Lista de Linguagem de Programação Orientada a Objetos

Linguagem de Programação Orientada a Objetos

FACOM – UFMS

Prof. Mariana Caravanti de Souza

Construtores

1. Você está modelando um sistema simples para gerenciar **veículos** em uma locadora. Cada veículo tem informações básicas, mas tipos específicos de veículos podem ter atributos adicionais.

a. Crie a classe Veiculo com: String marca, String modelo, int ano

b. Crie um construtor padrão (sem prâmetros) que inicializa os atributos com valores genéricos. Por exemplo:

marca = “Desconhecida”

modelo = “Genérico”

ano = 2000

c. Crie um construtor com parâmetros para inicializar todos os atributos.

d. Crie um método toString para exibir as informações do veículo.

2. Crie a classe Carro que herda de Veiculo. Ela possui os seguintes atributos adicionais:

- int portas

a. Crie um construtor que receba apenas portas e use o construtor padrão da superclasse

b. Crie um construtor que receba marca, modelo, ano, portas e chame o construtor parametrizado da superclasse Veiculo usando super(...)

c. Sobrescreva o método toString para que o carro também exiba o número de portas

3. Crie a classe Moto, que também herda de Veículo. Ela possui os seguintes atributos adicionais:

- boolean temBaú

a. Crie um construtor que receba apenas portas e use o construtor padrão da superclasse

b. Crie um construtor que receba marca, modelo, ano, temBaú e chame o construtor parametrizado da superclasse Veiculo usando super(...)

c. Sobrescreva o método toString para que o carro também exiba o número de portas

4. Na classe Main, crie Carros e uma Motos e chame cada um dos construtores. Exiba a informação de todos os objetos criados.

6. Crie a classe abstrata Profissional, com os atributos: String nome, String documento, double salarioBase

a. Crie um construtor padrão, definindo:

nome = “Sem nome”

documento = “N/A”

salarioBase = 0.0

b. Crie um construtor parametrizado, inicializando todos os atributos.

c. Crie um método toString para exibir as informações.

d. Crie um método abstrato, chamado double calculaSalarioFinal().

e. Crie a classe FuncionarioCLT que herda de Profissional. A classe possui os seguintes atributos:

- double bonus

f. Crie dois construtores. Um que recebe o bônus e chama o construtor padrão da superclasse, e um que recebe todos os atributos (nome, documento, salarioBase e bonus), e chame o construtor parametrizado de Profissional.

g. Sobrescreva o método calculaSalarioFinal() retornando salarioBase + bonus.

h. Sobrescreva toString para exibir as informações do Funcionario CLT

7. Crie a classe PrestadorServiço que herda de Profissional e possui os seguintes atributos: int horasTrabalhadas, double valorHora

a. Crie um construtor que recebe apenas valorHora e use o construtor padrão de Profissional

b. Crie um construtor completo que receba nome, documento, horasTrabalhadas, valorHora e defina salarioBase = 0 usando o super-construtor parametrizado.

c. Sobrescreva calcularSalarioFinal() retornando horasTrabalhadas * valorHora

d. Sobrescreva toString() incluindo as informações de horas e valor da hora.

8. Crie uma classe Gerente que herda de FuncionarioCLT. Ele possui como atributos os seguintes atributos adicionais:

- double gratificacaoGerencia

a. Defina um construtor que receba apenas gratificacaoGerencia e utilize o construtor padrão de FuncionarioCLT

b. Construtor completo que receba: nome, documento, salario, salarioBase, bonus, gratificacaoGerencia

c. chame explicitamente o construtor parametrizado da superclasse

d. Sobrescreva calcularSalarioFinal() retornando o salarioBase + bonus + gratificacaoGerencia

e. Sobrescreva toString() para exibir as informações incluindo gratificação.

9. Na main:

a. Crie ao menos 1 objeto de cada tipo, utilizando todos os construtores disponíveis. Armazene todos em uma lista do tipo List<Profissional>

b. Percorra a lista e impriva as informações e o salário final gerado por calculaSalarioFinal() de cada objeto.

