

CURSO

# Java Web com Spring Boot

 @alexandrejunior.dev

## Bem-vindo(a)!

Se você está interessado em aprender a programar em uma das linguagens de programação mais populares do mundo, este é o curso certo para você.

Java é uma linguagem de programação orientada a objetos amplamente utilizada em vários setores, incluindo finanças, saúde, educação, jogos, entre outros. Seu poder reside na capacidade de escrever código modular, escalável e fácil de manter. Além disso, a vasta comunidade de desenvolvedores e as inúmeras bibliotecas de código aberto disponíveis tornam Java uma das linguagens de programação mais flexíveis e versáteis.

O curso é projetado para ensinar a programação em Java, com foco especial na programação orientada a objetos e no desenvolvimento de aplicativos com Spring Boot. Você aprenderá como escrever código eficiente, modular e escalável, utilizando os recursos mais recentes da linguagem Java. Além disso, você aprenderá como usar o Spring Boot para criar aplicativos web e serviços RESTful.

Ao final do curso, você estará apto a desenvolver aplicativos Java completos e complexos, que utilizam padrões de programação modernos e ferramentas de desenvolvimento atualizadas. Você também estará pronto para aplicar seus conhecimentos em projetos

reais, contribuindo para o mercado de trabalho e ampliando suas oportunidades de carreira.

## Pré-requisitos

Fundamentos de lógica de programação.

## Ferramentas

- Java JDK (versão igual ou superior a 8)
- Visual Studio Code IDE (recomendado)
- Postman (opcional)

## Repositório do Curso

O código-fonte dos exemplos realizados neste curso podem ser encontrados no GitHub, através do link:

<https://github.com/alexandresjunior>

## Sumário

### Linguagem Java

1. Variáveis e Constantes
2. Operadores
3. Estruturas de Decisão
4. Laços de Repetição
5. Arrays
6. Funções
7. Tratamento de Exceções

### Programação Orientação a Objetos

8. Classes e Objetos
9. Métodos Estáticos
10. Herança
11. Polimorfismo
12. Overload e Override
13. Classes Abstratas e Interfaces

### Spring Framework

14. Introdução
15. Padrão de Projeto e Spring MVC
16. REST APIs
17. Criação um projeto com Spring Boot

## Linguagem Java

### Variáveis e Constantes

Em Java, variáveis são usadas para armazenar valores temporários que podem ser modificados ao longo do tempo. Já as constantes, como o nome sugere, são valores que não podem ser alterados após a sua inicialização.

#### Declaração de variáveis em Java

Em Java, uma variável **deve ser declarada antes de ser usada**. Para declarar uma variável, primeiro é necessário **indicar o tipo de dado** que será armazenado. Em seguida, é preciso **escolher um nome para a variável**. Por fim, o sinal de igual (=) é usado para **atribuir um valor** inicial à variável.

Veja alguns exemplos:

```
/* Variável idade do tipo inteiro
 * com valor inicial 30 */
int idade = 30;

/* Variável preco do tipo double
 * com valor inicial 10.5 */
double preco = 10.5;
```

```
/* Variável nome do tipo String
 * com valor inicial "João" */
String nome = "João";
```

#### Declaração de constantes em Java

Para declarar uma constante em Java, é preciso usar a palavra-chave "final" antes da declaração da variável. Veja alguns exemplos:

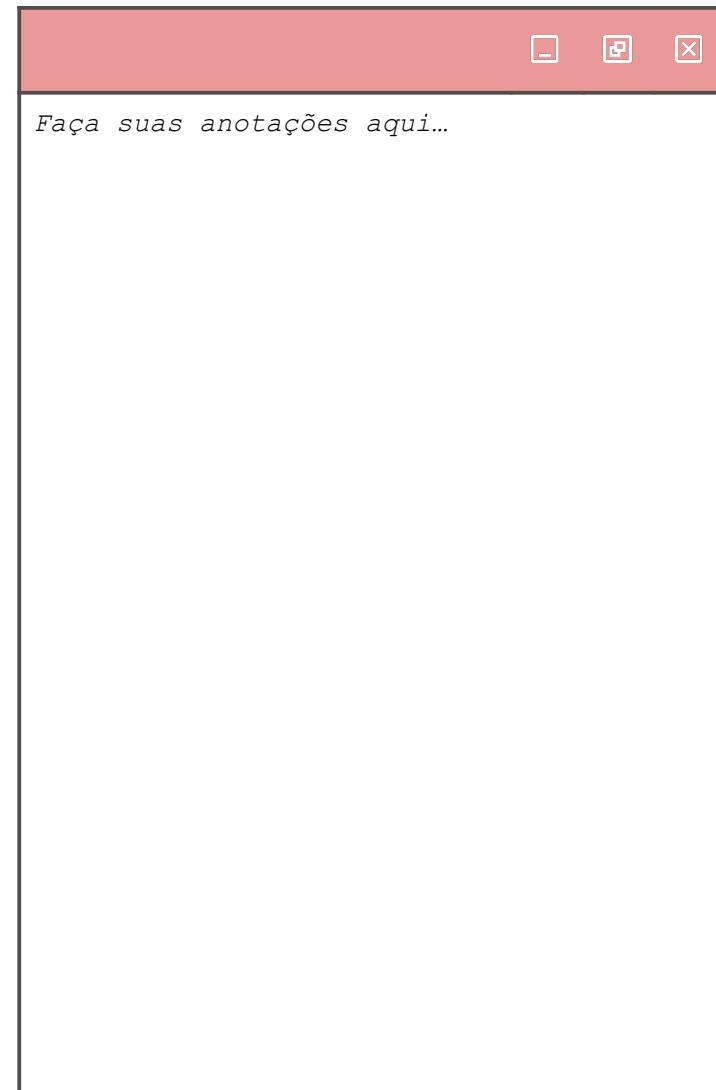
```
/* Constante PI do tipo double
 * com valor inicial 3.14159 */
final double PI = 3.14159;
```

```
/* Constante EMPRESA do tipo String
 * com valor inicial "Minha Empresa" */
final String EMPRESA = "Minha Empresa";
```

É importante lembrar que uma constante **não pode ser alterada após a sua inicialização**. Caso seja tentado atribuir um novo valor a uma constante, um erro será gerado pelo compilador.



*Faça suas anotações aqui...*



*Faça suas anotações aqui...*

## Operadores

Os operadores são elementos importantes em qualquer linguagem de programação e em Java não é diferente. Em Java, existem vários tipos de operadores, que podem ser usados para realizar diversas operações em variáveis e constantes.

A seguir serão apresentados alguns dos principais operadores em Java.

### Operadores Aritméticos

Os operadores aritméticos em Java são usados para realizar operações matemáticas em valores numéricos. São eles:

+	Soma
-	Subtração
*	Multiplicação
/	Divisão
%	Resto da divisão inteira

### Operadores de Atribuição

Os operadores de atribuição em Java são usados para atribuir valores a variáveis. São eles:

=	Atribuição simples
+=	Atribuição com adição
-=	Atribuição com subtração
*=	Atribuição com multiplicação
/=	Atribuição com divisão
%=	Atribuição com resto da divisão inteira

### Operadores de Comparaçāo

Os operadores de comparação em Java são usados para comparar valores e retornar um resultado **booleano** (*verdadeiro ou falso*). São eles:

==	Igualdade
!=	Diferente
<	Menor que
>	Maior que
<=	Menor ou igual a
>=	Maior ou igual a

## Operadores Lógicos

Os operadores lógicos em Java são usados para realizar operações lógicas em valores **booleanos**. São eles:

