
# Programação I
# Encapsulamento

Jorge Roberto Trento
Bacharel em Ciências da Computação - UNOESC
Especialização em Ciências da Computação - UFSC
Formação Pedagógica - Formadores de Educação Profissional – UNISUL
Especialização em Ensino Superior – FIE
Especialização em Gestão de Tecnologia da Informação – FIE

# Introdução


- Uma das grandes vantagens do paradigma de programação orientada a objetos é facilidade de criação de códigos reutilizáveis.
- O encapsulamento de código é um conjunto de técnicas e boas práticas de programação que garantam que o código escrito seja visto como uma "cápsula", algo que contém uma parte externa (a "casca") e uma parte interna (o "núcleo"). A parte externa pode ser facilmente visualizada e entendida, porém não é possível (ou é desnecessário) o conhecimento do seu interior.

# Introdução

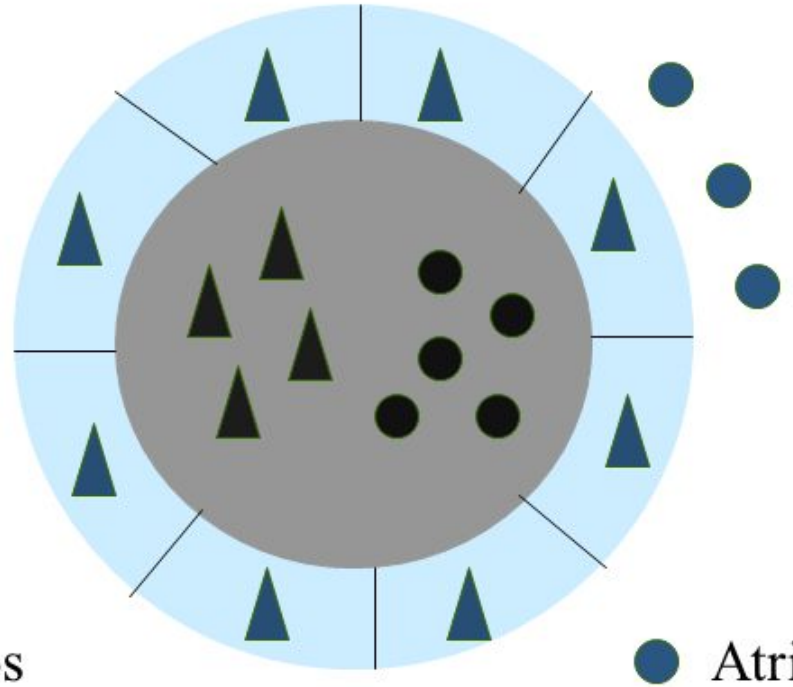
- Exemplo: podemos imaginar um celular. Ele possui uma parte externa bem definida (geralmente um invólucro de plástico com um teclado e uma tela) e uma parte interna desconhecida (circuitos, fios, chips, capacitores, etc). Ao utilizarmos um celular, não o vemos como um emaranhado de circuitos com uma antena; o vemos como uma espécie de caixa-preta que pode fazer e receber ligações.

- Podemos até conhecer e entender como as ligações são transmitidas (ondas de rádio através de antenas, etc), porém isso não é de nosso interesse. Não sabemos quais chips recebem corrente elétrica quando uma ligação é feita, ou quantos amperes cada capacitor recebe: apenas queremos fazer ligações e trocar mensagens.

# Introdução

- Na visão de orientação a objetos, o celular é uma cápsula;
- Ele possui um conjunto de funcionalidades que podem ser utilizadas através de uma interface pública (a tela e o teclado), porém o conteúdo do aparelho em si (e seu funcionamento) não nos interessa.
- Dessa forma, se algum dia o fabricante do celular decidir modernizar os componentes eletrônicos internos do celular para que as ligações sejam melhores, basta que o celular seja levado em uma loja e que os componentes sejam trocados. Depois disso, o celular fará e receberá chamadas melhores, porém a sua tela e o seu teclado continuam iguais. Qualquer um que utilize aquele celular continuará utilizando-o da mesma forma, porém agora com capacidade melhorada.

# Encapsulamento



Métodos públicos

Métodos privados

Atributos públicos

Atributos privados

# Encapsulamento


- O encapsulamento na programação orientada a objetos tenta criar esse cenário. As classes são desenhadas de forma a funcionarem como cápsulas, caixas-pretas: existe uma interface pública (os métodos públicos) que disponibiliza um conjunto de funcionalidades, porém, por trás disso existem códigos que implementam essas funcionalidades.
- Os códigos que implementam as funcionalidades ficam indisponíveis, e a única preocupação do programador ao usar uma classe é saber o que ela faz, não como ela faz.