10. Explique com suas próprias palavras os seguintes conceitos fundamentais da memória e do modelo de execução de programas orientados a objetos:

a. O que são variáveis locais? Onde elas são armazenadas? Qual é seu tempo de vida e escopo?

b. O que são variáveis de instância? Onde elas ficam armazenadas? Quando elas existem e quando deixam de existir?

c. Qual a função da Stack (pilha de execução) no programa? Que tipos de dados ou informações são guardados nela? O que acontece com os frames da stack (stack frames) quando métodos são chamados e quando retornam?

d. Qual a função da Heap no Java? Que tipos de valores estão na heap? Quem controla o ciclo de vida dos objetos armazenados nela?

11. Considere o seguinte trecho de código em Java:

```
class Produto {
    String nome;
    double preco;
```

```
    Produto(String nome, double preco) {
        this.nome = nome;
        this.preco = preco;
    }
}
public class Main {
    public static void main(String[] args) {
        int quantidade = 5;
        Produto p1 = new Produto("Café", 10.0);
        calculaTotal(p1, quantidade);

        Produto p2 = criarProdutoPromocional("Açúcar");
    }

    public static void calculaTotal(Produto prod, int q) {
        double total = prod.preco * q;
        System.out.println("Total: " + total);
    }

    public static Produto criarProdutoPromocional(String nomeBase) {
        String nomePromo = nomeBase + " Promoção";
        return new Produto(nomePromo, 5.0);
    }
}
```

Explique detalhadamente, usando conceitos de stack e heap, quais variáveis desse código ficam armazenadas na stack e quais ficam na heap. Em sua resposta, descreva também:

a. Quais variáveis existem apenas dentro do escopo de cada método e deixam de existir após sua execução.

b. Quais objetos continuam vivos na heap depois que cada método termina.

c. O que acontece com os parâmetros dos métodos quando os métodos retornam.

d. Qual a diferença entre as variáveis locais (armazenadas na stack) e os objetos referenciados por elas (armazenados na heap)?

Métodos Estáticos

1. Defina o conceito de método utilitário. Como os métodos utilitários são acessados?

2. Defina o conceito de classe utilitária. Qual a diferença de uma classe utilitária e uma classe comum na orientação a Objetos?

3. Defina o conceito de variáveis estáticas. Qual a diferença entre variáveis estáticas e variáveis de instância em uma classe? Qual a diferença em relação ao acesso de valores considerando esses dois tipos de variáveis? Quando uma variável estática é inicializada?

4. Qual a diferença entre uma variável estática e uma variável estática final (material EaD sobre números e elementos estáticos). Qual a convenção usada para criar nomes de constantes no Java? Quais são as formas de inicialização de variáveis estáticas?

5. Onde a palavra “final” pode ser usada no Java? O significado é sempre o mesmo? Variáveis finais sempre são estáticas?

6. O que é um método final? Qual sua utilidade na orientação a objetos?

7. O que é uma classe final? Em que contexto faz sentido a criação de uma classe final?

8. O que são classes wrapper? Qual sua utilidade na orientação a objetos?

9. O que são imports estáticos? Em quais contextos eles podem ser úteis? Em quais contextos não é adequado utilizá-los?

10. Em Java, crie a classe ContadorObjetos. Ele possui os seguintes atributos.
- private static int totalCriados: contar quantos objetos dessa classe foram instanciados no -total.
- private int id: identificador único para cada objeto criado

a. Crie um construtor padrão, sem parâmetros. A cada vez que um novo objeto é criado, a varivável totalCriados deve ser incrementada em 1 unidade. Além disso, atribua ao atributo id o valor atual de totalCriados.
b. Implemente o método getId(), que deve retornar o identificador do objeto.
c. crie os métodos estáticos:
- public static int getTotalCriados(): retorna quantos objetos já foram criados
- public static void resetarContador(): zera o contador totalCriados. O método não deve afetar os IDS dos objetos já criados, apenas o contador global.
d. Crie a classe Util: A classe deve possuir apenas métodos utilitários (sem atributos), contendo:
- `public static boolean ehPar(int x)` → retorna true se o número for par.
- `public static int maior(int a, int b)` → retorna o maior dos dois valores.

