

# Java Advanced

12/08/2024

**Prof. Dr. Marcel Stefan Wagner**

**Aula 03 – Revisão de POO**

**FIAP**

# Tópicos Abordados

- 1 Introdução à Programação Orientada a Objetos
- 2 Classes
- 3 Encapsulamento
- 4 Herança
- 5 Abstração
- 6 Polimorfismo
- 7 Temas para a Próxima Aula
- 8 Referências Bibliográficas

# Introdução



## Encapsulamento

- Permite definir quais partes (classes e/ou atributos) que podem ser acessadas ou não pelos objetos.
- E assim completamos nosso conceito de encapsulamento, em que temos classes dos objetos de negócio com atributos privados (acessíveis somente pela própria classe) e métodos públicos (acessíveis por quaisquer classes), *get* para recuperar os valores dos atributos e métodos públicos *set* para defini-los.

## Encapsulamento

Vamos agora começar a declarar os atributos. Neste momento, precisamos pensar quais as informações de aluno que são importantes para o meu sistema. Vamos hipoteticamente criar um sistema no qual queremos armazenar as informações do aluno relativas a nome, endereço, idade, nota de matemática, nota de português e nota de geografia. Teremos então a seguinte definição para nossa classe:

```
1 public class RegistraAluno {  
2     private String nome;  
3     private String endereco;  
4     private int idade;  
5     private double notaMatematica;  
6     private double notaPortugues;  
7     private double notaGeografia;  
8 }
```

Vale ressaltar que, quando queremos armazenar alguma informação textual sobre um objeto, o tipo mais ideal do atributo é a classe **String**, como fizemos com o atributo nome e endereço. Além disso, escolhemos o tipo primitivo **int** para armazenar a idade, já que é um valor inteiro, e o tipo primitivo **double** para armazenar as notas, já que estas são todas com precisão de duas casas à direita da vírgula.

Algumas observações de boas práticas:

- As declarações dos atributos são feitas sempre na primeira parte do código da classe.
- Declare um atributo por linha, mesmo que ele seja do mesmo tipo.
- Declare atributo como **private**; assim, somente a própria classe pode manipulá-lo; esse é o conceito de **encapsulamento**, muito importante na Orientação a Objeto.

## Encapsulamento

```

1 public class RegistraAluno {
2     private String nome;
3     private String endereco;
4     private int idade;
5     private double notaMatematica;
6     private double notaPortugues;
7     private double notaGeografia;
8
9     private static int contadorEstudante;
10
11     // retorna o nome do estudante
12     public String getNome(){
13         return nome;
14     }
15
16     // define ou altera o nome do estudante
17     public void setNome(String temp){
18         nome = temp;
19     }
20
21     // retorna o endereço do estudante
22     public String getEndereco(){
23         return endereco;
24     }
25
26     // define ou altera o endereço do estudante
27     public void setEndereco(String temp){
28         endereco = temp;
29     }
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66

```

```

30
31 // retorna a idade do estudante
32 public int getIdade(){
33     return idade;
34 }
35
36 // define ou altera idade do estudante
37 public void setEndereco(int temp){
38     idade = temp;
39 }
40
41 // define ou altera as notas
42 public void setNotaMatematica(double temp){
43     notaMatematica = temp;
44 }
45
46 public void setNotaPortugues(double temp){
47     notaPortugues = temp;
48 }
49
50 public void setNotaGeografia(double temp){
51     notaGeografia = temp;
52 }
53
54 // retorna a média do estudante
55 public double getMedia(){
56     double resultado = 0;
57     resultado = (notaMatematica + notaPortugues + notaGeografia) / 3;
58     return resultado;
59 }
60
61 // retorna a quantidade de estudantes cadastrados
62 public static int getQuantidadeAlunos(){
63     return contadorEstudante;
64 }
65
66 }

```

## Encapsulamento

Por fim, vamos implementar uma classe (classe **AppRegistraAluno**), que representa uma aplicação que utiliza nosso objeto aluno do mundo real, representado computacionalmente pela classe **RegistraAluno**:

```
1 public class AppRegistraAluno {
2     public static void main(String args[]){
3
4         // cria 3 objetos RegistraAluno
5         RegistraAluno ana = new RegistraAluno();
6         RegistraAluno beto = new RegistraAluno();
7         RegistraAluno carlos = new RegistraAluno();
8
9         ana.setNome("Ana Machado");
10        beto.setNome("Roberto da Silva");
11        carlos.setNome("Carlos Alberto");
12
13        System.out.println(ana.getNome());
14
15        System.out.println("Contador: "+RegistraAluno.getQuantidadeAlunos());
16    }
17 }
18 }
```

## Herança

- Permite que características comuns a diversas classes sejam “herdadas” de uma classe base, ou superclasse.
- É uma forma de reutilização de *software*.
  - Novas classes são criadas a partir das classes existentes, absorvendo seus atributos e comportamentos e adicionando novos recursos.



