

## I Apresentação

---

Nesta aula, estudaremos o conceito de encapsulamento, que permite uma classe encapsular atributos e métodos, ocultando os detalhes de implementação dos objetos. Trabalharemos também os tipos de visibilidade de membros de uma classe: public, protected, private e package. Desenvolveremos uma aplicação utilizando o conceito de encapsulamento em conjunto com os conceitos de herança e agregação.

## Objetivo

- Enumerar as vantagens da utilização da técnica de encapsulamento;
- Demonstrar a aplicação do conceito de encapsulamento.

## Primeiras palavras

No desenvolvimento de aplicações, temos situações nas quais a segurança é muito importante. Em muitas situações os membros de uma classe (atributos e métodos) precisam ter o seu acesso restringido para que não sejam burlados por meio das aplicações. **Esse processo de limitação de acesso aos membros de uma classe é chamado de Programação Orientada a Objetos de Encapsulamento.**

“Encapsulamento trata-se de um mecanismo que possibilita restringir o acesso a variáveis e métodos da classe (ou até à própria classe). Os detalhes de implementação ficam ocultos ao usuário da classe, isto é, o usuário passa a utilizar os serviços da classe sem saber como isso ocorre internamente. Somente uma lista das funcionalidades existentes torna-se disponível ao usuário da classe.”

(FURGERI, 2015)

Encapsulamento é o processo de separação dos membros de uma classe através da restrição ao seu acesso. Pode ocultar os atributos e métodos de uma classe, evitando que dados e detalhes de implementação de métodos sejam vistos (acessados diretamente) pela aplicação ou outras classes. Uma classe encapsula atributos e métodos, ocultando os detalhes de implementação dos objetos. Como um dos princípios do desenvolvimento orientado a objetos, o encapsulamento determina que a implementação de um objeto somente deve ser acessada através de uma interface visível e bem definida.

Os atributos não devem ser manipulados diretamente, podendo ser alterados ou consultados somente através dos métodos de acesso (**Setters e Getters**) do objeto. Ao restringir o acesso direto aos atributos de uma classe, evitamos que eles sejam manipulados diretamente pela aplicação ou outras classes, não permitindo que possam receber um valor qualquer, principalmente, valores inválidos para o seu contexto. Dessa forma, aumentamos a segurança e a confiança sobre os valores atribuídos.

Por exemplo:

```
public class Exemplo {  
    int idade;
```

#### Na aplicação:

```
public class AppEncapsulamento1 {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Exemplo exemplo = new Exemplo();  
        exemplo.idade = -20;  
        System.out.println("Idade = " + exemplo.idade);
```

#### Saída:

```
Idade = -20
```

**1** Como não temos restrição sobre o atributo Idade, a aplicação poderá realizar um acesso direto ao atributo;

---

**2** Na aplicação foi realizado um acesso direto ao atributo, sem o uso de um método de acesso (*Setter*), e o valor atribuído não é válido porque uma pessoa não pode ter idade negativa;

---

Métodos podem ter sua visibilidade restrita para evitar que detalhes de implementação ou um possível uso indevido possam ser realizados por outras classes ou aplicações. É muito comum que um método aparentemente simples, tal como **calcularImposto(double valor, tipoProduto)** não se restrinja a um simples cálculo. Ao se definir o tipoProduto, podemos ter diversas e diferentes formas de calcular o imposto sobre esse produto. Podemos então criar um método principal **calcularImposto()** e, através dele, chamar diversos outros métodos, cada um responsável por calcular o imposto para um determinado tipo de produto.

Realizar o acesso diretamente aos métodos chamados por **calcularImposto()** pode exigir muito conhecimento por parte do desenvolvedor que irá usar a classe, mas nem sempre este desenvolvedor terá os conhecimentos técnicos necessários. Provavelmente, houve o apoio de um especialista em processos fiscais para que o desenvolvedor original da classe pudesse desenvolver os métodos previstos. Para evitar o uso indevido de métodos que possam ser usados de forma equivocada ou com restrição de segurança, alteramos a visibilidade desses métodos.