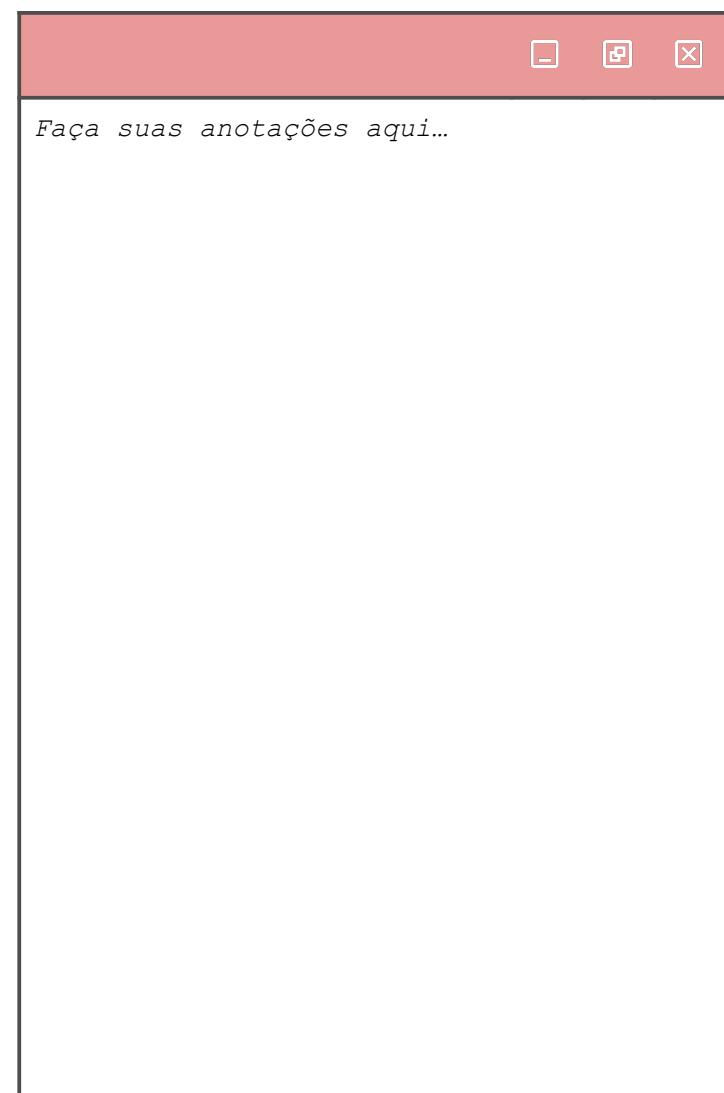
<b>&amp;&amp;</b>	"E" lógico
<b>  </b>	"OU" lógico
<b>!</b>	Negação lógica

## Operadores de Incremento e Decremento

Os operadores de incremento e decremento em Java são usados para **aumentar** ou **diminuir** o valor de uma variável em **uma unidade**, respectivamente. São eles:

<b>++</b>	Incremento
<b>--</b>	Decremento

Esses são alguns dos principais operadores em Java. É importante lembrar que o uso correto dos operadores é fundamental para escrever um código limpo e eficiente.





## Estruturas de Decisão

As estruturas de decisão são elementos fundamentais em qualquer linguagem de programação e em Java não é diferente. As estruturas de decisão permitem que o programa tome uma decisão baseada em uma condição lógica. Em Java, existem duas estruturas de decisão principais: o ***if*** e o ***switch***.

### ***if***

O ***if*** é a estrutura de decisão mais simples em Java e é usado para executar um bloco de código somente se uma condição for verdadeira. A sintaxe do ***if*** é a seguinte:

```
if (condicao) {  
    /* código a ser executado  
     * SE a condição for verdadeira */  
}
```

Por exemplo, o código a seguir verifica se uma variável chamada `idade` é maior ou igual a 18 e exibe uma mensagem na tela se a condição for verdadeira:

```
int idade = 20;  
if (idade >= 18) {  
    System.out.println("Você é de maior!");  
}
```

Aqui está um exemplo de uso do *if* e *else* juntos para tomar uma decisão com base em uma condição:

```
int idade = 20;
if (idade >= 18) {
    System.out.println("Você é de maior!");
} else {
    System.out.println("Você é de menor!");
}
```

Nesse exemplo, a variável *idade* é verificada usando a estrutura *if* para determinar se ela é maior ou igual a 18. Se a condição for verdadeira, a mensagem "Você é de maior!" será exibida na tela. Se a condição for falsa, a mensagem "Você é de menor!" será exibida na tela.

Note que o *else* é usado para fornecer um bloco de código a ser executado se a condição no *if* for falsa. Essa estrutura de *if/else* permite que o programa tome diferentes caminhos com base na condição da variável *idade*.

### **switch**

O *switch* é outra estrutura de decisão em Java e é usado para executar um bloco de código diferente com base em diferentes valores de uma variável.

A sintaxe do *switch* é a seguinte:

```
switch (variavel) {
    case valor1:
        /* Código a ser executado se
         * a variável for igual a valor1 */
        break;
    case valor2:
        /* Código a ser executado se
         * a variável for igual a valor2 */
        break;
    default:
        /* Código a ser executado se
         * a variável não for igual a
         * nenhum dos valores anteriores */
        break;
}
```

Por exemplo, o código a seguir verifica o valor de uma variável chamada *dia* e exibe uma mensagem na tela com base no valor:

```
int dia = 3;

switch (dia) {
    case 1:
        System.out.println("Hoje é "
                           + "segunda-feira.");
        break;
    case 2:
        System.out.println("Hoje é "
                           + "terça-feira.");
        break;
```

```
case 3:  
    System.out.println("Hoje é "  
                       + "quarta-feira.");  
    break;  
default:  
    System.out.println("Não sei "  
                       + "que dia é hoje.");  
    break;  
}
```

Essas são as duas principais estruturas de decisão em Java. É importante lembrar que o uso correto das estruturas de decisão é fundamental para escrever um código limpo e eficiente.





## Laços de Repetição

Os laços de repetição, também chamados de *loops*, são uma parte fundamental da programação em Java e permitem que o programa **execute o mesmo bloco de código várias vezes**. Em Java, existem três tipos principais de loops: *for*, *while* e *do-while*.

### *for*

O *for* é um tipo de *loop* que é **executado um número específico de vezes**. Ele é usado para iterar sobre uma sequência de valores, como uma lista ou uma matriz.

A sintaxe do *for* é a seguinte:

```
for (inicialização; condição; incremento)
{
    /* Código a ser executado
     * em cada iteração */
}
```

Por exemplo, o código a seguir usa um *for* para exibir os números de 1 a 5 na tela:

```
for (int i = 1; i <= 5; i++) {
    System.out.println(i);
}
```

### ***while***

O *while* é outro tipo de *loop* que é executado **enquanto uma condição for verdadeira**. Ele é usado **quando não se sabe exatamente quantas vezes** o loop deve ser executado.

A sintaxe do *while* é a seguinte:

```
while (condição) {
    /* Código a ser executado
     * em cada iteração */
}
```

Por exemplo, o código a seguir usa um *while* para exibir os números de 1 a 5 na tela:

```
int i = 1;
while (i <= 5) {
    System.out.println(i);
    i++;
}
```

### ***do-while***

O *do-while* é semelhante ao *while*, mas garante que o bloco de **código seja executado pelo menos uma vez**, mesmo que a condição seja falsa.

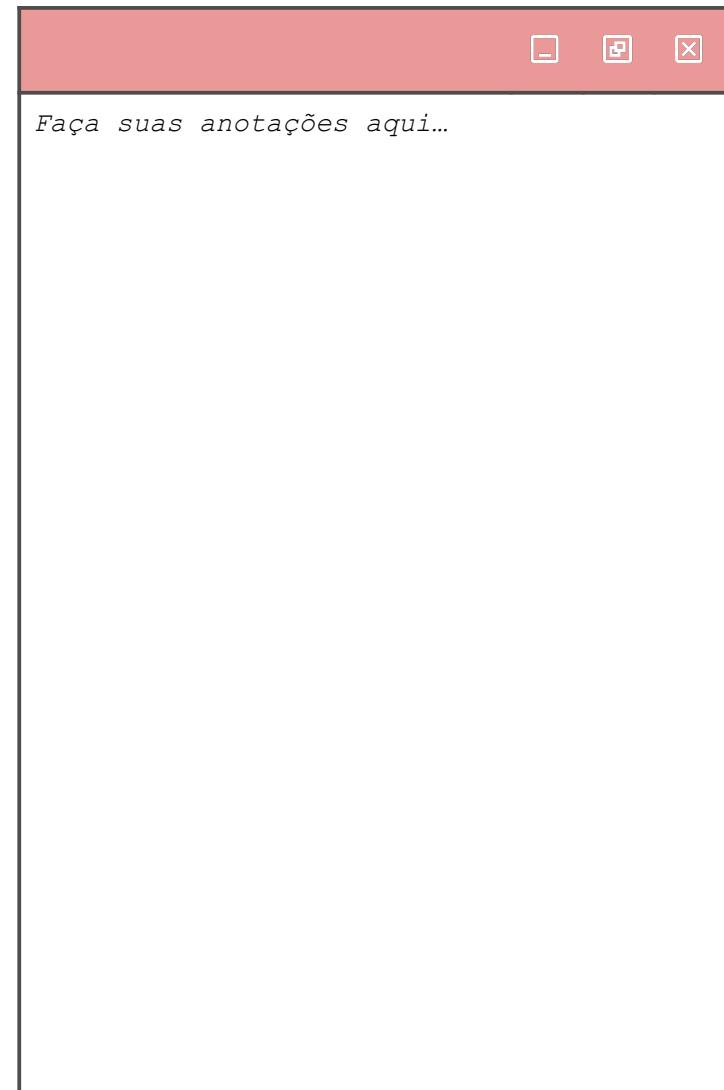
A sintaxe do *do-while* é a seguinte:

```
do {
    /* Código a ser executado
     * em cada iteração */
} while (condição);
```

Por exemplo, o código a seguir usa um *do-while* para exibir os números de 1 a 5 na tela:

```
int i = 1;
do {
    System.out.println(i);
    i++;
} while (i <= 5);
```

Esses são os três tipos principais de loops em Java. É importante lembrar que o uso correto dos *loops* é fundamental para escrever um código eficiente e **evitar loops infinitos** ou outros erros de lógica.



