

# Programação Orientada a Objetos

Prof. Dr. Rodrigo Plotze

[rodrigoplotze@gmail.com](mailto:rodrigoplotze@gmail.com)

# Conteúdo

- Programação Orientada a Objetos
  - Classes, Objetos, Atributos e Métodos
  - Encapsulamento
  - Construtores
  - Herança
  - Polimorfismo
  - Classes e Métodos Abstratos
  - Interfaces e Herança Múltipla

# **CLASSES, OBJETOS, ATRIBUTOS E MÉTODOS**

# Classes, Objetos, Atributos e Métodos

- Definição sobre ***Objetos***
  - Na OO, objeto é uma abstração dos objetos reais existentes.
  - Em uma sala de aula, por exemplo, existem diversos objetos: alunos, cadeiras, mesas, lousa etc.
  - Se for necessário manter controle de uma sala de aula, pode ser elaborado um software que manipula objetos desse tipo.

# Classes, Objetos, Atributos e Métodos

- Definição sobre ***Classes***
  - As classes representam um modelo formal para criação de objetos.
  - As classes são definidas como moldes para objetos.
    - Com a definição de uma classe é possível criar inúmeros objetos.
  - Uma classe pode ser definida como um tipo abstrato de dados definido pelo usuário.

# Classes, Objetos, Atributos e Métodos

- A especificação de uma classe é realizada por meio da abstração do problema, em termos de suas características e comportamentos.
- Estes conceitos são conhecidos em programação orientada a objetos como:
  - *atributos*
  - *métodos*

# Classes, Objetos, Atributos e Métodos

## ▪ *Pensando em Classes*

Carro	
<ul style="list-style-type: none"><li>— Cor</li><li>— Potência do motor</li><li>— Ano de fabricação</li><li>— Modelo</li><li>— Fabricante</li></ul>	Características
<ul style="list-style-type: none"><li>— Ligar o motor</li><li>— Desligar o motor</li><li>— Acelerar</li><li>— Frear</li><li>— Trocar de marcha</li></ul>	Comportamentos

# Classes, Objetos, Atributos e Métodos

- *Pensando em Objetos*





# Classes, Objetos, Atributos e Métodos

```
public class Carro {  
  
    private String cor;  
    private double potencia;  
    private int ano;  
    private String modelo;  
    private String fabricante;  
  
    public Carro() {  
    }  
    public void ligar(){  
        System.out.println("Carro ligado");  
    }  
    public void desligar(){  
        System.out.println("Carro desligado");  
    }  
    public void acelerar(){  
        System.out.println("Acelerando...");  
    }  
    public void frear(){  
        System.out.println("Freando...");  
    }  
    public void trocarmarcha(){  
        System.out.println("Trocar marcha");  
    }  
}
```

Carro
— Cor
— Potência do motor
— Ano de fabricação
— Modelo
— Fabricante
— Ligar o motor
— Desligar o motor
— Acelerar
— Frear
— Trocar de marcha

# Classes, Objetos, Atributos e Métodos

- **Objetos**

- O conceito de objeto está relacionado a instanciação de uma classe.

```
Carro c1 = new Carro();
```

```
c1.ligar();
```

```
c1.trocarmarcha();
```

```
c1.desligar();
```

# Classes, Objetos, Atributos e Métodos

## ■ Atributos

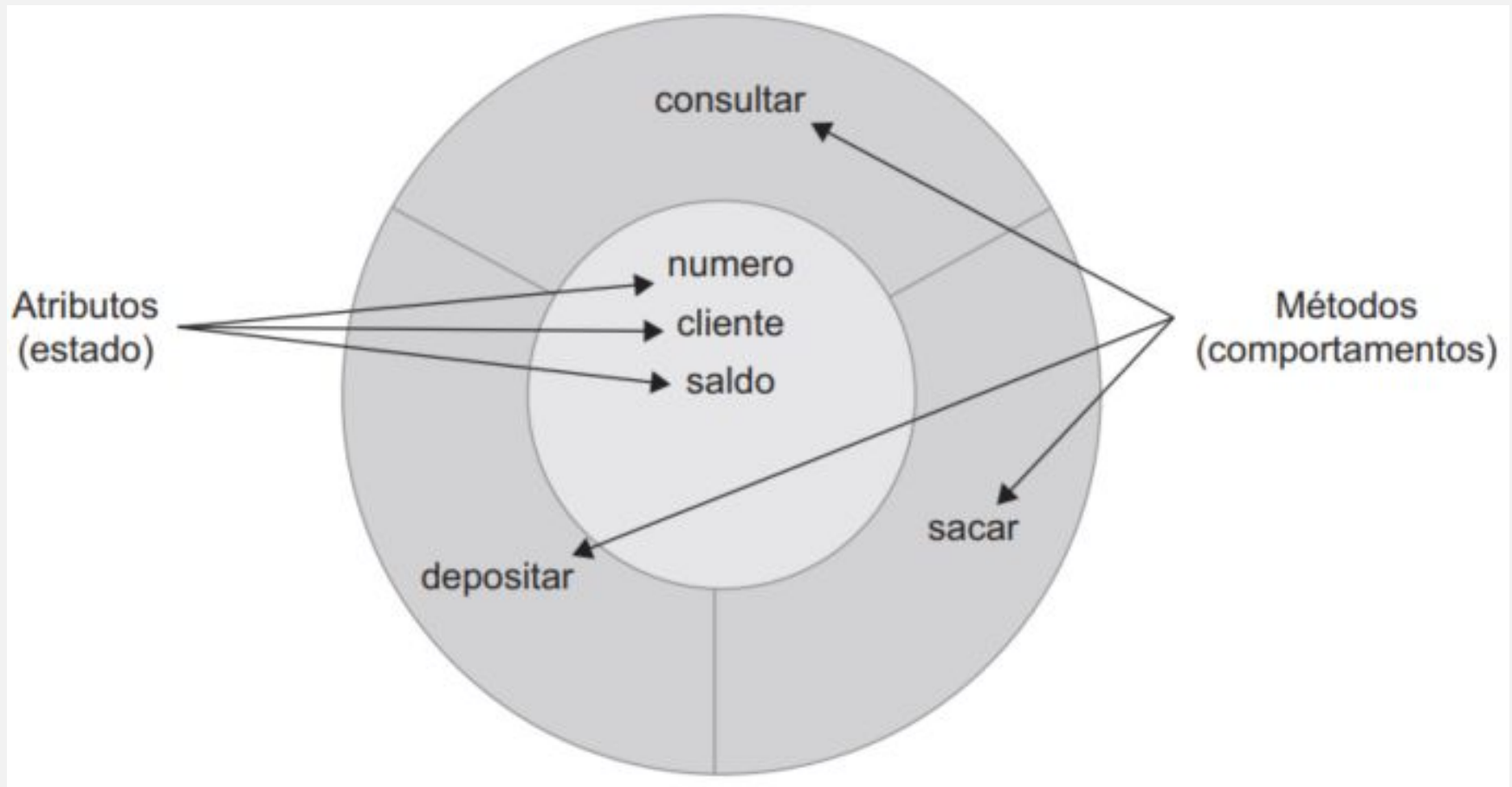
- são responsáveis pelo armazenamento dos dados dos objetos.
- As características dos dados são definidas na descrição da classe.
- Para cada atributo contido na classe é necessário especificar o tipo de dados que será armazenado.

