Lógica de Programação Orientada a Objetos

Aula 05

- Paradigma da Orientação a Objeto
- Classes
- Objetos
- Encapsulamento
- Herança
- Polimorfismo

Paradigma da Orientação a Objetos

Paradigma da Orientação a Objetos

- Termo criado por Alan Kay, autor da linguagem de programação *Smalltalk*.
- A primeira linguagem a utilizar as idéias deste paradigma foi a linguagem de programação *Simula 67,* criada em 1967.
- Conceito antigo, que só agora vem ganhando aceitação na comunidade de desenvolvimento de software.
- Paradigma é um modelo, um conjunto de idéias, uma base filosófica, um padrão a ser seguido no processo de desenvolvimento de software.

Paradigma da Orientação a Objetos

Programação Orientada a Objetos

- A programação orientada a objetos foi criada para tentar aproximar o mundo real do mundo virtual.
- O desenvolvedor é responsável por moldar o mundo dos objetos, definindo também como eles irão se relacionar.
- Utiliza-se nesta construção a técnica de **abstração de dados**.

Paradigma da Orientação a Objetos

Abstração de dados

- Capacidade de modelar o mundo real, onde restringimos o nosso universo de análise, desprezando as informações irrelevantes para a solução do problema.

Podemos demonstrar o uso de abstração facilmente, quando fechamos os olhos e pensamos em uma mesa.

Esta mesa imaginária provavelmente não vai ser igual à outra imaginada por outras pessoas, mas o que importa é que todas as pessoas que imaginaram uma mesa colocaram nessa as informações que para elas são necessárias para a sua função (de ser uma mesa).

Paradigma da Orientação a Objetos

Programação Orientada a Objetos

- **Linguagens de programação orientada a objetos**
  - *Smalltalk*
  - *Python*
  - *Ruby on Rails*
  - *Java*
  - *Object Pascal*
- **Linguagens de programação com suporte a orientação a objetos**
  - *PHP (a partir do 4.0)*
  - *ColdFusion*
  - *ActionScript*
  - *Visual Basic .NET*

Paradigma da Orientação a Objetos

Programação Orientada a Objetos

- Na modelagem do mundo real para o mundo virtual, o desenvolvedor implementa um conjunto de classes que definem a estrutura dos objetos presente no sistema.

- Cada classe define os comportamentos (métodos) e propriedades (atributos) de seus objetos, assim como o relacionamento entre eles.

Classes !? **Objetos !?**

Classes

Classes

- Uma classe define um objeto.
- A classe é um “esqueleto” do objeto.
- Através dela definimos quais propriedades e comportamentos seus objetos terão.
- São estruturas que modelam os objetos do mundo real, de acordo com a abstração que o desenvolvedor teve do cenário dado.

Classes

Suponha que estamos criando um software para gerenciar esta sala de aula. A partir deste cenário, podemos criar classes como:
Pessoa, Aluno, Professor, Computador, Monitor, Cadeira, Ar Condicionado, e assim por diante. Tudo que apresenta propriedades bem definidas é candidato a ser uma classe no mundo virtual.

Criada as classes, devemos definir as propriedades e comportamentos de cada uma, por exemplo:
Pessoa terá um nome, um cpf e um endereço como propriedades, e como comportamentos ela poderá andar, falar e comer.

Classes

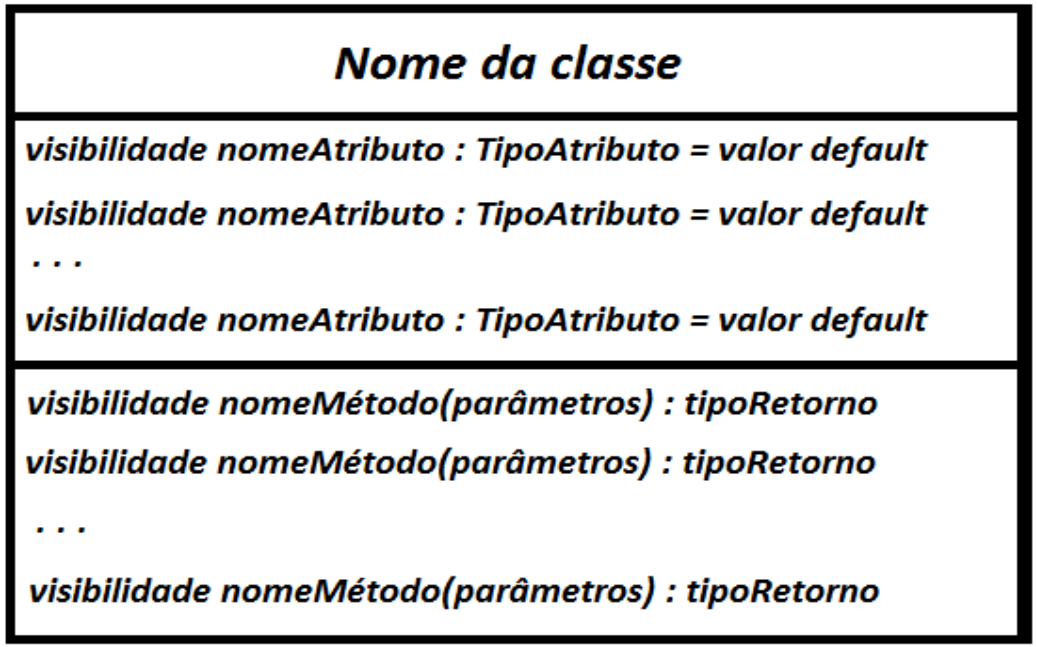
- As propriedades de uma classe serão representadas através de variáveis (atributos).