# Herança

## Herança

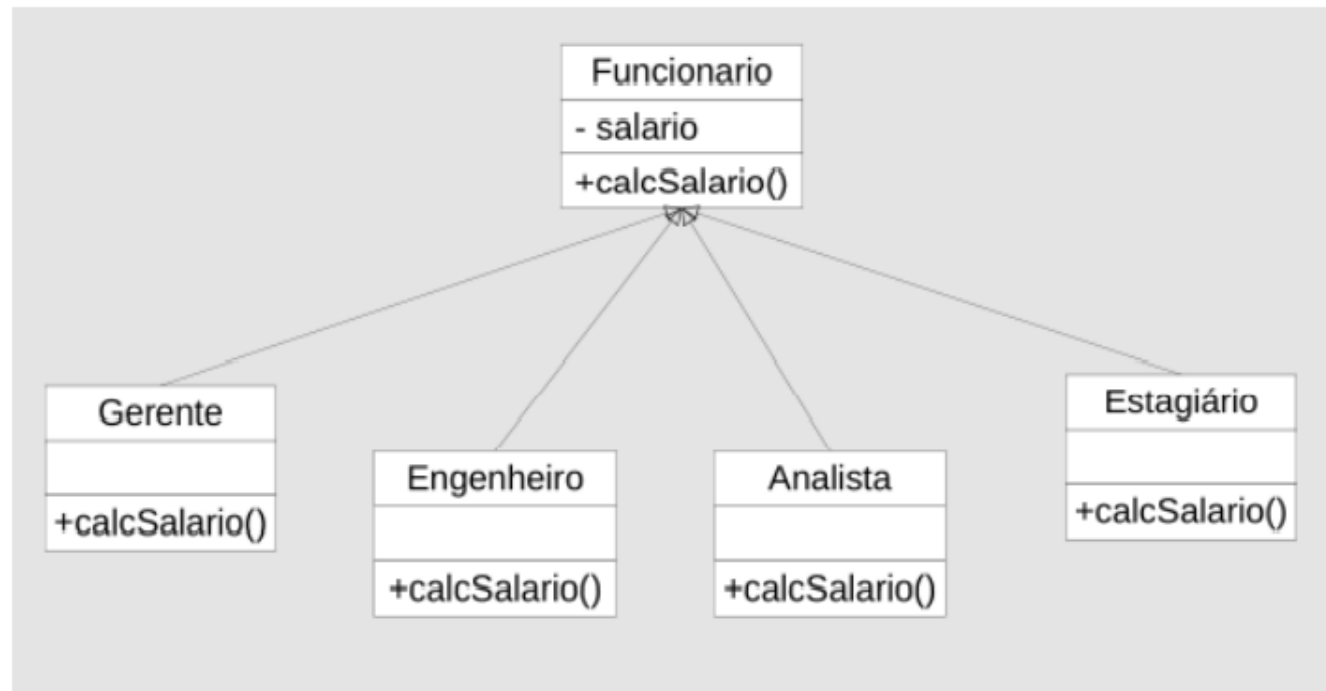


Diagrama de Classes em UML (*Unified Modeling Language*)

# Herança em Java

## Herança em Java

Encapsulamento

```
public class Funcionario {
    protected String nome;
    protected double salario;

    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public double getSalario() {
        return salario;
    }
    public void setSalario(double salario) {
        this.salario = salario;
    }
    public void printNome() {
        System.out.println("Nome: " + this.nome);
    }
    public void printSalario() {
        System.out.println("Salário: " + this.salario);
    }
    public void calcSalario() {
        this.salario = 1000;
    }
}
```

# Herança em Java

## Herança em Java

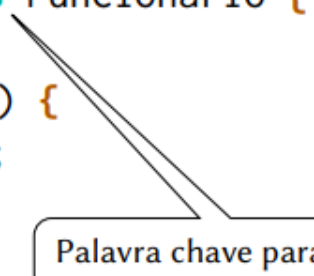
```
public class Gerente extends Funcionario {  
  
    public void calcSalario() {  
        this.salario = 20000;  
    }  
}
```

```
public class Engenheiro extends Funcionario {  
  
    public void calcSalario() {  
        this.salario = 10000;  
    }  
}
```