# Classes, Objetos, Atributos e Métodos

## ■ Métodos

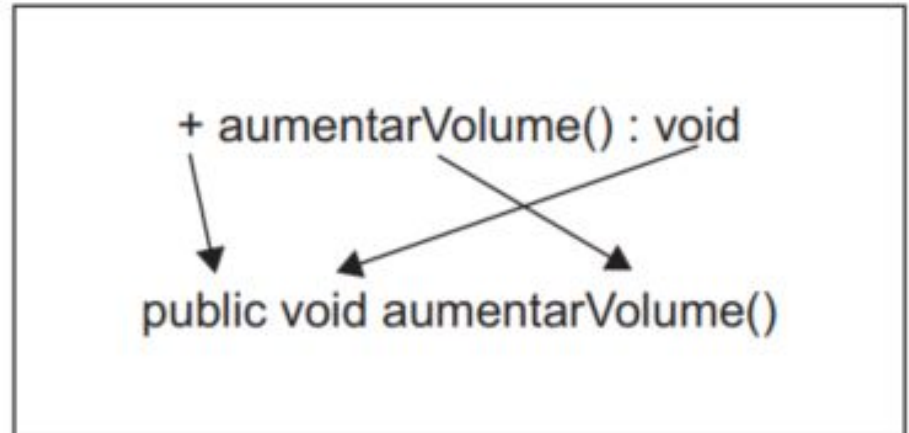
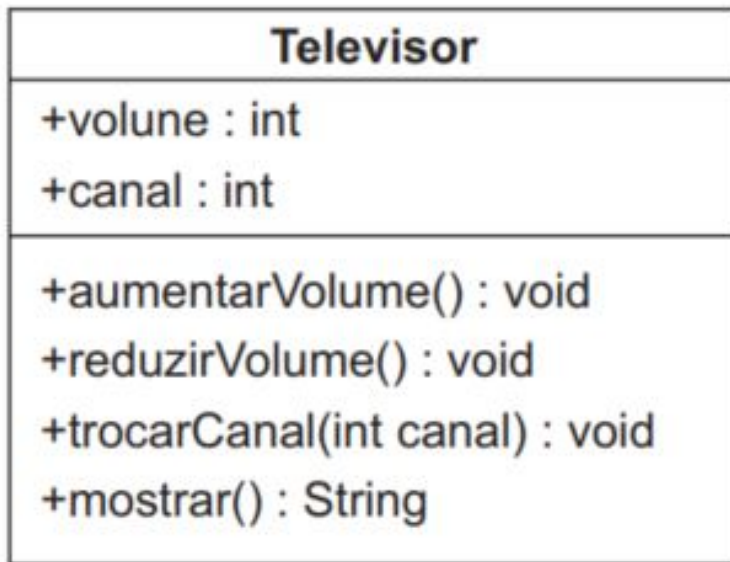
- Os métodos são utilizados para definir os comportamentos que serão executados pelos objetos.
- A especificação de um método é realizada no contexto da classe, assim, é possível definir quais ações serão realizadas quando o método for invocado.

# Abstração de uma *Conta Bancária*



# Classes, Objetos, Atributos e Métodos

- Linguagem de Modelagem Unificada (**UML**)
  - Permite representar graficamente os elementos da programação orientada a objetos.



**Correspondência UML e Java**

# **ATIVIDADE PRÁTICA**

# Atividade Prática

## ■ **Exercício 1**

- Especifique uma classe denominada **Pessoa** que contenha atributos para armazenar os seguintes dados: Nome, Peso e Altura
- Crie um método para: calcular o IMC:  
( $\text{peso}/(\text{altura}*\text{altura})$ )
- Especifique a classe **PessoaTest** contendo quatro objetos instanciados da classe Pessoa.



# Atividade Prática

## ■ **Exercício 2**

- Escreva uma classe que receba um número inteiro positivo e maior que zero.
- Especifique métodos para determinar:
  - a raiz quadrada do número;
  - o número ao quadrado;
  - o número ao cubo.
- Por fim, especifique a classe de teste.

Você foi contratado para desenvolver um **sistema simples de controle de contas bancárias em Java**. Sua tarefa é implementar a classe `ContaBancaria`, que representará uma conta bancária básica. A classe `ContaBancaria` deve possuir os seguintes atributos e métodos:

- **Atributos:**

- `numero_conta`: número da conta bancária
- `saldo`: saldo atual da conta bancária

- **Métodos:**

- `depositar(valor)`: Método que permite depositar um valor na conta. O valor do depósito deve ser maior que zero.
- `sacar(valor)`: Método que permite sacar um valor da conta, desde que haja saldo suficiente. O valor do saque deve ser maior que zero e não pode exceder o saldo disponível na conta.
- `verificar_saldo()`: Método que exibe o saldo atual da conta.

- Desenvolva a classe `ContaBancaria` e teste suas funcionalidades com algumas operações de depósito, saque e verificação de saldo.

# **ENCAPSULAMENTO**

# Encapsulamento

- Encapsulamento (em inglês *data hiding*) é mais um dos conceitos presentes na orientação a objetos.
- Trata-se de um mecanismo que possibilita restringir o acesso a variáveis e métodos da classe (ou até à própria classe).

# Encapsulamento

- Os detalhes de implementação ficam ocultos ao usuário da classe, isto é, o usuário passa a utilizar os serviços da classe sem saber como isso ocorre internamente.
- Somente uma lista das funcionalidades existentes torna-se disponível ao usuário da classe.
- O encapsulamento é também conhecido como ***acessibilidade***, pois define o que está acessível na classe.

# Encapsulamento

- Para determinar o nível de acesso dos elementos de uma classe, são usados os qualificadores de acesso:
  - **public**
  - **private**
  - **protected**
  - **package**

# Encapsulamento

- Visibilidade (***Restrições de Acesso***)
  - **public (+)**: um nível sem restrições, equivalente a não encapsular, ou seja, se uma variável for definida como pública, não será possível realizar o encapsulamento.
  - **Private (-)**: o nível de maior restrição em que apenas a própria classe pode ter acesso a variáveis e/ou métodos.

# Encapsulamento

- Visibilidade (***Restrições de Acesso***)
  - **protected (#)**: um nível intermediário de encapsulamento em que as variáveis e métodos podem ser acessados pela própria classe ou por suas subclasses (veremos isso mais adiante).
  - **package (~)**: nível em que a classe pode ser acessada apenas por outras classes pertencentes ao mesmo pacote (veremos a manipulação de pacotes mais à frente).



# Encapsulamento

- Visibilidade (***Restrições de Acesso***)

Televisor
<ul style="list-style-type: none"><li>- volume: int</li><li>- canal : int</li></ul>
<ul style="list-style-type: none"><li>+aumentarVolume() : void</li><li>+reduzirVolume() : void</li><li>+trocarCanal(int canal) : void</li><li>+mostrar() : String</li></ul>

# Encapsulamento

- Métodos Manipuladores de Acesso: GET/SET
  - **set** é utilizado para modificar a informação contida em um atributo
  - **get** é empregado no retorno da informação contida no atributo.

```
private String nome;  
  
public String getNome() {  
    return this.nome;  
}  
  
public void setNome(String nome) {  
    this.nome = nome;  
}
```

# Encapsulamento

- Uso da palavra reservada ***this***
  - Faz uma referência ao objeto corrente, isto é, ao objeto que chamou o método.

```
private String nome;
```

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

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

# ATIVIDADE PRÁTICA

# Atividade Prática

- **Exercício 1:** Escreva uma classe denominada **Televisor** que contenha dois atributos encapsulados: **volume** e **canal**.
- Especifique os métodos modificadores de acesso **set/get** para cada atributo, bem como, os seguintes métodos:
  - **aumentarVolume()**
  - **reduzirVolume()**
  - **trocarCanal(int canal)**