- Os comportamentos de uma classe serão representados através de métodos.

Classe	Propriedades	Comportamento
Carro	cor ano modelo nome	acelerar frear abrir a porta

Classes

Podemos representar uma classe através de uma linguagem de modelagem, como por exemplo a *UML.*



Classes

- O ***nome da classe*** é o seu identificador.

- Os ***atributos*** representam as ***propriedades*** da classe. Cada atributo é identificado por um nome e um tipo associado, ou seja, são variáveis da classe.

- Os ***métodos*** definem os comportamentos da classe, suas funcionalidades, ou seja, o que o objetos desta classe conseguem fazer.

Classes

- O ***modificador de acesso*** pode ser aplicado tanto a atributos quanto a métodos, chamados de membros da classe. Em geral podem ser especificados em três grupos:

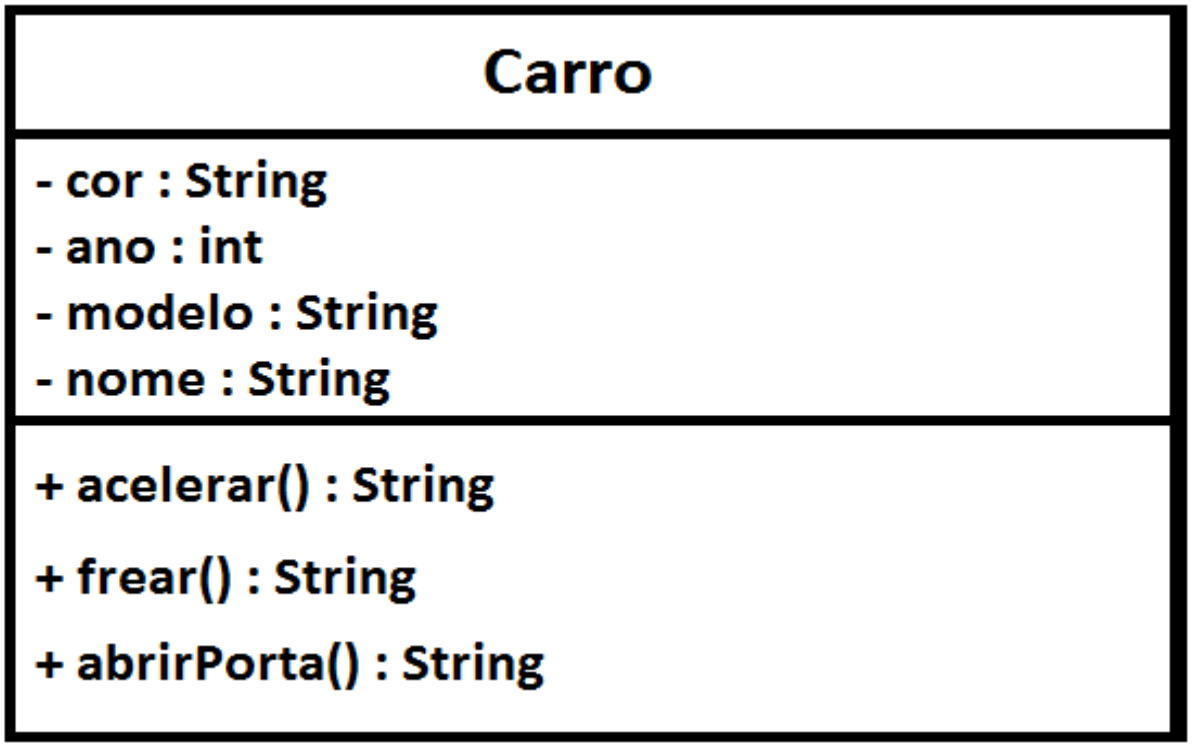
  - **públicos –** Denotado na UML pelo símbolo “+”. Indica que este membro pode ser acessado por qualquer outro objeto.