e. Na classe Main, crie três instâncias de ContadorObjetos e imprima seus IDs. Imprima o valor de ContadorObjetos.getTotalCriados().
f. Chame ContadorObjetos.resetarContador().
g. Crie mais dois objetos e exiba seus IDs e o total atualizado.
h. Use os métodos da classe Util para: verificar se um número é par; Encontrar o maior entre dois inteiros.

Coleções e Generics

1. Analise o código:
   `List<String> musicas = new ArrayList<>();`
   Explique o papel do operador diamante (<>) no código acima. O que significa usar new ArrayList<>() ao invés de new ArrayList<String>()?

2. Uma loja virtual deseja implementar um pequeno sistema para gerenciar e ordenar seus produtos. Cada produto possui nome, preço e categoria. O sistema deve permitir armazenar os produtos em uma lista (ArrayList), e possibilitar diferentes formas de ordenação — por nome, preço e categoria — utilizando Comparator e classes genéricas.
   Crie uma classe genérica chamada Catalogo<T> que:
   a. Armazene uma lista de objetos (ArrayList<T>)
   b. Tenha métodos:
   - void adicionar(T item)
   - void listarTodos()
   - void ordenar(Comparator<T> comparator)
   b. Crie uma classe Produto com os seguintes atributos:
   - String nome
   - double preco
   - String categoria
   - Construtor
   - getters
   - toString()
   c. Crie três classes que implementem Comparator<Produto>:
   - ComparadorPorNome
   - ComparadorPorPreco
   - ComparadorPorCategoria
   d. Na classe principal (Main ou App):
   - Crie um Catalogo<Produto>
   - Adicione alguns produtos com dados variados
   - Exiba os produtos antes da ordenação Ordene e exiba novamente a lista em cada critério (nome, preço e categoria)

3. Um sistema de recursos humanos precisa gerenciar diferentes tipos de pessoas, como Funcionários, Clientes e Fornecedores. Cada tipo de pessoa tem alguns atributos em comum (nome, idade) e outros específicos (salário, empresa, etc.). Você deve implementar uma estrutura genérica e reutilizável para armazenar e ordenar essas pessoas de diferentes tipos.
   a. Crie uma classe abstrata Pessoa com os seguintes atributos:
   - String nome,
   - int idade
   - Construtor
   - getters
   - toString()
   - Um método abstrato String getTipo() (que será implementado pelas subclasses)
   b. Crie três subclasses:
   - Funcionario, com o atributo adicional: double salario
   - Cliente, com o atributo adicional: String email
   - Fornecedor, com o atributo adicional: String empresa
   c. Crie uma classe genérica Cadastro<T extends Pessoa> que contém um ArrayList<T>, além dos seguintes Métodos:
   - void adicionar(T pessoa)
   - void listar()
   - void ordenar(Comparator<T> comparator)
   d. Implemente três comparadores (comparators):
   - ComparadorPorNome
   - ComparadorPorIdade
   - ComparadorPorTipo (usa getTipo() da classe abstrata)
   e. Na classe principal (Main ou App):
   - Crie um Cadastro<Pessoa>
   - Adicione diferentes tipos de pessoas (Funcionario, Cliente, Fornecedor)
   - Mostre a lista antes e depois de aplicar cada tipo de ordenação

