<LAB365>

JAVA AVANÇADO - PARTE 1

<LAB365> SENAI

AGENDA

- Introdução e conceitos de POO
- Classes, Objetos, Atributos e Métodos
- Construtores

INTRODUÇÃO

- Programação Orientada a Objetos (POO) é um paradigma de programação que organiza o código em torno de "objetos" que podem conter dados (conhecidos como atributos) e código (conhecido como métodos). Um dos princípios fundamentais da POO é a abstração.

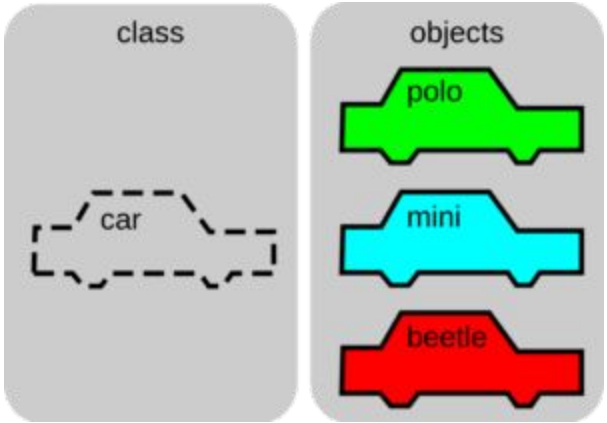
- Esse paradigma foi criado para nos aproximar ao máximo da vida real.

INTRODUÇÃO

- **Simplificação de Complexidade:** Permite lidar com sistemas complexos de forma mais gerenciável, ao focar apenas nos aspectos relevantes e ignorar os detalhes desnecessários.
- **Facilita a Reutilização de Código:** Ao criar abstrações de objetos e suas funcionalidades, é mais fácil reutilizar essas abstrações em diferentes partes de um programa ou mesmo em diferentes programas.
- **Promove a Modularidade:** Ajuda a dividir um sistema em módulos independentes, cada um com sua própria responsabilidade bem definida, facilitando a manutenção e o desenvolvimento contínuo do código.
- **Foco no Comportamento:** Ao invés de se preocupar com os detalhes internos de como um objeto realiza uma tarefa, os usuários de uma abstração apenas precisam saber como interagir com ela e quais resultados esperar.

CLASSES E OBJETOS

Classe
A classe é abstrata, como um molde. É na classe que definimos as características e os comportamentos.



Objeto
O objeto é criado a partir da classe, utilizando-a como um molde para a criação do objeto com suas características e comportamentos.

Fonte da imagem: https://www.alura.com.br/artigos/poo-programacao-orientada-a-objetos

CLASSES E OBJETOS

- Considere um objeto "Carro". Em uma aplicação, podemos abstrair um carro para incluir apenas os aspectos relevantes, como modelo, cor e velocidade atual.

- O código que utiliza essa abstração pode interagir com o carro para acelerar, frear, virar etc., sem precisar conhecer todos os detalhes internos do funcionamento do carro, como o motor, a transmissão ou o sistema de direção.

```
public class Carro {
    private String modelo;
    private String cor;
    private int velocidade;

    // Métodos para acelerar, frear, virar, etc.
}
```

PILARES

- Abstração
- Encapsulamento
- Herança
- Polimorfismo

PILARES (Abstração)

- Abstração é o processo de identificar as características essenciais de um objeto do mundo real e representá-las de forma simplificada no contexto de um programa de computador. Em outras palavras, abstração significa ignorar os detalhes não essenciais e focar apenas nos aspectos relevantes de um objeto.

- É necessário abstrair o que o objeto realizará e quais suas características, para que seja desenvolvida sua classe e posteriormente criado seus objetos.

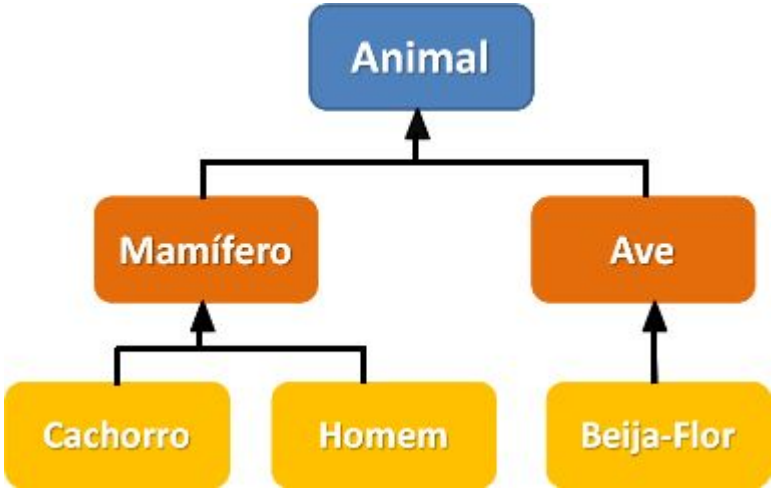
PILARES (Encapsulamento)

- O encapsulamento agrega segurança ao objeto.

