

# Programação Orientada a Objetos

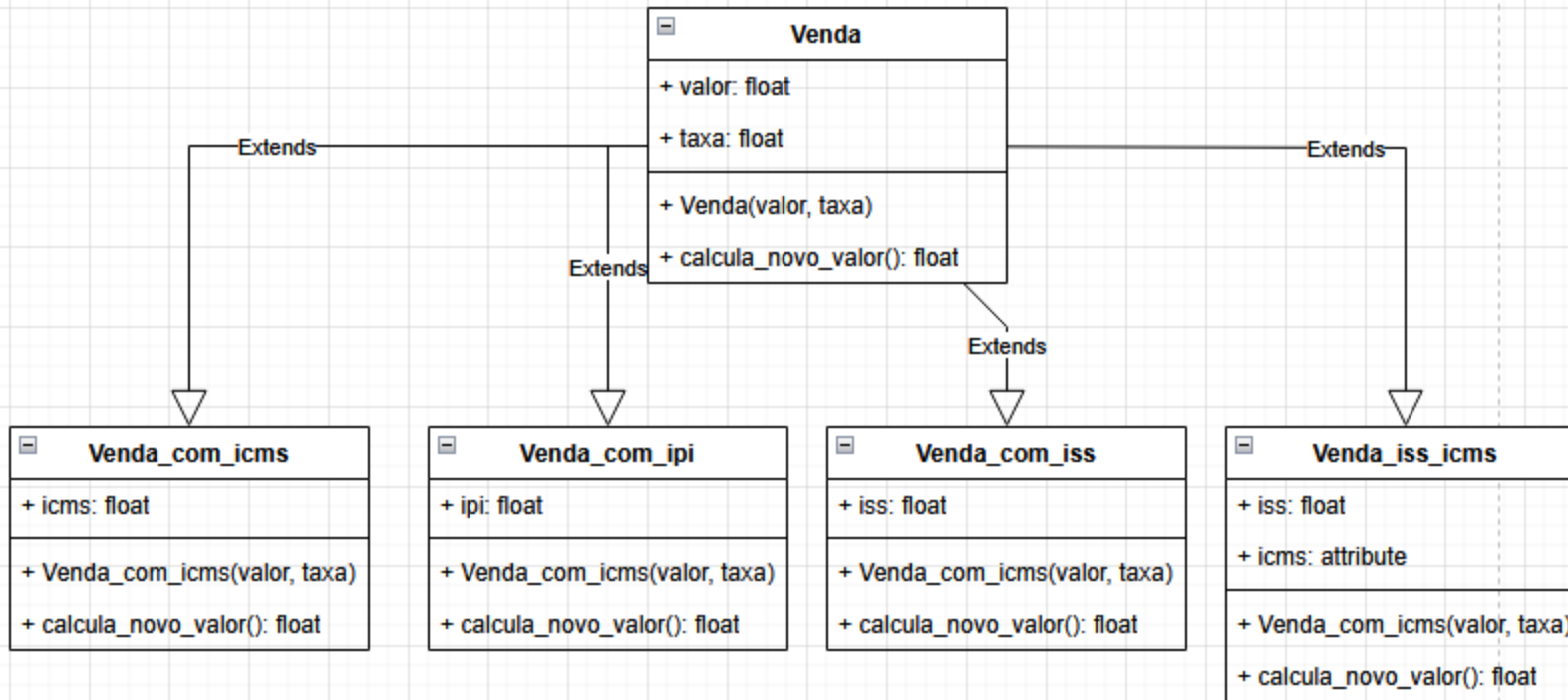
Aula 6 – Professor Diogo Orlando Nunes de Almeida

# Conteúdo

- Conceitos do paradigma da Programação orientada a objetos;
- Diagrama de classes (UML);
- Introdução a classes e objetos;
- Atributos, métodos e interação entre objetos;
- Construtores e destrutores;
- Agregação e composição de objetos;
- Encapsulamento;
- Herança e polimorfismo;
- Tratamento de Exceções.

