# Faculdade de Computação - FACOM

## Bacharelado em Sistemas de Informação

*FACOM32305 - Programação Orientada a Objetos*

Prof. Thiago Pirola Ribeiro

# Sumário

Antes de apresentarmos relacionamentos entre classes na orientação a objetos, é importante falarmos sobre a representação visual, através do uso de UML.

# UML – Linguagem de modelagem unificada I

- UML (*Unified Modeling Language*) – linguagem visual muito utilizada nas etapas de análise e projeto de sistemas computacionais no paradigma de orientação a objetos
- A UML se tornou a linguagem padrão de projeto de software, adotada internacionalmente pela indústria de Engenharia de Software.

# UML – Linguagem de modelagem unificada II

- UML não é uma linguagem de programação: é uma linguagem de modelagem utilizada para representar o sistema de software sob os seguintes parâmetros:
  - Requisitos
  - Comportamento
  - Estrutura lógica
  - Dinâmica de processos
  - Comunicação/interface com os usuários

# UML – Linguagem de modelagem unificada III

- O objetivo da UML é fornecer múltiplas visões do sistema que se deseja modelar, representadas pelos diagramas UML;
- Cada diagrama analisa o sistema sob um determinado aspecto, sendo possível ter enfoques mais amplos (externos) do sistema, bem como mais específicos (internos).

# Diagramas UML típicos

- Diagrama de casos e usos
- Diagrama de classes
- Diagrama de objetos
- Diagrama de sequência
- Diagrama de colaboração
- Diagrama de estado
- Diagrama de atividades

# Softwares para UML

Algumas ferramentas para criação de diagramas UML:

- Rational Rose – a primeira ferramenta case a fornecer suporte UML
- Yed
- StarUML
- Dia
- Argo UML
- Microsoft Visio - está no pacote do Office365 - UFU
- Enterprise Architect
- LucidChart (online)

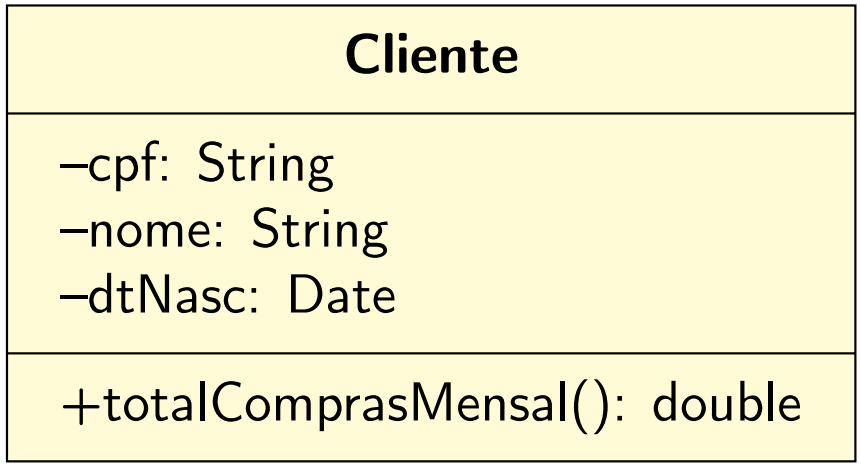
# Diagrama de classes I

- Diagrama de classes é o mais utilizado da UML
- Objetivos:
  - Ilustrar as classes principais do sistema;
  - Ilustrar o relacionamento entre os objetos.
- Apresentação da estrutura lógica: classes, relacionamentos.

# Diagrama de classes II

Nos diagramas UML, cada classe é dada por um retângulo dividido em três partes:

1. O nome da classe;
2. Seus atributos;
3. Seus métodos.



# Diagrama de classes III

Atributos no diagrama de classe:
visibilidade nome: tipo = valor inicial {propriedades}

- Visibilidade: + para `public`, − para `private`, # para `protected`;
- Tipo do atributo: Ex: `int`, `double`, `String`, `Date`;
- Valor inicial definido no momento da criação do objeto;
- Propriedades. Ex.: {`readonly`, `ordered`}

# Diagrama de classes IV

Métodos no diagrama de classe:
visibilidade nome (par1: tipo1, par2: tipo2, ...): tipo

- Visibilidade: + para `public`, − para `private`, # para `protected`;
- (par1: tipo1, par2:tipo2, ...): Se método contém parâmetros formais (argumentos), o nome e o tipo de cada 1. Ex: `(nome: String, endereco: String, codigo: int)`;
  - Se método não contém parâmetros, manter par de parênteses, vazio.
- tipo: tipo de retorno do método. Ex.: `void`, se não retorna nada; `int`, `Cliente` (nome de uma classe), etc.

# Diagrama de classes V

Mais detalhes sobre UML:
http://www.uml.org/what-is-uml.htm

# Relacionamentos entre classes

# Relacionamentos entre classes

- Como os objetos das diferentes classes se relacionam?

# Relacionamentos entre classes

- Como os objetos das diferentes classes se relacionam?
- Como diferentes classes se relacionam?

# Relacionamentos entre classes

- Como os objetos das diferentes classes se relacionam?
- Como diferentes classes se relacionam?
- Vamos identificar as principais formas de conexão entre classes:

# Relacionamentos entre classes

- Como os objetos das diferentes classes se relacionam?
- Como diferentes classes se relacionam?
- Vamos identificar as principais formas de conexão entre classes:
  - E representar esses relacionamentos no diagrama de classes;

# Relacionamentos entre classes

- Como os objetos das diferentes classes se relacionam?
- Como diferentes classes se relacionam?
- Vamos identificar as principais formas de conexão entre classes:
  - E representar esses relacionamentos no diagrama de classes;
  - Relações fornecem um caminho para a comunicação entre os objetos.

# Relacionamentos I

Tipos mais comuns:

- Entre objetos de diferentes classes:
  - Associação – "usa";
  - Agregação – "é parte de";
  - Composição – "é parte essencial de".
- Entre classes:
  - Generalização – "É um"

# Generalização ou Herança

# Generalização ou herança

- Representa relacionamentos entre classes do tipo "é um".

# Generalização ou herança

- Representa relacionamentos entre classes do tipo "é um".
  - Também chamado de herança.
    Exemplo: Um cachorro é um mamífero

# Generalização ou herança

- Representa relacionamentos entre classes do tipo "é um".
  - Também chamado de herança.
    Exemplo: Um cachorro é um mamífero

- Abstração de Generalização/Especialização:

# Generalização ou herança

- Representa relacionamentos entre classes do tipo "é um".
  - Também chamado de herança.
    Exemplo: Um cachorro é um mamífero

- Abstração de Generalização/Especialização:
  - A partir de duas ou mais classes, abstrai-se uma classe mais genérica;

# Generalização ou herança

- Representa relacionamentos entre classes do tipo "é um".
  - Também chamado de herança.
    Exemplo: Um cachorro é um mamífero

- Abstração de Generalização/Especialização:
  - A partir de duas ou mais classes, abstrai-se uma classe mais genérica;
    - Ou de uma classe geral, deriva-se outra mais específica.

# Generalização ou herança

- Representa relacionamentos entre classes do tipo "é um".
  - Também chamado de herança.
    Exemplo: Um cachorro é um mamífero

- Abstração de Generalização/Especialização:
  - A partir de duas ou mais classes, abstrai-se uma classe mais genérica;
    - Ou de uma classe geral, deriva-se outra mais específica.

  - Subclasses satisfazem todas as propriedades das classes as quais as mesmas constituem especializações;

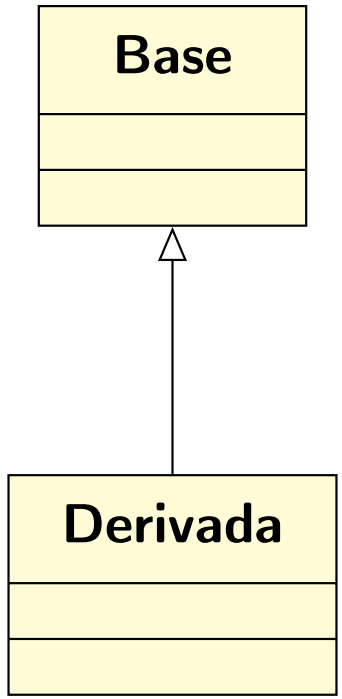
# Generalização ou herança

- Representa relacionamentos entre classes do tipo "é um".
  - Também chamado de herança.
    Exemplo: Um cachorro é um mamífero

- Abstração de Generalização/Especialização:
  - A partir de duas ou mais classes, abstrai-se uma classe mais genérica;
    - Ou de uma classe geral, deriva-se outra mais específica.
  - Subclasses satisfazem todas as propriedades das classes as quais as mesmas constituem especializações;
  - Deve haver ao menos uma propriedade que distingue duas classes especializadas.

