









# Java e Orientação a Objetos: Dos Conceitos à Prática

Professor: Francisco Molina

# Agenda da Apresentação

-  Por que Orientação a Objetos?
-  Blocos de Construção: Classes e Objetos
-  Os 4 Pilares da POO
-  Juntando as Peças: Interfaces e Pacotes
-  Hora da Prática: Exercícios
-  Resumo e Dúvidas

# Por que usar a Programação Orientada a Objetos?

- **Visualizar a complexidade do software**

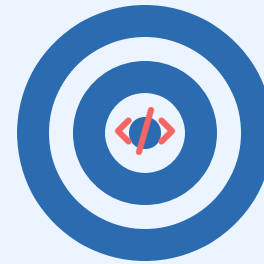
Modelar programas como vemos e interagimos com objetos do mundo real, tornando o código mais intuitivo e fácil de entender.

- **Acelerar o desenvolvimento**

Utilizar componentes modulares e reutilizáveis que permitem construir sistemas complexos a partir de blocos simples.

- **Aumentar qualidade e manutenibilidade**

Criar software mais robusto, mais fácil de testar, dar manutenção e evoluir ao longo do tempo.



# Classes: O Molde dos Objetos

- **Definição**

"Classes definem a estrutura e o comportamento de um tipo de objeto".

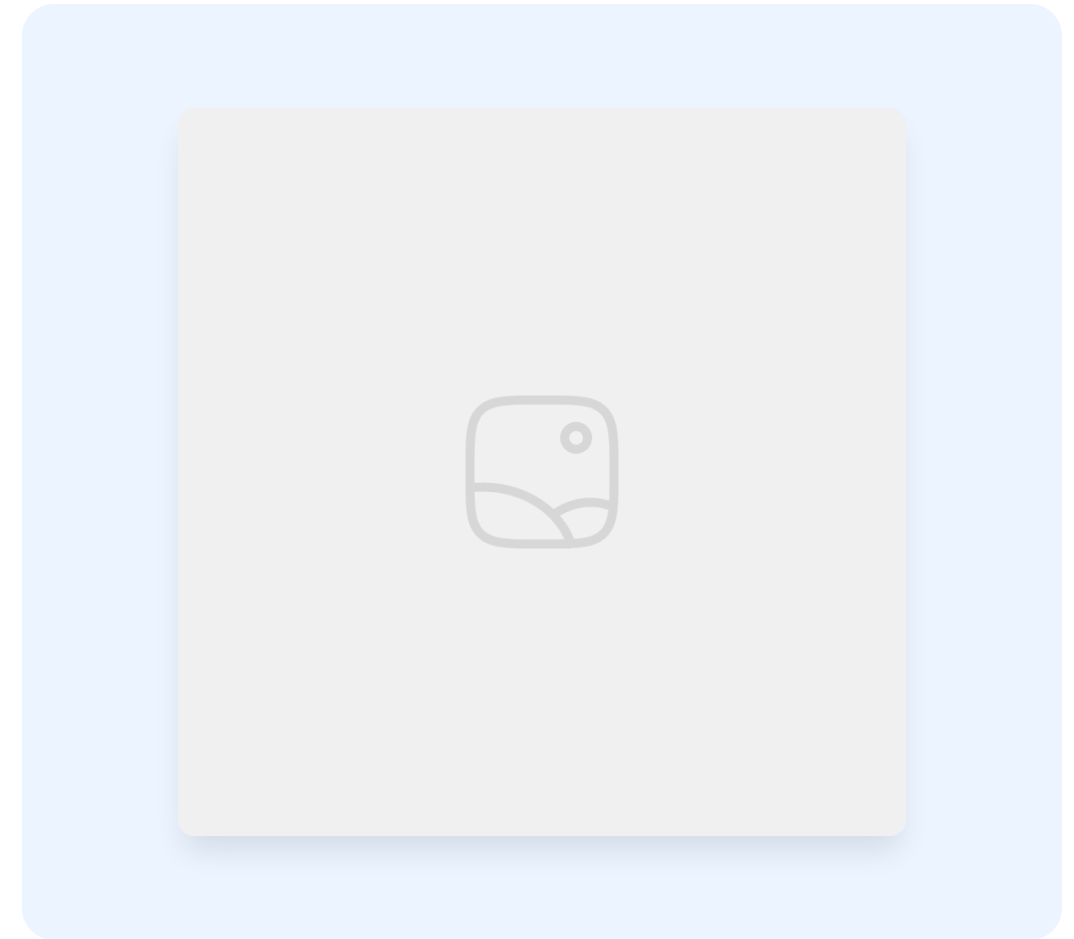
- **Templates**

Elas são **templates** que especificam:

