# Herança

Prof. Renato Pimentel

2024/2



## Sumário

Vamos retomar o conceito de generalização (herança) visto previamente.



## Herança I

No mundo real, através da genética, é comum herdarmos certas características e comportamentos de nossos pais.

Da mesma forma, em OO nossas classes também podem herdar atributos e comportamentos de uma classe já existente. Chamamos este processo de **herança**.

# Herança II

Herança permite a criação de classes com base numa classe já existente

**Objetivo**: proporcionar o reuso de software

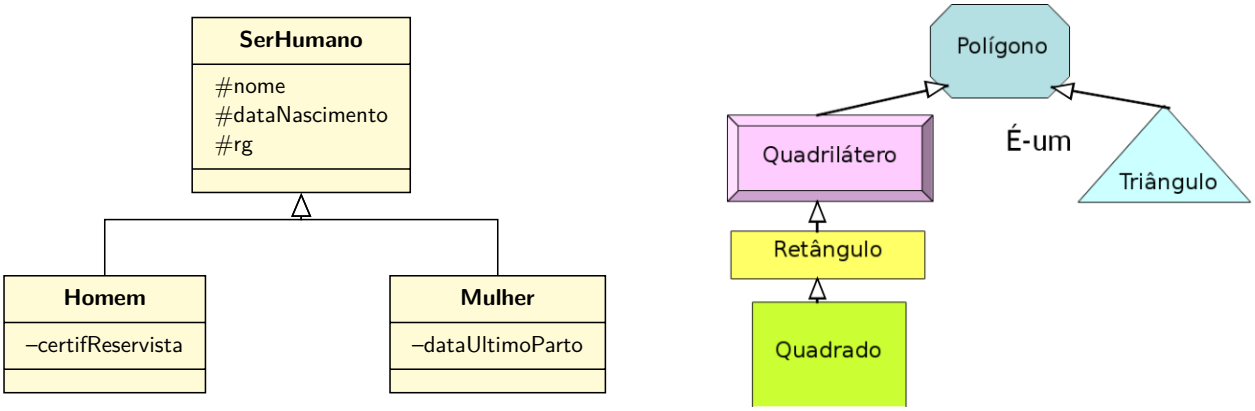
Herança é a capacidade de reusar código pela **especialização** de soluções genéricas já existentes;

- A ideia na herança é *ampliar* a funcionalidade de uma classe.

Todo objeto da subclasse também é um objeto da superclasse, mas **não** vice-versa.



# Herança III

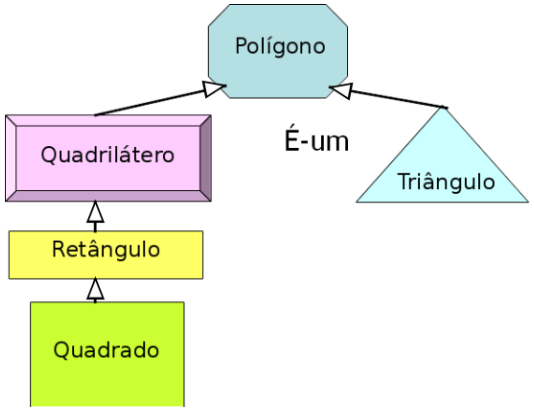



A representação gráfica do conceito de herança, na linguagem UML (*Unified Modeling Language*), é definida por retas com setas sem preenchimento apontando para a *classe-mãe*.