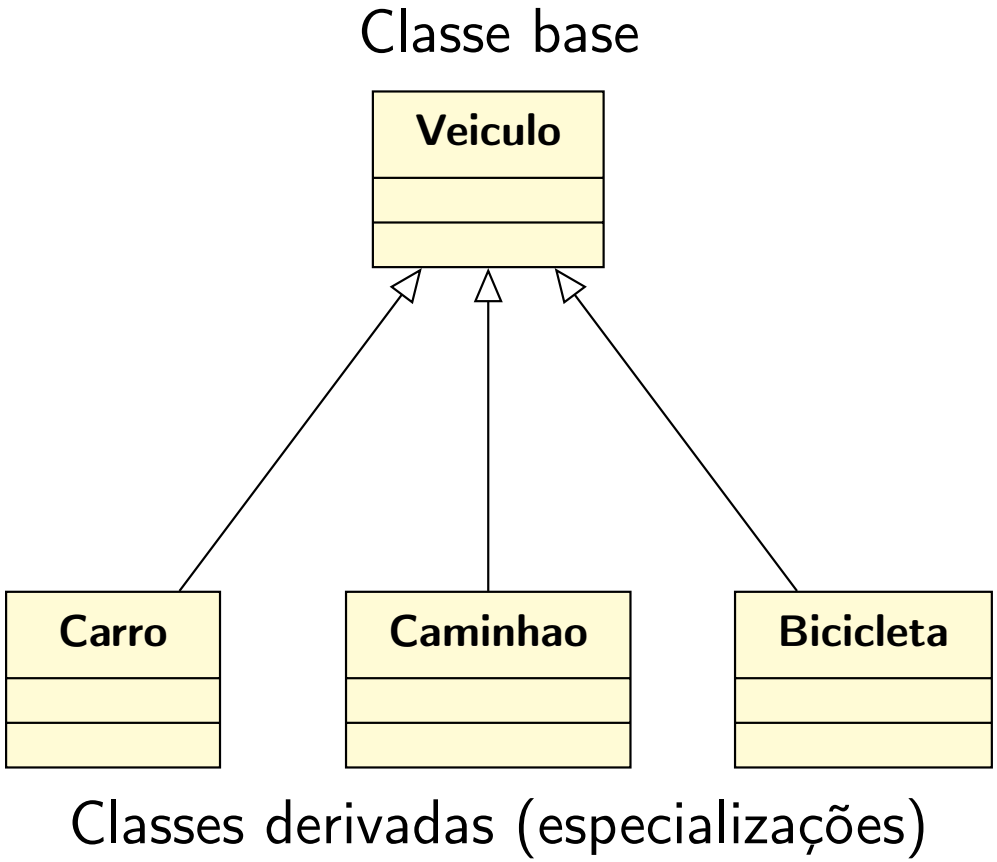
# Generalização ou herança I

No diagrama de classes, a generalização é representada por uma seta para o lado da classe mais geral (classe base).



# Generalização ou herança II

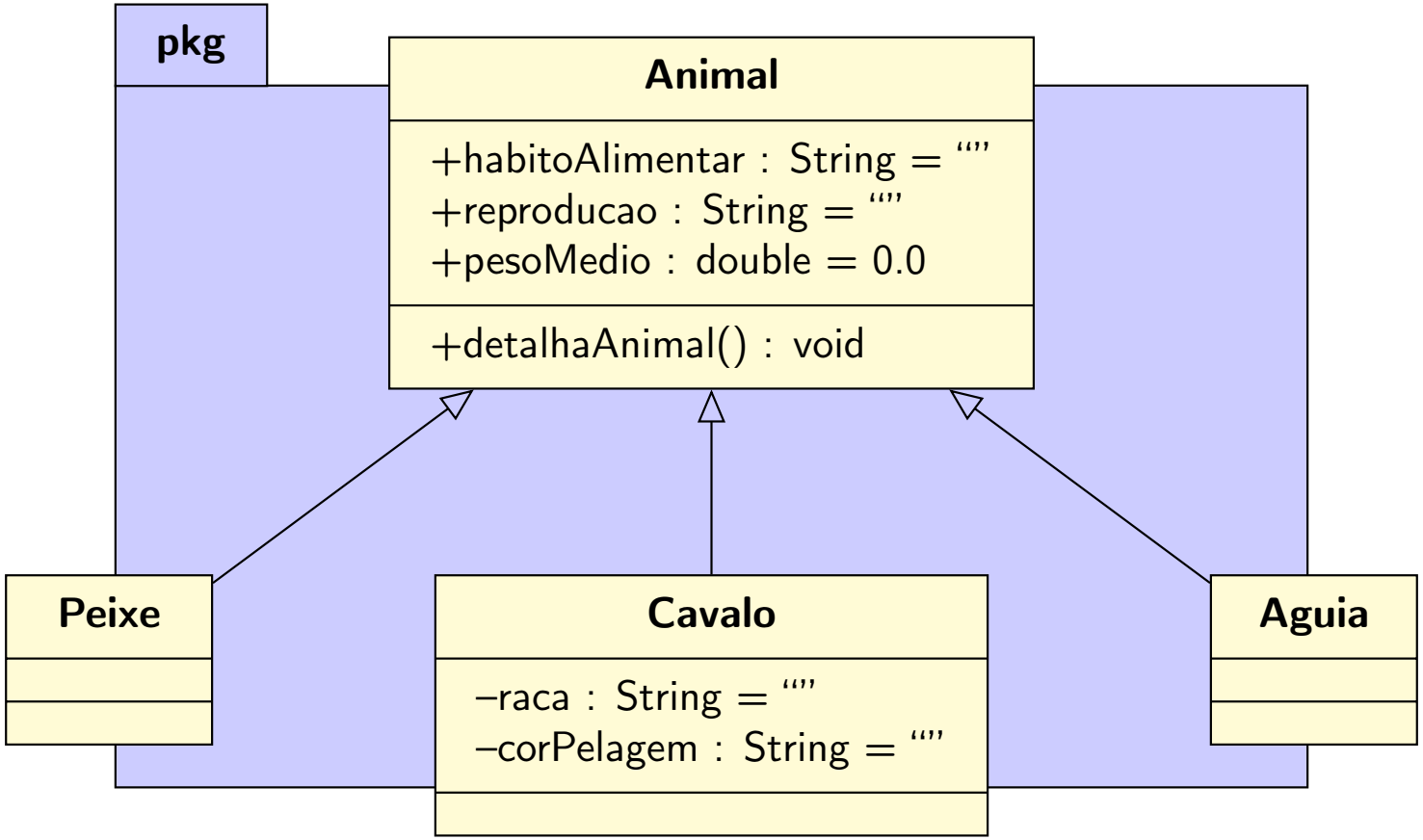
Exemplos:



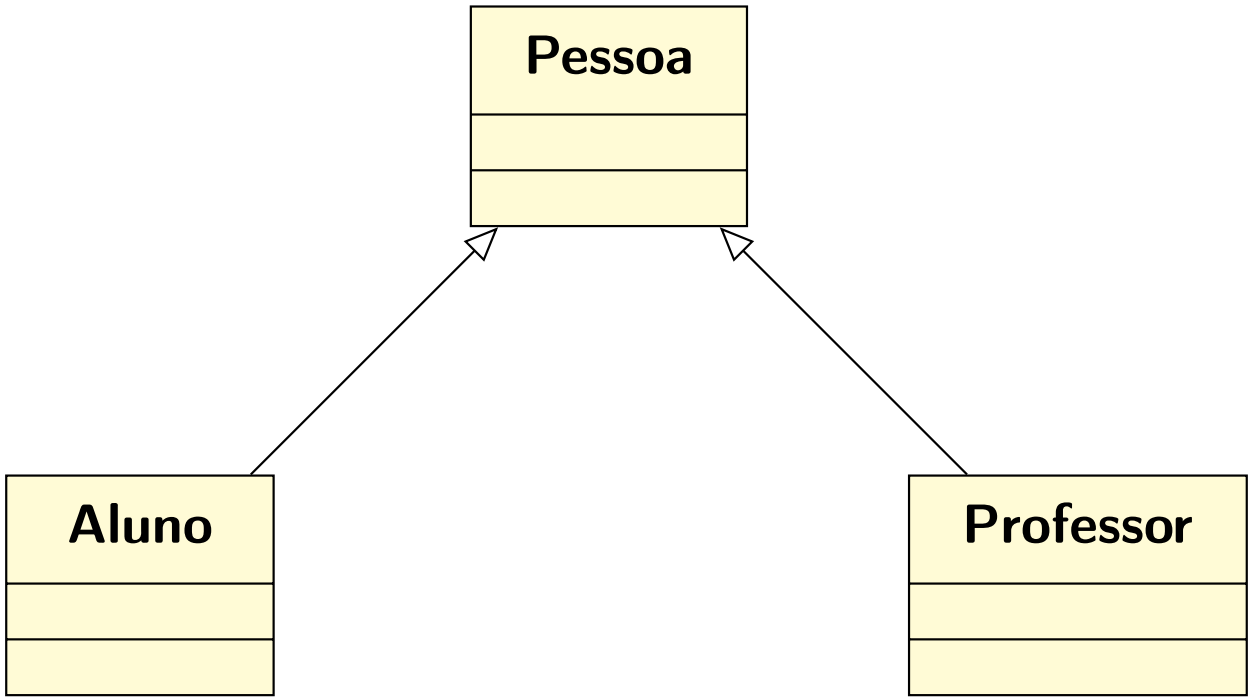

⇑
Generalização

Especialização
(herança)
⇓

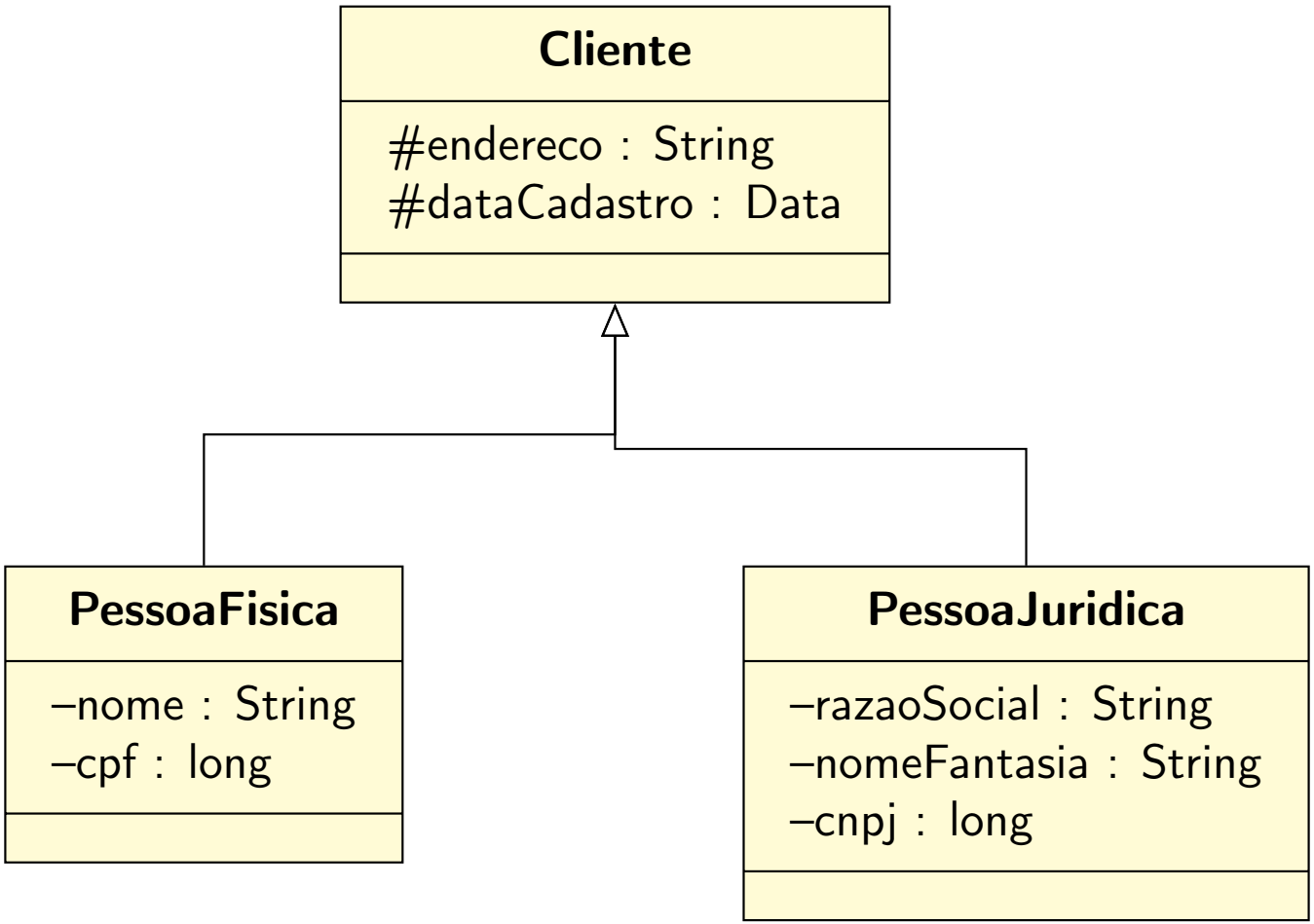
# Generalização ou herança III



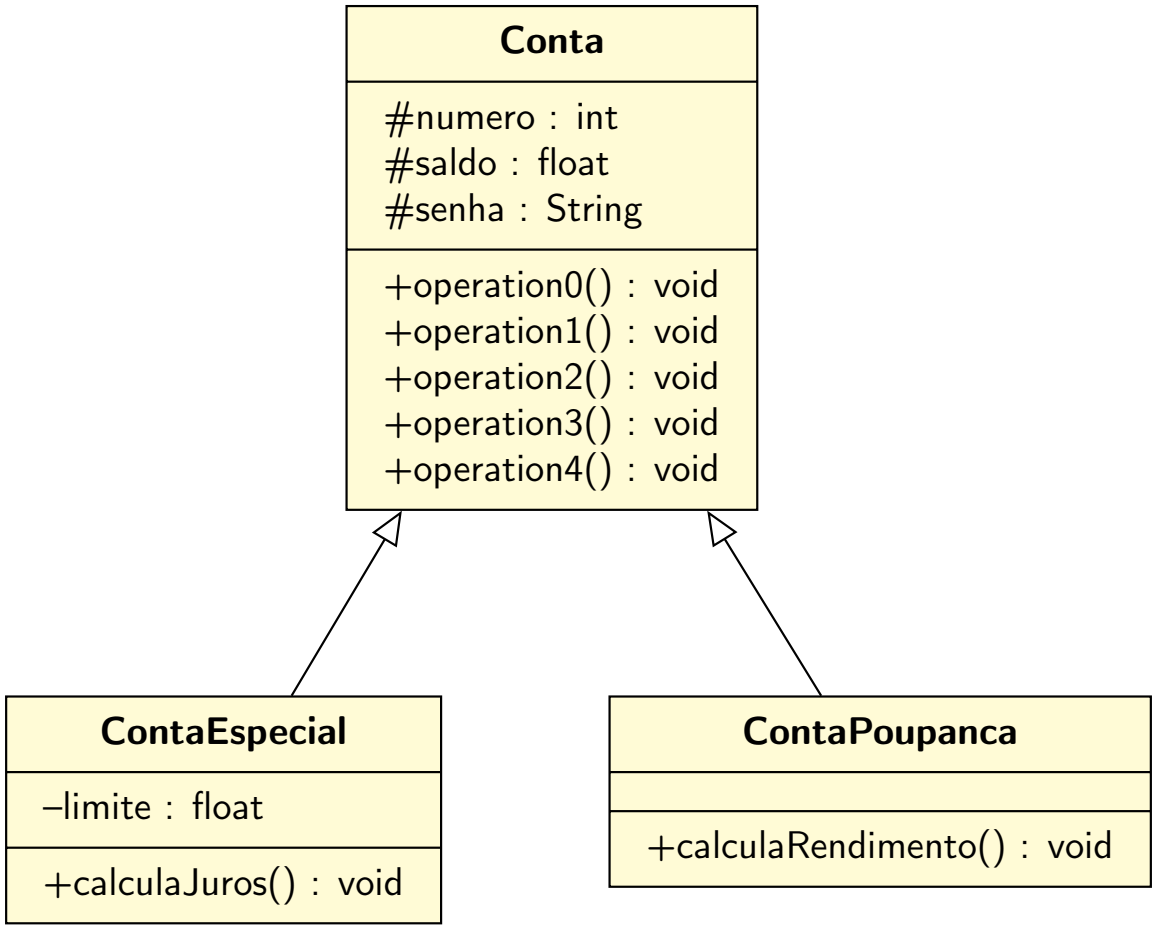
# Generalização ou herança IV



# Generalização ou herança V



# Generalização ou herança VI



# Generalização ou herança VII

- A generalização permite organizar as classes de objetos hierarquicamente;

- Ainda, consiste numa forma de reutilização de software:
  - Novas classes são criadas a partir de existentes, absorvendo seus atributos e comportamentos, acrescentando recursos que necessitem

# Sumário

# Mais sobre Herança

- No mundo real, através da Genética, é possível herdarmos certas características de nossos pais.

# Mais sobre Herança

- No mundo real, através da Genética, é possível herdarmos certas características de nossos pais.
- De forma geral, herdamos atributos e comportamentos de nossos pais.

# Mais sobre Herança

- No mundo real, através da Genética, é possível herdarmos certas características de nossos pais.
- De forma geral, herdamos atributos e comportamentos de nossos pais.