Por fim, escreva a  
classe  
**TelevisorTest**

# Atividade Prática

- **Exercício 2:** Escreva uma classe denominada **Funcionario** que contenha os atributos encapsulados: **nome**, **salario** e **ano de contratação**.
- Especifique os métodos modificadores de acesso **set/get** para cada atributo, bem como, os seguintes métodos:
  - **getBonificacao()**
    - 5% de bonificação para mais de 5 anos
    - 10% de bonificação para mais de 10 anos
    - 20% de bonificação para mais de 20 anos
  - **getSalarioTotal()**

# Atividade Prática

- **Exercício 3:** Desenvolver uma aplicação Java utilizando todos os conceitos de POO apresentados até o momento para Cálculo do Salário Total de um funcionário.
- Requisitos Funcionais
  - Entrada de Dados
    - Salário Base
    - Total de Horas Trabalhadas no Mês
    - Total de Horas Extras
  - Saída
    - Salário Total

# Atividade Prática

## ■ ***Exercício 3: (continuação)***

### ■ Exemplo de Cálculo

- Salário Base = R\$ 1000,00
- Horas Trabalhadas no Mês = 220
- Total de Horas Extras = 10
- Valor da Hora Trabalhada = R\$ 4,54  
(Salário Base/Horas Trabalhadas no Mês)
- Valor da Hora Extra = R\$ 6,81  
(Valor da Hora Trabalhada + 50%)
- Valor Total de Horas Extras = R\$ 68,18  
(Total de Horas Extras \* Valor da Hora Extra)
- Salário Total = R\$ 1068,18  
(Salário Base + Valor Total de Horas Extras)



# CONSTRUTORES

# Construtores

- No paradigma de programação estruturado o primeiro bloco de instruções a ser executado em um programa é o bloco principal, geralmente conhecido como *main*.
- No paradigma de programação orientado a objetos o primeiro bloco de instruções a ser executado quando um objeto é instanciado é um método conhecido como construtor.

# Construtores

- Os construtores podem ser definidos como métodos especiais que são responsáveis por especificar ações necessárias para a existência de um objeto.
- O nome do método construtor deve ser exatamente igual ao nome da classe, respeitando caracteres definidos como maiúsculo ou minúsculo.

# Construtores

- Geralmente as classes são compostas por vários construtores, sendo:
  - ***Construtor Padrão***: que não possui parâmetros e que inicializa os atributos com os valores *default*.
  - ***Construtores Sobrecarregados***: que permitem a passagem de parâmetros durante a instância da classe

# Construtores

```
public class Veiculo {  
  
    //ATRIBUTOS  
    private String marca;  
    private String modelo;  
    private String cor;  
    private int ano;  
  
    //CONSTRUTOR  
    public Veiculo() {  
        System.out.println("Executando construtor");  
        this.marca = "";  
        this.modelo = "";  
        this.cor = "";  
        this.ano = 0;  
    }  
}
```

...

# Construtores

- Sobrecarga de Construtores
  - Permite especificar vários construtores para mesma classe.
  - Durante o processo de instância da classe o programador pode escolher qual construtor é mais adequado para utilizar na criação do objeto.

# Construtores Sobrecarregados

```
public class Funcionario {  
    private String nome;  
    private int idade;
```

```
    public Funcionario() {  
        this.nome = "";  
        this.idade = 0;  
    }
```

```
    public Funcionario(String nome) {  
        this.nome = nome;  
    }
```

```
    public Funcionario(String nome, int idade) {  
        this.nome = nome;  
        this.idade = idade;  
    }  
}
```

```
Funcionario f1 = new Funcionario();  
Funcionario f2 = new Funcionario("Ana Maria");  
Funcionario f3 = new Funcionario("Carlos", 48);
```

# **ATIVIDADE PRÁTICA**



# Atividade Prática

## ■ *Exercício 1*

- Considere a representação da classe Brinquedo:

Brinquedo
<ul style="list-style-type: none"><li>– nome : String</li><li>– faixaEtaria : String</li><li>– preco : double</li></ul>
<ul style="list-style-type: none"><li>+ Brinquedo()</li><li>+ Brinquedo(nome : String)</li><li>+ Brinquedo(nome : String, preco : double)</li><li>+ toString()</li></ul>

# Atividade Prática

## ■ **Exercício 1 (continuação)**

- Elabore essa classe em Java **Brinquedo** contendo os métodos get e set necessários e os métodos construtores apresentados.
- O atributo faixaEtaria é um atributo do tipo String que deve receber apenas um dos valores seguintes: “0 a 2”, “3 a 5”, “6 a 10” e “acima de 10”.
- Outros valores são inválidos e não devem ser armazenados. Essa validação deve ser realizada no método setFaixaEtaria.
- Por fim, elabore a classe **BrinquedoTest** para testar as funcionalidades da classe Brinquedo.

# Atividade Prática

## ▪ **Exercício 2**

- Desenvolva uma classe denominada **Retângulo** com os atributos comprimento e largura.
- Forneça métodos que calculem o perímetro e a área do retângulo.
- Os métodos *set* devem verificar se o comprimento e largura são, cada um, números ponto flutuante maiores que 0,0 e menores que 20,0.

# Atividade Prática

## ▪ ***Exercício 2 (continuação)***

- A estrutura da classe deve conter:
  - Atributos encapsulados
  - 1 (um) construtor padrão
  - 2 (dois) construtores sobrecarregados
  - Métodos modificadores de acesso (get/set)
  - Método toString
- Escreva uma classe de teste para demonstrar o funcionamento da classe Retângulo e o uso de todos os construtores.

# Atividade Prática

## ■ ***Exercício 3***

- Elabore uma classe denominada Hora contendo os atributos hora, minuto e segundo.
- Os métodos *set* deverão garantir a integridade dos dados atributos ao objeto, de forma que a hora esteja no intervalo de 0 a 23, os minutos no intervalo de 0 a 59 e os segundos no intervalo de 0 a 59.
- Especifique métodos para as seguintes tarefas:
  - Incrementar e decrementar o valor da hora
  - Incrementar e decrementar o valor do minuto
  - Incrementar e decrementar o valor do segundo
  - Retornar a hora completa no formato hh:mm:ss

# Atividade Prática

## ▪ ***Exercício 3 (continuação)***

- A estrutura da classe deve conter:
  - Atributos encapsulados
  - 1 (um) construtor padrão
  - 4 (quatro) construtores sobrecarregados
  - Métodos modificadores de acesso (get/set)
  - Método toString
- Escreva uma classe de teste para demonstrar o funcionamento de todos os construtores.

# Atividade Prática

## ■ **Exercício 4**

- Crie uma classe chamada **Controle** que especifique os métodos “andar”, “virar” e “falar”.
- A seguir, crie uma classe chamada **Robô** que implemente esses métodos.
- Dentro de cada método imprima uma mensagem em tela contendo a ação correspondente.
- Para testar, elabore uma terceira classe chamada **RoboTest**.

