# Programação Orientada a Objetos em Java

Rone Ilídio da Silva
Universidade Federal de São João del-Rei
Campus Alto Paraopeba

# Objetivo

Apresentar os principais conceitos de Programção Orientada a Objeto com a linguagem de programação Java

# Apresentação

- Introdução
  - Orientação a Objeto
  - Java
- Classe
- Objeto
- Herança
- Encapsulamento
- Polimorfismo

# Orientação a Objetos

- O ser humano vêem os objetos reais através de suas características e funções

**Características**

• Tamanho

• Peso

• Cor

• Preço

**Funções**

• Liga

•Recebe ligações

•Envia Mensagens

•Recebe Mensagens

# Projetos Orientados a Objetos

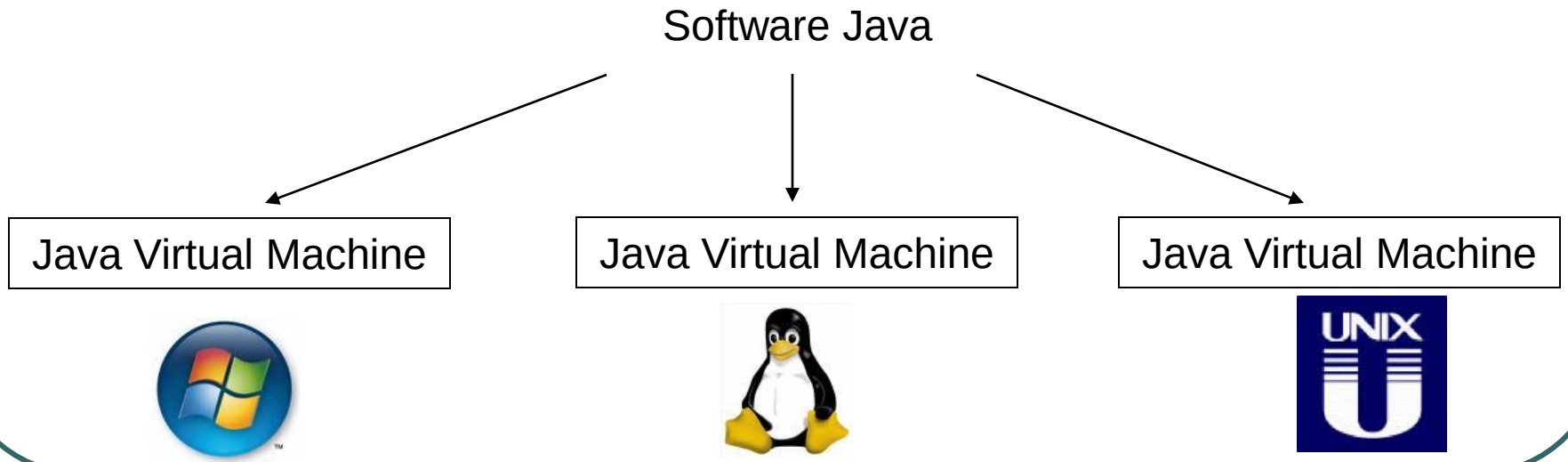
- Modelagem e desenvolvimento de softwares por objetos
- Os objetos possuem as características e funções de objetos reais
- Mais próximo do pensamento humano
- Facilita a modularização do software

# Linguagens de Programação Orientadas a Objeto

- Suporte à criação de objetos no computador
- Representação de objetos reais
- Exemplos de linguagens:
  - C++
  - C#
  - Objetct Pascal
  - Java

