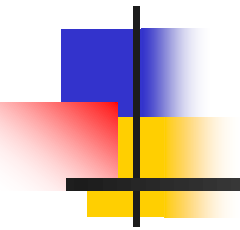
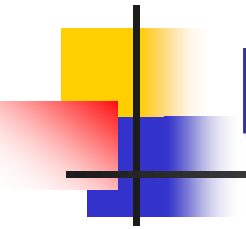


Herança

Alberto Costa Neto
DComp - UFS

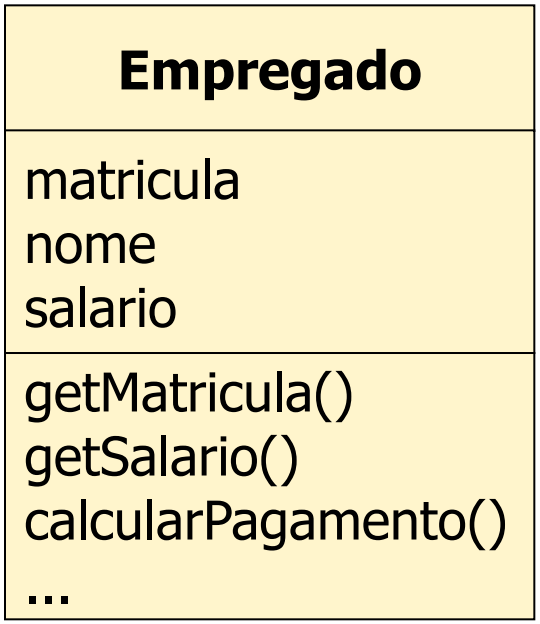


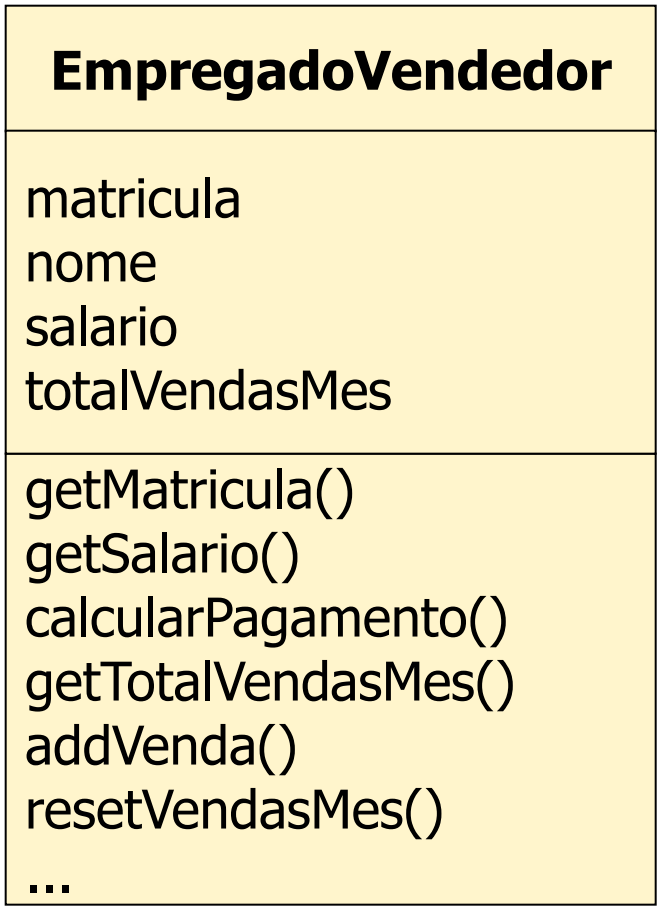
Motivação

- Vimos como se faz encapsulamento e a importância de fazê-lo...
- Contudo, também é possível fazer encapsulamento em algumas linguagens não OO
- O que mais uma linguagem OO tem a nos oferecer?

Motivação

- Vamos ver o exemplo:





Visualizando o código das classes

```
public class Empregado{

   private int matricula;
   private String nome;
   private float salario;

      ...

   public int getMatricula () {
        return this.matricula;
   }

   public void setMatricula (int mat) {
         this.matricula = mat;
   }

   public String getNome () {
        return this.nome;
   }

   ...
```

```
public class EmpregadoVendedor{

   private int matricula;
   private String nome;
   private float salario;
   private float totalVendasMes;

      ...

   public int getMatricula () {
        return this.matricula;
   }

   public void setMatricula (int mat) {
         this.matricula = mat;
   }

   public String getNome () {
        return this.nome;
   }

   ...
```

Visualizando o código das classes

```
// continuando Empregado

   public void setNome (String nm) {
         this.nome = nm;
   }

   public float getSalario () {
        return this.salario;
   }

   public void setSalario (float sal) {
         this.salario = sal;
   }

   public float calcularPagamento () {
        return this.getSalario();
   }
}
```

```
// continuando EmpregadoVendedor

   public void setNome (String nm) {
         this.nome = nm;
   }

   public float getSalario () {
        return this.salario;
   }

   public void setSalario (float sal) {
        this.salario = sal;
   }

   public float calcularPagamento () {
       float valor = this.getSalario() +
             (this.getTotalVendasMes() *
               0.10f)
       return valor;
   }
```

...

Visualizando o código das classes

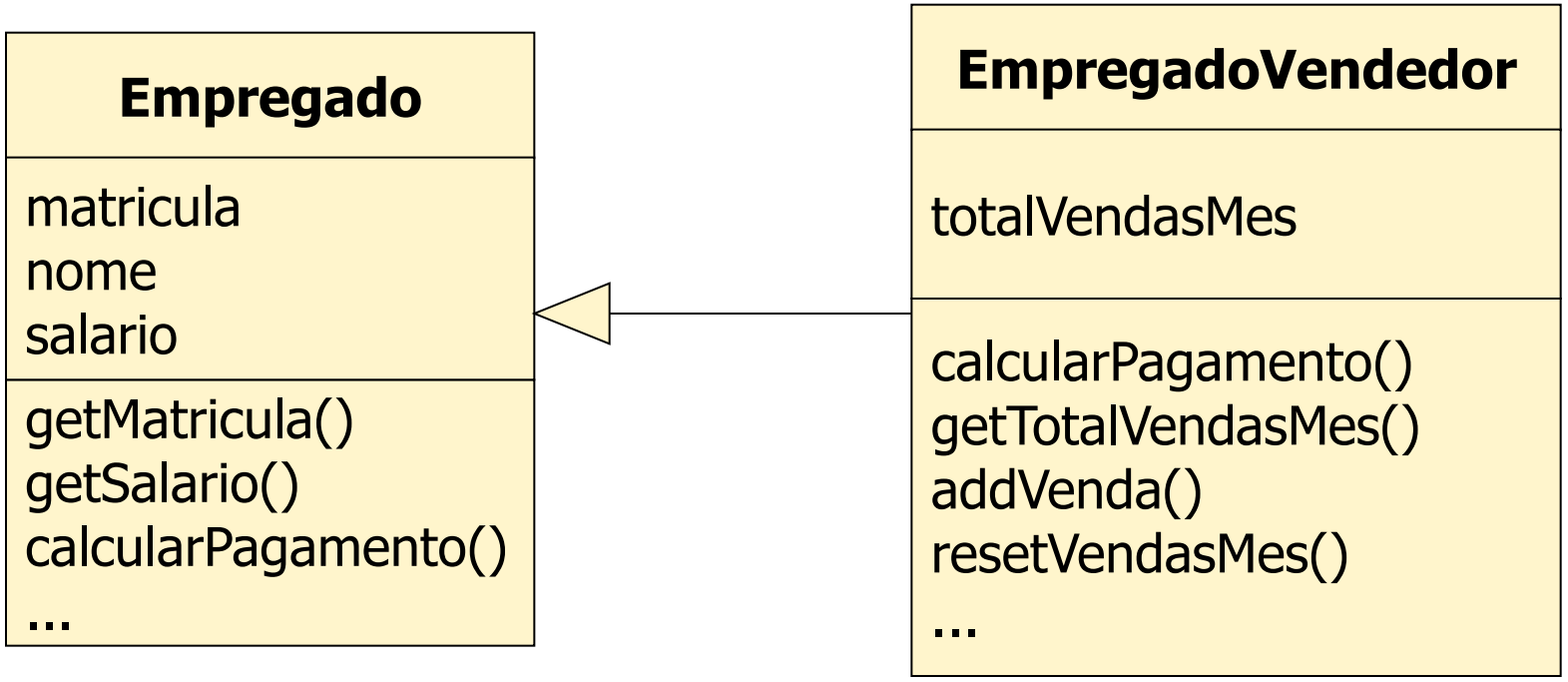
```
// continuando EmpregadoVendedor

   public float getTotalVendasMes() {
        return this.totalVendasMes;
   }

  public void addVenda (float venda) {
        this.totalVendasMes += venda;
   }

   public void resetVendas () {
        this.totalVendasMes = 0;
   }

}
```

Motivação

- Existe código redundante!!
- Qual problema surge?
- Será que EmpregadoVendedor não **é um tipo** especial de Empregado???
- O que acontece se outro tipo de empregado for necessário???