- Por exemplo: é possível esconder todas as características do objeto e dar publicidade somente aos métodos necessários, dessa forma, o método pode mascarar alguma característica quando necessário retorná-la.

PILARES (Herança)

- Quando a ClasseA é um tipo de ClasseB, é porque a ClasseA herda as características da ClasseB.



PILARES (Polimorfismo)

- Quando dois objetos de duas classes diferentes possuem o mesmo método, porém executam de formas diferentes e tem o mesmo efeito.

- Um exemplo disso pode ser dado com um carro e uma bicicleta, que possuem a função "acelerar". Essa função é feita de forma completamente diferente em ambos meios de locomoção, sendo um por combustão e outro por meio exercício.

PILARES

Na Programação Orientada a Objetos lidamos com uma representação de um objeto real.

Precisamos imaginar o que esse objeto realizará no sistema considerando:

- *Uma identidade* : forma de identificar o que o objeto é.
- *Características* : todo objeto possui características que o definem.
- *Ações* : quais funções o objeto exerce ou executa.

PILARES

Traduzindo para as classes:

- *Identidade* : Nome da classe
- *Características* : Atributos
- *Ações* : Métodos

CLASSES

- Classes são os templates dos Objetos, e geralmente representam algum conceito ou objeto do mundo real. É nelas que definimos os Métodos e Atributos. Tanto os métodos, quanto os atributos, quanto as Classes tem Modificadores de Acesso

- Classes são a definição central de tudo em uma linguagem orientada a objetos. Pois são onde temos todos os "ingredientes" dos objetos que virão a ser criados

- Por exemplo, se estivermos construindo um sistema para uma biblioteca, poderíamos ter uma classe "Livro" para representar os livros presentes na biblioteca. Uma classe é basicamente um modelo para criar objetos.

CLASSES

```
public class Livro {
    // Atributos (características do livro)
    String titulo;
    String autor;
    int anoPublicacao;

    // Métodos (comportamentos do livro)
    public void exibirDetalhes() {
        System.out.println("Título: " + titulo);
        System.out.println("Autor: " + autor);
        System.out.println("Ano de Publicação: " + anoPublicacao);
    }
}
```

OBJETOS

- Objetos são ocorrências de uma classe. Se pensarmos na classe Caneta, podemos ter o objeto BIC azul, BIC preta, Pentel Gel vermelha ou uma Momblac com detalhes em ouro. Cada exemplo anterior é o que chamamos de objetos. Todos terão os atributos e métodos da classe Caneta

- Sempre que utilizamos a palavra chave **new**, criamos um novo objeto dentro da memória do Java.

OBJETOS

```
public class Principal {
    public static void main(String[] args) {
        // Criando objetos da classe Livro
        Livro livro1 = new Livro();
        Livro livro2 = new Livro();

        // Configurando os atributos dos objetos
        livro1.titulo = "Dom Casmurro";
        livro1.autor = "Machado de Assis";
        livro1.anoPublicacao = 1899;

        livro2.titulo = "1984";
        livro2.autor = "George Orwell";
        livro2.anoPublicacao = 1949;

        // Chamando métodos dos objetos
        livro1.exibirDetalhes();
        livro2.exibirDetalhes();
    }
}
```

MÉTODOS

- **Métodos são ações que um objeto pode realizar**. Nós já vimos isso com o Scanner e o *scanner.nextInt()*. Vimos isso com o Math.max(int1, int2).

- Vamos pensar em métodos como as ações que um objeto pode realizar, seja essa ação uma conta, um print, uma troca de valores. Esses métodos podem utilizar dados externos ao objeto ou podem utilizar os atributos do objeto para realizarem as ações requeridas.

```
public class Pessoa {
   String nome;
   Integer idade;
   String paisOrigem;

   public void printPessoa(){
      System.out.println("Nome:" +this.nome+", Idade:" +this.idade+", Pais de Origem: "
+this.paisOrigem);
   }
}
```

MÉTODOS

- Podemos acessar o método que existe dentro de um objeto através da chamada do método:

  pessoa.printPessoa();

- Métodos tem um tipo de retorno, da mesma forma que temos um tipo de variável um método pode retornar diversos tipos: Objetos de Classe, Integer, Double, String, void, etc.