## Arrays

Arrays são estruturas de dados usadas para armazenar **um conjunto de valores do mesmo tipo** em uma única variável. Em Java, os arrays são objetos que contêm um **número fixo de elementos** do mesmo tipo. Cada elemento é identificado por um índice inteiro, que começa em 0 (zero).

Aqui está um exemplo simples de como declarar e inicializar um array de inteiros em Java:

```
int[] numeros = new int[5];

numeros[0] = 1;
numeros[1] = 2;
numeros[2] = 3;
numeros[3] = 4;
numeros[4] = 5;
```

Neste exemplo, estamos declarando um array chamado **numeros** com capacidade para armazenar 5 inteiros. Em seguida, atribuímos um valor a cada elemento do array, referenciando-o pelo seu **índice**.

Também é possível inicializar um array no momento da sua declaração. O exemplo a seguir inicializa um array com os mesmos valores do exemplo anterior:

```
int[] numeros = {1, 2, 3, 4, 5};
```

Podemos acessar os valores dos elementos de um array usando o seu índice, como no exemplo abaixo:

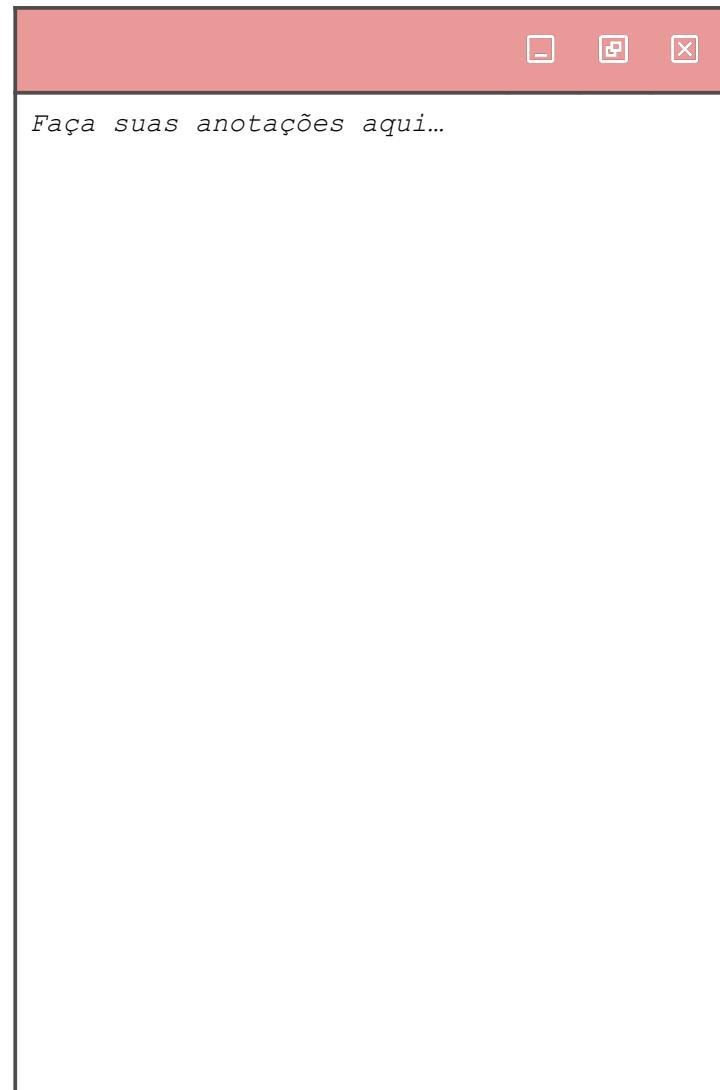
```
// segundoNumero agora contém o valor 2
int segundoNumero = numeros[1];
```

Também podemos percorrer os elementos de um array usando um *loop for*, como no exemplo a seguir:

```
for (int i = 0; i < numeros.length; i++) {
    System.out.println(numeros[i]);
}
```

A propriedade **length** retorna o número de elementos em um array. Usamos essa propriedade para controlar o número de iterações no *loop for* acima.

Os arrays em Java podem ser de tipos primitivos, como **int** e **float**, ou de tipos de **objetos**, como **String** ou **outras classes**. Eles são muito úteis para armazenar coleções de valores relacionados e são amplamente usados em programação.



## Funções

Em Java, funções são chamadas de **métodos**. Um método é um bloco de código que realiza uma tarefa específica e pode ser chamado em diferentes partes do programa. Eles são usados para que o código seja mais organizado, modular e reutilizável.

Aqui está um exemplo simples de como criar um método em Java:

```
public static int somar(int a, int b) {
    int resultado = a + b;
    return resultado;
}
```

Este método é chamado **somar** e recebe dois parâmetros inteiros **a** e **b**. Ele realiza a soma desses dois números e retorna o **resultado**.

A primeira palavra-chave **public** é um modificador de acesso que permite que o método seja chamado de outras classes. A palavra-chave **static** significa que o método pode ser chamado sem criar uma instância da classe em que ele está definido. O **tipo de retorno** do método é especificado antes do nome do método, que neste caso é **int**.

Para chamar o método **somar** em outra parte do programa, podemos simplesmente usar o seu nome e

fornecer os valores de entrada apropriados, como no exemplo a seguir:

```
// resultado agora contém o valor 5
int resultado = somar(2, 3);
```

Além disso, é possível definir **métodos sem retorno**, ou seja, que não retornam nenhum valor, usando a palavra-chave **void**. Por exemplo:

```
public static void imprimirMensagem(
    String mensagem) {
    System.out.println(mensagem);
}
```

Este método é chamado **imprimirMensagem** e recebe um parâmetro **mensagem** do tipo **String**. Ele simplesmente exibe a mensagem na tela usando o método **System.out.println()**.

Para chamar o método **imprimirMensagem**, basta passar uma String como parâmetro, como no exemplo a seguir:

```
// exibe "Olá, mundo!" na tela
imprimirMensagem("Olá, mundo!");
```

Existem muitas outras coisas que podemos fazer com métodos em Java, como **sobrecarga** de métodos, que permite definir **vários métodos com o mesmo nome**,

mas com **diferentes parâmetros de entrada**, e **métodos estáticos** de classe que podem ser chamados sem instanciar a classe. Mas esses são assuntos para serem vistos mais adiante. Por enquanto, ficaremos com as noções básicas de como criar e chamar métodos em Java.





## Tratamento de Exceções

O tratamento de erros é uma parte importante da programação em Java, pois ajuda a garantir que o programa possa lidar com problemas inesperados e falhas no sistema. Em Java, o tratamento de erros é feito por meio de exceções.

Uma exceção é um objeto que indica um erro ou condição anormal que ocorreu durante a execução do programa. Quando uma exceção é lançada, o fluxo normal de execução do programa é interrompido e o controle é transferido para o bloco de tratamento de exceção correspondente.

Em Java, existem duas categorias de exceções: **exceções verificadas** e **exceções não verificadas**. As exceções verificadas são aquelas que o compilador Java exige que o programador trate explicitamente, por meio do uso de **blocos try-catch** ou pela **declaração de exceções** em um método. As exceções não verificadas, por outro lado, são aquelas que o compilador Java não exige que o programador trate explicitamente, embora ainda possam ser capturadas e tratadas, se necessário.