Motivação

- Na POO, podemos fazer o seguinte...



A herança permite que a definição de uma classe seja baseada em outra classe já existente

Motivação

- Como fica o código?
  - Classe Empregado ➔ inalterado
  - Classe EmpregadoVendedor ➔ resumido
    - Herda atributos e comportamento
    - Método calcularPagamento será redefinido
    - Atributos e Métodos específicos serão acrescentados

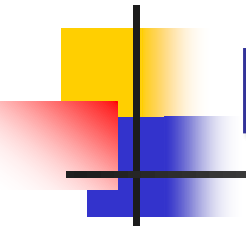
Visualizando o código de EmpregadoVendedor

```
public class EmpregadoVendedor extends Empregado {

    private float totalVendasMes;

     ...

    public float calcularPagamento () {
        float valor = this.getSalario() + (this.getTotalVendasMes() * 0.10f);
        return valor;
    }

    public float getTotalVendasMes() {
        return this.totalVendasMes;
    }

        public void addVenda (float venda) {
        this.totalVendasMes += venda;
    }

    public void resetVendas () {
        this.totalVendasMes = 0;
    }
}
```

- Como ficam os construtores?

```
// em  Empregado

    public Empregado (int mat, String nome, float salario) {
            this.matricula = mat;
            this.nome = nome;
            this.salario = salario;
    }
```

```
// em EmpregadoVendedor

    public EmpregadoVendedor (int mat, String nome, float salario) {
            super(mat, nome, salario);
            this.totalVendasMes = 0;
    }
```

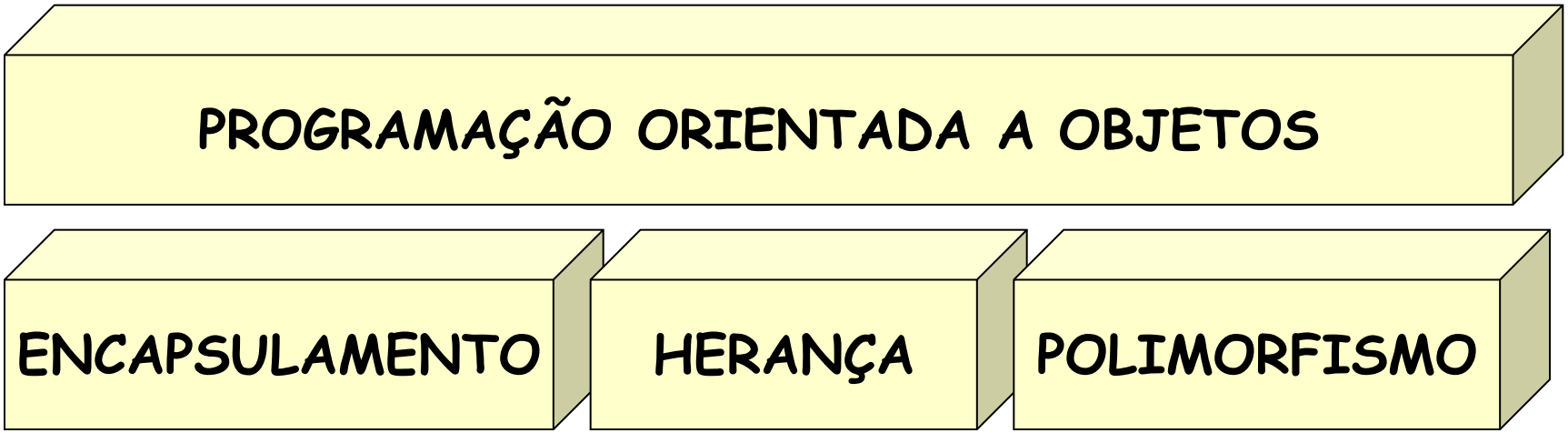


Roteiro

- Motivação
- Pilares da Programação Orientada a Objetos
- Prática
- Quando usar Herança?
- Quando não usar Herança?
- Conceitos
- Mecanismo de Funcionamento
- Usar Herança para...
- Dicas para Herança eficaz
- Por que usar Herança?

Pilares da POO

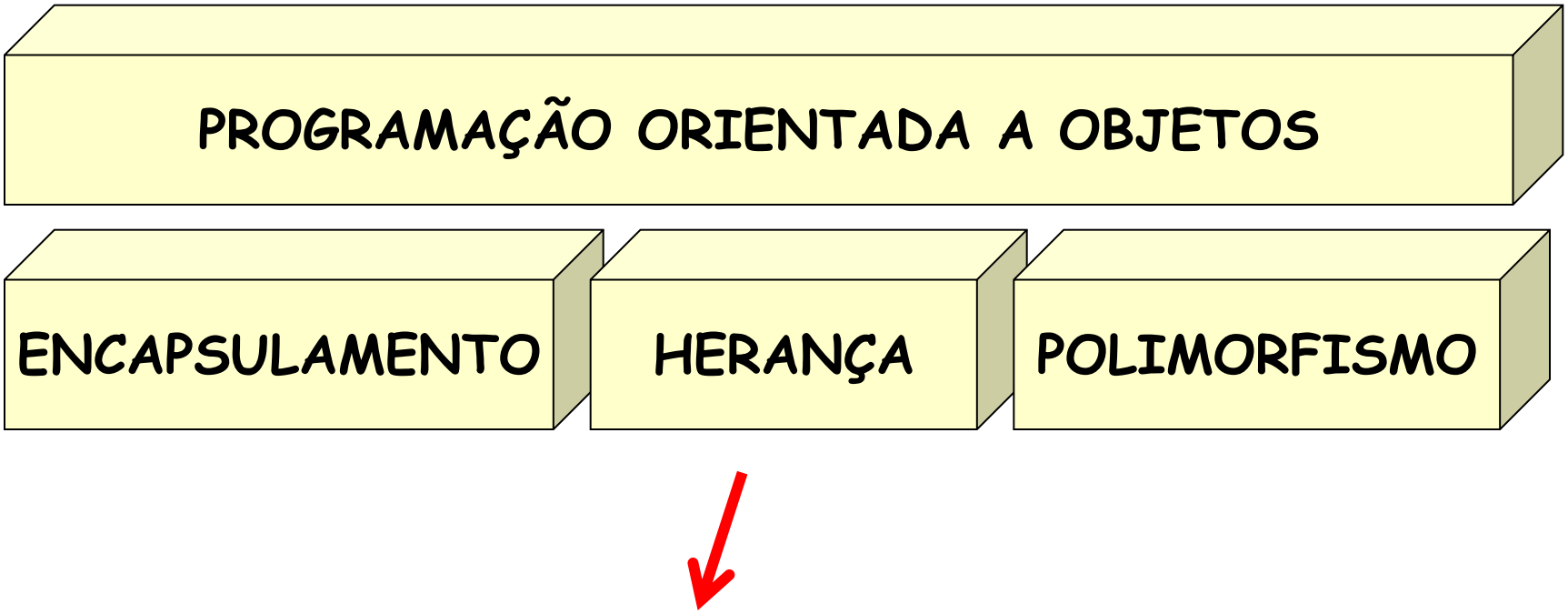
- O que vamos estudar hoje...



"Esconda seu jogo!"