4. Uma instituição de ensino deseja organizar os cursos oferecidos em uma plataforma. Cada curso possui informações básicas como nome, carga horária e nível de dificuldade. O setor de TI precisa criar uma estrutura que permita armazenar, listar e ordenar cursos de forma flexível.
   a. Crie uma classe Curso com os seguintes atributos: String nome int cargaHoraria String nivel (valores possíveis: “Básico”, “Intermediário”, “Avançado”)
   b. Implemente: Construtor, getters e toString().
   c. Faça Curso implementar Comparable<Curso> e compare cursos por nome.
   d. Crie uma classe genérica Gerenciador<T> que armazene objetos em um ArrayList<T> e tenha os seguintes métodos:
   - void adicionar(T item)
   - void listar()
   - void ordenar(Comparator<T> comparator)

- void ordenarPadrao() — usa o compareTo() se o tipo for Comparable
e. Crie duas classes que implementem Comparator<Curso>:
- ComparadorPorCargaHoraria
- ComparadorPorNivel (ordene de Básico → Intermediário → Avançado)
f. Na classe principal (App), crie um Gerenciador<Curso>. Adicione pelo menos 5 cursos com valores variados e liste-os:
- Em ordem original
- Ordenados por nome (ordem natural)
- Ordenados por carga horária
- Ordenados por nível

5. Crie uma classe Veiculo com os seguintes atributos: modelo, categoria (SUV, Sedan, Caminhão), ano (int), kmRodados (int). A classe deve conter os métodos:
- construtor
- getters
- toString
- equals() e hashCode() sobrescritos. Devem garantir que dois veículos com os mesmos dados sejam considerados iguais (para uso correto em HashSet).
a. A classe Veiculo deve implementar Comparable<Veiculo> para permitir ordenação padrão por categoria em ordem alfabética. Em caso de empate, a ordenação deve ser feita por modelo.
b. Na classe main, crie uma List<Veiculo> e adicione pelo menos 6 veículos, incluindo um duplicado, de diferentes categorias e anos.
c. Exiba todos os veículos cadastrados usando um método genérico imprimirColecao(Collection<T>).
d. Ordene a lista de veículos por quilometragem (decrescente) usando um comparator. Exiba novamente a lista ordenada.
e. Usando lambda, ordene a lista por ano. Em caso de empate, ordene por quilometragem (crescente).
f. Crie um HashSet<Veiculo> a partir da lista e exiba o resultado.
g. Crie um TreeSet<Veiculo>, que utiliza o critério definido em compareTo() para ordenar os veículos.
h. Crie um HashMap<String, Integer>. A chave deve ser a categoria; O valor é a soma das quilometragens de todos os veículos daquela categoria.

6. Uma biblioteca deseja organizar seus livros em diferentes ordens de classificação, dependendo da necessidade do usuário. Crie um livro com os seguintes atributos: titulo, autor, categoria, anoPublicado, paginasLidas (representa o total de páginas já manuseadas em edições antigas).
a. A classe deve conter os seguintes métodos:
- Construtor
- getters
- toString()

- equals() e hashCode() sobrescritos, garantindo que dois livros são iguais quando todos os atributos são iguais (para uso correto em HashSet)

b. A classe Livro deve implementar Comparable<Livro>, adotando o seguinte critério de ordenação padrão:

- Ordenar por categoria em ordem alfabética. Em caso de empate, ordenar por autor. Persistindo o empate, ordenar por título.

c. Crie no método main uma List<Livro> com pelo menos 8 livros, usando o método Factory **List.of**. A lista deve incluir:

- Pelo menos 2 livros duplicados
- Livros de várias categorias
- Diferentes anos de publicação
- Diferentes valores de páginas lidas

d. Crie um método genérico:

- public static <T> void imprimirColecao(Collection<T> colecao)

Esse método deve simplesmente percorrer a coleção e imprimir seus elementos. Use esse método para exibir todos os livros cadastrados.

e. Crie um Comparator<Livro> chamado ComparatorPorUso que ordena pelo valor de paginasLidas do maior para o menor. Exiba a lista ordenada.

f. Usando lambda, ordene a lista pelo ano de publicação (crescente). Em caso de empate, ordenar por páginas lidas (crescente).

g. Crie um HashSet<Livro> a partir da lista de livros. Esse conjunto deve eliminar os duplicados com base em equals() e hashCode().

f. Crie um TreeSet<Livro>. Esse conjunto deve usar o critério definido no método compareTo() da classe Livro.

g. Crie um HashMap<String, Integer>. A chave deve ser a categoria (String). O valor deve ser a soma das páginas lidas de todos os livros daquela categoria. Percorra a lista original e preencha o mapa adequadamente. Exiba o mapa final. Lembre-se: O método que devolve a lista de chaves de um Map é keySet(). O método que recupera o valor associado a uma chave é get(chave).