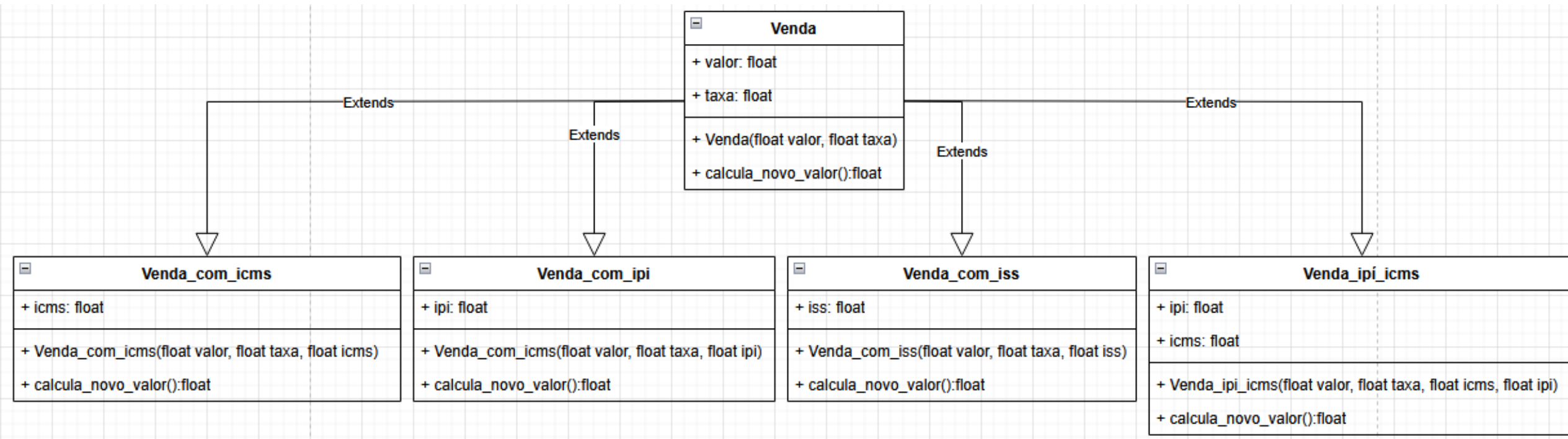
# Polimorfismo

- Vem do grego e quer dizer várias formas.
- Parte da sobrescrita de métodos.



# Polimorfismo (Override)

- Analise que a venda é realizada de diversas formas, e a o método da classe pai vai ter que ser sobrescrito para poder manter a lógica do programa.



# Polimorfismo (Override)

```
public class Venda {  
    public float valor;  
    public float taxa; //de_lucro  
  
    public Venda (float valor, float taxa){  
        this.valor = valor;  
        this.taxa = taxa;  
    }  
  
    public float calcula_novo_valor(){  
        return this.valor + (this.valor * this.taxa);  
    }  
}
```

Venda.java

```
public class Venda_com_ipi extends Venda {  
    public float ipi;  
  
    public Venda_com_ipi(float valor, float taxa, float ipi){  
        super(valor, taxa);  
        this.ipi = ipi;  
    }  
  
    @Override  
    public float calcula_novo_valor(){  
        return this.valor + (this.valor * this.ipi) + (this.valor * this.taxa);  
    }  
}
```

Venda\_com\_ipi.java

```
public class Venda_com_icms extends Venda {  
    public float icms;  
  
    public Venda_com_icms(float valor, float taxa, float icms){  
        super(valor, taxa);  
        this.icms = icms;  
    }  
  
    @Override  
    public float calcula_novo_valor(){  
        return this.valor + (this.valor * this.icms) + (this.valor * this.taxa);  
    }  
}
```

Venda\_com\_icms.java

```
public class Venda_com_iss extends Venda {  
    public float iss;  
  
    public Venda_com_iss(float valor, float taxa, float iss){  
        super(valor, taxa);  
        this.iss = iss;  
    }  
  
    @Override  
    public float calcula_novo_valor(){  
        return this.valor + (this.valor * this.iss) + (this.valor * this.taxa);  
    }  
}
```

Venda\_com\_iss.java

# Polimorfismo (Override)

```
public class Venda_ipi_icms extends Venda{
    public float ipi;
    public float icms;

    public Venda_ipi_icms(float valor, float taxa, float ipi, float icms){
        super(valor, taxa);
        this.ipi = ipi;
        this.icms = icms;
    }

    @Override
    public float calcula_novo_valor(){
        return this.valor + (this.valor * this.ipi) + (this.valor * this.icms) + (this.valor * this.taxa);
    }
}
```