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

# Mais sobre Herança

- Herança permite a criação de classes com base em uma classe já existente

# Mais sobre Herança

- Herança permite a criação de classes com base em uma classe já existente
- Objetivo: proporcionar o reuso de software

# Mais sobre Herança

- Herança permite a criação de classes com base em uma classe já existente
- Objetivo: proporcionar o reuso de software
- Herança é a capacidade de reusar código pela especialização de soluções genéricas já existentes

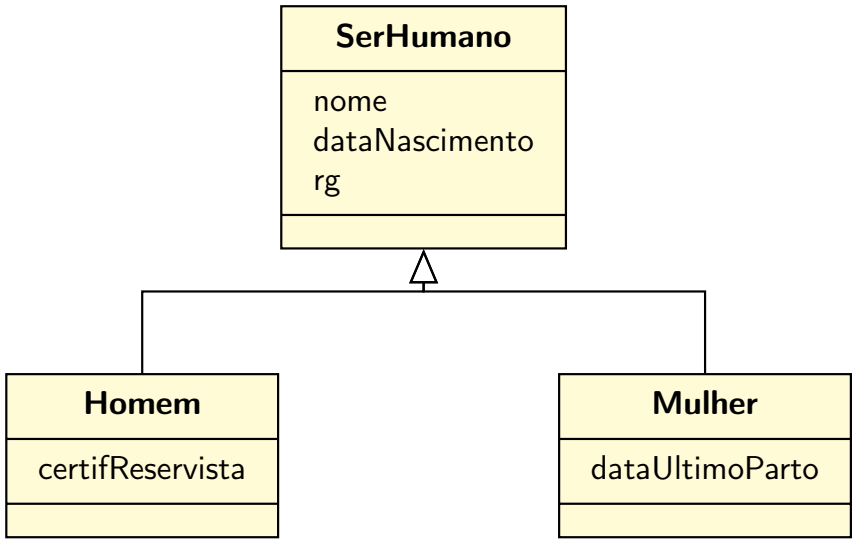
# Mais sobre Herança

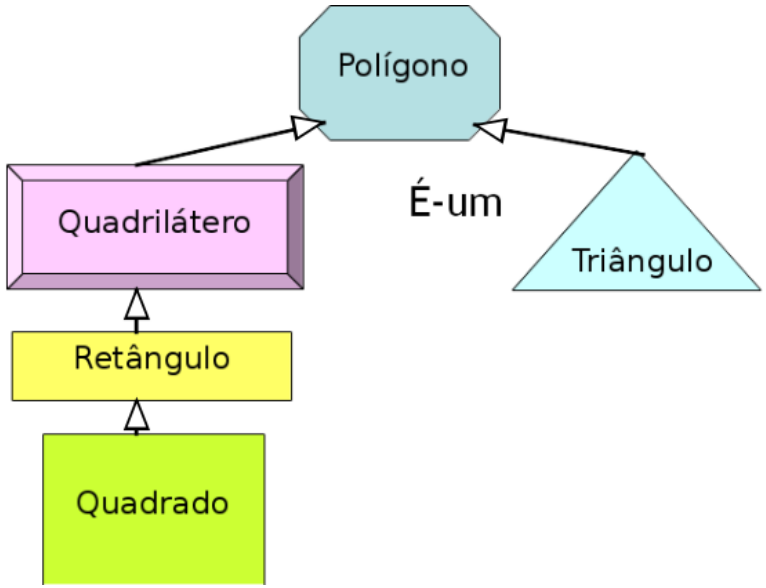
- Herança permite a criação de classes com base em uma classe já existente
- Objetivo: proporcionar o reuso de software
- Herança é a capacidade de reusar código pela especialização de soluções genéricas já existentes
  - A ideia na herança é *ampliar* a funcionalidade de uma classe

# Mais sobre Herança

- Herança permite a criação de classes com base em uma 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.

# Mais sobre Herança



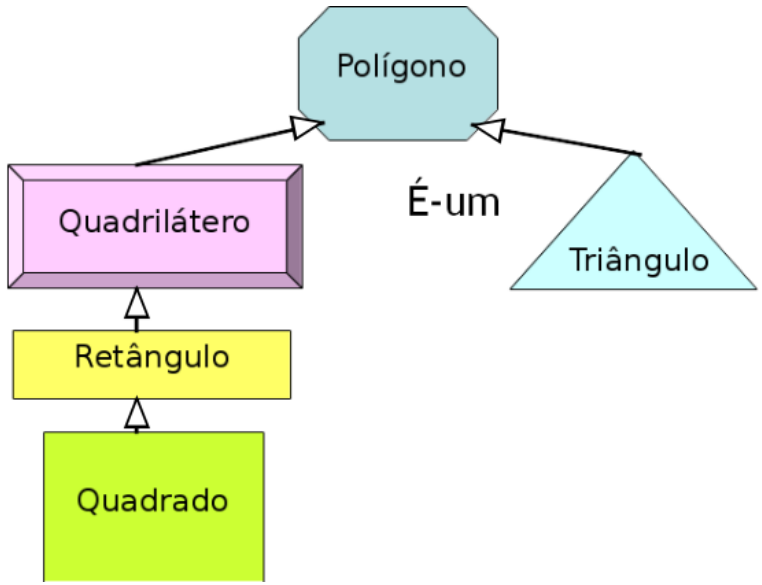




A representação gráfica do conceito de herança, na linguagem UML (*Unified Modeling Language*), é definida por retas com setas apontando para a *classe-mãe*.

# Mais sobre Herança

Herança representa um relacionamento de generalização entre classes:

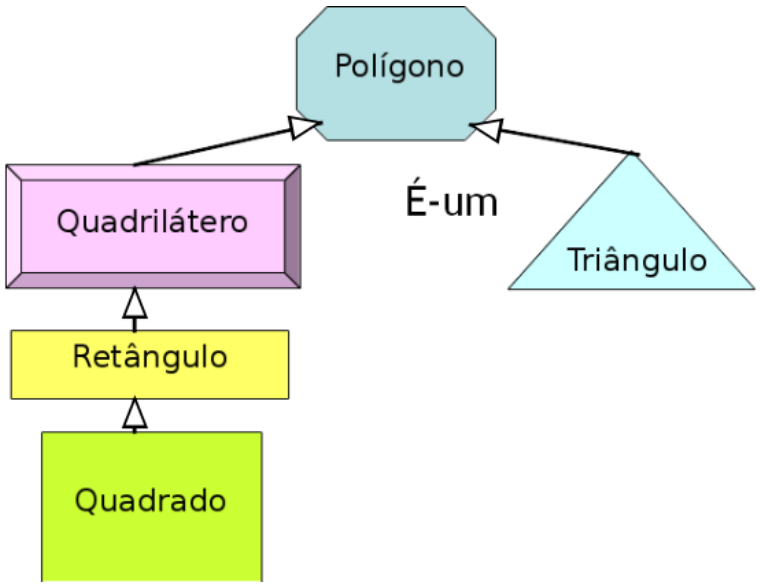
- É um;



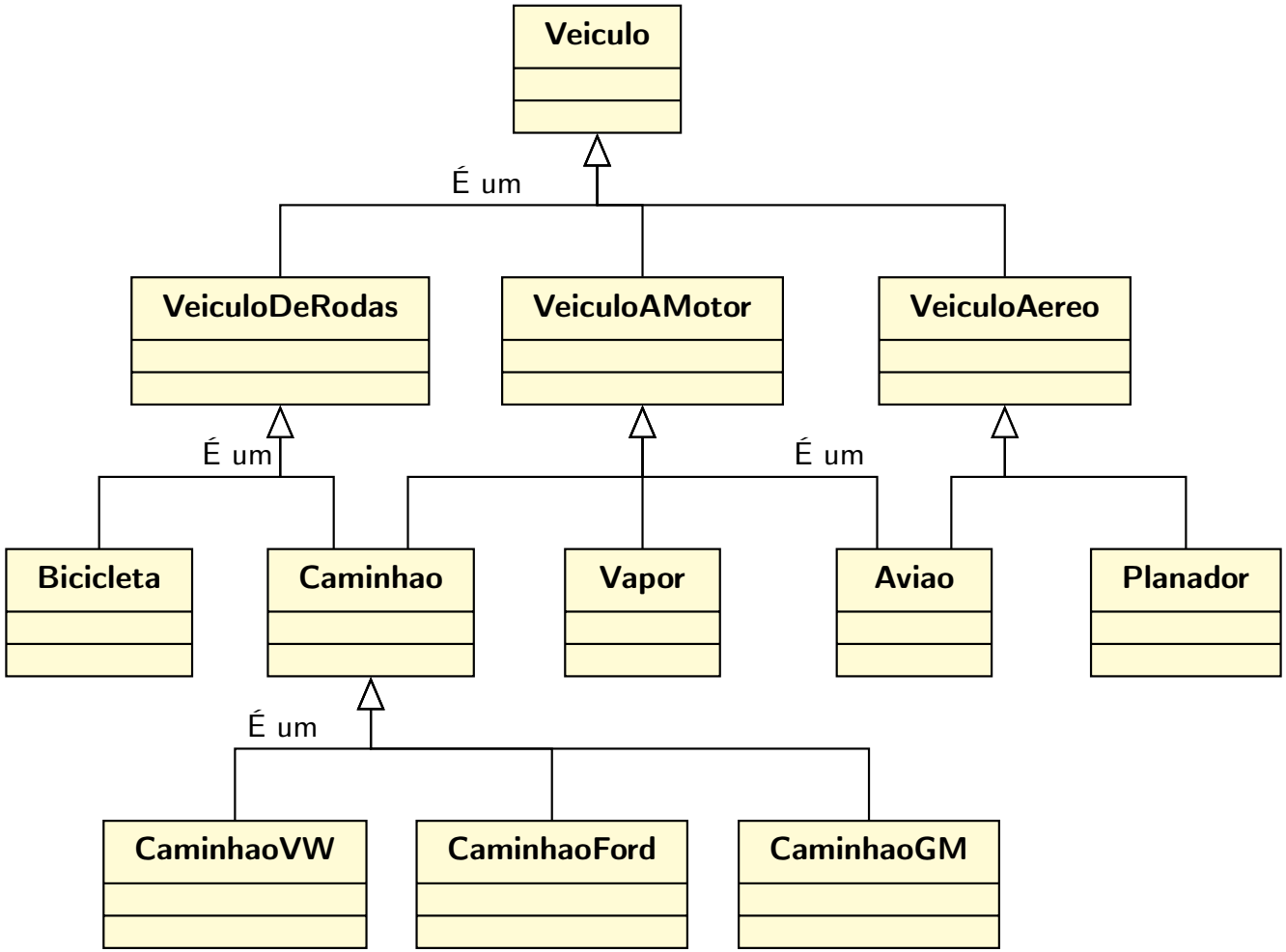
# Mais sobre Herança

Herança representa um relacionamento de generalização entre classes:

- É um;
- É um tipo de.