  - **privados –** Denotado na UML pelo símbolo “-”. Indica que este membro pode ser acessado apenas na classe onde foi declarado.

  - **protegidos –** Denotado na UML pelo símbolo “#”. Indica que o membro pode ser acessado por outros objetos criados de uma classe armazenada no mesmo pacote (diretório), ou por outros objetos que não se encaixam nesta regra, através de um mecanismo conhecido como herança.

Classes

Exemplo 1: ***Classe Carro em UML.***



Classes

Exemplo 2: ***Classe Carro em Java*.**

```
1  public class Carro {
2
3      private String cor;
4
5      private int ano;
6
7      private String modelo;
8
9      private String nome;
10
11     public Carro() {
12         super();
13     }
14
15     public String acelerar() {
16         return "acelerando...";
17     }
18
19     public String frear() {
20         return "freando...";
21     }
22
23     public String abrirPorta() {
24         return "abrindo porta...";
25     }
26
27 }
```

propriedades

comportamentos

Classes

Exemplo 3: ***Classe Carro em PHP.***

```php
<?php

    class Carro {

        private $cor;

        private $ano;

        private $modelo;

        private $nome;

        function _construct() {}

        function acelerar(){
            return "acelerando...";
        }

        function frear() {
            return "freando...";
        }

        function abrirPorta() {
            return "abrindo porta...";
        }

    }

?>
```

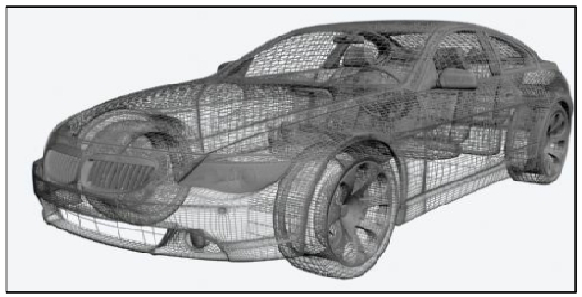
propriedades

comportamentos

Objetos

Objetos

- São a chave para a implementação de sistemas complexos e eficientes.
- Um objeto é uma instancia de uma classe.
- Na classe temos a definição do objeto. Ao atribuir valores a estas propriedades estamos instanciando a classe, ou seja, criando um objeto.



Classe

Objeto

Objetos

- Assim como as variáveis, um objeto ocupa um espaço na memória.
- Os objetos se comunicam através de invocação de métodos uns dos outros.
- Todo objeto possui um ciclo de vida que engloba o momento em que ele é declarado até sua eliminação.
- No instante em que ele é declarado, é alocado um espaço na memória para ele.

Objetos

- Para sua criação, é invocado seu construtor. Todo objeto possui um construtor, que é responsável pela sua criação.

- Após ser utilizado, a sua eliminação pode acontecer de duas formas:

    - A primeira, que todas as linguagens utilizam, elimina o objeto no final do programa se ele for global, no final de um método se for local, e no final de um bloco se for declarado dentro deste.

    - A segunda forma de eliminação é chamada de **Garbage Collection**.