# Herança IV

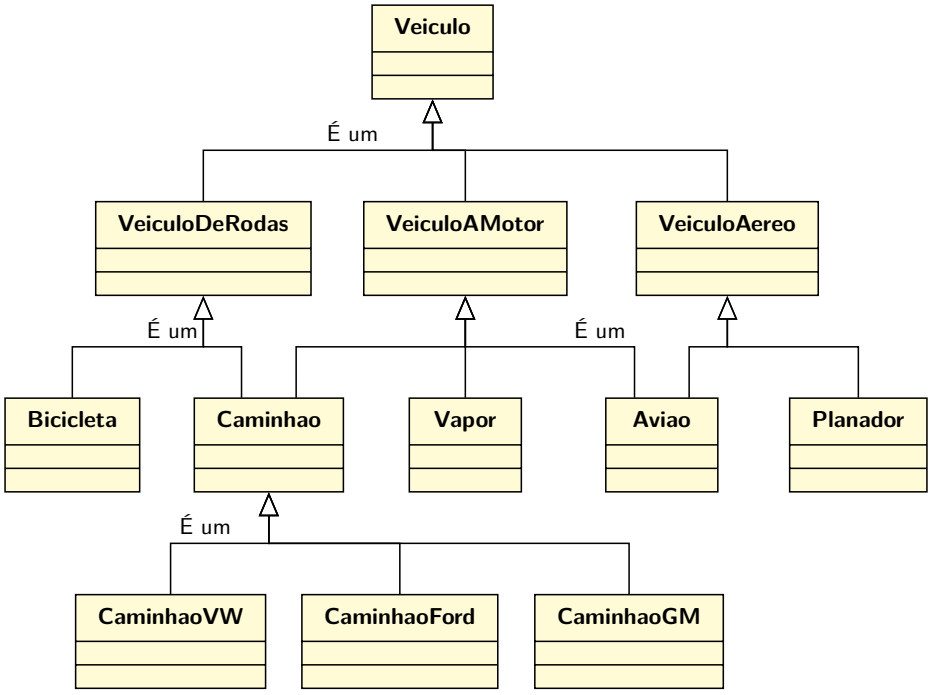
Herança representa um relacionamento de generalização entre classes:

- **É um**;
- **É um tipo de**.






# Herança V



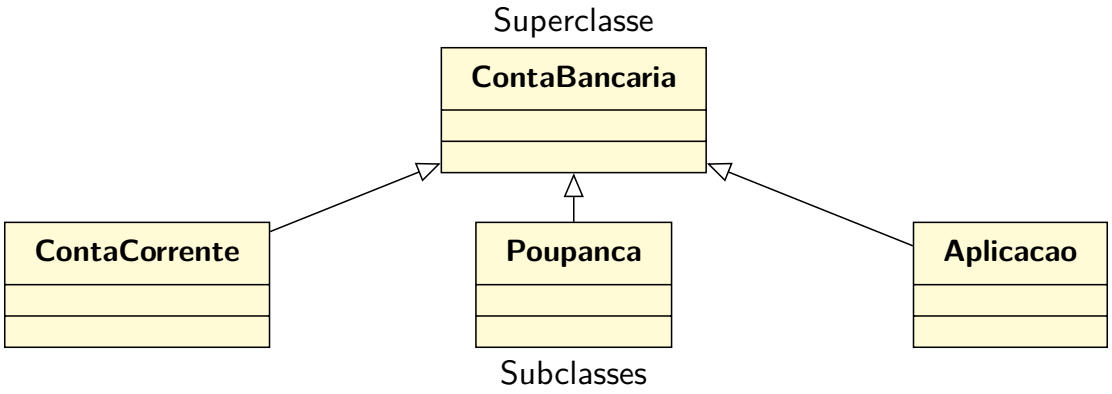
# Herança VI

Terminologia:

- **Classe mãe**, **superclasse**, **classe base**: A classe mais geral, a partir da qual outras classes herdam características (atributos e métodos);
- **Classe filha**, **subclasse**, **classe derivada**: A classe mais especializada, que herda características de uma classe mãe.



# Herança VII



# Herança e membros I

Uma subclasse **herda** atributos e métodos de sua superclasse, podendo possuir, no entanto, membros que lhe são próprios (exclusivos).

