Programação Orientada a Objetos

Módulo 1

Vaux Gomes [1]

[1]Instituto Federal de Educação, Ciência e Tecnologia do Ceará
Campus Jaguaribe

7 de Agosto de 2022

Sumário

Modificadores de acesso

Os **modificadores de acesso** são padrões de visibilidade de acessos às **classes**, **atributos** e **métodos**.

- **public, private, protected, package-private**
- Os modificadores de acesso são palavras reservadas no Java.
- Palavras reservadas não podem ser usadas como nome de métodos, classes ou atributos.

Modificadores de acesso

Public, Private, Protected

- **public**: Os membros da classe podem ser acessados de qualquer lugar e por qualquer entidade (objeto) que possam visualizar a classe a que ela pertence.
- **private**: Os membros da classe não podem ser acessados ou usados por nenhuma outra classe.
  - Esse modificador **não se aplica às classes**, somente para seus métodos e atributos.
  - Esses atributos e métodos também não podem ser visualizados pelas classes herdadas.

Modificadores de acesso

Public, Private, Protected

- **protected**: Torna o membro acessível às classes do mesmo pacote ou através de herança.
  - Esse modificador **não se aplica às classes**, somente para seus métodos e atributos.
  - Seus membros herdados não são acessíveis a outras classes fora do pacote em que foram declarados[1].
- **package-private [ou default (padrão)]**: A classe e/ou seus membros são acessíveis somente por classes do mesmo pacote, na sua declaração não é definido nenhum tipo de modificador, sendo este identificado pelo compilador.

Modificadores de acesso

Public, Private, Protected

Atenção

Como boas práticas do Java, a maioria das declarações de atributos são definidos com a palavra-chave **private**, para garantir a **segurança** de alterações acidentais, sendo somente acessíveis através dos métodos.

Essa ação tem como efeito ajudar no encapsulamento dos dados, preservando ainda mais a segurança e a aplicação de programação orientada a objetos do Java.

Modificadores de acesso

Public, Private, Protected

	private	default	protected	public
Mesma Classe	×	×	×	×
Mesmo Pacote		×	×	×
Pacotes Diferentes (Subclasses)			×	×
Pacotes Diferentes (Sem Subclasses)				×

Tabela 1: Tabela dos modificadores de acesso

[1] Retomaremos esse ponto quando falarmos de herança.

Modificadores de acesso

Exemplos

Exemplo 1

No exemplo abaixo:

- Declaramos uma classe Carro usando o modificador de acesso *default* (padrão) para os atributos; e
- Instanciamos um objeto dentro do método *main* em outra classe, mas no mesmo pacote da classe Carro.

Modificadores de acesso

Exemplos

```java
// Carro.java
package ifce.jbe.poo.carro;

public class Carro {
    String modelo;
    int ano;
    double combustivel;
    int kms;
}
```

```java
// Main.java
package ifce.jbe.poo.carro;
...
public static void main(String args[]) {
    Carro c = new Carro();
    c.modelo = "Fusca";
    c.ano = 1960;
    c.combustivel = 1;
    c.kms = 500000;

    // Imprime Fusca
    System.out.println(c.modelo);
}
```

Figura 1: Exemplo: Apenas modificadores *default* no mesmo *package*.

Modificadores de acesso

Exemplos

Exemplo 2

No exemplo abaixo:

- Declaramos uma classe Carro usando usando diversos modificadores de acesso para os atributos; e
- Instanciamos um objeto dentro do método *main* em outra classe, mas no mesmo pacote da classe Carro.

Modificadores de acesso

Exemplos

```java
package ifce.jbe.poo.carro;

public class Carro {
    String modelo;
    public int ano;
    protected double combustivel;
    private int kms;
}
```

```java
package ifce.jbe.poo.carro;
...
public static void main(String args[]) {
    Carro c = new Carro();
    c.modelo = "Fusca";
    c.ano = 1960;
    c.combustivel = 1;
    c.kms = 500000;

    System.out.println(c.modelo);
}
```

Figura 2: Exemplo: Modificadores diversos no mesmo *package*.

Modificadores de acesso

Exemplos

Exemplo 3

No exemplo abaixo:

- Declaramos uma classe Carro usando usando diversos modificadores de acesso para os atributos; e
- Instanciamos um objeto dentro do método *main* em outra classe e em outro *package*.