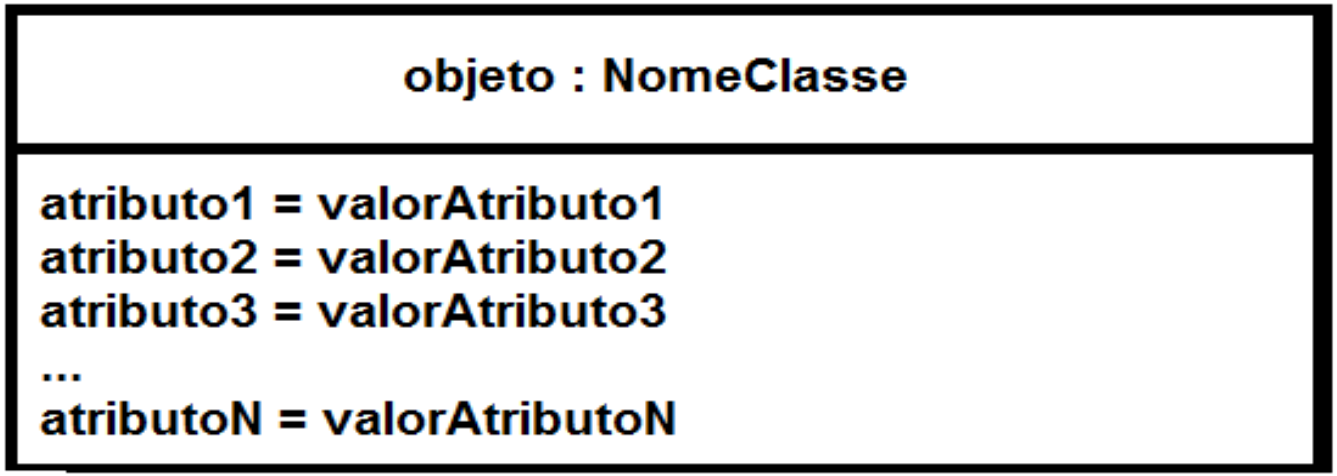
Objetos

Garbage Collection

- Consiste na eliminação do objeto pelo compilador, depois de sua última utilização.
- A partir do momento em que o objeto não pode ser mais utilizado, ou seja, não é mais referenciado, passa a ser inútil na memória, podendo ser eliminado.
- Garbage Collection é implementado em várias linguagens como Java e Smalltalk, enquanto por exemplo em C++ e Pascal não temos essa funcionalidade.

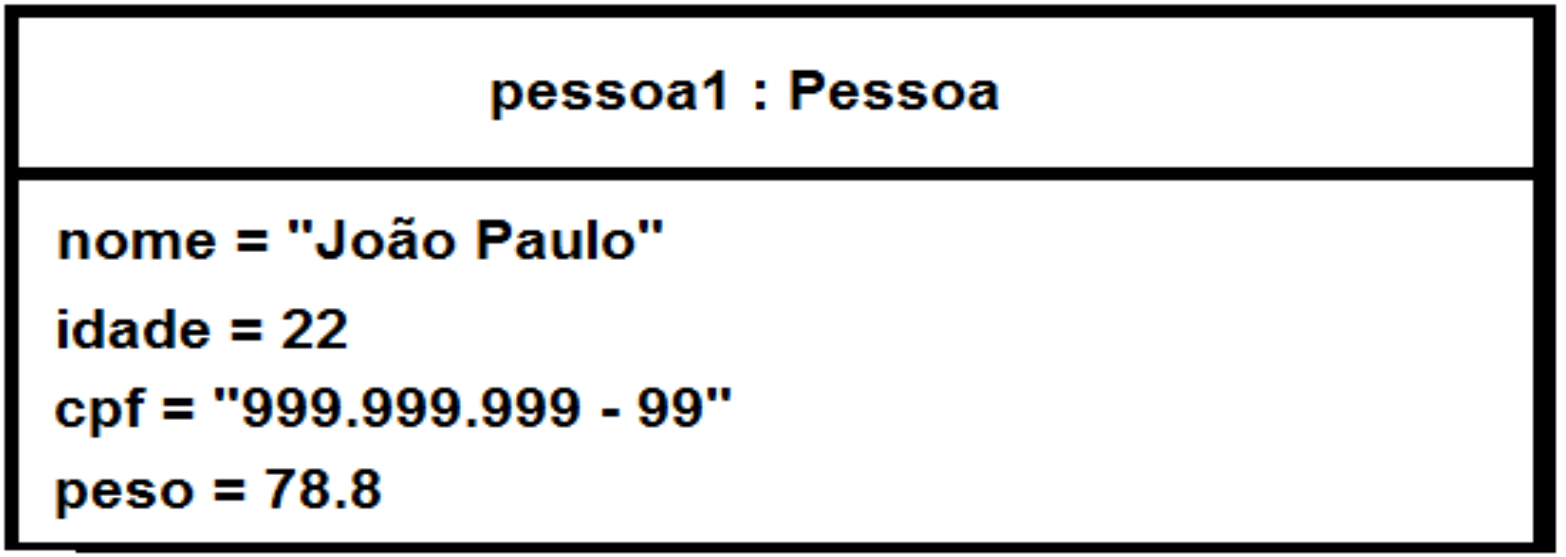
Objetos

Podemos representar um objeto através de uma linguagem de modelagem, como por exemplo a *UML:*