Por exemplo:

#### A classe Tributos:

```
public class Tributos {
    double imposto;

    public double calcularImposto( double preco, int tipoProduto){
        if(tipoProduto < 10) { // tipo 0
            // exemplo: 01, tipo 0, faixa 0
            // chamada método do tipo 0 e faixa: 00 = 0
            // se tipo = 0, faixa será o valor de tipoProduto
            imposto = calcularTipoProduto0(preco, tipoProduto);
        }
        else {
            if(tipoProduto < 20) { // tipo 1
                // se tipo = 1,
                // faixa será o valor de tipoProduto - 10
                // exemplo: 11, tipo 1, faixa 1
                // chamada método do tipo 1 e faixa: 11 - 10 = 1
                imposto = calcularTipoProduto1(preco, tipoProduto - 10);
            }
            else {
                // se tipo = 2,}
                // faixa será o valor de tipoProduto - 20
                // exemplo: 22, tipo 2, faixa 2
                // chamada método do tipo 1 e faixa: 22 - 20 = 2
                imposto = calcularTipoProduto2(preco, tipoProduto - 20);
            }
        }
        return imposto;
    }

    public double calcularTipoProduto0(double valor, int faixa) {
        // verificar as faixas e realizar o cálculo correto
        if(faixa == 0) {
            imposto = valor * 4.5 / 100;
        }
        else {
            if(faixa == 1) {
```

```

        imposto = valor * 5.5 / 100;
    }
    else {
        imposto = valor * 4.5 / 100;
    }
}
return imposto;
}

public double calcularTipoProduto1(double valor, int faixa) {
    // verificar as faixas e realizar o cálculo correto
    if(faixa == 0) {
        imposto = valor * 12.5 / 100;
    }
    else {
        if(faixa == 1) {
            imposto = valor * 12.8 / 100;
        }
        else {
            imposto = valor * 13.1 / 100;
        }
    }
    return imposto;
}

public double calcularTipoProduto2(double valor, int faixa) {
    // verificar as faixas e realizar o cálculo correto}
    if(faixa == 0) { imposto = valor * 25.8 / 100; }
    else {
        if(faixa == 1) {
            imposto = valor * 26.3 / 100;
        }
        else {
            imposto = valor * 28.1 / 100;
        }
    }
    return imposto;
}
}
}

```

### Na aplicação:

```

import java.util.Scanner;

public class Encapsulamento2 {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Scanner entrada = new Scanner(System.in);
        double preco = 0, valorPagar = 0;
        System.out.println("Digite o preco:");
        preco = Double.parseDouble(entrada.nextLine()); // no teste preco = 1000
        Tributos tributo = new Tributos();
        // alterando o atributo imposto diretamente:
        System.out.println("");
        tributo.imposto = -20;
        System.out.println("Imposto incorreto:");
        System.out.println("Valor do imposto [atribuição direta] = " +
            tributo.imposto);
        // alterando o método de cálculo do imposto com o método do tipo 00:
        System.out.println("");
        System.out.println("Imposto correto, mas método errado");
    }
}

```

```

System.out.println("A faixa estava correta por acaso, porque 00 = 0");
System.out.println("Valor do imposto[método errado 00]= "
    + tributo.calcularTipoProduto0(preco, 00));
// alterando o método de cálculo do imposto com o método do tipo 11:
System.out.println("");
System.out.println("Imposto e método errados");
System.out.println("Valor do imposto [método errado 11]= "
    + tributo.calcularTipoProduto1(preco, 11));
// alterando o método de cálculo do imposto com o método do tipo 22:
System.out.println("");
System.out.println("Imposto correto, por acaso, porque 22 foi para a última faixa");
System.out.println("Valor a pagar [método errado 22]= "
    + tributo.calcularTipoProduto2(preco, 22));
// usando o método principal de cálculo do imposto com o método do tipo 00:
System.out.println("");
System.out.println("Imposto e método corretos");
System.out.println("Valor a pagar [método correto 00]= " +
    tributo.calcularImposto(preco, 00));
// usando o método principal de cálculo do imposto com o método do tipo 11:
System.out.println("");
System.out.println("Imposto e método corretos");
System.out.println("Valor a pagar [método correto 00]= "
    + tributo.calcularImposto(preco, 11));
// usando o método principal de cálculo do imposto com o método do tipo 22:
System.out.println("");
System.out.println("Imposto e método corretos");
System.out.println("Valor a pagar [método correto 00]= "
    + tributo.calcularImposto(preco, 22));
}
}

```



#### Saída:

```

Digite o preco:
1000
Imposto incorreto:
Valor do imposto [atribuição direta] = -20.0
Imposto correto, mas método errado
A faixa estava correta por acaso, porque 00 = 0
Valor do imposto[método errado 00]= 45.0
Imposto e método errados
Valor do imposto [método errado 11]= 131.0
Imposto correto, por acaso, porque 22 foi para a última faixa
Valor a pagar [método errado 22]= 281.0
Imposto e método corretos
Valor a pagar [método correto 00]= 45.0
Imposto e método corretos
Valor a pagar [método correto 00]= 128.0
Imposto e método corretos
Valor a pagar [método correto 00]= 281.0

```

1

Imaginemos que tipo de produto é um valor entre os seguintes: [00, 01, 02, 10, 11, 12, 20, 21 e 22], em que o primeiro dígito se refere ao tipo do produto e o segundo à faixa do imposto;