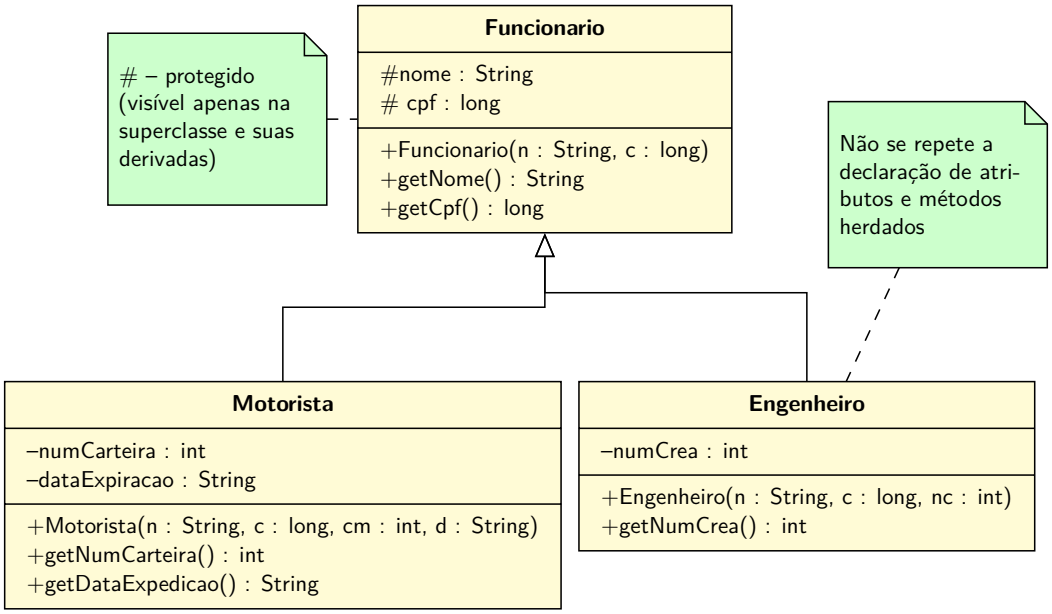
Acerca dos membros herdados pela subclasse:

- Tratados de forma semelhante a qualquer outro membro da mesma;
- Nem todos os membros da superclasse são acessíveis pela subclasse (encapsulamento) – se membro da superclasse é encapsulado como `private`, subclasse não o acessa;
- Na superclasse, tornam-se seus membros acessíveis *apenas* às subclasses usando o modificador de acesso `protected` (nos diagramas de classe UML: #) do Java.