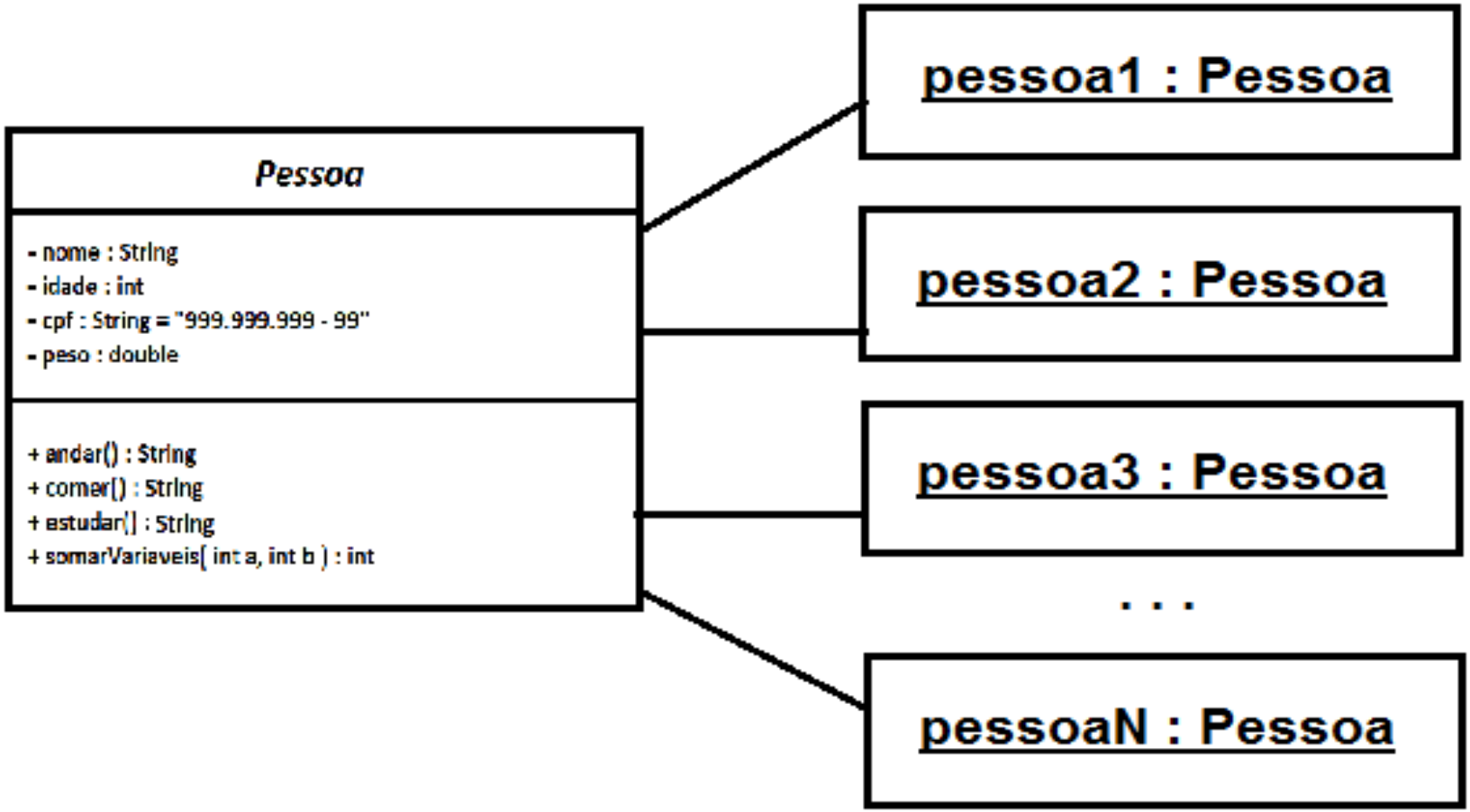
Objetos

Exemplo 1: ***Objeto pessoa1 em UML.***



Objetos

Exemplo 1: ***Representação dos objetos da classe Pessoa em UML.***



Objetos

Exemplo 1: ***Criação do objeto pessoa1 em Java.***

```
1  Pessoa pessoa1 = new Pessoa();
2
3  pessoa1.nome = "João Paulo";
4  pessoa1.idade = 22;
5  pessoa1.cpf = "999.999.999-99";
6  pessoa1.peso = 78.8;
```