7. Explique, com suas próprias palavras, a diferença fundamental entre as coleções List e Set. Quais tipos de problemas cada uma resolve melhor? Dê exemplos.

8. Como o comportamento de inserção e acesso de elementos difere entre List e Set? Comente sobre ordenação, repetição de elementos e posição (índices).

9. Por que um HashSet pode considerar que dois objetos são duplicados mesmo que ocupem posições diferentes na memória? Explique usando equals() e hashCode().

10. O que diferencia o Map das coleções List e Set em termos de estrutura de dados representada? Em quais situações é mais apropriado usar um Map?

11. Em uma aplicação real, quando você escolheria utilizar um TreeSet ao invés de um HashSet?

12. Explique por que um Map não pode ser diretamente considerado uma coleção de elementos como List ou Set.

13. Explique o conceito de Factory Method em orientação a objetos e descreva por que esse padrão é útil em projetos que precisam criar objetos de diferentes tipos.

14. O que são parâmetros de tipos genéricos em Java e qual problema eles resolvem em comparação ao uso de Object antes do Java 5?

15. Qual é a diferença entre declarar um tipo genérico na classe (class Caixa<T>) e declarar na assinatura de um método (<T> T identidade(T valor))?
    Em que situações cada forma é mais adequada?

16. Por que interfaces como Comparable<T> e classes como Collections usam tanto parâmetros de tipos genéricos? Dê um exemplo concreto mostrando como isso melhora a segurança de tipos.

17. Qual é a diferença entre usar List<Object> e List<?> como parâmetro de método? Em quais casos cada forma deveria ser usada?

18. Explique por que o código abaixo gera erro de compilação e como poderia ser corrigido:
    ```
    List<? extends Number> numeros = new ArrayList<Integer>();
    numeros.add(10);
    ```

19. Seja o compilador. Considere os seguintes métodos:
    ```
    private void takeDogs(List<Dog> dogs){}
    private void takeAnimals(List<Animal> animals){}
    private void takeSomeAnimals(List<? extends Animal> animals){}
    private void takeObjects(ArrayList<Object> objects){}
    ```
    Quais dos comandos abaixo compilam e quais não compilam?
    (Resposta na aula de Generics - Parte 4)

- [ ] `takeAnimals(new ArrayList<Animal>());`
- [ ] `takeDogs(new ArrayList<Animal>());`
- [ ] `takeAnimals(new ArrayList<Dog>());`
- [ ] `takeDogs(new ArrayList<>());`
- [ ] `List<Dog> dogs = new ArrayList<>();`
- [ ] `takeDogs(dogs);`
- [ ] `takeSomeAnimals(new ArrayList<Dog>());`

```
takeSomeAnimals(new ArrayList<>());
```

Tratamento de Exceções

1. Implemente um método chamado buscarUsuario que recebe um String id e:
   a. lança uma exceção IllegalArgumentException caso o id seja nulo ou vazio;
   b. caso contrário, retorna a mensagem "Usuário encontrado: " + id.
   c. Depois disso, crie um método main que:
   - Chama buscarUsuario com um valor válido.
   - Chama buscarUsuario passando null ou "".
   - Trata a exceção lançada usando try/catch.
   - Exibe uma mensagem amigável quando ocorrer erro.
   O método deve ser assim:

```
public static String buscarUsuario(String id) {
    if (id == null || id.isBlank()) {
        throw new IllegalArgumentException("ID inválido. Não pode ser
                                           nulo ou vazio.");
    }
    return "Usuário encontrado: " + id;
}
```

