

# Encapsulamento

Orientação a Objetos – DCC025

Gleiph Ghiotto Lima de Menezes

[gleiph.ghiotto@ufjf.br](mailto:gleiph.ghiotto@ufjf.br)

# Conteúdo da Aula

- Encapsulamento
- Métodos Getters e Setters

# Definição

- Encapsulamento é uma técnica utilizada para esconder detalhes de uma funcionalidade, tornando partes do sistema o mais independentes possível umas das outras
- A ideia de cápsula nos lembra algo que protege o conteúdo em seu interior



## Exemplo 1

- Quando apertamos um botão do controle remoto da TV para mudar de canal, simplesmente mudamos de canal. Mas não sabemos exatamente o mecanismo ou a programação do controle remoto que faz com que a TV mude de canal



## Exemplo 2

- Considere um objeto da classe **Automóvel** que disponibiliza para um objeto da classe **Pessoa** a direção como uma de suas tarefas, possibilitando que o carro possa ser guiado por uma pessoa para a esquerda ou para a direita
- Através da direção, o objeto da classe Pessoa solicita ao objeto da classe Automóvel esse serviço sem saber COMO serão feitos
- Apenas o objeto da classe Automóvel sabe quais mecanismos serão acionados para atender a solicitação do objeto da classe Pessoa



# Na programação orientada a objetos...

- Característica da OO que possibilita **ocultar parte da implementação das classes**
- Normalmente as classes ocultam os detalhes de implementação dos seus “usuários”
- O usuário se preocupa com a funcionalidade oferecida pelos itens das classes e não como tal funcionalidade é implementada
- Normalmente **podemos ver as classes apenas pelos “comportamentos”** (métodos) que elas oferecem

# Na programação orientada a objetos...

- Usar encapsulamento no desenvolvimento pode ser considerado como estabelecer um contrato que define o que o usuário pode fazer com os objetos de uma determinada classe
- Recurso que ajuda a desenvolver programas com **maior qualidade e flexibilidade** para **mudanças futuras**
- Uma das **principais vantagens** do paradigma OO é a possibilidade de **encapsular atributos e métodos de uma classe**

## Na programação orientada a objetos...

- **Muitas vezes é desejável que os atributos estejam ocultos** dos usuários da classe para evitar que os dados sejam manipulados diretamente
  - Normalmente é mais seguro que os atributos sejam manipulados através dos métodos da referida classe



**Será que todos os itens de uma classe (atributos ou métodos) devem estar ocultos?**



# Modificadores de acesso

- Existem quatro modificadores de acesso em Java:
  - Public
    - **Todas as classes** podem acessar métodos e atributos
  - Private
    - Apenas **métodos da própria classe** podem acessar métodos e atributos

# Modificadores de acesso

- Existem quatro modificadores de acesso em Java:
  - Protected
    - Apenas **métodos em classes do mesmo pacote** ou **subclasses** podem acessar os métodos e atributos
  - Package ou Friendly
    - Apenas **métodos em classes do mesmo pacote** podem acessar os métodos e atributos

# Exemplo 1

- Criar uma classe Agenda com os atributos
  - dia
  - mes
  - ano
  - anoAtual
  - anotacao – descrição compromisso
- Criar um métodos
  - validar
    - Verifica se uma data é válida
  - anotar
    - Cria um evento
  - mostrarAnotacao
    - Mostra a data a mensagem do evento



# Exemplo 1

```
package br.ufjf.dcc.oo.desenho.aula;

public class Agenda {

    int dia;
    int mes;
    int ano;
    int anoAtual = 2019;
    String anotacao;

    public Agenda() {
        this.dia = 0;
        this.mes = 0;
        this.ano = 0;
        this.anotacao = "Anotação padrão!";
    }
}
```

# Exemplo 1

```
public void validar() {
    if (this.dia < 1 || this.dia > 31
        || this.mes < 1 || this.mes > 12
        || this.ano < this.anoAtual) {

        this.dia = 0;
        this.mes = 0;
        this.ano = 0;
        this.anotacao = "Anotação não inserida. Data inválida...";
    }
}
```

# Exemplo 1

```

public void anotar(int dia, int mes, int ano, String anotacao) {
    this.dia = dia;
    this.mes = mes;
    this.ano = ano;
    this.anotacao = anotacao;

    this.validar();
}

public void mostrarAnotacao() {
    System.out.println(this.dia + "/" + this.mes + "/" + this.ano + "-" + this.anotacao);
}
}

```

## Exemplo 1

- Mas onde está o encapsulamento?