- 2 Primeiro, o método **calculaImposto()** irá determinar o método a ser usado para cada tipo de imposto, e a faixa será usada dentro do método específico;
- 3 O método específico será chamado pelo método principal, que retornará o valor correto do imposto;
- 4 Se um desenvolvedor sem conhecimento correto usar diretamente um dos métodos específicos, o cálculo poderá resultar em um valor errado, pois dificilmente ele irá determinar a faixa correta, pois acabará por passar como parâmetro o tipo do produto e não a faixa;
- 5 Se os testes forem realizados apenas com os tipos entre 00 e 02, provavelmente o resultado estará correto porque será passada apenas a faixa, mas, para os demais casos, de 10 a 12 ou de 20 a 22, provavelmente o resultado será 0 (zero) ou calculado pela última faixa;
- 6 Uma aplicação ainda poderá burlar os cálculos da classe, simplesmente determinando um valor ao atributo imposto **[tributo.imposto = -20;]**, o que resultaria em um imposto incorreto, aumentando o preço do produto; como o imposto é negativo e a operação aritmética é de subtração, haverá um sobre preço sobre o valor.

A falta de conhecimento sobre o uso de uma classe pode gerar erros porque, mesmo realizando testes, as faixas com problemas podem não ser identificadas. Para isso, devemos ocultar parte da implementação da classe.



Fonte: pixabay

## | Princípio do encapsulamento

Atributos não devem ser visíveis por nenhum objeto que não seja instância da própria classe ou de uma **classe descendente (herança)**.

Na linguagem Java, temos quatro diferentes tipos de encapsulamento:

Nível de restrição:	Tipo de Encapsulamento (modificador):
<div>Menor restrição:</div> <div>↕</div> <div>Maior restrição:</div>	<div>1. public – acesso irrestrito;</div> <div>2. (vazio ou omissão) – acesso padrão (package);</div> <div>3. protected;</div> <div>4. private.</div>

A relação apresentada está em ordem de nível de restrição, indo do menos restrito (*public*) até o mais restrito (*private*).

Visibilidade:

- **Public:** Uma classe definida como public pode ser acessada por qualquer classe ou aplicação, sem restrições. Seus membros são igualmente acessíveis (visíveis) por qualquer outra classe ou aplicação. Determina o nível menos restritivo de acesso e visibilidade aos membros (atributos e métodos) de uma classe;
- **Private:** Um membro definido como privado só pode ser acessado por membros da própria classe, ou seja, apenas métodos existentes na própria classe poderão ter acesso (visibilidade) aos atributos e métodos definidos como private. É o nível com maior restrição, pois nem mesmo subclasses dessa classe terão visibilidade sobre esses membros;
- **Protected:** Um membro definido como protegido pode ser acessado apenas por membros da própria classe, das suas subclasses e por outras classes ou aplicações que estejam no mesmo pacote (package);
- **Default** (padrão, omissão): Quando não é usado um modificador de encapsulamento, a visibilidade é dita padrão e os membros têm visibilidade, ou seja, só podem ser acessados por classes e aplicações que estejam no mesmo pacote.

Exemplos de visibilidade de membros



Membros públicos: é a forma normal para métodos de acesso (*Setters* e *Getters*).

```
public class AcessoMembrosPublicos {  
    public String >nome;  
    public void setNome(String no) {  
        if(!no.isEmpty()) {  
            nome = no;  
        }  
    }  
    public String getNome() {  
        return >nome;  
    }  
}
```

Membros com visibilidade padrão: devemos evitar o uso do acesso padrão, para que tenhamos sempre a visibilidade definida.

```
public class AcessoMembrosPadrao {  
    String nome;  
    void setNome(String no) {  
        if(!no.isEmpty()) {  
            nome = no;  
        }  
    }  
    String getNome() {  
        return nome;  
    }  
}
```

Membros privados: é a forma normal para os atributos de classe que não terá subclasses, mas não é adequada para os métodos de acesso (*Setters* e *Getters*).

```
public class AcessoMembrosPrivados {  
    private String nome;  
    public void setNome(String no) {  
        nome = analisaNome(no);  
    }  
    public String getNome() {  
        return nome;  
    }  
    private String analisaNome(String no) {  
  
        if(!no.isEmpty()) {  
            return no;  
        }  
        else {  
            System.out.println("Nome não preenchido!");  
            return "";  
        }  
    }  
}
```

Membros protegidos: é a forma normal para os atributos de classe que terão subclasses, mas também não é adequada para os métodos de acesso (*Setters* e *Getters*).

```
public class AcessoMembrosProtegidos {  
    protected String nome;  
    public void setNome(String no) {  
        nome = analisaNome(no);  
    }  
    public String getNome() {  
        return nome;  
    }  
    protected String analisaNome(String no) {  
        if(!no.isEmpty()) {  
            return no;  
        }  
        else {  
            System.out.println("Nome não preenchido!");  
            return "";  
        }  
    }  
}
```

Como vimos, o encapsulamento determina a visibilidade de classes ou de seus membros. É comum protegemos os atributos de uma classe para que eles não tenham acesso direto, e os valores a serem atribuídos possam ser analisados por um método antes da atribuição.