# Java

- Totalmente orientada a objetos
- Gratuita
- Multi plataforma

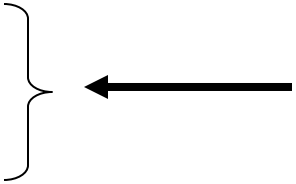


# Programação Orientada a Objetos em Java

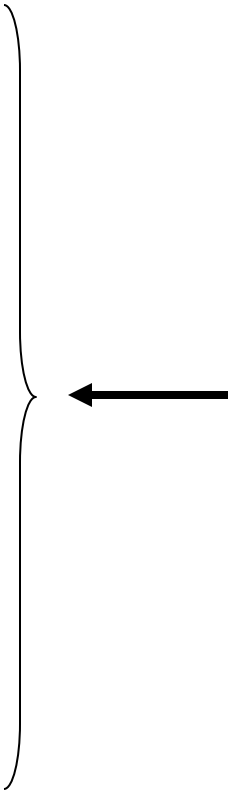
# Classe

- Tipo abstrato de dados
- Modelo ou protótipo de objetos reais
- Possui a definição das características e funções desses objetos
- Características → Atributos
  - definem o estado o objeto
- Funções → Métodos
  - Comportamento do objeto
  - Operações sobre os dados

```
public class Pessoa{

   private String nome;
   private int idade;

   public String getNome(){
         return nome;
   }
   public int getIdade(){
        return idade;
   }
   public void setNome(String n){
         nome = n;
   }
   public void setIdade(int i){
         idade = i;
   }
}
```



Atributos



Operações sobre os dados

# Classe

- Semelhante a uma struct (C) ou record (Pascal)
- Atributos: variáveis globais
- Métodos: funções que normalmente manipulam os atributos
- Métodos para leitura e escrita os atributos: interface de acesso
- Public: acesso externo
- Private: acesso interno

} Detalhes a frente

# Criação do Objeto

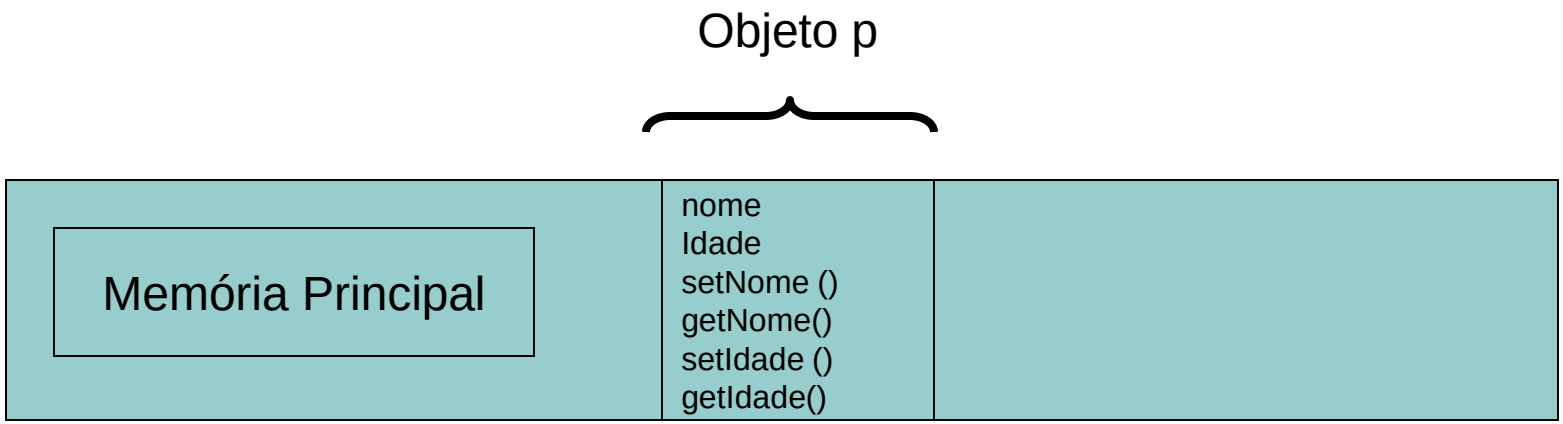
```
public class Principal{
        public static void main(String[] args){

                Pessoa p;
                p = new Pessoa();

                //p.nome = “José”;

                p.setNome(“José");
                p.setIdade(18);

                System.out.println("Nome="+ p.getNome());
                System.out.println("Idade="+ p.getIdade());
        }
}
```