- *Void* é uma tipo de retorno que não tem valor algum associado a ele, assim podemos executar o método e ele nunca retorna valor.

- Podemos executar qualquer tipo de validação ou loop dentro de um método, o que nos permite utilizar os dados da classe de diversas formas.

MÉTODOS

- Agora vamos falar de parâmetros dos métodos.
- Esses são forma de entrar dados no método, cada parâmetro atua como uma variável dentro do método, e quando chamamos o método podemos preencher essas “variáveis” com valores que vão ser executados dentro do método.

```
public Integer soma (Integer valor1, Integer valor2){
        return valor1 + valor2;
}
```

ATRIBUTOS

- **Atributos são valores definidos dentro de uma classes**, e cada objetos irá ser responsável por preencher esse valores.

- Uma classe deve ter atributos esses atributos vão ser preenchidos em cada objeto. Sendo assim, nós podemos ter um classe de pessoas e um objeto André, ou conhecido, ou qualquer outra pessoa que podemos criar. Vamos ver isso na prática

```
public static void main(String[] args) {
   Pessoa pessoa = new Pessoa();

   pessoa.nome = "André";
   pessoa.idade = 22;
   pessoa.paisOrigem = "Brasil";

}
```

Hands On

🙌 Mão na massa...

Hands On

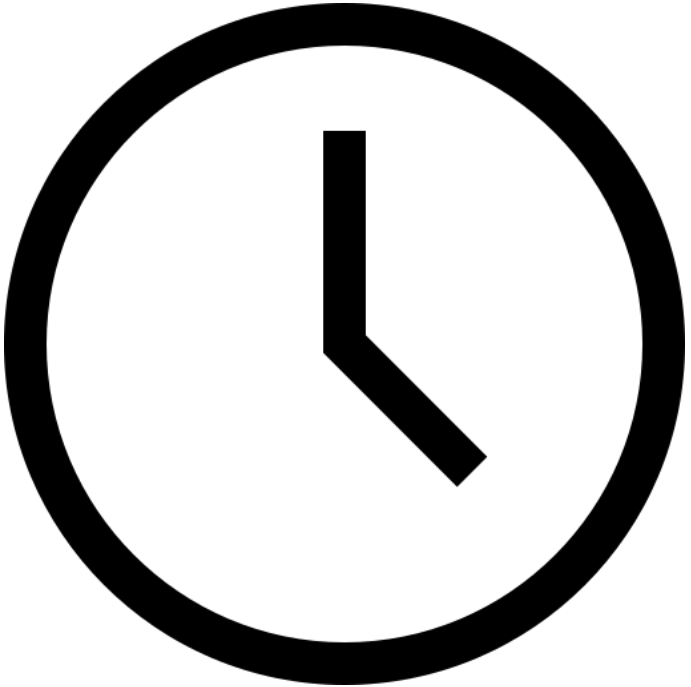
- Crie um sistema para monitoramento das áreas verdes da cidade de Joinville. O sistema deve permitir o registro e a consulta de informações sobre diferentes áreas verdes.

INTERVALO!

Finalizamos o nosso primeiro período de hoje. Que tal descansar um pouco?!

Nos vemos em 20 minutos.

Início: 20:20
Retorno: 20:40



CONSTRUTOR

- **Construtores**
  - São métodos especiais utilizados para inicializar objetos
  - Eles são chamados no momento da criação do objeto e têm o mesmo nome da classe
- **Tipos**
  - Construtor Padrão: É fornecido pela própria JVM se nenhum construtor é explicitamente definido na classe.
  - Não tem parâmetros e inicializa os membros do objeto com valores padrão.

```
public class Exemplo {
    // Construtor padrão implícito
}
```

CONSTRUTOR

- **Tipos**
  - Construtor Parametrizado: Permite a inicialização de objetos com valores específicos fornecidos.
  - Requer que parâmetros sejam passados durante a criação do objeto.

```
public class Exemplo {
    private int valor;

    // Construtor parametrizado
    public Exemplo(int valor) {
        this.valor = valor;
    }
}
```