   O método main deve conter:
   a. um try/catch envolvendo chamadas ao método;
   b. um catch(IllegalArgumentException e) apropriado;
   c. uma mensagem como: "Erro ao buscar usuário: " + e.getMessage().
   Observação: IllegalArgumentException é uma exceção em Java que é lançada quando um método é chamado com um argumento inválido, impróprio ou que não atende aos critérios de validação. pertencente ao pacote java.lang, portanto não precisa ser importada e pode ser usada diretamente. Trata-se de uma unchecked expection (estende RuntimeException)

2. Implemente um método chamado **lerTemperatura** que recebe um double valor e lança uma IllegalArgumentException caso a temperatura esteja fora do intervalo permitido [-50, 80], pois sensores reais costumam ter limites físicos de leitura.
   Caso contrário, retorna a mensagem: "Temperatura registrada: " + valor + "°C"
   a. Método a ser implementado:

```
public static String lerTemperatura(double valor) {
    if (valor < -50 || valor > 80) {
        throw new IllegalArgumentException(
            "Temperatura fora do intervalo permitido (-50 a 80°C)."
        );
    }
    return "Temperatura registrada: " + valor + "°C";
}
```

   a. No main:
   - Chame lerTemperatura(22.5) — valor válido.

- Chame lerTemperatura(150) — valor inválido.
- Use try/catch para capturar IllegalArgumentException.
- Exiba mensagem amigável ao usuário, como “Erro ao ler temperatura: Temperatura fora do intervalo permitido (...)”

3. Implemente um método chamado lerArquivoConfiguracao que recebe um String caminho e lança uma exceção IOException se o caminho for nulo ou vazio, simulando a falha ao acessar um arquivo. Caso contrário, retorna a mensagem “Arquivo carregado com sucesso: " + caminho.
   Esse método **deve declarar** na assinaturaa cláusula throws IOException:
   ```
   public static String lerArquivoConfiguracao(String caminho) throws IOException
   {
       if (caminho == null || caminho.isBlank()) {
           throw new IOException("Caminho inválido. O arquivo não pôde ser
   carregado.");
       }
       return "Arquivo carregado com sucesso: " + caminho;
   }
   ```
   Observação: IOException é uma checked exception, portanto obrigatoriamente deve ser declarada com throws ou tratada onde for chamada.
   No método main:
   a. Chame lerArquivoConfiguracao("config.txt") — valor válido.
   b. Chame lerArquivoConfiguracao("") — irá lançar IOException.
   c. Use **try/catch** para capturar IOException.
   d. Exiba uma mensagem amigável como:
   “Erro ao carregar arquivo: Caminho inválido. O arquivo não pôde ser carregado.”

4. Explique a diferença entre exceções checked e unchecked. Quando cada tipo deve ser utilizado e qual impacto isso tem no design de métodos e APIs?

5. O que significa dizer que um método “adia o tratamento” de uma exceção? Explique usando um exemplo com a cláusula throws e descreva quando essa abordagem é recomendada.

6. O que acontece quando um método contém múltiplos blocos catch? Explique como o Java escolhe qual bloco tratará a exceção e descreva a importância da ordem dos catch.

7. Para que serve o bloco finally? Dê exemplos de situações reais em que seu uso é fundamental, e explique o que acontece se um return aparece dentro de um try ou catch.

8. Como funciona o polimorfismo aplicado a exceções? Por que é possível capturar exceções específicas e também uma exceção mais genérica como Exception? Quais são os riscos de capturar exceções muito genéricas?

9. O que acontece quando uma exceção não é tratada dentro de um método?

10. Explique detalhadamente o papel da cláusula throws na assinatura de um método. Como ela afeta quem chama o método? Em que casos ela é obrigatória e quando é opcional?

