



## Contexto Geral: App de Delivery

A empresa DelivFast é uma plataforma de delivery de comida que conecta clientes e restaurantes.

Seu objetivo neste desafio será construir partes desse sistema, simulando a lógica de funcionamento da aplicação.

Você irá desenvolver classes, métodos e estruturas de dados que representam os principais elementos do sistema.

# Não é necessário criar interfaces gráficas ou interações com banco de dados.

# A ideia é trabalhar somente com lógica de programação e estruturas de dados em memória.

# Etapa 1 – Modelagem de Classes Básicas (POO)

---

## 1. Descrição / Objetivo

O objetivo desta etapa é estruturar os principais componentes do sistema de delivery utilizando conceitos de Programação Orientada a Objetos (POO).

Você deverá modelar as entidades fundamentais do sistema: Cliente, Produto e Restaurante, implementando suas classes com os respectivos atributos, construtores e métodos de acesso. Essas entidades servirão como base para o funcionamento do sistema nas etapas seguintes.

Ao final desta etapa, o sistema já deve estar preparado para:

- Criar instâncias de clientes
- Criar restaurantes com uma categoria definida
- Criar produtos

A funcionalidade de pedidos será implementada na próxima etapa.

---

## 2. Descrição Técnica

Implemente as seguintes estruturas, **construtores** e **métodos** (se atente a utilização de **public** e **private** nos momentos corretos). A utilização de encapsulamento **não será obrigatória** para ganharmos agilidade.

O uso de `toString()` é opcional, mas recomendado para facilitar os testes.

---

### Classe **Cliente**

- `nome` (String)
- `cpf` (String)
- `endereco` (String)
- `telefone` (String)

---

### Classe `Produto`

- `nome` (String)
  - `preco` (Double ou equivalente)
- 

### Classe `Restaurante`

- `nome` (String)
  - `endereco` (String)
  - `categoria` (Enum `CategoriaRestaurante`)
  - `cardapio` (lista de objetos `Produto`)
- 

### Enum `CategoriaRestaurante`

Deve conter algumas categorias fixas para os restaurantes.

Exemplo:

- `PIZZARIA`
  - `LANCHONETE`
  - `JAPONÊS`
  - `HAMBURGUERIA`
  - `OUTROS`
- 

## 3. Entregáveis (O que será avaliado)

Abaixo estão os itens que compõem esta etapa. Cada item será avaliado individualmente.

Utilize esta lista como checklist para verificar se completou tudo que é esperado.

- Classe `Cliente` implementada corretamente
- Classe `Produto` implementada corretamente
- Enum `CategoriaRestaurante` criado e utilizado na classe `Restaurante`
- Classe `Restaurante` implementada com os campos e estrutura correta

## Etapa 2 – Operações sobre o Cardápio

---

### 1. Descrição / Objetivo

O objetivo desta etapa é adicionar funcionalidades ao restaurante que permitam gerenciar seu cardápio de forma simples e eficaz.

Como não estamos utilizando banco de dados, o nome do produto **será considerado o identificador único dentro do cardápio de um restaurante.**

O foco será garantir que o cardápio não contenha itens duplicados, além de permitir exibição, busca e remoção de produtos.

Ao final desta etapa, o restaurante deve ser capaz de:

- Adicionar produtos ao cardápio, com validação
  - Exibir o cardápio de forma clara para o cliente
  - Buscar produtos no cardápio com base em um termo de pesquisa
- 

### 2. Descrição Técnica

Implemente os seguintes métodos dentro da classe **Restaurante**:

---

#### Método **adicionarProduto(Produto produto)**

- Parâmetro: objeto **Produto**
  - Verifica se já existe produto com o mesmo nome no cardápio (case-insensitive)
  - Se o produto já existir, exibe mensagem de erro e não adiciona
  - Se for válido, adicione ao cardápio e exibe mensagem de sucesso
- 

#### Método **exibirCardapio()**

- Parâmetro: nenhum
- Exibe todos os produtos do cardápio no formato numerado

- Exemplo: 1. Pizza – R\$35.00

- Caso o cardápio esteja vazio, exibe mensagem apropriada
- 

✚ Método `buscarProdutos(String termoBusca)` - priorizar utilização da estrutura `foreach`

- Parâmetro: `termoBusca` (String)
  - Retorna uma **lista de produtos** cujo nome contém o termo informado (case-insensitive)
  - Se nenhum produto for encontrado, a lista deve estar vazia e uma mensagem apropriada deve ser exibida
  - Para cada produto encontrado, o método deve exibir em tela
    - Exemplo: Pizza – R\$35.00
- 

### 3. Métodos de Teste (execução obrigatória)

Você deverá implementar os seguintes métodos de teste dentro da sua classe principal (por exemplo, no `main`).