Pilares da POO

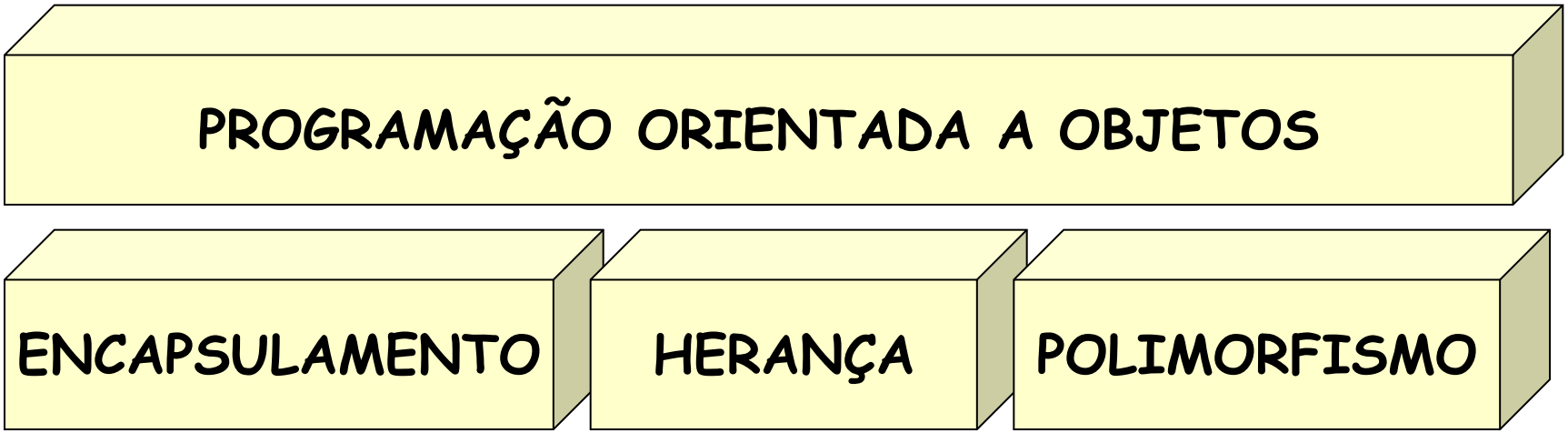
- Lembrando...



"Filho de peixe, peixinho é!"

Pilares da POO

- Lembrando...



"Sou camaleão!"

Pilares da POO

- O que vamos estudar hoje...

Dever de Sala

- Criar as classes Empregado e EmpregadoVendedor a partir do código contido nos slides.

Implementar também a classe Exemplo1Hera

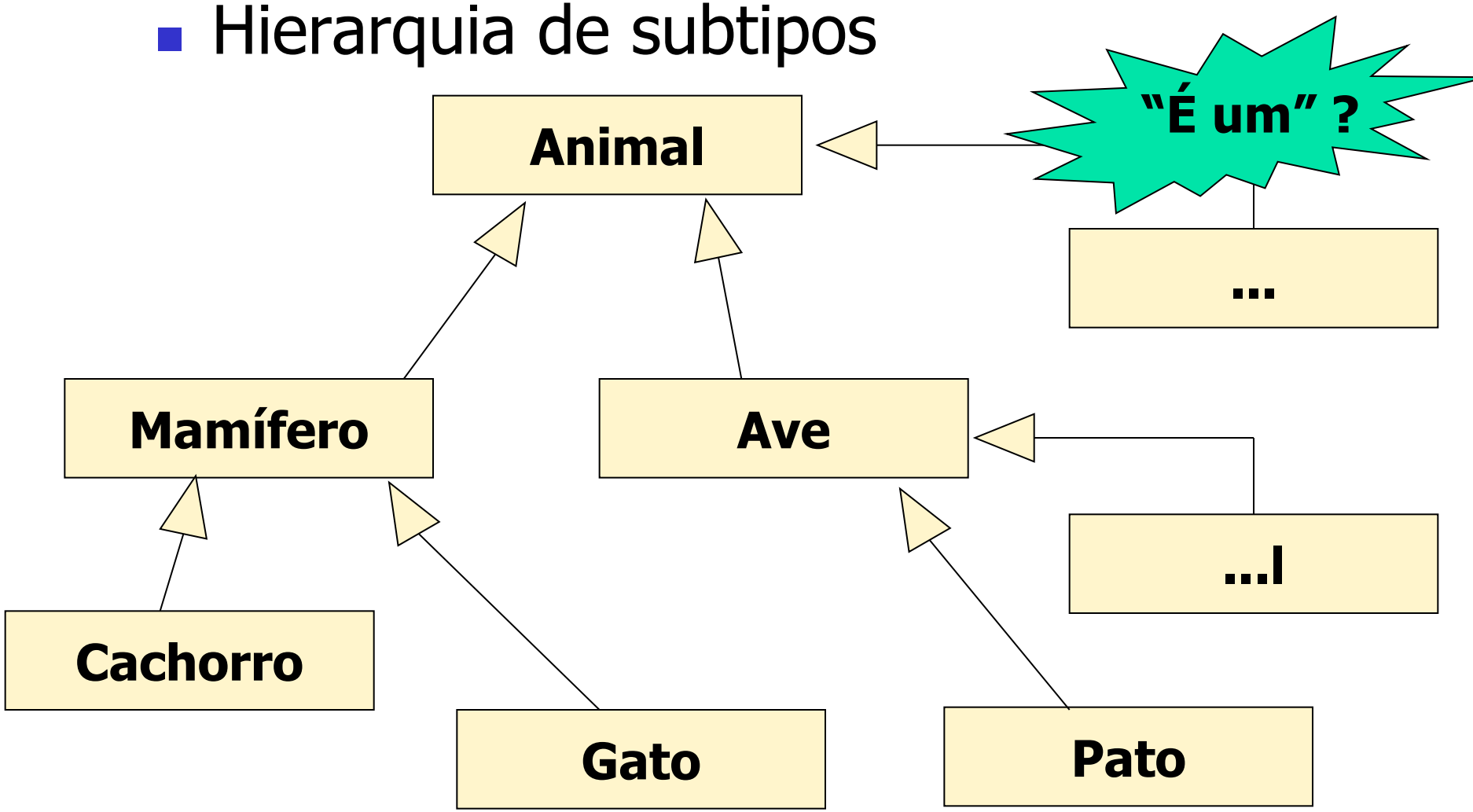
```
public class Exemplo1Hera {

    public static void main(String[] args) {
            EmpregadoVendedor e =
                new EmpregadoVendedor(12, "João", 1000.00f);

        e.addVenda(2000.0f);
        e.addVenda(3500.0f);

        System.out.println( "Empregado:" + e.getNome() );
        System.out.println( "Salario: " + e.getSalario() );
        System.out.println( "Vendas no mes: " +
                                        e.getTotalVendasMes());
        System.out.println( "Valor Recebido: " +
                                        e.calcularPagamento());
    }
}
```

Quando usar herança?

Quando uma classe é do mesmo tipo de outra.

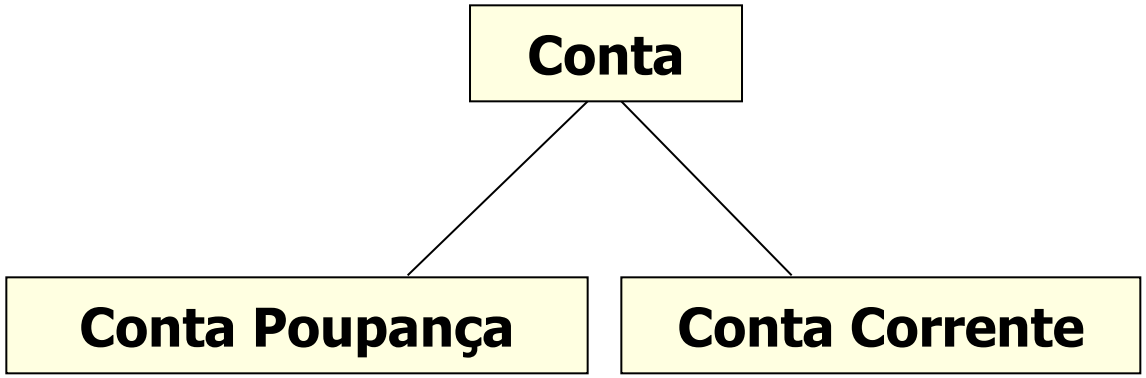
Quando usar herança?