O tratamento de erros em Java é feito por meio dos seguintes blocos de código:

**1. Bloco *try-catch*:** este bloco de código é usado para capturar exceções lançadas por uma seção específica do código. O bloco *try* contém o código que pode lançar uma exceção, enquanto o bloco *catch* contém o código que é executado quando a exceção é capturada.

```
try {
    // código que pode lançar uma exceção
} catch (IOException e) {
    // código que trata a exceção
}
```

**2. Bloco *finally*:** este bloco de código é usado para executar o código que deve ser executado independentemente de a exceção ter sido lançada ou não. Por exemplo, o bloco *finally* pode ser usado para liberar recursos que foram alocados durante a execução do código no bloco *try*.

```
try {
    // código que usa um recurso
} finally {
    /* código que fecha o recurso,
     * independentemente de ter
     * ocorrido uma exceção ou não */
}
```

**3. Declaração *throws*:** esta declaração é usada para indicar que um método pode lançar uma exceção verificada. Se um método pode lançar uma exceção

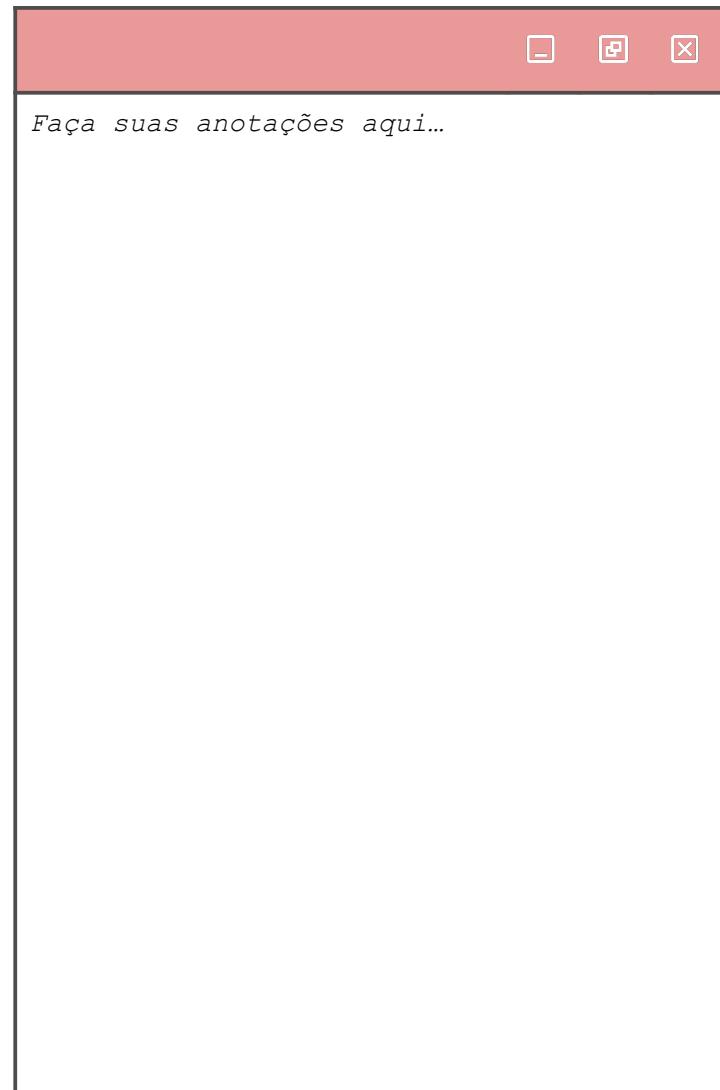
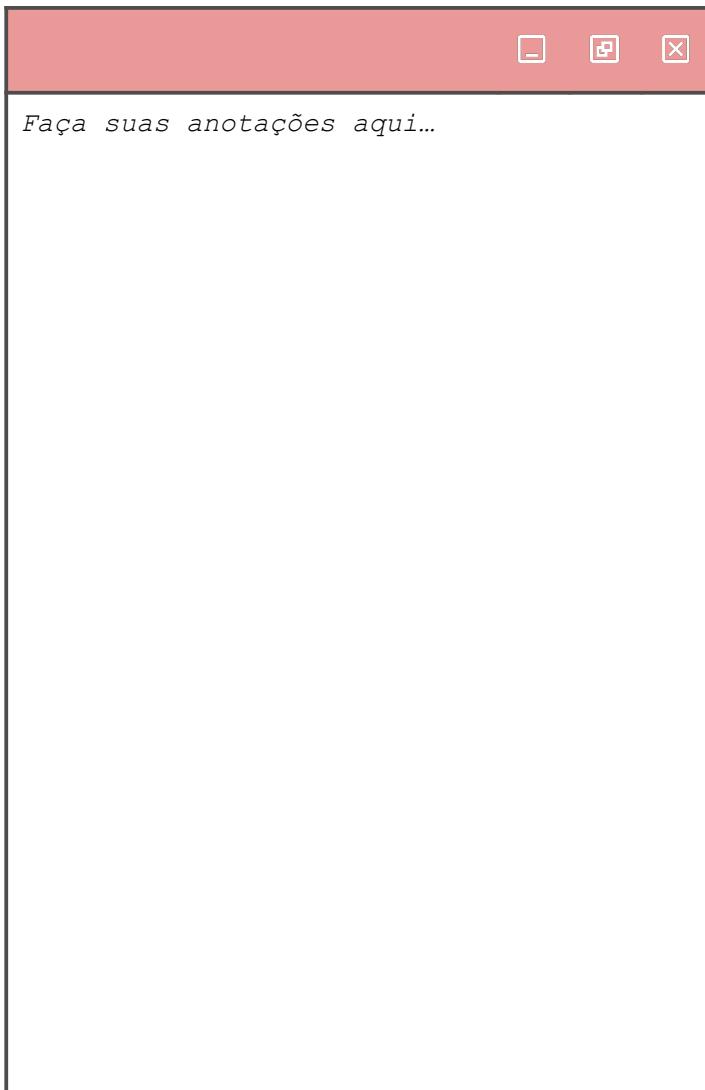
verificada, ele deve declarar isso em sua assinatura, para que o chamador do método saiba que deve tratá-la.

```
public void lerArquivo(String nomeArquivo)
    throws IOException {
    // código que lê um arquivo
}
```

**4. Bloco *try-with-resources*:** este bloco de código é usado para garantir que os recursos alocados dentro do bloco sejam fechados independentemente de ter ocorrido uma exceção ou não. Esse bloco de código pode ser usado em vez do bloco *finally*.

```
try (BufferedReader br =
      new BufferedReader(
          new FileReader("arquivo.txt"))) {
    // código que usa o BufferedReader
}
```

Em resumo, o tratamento de erros em Java é uma parte crítica da programação em Java, permitindo que os programadores lidem com erros e exceções inesperadas. O uso de exceções e dos blocos *try-catch*, *finally* e *throws* é fundamental para garantir que o código Java seja robusto e confiável.



## Programação Orientada a Objetos

### Classes e Objetos

A programação orientada a objetos (POO) é um paradigma de programação que se concentra na representação de objetos, que são entidades que possuem um **estado** e um **comportamento**. Em Java, a POO é amplamente utilizada e todos os programas são escritos usando esse paradigma.

Classes são a base da POO em Java. Uma classe é uma entidade que encapsula dados e comportamentos relacionados em um único lugar. Os dados são representados por variáveis, que são chamadas de **atributos**, e o comportamento é representado por **métodos** (funções).

Aqui está um exemplo de como definir uma classe em Java:

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

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

```
public void imprimirDados() {
    System.out.println("Nome: " + nome);
    System.out.println("Idade: "
        + idade);
}
```

Neste exemplo, estamos definindo uma classe chamada **Pessoa**. A classe tem dois campos **privados**, **nome** e **idade**, que são acessíveis apenas dentro da classe. A classe também tem um **construtor** que recebe um nome e uma idade e inicializa os campos correspondentes. Por fim, a classe tem um método público chamado **imprimirDados** que exibe os dados de uma pessoa na tela.

Para criar um objeto da classe **Pessoa** em outra parte do programa, podemos usar a palavra-chave **new** seguida do nome da classe e dos argumentos necessários para o construtor, como no exemplo abaixo:

```
Pessoa pessoa = new Pessoa("João", 25);
```

Este código cria um objeto da classe **Pessoa** chamado **pessoa** e o inicializa com o nome "João" e idade 25. Podemos acessar os campos de um objeto usando o operador ponto, como no exemplo a seguir:

```
// nome agora contém "João"  
String nome = pessoa.nome;  
  
// idade agora contém 25  
int idade = pessoa.idade;
```

Também podemos chamar os métodos de um objeto usando o operador ponto, como no exemplo abaixo:

```
/* exibe "Nome: João" e  
 * "Idade: 25" na tela */  
pessoa.imprimirDados();
```

O código acima chama o método **imprimirDados** no objeto **pessoa**, que exibe o nome e a idade na tela.