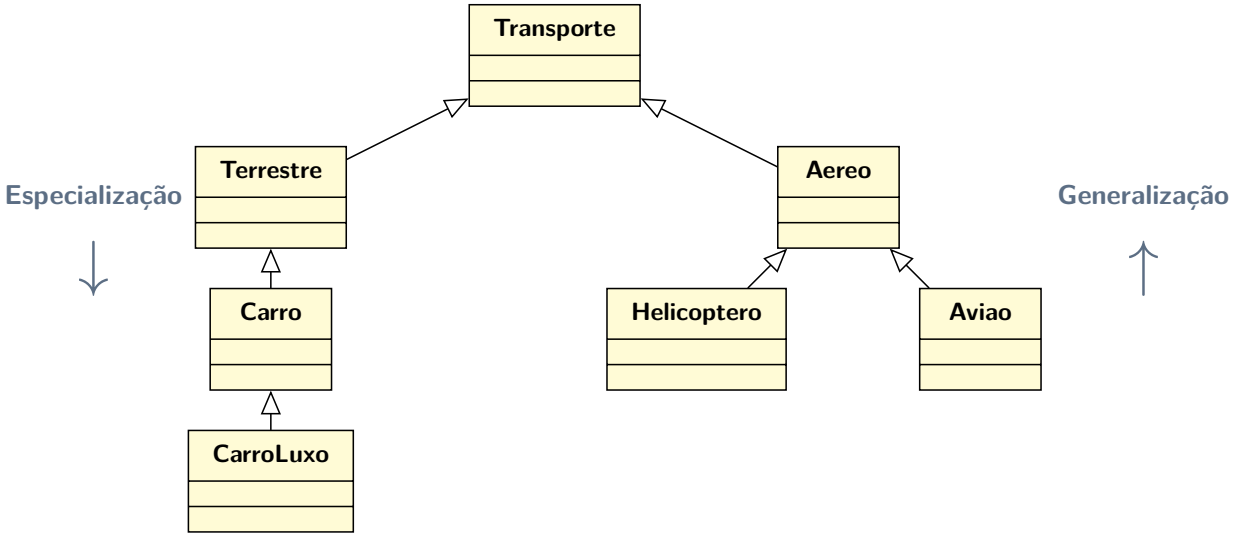
# Herança e membros II

Exemplo:



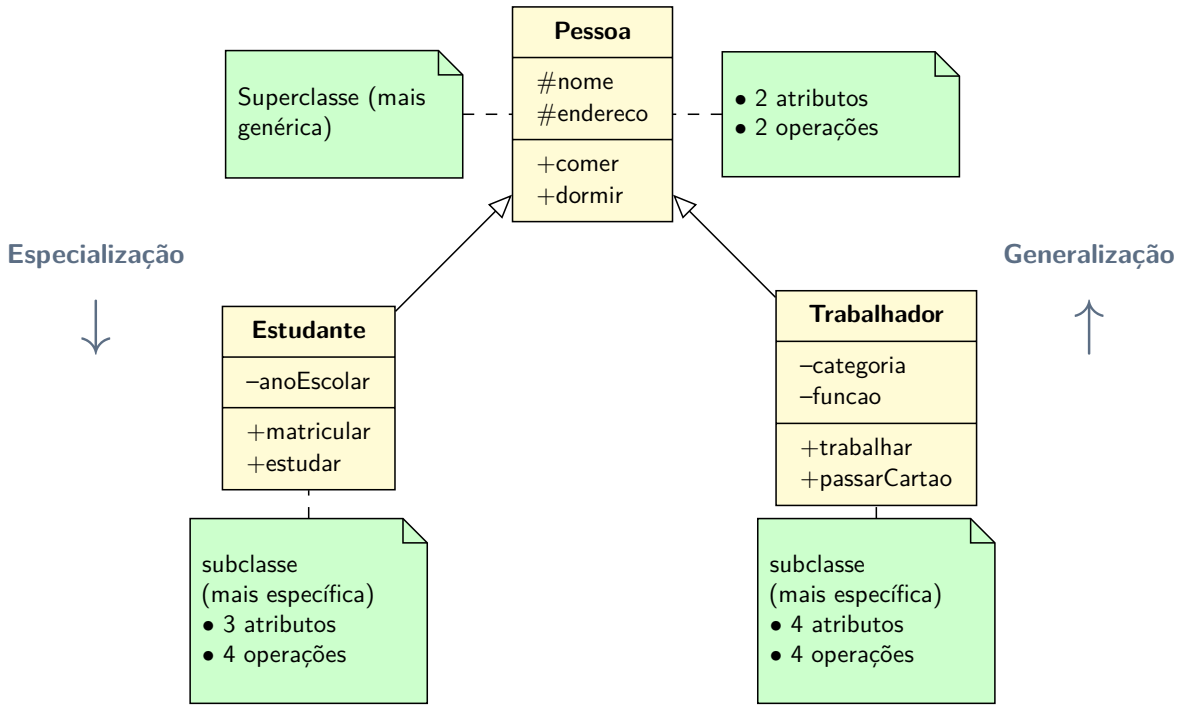
# Especialização vs. generalização I

Formas diferentes de se pensar a hierarquia de classes, e também de se modelar sistemas em POO.