- **Atributos** (os dados, as características)
- **Métodos** (as ações, os comportamentos)

- **Analogia**

Pense em uma classe como uma planta baixa de uma casa ou um molde de biscoitos. Ela não é o objeto final, mas o projeto que define tudo sobre ele.



# Objetos: A Instância Concreta

- Definição de Objeto

Um objeto é uma **instância concreta de uma classe** com valores específicos para seus atributos e capacidade de executar ações definidas pelos métodos.

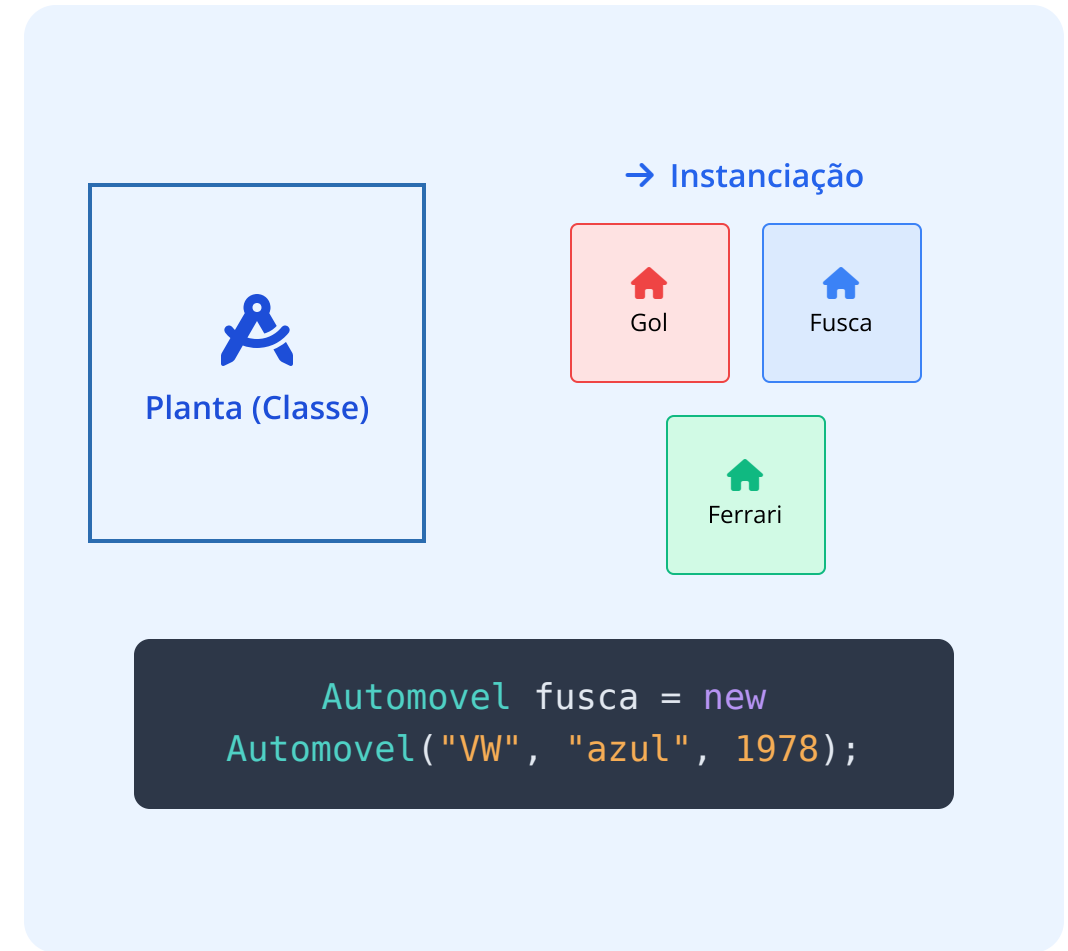
- Analogia com o mundo real

Se a classe é a planta, o objeto é a casa construída. A partir de uma única planta (classe), podemos construir várias casas (objetos) diferentes, cada uma com suas características únicas.