Em resumo, uma classe é uma estrutura que encapsula dados e comportamentos relacionados, enquanto um objeto é uma instância dessa classe que contém seus próprios dados e comportamentos. As classes são usadas para definir objetos em Java, e esses objetos são criados a partir dessas classes e usados em diferentes partes do programa.





## Métodos Estáticos

Métodos estáticos são métodos que pertencem a uma classe e não a uma instância dessa classe. Eles são definidos com a palavra-chave **static** e podem ser chamados diretamente a partir do nome da classe, sem a necessidade de criar um objeto dessa classe.

Aqui está um exemplo de como definir um método estático em Java:

```
public class Calculadora {
    public static int somar(int a, int b)
    {
        return a + b;
    }
}
```

Neste exemplo, estamos definindo um método estático chamado `somar` na classe **Calculadora**. O método recebe dois argumentos inteiros **a** e **b**, e retorna a **soma** desses valores.

Podemos chamar o método estático diretamente a partir do nome da classe, sem a necessidade de criar um objeto da classe `Calculadora`. Por exemplo:

```
int resultado = Calculadora.somar(2, 3);
System.out.println(resultado);
```

Neste código, estamos chamando o método estático **somar** da classe Calculadora e armazenando o resultado na variável resultado. O método é chamado diretamente a partir do nome da classe, `Calculadora.somar`, sem a necessidade de criar um objeto da classe.

Os métodos estáticos são úteis quando queremos fornecer funcionalidade que não depende de um estado de objeto específico, como é o caso de métodos utilitários que realizam cálculos matemáticos simples, por exemplo. Também são usados para criar métodos de fábrica, que retornam uma nova instância de uma classe, ou para criar métodos auxiliares, que realizam operações úteis para uma classe ou aplicação.

*Faça suas anotações aqui...*



## Herança

Herança é um dos conceitos fundamentais da programação orientada a objetos (POO) e permite que uma classe herde atributos e métodos de outra classe. Na herança, uma classe é chamada de **classe derivada** ou **subclasse**, e a classe da qual ela herda é chamada de **classe base** ou **superclasse**.

Para implementar a herança em Java, usamos a palavra-chave **extends**. A sintaxe básica é a seguinte:

```
class Subclasse extends Superclasse {  
    // corpo da subclasse  
}
```

Neste exemplo, subclasse é a classe derivada e superclasse é a classe base. A subclasse herda todos os atributos e métodos públicos e protegidos da superclasse, mas não pode acessar os atributos privados.

Vamos supor que temos uma classe **Pessoa** que contém os atributos **nome** e **idade**, e um método **mostrarDados**. Podemos criar uma subclasse chamada **Funcionario** que herda da classe **Pessoa** e adiciona um novo atributo **salario** e um novo método **mostrarSalario**:

```

class Pessoa {
    String nome;
    int idade;

    public void mostrarDados() {
        System.out.println("Nome: " + nome);
        System.out.println("Idade: " + idade);
    }
}

class Funcionario extends Pessoa {
    double salario;

    public void mostrarSalario() {
        System.out.println("Salário: "
                           + salario);
    }
}

```

Neste exemplo, a classe Funcionario herda os atributos nome e idade da classe Pessoa e adiciona um novo atributo salario. A classe Funcionario também herda o método **mostrarDados** da classe Pessoa e adiciona um novo método **mostrarSalario**. Agora podemos criar um objeto Funcionario e chamar seus métodos:

```

Funcionario func = new Funcionario();
func.nome = "João";
func.idade = 30;
func.salario = 2000.0;

```

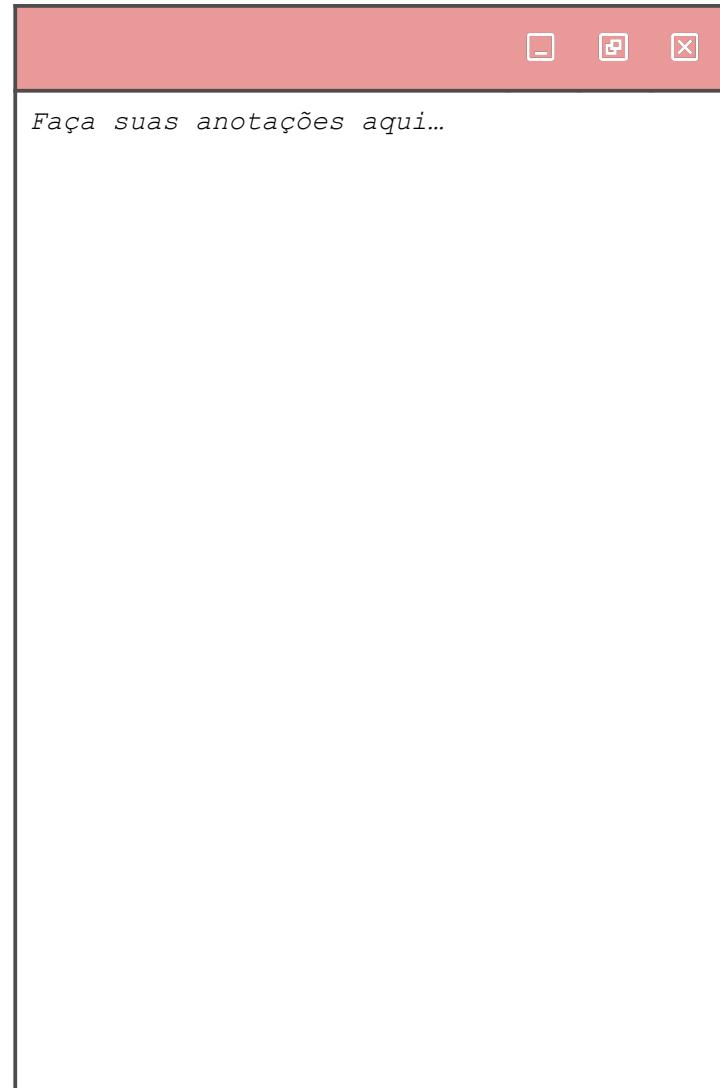
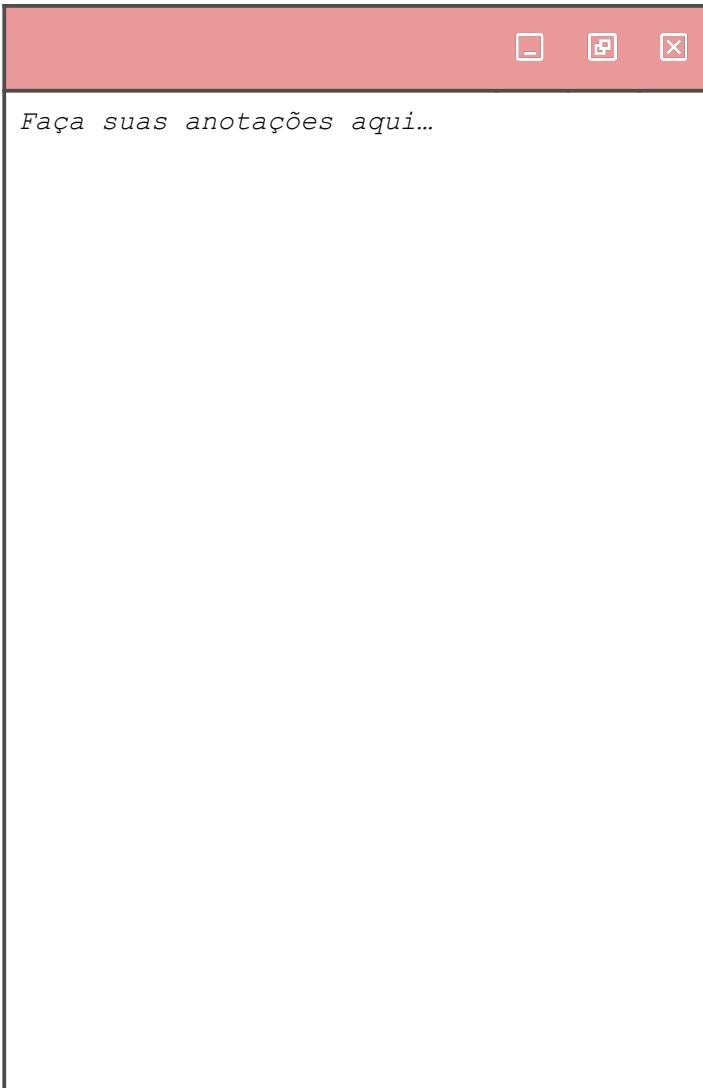
```

// exibe "Nome: João" e "Idade: 30"
func.mostrarDados();

// exibe "Salário: 2000.0"
func.mostrarSalario();

```

Neste código, estamos criando um objeto Funcionario chamado func, definindo seus atributos e chamando seus métodos. Observe que o método **mostrarDados** é herdado da classe Pessoa e o método **mostrarSalario** é definido na classe Funcionario. Isso mostra como a herança pode ser usada para reutilizar código e adicionar funcionalidade em classes derivadas.