Utilize a mesma base de dados para todos os testes.

---

Restaurante base:

- Nome: "Bella Pizza"
- Categoria: PIZZARIA
- Endereço: "Av antártico 381"

Produtos base:

- `Produto("Pizza Mussarela", 35.00)`
  - `Produto("Pizza Calabresa", 38.00)`
  - `Produto("Refrigerante", 8.00)`
  - `Produto("Pastel", 6.50)`
- 

✚ Método `testAdicionarProduto()`

1. Criar o restaurante "Bella Pizza" com cardápio vazio
  2. Adicionar o produto "Pizza Mussarela"
  3. Adicionar novamente o produto "Pizza Mussarela" (deve exibir mensagem de erro por duplicidade)
  4. Adicionar o produto "Refrigerante"
  5. Demonstrar a quantidade de itens no cardápio
- 

#### Método `testExibirCardapio()`

1. Criar o restaurante
  2. Chamar `exibirCardapio()` → deve mostrar mensagem de cardápio vazio
  3. Adicionar "Pizza Mussarela" e "Refrigerante"
  4. Chamar `exibirCardapio()` → deve mostrar os dois itens numerados
- 

#### Método `testBuscarProdutos()`

1. Criar o restaurante e adicionar todos os produtos da base
  2. Buscar por "pizza" → deve retornar dois produtos: "Pizza Mussarela" e "Pizza Calabresa"
  3. Buscar por "calabresa" → deve retornar apenas "Pizza Calabresa"
  4. Buscar por "batata" → deve retornar uma lista vazia e exibir mensagem: "Nenhum produto encontrado para 'batata'."
- 

## 4. Entregáveis (O que será avaliado)

Abaixo estão os itens que compõem esta etapa. Cada item será avaliado individualmente.

Utilize esta lista como checklist para verificar se completou tudo que é esperado.

- Método `adicionarProduto()` com verificação de duplicidade
- Método `exibirCardapio()` com formatação amigável, e mensagem adequada para cardápio vazio
- Método `buscarProdutos()` funcionando corretamente com múltiplos resultados
- Método `testAdicionarProduto()` criado e funcional
- Método `testExibirCardapio()` criado e funcional

- Método `testBuscarProdutos()` criado e funcional

## Etapa 3 – Estruturação e Publicação de Pedidos

---

### 1. Descrição / Objetivo

O objetivo desta etapa é estruturar e implementar a lógica de criação e confirmação de pedidos no sistema de delivery.

Um pedido envolve um cliente, um restaurante e uma lista de itens escolhidos – cada um com seu respectivo produto e quantidade.


Você deverá implementar a estrutura completa de um pedido, validar a existência dos produtos no cardápio do restaurante, calcular o valor total com base nas quantidades e confirmar o pedido, alterando seu status.

---

### 2. Descrição Técnica

A estrutura do pedido será composta por itens de pedido, representando a quantidade solicitada de um determinado produto.

Cada pedido conterá uma lista desses itens, e não diretamente uma lista de produtos.