# Criação do Objeto

- Foi declarada a classe Principal (aplicativo Java)
- Pessoa p;
- p = new Pessoa();
  - **new** cria uma instância de Pessoa
  - o objeto é denominado p

# Objeto

- Instância de uma classe
- Área de memória com as características da classe



# Objeto

```
public class Principal{
    public static void main(String[] args){
            Pessoa p1, p2;
            p1 = new Pessoa()
            p2 = new Pessoa();

            p1.setNome(“José");
            p1.setIdade(18);
            System.out.println("Nome="+ p1.getNome());
            System.out.println("Idade="+ p1.getIdade());

            p2.setNome(“João");
            p2.setIdade(35);
            System.out.println("Nome="+ p2.getNome());
            System.out.println("Idade="+ p2.getIdade());
    }
}
```

# Métodos Construtores

- Executados na criação dos objetos
- Possuem o mesmo nome da classe
- Não possuem tipo de retorno
- Normalmente são utilizados para inicializar atributos
- Se não forem criados o compilador cria um construtor padrão
- Chamada do construtor

```
Pessoa p = new Pessoa();
```

# Métodos Construtores

```
public class Pessoa{
      private String nome;
      private int idade;
      public Pessoa()  {     }
      public String getNome(){
            return nome;
      }
      public int getIdade(){
           return idade;
      }
      public void setNome(String n){
            nome = n;
      }
      public void setIdade(int i){
            idade = i;
      }
  }
```

- Construtor vazio
- Se não for declarado o compilador cria

# Métodos Construtores

```
public class Pessoa{
      private String nome;
      private int idade;
      public Pessoa(String n, int i)  {
            nome = n;
            idade = i;
     }
     ...
```

- O construtor recebe dois parâmetros, uma string e um inteiro
- Os valores desses parâmetros são passados para os atributos

# Métodos Construtores

- Na chamada do construtor é obrigatória a passagem dos parâmetros

```
public class Principal{
    public static void main(String[] args){

      Pessoa p;
      p = new Pessoa(“José” , 18);

      System.out.println("Nome="+ p.getNome());
      System.out.println("Idade="+ p.getIdade());
    }
}
```

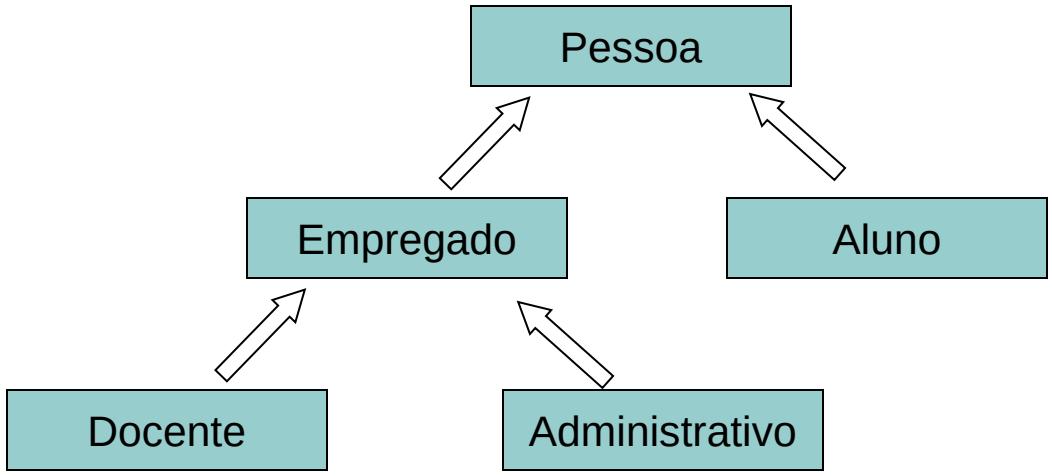
# Herança

"Forma de reutilização de código onde uma classe é criada absorvendo membros de uma classe existente e aprimorada com capacidades novas ou modificadas"