**HERANÇA**



# Herança

- Na área de desenvolvimento de software um dos principais desafios encontrados pelos programadores é a reutilização de software.
  - o reuso de software tem como objetivo incorporar em um novo projeto as codificações de projetos já realizados

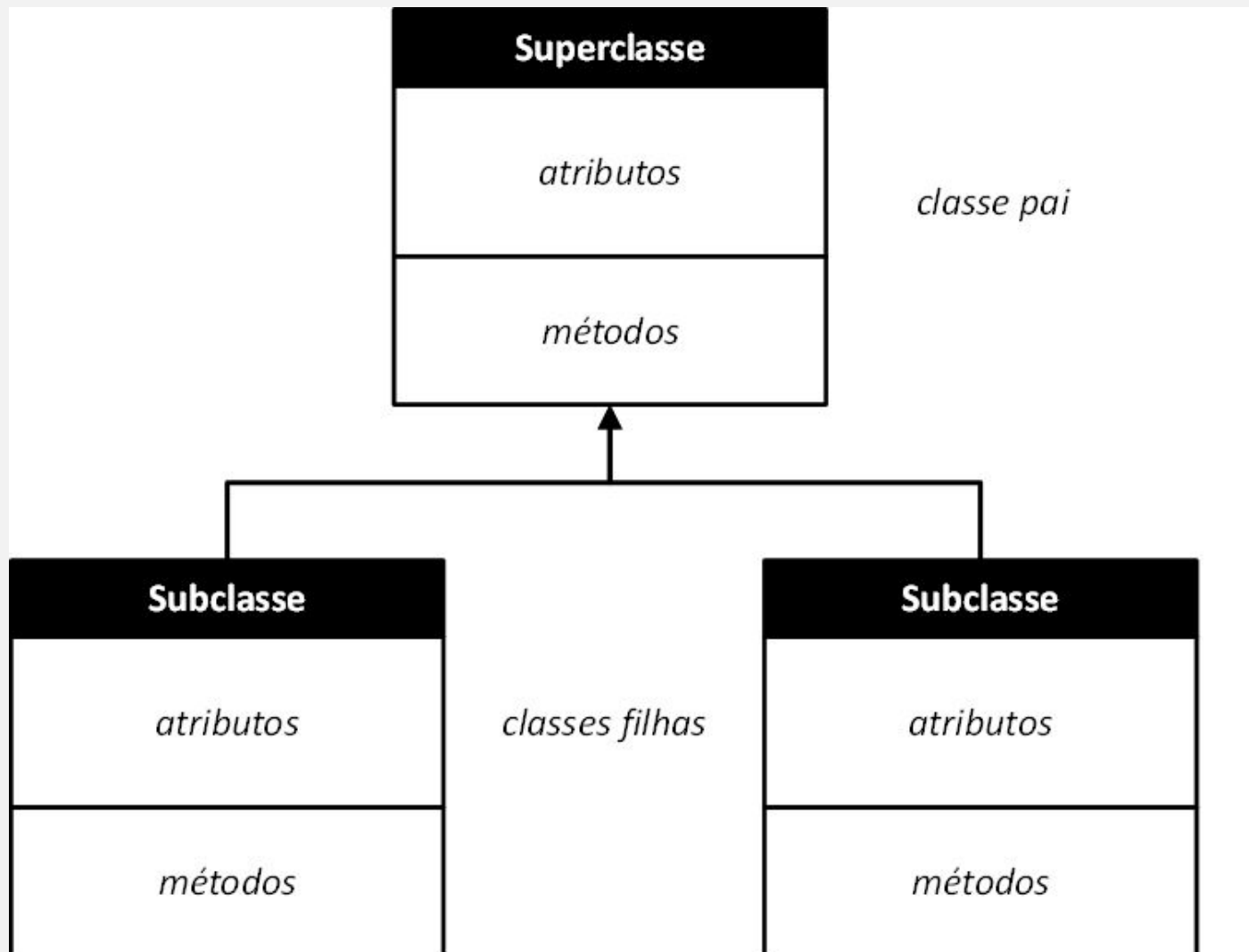
# Herança

- O paradigma de programação orientado a objetos possui uma metodologia clara e objetiva para reutilização de software.
- Essa metodologia é conhecida como herança e permite a especificação de novas classes a partir de definições de classes existentes.
- As novas classes criadas *herdam* características e funcionalidades de classes previamente especificadas

# Herança

- A técnica de herança pode ser entendida como uma relação pai e filho.
  - A classe filha herda as características e funcionalidades da classe pai.
- Outra nomenclatura comumente utilizada é a relação superclasse e subclasse.
  - A subclasse estende os recursos da superclasse.
- Algumas linguagens de programação também utilizando da terminologia classe base e classe derivada para determinar a herança.

# Herança



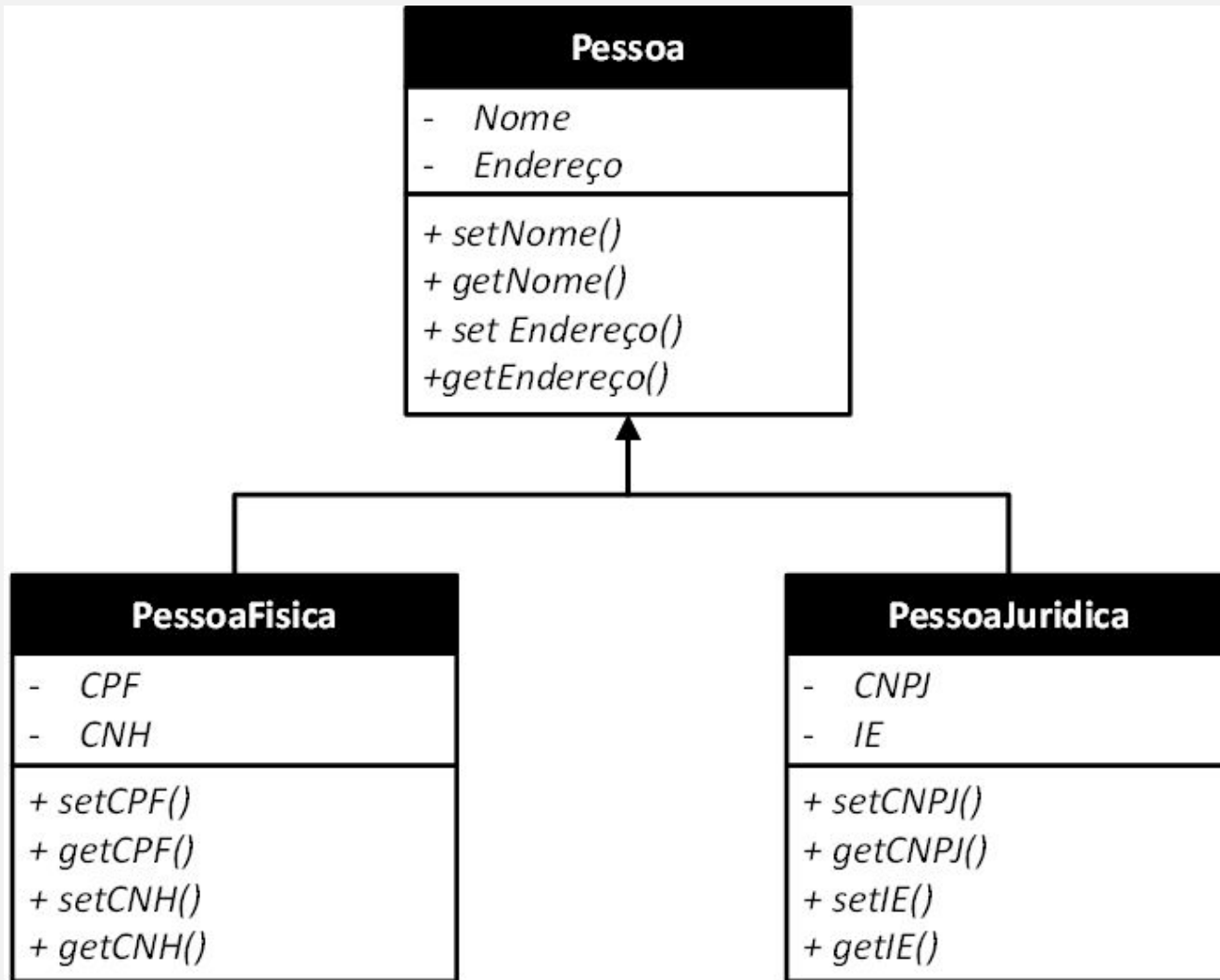
# Herança

- Tipos de Herança
  - ***Herança simples***: quando uma classe derivada, ou classe filha, estende características e funcionalidades de um único pai.
  - ***Herança múltipla***: ocorre quando uma subclasse herda recursos originários de mais de uma superclasse.