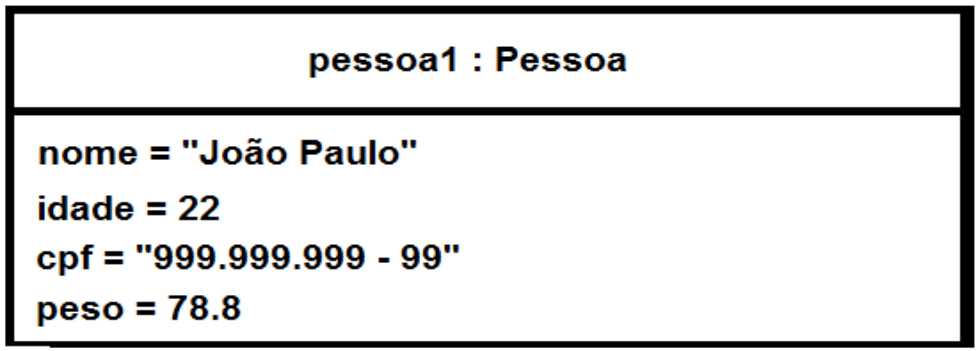
pessoa1 : Pessoa
nome = "João Paulo" idade = 22 cpf = "999.999.999 - 99" peso = 78.8

Objetos

Exemplo 1: ***Criação do objeto pessoa1 em PHP.***

```php
$pessoa1 = new Pessoa();

$pessoa1->nome = 'João Paulo';
$pessoa1->idade = 22;
$pessoa1->cpf = '999.999.999-99';
$pessoa1->peso = 78.8;
```



Encapsulamento

Encapsulamento

- Uma das técnicas mais utilizadas no paradigma da orientação a objetos.

- Encapsular, na programação, significa esconder os dados das propriedades de uma classe de seus utilizadores, pois é responsabilidade da classe manter seus dados íntegros.

- Para implementar o encapsulamento em uma classe, devemos:
    - Manter as propriedades privadas.
    - Criar métodos públicos getters (para obter) e setters (para alterar) o valor destas propriedades.

Encapsulamento

- Sintaxe do método **getter**, para **obter** o valor do atributo:

```
public tipoAtributo getNomeAtributo() {
        return atributo ;
}
```

- Sintaxe do método **setter**, para **alterar** o valor do atributo:

```
public void setNomeAtributo ( tipoAtributo parametro ) {
        atributo = parametro;
}
```


Encapsulamento

Na codificação da classe pessoa, os métodos getter e setter do atributo *nome*, por exemplo, poderiam ser implementados da seguinte forma:

```
1  private String nome;
2
3  public String getNome() {
4      return nome;
5  }
6
7  public void setNome(String n) {
8      nome = n;
9  }
```

propriedade privada

métodos de acesso público

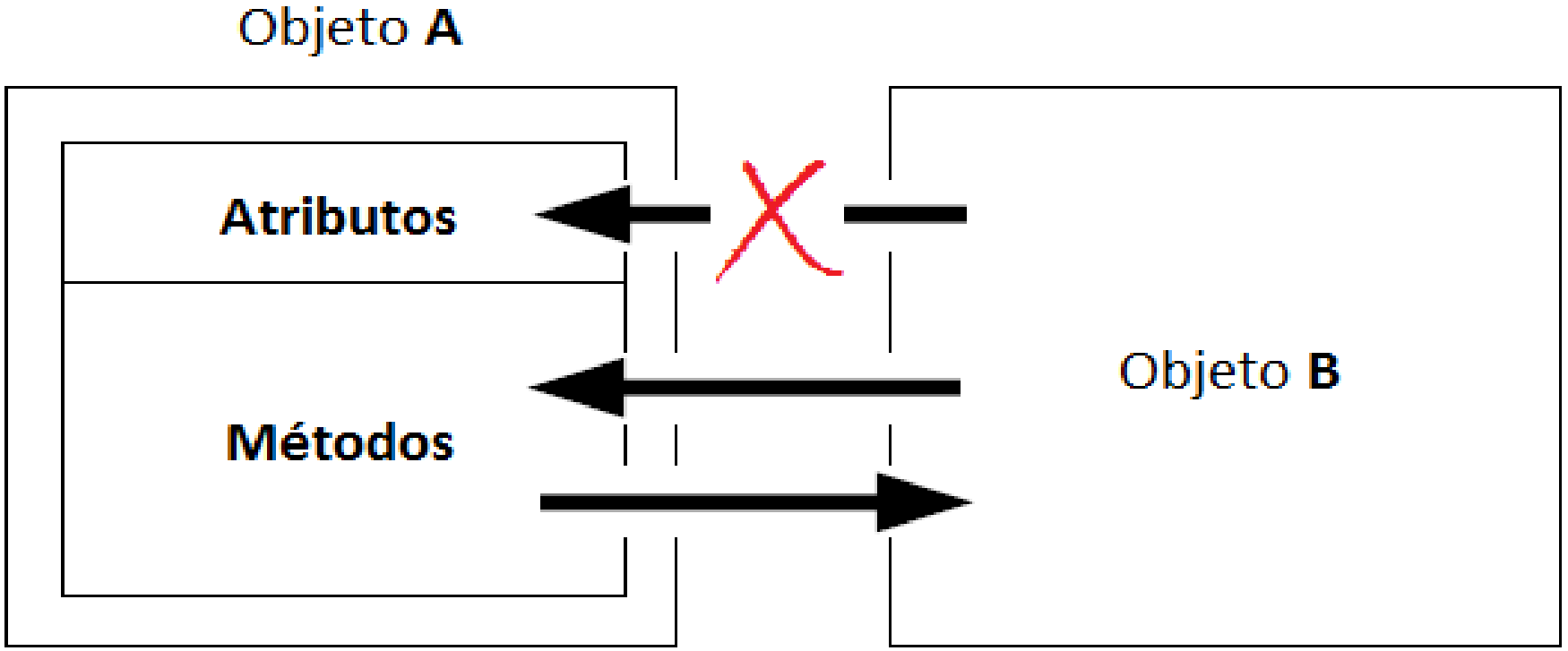
```
Pessoa pessoa1 = new Pessoa();

pessoa1.setNome("Marco Aurélio");
```

```
pessoa1.nome = "Marco Aurélio";
```