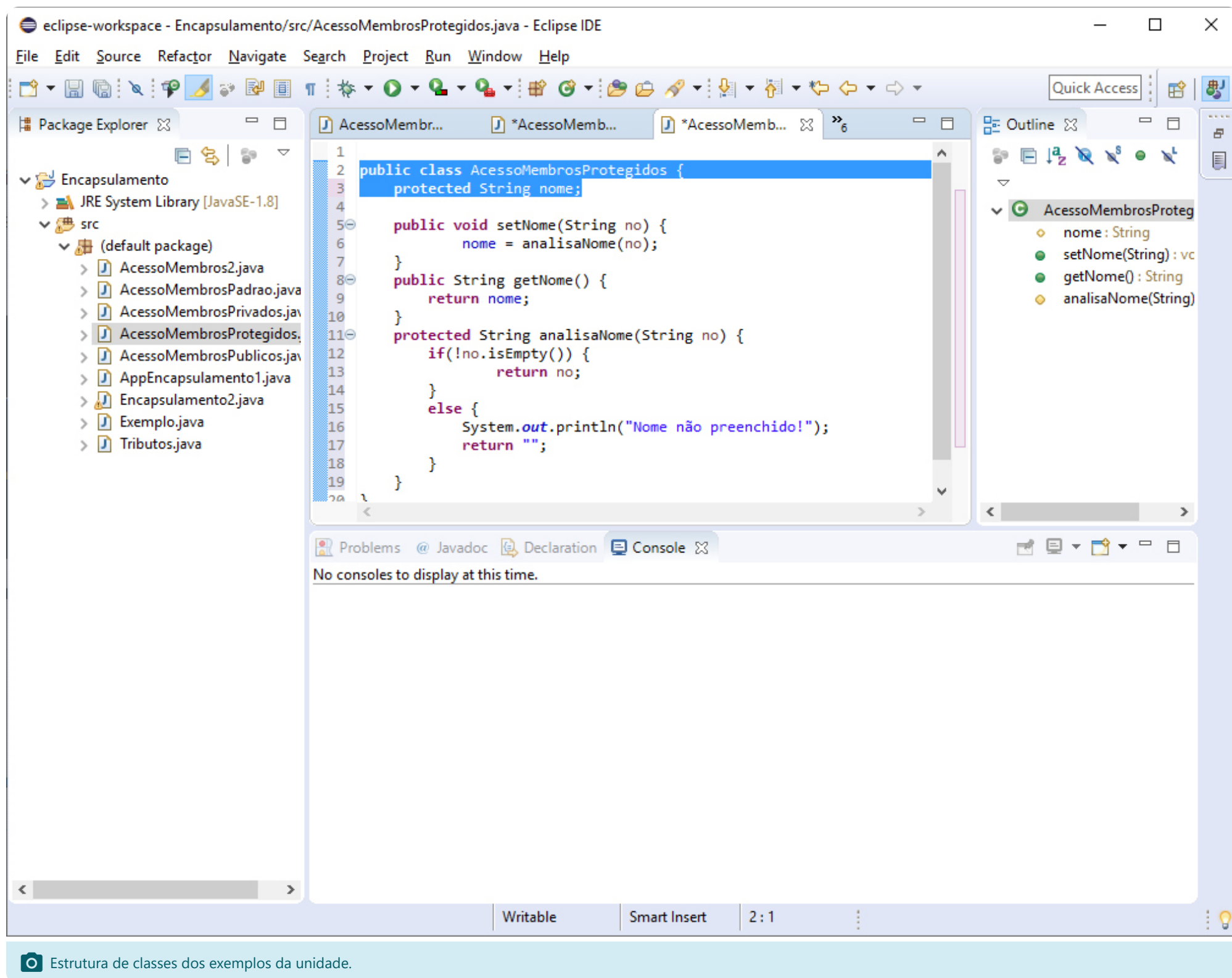
**Atenção!** Aqui existe uma videoaula, acesso pelo conteúdo online