# Exemplo 1

- Mas onde está o encapsulamento?
  - O fato de ter inserido a validação da data não encapsula (“esconde”) o atributo
  - Primeiro vamos ver a fragilidade de uma classe sem encapsulamento
  - Vamos criar dois objetos da classe agenda - agenda1 e agenda2
  - Guardaremos em agenda1 uma anotação com data válida
  - Guardaremos em agenda2 uma anotação com data inválida
  - Em seguida consultaremos os dados armazenados usando o método mostrarAnotacao() dos respectivos objetos



# Exemplo 1

```
package br.ufjf.dcc.oo.desenho.aula;

public class TesteAgenda {

    public static void main(String[] args) {
        Agenda agenda1 = new Agenda();
        Agenda agenda2 = new Agenda();

        agenda1.anotar(20, 11, 2020, "Aula de DCC025");
        agenda2.anotar(20, 11, 2015, "Aula de DCC025");

        agenda1.mostrarAnotacao();
        agenda2.mostrarAnotacao();
    }
}
```

## Exemplo 1

- Na classe Agenda não foram utilizados recursos que possibilitam o Encapsulamento e tudo funcionou normalmente
- As vias de acesso estão protegidas?
- Vamos identificar falhas na implementação no contexto do encapsulamento
- E se o método main da classe Teste sofrer uma pequena alteração?

# Exemplo 1

- O que seria impresso agora?

```
package br.ufjf.dcc.oo.desenho.aula;

public class TesteAgenda {

    public static void main(String[] args) {
        Agenda agenda1 = new Agenda();
        Agenda agenda2 = new Agenda();

        agenda1.anotar(20, 11, 2020, "Aula de DCC025");
        agenda2.anotar(20, 11, 2015, "Aula de DCC025");

        agenda1.mostrarAnotacao();
        agenda2.mostrarAnotacao();

        agenda1.ano = 1994;

        agenda1.mostrarAnotacao();
        agenda2.mostrarAnotacao();
    }
}
```

# Exemplo 1

- Executando a aplicação é possível ver que o código implementado permite que seja escolhidas **datas inválidas** para uma anotação. Isto significa que o **código** está **suscetível a falhas**
- Imagine este tipo de problema em um programa utilizado na vida diariamente pelas pessoas.....**Qual é a solução?**
- Como mencionado anteriormente, esta classe ainda não possui a ideia do encapsulamento implementada. Todos os itens que compõem a classe estão liberados para o usuário (da classe). Estamos verificando os efeitos de não utilizar o encapsulamento
- O encapsulamento (ocultação) pode ser obtido aplicando modificadores de acesso aos itens das classes

Atributos sem  
modificadores de  
acesso (Package)

Métodos e  
construtor com  
modificadores de  
acesso (Public)

```
package br.ufjf.dcc.oo.desenho.aula;
```

```
public class Agenda {
```

```
    int dia;
```

```
    int mes;
```

```
    int ano;
```

```
    int anoAtual = 2019;
```

```
    String anotacao;
```

```
    public Agenda() {...6 lines }
```

```
    public void validar() {...11 lines }
```

```
    public void anotar(int dia, int mes, int ano, String anotacao
```

```
    public void mostrarAnotacao() {...3 lines }
```

```
}
```

## Exemplo 1

- Para corrigir este problema na classe Agenda deve ser aplicada a ideia de **encapsulamento**, tornando os **atributos** desta classe **privados**
- Na orientação a objetos é prática quase obrigatória proteger os atributos com **private**
- Cada classe é responsável por controlar seus atributos