X

Encapsulamento



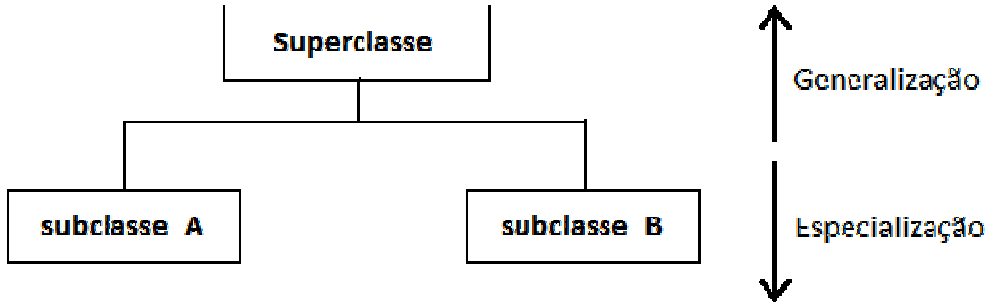
Herança

Herança

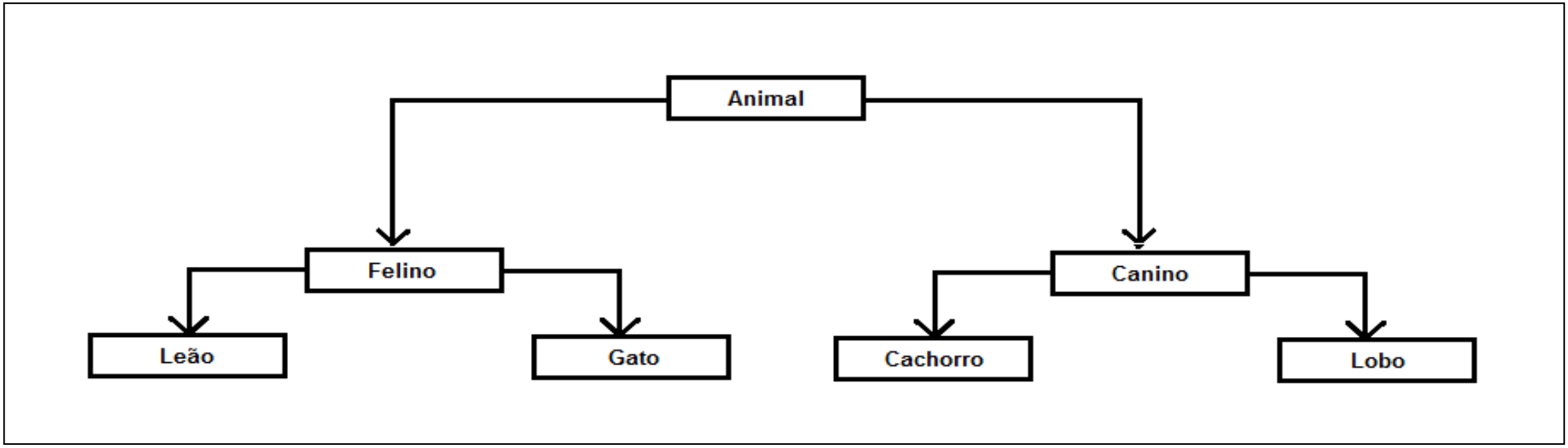
- É um mecanismo que permite que características comuns a diversas classes sejam agrupadas em uma classe base, ou superclasse.