- Hierarquia de subtipos



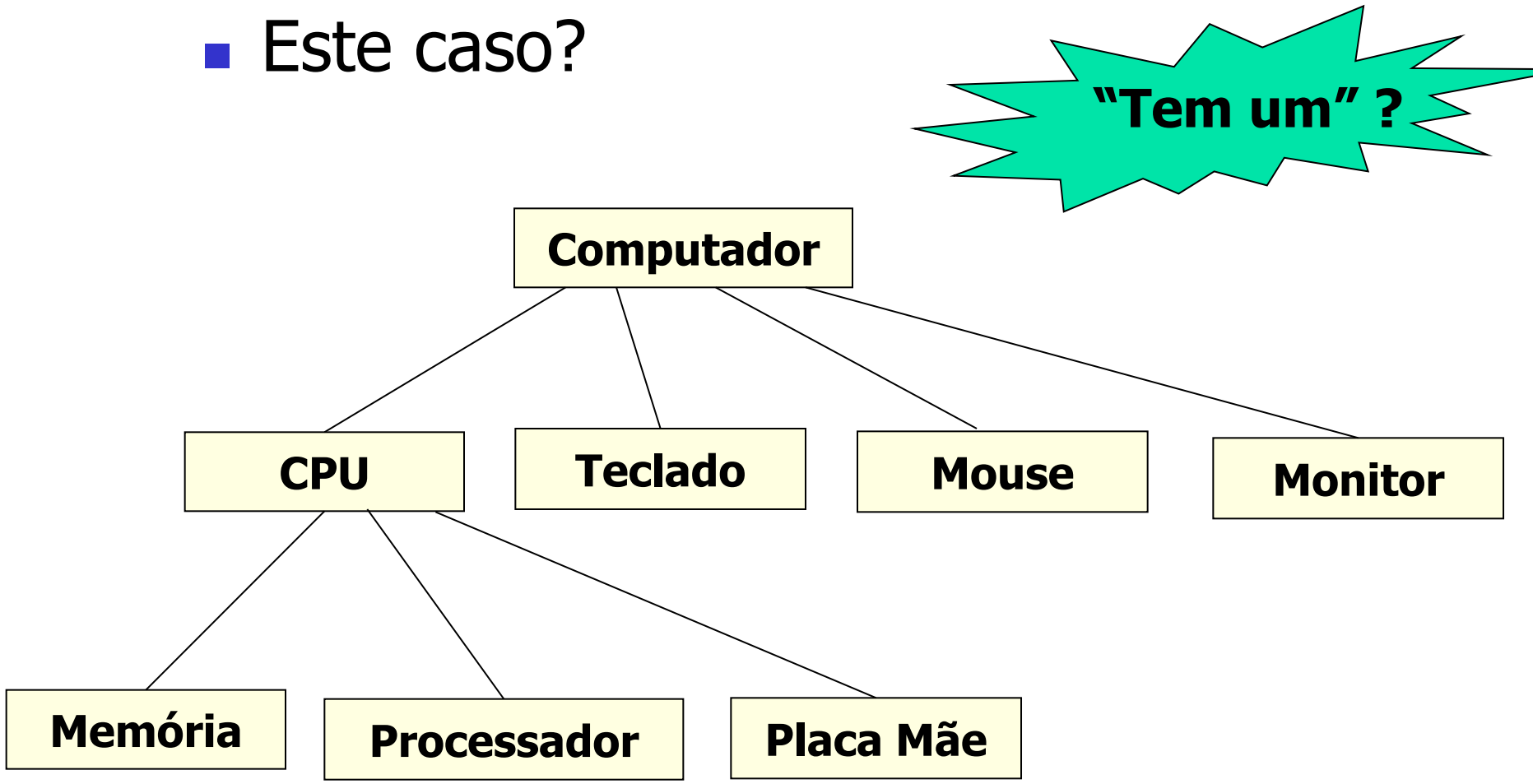
Quando usar herança?

- Este caso?



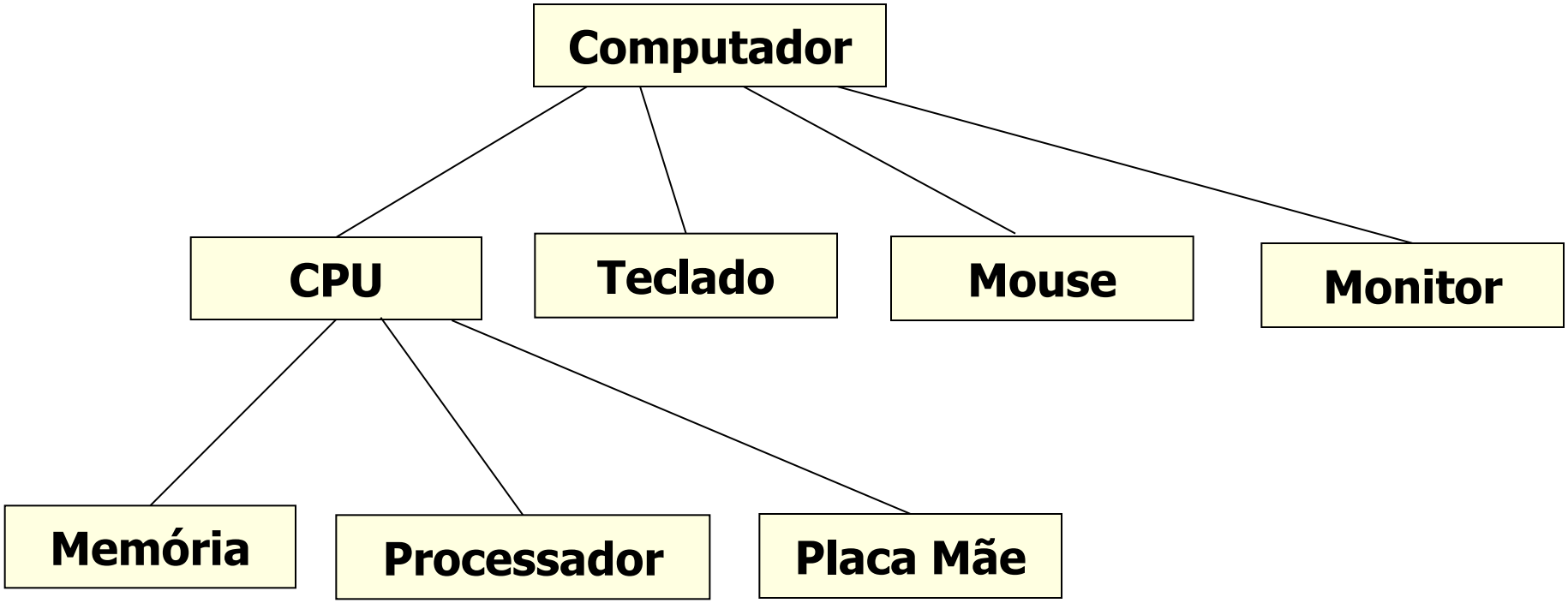
Quando usar herança?

- Este caso?



Quando não usar herança?

- Hierarquia de Agregação
  - Relacionamento do tipo "Tem um"



Dever de sala

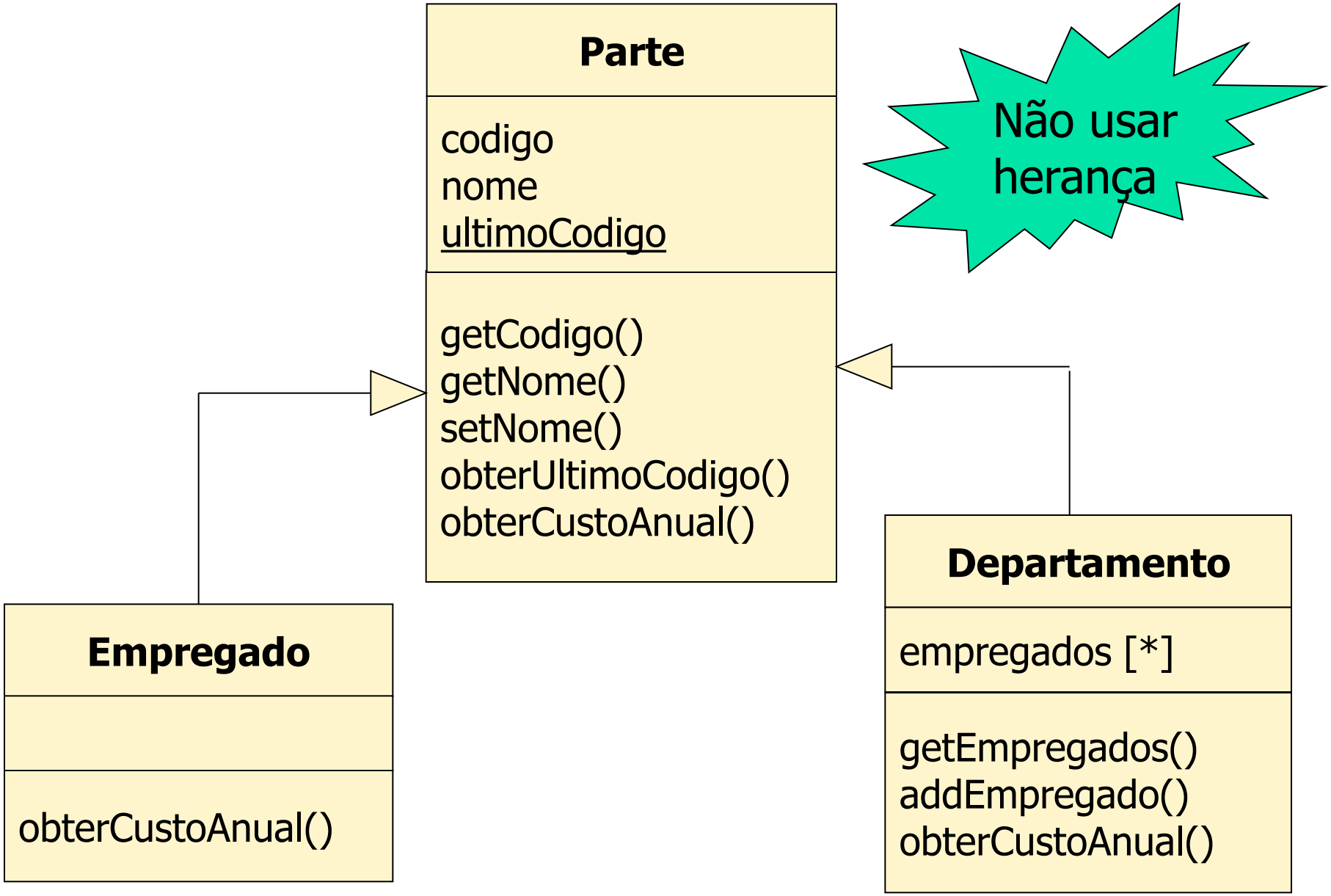
- Identificar exemplos de:
  - Hierarquia de Subtipos
  - Hierarquia de Agregação

Quando não usar herança?