# Herança

- Na linguagem de programação Java a relação de herança entre duas classes é definida por meio da palavra reservada ***extends***.
- Assim, durante a especificação de uma classe filha (ou subclasse) é necessário indicar que essa nova classe estende de uma classe pai (ou superclasse).

# Herança



```
public class PessoaFisica extends Pessoa{

    private String cpf;
    private String cnh;

    public PessoaFisica() {
        super();    //chamada construtor classe pai
        this.cpf = "";
        this.cnh = "";
    }

    ...
}
```

```
public class PessoaJuridica extends Pessoa {

    private String cnpj;
    private String ie;

    public PessoaJuridica() {
        super();
        this.cnpj = "";
        this.ie = "";
    }

    ...
}
```



# Herança

```
//Objeto Pessoa Física
```

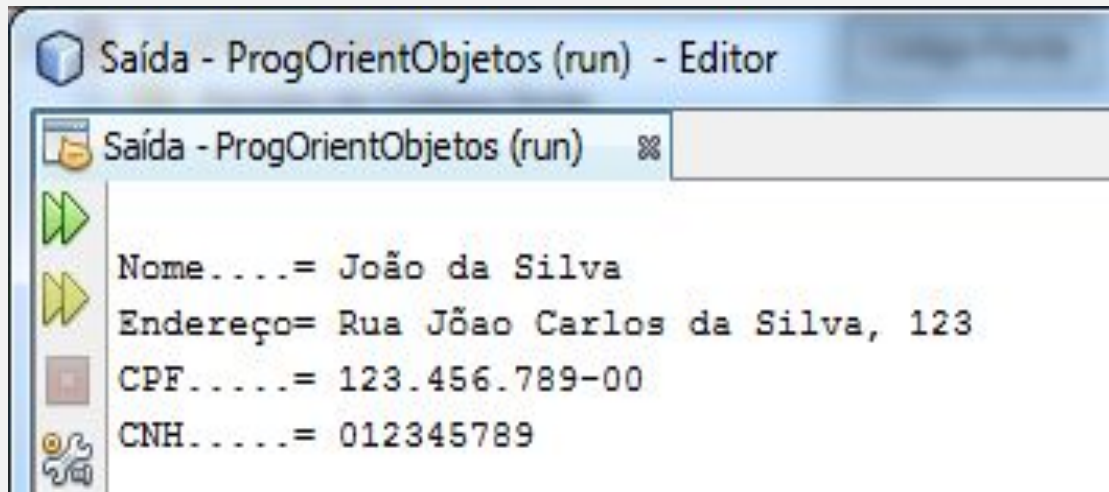
```
PessoaFisica pf = new PessoaFisica();
```

```
pf.setNome("João da Silva");
```

```
pf.setEndereco("Rua João Carlos da Silva, 123");
```

```
pf.setCpf("123.456.789-00");
```

```
pf.setCnh("012345789");
```



# **ATIVIDADE PRÁTICA**

- **Exercício 1:** Desenvolver um sistema de gerenciamento de veículos para uma concessionária.
  - Crie uma classe base chamada **Veiculo** com os seguintes atributos: marca (string), modelo (string) e ano (int).
  - Crie uma classe **Carro**, que herda da classe **Veiculo**, com um atributo adicional: numPortas (int): representando o número de portas do carro.
  - Crie uma classe **Moto**, que também herda da classe **Veiculo**, com um atributo adicional: cilindradas (double): representando a capacidade em cilindradas do motor da moto.
- Por fim, escreva a classe **VeiculoTest** para demonstrar o funcionamento da solução.

**POLIMORFISMO**

# Polimorfismo

- O termo polimorfismo pode ser traduzido literalmente como algo contendo muitas formas.
- Em programação de computadores, o conceito de polimorfismo é utilizado para referenciar objetos que contém características distintas para cada filho.

# Polimorfismo

- Tipos de Polimorfismo
  - Polimorfismo por Sobrecarga
  - Polimorfismo por Sobreposição
  - Polimorfismo por Generalização

# Polimorfismo

- Polimorfismo por Sobrecarga
  - Tipo mais comum de especificação de polimorfismo.
  - Tem como principais características a definição de métodos com o mesmo nome, porém com listas de parâmetros diferentes.
  - A diferença entre os métodos é identificada em tempo de execução e, é realizada pela alteração dos nomes e dos tipos definidos na lista de parâmetros.

# Polimorfismo por Sobrecarga

```
public class Soma {  
  
    public int getSoma(int valor1, int valor2){  
        return (valor1+valor2);  
    }  
  
    public int getSoma(int valor1, int valor2, int valor3){  
        return (valor1+valor2+valor3);  
    }  
  
    public double getSoma(double valor1, double valor2){  
        return (valor1+valor2);  
    }  
  
    public double getSoma(double valor1, double valor2, double valor3){  
        return (valor1+valor2+valor3);  
    }  
  
    public String getSoma(String valor1, String valor2){  
        double v1 = Double.parseDouble(valor1);  
        double v2 = Double.parseDouble(valor2);  
        return String.valueOf(v1+v2);  
    }  
}
```



# Polimorfismo

- Polimorfismo por Sobreposição
  - Tem como objetivo especificar métodos que contêm nomes e lista de parâmetros comuns, porém tenham funcionalidades diferentes.
  - É possível definir métodos com funcionalidades diferentes nas classes derivadas.

# Polimorfismo

- Polimorfismo por Sobreposição
  - um método definido em uma superclasse pode ser reescrito, ou sobreposto, pelas classes derivadas de forma que cada classe derivada possa realizar uma especificação diferente.
  - Em termos de programação, este tipo de polimorfismo é identificado pela notação ***@Override***.

# Polimorfismo por Sobreposição

```
public double getMedia(){  
    return (this.nota1+this.nota2)/2;  
}
```

**Superclasse**

```
@Override  
public String toString() {  
    return "Média= "+String.format("%.1f", getMedia());  
}
```

**Subclasse**

```
@Override  
public double getMedia(){  
    return (super.getNota1()*0.40  
            +super.getNota2()*0.60);  
}
```

```
@Override  
public String toString() {  
    return "\nAluno de Graduação \n" +  
            super.toString();  
}
```

# Polimorfismo

- Polimorfismo por Generalização
  - A representação de objetos com muitas formas é realizada por meio da coerção entre objetos.
  - Um objeto pode assumir formas diferentes da sua origem a partir da sua coerção.
  - Na programação de computadores, o termo coerção é comumente conhecido como ***casting***.

# Polimorfismo por Generalização

```
Aluno[] aluno = new Aluno[5];  
  
aluno[0] = new AlunoGraduacao(8.50, 7.80);  
aluno[1] = new AlunoPosGraduacao(9.50,6.90,8.40,8.80);  
aluno[2] = new AlunoPosGraduacao(8.20,9.40,6.50,7.30);  
aluno[3] = new AlunoGraduacao(6.20, 9.90);  
aluno[4] = new AlunoGraduacao(9.50, 5.90);
```

## Casting

```
((AlunoPosGraduacao)aluno[1]).setNotaMonografia(9.50);  
((AlunoPosGraduacao)aluno[1]).setNotaProjeto(9.90);  
  
((AlunoPosGraduacao)aluno[2]).setNotaMonografia(8.80);  
((AlunoPosGraduacao)aluno[2]).setNotaProjeto(9.20);
```

# **ATIVIDADE PRÁTICA**

- **Exercício 1:** Desenvolver um sistema de gerenciamento de funcionários para uma empresa.