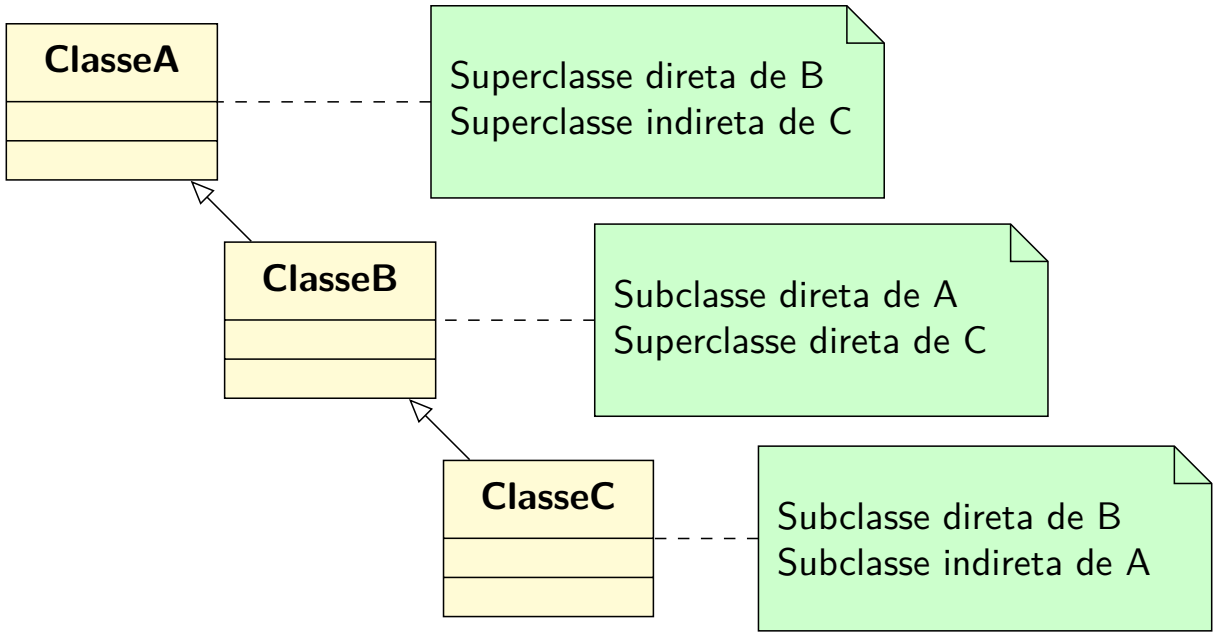



# Especialização vs. generalização II



# Hierarquia de classes

O processo de herança pode ser repetido *em cascata*, criando **várias gerações** de classes.



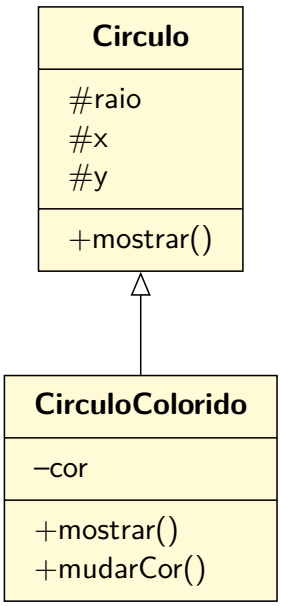



# Diferenciação I

Uma subclasse pode diferenciar-se de sua classe mãe:

- Com *operações adicionais*, diferentes das herdadas;
- Da mesma forma, com *atributos adicionais*;
- *Redefinindo*-se comportamentos existentes, oferecidos pela superclasse, contudo inapropriados para a subclasse (**overriding**)

Exemplo:




- A classe `CirculoColorido` herda da classe `Circulo` os atributos `raio`, `x` e `y`;
- Entretanto, define um novo atributo **cor**, **redefine** o método `mostrar()`, e implementa o método `mudarCor()`.



# Sobrescrita de métodos (*overriding*) I

A *redefinição* de um método herdado pela subclasse é feita ao definirmos na mesma um método *com mesmo nome, tipo de retorno e número de argumentos* (procedimento denominado **sobrescrita** ou **overriding**).

- **Ideia:** reimplementar o método definido previamente na superclasse de forma diferente, mais apropriada para a subclasse em questão.
- Ou seja, na classe derivada pode(m) haver método(s) com o mesmo nome de métodos da classe mãe, mas, por exemplo, com *funcionalidades diferentes*.