## Pacotes

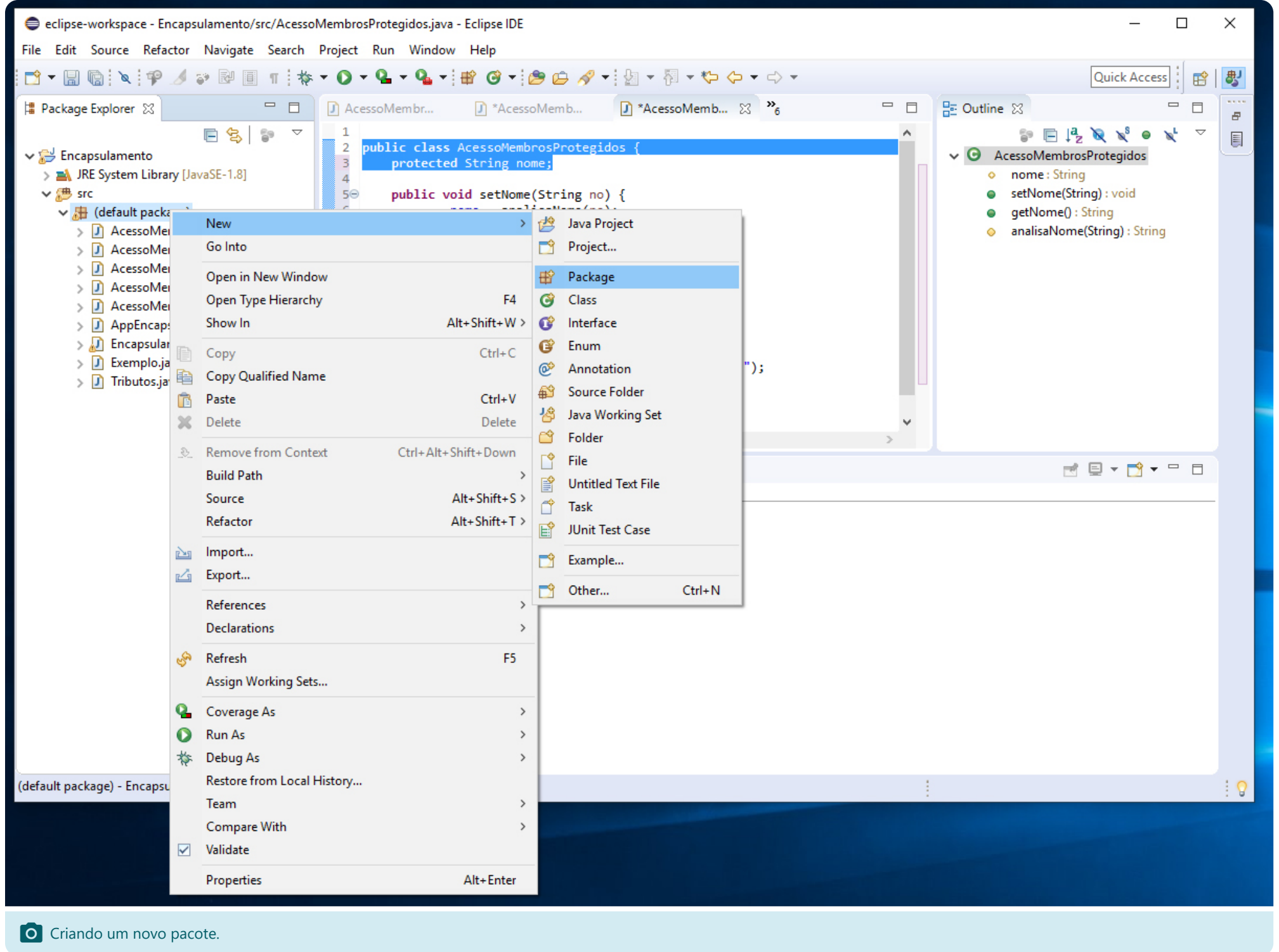
Pacotes em Java são usados para facilitar o armazenamento e controle da biblioteca de classes. Como vimos até o momento, nossa biblioteca de classes vem crescendo e, dessa forma, várias classes foram criadas e estão em diferentes locais. É necessário organizarmos nossas classes e, para isso, podemos usar os pacotes.

Pacotes não passam de uma estrutura de diretórios em que colocamos as nossas classes por afinidade. Por afinidade devemos entender que são classes com algum tipo de aderência, similaridade ou que pertencem a um mesmo assunto.

No projeto criado para nossos exemplos dessa unidade, temos a seguinte estrutura:



Note que temos apenas um pacote, o pacote default do projeto, e todas as classes estão juntas. Para criarmos novos pacotes, basta clicar sobre o projeto e escolher **New / Package**, como você pode observar na imagem a seguir.




Como foram criados três grupos de classes para os exemplos, vamos separar nossas classes em três pacotes, sendo eles: **parte1**, **parte2** e **parte3**.

New Java Package

Java Package

Create a new Java package.



Creates folders corresponding to packages.

Source folder:


Encapsulamento/src

Browse...