- **Classe Funcionario**

- nome (string): representando o nome do funcionário.
- salario\_base (double): representando o salário base do funcionário.
- Além disso, implemente um método chamado `calcular_salario()` que retorna o salário total do funcionário.

- **Classe FuncionarioIntegral**

- bonus (double): representando o bônus adicional que o funcionário em período integral recebe.
- Sobrescreva o método `calcular_salario()` para incluir o bônus no cálculo do salário total.

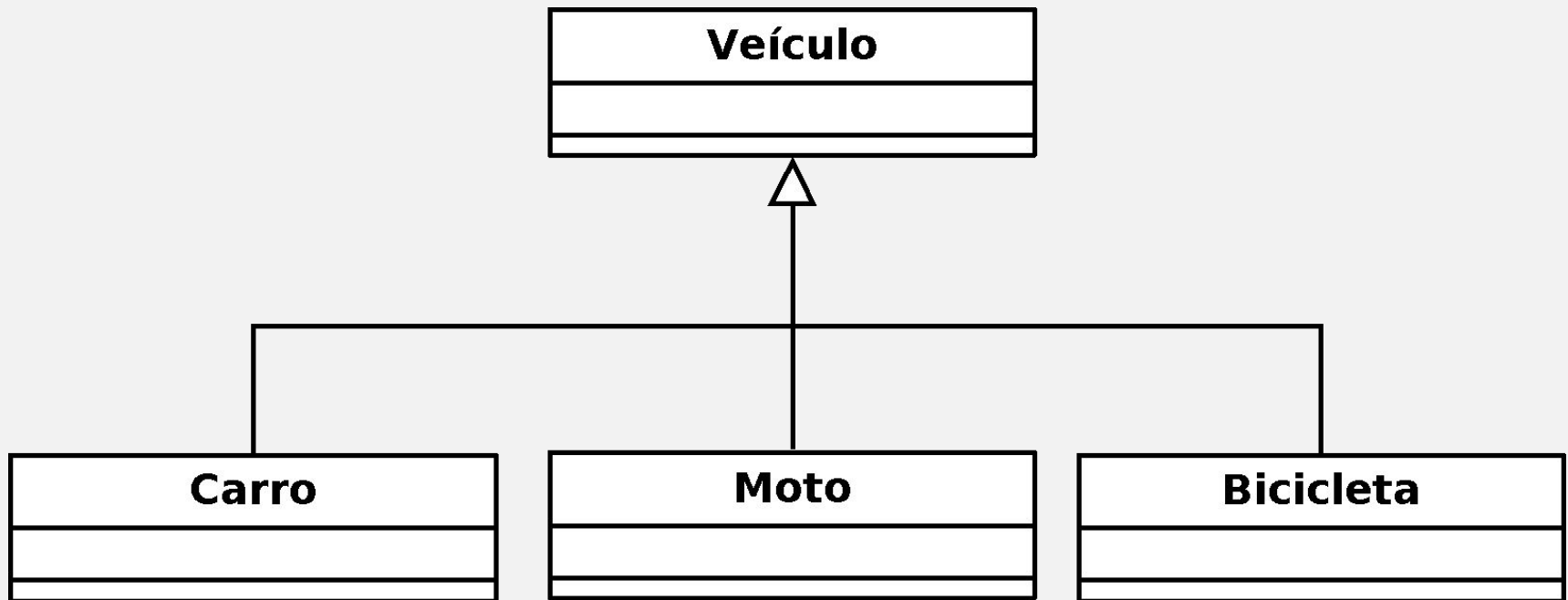
- **Classe FuncionarioMeioPeriodo**

- horas\_trabalhadas (int): representando o número de horas trabalhadas pelo funcionário em meio período.
- Sobrescreva o método `calcular_salario()` para calcular o salário total com base nas horas trabalhadas multiplicadas pelo salário por hora.

# Atividade Prática

## ▪ *Exercício 2*

- Considere a seguinte hierarquia de classes:





# Atividade Prática

## ▪ ***Exercício 2 (continuação)***

- Especifique as classes e seus respectivos atributos. Cada classe deverá conter pelo menos dois atributos.
- Na classe Carro crie um método denominado *velocidadeMaxima* capaz de indicar se o veículo atingiu a velocidade máxima permitida.
- Demonstre a utilização do polimorfismo por generalização.

- **Exercício 3:** Desenvolver um sistema para uma livraria online que vende livros físicos e ebooks.
  - **Classe base chamada Livro:**
    - Atributo precoBase do tipo double para representar o preço base do livro.
    - Um construtor que inicializa o preço base do livro.
    - Um método calcularPreco() que retorna o preço do livro. Esse método será sobreposto pelas subclasses.
  - Crie duas subclasses de Livro: **LivroFisico** e **Ebook**.
    - LivroFisico deve ter um atributo adicional custoEnvio do tipo double para representar os custos de envio.
    - Ebook não possui atributos adicionais.
    - Implemente o método calcularPreco() nas subclasses de acordo com as seguintes especificações:
      - Para LivroFisico: o preço do livro físico deve incluir o custo de envio.
      - Para Ebook: o preço do ebook não inclui custos de envio.
  - Escreva uma classe **LivroTest** para demonstrar o polimorfismo por sobreposição. Neste método, crie pelo menos um objeto de cada tipo (livro físico e ebook) e chame o método calcularPreco() para cada objeto, mostrando o preço calculado.

- **Exercício 4:** Criar uma hierarquia de classes que representam diferentes tipos de animais e demonstrar o polimorfismo através de um programa de teste.
  - Crie uma classe base chamada **Animal**:
    - Atributo nome do tipo String para representar o nome do animal.
    - Um construtor que inicializa o nome do animal.
    - Um método emitirSom() que imprime uma mensagem genérica para representar o som do animal.
  - Crie pelo subclasses de Animais que representam tipos específicos de animais, como Cachorro, Gato, Pato, etc.
    - Cada uma dessas subclasses deve sobrescrever o método emitirSom() para representar o som específico do animal.
  - Implemente o TesteAnimais. Neste método, crie uma lista de animais que inclua objetos de diferentes tipos (cachorro, gato, pato, etc.). Em seguida, percorra a lista e chame o método emitirSom() para cada animal, observando como o método correto é chamado de acordo com o tipo real do animal na lista.

# **CLASSES E MÉTODOS ABSTRATOS**

# Classes e Métodos Abstratos

- A técnica de classes abstratas permite ao programador declarar classes que nunca serão instanciadas.
- Uma classe abstrata pode ser utilizada apenas em situações de herança e, não permitem que objetos sejam diretamente criados.

# Classes e Métodos Abstratos

- Uma classe abstrata é utilizada para oferecer as classes derivadas uma superclasse com total integridade em relação aos seus atributos e métodos.
- As classes filhas podem acessar as propriedades da classe pai de maneira apropriada.
- A especificação de uma classe abstrata é realizada por meio da palavra reservada ***abstract***

# Classes e Métodos Abstratos

- Uma classe abstrata pode conter métodos que serão utilizados como base para as classes derivadas.
- Estes métodos são conhecidos como ***métodos abstratos***.
- Um método abstrato não possui implementação, assim, é especificado apenas a assinatura do método abstrato.