## Polimorfismo

Polimorfismo é um conceito fundamental da programação orientada a objetos (POO) em Java, que permite que um objeto possa assumir várias formas ou comportamentos. Em Java, o polimorfismo é implementado através do uso de herança e interfaces.

O polimorfismo é uma forma de aproveitar a herança, permitindo que um objeto de uma classe mais específica (subclasse) possa ser tratado como um objeto de sua classe mais geral (superclasse). Isso significa que o mesmo método pode ter comportamentos diferentes em diferentes classes.

Por exemplo, considere uma hierarquia de classes onde a classe **Animal** é a superclasse e as classes **Cachorro**, **Gato** e **Passaro** são subclasses. Cada uma dessas classes tem um método **emitirSom**, que produz um som característico do animal correspondente.

Agora, se tivermos uma variável do tipo **Animal**, podemos atribuir a ela um objeto de qualquer uma dessas classes. Se chamarmos o método **emitirSom** nessa variável, o som correspondente ao tipo específico de animal será emitido, porque cada subclasse tem sua própria implementação desse método.

```
class Animal {
    public void emitirSom() {
        System.out.println("Som desconhecido");
    }
}

class Cachorro extends Animal {
    public void emitirSom() {
        System.out.println("Au Au");
    }
}

class Gato extends Animal {
    public void emitirSom() {
        System.out.println("Miau");
    }
}

class Passaro extends Animal {
    public void emitirSom() {
        System.out.println("Piupiu");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal1 = new Cachorro();
        Animal animal2 = new Gato();
        Animal animal3 = new Passaro();
```

```
animal1.emitirSom(); // Au Au  
animal2.emitirSom(); // Miau  
animal3.emitirSom(); // Piupiu  
}  
}
```

Observe que o método **emitirSom** foi chamado em cada um dos objetos, mas o som produzido depende do tipo de objeto (ou seja, de qual subclasse foi instanciada).

O polimorfismo é uma forma poderosa de escrever código reutilizável e genérico, permitindo que diferentes objetos possam ser tratados de forma semelhante, independentemente de sua classe específica. Ele também é usado em interfaces, onde vários objetos podem implementar a mesma interface, mas fornecer diferentes comportamentos para seus métodos.





## Overload e Override

Tanto o overload quanto o override são conceitos importantes na programação orientada a objetos (POO) em Java, e ambos envolvem a definição de métodos em classes.

O **overload (sobrecarga)** ocorre quando uma classe tem dois ou mais métodos com o **mesmo nome**, mas com **parâmetros diferentes**. A ideia é que esses métodos **realizem a mesma operação**, mas com **argumentos diferentes**. A sobrecarga permite que uma classe tenha vários métodos com o mesmo nome, facilitando a legibilidade e a manutenção do código. O overload é **resolvido em tempo de compilação**, selecionando o método correto com base nos tipos e quantidade de parâmetros fornecidos.

Por exemplo, podemos criar uma classe `Calculadora` com dois métodos `soma`, um que recebe dois inteiros e outro que recebe dois números de ponto flutuante. Assim, podemos usar o mesmo nome de método `soma`, mas ter um comportamento diferente com base nos argumentos fornecidos:

```
class Calculadora {
    public int soma(int a, int b) {
        return a + b;
    }
}
```

```
public double soma(double a, double b) {
    return a + b;
}
```

Já o **override (sobrescrita)** ocorre quando uma subclasse redefine um método que já está definido em sua superclasse. O objetivo é **substituir o comportamento original do método na superclasse por um novo comportamento na subclasse**. A sobrescrita permite que uma subclasse especialize o comportamento de um método herdado da superclasse. O override é **resolvido em tempo de execução**, selecionando o método correto com base no tipo do objeto que está sendo utilizado.

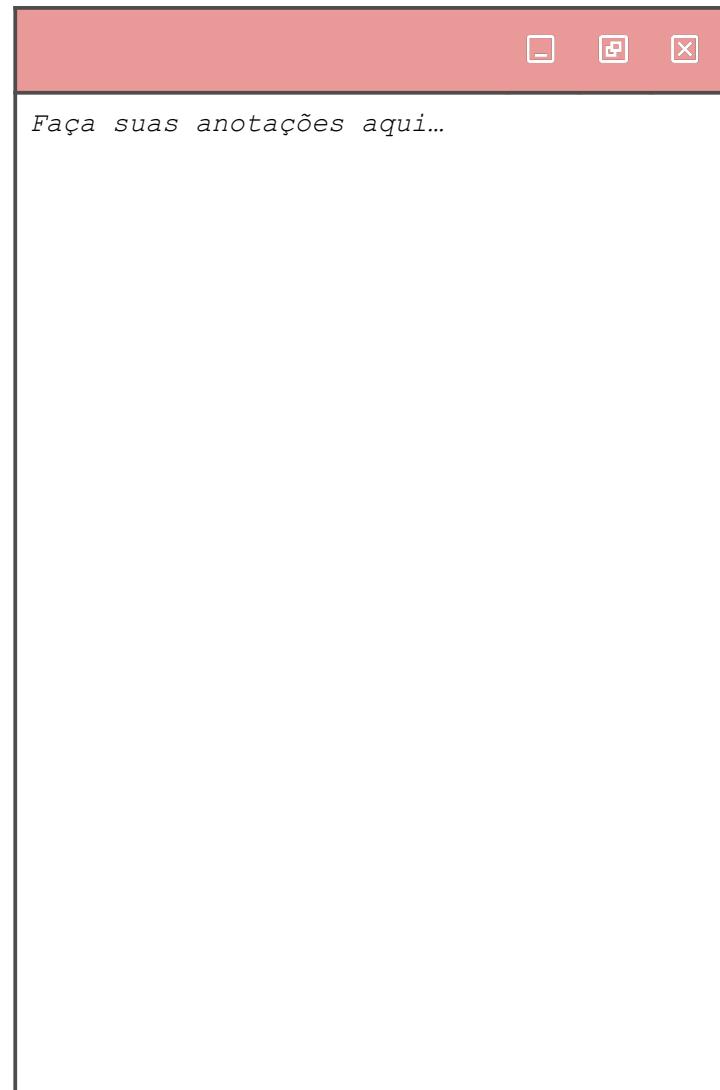
Por exemplo, podemos criar uma classe Animal com um método emitirSom, e em seguida criar uma subclasse Cachorro que sobrescreve o método emitirSom para imprimir "Au Au":

```
class Animal {
    public void emitirSom() {
        System.out.println("Som desconhecido");
    }
}

class Cachorro extends Animal {
    @Override
```

```
    public void emitirSom() {
        System.out.println("Au Au");
    }
}
```

Neste exemplo, a subclasse Cachorro sobrescreve o método emitirSom da superclasse Animal, fornecendo um novo comportamento para esse método. Ao chamar o método emitirSom em um objeto Cachorro, o resultado será "Au Au", mas se chamarmos o método em um objeto Animal genérico, o resultado será "Som desconhecido".



## Classes Abstratas e Interfaces

Em Java, classes abstratas e interfaces são mecanismos usados para implementar o conceito de abstração, que é fundamental na programação orientada a objetos. Ambas as abstrações fornecem uma maneira de definir um conjunto de métodos que podem ser implementados por classes concretas, mas com algumas diferenças fundamentais.

Uma **classe abstrata** é uma classe que **não pode ser instanciada** e **contém pelo menos um método abstrato**, que é um método sem corpo, sem implementação. Uma classe abstrata serve como um tipo de esqueleto para suas subclasses, definindo os métodos e as propriedades básicas que cada subclass deve ter, mas sem implementá-los completamente. **As subclasses da classe abstrata devem implementar os métodos abstratos**, fornecendo uma implementação completa para cada um deles.