- Instanciação em Java

Para criar um objeto em Java, utilizamos a palavra-chave `new`, seguida pelo construtor da classe:

```
Automovel gol = new Automovel();
```



# Pilar 1: Abstração

- **Definição**

Capacidade de focar no que é essencial e ignorar detalhes irrelevantes para o contexto.

- **Objetivo**

Simplificar a complexidade do problema, extraíndo apenas as características e comportamentos necessários.

- **Na prática**

Uma mesma entidade pode ter diferentes representações abstratas dependendo do contexto e de quem a utiliza.



## Visão do Piloto

Foco na operação e desempenho

Cor

VelocidadeMaxima

Acelerar()

Frear()



## Visão do Policial

Foco na identificação e regulamentação

Placa

Cor

NumeroChassi

AplicarMultas()

VerificarDocumentos()

# Pilar 2: Encapsulamento

- Definição

Agrupar dados (atributos) e comportamentos (métodos) dentro de uma "cápsula" (o objeto) e proteger o acesso direto aos dados.

- Exemplo em código

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

- Modificadores de visibilidade

public

protected

private



Como uma cápsula de remédio

O invólucro protege o conteúdo, controlando o acesso



Atributos  
private



Métodos  
getters  
public



Métodos  
setters  
public

# Pilar 3: Herança

- Mecanismo de reutilização de código

Permite que uma classe (subclasse) herde atributos e métodos de outra classe (superclasse), estabelecendo uma relação "é um".

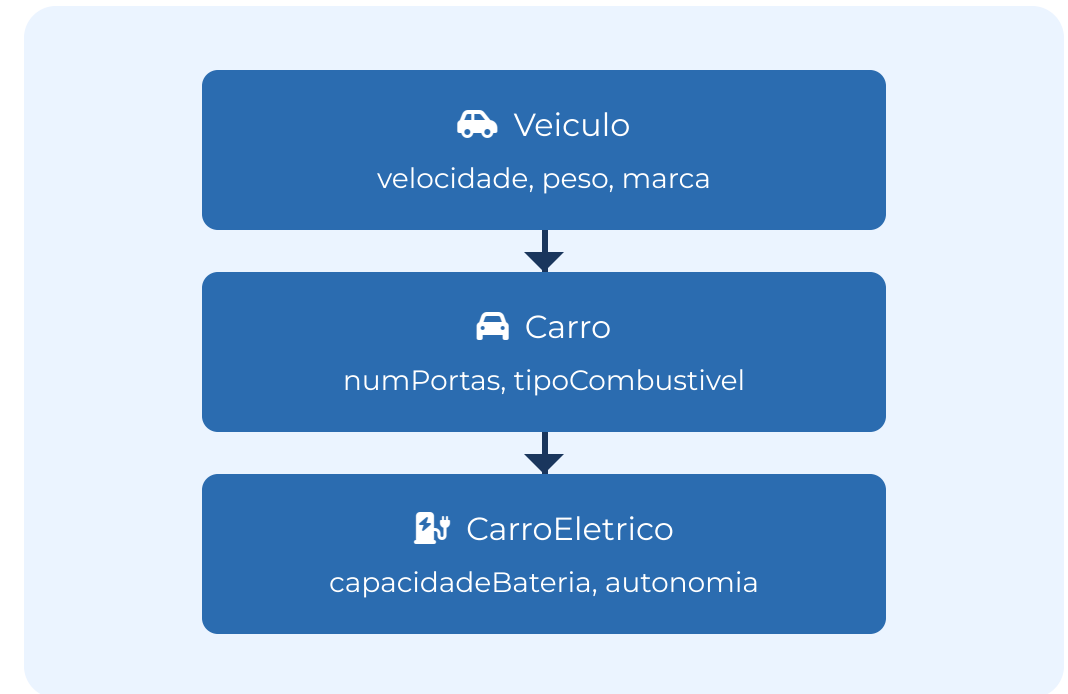
- Palavras-chave em Java

**extends** : declara que uma classe herda de outra

**super** : referencia membros da classe pai

- Benefícios

Reutilização de código, organização hierárquica e especialização gradual dos componentes do sistema.