- A partir desta superclasse, outras classes podem ser especificadas. Cada classe especificada, ou subclasse, herda as propriedades e comportamentos de sua superclasse e acrescenta o que for definido de particularidade para ela.

- Evita a redundância no código e aumenta sua capacidade de reutilização.



Herança



Podemos ver na figura acima, que quanto mais descemos na árvore da herança, mais específicos nós somos, e quanto mais subimos, mais gerais nós somos.
Pela definição, Leão e Gato (subclasses) são Felinos (superclasse), portanto eles herdam comportamentos em comum de um Felino, mas acrescentam propriedades particulares de cada um, como porte físico por exemplo. Da mesma forma, Felino e Canino (subclasses) são Animais (superclasse), portanto tanto Canino quanto Felino tem comportamentos de animais, pois os herdaram, mas Felino tem suas particularidades e Canino também.

Herança

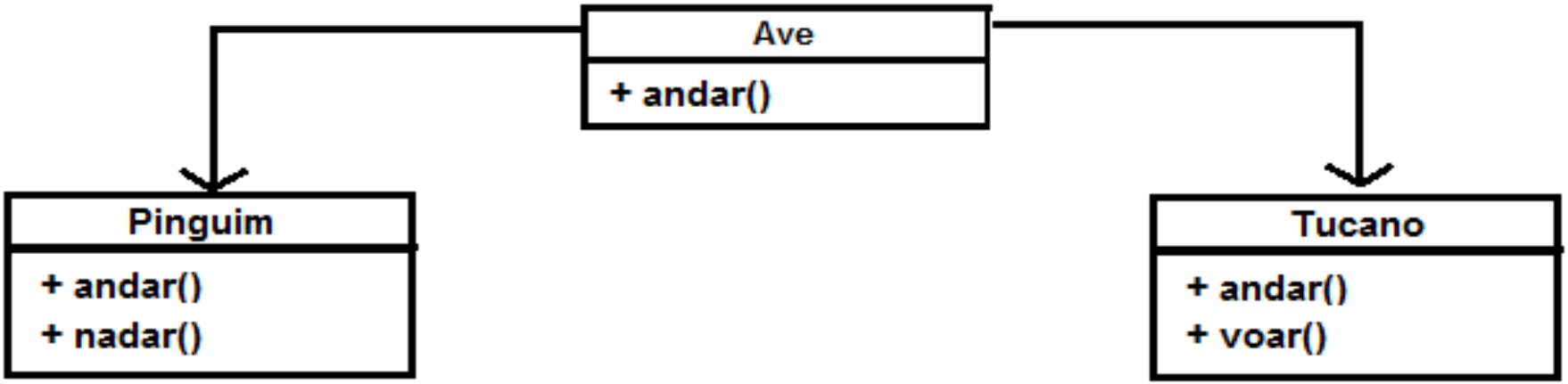
É assim que funciona o mecanismo da herança, as subclasses herdam propriedades e comportamentos da superclasse, e acrescenta o que é particular de cada uma. Segue um exemplo para deixar claro o funcionamento da herança:

Polimorfismo

Polimorfismo

- Polimorfismo = “muitas formas”.
- Está intimamente ligado ao conceito de herança.
- É o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que tem a mesma assinatura, mas implementações distintas, usando para isso uma variável do tipo da superclasse.

Polimorfismo



Por exemplo, suponha que o algoritmo utilizado no método **andar()** da classe **Pinguim** retorne o texto **"Pinguim andando..."** e o algoritmo utilizado no método **andar()** de **Tucano** retorne **"Tucano andando...".** Pela definição de herança os dois objetos, Pinguim e Tucano, podem andar, mas Pinguim anda do seu jeito e Tucano do seu, por isso que os métodos possuem implementações diferentes.

Polimorfismo

Utilizando a linguagem Java como exemplo:

```
1    Ave a = new Pinguim();
2    a.andar();
3
4    a = new Tucano();
5    a.andar();
```

Saída:

```
Pinguim andando...
Tucano andando...
```

Na linha 1 criamos um objeto do tipo Pinguim e armazenamos em uma variável do tipo Ave. Isso é possível porque Pinguim é uma Ave, segundo nossa representação.

Na linha 2 invocamos o método andar() do objeto Pinguim (subclasse) através de uma variável Ave (superclasse), fazendo uso do Polimorfismo.
Da mesma forma, utilizamos o polimorfismo na linha 5, através do objeto Tucano.

Revisão LPOO