Deitel [2006]

# Herança

- A classe já existentes é chamada superclasse ou classe mãe
- A classe derivada é chamada subclasse ou classe filha
- Relacionamento “é um”



# Herança

- Inicialmente, considere a classe Pessoa sem construtor ou com construtor vazio

```
public class Pessoa{
    private String nome;
    private int idade;
    public Pessoa()  {     }
    . . .
}
```

# Herança

```
public class Aluno extends Pessoa{
    private String matricula;

    public String getMatricula() {
        return matricula;
    }

    public void setMatricula(String matricula) {
        this.matricula = matricula;
    }
}
```

# Herança

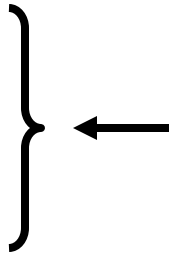
```
public class Principal {
    public static void main(String[] args) {
        Aluno a = new Aluno();

        a.setMatricula("11111");
        a.setIdade(18);
        a.setNome("Maria");

        System.out.println("Nome: "+ a.getNome());
        System.out.println("Idade: "+ a.getIdade());
        System.out.println("Matricula : "+ a.getMatricula());
    }
}
```

# Herança e Método Construtor

- Se a superclasse possui a definição de um método construtor suas filhas devem:
  - Também possuir a definição de um construtor
  - Chamar o construtor da mãe na primeira linha de seu construtor
- Utilização da palavra reservada “super”

# Herança e Construtores

```
public class Pessoa{
      private String nome;
      private int idade;
      public Pessoa(String n, int i) {
            nome = n;
            idade = i;
      }

      public String getNome(){return nome;}
      public int getIdade(){return idade;}
      public void setNome(String n){nome = n;}
      public void setIdade(int i){idade = i;}
   }
```



Definição do método construtor

# Herança e Construtores

```
public class Aluno extends Pessoa{
    private String matricula;
    public Aluno(String nome, int idade, String matricula){
        super(nome,idade);
        this.matricula = matricula;
    }
    public String getMatricula() {
        return matricula;
    }
    public void setMatricula(String matricula) {
            this.matricula = matricula;
        }
    }
```