# Pilar 4: Polimorfismo

- **Polimorfismo = "Muitas Formas"**

Capacidade de um objeto se comportar de maneiras diferentes dependendo do contexto.

- **Sobreposição (Overriding)**

Uma subclasse fornece uma implementação específica para um método da superclasse.

```
@Override
public void fazerSom() {
    System.out.println("Au Au!");
}
```

- **Sobrecarga (Overloading)**

Vários métodos com o mesmo nome na mesma classe, mas com parâmetros diferentes.

```
void fazerSom() { ... }
void fazerSom(int volume) { ... }
```

**fazerSom()** em diferentes animais:



"Au Au!"

Cachorro



"Miau!"

Gato



"Piu Piu!"

Pássaro

Mesmo método **fazerSom()**, comportamentos diferentes!

Cada subclasse implementa o método à sua maneira

# Interfaces e Classes Abstratas

- **Classe Abstrata**

Um "molde" parcialmente pronto que não pode ser instanciado diretamente.

Pode conter métodos concretos e abstratos.

**Relação "é um"**

```
public abstract class Veiculo {  
    protected String marca;  
    public abstract void mover();  
}
```

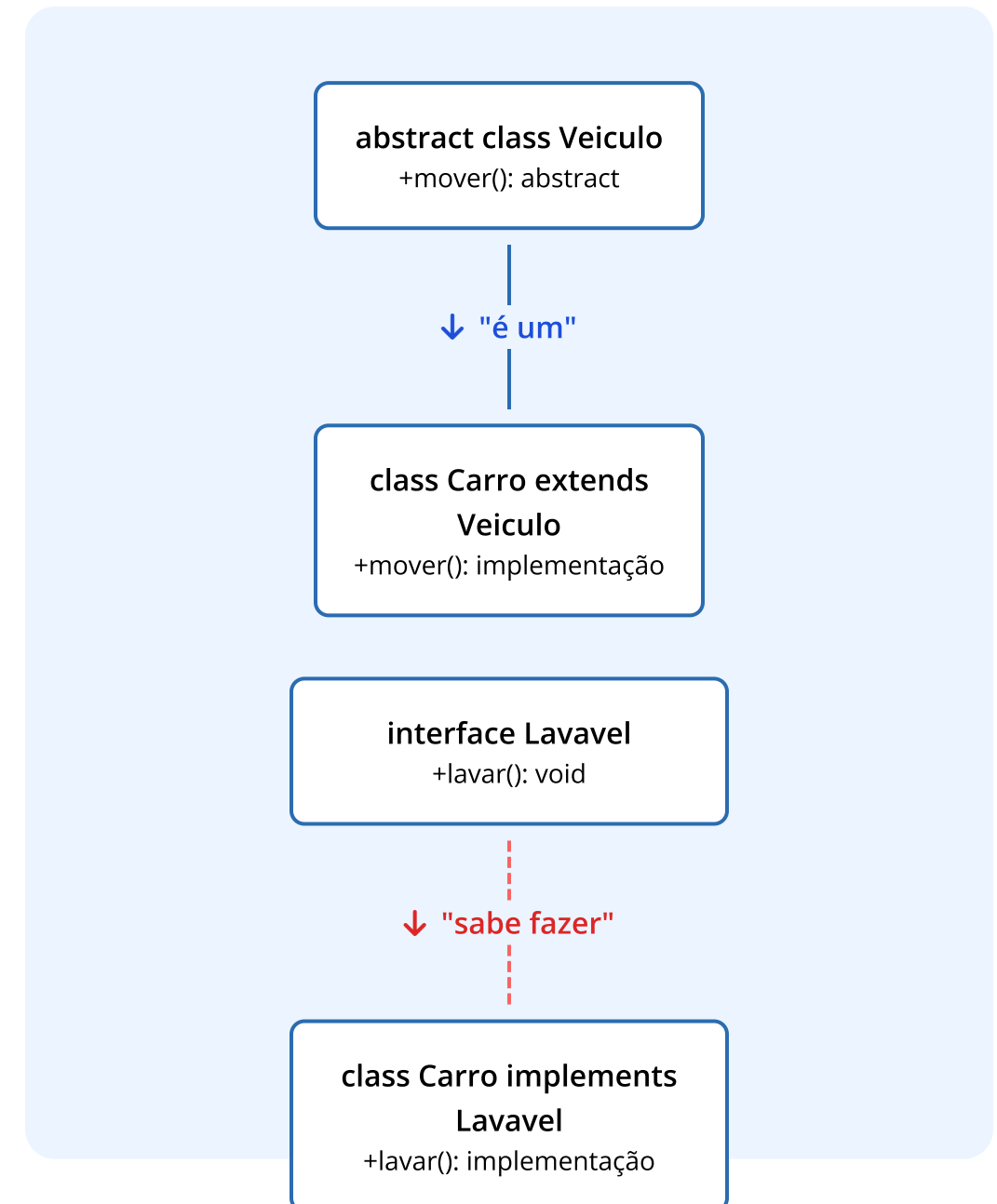
- **Interface**

Um "contrato" de comportamento que garante implementações específicas.

Todos os métodos são implicitamente públicos e abstratos.

**Relação "sabe fazer"**

```
public interface ControleRemoto {  
    void ligar();  
    void desligar();  
}
```



# Exercício Prático 1: Abstração e Encapsulamento

## Desafio

Modele uma classe **ContaBancaria** utilizando os conceitos de abstração e encapsulamento vistos.

- **Quais são os atributos essenciais?**  
Pense nos dados que uma conta bancária precisa armazenar.
- **Quais são os comportamentos essenciais?**  
Quais operações uma conta bancária pode realizar?
- **Lembre-se do encapsulamento!**  
Proteja os dados usando modificadores de acesso adequados.



Tempo para resolução: **15 minutos**

Discuta com seus colegas ou trabalhe individualmente

ContaBancaria.java

```
public class ContaBancaria {  
    // Atributos (encapsulados)  
    private String titular;  
    private String numeroConta;  
    private double saldo;  
  
    // Construtor  
    public ContaBancaria(String titular, String numeroConta)  
    {  
        this.titular = titular;  
        this.numeroConta = numeroConta;  
        this.saldo = 0.0;  
    }  
  
    // Métodos  
    public void depositar(double valor) {  
        // Implemente aqui...  
    }  
  
    public boolean sacar(double valor) {  
        // Implemente aqui...  
        return false;  