CONSTRUTOR

- **Sobrecarga de Construtores**: Uma classe pode ter mais de um construtor, diferenciados pela lista de parâmetros.
- ***this* Keyword**: Usada para referenciar a instância atual do objeto dentro de um construtor ou método.

```
public class Livro {
    private String titulo;
    private String autor;

    // Construtor padrão
    public Livro() {
        titulo = "Desconhecido";
        autor = "Desconhecido";
    }
```

```
    // Construtor parametrizado
    public Livro(String titulo,
String autor) {
        this.titulo = titulo;
        this.autor = autor;
    }
}
```

<LAB365>

JAVA AVANÇADO - PARTE 2

<LAB365> SENAI

AGENDA

- Modificadores de Acesso
- Encapsulamento
- Sobrecarga

MODIFICADORES DE ACESSO

- Controlar a visibilidade e o acesso às variáveis, métodos e construtores de uma classe.
- Fundamental para o princípio de encapsulamento em Programação Orientada a Objetos (POO)

MODIFICADORES DE ACESSO

- ***private***
  - Acesso permitido apenas dentro da própria classe.
  - Ideal para variáveis de instância e métodos que não devem ser acessíveis externamente.

    private int idade;

MODIFICADORES DE ACESSO

- ***default (sem modificador)***
  - Acesso permitido apenas dentro do mesmo pacote.
  - Menos restritivo que *private*, mas sem acesso para classes de outros pacotes.

    int numero;

MODIFICADORES DE ACESSO

- ***protected***
  - Acesso permitido dentro do mesmo pacote e por subclasses em pacotes diferentes.
  - Útil quando se quer permitir herança de propriedades e métodos.
    protected String nome;

MODIFICADORES DE ACESSO

- ***public***
  - Acesso permitido de qualquer lugar.
  - Deve ser usado com cautela para evitar exposição desnecessária de dados.

    public void exibirInfo() {

    }

MODIFICADORES DE ACESSO

- **Tabela de Visibilidade**

Modificador	Classe	Pacote	Subclasse	Mundo
private	✅	❌	❌	❌
default	✅	✅	❌	❌
protected	✅	✅	✅	❌
public	✅	✅	✅	✅

MODIFICADORES DE ACESSO

- **Exemplo**

```
public class Conta {
    private double saldo; // Acesso somente dentro de Conta
    int numeroConta; // Acesso dentro do pacote
    protected String titular; // Acesso em subclasses
    public void depositar(double valor) { // Acesso de qualquer lugar
        // ...
    }
}
```

ENCAPSULAMENTO

- Encapsulamento é um dos quatro pilares fundamentais da Programação Orientada a Objetos (POO).
- Permite que os dados de um objeto sejam ocultados e protegidos de acesso externo direto.
    - Para controlar como os dados são acessados ou modificados.
    - Para proteger o estado interno do objeto contra alterações indesejadas.
    - Para flexibilizar a manutenção e evolução do código sem quebrar o uso por outras partes do programa.

ENCAPSULAMENTO

- Como implementar?
  - Utilize modificadores de acesso (*private*, *protected*) para restringir o acesso direto aos membros da classe.
  - Forneça métodos públicos (*getters e setters*) para permitir a manipulação segura das propriedades.

ENCAPSULAMENTO

- **Exemplo**

```
public class ContaBancaria {
        private double saldo; // Acesso restrito
          // Getter para saldo
          public double getSaldo() {
              return saldo;
          }
          // Setter para saldo
          public void depositar(double valor) {
              if (valor > 0) {
                  saldo += valor;
              }
          }
}
```

ENCAPSULAMENTO

- **Recomendações**
  - Uso de Modificadores de Acesso: Utilizar os modificadores de acesso corretos para variáveis e métodos é essencial. Variáveis devem ser, na maioria das vezes, privadas (*private*), enquanto métodos que são a interface pública da classe podem ser públicos (*public*).
  - Métodos Acessores e Mutantes: Para cada variável de instância privada, deve-se fornecer métodos acessores (*getters*) para consultar o valor da variável e mutantes (*setters*) para modificar o valor, se necessário.
  - Validação de Dados: Os métodos mutantes devem garantir a valididade dos dados antes de modificar qualquer variável de instância. Isso ajuda a manter a integridade dos dados e a consistência do estado do objeto.