Chama o construtor da mãe

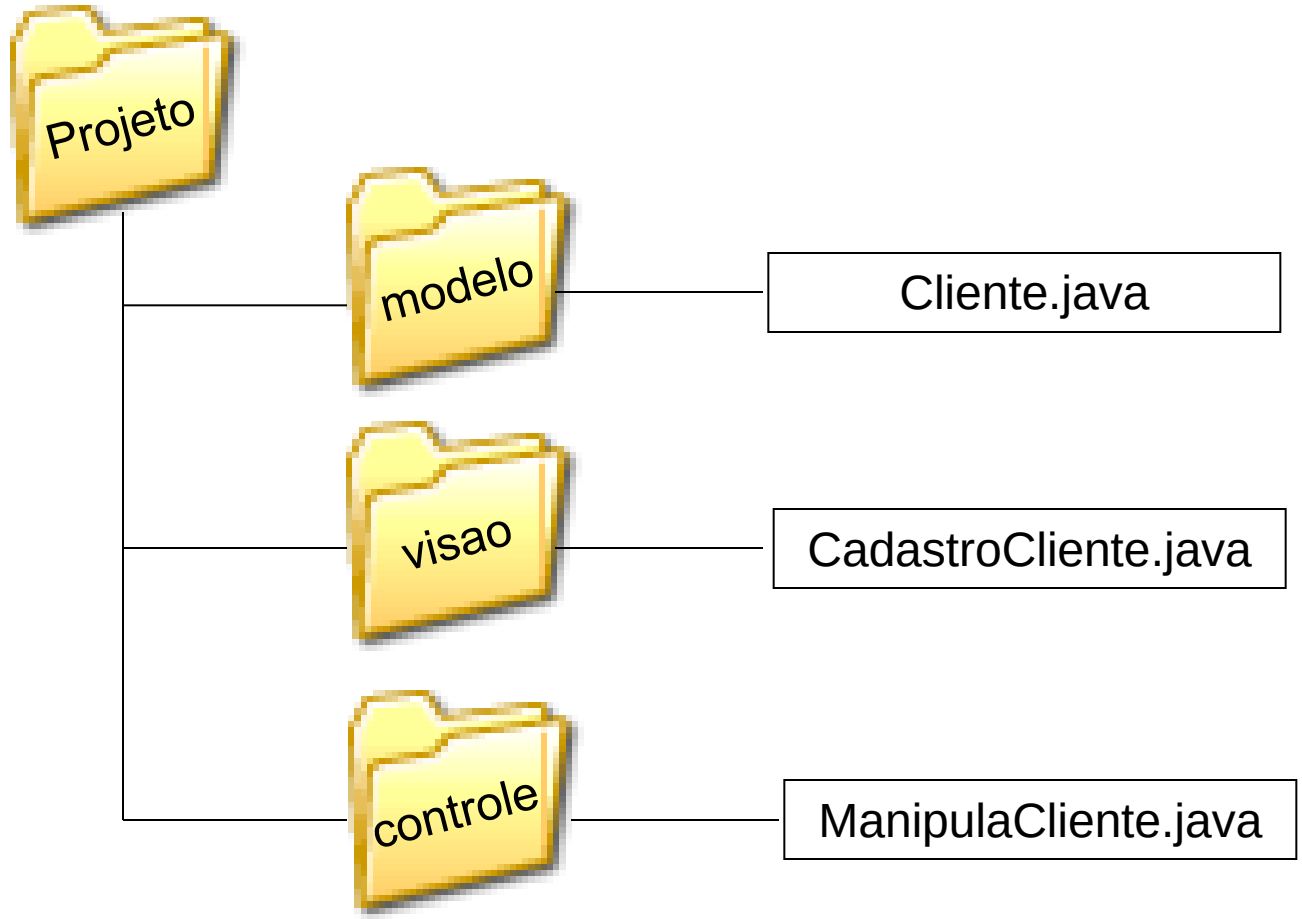
# Pacotes

- Um pacote é um conjunto de classe relacionadas
- O código de classes de mesmo pacote encontram-se na mesma pasta
- Palavras reservadas:
  - *package :* define o nome de um pacote
  - import : informa a utilização de um pacote
- Organizam o código e facilitam a reutilização de código

# Pacotes

- Importante
  - Só pode existir uma instrução *package* em um arquivo de código-fonte Java
  - Essa instrução deve ser a primeira linha do código
  - Fora do bloco da classe só podem existir dois comando: *package* e *import*

# Pacotes


Projeto
modelo
Cliente.java
visao
CadastroCliente.java
controle
ManipulaCliente.java

# Pacotes

```
package modelo;
public class Cliente {
...
}
```

```
package visao;
public class CadastroCliente{
...
}
```

```
package controle;
public class ManipulaCliente{
...
}
```

# Pacotes

- Criação de um objeto Cliente na classe ManipulaCliente

```
package controle;
import modelo.Cliente;
public class ManipulaCliente{
    ...
    Cliente c = new Cliente();
    ..
}
```

# Pacotes

- No import pode-se informar a importação de todas as classe de um pacote

```
package controle;
import modelo.*;
public class ManipulaCliente{
    ...
    Cliente c = new Cliente();
    ..
}
```

# Pacotes

- O import pode ser substituído se o nome da classe tiver o pacote onde ela se encontra

```
package controle;
public class ManipulaCliente{
    ...
    modelo.Cliente c = new modelo.Cliente();
    ..
}
```