# Encapsulamento


- Claro que neste momento estamos preocupados em saber como uma classe faz algo, visto que estamos implementando estes métodos.
- Mas no exemplo da classe que implementava a caixa de diálogo, só estávamos preocupados com o que a classe fazia e não como fazia.

```
import javax.swing.JOptionPane;

public class Dialogo{
     public static void main(String[] args){

     JOptionPane.showMessageDialog(null, "É Java \nMano!");

     }
}
```

# Modificadores de visibilidade


- É através dos **modificadores de visibilidade (ou acesso),** como o **private** que vimos, que o encapsulamento é obtido.
- Esses modificadores são palavras-chave que são colocadas junto a **métodos e atributos** da classe e indicam quais desses membros são visíveis para classes externas.
- Em Java, existem três tipos de modificadores de visibilidade: **public**, **protected** e **private**.

# Exemplo


```
class Pessoa {
public String nome;
protected String sobrenome;
private int idade;

public void metodoPublico() {
}
protected void metodoProtected() {
}
private void metodoPrivate() {
}
}
```

# Modificadores de visibilidade

| Modificador | Classe | Pacote | Subclasse | Mundo |
|---|---|---|---|---|
| **public** | Sim | Sim | Sim | Sim |
| **protected** | Sim | Sim | Sim | Não |
| **private** | Sim | Não | Não | Não |
| **Sem modificador** | Sim | Sim | Não | Não |

# Modificadores de visibilidade (UML)


+ public;

# protected;

- private

# Modificadores de visibilidade


```
class Teste {
public static void main(String[] args) {
Pessoa p = new Pessoa();
p.idade = 12;
}
}
```

**Isso vai funcionar?**

# Modificadores de visibilidade


```
class Teste {
public static void main(String[] args) {
Pessoa p = new Pessoa();
// p.idade = 12;
p.nome = "Fulano";
p.sobrenome = "Silva";
}
}
```

**Isso vai funcionar?**

# Outro Problema


- O modificador **private** faz com que ninguém consiga modificar, nem mesmo ler, o atributo. Com isso, temos um problema: como fazer para mostrar/alterar a idade de uma Pessoa, já que nem mesmo podemos acessá-lo para leitura?

# Métodos de Acesso


Sempre que precisamos arrumar uma maneira de fazer alguma coisa com um objeto, utilizamos os **métodos!**

Segundo as boas práticas de programação e o conceito de encapsulamento:

> "Um atributo deve ser alterado somente por métodos da classe que o possui; além disso um atributo deve ser lido através de um método, e não deve ser acessado diretamente."

Vamos então criar um método para alterar e ler a idade.

# Getters e Setters


- Para permitir o acesso aos atributos **privados** de uma maneira controlada, a prática mais comum é criar dois métodos, um que retorna o valor e outro que muda o valor.

- Já que esses métodos quase sempre devem ser implementados quando os atributos forem do tipo **private**, foi criado um padrão para os nomes, evitando que acabemos por criar vários métodos com nomes diferentes (perdendo tempo para criar e para encontra-los depois).

# Getters e Setters


A convenção para esses métodos é de colocar a palavra **get** ou **set** antes do nome do atributo;

- get → pegar;
- set → setar;

  OBS: Métodos get e set são sempre públicos;

Por exemplo, dar acesso a leitura e escrita a todos os atributos da classe pessoa:

# Exemplo Getters e Setters


```
class Pessoa {
   private String nomeCompleto;
   private int idade;

   public String getNomeCompleto() {
   return this.nomeCompleto;
   }
   public void setNomeCompleto(String n) {
   this.nomeCompleto = n;
   }
   public int getIdade() {
   return this.idade;
   }
   public void setIdade(int i) {
   this.idade = i;
   }
}
```

# Exemplo Getters e Setters


```
class Programa
 {

 public static void main(String[] args)
  {
   Pessoa p = new Pessoa();

   p.setNomeCompleto("Fulano da Silva");
   p.setIdade(29);

   System.out.println("Nome = " + p.getNomeCompleto());
   System.out.println("Idade = " + p.getIdade());
   }
}
```

# Exemplo: Getters e Setters


```
class Conta {
private double saldo;
private double limite;

public double getSaldo() {
   return this.saldo;
   }
public void setSaldo(double saldo) {
   this.saldo = saldo;
   }
   public double getLimite() {
   return this.limite;
   }
public void setLimite(double limite) {
   this.limite = limite;
   } }
```

# OBSERVAÇÕES


É uma má prática criar uma classe e, logo em seguida, criar getters e setters para todos seus atributos. Você só deve criar um getter ou setter quando tiver a real necessidade. Repare que no exemplo da classe Conta, set Saldo não precisa ser sido criado, já que queremos que todos usem **deposita()** e **saca()** para que o saldo seja alterado!