    }  
  
    // Getters e setters  
    public String getTitular() {  
        return titular;  
    }  
  
    // Continue implementando...  
}
```

# Solução do Exercício 1

## Solução

✓ **Abstração**

Identificamos atributos essenciais: titular, numeroConta e saldo.

✓ **Encapsulamento**

Atributos privados com getters e setters controlados.

✓ **Validações**

Aplicadas nos métodos para garantir integridade dos dados.

Exemplo de uso:



```
ContaBancaria conta = new ContaBancaria(
    "Ana Silva", "12345-6");
conta.depositar(1000);
boolean sucesso = conta.sacar(500);
System.out.println(conta.getSaldo());
```

```
ContaBancaria.java

public class ContaBancaria {
    // Atributos encapsulados
    private String titular;
    private String numeroConta;
    private double saldo;

    // Construtor
    public ContaBancaria(String titular, String numeroConta) {
        this.titular = titular;
        this.numeroConta = numeroConta;
        this.saldo = 0.0;
    }

    // Métodos principais
    public void depositar(double valor) {
        if (valor > 0) {
            this.saldo += valor;
            System.out.println("Depósito de " + valor + " realizado com sucesso.")
        } else {
            System.out.println("Valor inválido para depósito.");
        }
    }

    public boolean sacar(double valor) {
        if (valor > 0 && valor <= saldo) {
            this.saldo -= valor;
            System.out.println("Saque de " + valor + " realizado com sucesso.");
            return true;
        } else {
            System.out.println("Saldo insuficiente ou valor inválido.");
            return false;
        }
    }

    // Getters e setters
    public String getTitular() {
        return titular;
    }

    public String getNumeroConta() {
        return numeroConta;
    }

    public double getSaldo() {
        return saldo;
    }

    // Não permitimos alteração direta do saldo
    // Não fornecemos setters para campos que não devem
    // ser alterados após a criação
}
```

# Exercício Prático 2: Herança e Polimorfismo

## Desafio

A partir da classe `ContaBancaria` que criamos no exercício anterior, crie:

- **ContaCorrente**  
Deve ter um limite de cheque especial e um método sacar que o considere (sobreposição).
- **ContaPoupanca**  
Deve ter um método para aplicar rendimento mensal sobre o saldo.
- **Aplique o polimorfismo!**  
Demonstre como o mesmo método (sacar) pode se comportar diferentemente em cada tipo de conta.