# Herança em Java

## Herança em Java

```
public class Gerente extends Funcionario {  
  
    public void calcSalario() {  
        this.salario = 20000;  
    }  
}
```



Palavra chave para herança

```
public class Engenheiro extends Funcionario {  
  
    public void calcSalario() {  
        this.salario = 10000;  
    }  
}
```

# Herança em Java

## Herança em Java

```
public class Analista extends Funcionario {  
  
    public void calcSalario() {  
        this.salario = 5000;  
    }  
}  
  
public class Estagiario extends Funcionario {  
  
}
```

# Herança em Java

## Herança em Java

```
public class Analista extends Funcionario {  
  
    public void calcSalario() {  
        this.salario = 5000;  
    }  
}  
  
public class Estagiario extends Funcionario {  
  
}
```

Não é necessário reimplementar a classe pai

# Herança em Java

## Herança em Java

```
public static void main(String[] args) {  
    Gerente g = new Gerente();  
    g.setNome("Pedro")  
    g.printNome();  
    g.calcSalario();  
    g.printSalario();  
  
    Engenheiro e = new Engenheiro();  
    e.setNome("Patricia")  
    e.printNome();  
    e.calcSalario();  
    e.printSalario();  
  
    Analista a = new Analista();  
    a.setNome("José")  
    a.printNome();  
    a.calcSalario();  
    a.printSalario();  
  
    Estagiario estag = new Estagiario();  
    estag.setNome("Julia")  
    estag.printNome();  
    estag.calcSalario();  
    estag.printSalario();  
}
```

# Herança em Java

## Herança em Java

- Saída do programa:

```
Nome: Pedro  
Salario: 20000.0  
Nome: Patricia  
Salario: 10000.0  
Nome: José  
Salario: 5000.0  
Nome: Julia  
Salario: 1000.0
```



## Modificadores de Acesso em Java

Modifier	Class	Package	Subclass	World
<code>public</code>	✓	✓	✓	✓
<code>protected</code>	✓	✓	✓	✗
<code>no modifier*</code>	✓	✓	✗	✗
<code>private</code>	✓	✗	✗	✗

# Herança em Java

## Exercícios

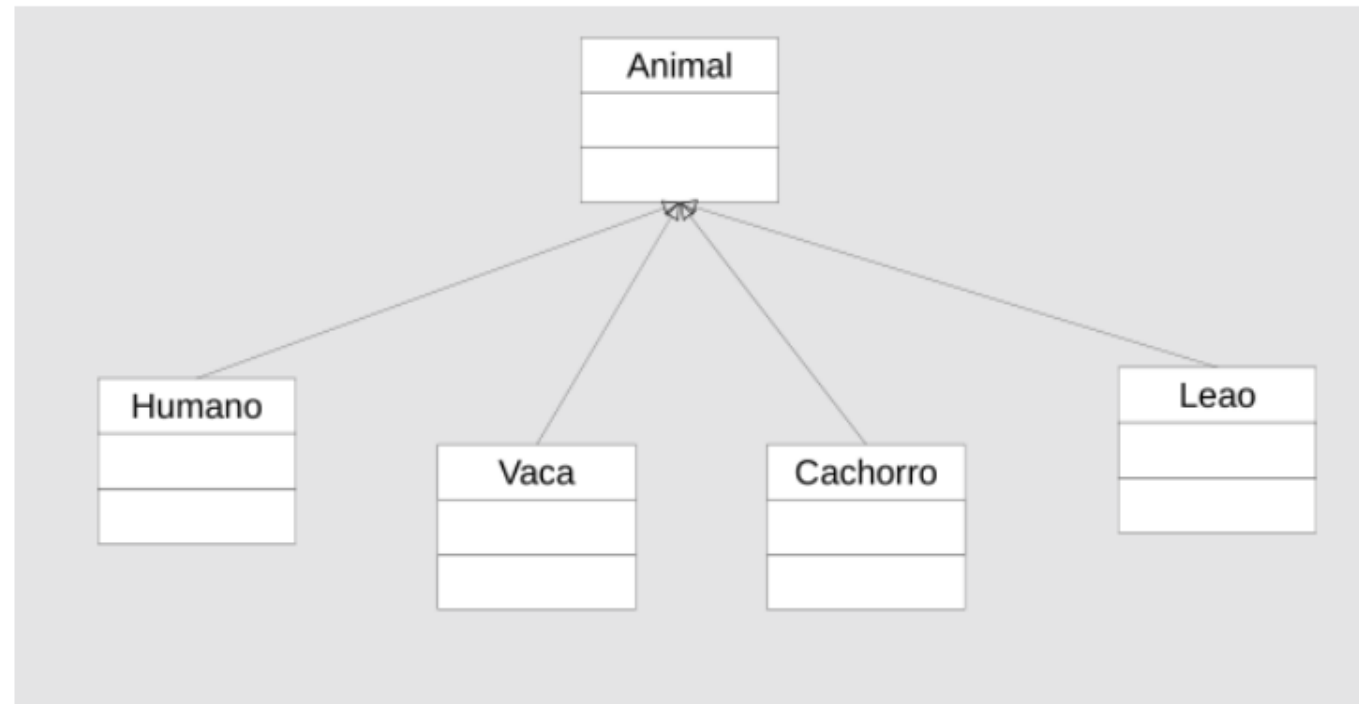
- **Exercício 2** - Faça um programa em Java para uma concessionária, onde ela vende carros, motos e caminhões. Utilize herança.