Venda\_ipi\_icms.java

```
public class Main {
    Run | Debug
    public static void main(String[] args) {
        Venda produtoA = new Venda(valor:15.00f, taxa:0.10f);
        System.out.println(produtoA.calcula_novo_valor());
        Venda_com_icms produtoB = new Venda_com_icms(valor:15.00f, taxa:0.10f, icms:0.05f);
        System.out.println(produtoB.calcula_novo_valor());
        Venda_ipi_icms produtoC = new Venda_ipi_icms(valor:15.00f, taxa:0.10f, ipi:0.025f, icms:0.05f);
        System.out.println(produtoC.calcula_novo_valor());
    }
}
```

16.5  
17.25  
17.625

Resultado

Main.java

# Polimorfismo (Override)

- Veja que a superclasse Venda.java tem o método `calcula_novo_valor` e sua implementação.
- Porém todas as classes filhas (subclasses) também tiveram a mesma implementação do método `calcula_novo_valor` de formas diferentes usando `@Override` como forma de sinalizar o polimorfismo criado pela herança.
- Cada classe terá sua implementação sem interferir uma na outra.

# Polimorfismo (Overload)

- O método `calcula_novo_valor` pode sofrer polimorfismo dentro da mesma classe.
- Para que isso ocorra os parâmetros do método devem ser diferentes ou com uma outra quantidade.
- O construtor pode sofrer polimorfismo.

```
public class Venda_overload {
    public float valor;
    public float taxa; //de_lucro
    public float ipi;
    public float icms;

    public Venda_overload(float valor, float taxa, float ipi, float icms){
        this.valor = valor;
        this.taxa = taxa;
        this.ipi = ipi;
        this.icms = icms;
    }

    public Venda_overload(float valor, float taxa){ //Overload construtor
        this.valor = valor;
        this.taxa = taxa;
        this.ipi = 0;
        this.icms = 0;
    }

    public float calcula_novo_valor(){
        return this.valor + (this.valor * this.taxa);
    }

    public float calcula_novo_valor(float ipi){ //Overload calcula_novo_valor
        this.ipi = ipi;
        return this.valor + (this.valor * this.ipi) + (this.valor * this.taxa);
    }

    public float calcula_novo_valor(float ipi, float icms){ //Overload calcula_novo_valor
        this.ipi = ipi;
        this.icms = icms;
        return this.valor + (this.valor * this.ipi) + (this.valor * this.icms) + (this.valor * this.taxa);
    }
}
```

# Polimorfismo overload

```
public class Main {  
    Run | Debug  
    public static void main(String[] args) {  
        Venda_overload produtoA = new Venda_overload(valor:15.00f, taxa:0.10f);  
        System.out.println(produtoA.calcula_novo_valor());  
        System.out.println(produtoA.calcula_novo_valor(ipi:0.05f));  
        System.out.println(produtoA.calcula_novo_valor(ipi:0.10f, icms:0.15f));  
        Venda_overload produtoB = new Venda_overload(valor:20.00f, taxa:0.20f, ipi:0.05f, icms:0.15f);  
        System.out.println(produtoB.calcula_novo_valor());  
        System.out.println(produtoB.calcula_novo_valor(ipi:0.05f));  
        System.out.println(produtoB.calcula_novo_valor(ipi:0.10f, icms:0.15f));  
    }  
}
```

16.5

17.25

20.25

24.0

25.0

29.0

# Método abstrato

- O método abstrato é um método que não há nenhuma implementação, porém obriga que as classes filhas tenham aquela classe.

```
public class Venda {  
    public float valor;  
    public float taxa; //de_lucro  
  
    public Venda (float valor, float taxa){  
        this.valor = valor;  
        this.taxa = taxa;  
    }  
  
    public float calcula_novo_valor; //Classe abstrata  
}
```

```
public class Venda {  
    public float valor;  
    public float taxa; //de_lucro  
  
    public Venda (float valor, float taxa){  
        this.valor = valor;  
        this.taxa = taxa;  
    }  
  
    public float calcula_novo_valor(){  
        return this.valor + (this.valor * this.taxa);  
    }  
}
```