# Sobrescrita de métodos (*overriding*) II

## Sobrescrita e sobrecarga

Se o nome do método na subclasse for o mesmo de outro na classe mãe, mas os parâmetros forem diferentes, então ocorrerá uma **sobrecarga**, e não uma sobrescrita.

Outras observações importantes:

- O modificador de acesso do método da subclasse pode **relaxar o acesso**, mas não o contrário
  - Ex.: um método `protected` na superclasse pode se tornar `public` na subclasse; porém não `private`.
- Também há **sobrecarga** quando vários métodos numa mesma classe têm mesmo nome, mas diferentes parâmetros (comum para métodos construtores).

Por fim, uma subclasse não pode sobrescrever um *método de classe* (isto é, `static`) da superclasse.



# Vantagens

- **Reutilização** do código;
- Modificação de uma classe sem mudanças na classe original;
- É possível modificar uma classe para criar uma nova, de comportamento apenas ligeiramente diferente;
- Pode-se ter diversos objetos que executam ações diferentes, mesmo possuindo a **mesma origem**.

# Um exemplo prático I

Os professores de uma universidade dividem-se em 2 categorias:

- Professores em dedicação exclusiva (DE)
- Professores horistas

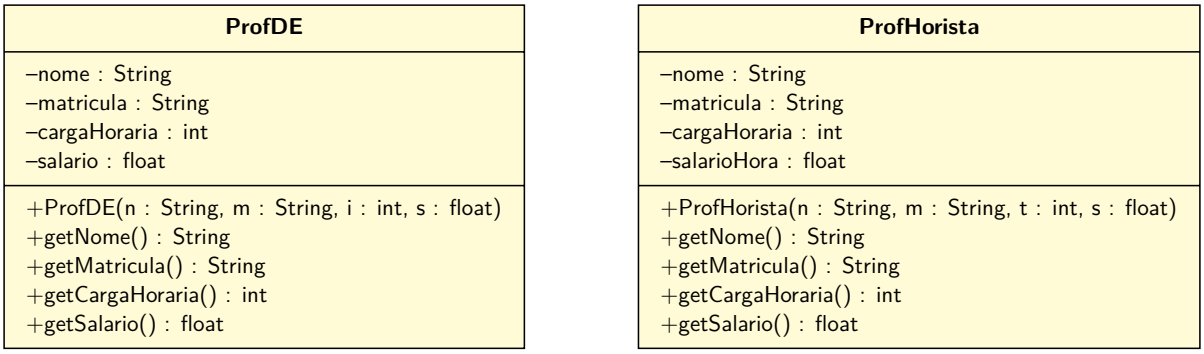
Considerações:

- Professores DE possuem um salário fixo para 40 horas de atividade semanais;
- Professores horistas recebem um valor por hora;
- O cadastro de professores desta universidade armazena o nome, idade, matrícula e informação de salário.



# Um exemplo prático II

Modelagem:



# Um exemplo prático III

Análise:

- As classes têm alguns atributos e métodos iguais;
- O que acontece se precisarmos alterar a representação de algum atributo, como por exemplo, o número de matrícula para inteiros ao invés de uma `String`?
  - Será necessário alterar os construtores e os métodos getMatricula() nas duas classes, o que é ruim para a programação: **código redundante**.
- Como resolver? **Herança**.