# OBSERVAÇÕES


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.

# Interface


Para que a reutilização de código e o encapsulamento funcionem, os métodos e propriedades públicas da classe não podem ser alterados depois que a classe estiver em uso.

O conjunto de métodos e propriedades públicas de uma classe **são a sua interface pública** para com o mundo, é a forma como o mundo externo enxerga a classe. Recordando os componentes do celular, todos os componentes internos do celular podem ser alterados, porém ninguém irá notar (ou se importar com) isso se a interface pública permanecer a mesma.

# Reusabilidade


No caso da orientação a objetos, a classe *Pessoa* com *getters* e *setters* é o nosso celular. Podemos utilizar essa classe em quantos programas quisermos e podemos modificá-la à vontade, **contanto que os métodos e propriedades públicas não sejam alterados**. Vamos imaginar que, por alguma razão, precisamos trocar o nome de todas as propriedades da classe *Pessoa* e que precisamos adicionar um método novo. A regra básica do encapsulamento e da **reusabilidade de código** é não alterar a interface pública, então vamos identificar quais são as propriedades e métodos dessa classe que não poderemos mexer:

# Reusabilidade


**Propriedades públicas: *não existem***

**Métodos públicos:**

public String getNomeCompleto()

public void setNomeCompleto(String n)

public int getIdade()

public void setIdade(int i)

# Reusabilidade


Os métodos listados são aqueles que nós não podemos mexer na assinatura. O corpo do método pode ser alterado, contanto que o **nome do método, o tipo de retorno e os parâmetros recebidos** continuem os mesmos. Agora que já sabemos o que não pode ser alterado, vamos para a alteração (trocar o nome das propriedades e adicionar um método novo). Estão marcados em vermelho os códigos que sofreram alterações.

```
class Pessoa {
      private String meuNomeCompleto;
      private int minhaIdade;

      public String getNomeCompleto() {
         return this.meuNomeCompleto;
      }
      public void setNomeCompleto(String n) {
         this.meuNomeCompleto = n;
      }
      public int getIdade() {
         return this.minhaIdade;
      }
      public void setIdade(int i) {
         this.minhaIdade = i;
      }
      public void limpaDados() {
         this.meuNomeCompleto = “Sem nome?”;
         this.minhaIdade = 0;
      } }
```

# Encapsulamento e Reusabilidade


A vantagem ao fazer alterações é que o resto do seu sistema continuaria funcionando. O programa anterior, que usa a classe pessoa, continuaria funcionando normalmente e *não precisaria* ser adaptado ou recompilado para utilizar  a nossa nova classe *Pessoa*. O código a seguir continuaria funcionando exatamente da mesma maneira que antes.

# Exemplo


```
class Teste {
public static void main(String[] args) {
Pessoa p = new Pessoa();
p.setNomeCompleto(“Fulano da Silva”);
p.setIdade(29);

System.out.println(“Nome = ” + p.getNomeCompleto());
System.out.println(“Idade = ” + p.getIdade());
}
}
```

# Modificador final


A palavra chave **final** também é um modificador.

Esse modificador define que o valor atribuído a variável não poderá ser alterado.

```
private final int INCREMENT;
```

Essa variável, depois de inicializada não poderá mais ser alterada, ou seja, é uma **constante.**

# Modificador final


Embora essas variáveis possam ser inicializadas quando forem declaradas, isso não é obrigatório:

```
private final int INCREMENT;
private final int INCREMENT = 10;
```

Elas podem ser inicializada por cada um dos construtores da classe para que cada objeto da classe tenha um valor diferente, que não poderá ser alterado.

# Resumo


**Os modificadores de acesso tornam propriedades e/ou métodos visíveis ou invisíveis para outras classes. Os modificadores são: *public*, *protected* e *private.***

O encapsulamento de classes permite a reutilização de código.

Pelas regras de boa programação, as propriedades devem ter seu valor alterados e acessados apenas por métodos da classe.

Métodos de acesso são chamados *getters.*

Métodos modificadores são chamados *setters.*

A palavra chave ***final*** é usada para definir variáveis constantes no Java.

# Encapsulamento

Autor:
Prof. Doglas André Finco
doglas.andref@uffs.edu.br

jorge.trento@uffs.edu.br