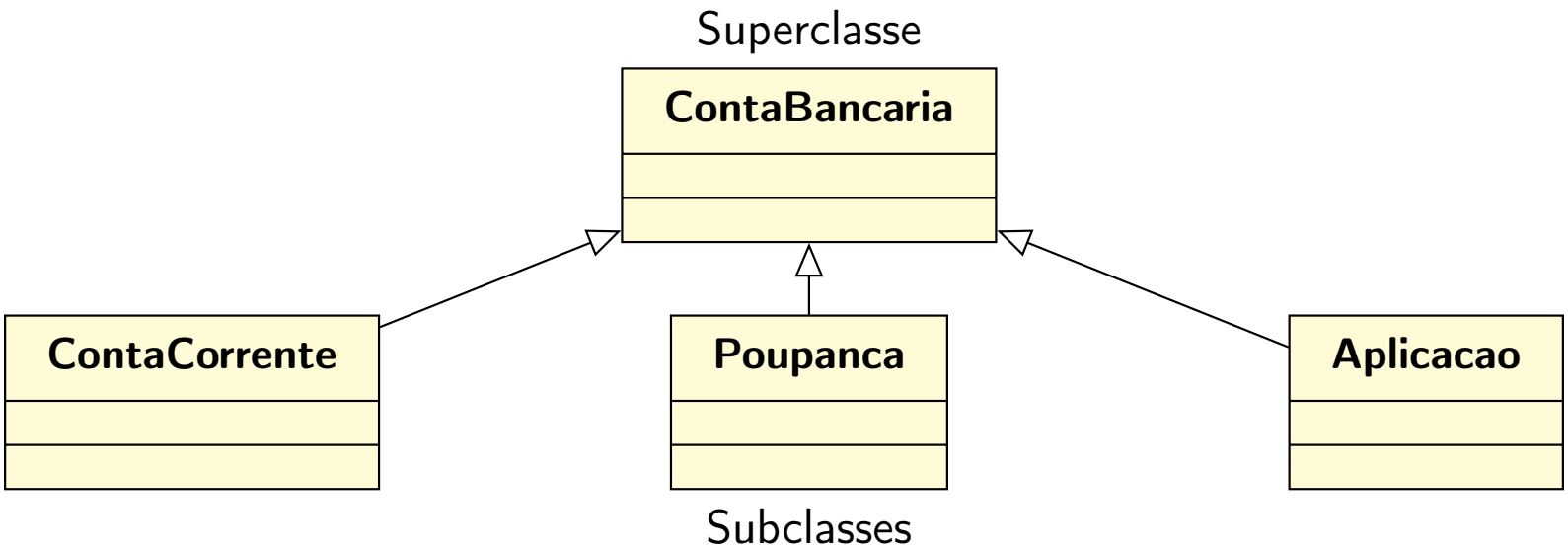
# Mais sobre Herança



# Mais sobre Herança

Terminologia:

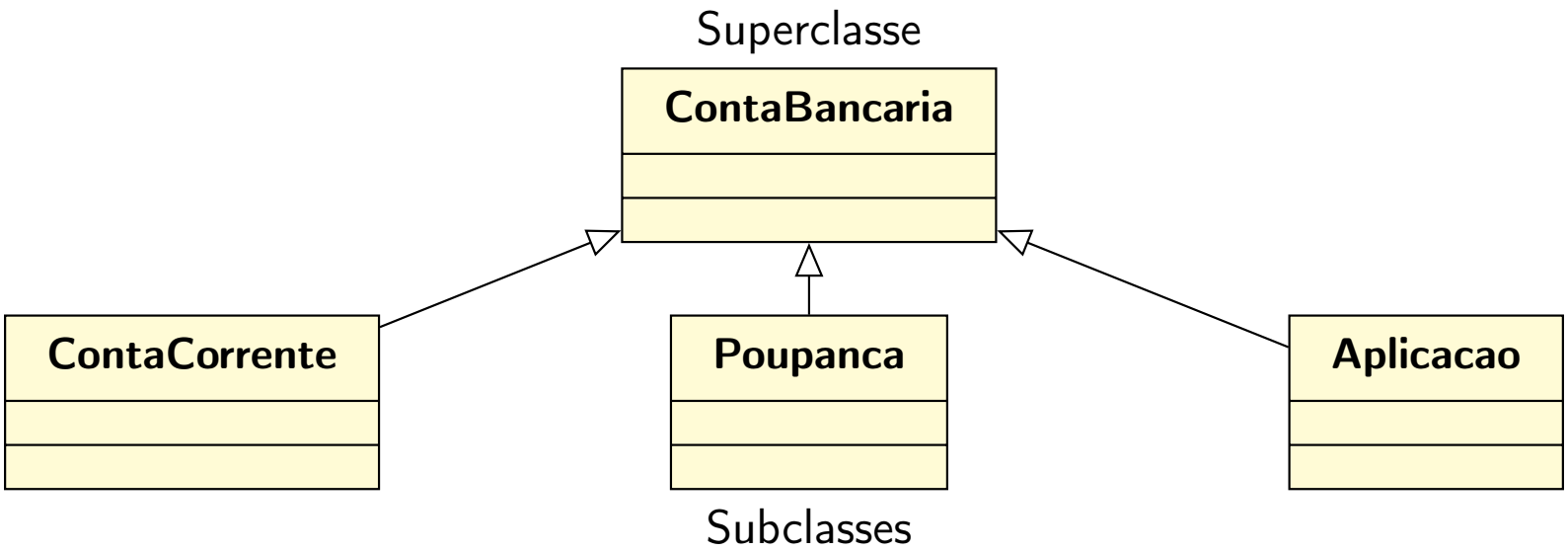
- Classe mãe, superclasse, classe base: A classe mais geral, a partir da qual outras classes herdam características (atributos e métodos);



# Mais sobre Herança

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 e membros

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

Acerca dos membros herdados pela subclasse:

# Herança e membros

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

Acerca dos membros herdados pela subclasse:

- Tratados de forma semelhante a qualquer outro membro da subclasse;

# Herança e membros

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

Acerca dos membros herdados pela subclasse:

- Tratados de forma semelhante a qualquer outro membro da subclasse;
- Nem todos os membros da superclasse são acessíveis pela subclasse (encapsulamento) – se membro da superclasse é encapsulado como `private`, subclasse não o acessa;

# Herança e membros

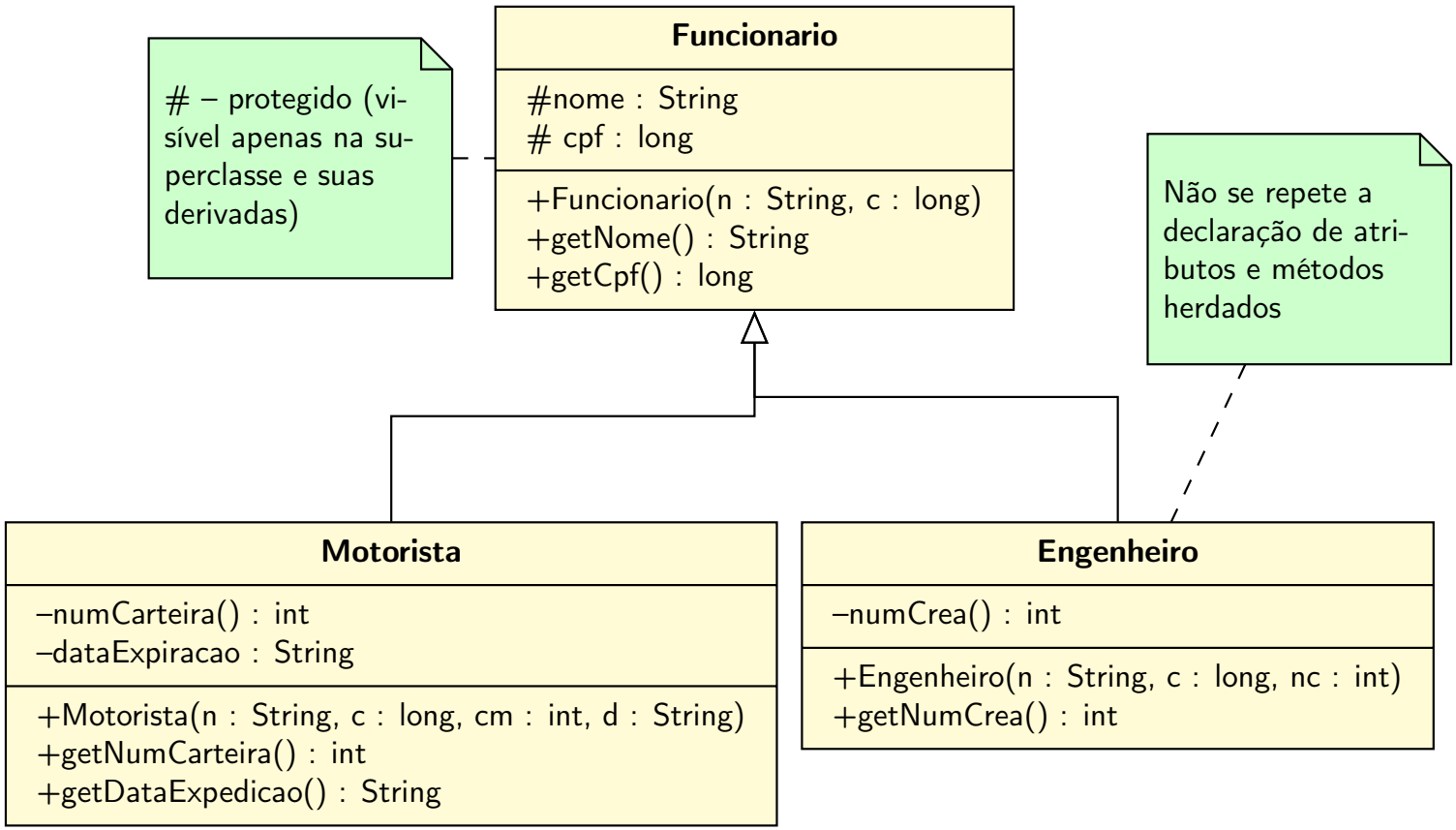
Uma subclasse herda atributos e métodos de sua superclasse, podendo possuir, no entanto, membros que lhe são próprios.

