco() { return endereco; }
}
```

Main.java

O código abaixo, é da **classe principal**, que será a primeira a ser executada. Como podemos perceber, ela é feita praticamente da mesma forma que as classes principais dos exemplos mostrados nas aulas anteriores. A diferença aqui é o **uso dos métodos *set* e *get* para definirmos e lermos os valores**.

Começamos dando o *import* necessário para usarmos o *JOptionPane* e depois **declaramos e instanciamos o objeto** da classe Pessoas para podermos usar seus **métodos *set* e *get***.

Podemos ver que usamos o **método set** da seguinte forma:

```
objPessoa.setNome(JOptionPane.showInputDialog("Digite o nome"));
```

Isso fez com que aparecesse uma caixa de diálogo (*JOptionPane.showInputDialog*) pedindo para o usuário digitar um nome e depois esse nome foi passado pelo método set para o atributo nome da classe Pessoa. Ou seja, o valor dentro dos parentes do método set é o que foi definido. Uma forma de passar o valor direto no código, sem a **interação do usuário**, seria:

```
objPessoa.setNome("Marcos Penha");
```

Depois usamos o **método get** sem parâmetros adicionais para retornar o valor. O *JOptionPane.showMessageDialog* mostra uma caixa de diálogo com uma mensagem específica. No código ele foi colocado com os métodos get justamente para **mostrar a caixa de mensagens com os valores digitados pelo usuário**.

```
import javax.swing.JOptionPane;

public class Main {
    public static void main(String[] args) {
        Pessoa objPessoa = new Pessoa(); //declaração e instanciação de objeto

        objPessoa.setNome(JOptionPane.showInputDialog("Digite o nome"));
        objPessoa.setTelefone(JOptionPane.showInputDialog("Digite o telefone"));
        objPessoa.setEndereco(JOptionPane.showInputDialog("Digite o endereço"));

        JOptionPane.showMessageDialog(null, "Nome: " + objPessoa.getNome() +
                "\nTelefone: " + objPessoa.getTelefone() +
                "\nEndereço: " + objPessoa.getEndereco());
    }
}
```

O *NetBeans* possui uma funcionalidade que realiza o encapsulamento dos dados para nós. Para isso, clique com o botão direito sobre a classe, escolha a opção "Refatorar" e depois clique em "Encapsular campos".
Aparecerá uma janela semelhante à mostrada na Figura 24:

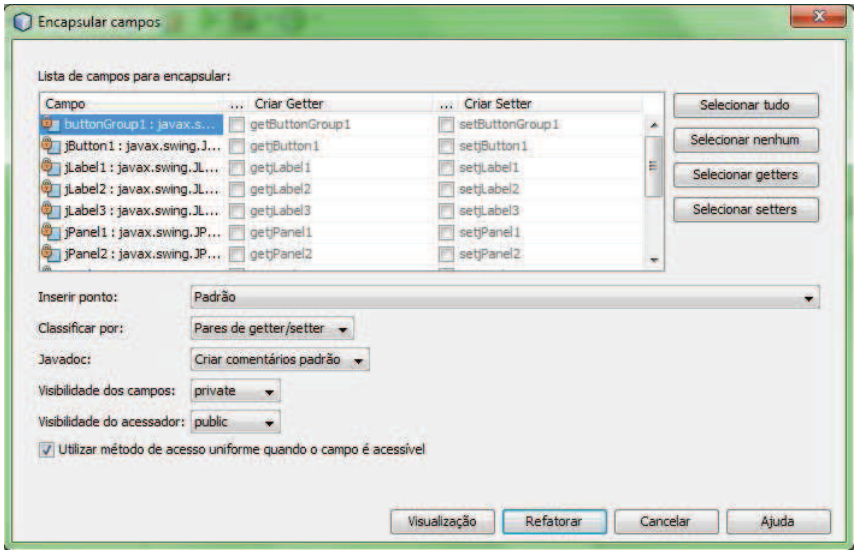


Figura 24: Encapsulamento dos dados

Clique no botão "Selecionar tudo" para que sejam criados todos os métodos que modificam e os que retornam os atributos da classe. Em seguida, clique em "Refatorar". Veja que o código da classe foi automaticamente modificado pelo *NetBeans*. Os atributos agora possuem o modificador "*private*" e já foram criados os métodos para manipularmos nossos dados.