Name:


parte2

☐ Create package-info.java



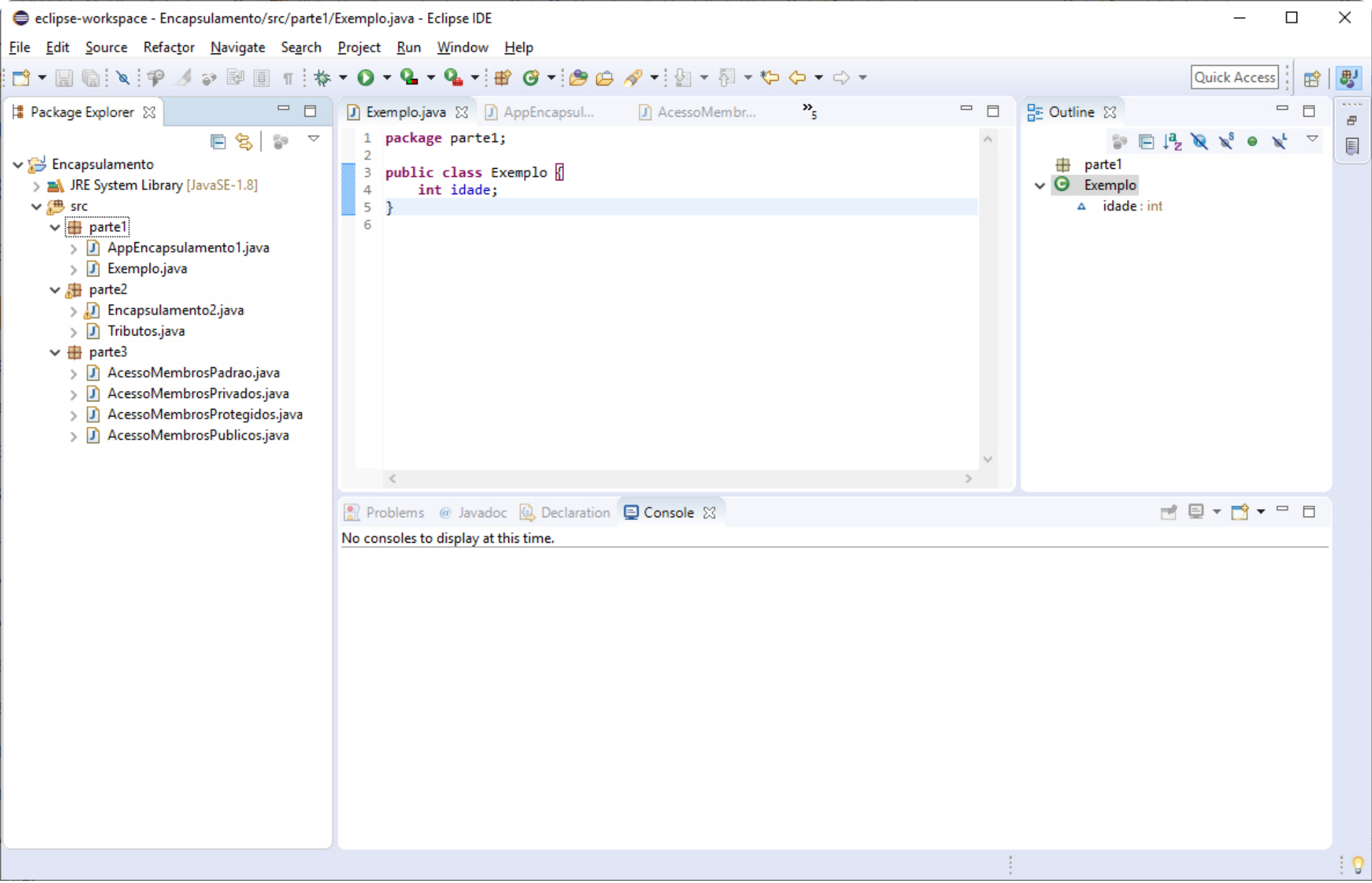
Finish

Cancel

 Determinar o nome do pacote.