## Abstração

- Essa característica permite que grandes sistemas sejam especificados em um nível muito geral, muito antes de ocorrer a implementação dos métodos individuais.
- Classes que não podem ser instanciadas!
- Permite definir métodos sem implementação - que devem ser redefinidos nas subclasses.

# Classes Abstratas

## Abstração



# Classes Abstratas

## Abstração em Java

```
public abstract class Animal {  
    public abstract void falar()  
}  
  
public class Humano extends Animal {  
    public void falar() {  
        System.out.println("Eu posso falar - bla bla bla.");  
    }  
}  
  
public class Vaca extends Animal {  
    public void falar() {  
        System.out.println("Eu posso mugir - muuuuuu.");  
    }  
}
```

# Classes Abstratas

## Abstração em Java

```
public abstract class Animal {  
    public abstract void falar();  
}
```

Define a classe abstrata

Define o método abstrato

```
public class Humano extends Animal {  
    public void falar() {  
        System.out.println("Eu posso falar - bla bla bla.");  
    }  
}
```

```
public class Vaca extends Animal {  
    public void falar() {  
        System.out.println("Eu posso mugir - muuuuuu.");  
    }  
}
```

# Classes Abstratas

## Abstração em Java

```
public class Cachorro extends Animal {  
    public void falar() {  
        System.out.println("Eu posso latir - au au au.");  
    }  
}  
  
public class Leao extends Animal {  
    public void falar() {  
        System.out.println("Eu posso rugir - roooooaaaarr.");  
    }  
}
```

# Classes Abstratas

## Abstração em Java

```
public class Main {  
  
    public static void main (String[] args) {  
        Humano h = new Humano();  
        h.falar();  
  
        Vaca v = new Vaca();  
        v.falar();  
  
        Cachorro c = new Cachorro();  
        c.falar();  
  
        Leao l = new Leao();  
        l.falar();  
    }  
}
```



# Classes Abstratas

## Abstração em Java

- Será mostrado na tela:

Eu posso falar - bla bla bla.

Eu posso mugir - muuuuuu.

Eu posso latir - au au au.

Eu posso rugir - roooooaaaarr.

## Referências

George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. **Sistemas Distribuídos: Conceitos e Projeto**. Bookman Editora, 5 edition, 2013.

Harvey M Deitel, Paul J Deitel, David R Choffnes, et al. **Sistemas Operacionais**. Pearson/Prentice Hall, 3 edition, 2005.

Maarten Van Steen and A Tanenbaum. **Sistemas Distribuídos: Princípios e Paradigmas**. Pearson/Prentice Hall, 2 edition, 2007.

Harvey M Deitel and Paul J Deitel. **Java, como programar**. Ed. Pearson/Prentice Hall, 8 edition, 2010.

.....

# Obrigado!

Agradecimento pela parceria e elaboração de materiais aos professores:  
Prof. Me. Gustavo Torres Custódio  
Prof. Thiago Yamamoto

.....

Contato: [profmarcel.wagner@fiap.com.br](mailto:profmarcel.wagner@fiap.com.br)

Cursos:

Tecnologia em Análise e Desenvolvimento de Sistemas (TDS)

Tecnologia em Defesa Cibernética (TDC)

Engenharia de Software (ES)