- Herança para Reutilização de Implementação
  - Quando usa a herança para poder utilizar a implementação de outra classe
  - Herança pobre

Evitar esse tipo de herança!!

- Herança para Reutilização de Implementação



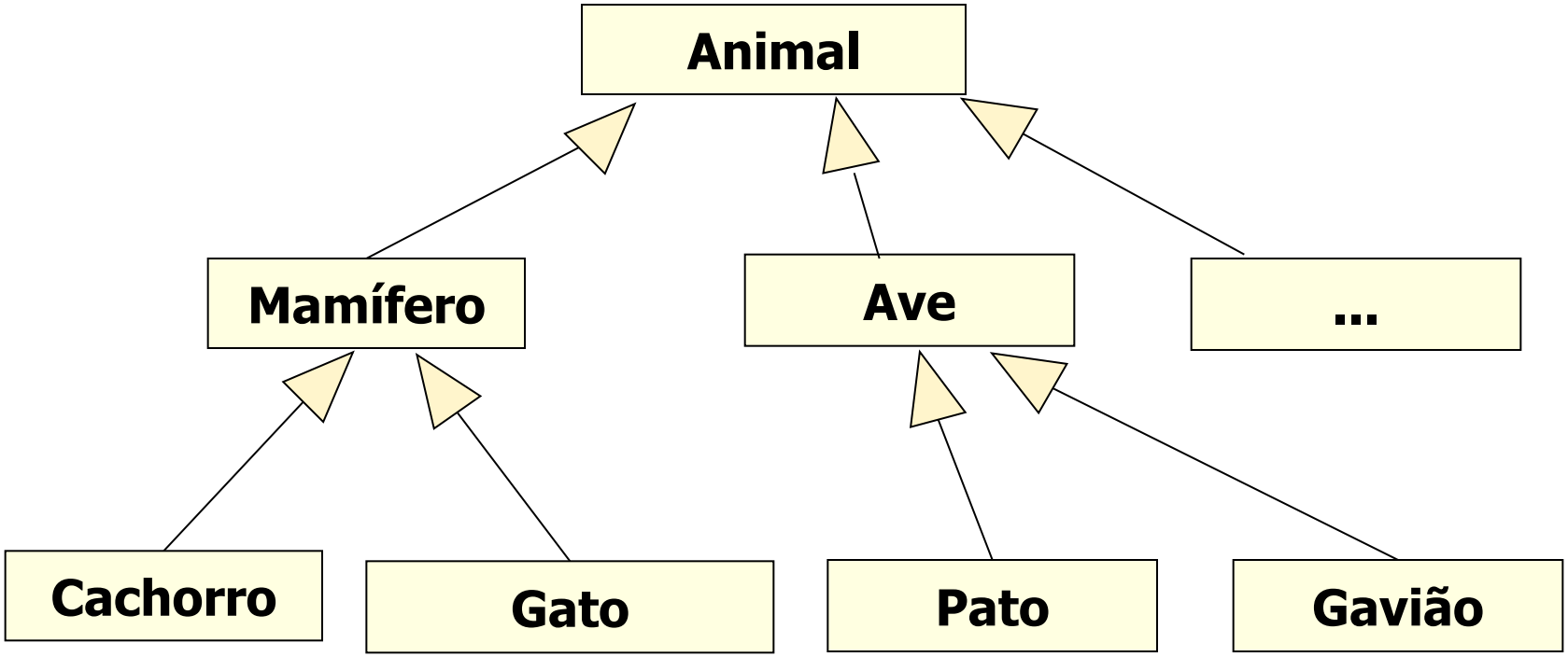
Conceitos

- Segundo Anthony Sintes:

"Herança é o mecanismo que permite estabelecer relacionamentos 'é um' entre classes".

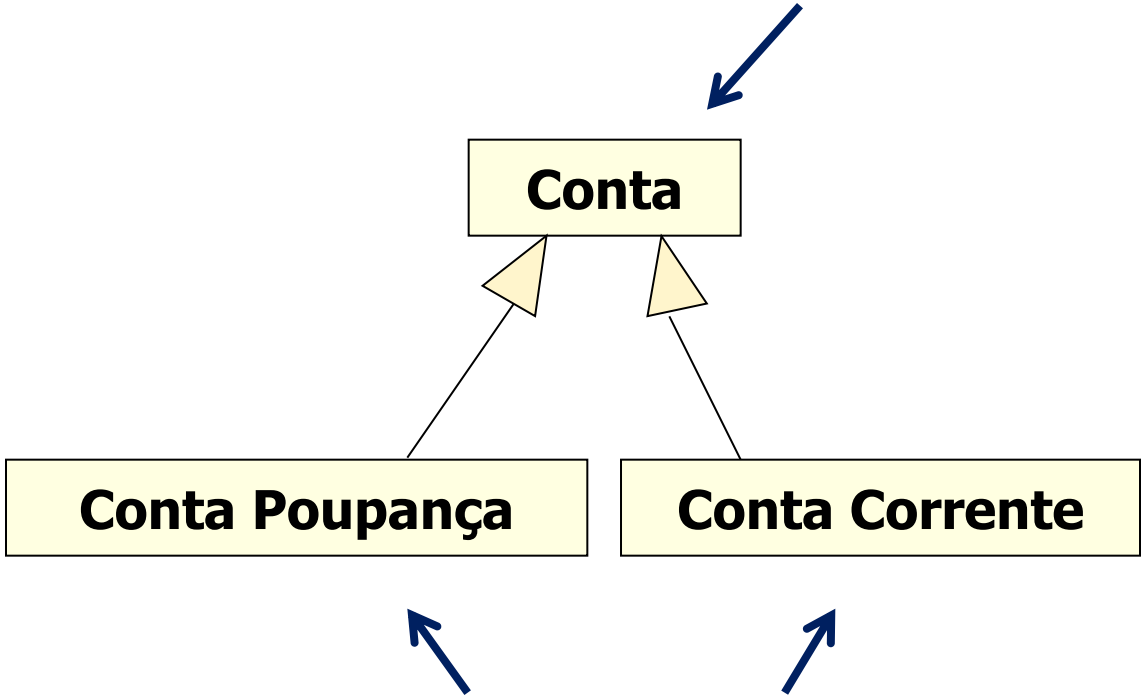
Conceitos

- Hierarquia de Herança: é um mapeamento do tipo árvore de relacionamentos que se formam entre classes como resultado da herança