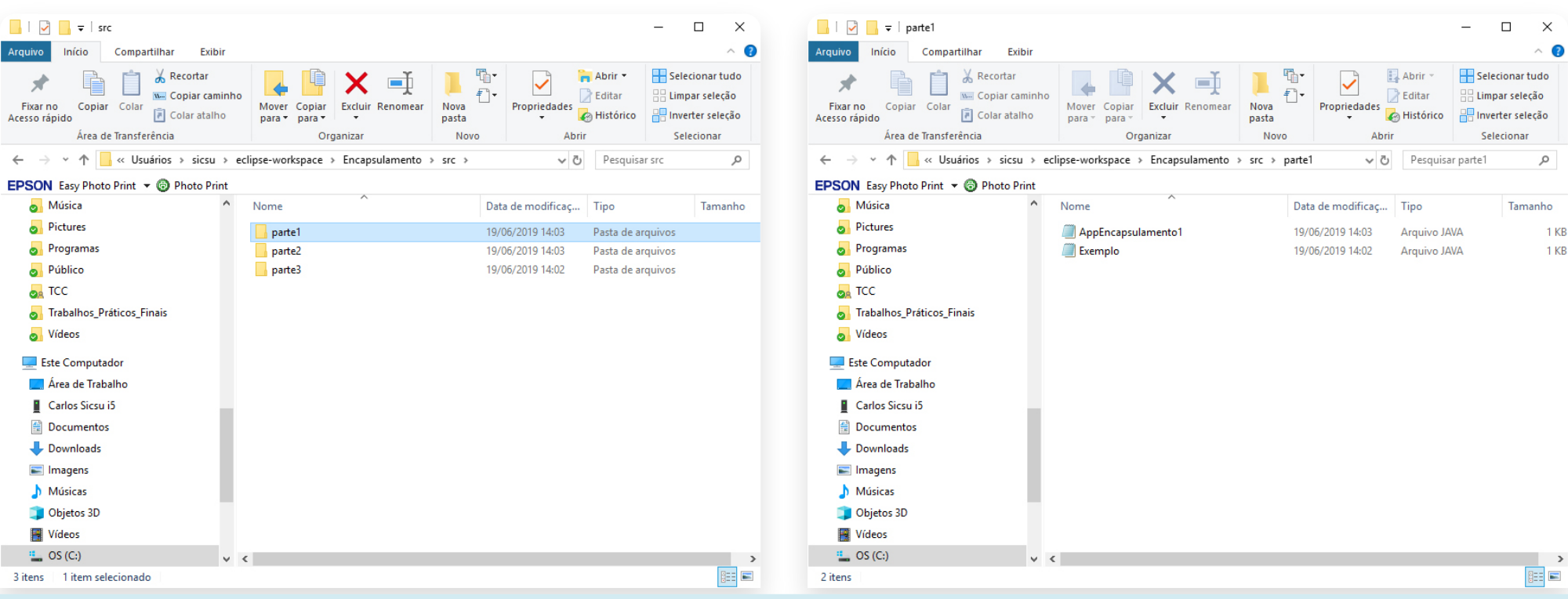
Basta agora arrastar as classes para seus respectivos pacotes.





Clases organizadas em pacotes.

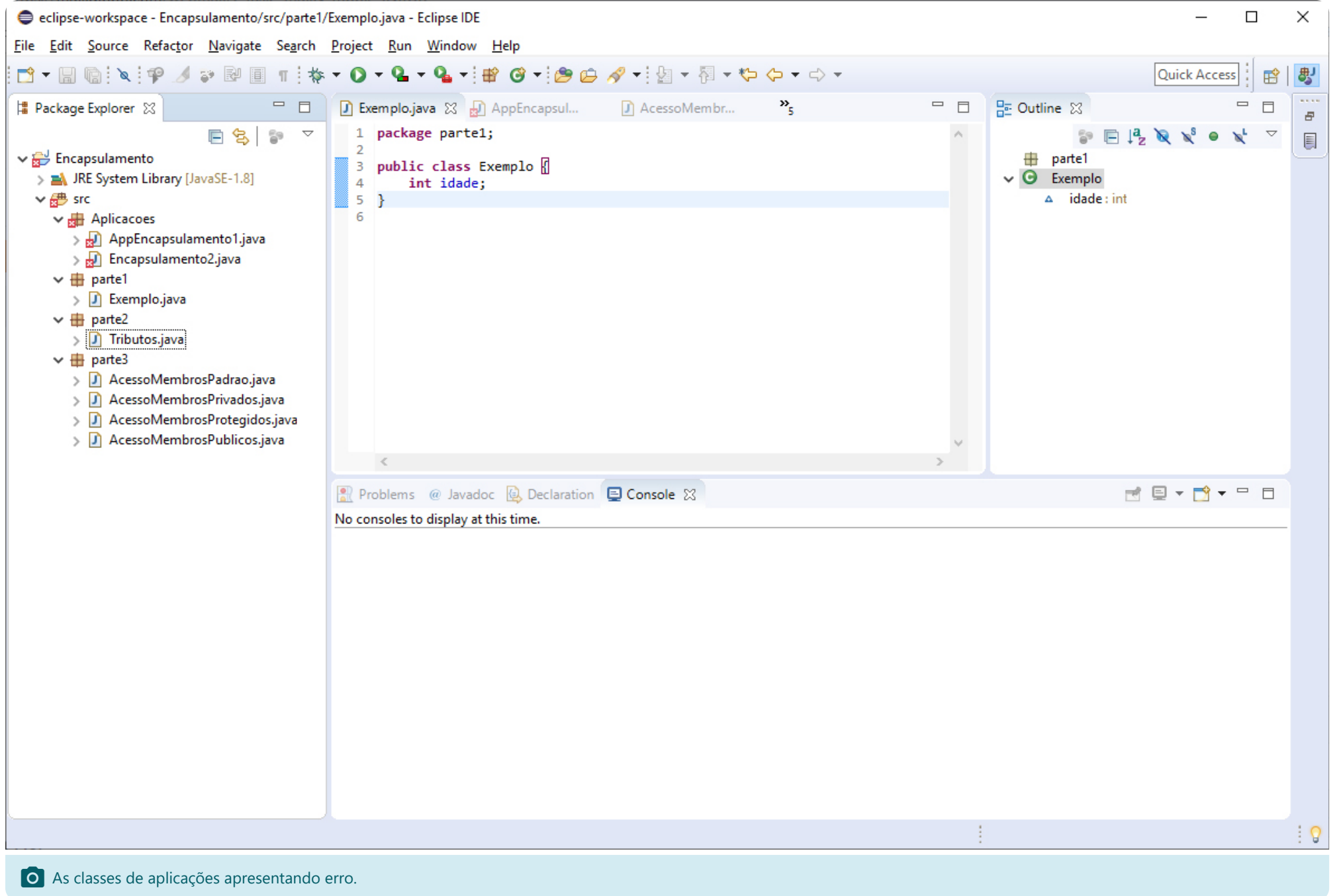
Fisicamente, os arquivos ficarão dentro dos respectivos diretórios, podendo ainda ser criados subpacotes. A separação em pacotes permite duas ou mais classes com o mesmo nome, bastando que elas estejam em diferentes pacotes.