# Interface

- É uma classe que define as regras para suas classes filhas, do que ela precisa ter. A interface **não** deve ter construtor.

```
public interface FormaGeometrica {  
    double calculaArea();  
    double calculaPerimetro();  
}  
  
public class Retangulo implements FormaGeometrica{  
    private double largura;  
    private double altura;  
  
    public Retangulo(double largura, double altura){  
        this.largura = largura;  
        this.altura = altura;  
    }  
  
    @Override  
    public double calculaArea(){  
        return largura * altura;  
    }  
  
    @Override  
    public double calculaPerimetro(){  
        return 2 * (largura + altura);  
    }  
  
    public void quadrado(){  
        if(largura == altura){  
            System.out.println(x:"É quadrado");  
        }  
    }  
}  
  
public class Triangulo_equilatero implements FormaGeometrica{  
    private double base;  
    private double altura;  
  
    public Triangulo_equilatero(double base, double altura){  
        this.base = base;  
        this.altura = altura;  
    }  
  
    @Override  
    public double calculaArea(){  
        return 0.5 * base * altura;  
    }  
  
    @Override  
    public double calculaPerimetro(){  
        double lado = base;  
        return 3 * lado;  
    }  
}
```

# Interfaces

```
public class Main {  
    Run | Debug  
    public static void main(String[] args) {  
        Triangulo_equilatero t1 = new Triangulo_equilatero(base:10, altura:10);  
        System.out.println(t1.calculaArea());  
        Retangulo q1 = new Retangulo(largura:10, altura:10);  
        q1.quadrado();  
    }  
}
```

50.0

É quadrado

# Tratamento de exceções

- No java se usa o Try – catch. No código abaixo iremos verificar se o usuário vai digitar um número, caso seja inválido retornará erro.

```
import java.util.Scanner;

public class Main {
    Run | Debug
    public static void main(String[] args) {
        int numero;
        System.out.println(x:"Digite um número!");
        Scanner teclado1 = new Scanner(System.in);
        try{
            numero = teclado1.nextInt();
        }catch(Exception e){
            System.out.println(x:"Erro! Digite um número!");
        }
    }
}
```

Digite um número!

a

Erro! Digite um número!

# Tratamento de exceções

```
public class IMC {  
    private double peso;  
    private double altura;  
  
    public IMC(double peso, double altura){  
        if (peso<0){  
            throw new IllegalArgumentException(s:"Peso menor que zero!");  
        }  
        if (altura<0){  
            throw new IllegalArgumentException(s:"Altura menor que zero!");  
        }  
        this.peso = peso;  
        this.altura = altura;  
    }  
  
    public double calcula_IMC(){  
        return peso/(altura*altura);  
    }  
}
```

- Para evitar que um objeto não seja criado em Java, deve-se gerar um erro de compilação.
- Para gerar erro de compilação use o comando `throw new IllegalArgumentException(mensagem)` para gerar um erro de compilação pelo mal uso da biblioteca.

# Tratamento de exceções

```
import java.io.*;

public class Main {
    Run | Debug
    public static void main(String[] args) {
        IMC sujeito1 = new IMC(peso:75, altura:1.68);
        System.out.println(sujeito1.calcula_IMC());
        IMC sujeito2 = new IMC(-15, -12);
    }
}
```

26.573129251700685

Exception in thread "main" java.lang.IllegalArgumentException: Peso menor que zero!  
at IMC.<init>(IMC.java:7)  
at Main.main(Main.java:7)

# Exercícios com o professor

- Aplique os conceitos dessa aula no projeto da aula passada.

# Exercícios

- Pesquise por 3 implementações de interfaces em Orientação Objetos e implemente as em Java.
- Crie 5 pequenos projetos que possam ser usados na sua vida profissional ou na área de jogos, com poucas classes usando o polimorfismo.

# Referências

- PAGE-JONES, Meilir. **Fundamentos do desenho orientado a objeto com UML**. São Paulo: Pearson, 2001. E-book. Disponível em: <https://plataforma.bvirtual.com.br>. Acesso em: 25 mar. 2025.