Acerca dos membros herdados pela subclasse:

- Tratados de forma semelhante a qualquer outro membro da subclasse;
- 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` (diagramas de classe UML: #) do Java.

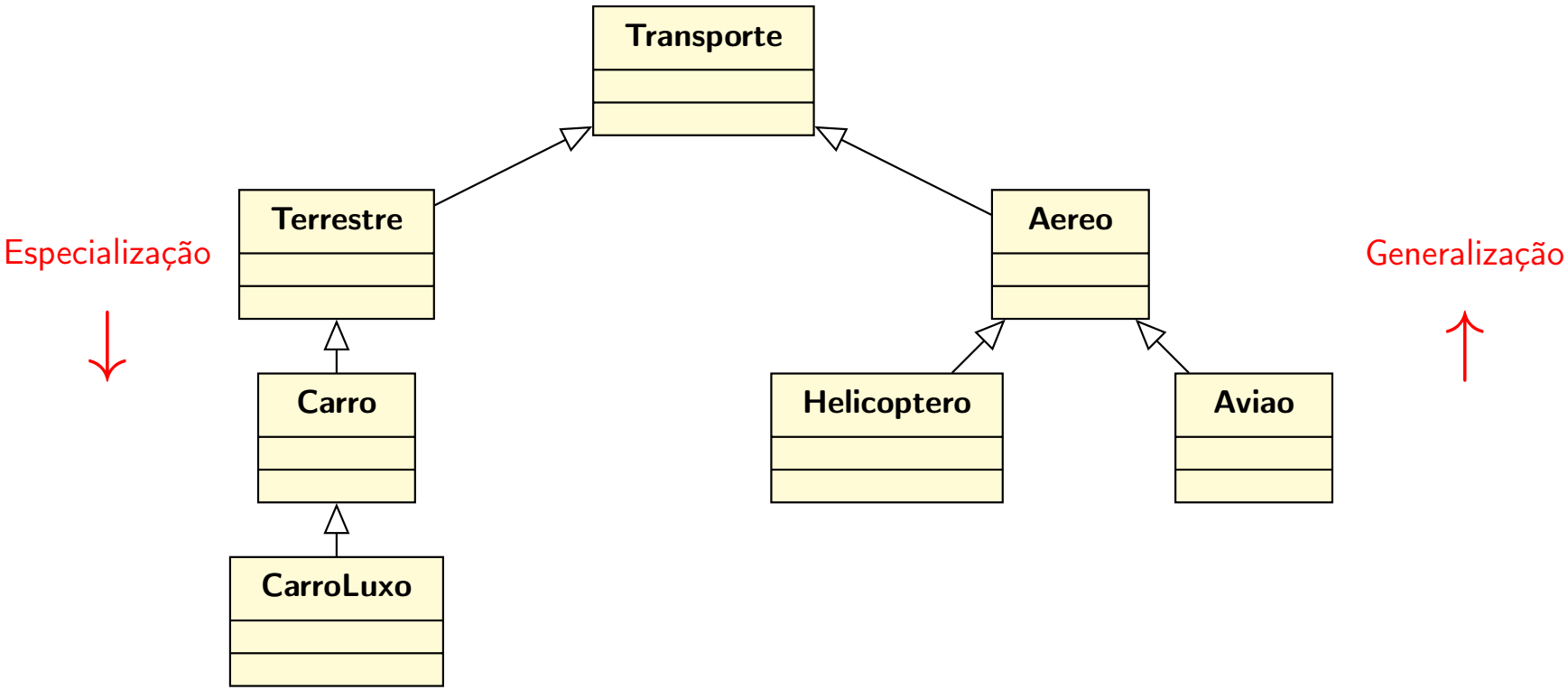
# Herança e membros

Exemplo:

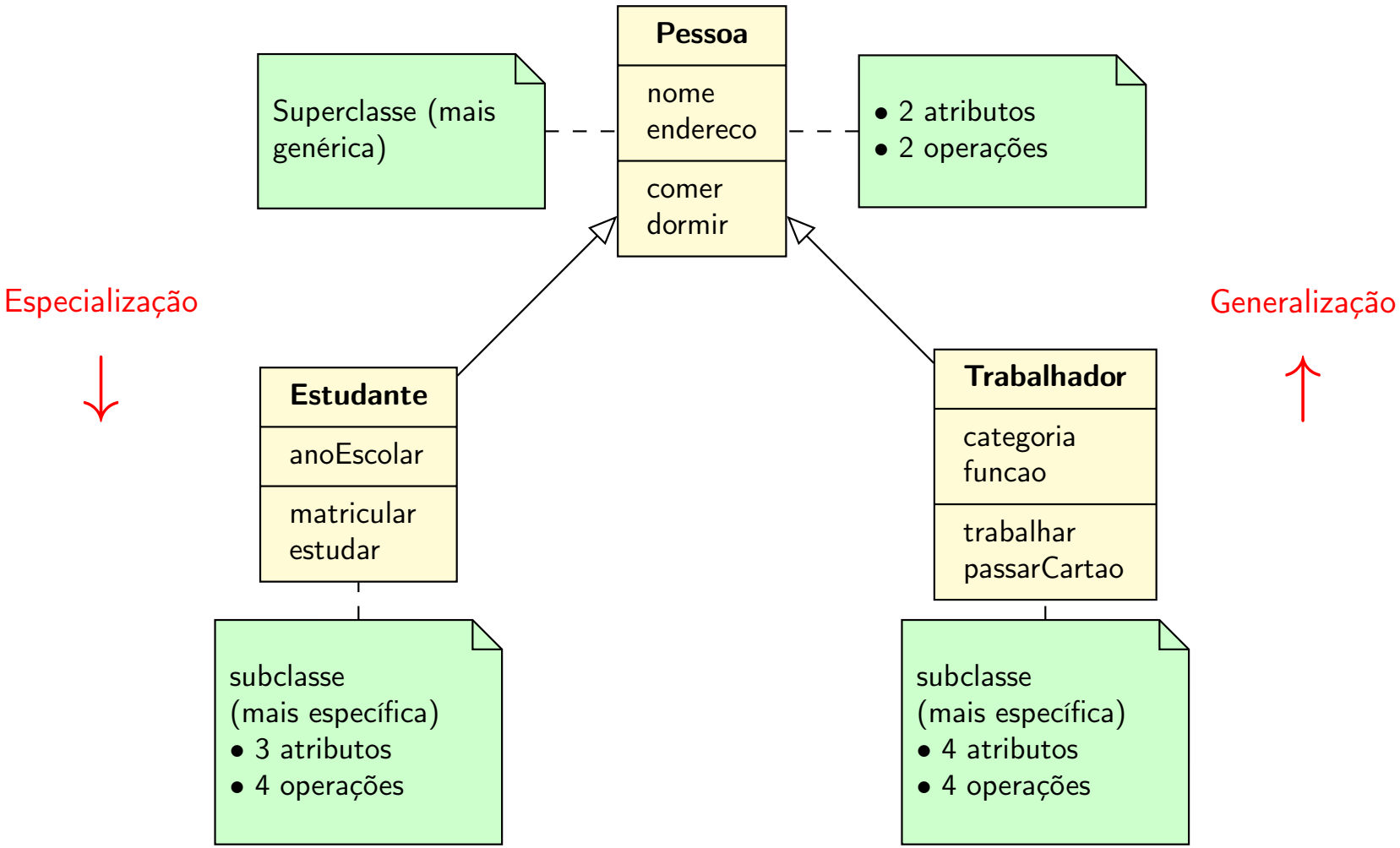


# Especialização vs. generalização I

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



# Herança e membros



# É herança?

Considere as três classes mencionadas abaixo e verifique se pode existir herança entre elas:

SIM: identifique a superclasse e as subclasses, e o que poderia diferenciá-las.