Por exemplo:

```
public abstract class Animal {
    private String nome;

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

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

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

public abstract void emitirSom();
}

public class Cachorro extends Animal {
    public Cachorro(String nome) {
        super(nome);
    }

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

```
public class Gato extends Animal {
    public Gato(String nome) {
        super(nome);
    }
}
```

```

@Override
public void emitirSom() {
    System.out.println("Miau!");
}
}

```

Neste exemplo, a classe Animal é abstrata e contém um método abstrato emitirSom. As subclasses Cachorro e Gato estendem a classe Animal e implementam o método emitirSom, fornecendo sua própria implementação específica de cada tipo de animal.

**Interfaces** são semelhantes às classes abstratas, mas definem **apenas um conjunto de métodos que devem ser implementados pelas classes que as implementam**. Uma interface não contém nenhum código real e não pode ser instanciada diretamente, mas serve como uma especificação do conjunto de métodos que uma classe deve implementar para ser considerada compatível com a interface.

Por exemplo:

```

public interface Animal {
    String getNome();

    void setNome(String nome);
    void emitirSom();
}

```

```

public class Cachorro implements Animal {
    private String nome;

    public Cachorro(String nome) {
        this.nome = nome;
    }

    @Override
    public String getNome() {
        return nome;
    }

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

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

public class Gato implements Animal {
    private String nome;