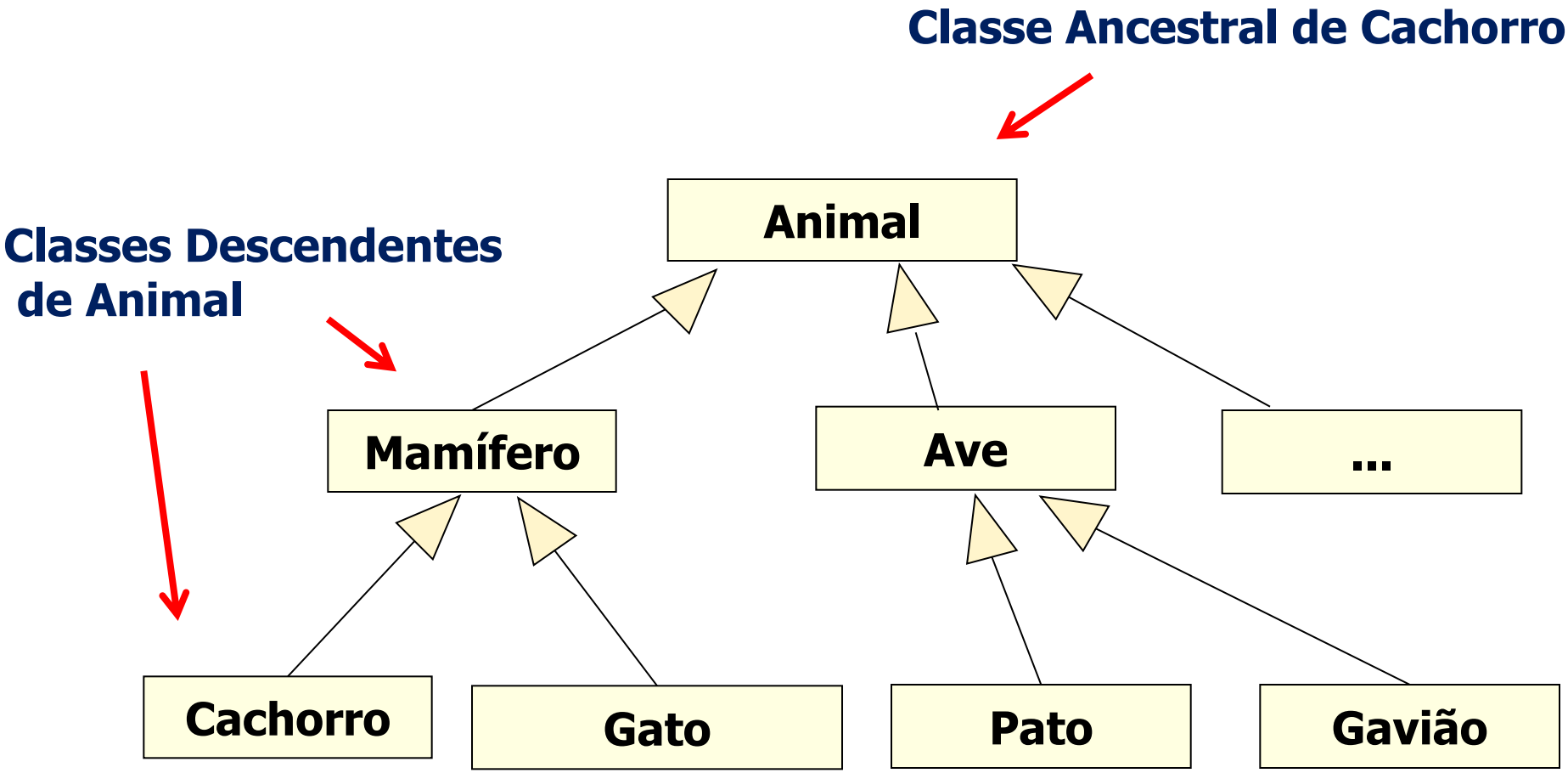
Conceitos

Classe Mãe ou **Progenitora** ou **Superclasse** ou **Classe Base**

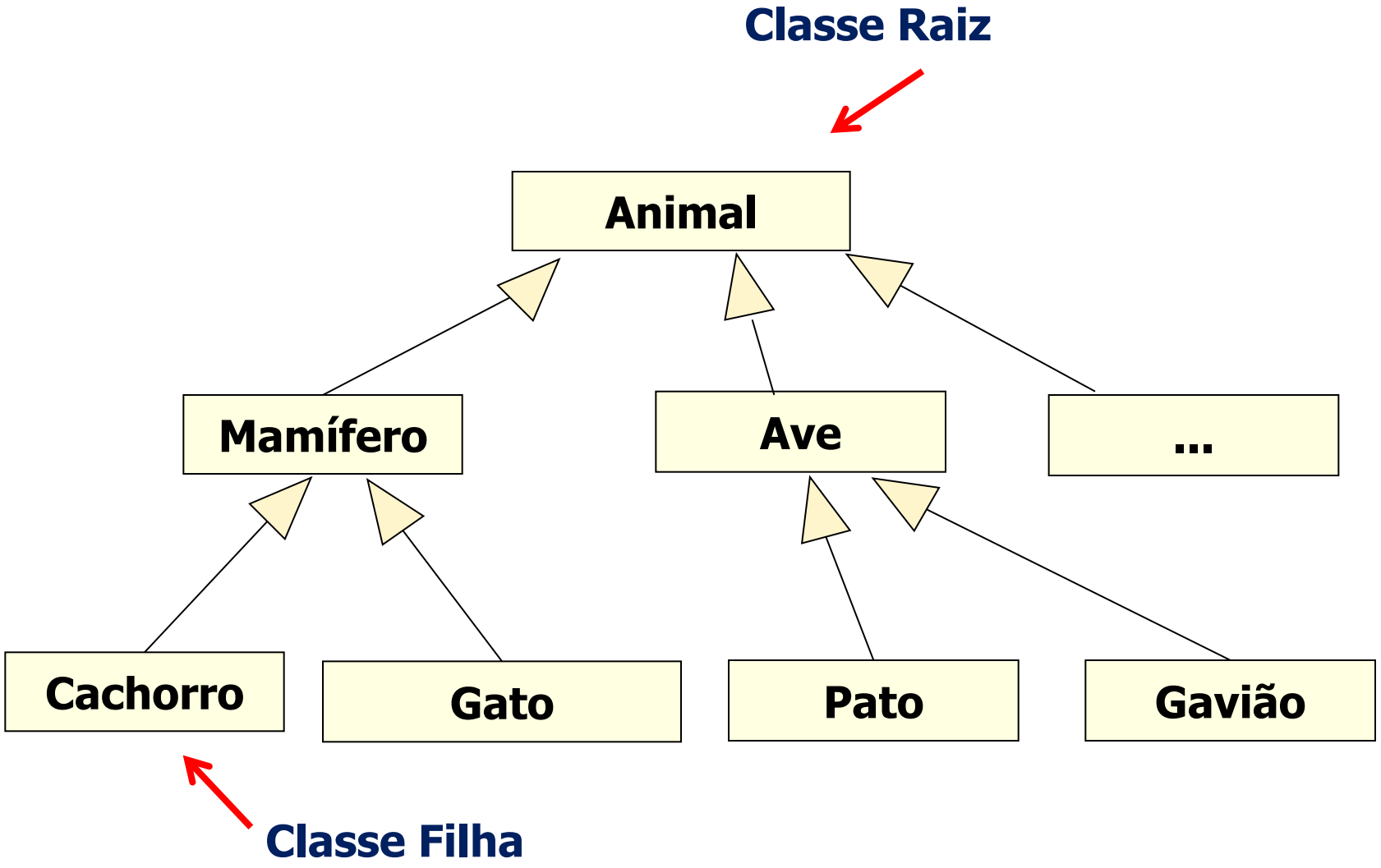


Classe Filha ou **Subclasse** ou **Classe Derivada**

Conceitos


Classe Ancestral de Cachorro
Classes Descendentes de Animal
Animal
Mamífero
Ave
...
Cachorro
Gato
Pato
Gavião

