

## Instruções Gerais para o Projeto Final da Turma 1433

**Objetivo:** Projetar e implementar a lógica de negócios de um sistema completo utilizando os pilares da Programação Orientada a Objetos com TypeScript. O foco **não é a interface visual**, mas sim um modelo de domínio robusto, seguro e bem estruturado que *poderia* servir de base para uma aplicação front-end.

### Entregáveis:

1. **Código-Fonte Completo:** Um link para um repositório no GitHub com todo o projeto TypeScript.
  2. **Diagrama de Classes UML:** Uma imagem (`.png`, `.jpg`) ou `.pdf` do blueprint do sistema, criado antes da codificação, mostrando as classes, atributos, métodos e, principalmente, os relacionamentos.
  3. **Arquivo README.md:** Na raiz do repositório, um arquivo explicando:
    - A descrição do projeto.
    - As decisões de design que você tomou (ex: "Usei Composição entre Pedido e ItemPedido porque...").
    - Como executar o projeto (basicamente, rodar o `index.ts`).
  4. **Arquivo de Demonstração (`index.ts`):** Um arquivo `index.ts` bem organizado que cria instâncias das suas classes e chama seus métodos para demonstrar todas as funcionalidades implementadas, com `console.log` claros para mostrar os resultados.
- 

## Opção 1: E-commerce - A Evolução Final

**Conceito:** Expandir o projeto que desenvolvemos em aula para um sistema de e-commerce mais completo e realista. Você irá refatorar, formalizar e adicionar novas funcionalidades complexas, demonstrando maestria no domínio que já exploramos.

### Classes Essenciais:

- **Cliente:** Gerencia dados do cliente e seu histórico.
- **Produto:** Descreve um item à venda, incluindo controle de estoque.
- **Carrinho:** Modela o carrinho de compras, que é temporário e pertence a um cliente.
- **ItemCarrinho:** Representa um produto dentro do carrinho, com uma quantidade específica.
- **Pedido:** Representa uma compra finalizada, gerada a partir de um carrinho.

### Requisitos de POO:

- **Encapsulamento Forte:** Todos os atributos devem ser `private`. O acesso e a modificação devem ser feitos exclusivamente por getters, setters com validação e

métodos públicos. (Ex: não deve ser possível adicionar um produto com estoque negativo a um carrinho).

- **Relacionamentos Claros (UML):**
  - **Cliente** e **Pedido** (Associação: 1 para N).
  - **Carrinho** e **ItemCarrinho** (Composição: 1 para N).
  - **Pedido** e **Produto** (Associação via itens do pedido).
- **Lógica de Negócio em Métodos:**
  - A classe **Carrinho** deve ter métodos como `adicionarProduto(produto, quantidade)`, `removerProduto(produtoId)`, `calcularTotal()`.
  - A classe **Produto** deve ter um método `baixarEstoque(quantidade)` que só funciona se houver estoque suficiente.
  - A classe **Cliente** deve ter um método `finalizarCompra()` que transforma seu **Carrinho** em um **Pedido**.
- **Serialização:** Todas as classes devem ter um método `toJSON()` para simular o envio de dados para uma API.

### Desafios Avançados (Para ir além):

- Implemente um sistema de **Cupons de Desconto**. Crie uma classe abstrata **Cupom** e classes filhas como **CupomPercentual** e **CupomValorFixo**, aplicando **Herança e Polimorfismo**. O **Carrinho** poderá receber um cupom e aplicar o desconto no total.
- Crie uma classe **Pagamento** com diferentes métodos (**CartaoCredito**, **Pix**), usando polimorfismo.

---

## Opção 2: Sistema Bancário Simplificado - O Desafio Lógico

**Conceito:** Modelar a lógica de um pequeno sistema bancário, focando na segurança das transações e na integridade dos dados. Este projeto é excelente para demonstrar um encapsulamento rigoroso e regras de negócio bem definidas.

### Classes Essenciais:

- **Cliente:** Gerencia os dados do correntista.
- **Conta:** Modela a conta bancária, com saldo e histórico de transações.
- **Transacao:** Representa uma operação única (depósito, saque, transferência).

