TEMA 9 – ENCAPSULAMENTO E OS MÉTODOS MODIFICADORES E ASSESSORES

Encapsulamento

O processo de vincular **dados e métodos correspondentes (comportamento) juntos em uma única unidade** é chamado de encapsulamento em Java. Em outras palavras, o encapsulamento é uma técnica de programação que **une os membros da classe (variáveis e métodos) e evita que sejam acessados por outras classes**.

Assim, podemos **manter variáveis e métodos protegidos de interferências externas e uso indevido**.

Cada **classe Java é um exemplo de encapsulamento** porque escrevemos tudo dentro da classe apenas o que vincula variáveis e métodos e esconde sua complexidade de outras classes.

Outro exemplo de encapsulamento é uma cápsula. Basicamente, a cápsula encapsula várias **combinações de medicamentos**. Se as combinações de medicamentos são variáveis e métodos, a cápsula funcionará como uma classe e todo o processo é denominado Encapsulamento.

Na técnica de encapsulamento, **declaramos os campos como privados** na classe para evitar que outras classes os acessem diretamente. Os dados encapsulados necessários podem ser **acessados usando os métodos getter e setter públicos** (serão explicados na sequência).

Se o **campo for declarado privado na classe, ele não poderá ser acessado por ninguém de fora da classe** e oculta o campo dentro da classe. Portanto, também é chamado de **ocultação de dados**.

Por exemplo:

A **mochila escolar é um exemplo de encapsulamento**. Ela pode guardar nossos livros, canetas etc.

Outro exemplo é a validação de acesso. Quando você faz login em suas contas de e-mail, como Gmail, por exemplo, há muitos processos internos ocorrendo no back-end e você não tem controle sobre eles.

Quando você insere a senha para registro, eles são recuperados de forma criptografada e verificados, e então você recebe acesso à sua conta. Você não tem controle sobre como a senha foi verificada.

Assim, ele **mantém nossa conta protegida contra uso indevido**.

Como implementar encapsulamento em Java?

Existem duas formas pelas quais podemos alcançar ou implementar o encapsulamento em JAVA:

1. Declarando a variável de instância da classe como privada. para que não possa ser acessado diretamente por ninguém de fora da classe.
2. Fornecendo os métodos setter e getter públicos na classe para acessar e/ou modificar os valores das variáveis.

Vantagens do encapsulamento em Java

- O **código encapsulado é mais flexível e fácil** de alterar com novos requisitos.
- Impede que outras classes acessem os campos privados.
- O encapsulamento **permite modificar o código implementado sem quebrar outro código que o implementou**.

• Ele mantém os dados e códigos protegidos de herança externa. Assim, o encapsulamento **ajuda a alcançar a segurança**.
• Melhora a capacidade **de manutenção da aplicação**.
• Se você não definir o método setter na classe, os campos podem se tornar somente leitura.
• Se você não definir o método getter na classe, os campos podem ser feitos somente para gravação.

Desvantagem do encapsulamento em Java

A principal desvantagem do encapsulamento em Java é que **aumenta o comprimento do código e atrasa o fim da execução**.
Vamos entender alguns programas de exemplo baseados em encapsulamento em Java.

Classe Estudante:

```
public class Estudante {
    private String nome;

    public String getNome() {
        return nome;
    }

    public void setNome(String nomeEstudante) {
        this.nome = nomeEstudante;
    }
}
```

Programa principal:

```
public class Tema9Estudante {

    public static void main(String[] args) {
        Estudante e1 = new Estudante();

        // Para atribuir um nome ao estudante, use o setter, pois 'nome' é privado
        e1.setNome("Joaozinho");

        // Para obter o nome do estudante, use o getter
        String nomEstudante = e1.getNome();
        System.out.println(nomEstudante);
    }
}
```

A saída será:

```
Joaozinho
```

Modificadores e Assessores

Getters e setters são conhecidos como **métodos assessores (getters)** e **modificadores (setters)**, usados para proteger os dados do código, principalmente ao criar classes. Para cada variável de instância, um **método getter retorna seu valor** enquanto um **método setter define ou atualiza seu valor**. Por isso, getters e setters também são conhecidos

como **assessores e modificadores, respectivamente**.
Por convenção, getters **começam** com a palavra **"get"** e setters com a palavra **"set"**, seguido por um nome de variável. Em ambos os casos, **a primeira letra do nome da variável é maiúscula**, por exemplo:

```
public class Veiculo {
    private String cor;

    // Getter
    public String getCor() {
        return cor;
    }

    // Setter
    public void setCor(String c) {
        this.cor = c;
    }
}
```

O método getter retorna o valor do atributo. O método setter pega um parâmetro e o atribui ao atributo.
Uma vez que getter e setter foram definidos, podemos usá-los no nosso programa principal:

PROGRAMA PRINCIPAL (Suponho que isso estaria dentro de uma outra classe, talvez Main ou TesteVeiculo):

```
public class TesteVeiculo {
    public static void main(String[] args) {
        Veiculo v1 = new Veiculo();
        v1.setCor("Verde");
        System.out.println(v1.getCor());
    }
}

// A saída será "Verde"
```

Getters e setters permitem controle sobre os valores. Você pode validar o valor fornecido no configurador antes de definir o valor.

É obrigatório usar getters e setters?

Não, mas é uma boa prática. Os getters e setters permitem controlar como variáveis importantes são acessadas e atualizadas no código. Por exemplo, considere este método setter:

```
public void setNum(int num) {
    if (num < 1 || num > 10) {
        System.out.println("Erro do valor numérico");
    }
    this.num = num;
}
```

Ao usar o método **setNum**, você pode ter certeza de que o valor de num está **sempre entre 1 e 10**. Isso é muito melhor do que atualizar a variável num diretamente, como por exemplo:

```
obj.num = 13;
```

Se você atualizar num diretamente, é possível que cause efeitos colaterais indesejados em algum outro lugar do seu código. Aqui, definir num como 13 viola a restrição de 1 a 10 que queremos estabelecer. **Criar num como uma variável privada e usar o rmétodo setNum evitaria que isso acontecesse**. Por outro lado, a única maneira de ler o valor de num é usando um método getter:

```
public int getNum() {
   return this.num;
}
```

Resumindo:

1. Se você definir apenas o **método getter, ele pode se tornar somente leitura**.

2. Se você definir apenas o **método setter, ele pode ser feito somente para gravação**.

3. Se você definir os **métodos getter e setter, eles poderão ser utilizados para leitura e gravação**.