Conceitos

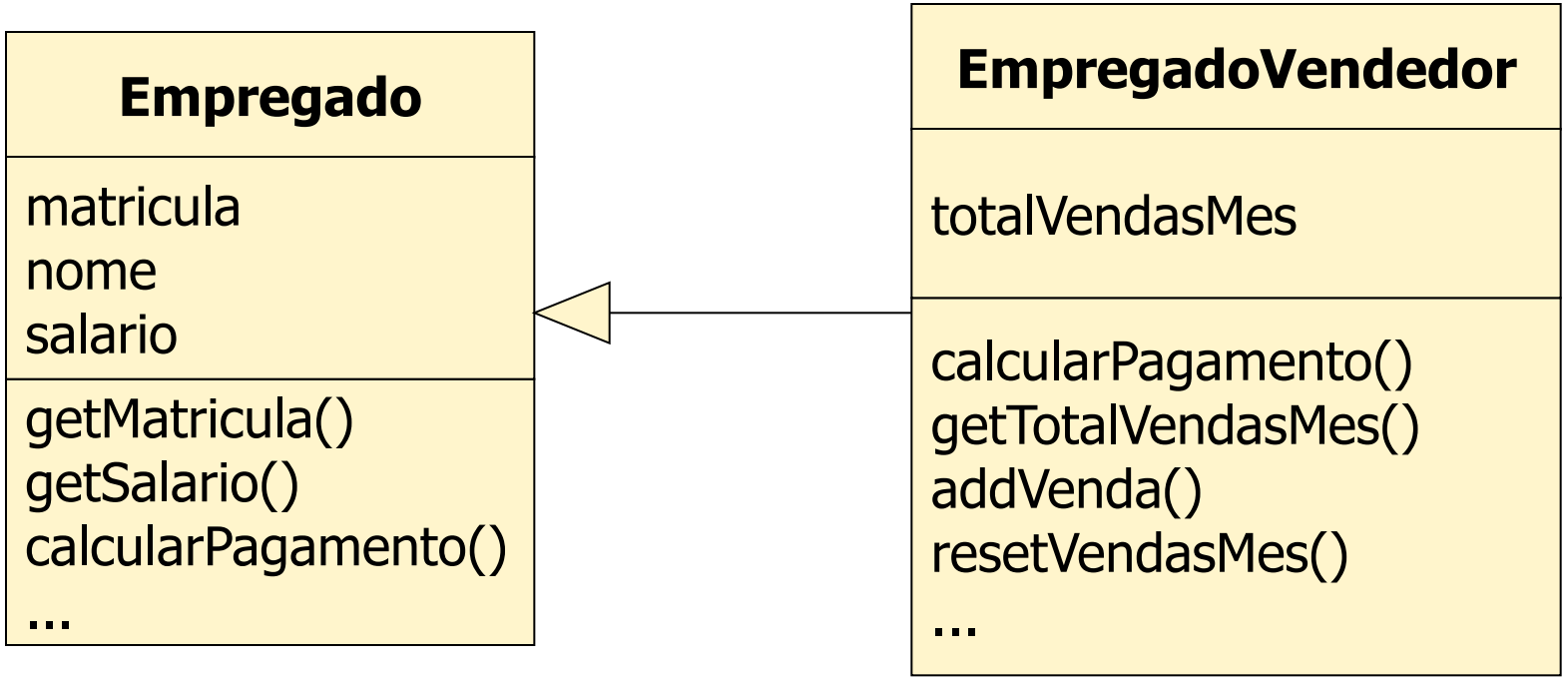

Classe Raiz
Animal
Mamífero
Ave
...
Cachorro
Gato
Pato
Gavião
Classe Filha

Mecanismo da Herança

- Quando uma classe herda de outra, ela herda implementação, comportamento e atributos
- Uma classe filha não poder remover nenhum dos elementos herdados
- Entretanto, métodos e atributos podem ser:
  - Sobrepostos (redefinidos)
  - Novos
  - Recursivo

Mecanismo da Herança

- Voltando ao exemplo inicial...
  - O que foi sobreposto, recursivo ou novo?



Mecanismo da Herança

- Como fica a questão da visibilidade com relação à Herança?

Mecanismo da Herança

- Poderíamos fazer a seguinte mudança?

```
public class EmpregadoVendedor extends Empregado {

    private float totalVendasMes;

     ...

    public float calcularPagamento () {
        float valor = this.salario + (this.getTotalVendasMes() * 0.10f);
        return valor;
    }

    ...
}
```

Mecanismo da Herança

- Se alterássemos a visibilidade do atributo salário??

```
public class Empregado{

   private int matricula;
   private String nome;
   protected float salario;

      ...
```

```
public class EmpregadoVendedor extends Empregado {

      public float calcularPagamento () {
       float valor = this.salario + (this.getTotalVendasMes() * 0.10f);
       return valor;
    }

    ...
}
```

Mecanismo da Herança

- A seguinte modificação poderia ser realizada??

```
public class Exemplo1Hera {

    public static void main(String[] args) {
            EmpregadoVendedor e =
                    new EmpregadoVendedor(12, "João", 1000.00f);

        e.addVenda(2000.0f);
        e.addVenda(3500.0f);

        System.out.println( "Empregado:" + e.getNome() );
        System.out.println( "Salario: " + e.salario );
        System.out.println( "Vendas no mes: " +
                                        e.getTotalVendasMes());
        System.out.println( "Valor Recebido: " +
                                        e.calcularPagamento());
    }
}
```

Mecanismo da Herança

- O acesso da classe filha aos elementos herdados da classe mãe depende do nível de acesso definido
- Atributos não-constantes e qualquer método destinados unicamente à própria classe devem ser definidos como privados
- Usar métodos protegidos somente para métodos que alguma subclasse precisará utilizar
  - Somente subclasses terão acesso
- Usar atributos protegidos quando for necessário acesso direto por parte das subclasses

Mecanismo da Herança

- Com a sobreposição de métodos...

```
public class Exemplo1Hera {

    public static void main(String[] args) {
            EmpregadoVendedor e =
                    new EmpregadoVendedor(12, "João", 1000.00f);

        e.addVenda(2000.0f);
        e.addVenda(3500.0f);

        System.out.println( "Empregado:" + e.getNome() );
        System.out.println( "Salario: " + e.getSalario() );
        System.out.println( "Vendas no mes: " +
                                        e.getTotalVendasMes());
        System.out.println( "Valor Recebido: " +
                                        e.calcularPagamento());
    }
}
```

Como o objeto sabe qual definição utilizar?

Mecanismo da Herança

- Quando for necessário um método sobreposto chamar a versão da classe Mãe ou de algum outro ancestral?
  - Em Java ➔ super.<nome-do-membro>

```
public float calcularPagamento () {
        float valor = super.calcularPagamento() +
                      (this.getTotalVendasMes() * 0.10f);
        return valor;
    }
```

Mecanismo da Herança

- Usamos o mecanismo do super(), nos construtores

```
// em  Empregado

    public Empregado (int mat, String nome, float salario) {
            this.matricula = mat;
            this.nome = nome;
            this.salario = salario;
    }
```

```
// em EmpregadoVendedor

    public EmpregadoVendedor (int mat, String nome, float salario) {
            super(mat, nome, salario);
            this.totalVendasMes = 0;
    }
```

Mecanismo da Herança

- Construtores não são herdados
- Se nenhum construtor for definido na superclasse, as duas classes utilizarão o construtor padrão (sem parâmetros)
- Se um construtor for definido na superclasse, a subclasse só poderá usar o padrão se este também for definido na superclasse
- Para definir um construtor na subclasse, na superclasse pelo menos o construtor padrão deve ser definido

Usar Herança para...

- Codificar apenas as diferenças
- Substituição de tipo

Usar Herança para...

- Codificar apenas as diferenças

Permite herdar o comportamento de uma classe adicionando apenas o código que for específico.

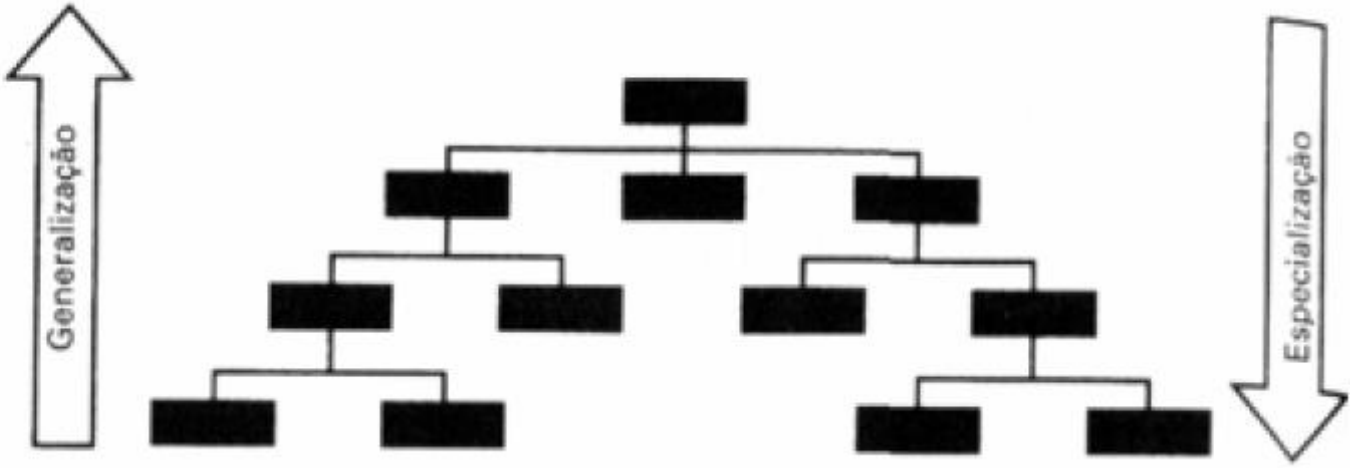
Processo de Especialização

Mãe = comportamento generalizado ➔
Filha = comportamento especializado

Usar Herança para...

- Hierarquia de Generalização e Especialização

Quando percorre uma hierarquia para cima, você generaliza. Quando percorre uma hierarquia para baixo, você especializa.



Usar Herança para...

- Quais as vantagens?

Usar Herança para...

- Substituição de tipo

Princípio de Substituição

Em qualquer código onde uma instância da classe Mãe é esperada, uma instância da classe filha pode ser passada

Usar Herança para...