ENCAPSULAMENTO

- **Recomendações**
  - Imutabilidade quando Possível: Se um objeto não precisa ter seu estado alterado após a criação, deve-se torná-lo imutável. Isso é feito não fornecendo métodos *setters* e marcando as variáveis de instância como *final*.
  - Minimizar a Visibilidade: Deve-se sempre minimizar a visibilidade das classes, interfaces, métodos e variáveis de instância para o menor nível necessário para que funcionem conforme o esperado.

Java: Como Programar. DEITEL, PAUL.

SOBRECARGA

- **Sobrecarga de Métodos (Overloading)**
  - Sobrecarga permite que métodos com o mesmo nome tenham diferentes listas de parâmetros (assinaturas).
  - É uma forma de polimorfismo que permite múltiplas versões de um método na mesma classe.

SOBRECARGA

- **Como funciona?**
  - A JVM identifica o método correto a ser chamado não pelo nome, mas pelos parâmetros que são passados na chamada do método.
  - A sobrecarga é determinada pelo número e tipo dos argumentos.
  - O retorno do método e os modificadores de acesso não afetam a sobrecarga.

MATERIAL COMPLEMENTAR

- Loiane Groner

- Livro: Java: Como Programar. DEITEL, PAUL.

<LAB365>

JAVA AVANÇADO - PARTE 3

AGENDA

- Relacionamento entre classes
- Métodos da Classe ArrayList

RELACIONAMENTO ENTRE CLASSES

- No Java, o relacionamento entre classes define como elas interagem e colaboram entre si. Existem três principais tipos de relacionamentos:
  - Associação
  - Agregação
  - Composição
  - Dependência

RELACIONAMENTO ENTRE CLASSES

- **Associação**
  - Representa uma relação onde uma classe usa outra, mas sem dependência direta.
  - Exemplo: Um **Aluno** estuda em uma **Escola**.

```
class Escola {  2 usages
    String nome;  no usages
}

class Aluno{  no usages
    String nome;  1 usage
    Escola escola;  1 usage

    Aluno(String nome, Escola escola){  no usages
        this.nome = nome;
        this.escola = escola;
    }
}
```

RELACIONAMENTO ENTRE CLASSES

- **Agregação**
  - É um tipo de associação onde uma classe contém outra, mas a existência da classe contida não depende da classe principal.
  - Exemplo: Uma **Turma** tem vários **Alunos**, mas um **Aluno** pode existir sem estar em uma **Turma**.

RELACIONAMENTO ENTRE CLASSES

```java
class Aluno{  2 usages
    String nome;  1 usage

    Aluno(String nome){  no usages
        this.nome = nome;
    }
}

class Turma{  no usages
    String nome;  1 usage
    List<Aluno> alunos; // Lista de alunos (agregação)  1 usage

    Turma(String nome, List<Aluno> alunos) {  no usages
        this.nome = nome;
        this.alunos = alunos;
    }
}
```

RELACIONAMENTO ENTRE CLASSES

- **Composição**
  - Semelhante à agregação, mas com dependência mais forte: se a classe principal for destruída, a classe contida também será.
  - Exemplo: Um **Carro** tem um **Motor**, e o motor não faz sentido existir sem o carro.

RELACIONAMENTO ENTRE CLASSES

```java
class Motor{  2 usages
    String tipo;  1 usage

    Motor(String tipo){  1 usage
        this.tipo = tipo;
    }
}

class Carro{  no usages
    String modelo;  1 usage
    Motor motor;  1 usage

    public Carro(String modelo, String tipoMotor) {  no usages
        this.modelo = modelo;
        this.motor = new Motor(tipoMotor);
    }
}
```

RELACIONAMENTO ENTRE CLASSES

- **Dependência**
  - Uma classe usa outra temporariamente, sem mantê-la como atributo.
  - Exemplo: Uma **Pessoa** usa um **Celular**.

RELACIONAMENTO ENTRE CLASSES

```
class Celular {  1 usage
    void ligar() {  1 usage
        System.out.println("Chamando...");
    }
}

class Pessoa {  no usages
    void fazerChamada(Celular celular) { // Dependência  no usages
        celular.ligar();
    }
}
```

MÉTODOS DA CLASSE ARRAYLIST

- A classe ArrayList em Java faz parte do pacote java.util e oferece vários métodos avançados para manipulação de listas.
  - ensureCapacity(int minCapacity)
  - trimToSize()
  - subList(int fromIndex, int toIndex)
  - replaceAll(UnaryOperator<E> operator)
  - removeIf(Predicate<E> filter)
  - sort(Comparator<E> c)
  - forEach(Consumer<E> action)

MÉTODOS DA CLASSE ARRAYLIST

- ensureCapacity(int minCapacity)
  - Garante que a lista tenha a capacidade mínima especificada para evitar realocações frequentes.