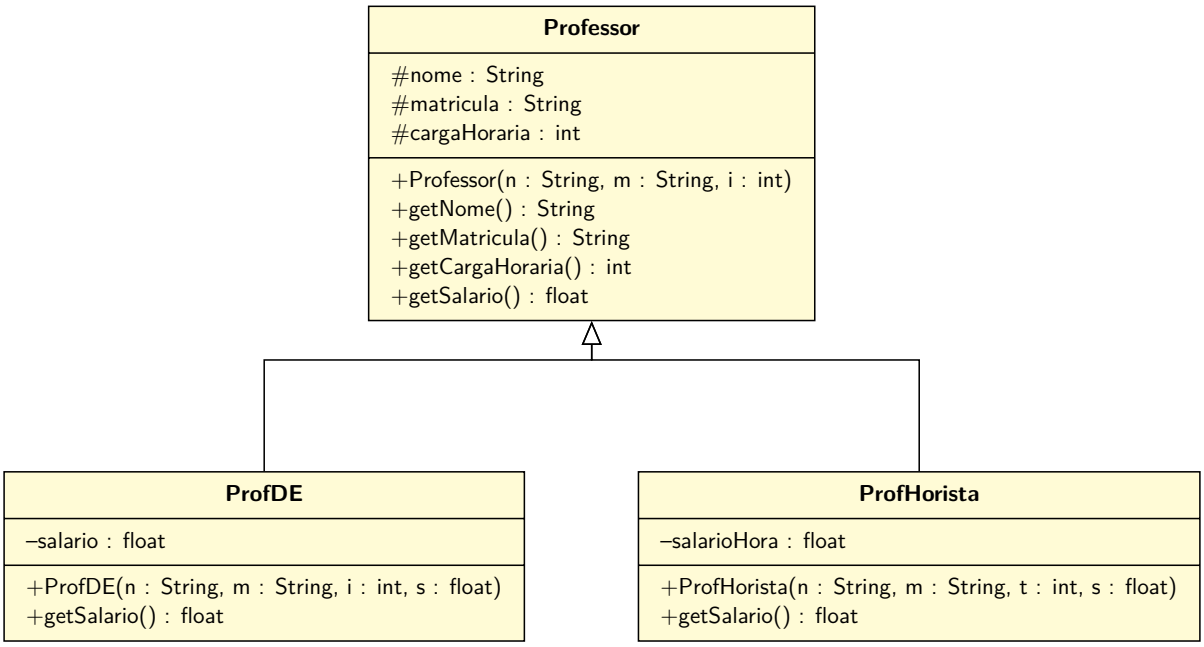


# Um exemplo prático IV

Sendo assim:

- Cria-se uma classe `Professor`, que contém os *membros* – atributos e métodos – comuns aos dois tipos de professor;
- A partir dela, cria-se duas novas classes, que representarão os professores horistas e DE;
- Para isso, essas classes deverão "herdar" os atributos e métodos declarados pela classe "pai", `Professor`.

# Um exemplo prático V






# super e this I

A palavra `this` é usada para referenciar membros do objeto em questão.

- É obrigatória quando há ambiguidades entre variáveis locais e de instância (atributos).

`super`, por sua vez, se refere à **superclasse**.

```
class Numero {
   public int x = 10;
}
```

```
class OutroNumero extends
   Numero {
   public int x = 20;
   public int total() {
      return this.x +
   super.x;
   }
}
```

# super e this II

`super()` e `this()` – *note os parênteses* – são usados **somente nos construtores**.

- `this()`: para chamar outro construtor, na mesma classe.

Exemplo:

```
public class Livro {
   private String titulo;
   public Livro() {
      this.titulo = "Sem titulo";
   }
   public Livro(String titulo) {
      this.titulo = titulo;
   }
}
```

Reimplementando:

```
1 public class Livro {
2    private String titulo;
3    public Livro() {
4       this("Sem titulo");
5    }
6    public Livro(String titulo) {
7       this.titulo = titulo;
8    }
9 }
```



# super e this III

- `super()`: para chamar construtor da classe base a partir de um construtor da classe derivada.

Exemplo:

```
1 class Ave extends Animal {
2    private int altura;
3    Ave() {
4       super();
5       altura = 0.0; // ou this.altura = 0.0;
6    }
7 }
```

# super e this IV

Algumas regras para uso de `super()`:

- Construtores da classe mãe são chamados pela palavra reservada `super`, seguida pelos argumentos a serem passados para o construtor entre parênteses.
  - Se não houver argumentos, basta usar `super()`.
- Construtores de superclasses somente poder ser invocados de dentro de construtores de subclasses, e **na primeira linha de código** dos mesmos.
- Se não houver, no construtor da subclasse, uma chamada explícita ao construtor da superclasse, o construtor sem argumento é chamado por padrão.
  - Se não houver construtor sem parâmetros na superclasse, o compilador apontará erro.