 Classe **Pedido**:

- **cliente** (Cliente)
- **restaurante** (Restaurante)
- **itens** (List<ItemPedido>)
- **status** (Enum **StatusPedido**)
- **total** (Double)

Configurações adicionais

- O construtor dessa classe deve receber **Cliente e Restaurante**, tornar a lista de itens vazia e o status como CARRINHO
- 

 Classe **ItemPedido**



- `produto` (Produto)
- `quantidade` (int)
- `subTotal` (Double)

Configurações adicionais

- `calcularSubtotal()` → preenche o subtotal do item (`preço do produto * quantidade`). Utilize esse método no construtor da classe.
  - O construtor da classe deve receber `Produto` e `Quantidade`
- 

 Enum `StatusPedido`

- `CARRINHO` (estado inicial)
  - `EM_PROCESSAMENTO`
  - `ENTREGUE`
- 

 Método `adicionarItemPedido(Produto produto, int quantidade)` - Classe `Pedido`

- Verifica se o produto existe no cardápio do restaurante
    - Se sim, cria um novo **ItemPedido**, e o adiciona ao pedido
    - Se não, exibe mensagem informando que o produto não está disponível e não o adiciona
  - Se a adição é executada com sucesso
    - Exibe mensagem de adição
    - Calcula um novo total do pedido utilizando `calcularTotal()`
  - Se o produto já existir no Pedido atual, adicione o comportamento
    - Somar a nova quantidade solicitada à quantidade já existente
    - Calcula um novo total do pedido utilizando `calcularTotal()`
- 

 Método `calcularTotal()` - Classe `Pedido`

- Percorre todos os itens e soma os subtotais (`item.calcularSubtotal()`)
-

### Método `confirmarPedido()` – Classe Pedido

- Só permite confirmar se:
    - O status atual for `CARRINHO`
    - A lista de itens não estiver vazia
  - Se confirmado, muda status para `EM_PROCESSAMENTO`
  - Exibe mensagem de sucesso ou erro de acordo com o devido cenário
- 

### Método `exibirResumo()` – Classe Pedido

Exibe no console:

- Nome do cliente
  - Nome do restaurante
  - Lista de itens: produto, quantidade e subtotal
  - Valor total
  - Status do pedido
- 

## 3. Métodos de Teste (execução obrigatória)

Utilize os dados abaixo:

Cliente base:

- `Cliente("Ana", "123.456.789-00", "Rua Central, 123", "(11) 99999-8888")`

Produtos base:

- `Produto("Pizza Mussarela", 35.00)`
- `Produto("Pizza Calabresa", 38.00)`
- `Produto("Refrigerante", 8.00)`
- `Produto("Pastel", 6.50)`

Restaurante base:

- Nome: `"Bella Pizza"`
- Categoria: `PIZZARIA`

- Cardápio: lista contendo os três primeiros produtos (o "Pastel" não deve estar no cardápio)
- 

#### Método `testCriarPedidoComProdutosValidos()`

1. Criar cliente, restaurante e produtos
  2. Criar pedido com este cliente e restaurante
  3. Adicionar itens ao pedido
    - "Pizza Mussarela" com quantidade 2
    - "Refrigerante" com quantidade 1
  4. Exibir resumo do pedido
  5. Confirmar pedido
  6. Verificar status: deve estar como `EM_PROCESSAMENTO`
- 

#### Método `testAdicionarProdutoInvalido()`

1. Criar cliente, restaurante e produtos
  2. Criar pedido
  3. Tentar adicionar "Pastel" → deve exibir erro e não adicionar
  4. **Confirmar pedido** com lista vazia → deve exibir erro
  5. Adicionar "Pizza Calabresa" com quantidade 1
  6. Confirmar pedido → deve funcionar
  7. Exibir resumo
- 

## 4. Entregáveis (O que será avaliado)

- Estrutura do item de pedido implementada corretamente com `produto`, `quantidade` e `calcularSubtotal()`
- Classe `Pedido` implementada utilizando itens de pedido
- Enum `StatusPedido` criado e utilizado corretamente
- Método `adicionarItemPedido(produto, quantidade)` com validações e soma correta
- Método `calcularTotal()` funcional e preciso

- Método `confirmarPedido()` com validações de status e lista de itens
  - Método `exibirResumo()` exibe dados de forma clara e
  - Método `testCriarPedidoComProdutosValidos()` implementado e funcional
- Método `testAdicionarProdutoInvalido()` implementado e funcional

## Etapa 4 – Relatórios e Lógica de Processamento

---

### 1. Descrição / Objetivo

Esta etapa é voltada à construção de funcionalidades mais avançadas, com foco em relatórios e processamento de coleções.