Lambdas, Streams e Interfaces Funcionais

1. Considere a interface abaixo:

```
public interface Operacao {
    int executar(int a, int b);

    default void imprimir(int resultado) {
        System.out.println("Resultado: " + resultado);
    }

    static boolean isPositivo(int n) {
          return n ⩾ 0;
    }
}
```

   a. Explique por que essa interface pode ser considerada uma interface funcional.
   b. Quantos métodos abstratos uma interface funcional pode ter? Explique.

2. Crie uma interface funcional chamada Transformador, que recebe uma String e retorna outra String.
   A interface deve ter:
   a. um único método abstrato String transformar(String s);
   b. um método default que imprime a string transformada
   c. um método static que verifica se a string é vazia

3. Considere as interfaces funcionais:

```
@FunctionalInterface
interface Somador {
    int somar(int a, int b);
}

@FunctionalInterface
interface Mensagem {
    String formatar(String nome);
}
```

   a. No método main, crie uma expressão lambda para Somador que some dois números.
   b. No método main, crie uma expressão lambda para Mensagem que retorne `"Olá, <nome>! Bem-vindo."`
   c. Atribua cada lambda a uma variável com referência à interface funcional e escreva um

exemplo de chamada:

```
Somador s = <expressão lambda>
Mensagem m = <expressão lamda>
int resultado = s.somar(10, 5);
System.out.println(resultado);
String msg = m.formatar("Ana");
System.out.println(msg);
```

4. Utilizando a interface:

```
@FunctionalInterface
interface CalculadoraArea {
    double calcular(double largura, double altura);
}
```

Crie uma expressão lambda com bloco de código, que:
a. valida se largura e altura são positivas. Se não forem, retorna 0. Caso contrário, retorna a área (largura × altura).
b. Na main, escreva um exemplo de chamada do método:

```
CalculadoraArea calc = ( ... ) → { ... };
```

Exiba o resultado.

5. Considere a classe:

```
class Produto {
    private String nome;
    private double preco;

    public Produto(String nome, double preco) {
        this.nome = nome;
        this.preco = preco;
    }
    public String getNome() { return nome; }
    public double getPreco() { return preco; }

    @Override
    public String toString() {
        return nome + " - R$ " + preco;
    }
}
```

E a lista:

```
List<Produto> produtos = List.of(
    new Produto("Camiseta", 50),
    new Produto("Calça", 120),
    new Produto("Boné", 35),
    new Produto("Jaqueta", 350),
    new Produto("Meias", 20)
);
```

Usando streams:
a. Gere uma lista contendo apenas os produtos com preço **acima de 50**.
b. Imprima os resultados usando forEach.
c. Usando a stream map, gere uma lista somente com os nomes dos produtos em letras maiúsculas. Para isso, basta usar o método toUpperCase() da classe String.
d. Usando streams, obtenha, retornando uma lista:
- os nomes dos produtos **com preço abaixo de 100**,
- ordenados alfabeticamente.

6. A partir do código disponibilizado no AVA, “JukeboxDoLou”, modifique-o para:
   a. Exiba as 5 músicas mais tocadas.
   b. Encontre a música mais antiga e a música mais recente da lista.
   c. crie uma lista com as músicas clássicas, lançadas antes do ano 2000. Ordene-as por gênero.
   c. Conte o número de artistas únicos cadastrados.
   d .Exiba todas as músicas do gênero POP em ordem decrescente desconsiderando letras maiúsculas e minúsculas.
   e .Existem músicas com o mesmo nome, mas artistas diferentes? Liste-as.

7. Descreva a relação entre expressões lambda e interfaces funcionais.
   Use um exemplo para explicar como a JVM sabe qual método deve ser executado.

8. Explique como funcionam as lambdas com bloco de código { ... }.
   Em quais situações elas são mais apropriadas do que lambdas simples (de uma linha)?

9. O que significa dizer que uma “stream é uma pipeline de operações”? Explique o que são operações intermediárias e terminais.

10. Compare a operação map com a operação filter. Qual é o objetivo de cada uma? Quando usar uma e quando usar a outra?