    public Gato(String nome) {
        this.nome = nome;
    }
}

```

```

@Override
public String getNome() {
    return nome;
}

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

@Override
public void emitirSom() {
    System.out.println("Miau!");
}
}

```

Neste exemplo, a interface Animal define os métodos `getNome`, `setNome` e `emitirSom`, que devem ser implementados por qualquer classe que implemente a interface. As classes Cachorro e Gato implementam a interface Animal e fornecem uma implementação completa para cada um dos seus métodos. Observe que as classes implementam todos os métodos definidos na interface, pois isso é obrigatório.

As interfaces fornecem uma maneira de **definir contratos** que as classes devem cumprir para interagir com outras classes. Por exemplo, se tivermos um método que espera um parâmetro do tipo Animal,

podemos passar qualquer objeto que implemente a interface Animal para esse método, independentemente de qual classe específica seja. Isso é possível porque a interface garante que o objeto terá os métodos necessários implementados para cumprir o contrato definido pela interface.

**Uma classe pode implementar várias interfaces, mas só pode herdar de uma classe.** Isso significa que as interfaces são uma maneira útil de adicionar funcionalidade a uma classe sem herdar uma classe base adicional, o que pode criar problemas com herança múltipla. As interfaces também permitem que as classes definam comportamentos específicos, semelhantes às classes abstratas, mas sem fornecer implementações básicas para os métodos.



*Faça suas anotações aqui...*



*Faça suas anotações aqui...*

## Spring Framework

### Introdução

Spring Framework é um framework de código aberto para desenvolvimento de aplicativos em Java, lançado em 2002. Ele oferece uma ampla variedade de recursos e facilidades para desenvolvedores, com o objetivo de simplificar o processo de desenvolvimento e torná-lo mais eficiente.



O Spring Framework é baseado no padrão de **programação inversão de controle (IoC)** e **injeção de dependência (DI)**, o que significa que o controle do fluxo do programa é delegado a um **contêiner IoC**, que gerencia as dependências entre os componentes do sistema. Isso permite que os desenvolvedores se concentrem no desenvolvimento de componentes de negócios, enquanto o contêiner IoC gerencia a configuração e a injeção de dependências entre eles.

O Spring Framework é dividido em vários módulos, cada um dos quais fornece recursos para um aspecto específico do desenvolvimento de aplicativos Java. Alguns dos módulos mais importantes incluem:

**Spring Core:** contém as classes fundamentais do Spring Framework, incluindo o **contêiner IoC** e o mecanismo de **injeção de dependência**.

**Spring MVC:** fornece recursos para desenvolvimento de aplicativos web baseados no **padrão de arquitetura MVC (Model-View-Controller)**.

**Spring Data:** facilita o acesso a **bancos de dados** e reduz a quantidade de código necessário para implementar operações de persistência.

**Spring Security:** fornece recursos para **autenticação** e **autorização** em aplicativos baseados em Spring.

**Spring Boot:** um módulo que torna mais fácil a configuração e o desenvolvimento de aplicativos baseados em Spring.

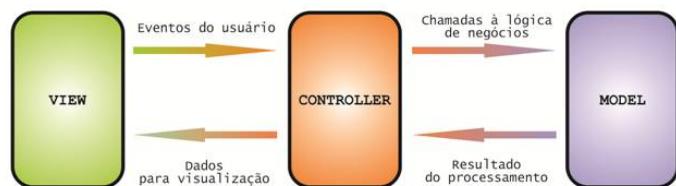
O Spring Framework é amplamente utilizado em projetos de desenvolvimento de aplicativos Java de todos os tipos, desde aplicativos corporativos até aplicativos web e aplicativos móveis. Sua abordagem modular e flexível torna-o uma escolha popular para projetos de todos os tamanhos. Além disso, sua comunidade de desenvolvedores ativa e recursos de suporte tornam mais fácil para os desenvolvedores encontrar soluções para problemas e aprender a usar o framework de maneira eficiente.



## Padrão de Projeto e Spring MVC

Padrões de projeto são **soluções comprovadas para problemas recorrentes** que ocorrem durante o desenvolvimento de software. Eles representam uma abordagem testada e comprovada para projetar e implementar soluções de software que sejam **escaláveis, flexíveis e de fácil manutenção**.

O padrão **Model-View-Controller (MVC)** é um dos padrões de projeto mais conhecidos e utilizados na programação de aplicativos web. Ele divide uma aplicação em três partes principais: o **modelo (model)**, que representa os dados e a lógica de negócios; a **visualização (view)**, que apresenta os dados ao usuário; e o **controlador (controller)**, que gerencia as solicitações do usuário e faz a comunicação entre o modelo e a visualização.



O Spring MVC é um framework baseado no padrão MVC que é integrado ao Spring Framework. Ele fornece recursos para desenvolvimento de aplicativos web baseados no padrão de arquitetura MVC. O

Spring MVC ajuda os desenvolvedores a criar aplicativos web escaláveis e flexíveis, com uma estrutura organizada e clara para o código do aplicativo.

O Spring MVC fornece um modelo de **desenvolvimento orientado a anotações**, que torna a configuração e a implementação do padrão MVC simples e intuitiva. O modelo de anotações do Spring MVC permite que os desenvolvedores especifiquem como **as solicitações do usuário devem ser mapeadas aos controladores**, como **os parâmetros de solicitação devem ser vinculados aos objetos do modelo**, e **como a visualização deve ser renderizada**.

O Spring MVC também fornece recursos para lidar com erros e exceções, validação de formulários, internacionalização, e muito mais. Além disso, o Spring MVC é altamente integrado com outros módulos do Spring Framework, como o Spring Security e o Spring Data.

Em resumo, o Spring MVC é uma das estruturas mais populares e poderosas para desenvolvimento de aplicativos web em Java, fornecendo um conjunto abrangente de recursos e facilidades que permitem aos desenvolvedores criar aplicativos web escaláveis e flexíveis, com uma estrutura organizada e clara para o código do aplicativo.



## REST APIs

REST (*Representational State Transfer*) é um estilo de arquitetura de software que define **um conjunto de restrições** para a criação de APIs (*Application Programming Interfaces*) na web. Uma API RESTful é uma API baseada em REST, que segue essas **restrições e convenções**.

Uma API RESTful utiliza o protocolo HTTP para transferir dados entre o cliente e o servidor. As solicitações são feitas utilizando os métodos HTTP, como **GET, POST, PUT, DELETE**, etc. As respostas são retornadas em **formato JSON**.

As APIs RESTful são projetadas para serem escaláveis e fáceis de usar. Elas são amplamente utilizadas para criar serviços da web para aplicativos móveis, sites e outras aplicações que precisam de acesso a dados e recursos da web.

As principais características das APIs RESTful são:

- **Stateless:** cada solicitação contém todas as informações necessárias para processá-la. Não há estado armazenado no servidor entre as solicitações.
- **Identificação de recursos:** cada recurso (ou conjunto de recursos) é identificado por um URI (*Uniform Resource Identifier*).

- **Métodos HTTP:** os métodos HTTP são usados para indicar a ação que deve ser executada no recurso identificado pelo URI.
- **Representação dos recursos:** a representação dos recursos é transferida entre o cliente e o servidor em um formato padrão, como JSON.
- **Interface uniforme:** uma interface uniforme é definida para interagir com os recursos, facilitando a utilização e evolução da API.

Ao criar uma API RESTful, é importante seguir as boas práticas e convenções para garantir que a API seja fácil de usar e escalável. Algumas dessas práticas incluem:

- Utilizar os métodos HTTP apropriados para cada ação (GET para obter informações, POST para criar novos recursos, PUT para atualizar recursos, DELETE para remover recursos).
- Utilizar URIs claras e descriptivas para identificar os recursos (exemplos: "/clientes" para obter ou manipular todos os clientes e "/clientes/1" para obter ou manipular um cliente específico).
- Utilizar códigos de status HTTP apropriados para indicar o resultado da solicitação (exemplos: 200 para OK, 404 para não encontrado).
- Utilizar o formato JSON para representar os dados dos recursos.
- Utilizar a autenticação e autorização para proteger a API contra acesso não autorizado.

Em resumo, uma API RESTful é uma interface de programação de aplicativos que segue um conjunto de restrições e convenções para garantir que seja fácil de usar e escalável. Ela é amplamente utilizada para criar serviços da web para aplicativos móveis, sites e outras aplicações que precisam de acesso a dados e recursos da web.



## Criando um projeto com Spring Boot

Para criar um projeto Spring Boot usando o padrão MVC e REST API, você pode seguir os seguintes passos:

### Passo 1: Configurar o ambiente

Certifique-se de que o Java JDK esteja instalado em sua máquina. Você também pode usar uma IDE, como Visual Studio Code, o IntelliJ IDEA ou o Eclipse, para desenvolver o projeto.

### Passo 2: Criar um novo projeto

Crie um novo projeto Spring Boot usando o Spring Initializr, através do link <https://start.spring.io/>.

Certifique-se de selecionar as seguintes dependências:

- Spring Web
- Spring Data JPA
- Spring Boot DevTools
- H2 Database (opcional, dependendo do banco de dados que você deseja usar)

### Passo 3: Configurar as entidades

Defina as **entidades** do seu projeto e configure o **relacionamento** entre elas, se houver. Você também pode usar anotações JPA - **@Entity**, **@OneToOne** etc - para mapear as entidades para tabelas do banco de dados.

### Passo 4: Configurar os repositórios

Crie **interfaces** de repositório para as entidades e defina as operações que você deseja executar no banco de dados.

### Passo 5: Configurar os controladores

Crie controladores para definir os endpoints da API RESTful. Use as anotações **@RestController** e **@RequestMapping** para configurar os endpoints e os métodos HTTP (GET, POST, PUT, DELETE) que você deseja expor.

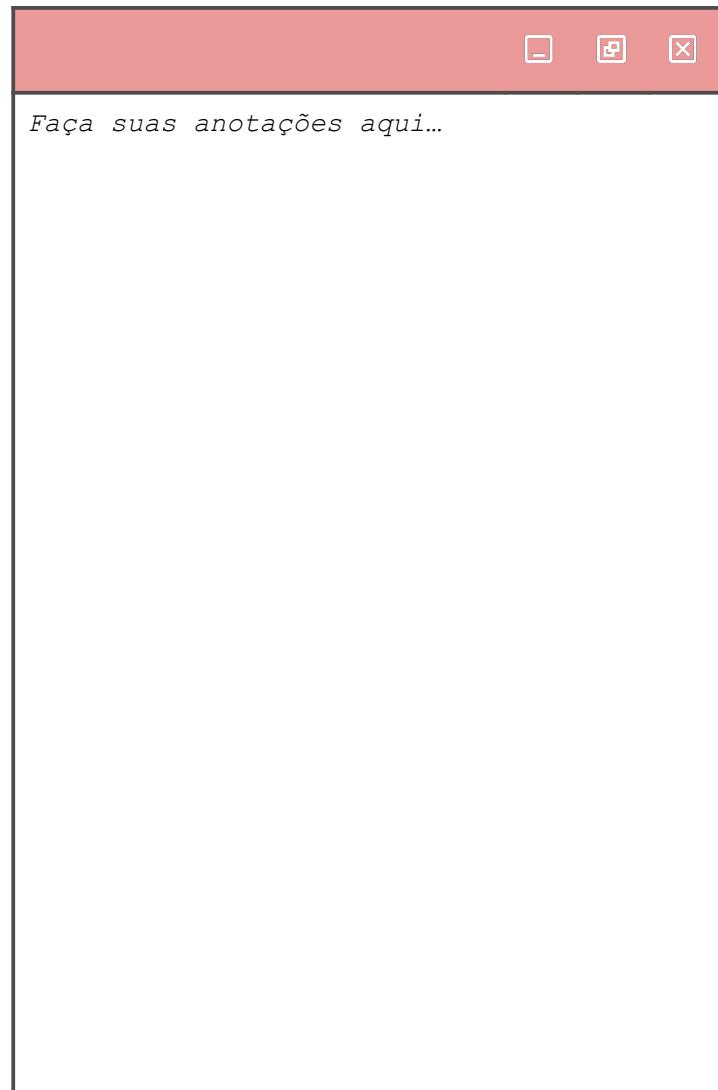
### Passo 6: Implementar a lógica de negócios

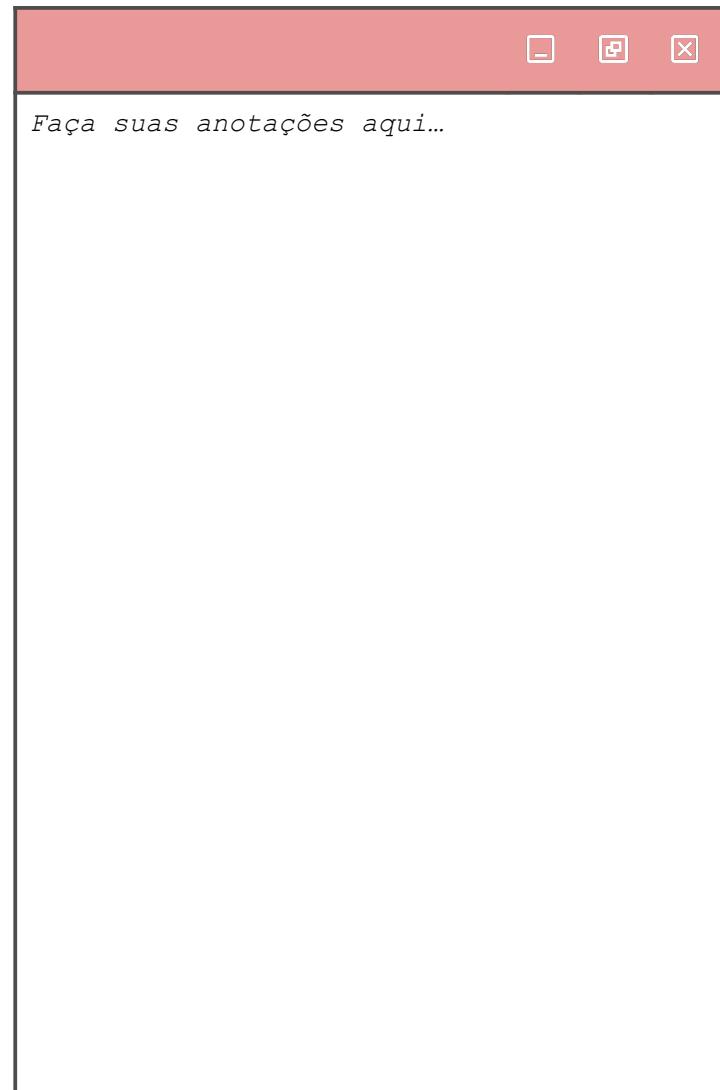
Implemente a lógica de negócios em classes de serviço e injete os repositórios relevantes nessas classes. Os controladores devem chamar os métodos dos serviços para realizar as operações no banco de dados.

### Passo 7: Executar o aplicativo

Compile e execute o aplicativo usando o **Maven** ou o **Gradle**. Teste os endpoints da API RESTful usando um cliente REST, como o Postman.

Com esses passos, você deve ter criado um projeto Spring Boot usando o padrão MVC e REST API. A partir daqui, você pode expandir e personalizar o projeto de acordo com as necessidades da sua aplicação.







[www.treinarecife.com.br](http://www.treinarecife.com.br)



SAIBA MAIS EM:  
[www.alexandrejunior.dev](http://www.alexandrejunior.dev)