A ideia é testar sua capacidade de trabalhar com listas, mapas e filtros dentro de regras de negócio mais próximas da realidade.

Você deverá implementar dois recursos:

1. Um relatório dos produtos mais vendidos com base em uma lista de pedidos
2. Um filtro para pedidos feitos por clientes específicos em um restaurante

Ambas as funcionalidades devem retornar estruturas organizadas e também apresentar seus resultados no console de forma organizada.

---

### 2. Descrição Técnica

---

 Método `gerarRelatorioProdutosMaisVendidos(List<Pedido> pedidos)`

- Parâmetro: lista de pedidos
- Contar quantas unidades de cada produto foram vendidas (usando a quantidade do `ItemPedido`)
- Ordenar os produtos pela quantidade vendida, do maior para o menor
- Exibir no console os 3 produtos mais vendidos
- Retornar um `Map<Produto, Integer>` com o total de unidades vendidas por produto

Atenção: A contagem deve considerar a quantidade de cada `ItemPedido`, e não apenas a ocorrência de produtos.

Exemplo de saída no console:

Top 3 produtos mais vendidos:

1. Pizza Mussarela – 7 unidades

2. Refrigerante – 3 unidades

3. Pizza Calabresa – 2 unidades

Se houver menos de 3 produtos, exibir apenas os disponíveis, mantendo a ordem decrescente. Em caso de empate, considerar ordem alfabética.

---

 Método `gerarRelatorioPedidosPorCliente(List<Pedido> pedidos, List<String> cpfs)`

- Parâmetros:
  - Lista de pedidos (`List<Pedido>`)
  - Lista de CPFs (`List<String>`)
- Retornar um `Map<String, List<Pedido>>`, onde a chave é o CPF do cliente e o valor é a lista de pedidos que ele fez naquele restaurante
- Exibir no console de forma clara

Exemplo de saída no console:

Cliente: 123.456.789-00 – 2 pedidos

Cliente: 987.654.321-00 – 1 pedido

---

### 3. Métodos de Teste (execução obrigatória)

Utilize os dados abaixo para garantir que seu teste seja completo e fácil de validar.

---

Clientes base:

- `Cliente("Ana", "123.456.789-00", "Rua Central, 123", "(11) 99999-8888")`
- `Cliente("Carlos", "987.654.321-00", "Rua das Flores, 45", "(11) 98888-7777")`
- `Cliente("Julia", "111.222.333-44", "Av. Brasil, 100", "(11) 97777-6666")`

Produtos base:

- "Pizza Mussarela"
- "Pizza Calabresa"
- "Pizza Portuguesa"
- "Refrigerante"
- "Pastel"

Restaurante base:

- Nome: "Bella Pizza"
  - Categoria: PIZZARIA
  - Cardápio: 3 primeiros produtos
- 

📌 Método `testRelatorioProdutosMaisVendidos()`

1. Criar 3 pedidos com produtos variados, ex:
    - Pedido 1: Mussarela (2x), Calabresa (3x)
    - Pedido 2: Mussarela (3x), Refrigerante (1x)
    - Pedido 3: Mussarela (2x), Pastel (2x)
  2. Gerar e exibir o relatório ordenado, mostrando os 3 produtos mais vendidos
- 

📌 Método `testRelatorioPedidosPorCliente()`

1. Criar pedidos variados de 3 clientes, todos para o restaurante "Bella Pizza"
  2. Criar uma lista de CPFs com 2 deles
  3. Gerar uma lista completa de pedidos
  4. Exibir no console agrupado por cliente
- 

#### 4. Entregáveis (O que será avaliado)

- Método `gerarRelatorioProdutosMaisVendidos()` funcionando corretamente  
consideração da quantidade de cada `ItemPedido` no cálculo, ordenação correta dos  
produtos por unidades vendidas, exibição no console dos 3 produtos mais vendidos

- Método `testRelatorioProdutosMaisVendidos()` criado e funcional
- Método `()` funcionando corretamente
- Método `testRelatorioPedidosPorCliente()` criado e funcional
- Uso correto de coleções (`Map`, `List`, `Set`)
- Lógica de agrupamento, filtro e contagem bem implementada
- Exibição clara dos resultados no console