# super e this V

Vimos anteriormente um exemplo de uso de `super` para acessar um atributo da superclasse.
A palavra reservada `super` também pode ser usada para acessar métodos da superclasse. Algumas considerações:

- Outros métodos podem ser chamados pela palavra-chave `super` seguida de um ponto, do nome do método e argumento(s), se existente(s), entre parênteses:
  `super.nomeDoMetodo( [argumento(s)] );`
- Se a implementação do método for a mesma para a super e a subclasse – ou seja, não for uma sobrescrita – então instâncias da subclasse poderão chamar diretamente o método como se fosse de si próprias.

# Exercício I

Faça os diagramas de classes a partir das descrições:

1. Defina a classe `Produto`.
   - Os atributos de um produto são: código, descrição e quantidade, com a visibilidade protegida;
   - O construtor deve receber todos atributos por parâmetro;
   - A classe deve oferecer rotinas tipo acessoras (*getters*) para todos os campos;
   - Deve oferecer uma rotina onde se informa certa quantidade a ser retirada do estoque e outra onde se informa uma certa quantidade a ser acrescida ao estoque;
   - A rotina onde se informa uma quantidade a ser retirada do estoque deve retornar a quantidade que efetivamente foi retirada (para os casos em que havia menos produtos do que o solicitado).



# Exercício II

2. Defina a classe `ProdutoPerecivel`.
   - Esta deve ser **derivada** de Produto;
   - Possui um *atributo extra* que guarda a data de validade do produto;
   - As rotinas através das quais se informa as quantidades a serem retiradas ou acrescidas do estoque devem ser alteradas:
     - ★ A rotina de retirada deve receber também por parâmetro a data do dia corrente;
     - ★ Se os produtos já estiverem armazenados há mais de 2 meses a rotina deve zerar o estoque e devolver 0, pois produtos vencidos são descartados;
     - ★ A rotina de acréscimo no estoque só deve acrescentar os novos produtos caso o estoque esteja zerado, de maneira a evitar misturar produtos com prazos de validade diferenciados.

# Exercício III

3. Defina a classe `ProdutoPerEsp`.
   - Esta é derivada de `ProdutoPerecivel`;
   - Oferece uma rotina de impressão de dados capaz de imprimir uma nota de controle onde consta o código, a descrição, a quantidade em estoque e a data de validade do produto.
4. Defina a classe `ProdutoComPreco`.
   - Esta é derivada de `Produto`;
   - Deve possuir campos para armazenar o preço unitário do produto;
   - A classe deve oferecer rotinas para permitir obter e alterar o preço unitário (sempre positivo).



# Exercício IV

5. Defina a classe `Estoque`.
   - Esta mantém uma lista com os produtos em estoque (do tipo `ProdutoComPreco`);
   - A classe deve ter métodos para cadastrar e consultar produtos, inseri-los e retirá-los do estoque, bem como para informar o custo total do estoque armazenado.

# Referências


1. Addison-BOOCH, G., RUMBAUGH, J., JACOBSON, I. *UML, Guia do Usuário*. Rio de Janeiro: Campus, 2000.
2. FOWLER, M. *UML Essencial*, 2a Edição. Porto Alegre: Bookman, 2000.
3. LARMAN, C. *Utilizando UML e Padrões: Uma Introdução à Análise e ao Projeto Orientado a Objetos*. Porto Alegre: Bookman, 2001.


Os slides dessa apresentação foram cedidos por:

- Graça Marietto e Francisco Zampirolli, UFABC
- Profa Katti Faceli, UFSCar/Sorocaba
- Marcelo Z. do Nascimento, FACOM/UFU

LaTeXagem e adaptações: Renato Pimentel, FACOM/UFU