Estrutura dos diretórios em pacotes.

Quando temos nossas classes separadas em pacotes, sempre que precisarmos usá-las devemos importar a(s) classe(s) de seus respectivos pacotes.

Dessa forma, criamos um pacote apenas para as aplicações e transferimos as aplicações de encapsulamento para esse pacote. Note que as classes agora estão apresentando erros.



Isso ocorre porque as aplicações não estão encontrando as respectivas classes e, para que elas sejam encontradas, devemos importar as classes:

**Para a aplicação do primeiro exemplo:**

**import** parte1.Exemplo;

**Para a aplicação do segundo exemplo:**

**import** parte2.Tributos;

Outro ponto importante é que os atributos das classes Exemplo e Tributos estavam com a visibilidade padrão e, para continuar a funcionar, é necessário alterar a visibilidade dos atributos para pública (public), uma vez que essas classes agora estão em diferentes pacotes.

**Atenção!** Aqui existe uma videoaula, acesso pelo conteúdo online

Veja agora um [exemplo de encapsulamento](#) em conjunto com os conceitos de herança e agregação.

**Atenção!** Aqui existe uma videoaula, acesso pelo conteúdo online

## Atividades

1) [Concurso: FCC – 2013 – AL-RN – Analista Legislativo – Analista de Sistemas] - Um dos conceitos básicos de orientação a objetos é o fato de um objeto, ao tentar acessar as propriedades de outro objeto, deve sempre fazê-lo por uso de métodos do objeto ao qual se deseja atribuir ou requisitar uma informação, mantendo ambos os objetos isolados. A essa propriedade da orientação a objetos se dá o nome de:

- ☐ a) Herança
- ☐ b) Abstração
- ☐ c) Polimorfismo
- ☐ d) Mensagem
- ☐ e) Encapsulamento

2) [Concurso: FCC – 2015 – DPE-SP-Programador] - Na programação orientada a objetos com Java, os modificadores de acesso são padrões de visibilidade de acesso às classes, atributos e métodos. Um método com o modificador:

- ☐ a) *Default* pode ser acessado de dentro da própria classe, de qualquer classe do pacote e de subclasses que herdam da classe que contém o método.
- ☐ b) *Public* pode ser acessado somente a partir de classes que estão no mesmo pacote.
- ☐ c) *Protected* pode ser acessado somente de dentro da própria classe ou de classes que estão no mesmo pacote.
- ☐ d) *Private* pode ser acessado somente de dentro da própria classe.
- ☐ e) *Static* pode ser acessado a partir de qualquer classe da aplicação.

## Notas

### Referências

DEITEL, Paul. **Java**: como programar (Biblioteca Virtual). 10. ed. São Paulo: Pearson, 2017.

FURGERI, Sérgio. **Java 8** – ensino didático: desenvolvimento e implementação de aplicações. São Paulo: Érica, 2015.

### Próxima aula

- Classes abstratas e interfaces.

Explore mais

---

Leia o texto:

- [Encapsulamento, Polimorfismo, Herança em Java.](#)