## Exemplo 1

- Cada classe deve julgar se um novo valor é válido ou não para um atributo
  - Esta validação não deve ser controlada por quem está usando a classe
- O método validar() também pode ser privado da classe Agenda
- Apenas os métodos anotar() e mostrarAnotacao() poderão ser acessados pelos usuários (da classe)



```
package br.ufjf.dcc.oo.desenho.aula;

public class Agenda {

    int dia;
    int mes;
    int ano;
    int anoAtual = 2019;
    String anotacao;

    public Agenda() { ...6 lines }

    private void validar() { ...11 lines }

    public void anotar(int dia, int mes, int ano, String anotacao)

    public void mostrarAnotacao() { ...3 lines }
}
```

# Programando voltado para a interface e não para a implementação

- É sempre bom **programar pensando na interface** da sua classe, como seus usuários a estarão utilizando e não somente em como ela irá funcionar
- A implementação em si, o conteúdo dos métodos, não tem tanta importância para o usuário dessa classe, uma vez que ele só precisa saber o que cada método pretende fazer e não como ele faz, pois isto pode mudar com o tempo
  - Design Patterns: Elements of Reusable Object-Oriented Software - Gamma, E. and Helm, R. and Johnson, R. and Vlissides, J. - 1994.

# Métodos Getters e Setters

- O encapsulamento oculta os atributos ou métodos dentro de uma classe
- Devemos ter meios para que o usuário acesse os membros privados
- Isso é possível com a criação de métodos
- A prática mais comum é criar dois métodos, um que retorna e outro que muda o valor do atributo
- Esses **métodos de acesso** são conhecidos como métodos **getters** e **setters**

# Métodos Getters e Setters

- Exemplo de Get e Set para a classe Agenda

```
package br.ufjf.dcc.oo.desenho.aula;
```

```
public class Agenda {
```

```
    private int dia;  
    int mes;  
    int ano;  
    int anoAtual = 2019;  
    String anotacao;
```

```
    public int getDia(){  
        return this.dia;  
    }
```

```
    public void setDia(int dia){  
        this.dia = dia;  
        this.validar();  
    }
```

# Métodos Getters e Setters

- Exemplo de Get e Set para a classe Agenda

```
public Agenda() { ...6 lines }

private void validar() { ...11 lines }

public void anotar(int dia, int mes, int ano, String anotacao) { ...8 lines }

public void mostrarAnotacao() { ...3 lines }
}
```

## Exemplo 2

- Considere o exemplo que contém a classe ContaBancaria. Existe algum problema de encapsulamento?

```
package br.ufjf.dcc.oo.desenho.aula;

public class ContaBancaria {

    private String agencia;
    private String numero;
    private String cliente;
    private float saldo;

    public void sacar(int valor){
        if(this.saldo >= valor)
            this.saldo -= valor;
    }

}
```

## Exemplo 2

- O usuário da classe é obrigado a realizar saques utilizando o método sacar?

```
package br.ufjf.dcc.oo.desenho.aula;

public class ContaBancaria {

    private String agencia;
    private String numero;
    private String cliente;
    private float saldo;

    public void sacar(int valor){
        if(this.saldo >= valor)
            this.saldo -= valor;
    }

}
```

## Exemplo 2

- Seria uma boa ideia inserir os métodos de acesso nesta classe?

```
package br.ufjf.dcc.oo.desenho.aula;

public class ContaBancaria {

    private String agencia;
    private String numero;
    private String cliente;
    private float saldo;

    public void sacar(int valor){
        if(this.saldo >= valor)
            this.saldo -= valor;
    }

}
```



## Exemplo 2

### • OBSERVAÇÕES

- Não é boa prática criar uma classe e implementar getters e setters para todos os seus atributos
- Deve-se criar um getter ou setter apenas se tiver a real necessidade
- Na classe banco por exemplo setSaldo não precisaria ter sido criado, já temos o método sacar() e também poderíamos criar o método depositar().

## Exercício 1

- Implemente o método depositar() na classe ContaBancaria.
- Depois disso nossa classe estará segura no que diz respeito à movimentação do atributo saldo, graças ao **encapsulamento**

# Encapsulamento

Orientação a Objetos – DCC025  
Gleiph Ghiotto Lima de Menezes  
[gleiph.ghiotto@ufjf.br](mailto:gleiph.ghiotto@ufjf.br)