Modificadores de acesso

Exemplos

```java
// Carro.java
package ifce.jbe.poo.carro;

public class Carro {
    String modelo;
    public int ano;
    protected double combustivel;
    private int kms;
}
```

```java
// Main.java
package ifce.jbe.poo.prova;
...
public static void main(String args[]) {
    Carro c = new Carro();
    c.modelo = "Fusca";
    c.ano = 1960;
    c.combustivel = 1;
    c.kms = 500000;

    System.out.println(c.modelo);
}
```

Figura 3: Exemplo: Modificadores diversos em *packages* diferentes.

Modificadores de acesso

Exemplos

Exemplo 4

No exemplo abaixo:

- Declaramos uma classe Carro usando usando diversos modificadores de acesso para os atributos; e
- Instanciamos um objeto dentro do método *main* dentro da classe Carro.

Modificadores de acesso

Exemplos

```java
// Carro.java
package ifce.jbe.poo.carro;

public class Carro {
    String modelo;
    public int ano;
    protected double combustivel;
    private int kms;

    public static void main(String args[]) {
        Carro c = new Carro();
        c.modelo = "Fusca";
        c.ano = 1960;
        c.combustivel = 1;
        c.kms = 500000;

        System.out.println(c.modelo);
    }
}
```

Modificadores de acesso

Exemplos

Figura 4: Exemplo: Classe Carro com função *main*.

Modificadores de acesso

Exemplos

Exemplo 5

No exemplo abaixo:

- Declaramos uma classe Carro com modificador *default*
- Instanciamos um objeto dentro do método *main* em outra classe no mesmo *package* e em outro *package*.

Modificadores de acesso

Exemplos

```java
// Carro.java
package ifce.jbe.poo.carro;

class Carro {
    String modelo;
    public int ano;
    protected double combustivel;
    private int kms;
}
```

```java
// Main.java
package ifce.jbe.poo.carro;
...
public static void main(String args[]) {
    Carro c = new Carro();
}

// ****************

// Main.java
package ifce.jbe.poo.prova;
...
public static void main(String args[]) {
    Carro c = new Carro();
}
```

Figura 5: Exemplo: Modificador *defaul* na classe.

Encapsulamento

Em programação orientada a objetos, **encapsular** significa separar o programa em partes, o mais **isolado** possível. A ideia é **tornar o software mais flexível**, e **fácil de modificar**.

- O Encapsulamento serve para controlar o acesso aos atributos e métodos de uma classe.
- É uma forma eficiente de proteger os dados manipulados dentro da classe, além de determinar onde esta classe poderá ser manipulada.
- Utilizam-se os modificadores de acesso.

Modificadores de acesso e Encapsulamento

Encapsulamento

- É fácil encontrarmos casos onde não seria sábio deixar determinado atributo com acesso público.
- Por exemplo, não seria bom deixar que qualquer parte do código modificasse os valores dos atributos da classe abaixo[2].

```
class ResumoPedido {
    double valorPedido;
    String comprador;
    String enderecoComprador;
}
```

[2] Não por causa da natureza (financeira) do programa, mas porque a maioria dos programas utilizam regras de negócio específicas e para mantermos e implementarmos estas regras é mais fácil usar encapsulamento de código.

Encapsulamento

Métodos Get e Set

Quando um atributo não é acessível a outras classes, mas ainda assim precisamos modificá-lo ou utilizar o seu valor fora da sua classe de origem é necessário criarmos métodos que realizem estas ações:

- **Método Get:** Realiza a ação de **buscar** o valor do atributo;
- **Método Set:** Realiza a ação de **atribuir** um valor ao atributo.

Encapsulamento

Métodos Get e Set

- A nomenclatura dos métodos segue um mesmo padrão.
- Muitas vezes o IDE facilita a escrita.
- Resta a nós sabermos **como**, **quando** e **quando não** usá-los.

```
class ResumoPedido {
    private double valorPedido;
    private String comprador;
    private String enderecoComprador;

    /** Get de Valor do Pedido **/
    public double getValorPedido() {
        return this.valorPedido;
    }

    /** Set de Valor do Pedido **/
    public void setValorPedido(double valorPedido) {
        this.valorPedido = valorPedido;
    }

    // ...
}
```

Encapsulamento

Métodos Get e Set

- Da maneira que definimos a classe `ResumoPedido` podemos, facilmente e desreguladamente, modificar o valor do pedido quando um novo produto for adicionado ao carrinho de compras. Por exemplo:

```
ResumoPedido p = new ResumoPedido();

// Produto a ser adicionado ao carrinho
double valorProduto = ???;

// Adicionar valor do produto no valor do pedido
p.setValorPedido(p.getValorPedido() + valorProduto);
```

- Agora imagine que a regra de negócio do pedido mude: todo item comprado ganha desconto de 5% se o valor dele for superior a 300 reais.