# Classes e Métodos Abstratos

Superclasse

```
public abstract class Funcionario {  
  
    private String nome;  
    protected double salario;  
  
    public Funcionario() { ... }  
  
    public Funcionario(String nome, double salario) { ... }  
  
    public abstract double getSalario();  
  
    @Override  
    public String toString() {  
        return "\nNome...= " + nome  
            + "\nSalário= "  
            + String.format("%.2f",getSalario());  
    }  
}
```



# Classes e Métodos Abstratos

```
public class Vendedor extends Funcionario {  
  
    private double totalvendas;  
  
    public Vendedor() {...}  
  
    public Vendedor(...) {... }  
  
    @Override  
    public double getSalario() {  
        return super.salario +  
            (this.totalvendas*0.08);  
    }  
  
}
```

**Subclasse**

# Classes e Métodos Abstratos

```
public class Gerente extends Funcionario{  
  
    private double volumevendas  
  
    public Gerente() {...}  
  
    public Gerente(...) {... }  
  
    @Override  
    public double getSalario() {  
        return super.salario + (this.getVolumevendas()*0.01);  
    }  
  
}
```

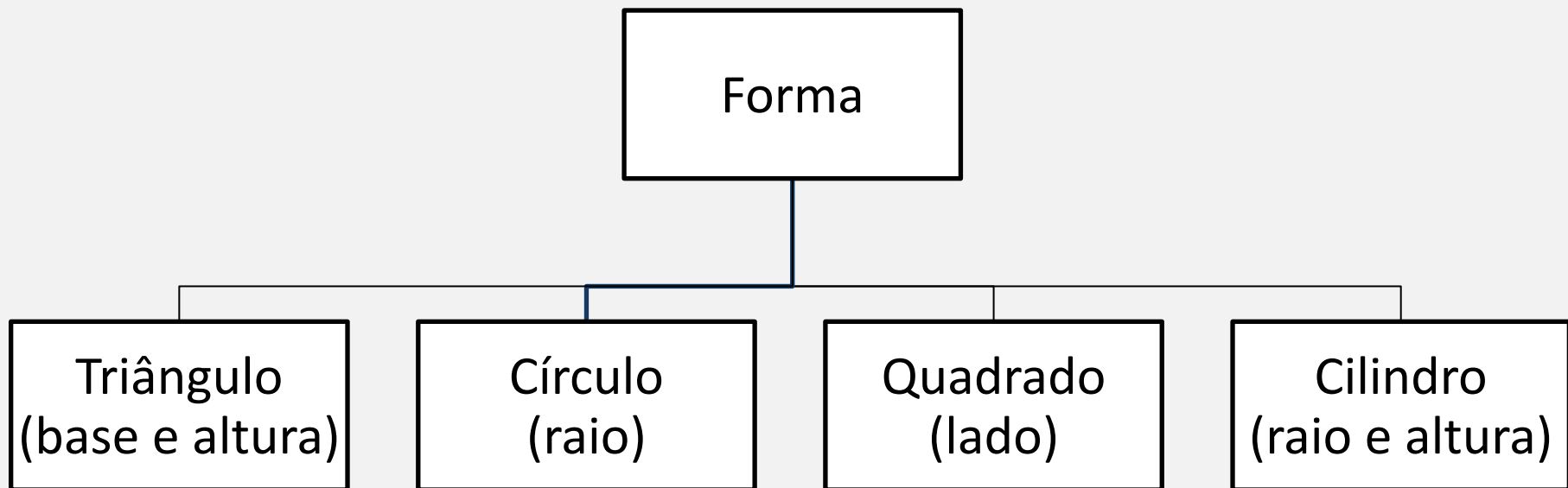
**Subclasse**

# **ATIVIDADE PRÁTICA**

# Atividade Prática

## ■ **Exercício 1**

- Utilize o Paradigma Orientado a Objetos para codificar o seguinte diagrama:



# Atividade Prática

## ▪ **Exercício 1 (continuação)**

- Para isso, considere a seguinte classe abstrata **Forma**:

```
public abstract class Forma{

    public abstract String getNome();
    public abstract double getArea();

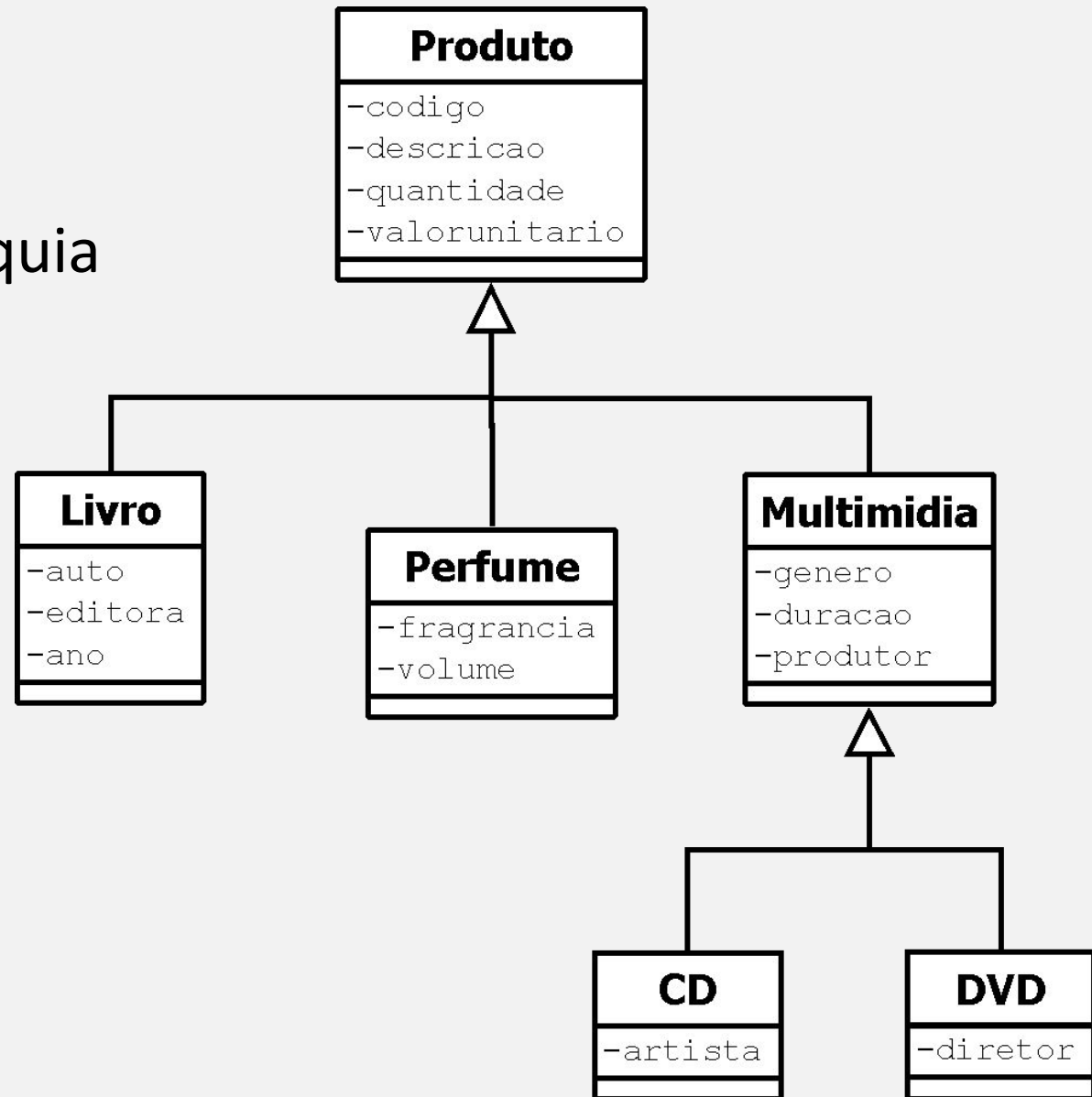
    @Override
    public String toString(){
        return "Forma = " + getNome() + "\n"+
               "Área  = " + String.format("%.2f",getArea()) + "\n";
    }
}
```

- No final, crie uma classe de testes capaz de instanciar quatro formas diferentes utilizando polimorfismo por generalização. Apresente os resultados na tela.