Tempo para resolução: **20 minutos**

Pense nas diferenças entre os tipos de conta e como implementá-las

### Estrutura das Classes

```
// Já temos esta classe do exercício anterior
public class ContaBancaria {
    private String titular;
    private String numeroConta;
    private double saldo;

    // Construtor, getters e setters...

    public boolean sacar(double valor) {
        if (valor <= saldo && valor > 0) {
            saldo -= valor;
            return true;
        }
        return false;
    }
}

// Crie esta classe:
public class ContaCorrente extends ContaBancaria {
    private double limiteChequeEspecial;

    // Como implementar a sobrescrita do método sacar
    // para considerar o cheque especial?
    @Override
    public boolean sacar(double valor) {
        // Implemente aqui...
    }
}

// E também esta:
public class ContaPoupanca extends ContaBancaria {
    private double taxaRendimento;

    // Como implementar o método para aplicar rendimento?
    public void aplicarRendimento() {
        // Implemente aqui...
    }
}
```

# Solução do Exercício 2: Herança e Polimorfismo

## Solução

Criamos duas subclasses que herdam de **ContaBancaria**, cada uma com comportamentos especializados:

- ✓ **ContaCorrente** implementa um limite de cheque especial e sobrepõe o método **sacar()** para considerar esse limite.
- ✓ **ContaPoupanca** adiciona um método **aplicarRendimento()** e mantém o comportamento original de saque.
- ✓ **Polimorfismo demonstrado** pelo mesmo método **sacar()** comportando-se de maneiras diferentes nas subclasses.



### Pontos importantes:

- Reutilização de código da classe pai
- Uso da palavra-chave **super** para acessar métodos da superclasse

```
ContaCorrente.java  ContaPoupanca.java

public class ContaCorrente extends ContaBancaria {
    private double limiteChequeEspecial;

    public ContaCorrente(String titular, String numeroConta,
                        double limiteChequeEspecial) {
        super(titular, numeroConta); // Chama construtor da
        this.limiteChequeEspecial = limiteChequeEspecial;
    }

    // Sobrepõe (override) o método sacar da classe pai
    @Override
    public boolean sacar(double valor) {
        // Verifica se o valor está dentro do limite dispon.
        if (valor > 0 && valor <= (getSaldo() + limiteChequeEspecial)) {
            // Atualiza o saldo usando método setSaldo herdado
            setSaldo(getSaldo() - valor);
            return true;
        }
        return false; // Saldo insuficiente ou valor inválido
    }

    // Getters e setters específicos
    public double getLimiteChequeEspecial() {
        return limiteChequeEspecial;
    }

    public double getSaldoDisponivel() {
        return getSaldo() + limiteChequeEspecial;
    }
}
```

# Resumo e Próximos Passos

## Recapitulando:



### Abstração

Focar no essencial e ignorar detalhes irrelevantes



### Encapsulamento

Proteger os dados e expor apenas o necessário



### Herança

Reutilizar código e estender funcionalidades



### Polimorfismo

Múltiplas formas de comportamento para uma mesma interface

## Próximos passos:



### Pratique!

A melhor forma de aprender é criando seus próprios projetos. Comece com exemplos simples e evolua gradativamente.



### Documentação Oficial

A documentação do Java é uma excelente fonte para aprofundar seus conhecimentos:

[Java Tutorials \(Oracle\)](#)



### Participe da comunidade

Compartilhe seu código, faça perguntas e contribua em fóruns e plataformas de código aberto.

# Dúvidas



## Alguma dúvida?

Este é o momento para esclarecer questões e aprofundar os conceitos apresentados.

Fique à vontade para perguntar!

### Informações de contato:



#### Email

[apresentador@empresa.com](mailto:apresentador@empresa.com)



#### Website

[www.empresa.com](http://www.empresa.com)



#### LinkedIn

[linkedin.com/in/apresentador](https://linkedin.com/in/apresentador)



#### GitHub

[github.com/apresentador](https://github.com/apresentador)