# Pacotes

- Divide o programa em módulos
- Criação de bibliotecas de classe
- Auxilia a reutilização de código
- O Java já possui uma vasta biblioteca de classe, exemplos de pacotes
  - javax.swing: interface gráfica
  - java.net: rede
  - javax.sql: acesso a banco de dados

# Encapsulamento

- Encapsular: esconder
- Definir a quais membros da classe serão visíveis
- Tipos de encapsulamento

| | Local | Filha | Objeto |
|---|---|---|---|
| public | Sim | Sim | Sim |
| protected | Sim | Sim | Sim/Não |
| private | Sim | Não | Não |

# Encapsulamento

```
public class Protege {
    public int a;
    protected int b;
    private int c;

    public void exibe(){
        System.out.println("a:" + a);
        System.out.println("b: "+ b);
        System.out.println("c: "+ c);
    }
}
```

# Encapsulamento

```
public class Filha extends Protege{
    public void exibe(){
        System.out.println("a:" + a);
        System.out.println("b: "+ b);
    }
}
```

# Encapsulamento

```
public class Principal {
    public static void main(String[] args) {
        Protege p = new Protege();
        System.out.println("a: " + p.a);
        System.out.println(“b: " + p.b);
    }
}
```

Somente dentro do mesmo pacote

# Polimorfismo

- Conceitos
  - Método abstrato: não possui corpo
  - Classe abstrata: não instancia objeto
  - Interface: semelhante a classe abstrata, mas só possui métodos abstratos
- Utilização da palavra reservada *abstract*
- Importante:
  - Variáveis de superclasses podem receber objetos de classes subclasses

# Polimorfismo

```
public abstract class Figura {
    private String cor;
    public String getCor() {
        return cor;
    }
    public void setCor(String cor) {
        this.cor = cor;
    }
    public abstract double area();
}
```

# Polimorfismo

```
public class Quadrado extends Figura{
    private double lado;
    public double getLado() {
        return lado;
    }
    public void setLado(double lado) {
        this.lado = lado;
    }
    public double area(){
        return getLado() * getLado();
    }
}
```

# Polimorfismo

```
public class Circulo extends Figura{
    private double raio;
    public double getRaio() {
        return raio;
    }
    public void setRaio(double raio) {
        this.raio = raio;
    }
    public double area(){
        return 3.14 * getRaio() * getRaio();
    }
}
```

```
import javax.swing.*;
public class Principal {
	public static void main (String args[]){
		Figura f;
		String e;
		e=JOptionPane.showInputDialog("1-Quadrado\n"+"2-Circulo");
		if(e.equals("1")){
			Quadrado q = new Quadrado();
			q.setCor(JOptionPane.showInputDialog("cor"));
			double lado = Double.parseDouble(JOptionPane.showInputDialog("Lado"));
			q.setLado(lado);
			f = q;
		}
		else{
			Circulo c = new Circulo();
			c.setCor(JOptionPane.showInputDialog("cor"));
			double raio = Double.parseDouble(JOptionPane.showInputDialog("Raio"));
			c.setRaio(raio);
			f = c;
		}
		JOptionPane.showMessageDialog(null, "Area:" + f.area());
	}
}
```

# Polimorfismo

- Interface
  - Equivalente a uma classe abstrata
  - Possui somente constantes e métodos abstratos
  - Herança através da palavra implements
  - Suas subclasses herdam a “obrigação” de implementar os métodos definidos na interface

# Polimorfismo

```
public interface Boneco {
    public abstract void correr();
    public abstract void andar();
    public abstract void chutar();
}

public class Jogador implements Boneco{
    public void correr(){...}
    public void andar(){...}
    public void chutar(){...}
}
```

# Conclusão

- Apresentados os principais conceito de Programação Orientada a Objetos
  - Classe
  - Objeto
  - Herança
  - Encapsulamento
  - Polimorfismo
- Conceito não mencionado
  - Sobrecarga de métodos
  - Membros static
  - Herança múltipla de interfaces

# Fim