### Requisitos de POO:

- **Encapsulamento Máximo:** O atributo `saldo` da **Conta** deve ser `private` e não deve ter um `setter` público. A única forma de alterar o saldo é através de métodos como `depositar(valor)` e `sacar(valor)`.
- **Validação em Métodos:**

- `sacar(valor)`: Deve verificar se `valor` é positivo e se há saldo suficiente.
- `depositar(valor)`: Deve verificar se `valor` é positivo.
- `transferir(valor, contaDestino)`: Deve orquestrar uma chamada de `sacar()` em si mesma e `depositar()` na conta de destino, garantindo que a operação seja atômica.
- **Relacionamentos Claros (UML):**
  - `Cliente` e `Conta` (Associação: 1 para N).
  - `Conta` e `Transacao` (Composição: uma conta é composta por seu extrato de transações).
- **Métodos Estáticos:** Crie um método `Transacao.criarTransferencia(...)` que gera duas instâncias de `Transacao`: um débito para a conta de origem e um crédito para a de destino.
- **Serialização:** Implemente `toJSON()` para `Cliente` e `Conta`, tomando cuidado para não expor informações sensíveis. O extrato (lista de transações) deve ser serializável.

#### Desafios Avançados (Para ir além):

- Aplique **Herança**: Crie uma classe abstrata `Conta` e classes filhas `ContaCorrente` (com limite de cheque especial) e `ContaPoupanca` (com um método `renderJuros()`).
- Crie um sistema de "chaves PIX", onde um `Cliente` pode registrar chaves (`email`, `cpf`) e as transferências podem ser feitas usando essas chaves em vez da referência direta do objeto `Conta`.

## Opção 3: Ferramenta de Gerenciamento de Conteúdo (Blog) - O Desafio Criativo

**Conceito:** Criar o back-end lógico para um sistema de blog ou CMS. Este projeto foca em modelar diferentes tipos de conteúdo e as interações entre usuários, posts e comentários.

#### Classes Essenciais:

- `Usuario`: Modela o autor do conteúdo, com nome, email e senha (não precisa de criptografia real).
- `Post`: Representa uma publicação no blog, com título, conteúdo e autor.
- `Comentario`: Modela um comentário feito em um `Post`, contendo texto e o autor do comentário.
- `Categoria`: Usada para organizar os `Posts`.

#### Requisitos de POO:

- **Encapsulamento:** Proteja os dados, especialmente os relacionados à autoria. Um `Post` só pode ter seu autor alterado por um método específico, por exemplo.

- **Relacionamentos Claros (UML):**
  - `Usuario` e `Post` (Associação: 1 para N).
  - `Post` e `Comentario` (Composição: 1 para N).
  - `Post` e `Categoria` (Associação: N para N, pode ser simplificada para 1 para N onde um post tem uma categoria).
- **Lógica de Negócio em Métodos:**
  - `Usuario` deve ter um método `publicarPost(titulo, conteudo)`.
  - `Post` deve ter métodos como `adicionarComentario(texto, autor)` e `exibir()`.
  - `Post` deve ter um sistema de "likes" (`adicionarLike()`, `obterTotalLikes()`).
- **Getters para Dados Computados:** A classe `Post` pode ter um getter `get resumoConteudo(): string` que retorna os primeiros 100 caracteres do conteúdo.
- **Serialização:** Todas as classes devem ser serializáveis via `toJSON()`, mostrando como uma API de blog retornaria os dados.

#### Desafios Avançados (Para ir além):

- Aplique **Herança e Polimorfismo**: Crie uma classe base abstrata `Publicacao` e classes filhas como `Artigo` (com texto) e `VideoPost` (com URL de vídeo e duração). O sistema deve ser capaz de lidar com qualquer tipo de `Publicacao` de forma polimórfica.
- Implemente um sistema de **permissões**, onde um `Usuario` pode ter um papel (`Admin`, `Editor`, `Leitor`), e certas ações (como `deletarPostDeOutroUsuario`) só podem ser executadas por um `Admin`.

#### FORMAS DE ENTREGA:

Link do GitHub ou projeto zipado (não enviar junto a pasta `node_modules`).

Pode ser enviado direto pelo LMS.

Pode ser enviado para o e-mail [roosevelt.franklin81@gmail.com](mailto:roosevelt.franklin81@gmail.com)

**Data limite para entrega 22/09/2025**