Encapsulamento

Métodos Get e Set

- Implementar essa mudança não será tarefa fácil **pois precisaríamos fazê-la em diferentes classes do sistema**.

```
ResumoPedido p = new ResumoPedido();

// Produto a ser adicionado ao carrinho
double valorProduto = ???;

// Adicionar valor do produto no valor do pedido
if (valorProduto < 300) {
    p.setValorPedido(p.getValorPedido() + valorProduto);
} else {
    p.setValorPedido(p.getValorPedido() + valorProduto * 0.95);
}
```

- O desenho da classe `ResumoPedido` não está bom [3].

Encapsulamento

Métodos Get e Set

- Para resolvermos o nosso problema vamos adicionar um novo método chamado `adicionaValor`. Assuma também que `setValorPedido` já não existe ou está com acesso privado.

Encapsulamento

Métodos Get e Set

```
class ResumoPedido {
    // ...
    public void adicionaValor(double valorProduto) {
        if (valorProduto < 300) {
            setValorPedido(getValorPedido() + valorProduto);
        } else {
            setValorPedido(getValorPedido() + valorProduto * 0.95);
        }
    }
    // ...
}

//...
ResumoPedido p = new ResumoPedido();

// Produto a ser adicionado ao carrinho
double valorProduto = ???;

// Adicionar valor do produto no valor do pedido
p.adicionaValor(valorProduto);
```

Encapsulamento

Métodos Get e Set

- Assim, usaremos o **set** para implementar algumas regras de negócio quando for necessário.
- Ainda assim devemos ter cuidado com o acesso que damos para modificar os valores das variáveis dentro dos objetos.

```
// Talvez private ou protected fosse melhor
public setValorPedido(double valorPedido) {
    if (valorPedido < 0) {
        // Neste ponto podemos avisar ao user que existe algo errado na entrada do valor
        this.valorPedido = 0;
    } else {
        this.valorPedido = valorPedido;
    }
}
```

[3] Deixamos o atributo `valorPedido` como privado, mas continuamos dando acesso a ele por meio do método `setValorPedido`

Exercício

Parte

1. Adicione o pacote `ifce.jbe.poo.atm` ao projeto `POOJBE201901`
2. Implemente a classe Conta com os seguintes atributos e métodos
   - **Atributos**: usuario, saldo, limiteEspecial
     - **usuario** é o nome do dono da conta
     - O **limiteEspecial** é o valor máximo negativo de saldo do usuário. Padrão -200. Este valor não poderá ser positivo.
   - **Métodos**: saque, deposito, transferencia, verificarSaldoFuturo
     - **saque**: Não permitir saque que deixe o saldo abaixo do limiteEspecial ou se o valor passado for menor ou igual a zero. Não permita saques se saldo estiver negativo.
     - **deposito**: Não permitir de valor menor ou igual a zero.
     - **tranferencia**: Receber 1 contas e um valor e faz a transferência usando os métodos **deposito** e **saque**. Caso os usuários sejam diferentes, cobre uma taxa de 5 da conta que está transferindo e informe por meio de mensagem.
     - **verificarSaldoFuturo**: Caso o saldo seja positivo, retornar saldo + 0.3%; caso o saldo seja negativo, retornar saldo - 0.1%.
     - **OBS**: Implemente todos os métodos set e get como público

Exercício

Parte

- **OBS**: Escreva mensagens para todos os resultados dos métodos.
- Sobrescreva o método toString() para exibir o estado da conta. Imprima todas as variáveis com legenda e o saldo futuro. Imprima um alerta caso a conta esteja negativa.

3. Crie uma classe Main com um método main no mesmo pacote.
4. Crie três Contas (duas com o mesmo usuário)
5. Teste/Quebre o seu programa (Deixe o código bem comentado)
   - Imprima os dados das contas
   - Faça saque positivo
   - Faça saque negativo
   - Faça saque que ultrapasse o limite especial
   - Faça um saque que deixe o saldo negativo e em seguida tente fazer outro saque
   - Faça depósito positivo
   - Faça depósito negativo
   - Faça depósito de valor zero
   - Use set para mudar o valor do limiteEspecial para um valor positivo

Exercício

Parte

- Faça transferência de valores negativos
- Faça transferência com contas de mesmo dono
- Faça transferência de modo que o saldo da conta ultrapasse o limiteEspecial
- Crie uma nova função chamada passarMes que substitui o valor do saldo pelo valor retornado em `verificarSaldoFuturo` e cobra uma taxa de 10 do saldo restante.