NÃO: explique o porquê.

1. Médico, Paciente, Pessoa

# É herança?

Considere as três classes mencionadas abaixo e verifique se pode existir herança entre elas:

SIM: identifique a superclasse e as subclasses, e o que poderia diferenciá-las.

NÃO: explique o porquê.

1. Médico, Paciente, Pessoa
2. Técnicos, Jogadores, Time

# É herança?

Considere as três classes mencionadas abaixo e verifique se pode existir herança entre elas:

SIM: identifique a superclasse e as subclasses, e o que poderia diferenciá-las.

NÃO: explique o porquê.

1. Médico, Paciente, Pessoa
2. Técnicos, Jogadores, Time
3. Cliente, ContaPessoaFísica, ContaPessoaJurídica

# É herança?

Considere as três classes mencionadas abaixo e verifique se pode existir herança entre elas:

SIM: identifique a superclasse e as subclasses, e o que poderia diferenciá-las.

NÃO: explique o porquê.

1. Médico, Paciente, Pessoa
2. Técnicos, Jogadores, Time
3. Cliente, ContaPessoaFísica, ContaPessoaJurídica
4. Perecível, NãoPerecível, Produto

# É herança?

Considere as três classes mencionadas abaixo e verifique se pode existir herança entre elas:

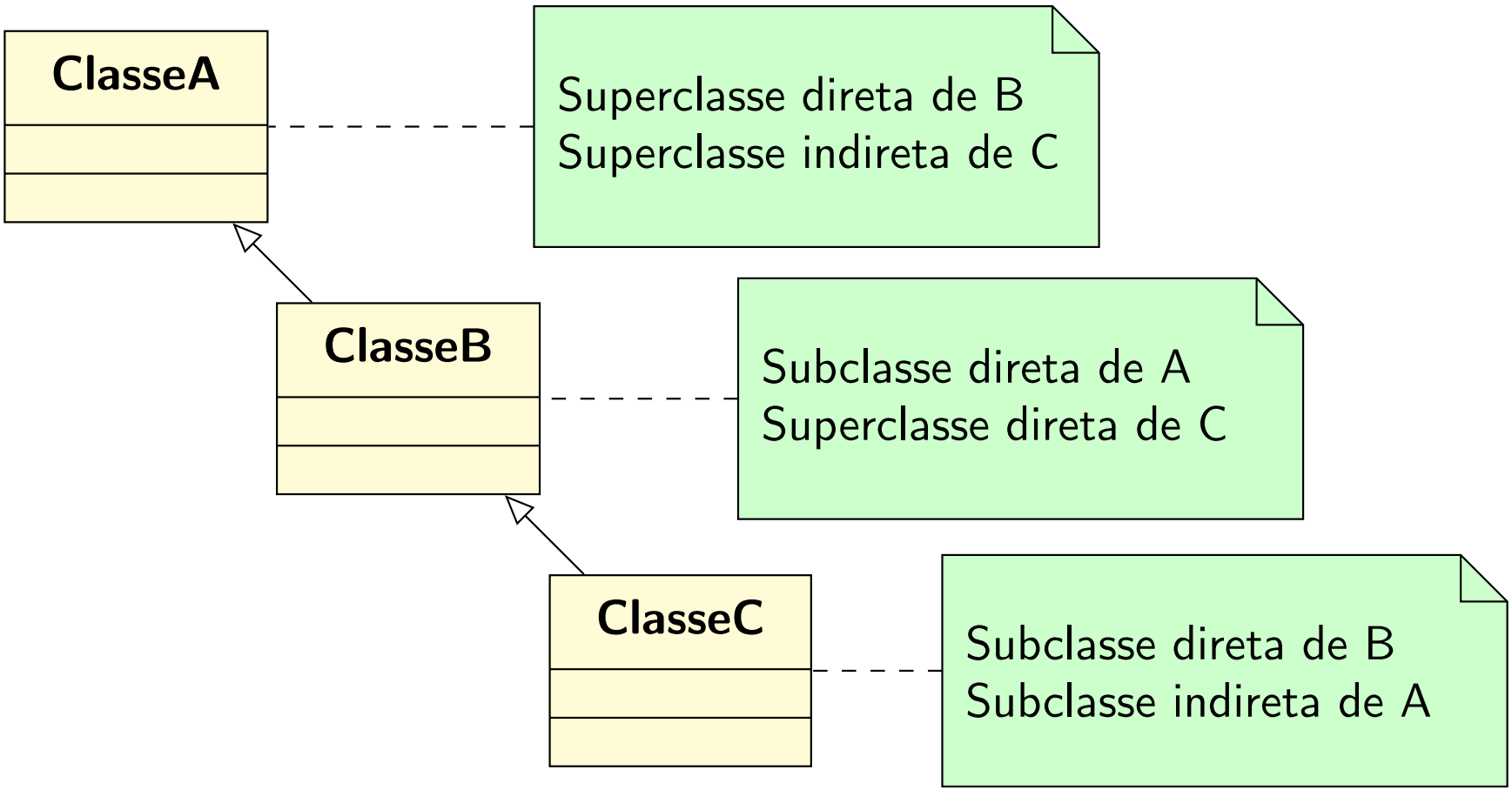
SIM: identifique a superclasse e as subclasses, e o que poderia diferenciá-las.

NÃO: explique o porquê.

1. Médico, Paciente, Pessoa
2. Técnicos, Jogadores, Time
3. Cliente, ContaPessoaFísica, ContaPessoaJurídica
4. Perecível, NãoPerecível, Produto
5. Professor, Universidade, Aluno

# Hierarquia de classes

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



# Diferenciação

- Uma subclasse pode diferenciar-se de sua classe mãe:

# Diferenciação

- Uma subclasse pode diferenciar-se de sua classe mãe:
  - Com *operações adicionais*, diferentes das herdadas;

# Diferenciação

- Uma subclasse pode diferenciar-se de sua classe mãe:
  - Com *operações adicionais*, diferentes das herdadas;
  - *Acrescentando* atributos adicionais;

# Diferenciação

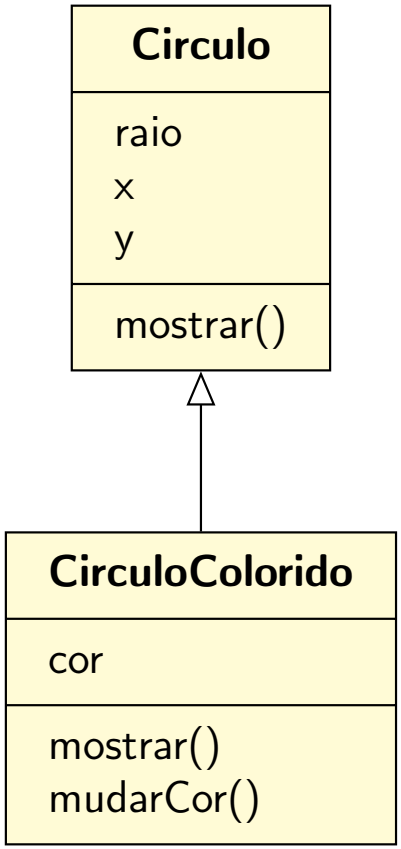
- Uma subclasse pode diferenciar-se de sua classe mãe:
  - Com *operações adicionais*, diferentes das herdadas;
  - *Acrescentando* atributos adicionais;
  - *Sobrepondo* comportamentos existentes, oferecidos pela superclasse, contudo inapropriados para a nova classe (overriding)

# Diferenciação

Exemplo:

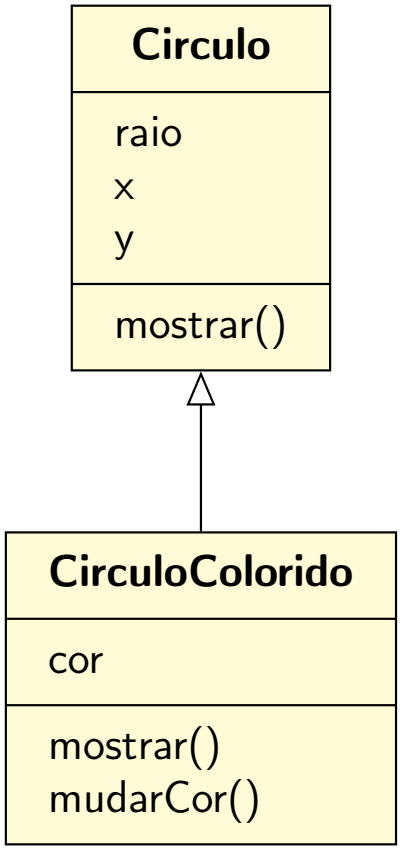
# Diferenciação

Exemplo:



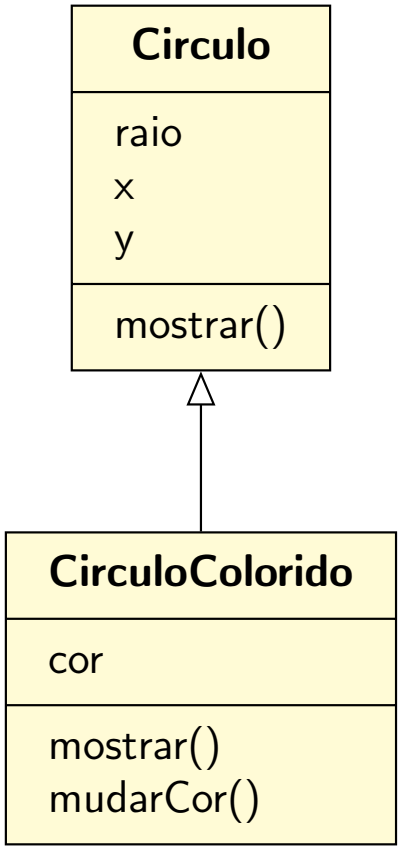
# Diferenciação

Exemplo:



# Diferenciação

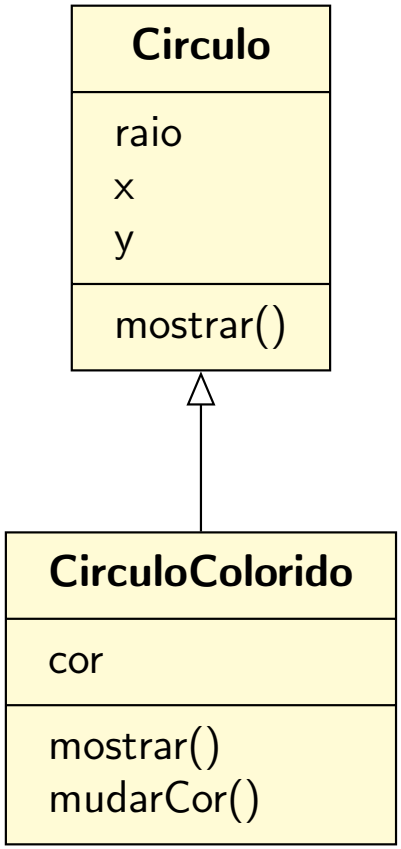
Exemplo:




- A classe `CirculoColorido` herda da classe `Circulo` os atributos `raio`, `x` e `y`;

# Diferenciação

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*)

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.

# Sobrescrita de métodos (*overriding*)

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*)

## 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.

# Sobrescrita de métodos (*overriding*)

## 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.

- Outra observação importante:

# Sobrescrita de métodos (*overriding*)

## 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.

- Outra observação importante:
  - O modificador de acesso do método da subclasse pode relaxar o acesso, mas não o contrário

# Sobrescrita de métodos (*overriding*)

## 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.

- Outra observação importante:
  - 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, mas não `private`.

# Sobrescrita de métodos (*overriding*)

## 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.

- Outra observação importante:
  - 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, mas não `private`.

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

# Vantagens

- Reutilização do código;

# Vantagens

- Reutilização do código;
- Modificação de uma classe sem mudanças na classe original;

# 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;

# 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

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

# Um exemplo prático

- Os professores de uma universidade dividem-se em 2 categorias:
  - Professores em dedicação exclusiva (DE)

# Um exemplo prático

- Os professores de uma universidade dividem-se em 2 categorias:
  - Professores em dedicação exclusiva (DE)
  - Professores horistas

# Um exemplo prático

- Os professores de uma universidade dividem-se em 2 categorias:
  - Professores em dedicação exclusiva (DE)
  - Professores horistas
- Considerações:

# Um exemplo prático

- 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;

# Um exemplo prático

- 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;

# Um exemplo prático

- 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

Modelagem:

| **ProfDE** |
| --- |
| –nome : String<br>–matricula : String<br>–cargaHoraria : int<br>–salario : float |
| +ProfDE(n : String, m : String, i : int, s : float)<br>+getNome() : String<br>+getMatricula() : String<br>+getCargaHoraria() : int<br>+getSalario() : float |

| **ProfHorista** |
| --- |
| –nome : String<br>–matricula : String<br>–cargaHoraria : int<br>–salarioHora : float |
| +ProfHorista(n : String, m : String, t : int, s : float)<br>+getNome() : String<br>+getMatricula() : String<br>+getCargaHoraria() : int<br>+getSalario() : float |

# Um exemplo prático

Análise:

- As classes têm alguns atributos e métodos iguais;

# Um exemplo prático

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`?

# Um exemplo prático

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.

# Um exemplo prático

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

Sendo assim:

- Cria-se uma classe `Professor`, que contém os *membros* – atributos e métodos – comuns aos dois tipos de professor;

# Um exemplo prático

Sendo assim:

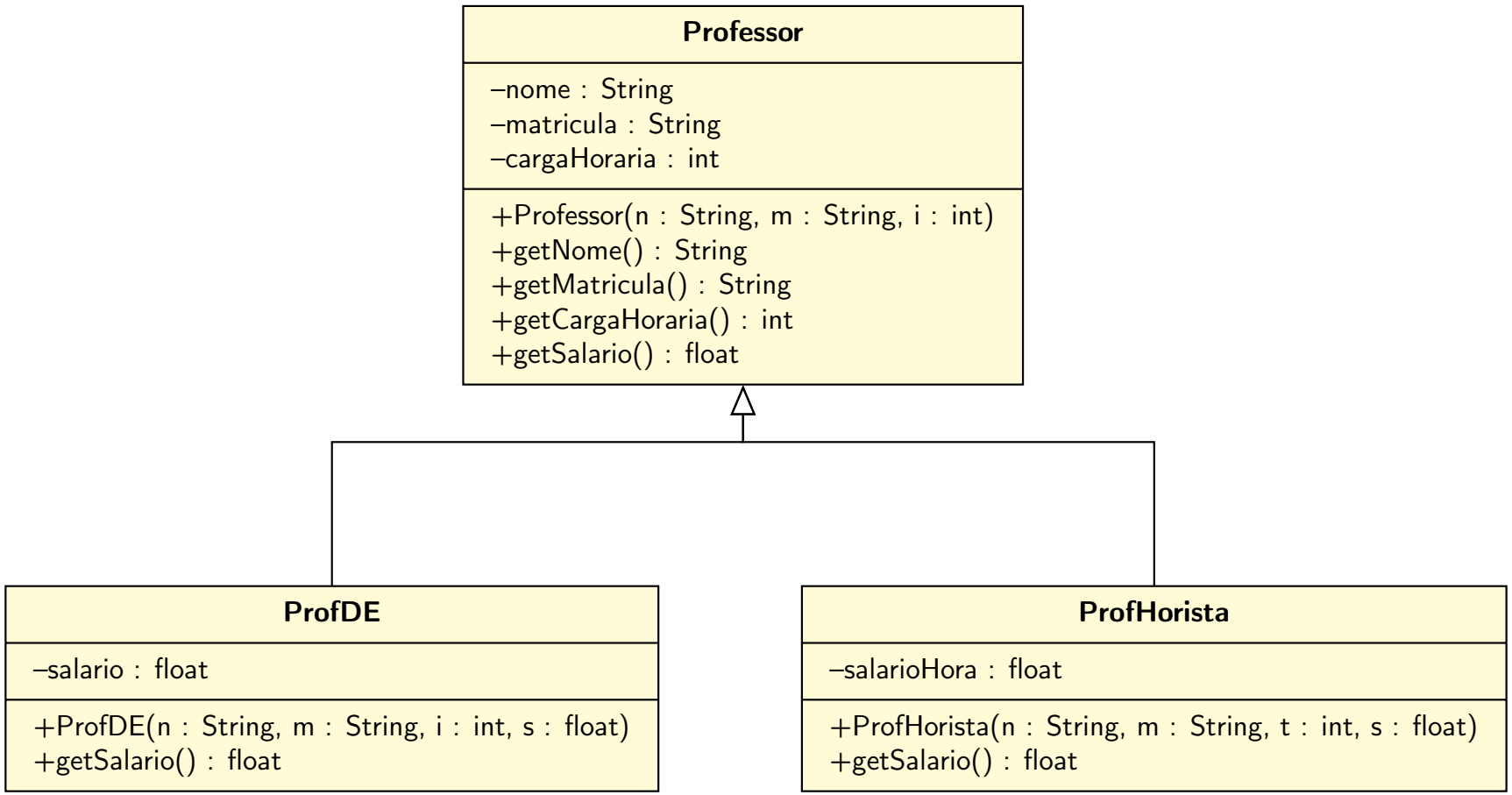
- Cria-se uma classe `Professor`, que contém os *membros* – atributos e métodos – comuns aos dois tipos de professor;
- A partir dela, cria-se duas novas classes, que representarão os professores horistas e DE;

# Um exemplo prático

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



# super e this

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

`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

- `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

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.

# super e this

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()`.

# super e this

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 podem ser invocados de dentro de construtores de subclasses, e na primeira linha de código dos mesmos.

# super e this

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 podem 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

# super e this

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 podem 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

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 podem chamar diretamente o método como se fosse de si próprias.
  ```
  nomeDoMetodo( [argumento(s)] );
  ```

# 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