MÉTODOS DA CLASSE ARRAYLIST

```java
public class Test {
    public static void main(String[] args) {
        ArrayList<Integer> numeros = new ArrayList<>();
        numeros.ensureCapacity( minCapacity: 100); // Garante capacidade mínima de 100 elementos

        for (int i = 0; i < 50; i++) {
            numeros.add(i);
        }

        System.out.println("Tamanho atual: " + numeros.size());
    }
}
```

MÉTODOS DA CLASSE ARRAYLIST

- trimToSize()
  - Reduz a capacidade da ArrayList para corresponder ao número de elementos armazenados, economizando memória.

MÉTODOS DA CLASSE ARRAYLIST

```java
public class Test {
    public static void main(String[] args) {
        ArrayList<String> lista = new ArrayList<>( initialCapacity: 100);
        lista.add("Java");
        lista.add("Spring");
        lista.add("Hibernate");

        System.out.println("Capacidade antes de trimToSize(): " + lista.size());
        lista.trimToSize(); // Remove espaço extra
    }
}
```

MÉTODOS DA CLASSE ARRAYLIST

- subList(int fromIndex, int toIndex)
  - Retorna uma visão da lista dentro do intervalo especificado.

MÉTODOS DA CLASSE ARRAYLIST

```java
public class Test {
    public static void main(String[] args) {
        ArrayList<String> nomes = new ArrayList<>();
        nomes.add("Alice");
        nomes.add("Bob");
        nomes.add("Charlie");
        nomes.add("David");
        nomes.add("Eve");

        List<String> subLista = nomes.subList(1, 4); // Pegando elementos de índice 1 a 3
        System.out.println(subLista);
    }
}
```

MÉTODOS DA CLASSE ARRAYLIST

- replaceAll(UnaryOperator<E> operator)
  - Substitui cada elemento da lista aplicando uma função.

MÉTODOS DA CLASSE ARRAYLIST

```java
public class Test {
    public static void main(String[] args) {
        ArrayList<Integer> numeros = new ArrayList<>();
        numeros.add(1);
        numeros.add(2);
        numeros.add(3);
        numeros.add(4);

        // Multiplica cada número por 2
        numeros.replaceAll(n -> n * 2);

        System.out.println(numeros);
    }
}
```

MÉTODOS DA CLASSE ARRAYLIST

- removeIf(Predicate<E> filter)
  - Remove elementos da lista que satisfaça um critério.

MÉTODOS DA CLASSE ARRAYLIST

```java
public class Test {
    public static void main(String[] args) {
        ArrayList<Integer> numeros = new ArrayList<>();
        for (int i = 1; i <= 10; i++) {
            numeros.add(i);
        }

        // Remove números pares
        numeros.removeIf(n -> n % 2 == 0);

        System.out.println(numeros);
    }
}
```

MÉTODOS DA CLASSE ARRAYLIST

- sort(Comparator<E> c)
  - Ordena os elementos com um comparador personalizado.

MÉTODOS DA CLASSE ARRAYLIST

```java
public class Test {
    public static void main(String[] args) {
        ArrayList<String> nomes = new ArrayList<>();
        nomes.add("Maria");
        nomes.add("Carlos");
        nomes.add("Ana");
        nomes.add("Bruno");

        // Ordenação personalizada (ordem inversa)
        nomes.sort(Comparator.reverseOrder());

        System.out.println(nomes);
    }
}
```

MÉTODOS DA CLASSE ARRAYLIST

- forEach(Consumer<E> action)
  - Executa uma ação para cada elemento da lista.

MÉTODOS DA CLASSE ARRAYLIST

```
public class Test {
    public static void main(String[] args) {
        ArrayList<String> frutas = new ArrayList<>();
        frutas.add("Maçã");
        frutas.add("Banana");
        frutas.add("Laranja");

        // Imprime cada fruta na lista
        frutas.forEach(fruta -> System.out.println("Fruta: " + fruta));
    }
}
```

AVALIAÇÃO DOCENTE

O que você está achando das minhas aulas neste conteúdo?

Clique aqui ou escaneie o QRCode ao lado para avaliar minha aula.

Sinta-se à vontade para fornecer uma avaliação sempre que achar necessário.

<LAB365>