- Exemplo: Empregado / EmpregadoVendedor

```
public class FolhaDePagamento {

    public float calculaSalarioLiquido(Empregado e) {
                return e.getSalario() * 0.95f;
    }

}
```

- Exemplo: Empregado / EmpregadoVendedor

```
public class Exemplo2Hera {

        public static void main(String[] args) {
            EmpregadoVendedor ev = new EmpregadoVendedor (67, "Pedro",
                                                          1000.00f);

            Empregado e = new   Empregado (32, "Marcelo", 1000.00f);

           FolhaDePagamento fp = new FolhaDePagamento();

            float salarioEV = fp.calculaSalarioLiquido( ev );
            float salarioE  = fp.calculaSalarioLiquido( e );

           System.out.println (ev.getNome() + " " +  salarioEV );
           System.out.println (e.getNome() + " " +  salarioE );

        }
}
```

Usar Herança para...

- Por que pode haver substituição de tipo?

- Qual a vantagem?

Modificador Final

- Permite definir elementos "constantes"
  - não poderão ser "modificados"

Classe ➔ não pode ser estendida

Método ➔ não pode ser sobreposto

Atributo ➔ conteúdo constante

Modificador Final

- Classe ➔ não pode ser estendida
  - Classe Folha = não pode ter descendentes

```
public final class ClasseX {
    // Implementação da classe
}
```

```
//Erro! Classe final não pode ser estendida
public class ClasseY extends ClasseX {
    ...
}
```

Modificador Final

- Método → não pode ser sobreposto

```
public final void metodoFinal() {

    // Implementação do método que não
    // poderá ser sobrescrita nas subclasses

}
```

Modificador Final

- Atributo ➔ conteúdo constante
  - Após inicializado, não poderá ser alterado

```
final float PI = 3.14f;
```

Modificador Final

Classe não pode ser herdada

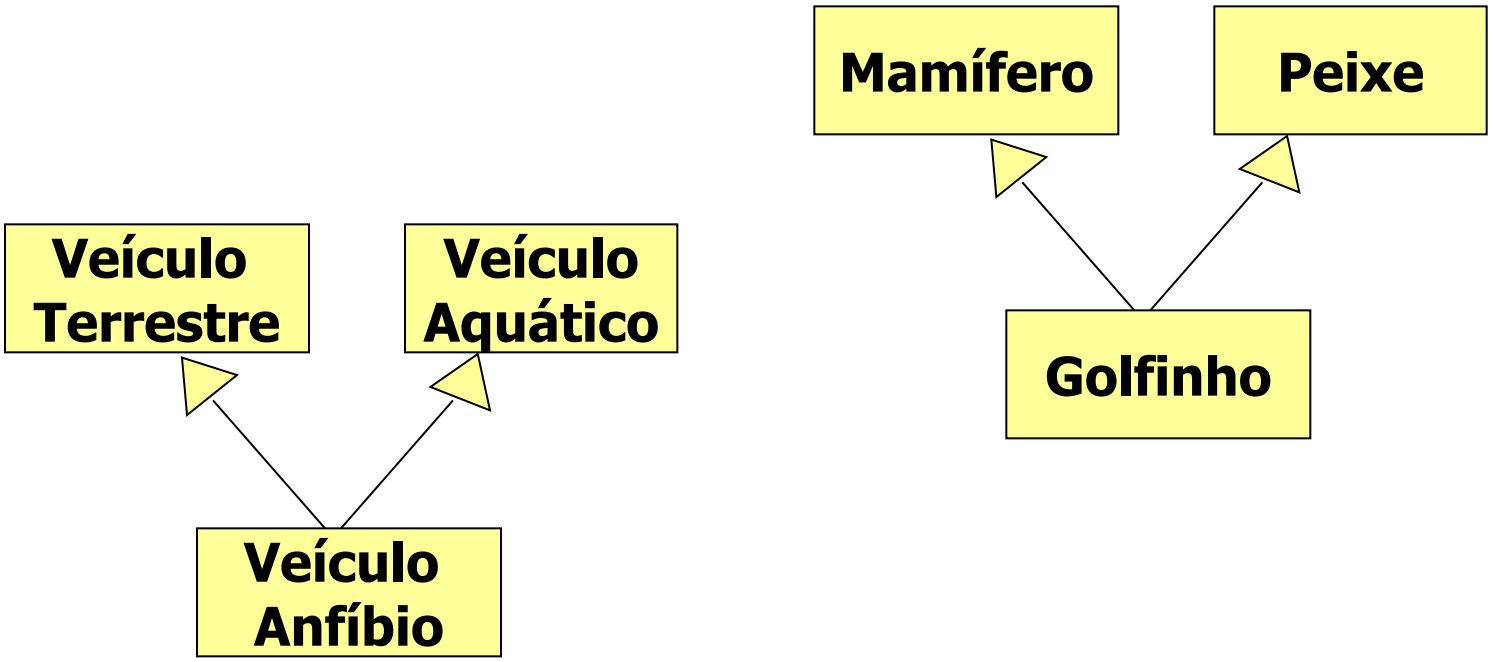
Atributo constante

Método não pode ser redefinido

```
public final class CalculaArea {

        static final  float PI = 3.14f;

        public static final float area(float raio) {
                return PI * raio * raio;
        }

        public static void main(String[] args) {
                System.out.println(CalculaArea.PI );
                System.out.println(CalculaArea.area(5.3f));
        }
}
```

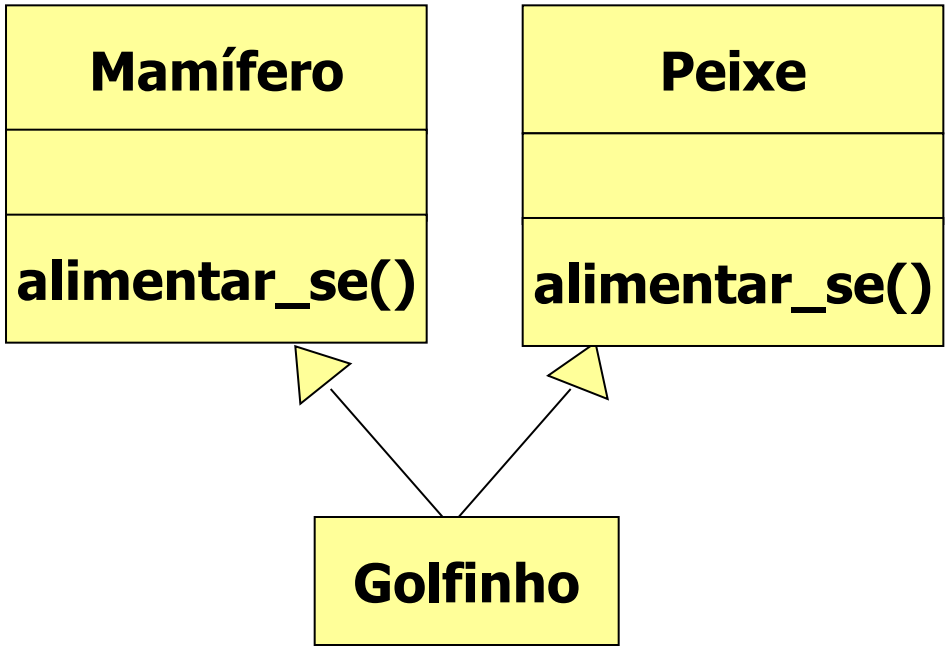
Herança Múltipla

- Uma classe herda características de mais de uma superclasse



Herança Múltipla

- Alguns problemas são introduzidos



- Java não permite herança múltipla

Dicas para Herança Eficaz

- Sempre use a regra "é um"
- Em geral, use herança para reutilização de interface e para definir relacionamentos de substituição.
- Como regra geral, mantenha suas hierarquia de classes relativamente rasas.

Dicas para Herança Eficaz

- Se você adicionar métodos ou atributos especificamente para uso por subclasses, certifique-se de torná-los protegidos.

- Em geral, evite abrir a implementação interna do seu objeto para subclasses. A subclasse pode se tornar dependente da implementação.

Por que usar Herança?

- Natural: permite modelar o mundo mais naturalmente. Permite trazer tendências de categorização, natural para os seres humanos, para implementação.
- Reutilizável: auxilia na reutilização. Classes antigas reutilizadas na construção de novas classes.
- Confiável: a reutilização de código testado significa que possivelmente haverá menos erros

Por que usar Herança?

- Manutenível: modificações localizadas
- Extensível: torna a extensão ou especialização da classe possível.
- Oportuno: ajuda a diminuir o tempo de desenvolvimento.

Referências

- Slides “Pilares da POO - Herança”. Profª. Débora. UFS. 2010.
- Slides “OO-Herança” Prof. Marcos Dósea. UFS. 2010.
- Slides “Herança” Prof. Giovanny . Java.UFS. 2009.
- Aprenda Programação Orientada a Objetos em 21 dias.
  - Dia 4