# Atividade Prática

## ■ *Exercício 2*

- Especifique a seguinte hierarquia de classes



# Atividade Prática

## ▪ ***Exercício 2 (continuação)***

- Considerando o diagrama especifique as classes utilizando POO. Cada classe deve conter:
  - Atributos encapsulados
  - Construtor Padrão
  - Construtor Sobrecarregado
  - Método GET/SET para todos os atributos
  - Método *toString*

# Atividade Prática

## ▪ ***Exercício 2 (continuação)***

- Para especificação, identifique a possibilidade de utilizar os conceitos de:
  - Polimorfismo:
    - sobrecarga, sobreposição e generalização
  - Classe abstrata
  - Métodos abstratos



# **INTERFACES E HERANÇA MÚLTIPLA**

# Interfaces e Herança Múltipla

- O conceito de herança pode ser definido de duas maneiras denominadas:
  - herança simples
  - herança múltipla.
- Algumas linguagens de programação orientadas a objetos não implementam nativamente a herança múltipla
- Para essas linguagens, a herança múltipla pode ser definida utilizando um recurso denominado interfaces.

# Interfaces e Herança Múltipla

## ■ Interfaces

- São recursos das linguagens de programação que permitem a especificação de métodos que serão definidos por outras classes.
- Permitem a definição de assinaturas de métodos que serão codificados por classes que implementam essas interfaces.
- Na linguagem de programação Java este procedimento é indicado no início da definição da classe com a palavra reservada ***implements***

# Interfaces e Herança Múltipla

```
public interface ICalculadora {  
  
    public abstract double getSoma();  
    public abstract double getSubtracao();  
    public abstract double getDivisao();  
    public abstract double getMultiplicacao();  
    public abstract void exibirResultados();  
  
}
```

# Interfaces e Herança Múltipla

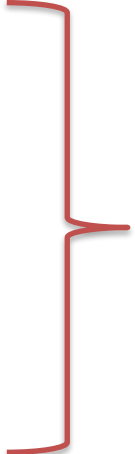
```
public class Calculadora implements ICalculadora{

    private double valor1;
    private double valor2;

    public Calculadora() {...}
    public Calculadora(double valor1, double valor2) {...}

    @Override
    public double getSoma() {
        return this.valor1+this.valor2;
    }

    @Override
    public double getSubtracao() {
        return this.valor1-this.valor2;
    }
}
```



*Codificação obrigatória  
dos métodos definidos  
na interface*

# Interfaces e Herança Múltipla

```
@Override  
public double getDivisao() {  
    return this.valor1/this.valor2;  
}
```

```
@Override  
public double getMultiplicacao() {  
    return this.valor1*this.valor2;  
}
```

```
@Override  
public void exibirResultados() {  
    System.out.println("Resultados");  
    System.out.println("Soma.....= " + getSoma());  
    System.out.println("Subtração....= " + getSubtracao());  
    System.out.println("Divisão.....= " + getDivisao());  
    System.out.println("Multiplicação= " + getMultiplicacao());  
}
```

*Codificação obrigatória  
dos métodos definidos  
na interface*

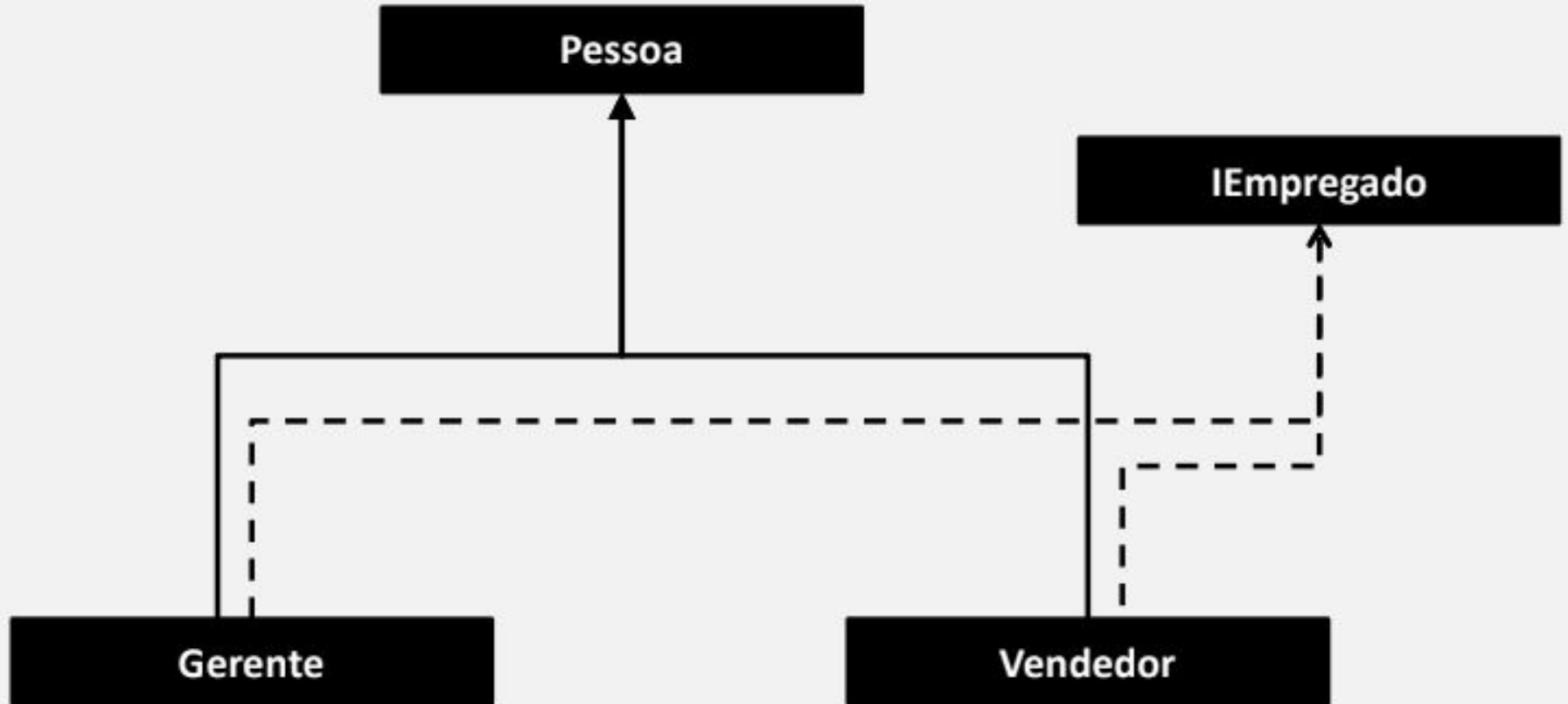
```
}
```

# Interfaces e Herança Múltipla

- Herança Múltipla

- Representa a especificação de subclasses a partir da herança de múltiplas superclasses.
- Na linguagem de programação Java este recurso é implementado utilizando interfaces.

# Interfaces e Herança Múltipla





# Interfaces e Herança Múltipla

```
public class Pessoa {  
    ...  
}  
  
public interface IEmpregado {  
    ...  
}  
  
public class Gerente extends Pessoa implements IEmpregado{  
    ...  
}  
  
public class Vendedor extends Pessoa implements IEmpregado{  
    ...  
}
```

# **CONSIDERAÇÕES FINAIS**

# Considerações Finais

- Paradigma de Programação Orientado a Objetos
  - Metodologia profissional para o desenvolvimento de software
  - Construção de aplicações baseadas em um modelo formal
  - Qualidade na codificação dos projetos
  - Reutilização de software

**FIM**