Nota para os Alunos: Vocês têm liberdade para escolher a linguagem de programação (Java, Python, C#, JavaScript, etc.) e o framework de testes (JUnit, PyTest, NUnit, Jest, etc.) que preferirem para completar as tarefas.

Tarefa 1: Módulo de Cálculo de Frete

- **Contexto:** Você foi encarregado de desenvolver um módulo para uma plataforma de e-commerce que calcula o custo do frete com base no peso do pacote e na região de destino. As regras de negócio são cruciais para a lucratividade da empresa.
- **Requisitos Funcionais:**

  1. Crie uma classe CalculadoraFrete com um método calcular(double pesoKg, String regiao).
  2. O método deve retornar o valor do frete com base nas seguintes regras:
     - **Região "Sudeste":** Custo fixo de R10,00 mais R2,00 por quilo adicional acima de 1 kg. Pacotes com 1 kg ou menos têm apenas o custo fixo.
     - **Região "Sul":** Custo fixo de R15,00 mais R2,50 por quilo adicional acima de 1 kg.
     - **Região "Centro-Oeste":** Custo fixo de R20,00 mais R3,00 por quilo adicional acima de 1 kg.
     - **Outras Regiões:** Custo fixo de R30,00maisR5,00 por quilo adicional acima de 1 kg.
  3. O método deve lançar uma exceção IllegalArgumentException se o pesoKg for menor ou igual a zero ou se a regiao for nula ou vazia.
- **Requisitos de Teste:**

  1. Implemente uma suíte de testes unitários que valide todos os cenários de negócio.
  2. Crie testes para cada região, incluindo casos de limite (ex: peso exatamente 1 kg e peso ligeiramente acima de 1 kg).
  3. Valide os caminhos de exceção (peso negativo/zero e região inválida).
  4. Utilize nomes de métodos de teste descritivos e siga o padrão Arrange-Act-Assert (AAA).
- **Entregáveis:**

1. Código-fonte da classe CalculadoraFrete.
2. Código-fonte da classe de testes.
3. Relatório de cobertura de código (gerado por ferramentas como JaCoCo, Karma, etc.), com o objetivo de atingir **100% de cobertura de linhas e ramos (branches)**.

Tarefa 2: Validador de Cupom de Desconto

- **Contexto:** Um sistema de marketing precisa de um componente para validar e verificar a elegibilidade de cupons de desconto. Os cupons têm regras de validade, valor mínimo de compra e podem ser de uso único.
- **Requisitos Funcionais:**

  1. Desenvolva uma classe ValidadorCupom com um método validar(Cupom cupom, double valorCompra). O objeto Cupom deve ter os atributos: codigo (String), dataValidade (Data), valorMinimoCompra (double) e usado (boolean).
  2. O método validar deve retornar true se o cupom for válido ou lançar exceções específicas para cada regra de negócio violada.
  3. **Regras de Validação:**
     - Se o cupom já foi usado (usado == true), deve lançar CupomExpiradoException.
     - Se a data atual for posterior à dataValidade do cupom, deve lançar CupomExpiradoException.
     - Se o valorCompra for menor que o valorMinimoCompra do cupom, deve lançar ValorCompraInvalidoException.
- **Requisitos de Teste:**

  1. Escreva testes unitários para o "caminho feliz" (cupom perfeitamente válido).
  2. Crie um teste específico para cada condição de falha, garantindo que a exceção correta é lançada. Utilize asserções como assertThrows.
  3. Inclua casos de borda, como testar um cupom no último dia de validade e um valor de compra exatamente igual ao valor mínimo.
- **Entregáveis:**

1. Código-fonte das classes ValidadorCupom, Cupom e das exceções customizadas.
2. Código-fonte da classe de testes.
3. Relatório de cobertura de código, com o objetivo de atingir **100% de cobertura de linhas e ramos**.

Tarefa 3: Controle de Estoque de Produtos

- **Contexto:** Um sistema de gerenciamento de inventário precisa de um objeto de domínio robusto para representar um Produto, garantindo que as operações de estoque mantenham a integridade dos dados.
- **Requisitos Funcionais:**
    1. Crie uma classe EstoqueProduto.
    2. O construtor deve receber o nomeProduto e a quantidadeInicial. Deve lançar IllegalArgumentException se a quantidade inicial for negativa.
    3. Implemente um método adicionar(int quantidade) que aumenta o estoque. Deve lançar IllegalArgumentException se a quantidade a ser adicionada for menor ou igual a zero.
    4. Implemente um método remover(int quantidade) que diminui o estoque. Deve lançar IllegalArgumentException se a quantidade for menor ou igual a zero.
    5. O método remover também deve lançar uma exceção customizada EstoqueInsuficienteException se a remoção resultar em um estoque negativo.
    6. Inclua um método getQuantidadeAtual() que retorna o estoque corrente.
- **Requisitos de Teste:**
    1. Teste a criação de um produto com estoque inicial válido e a tentativa com estoque negativo.
    2. Valide a adição e remoção de itens com sucesso.
    3. Crie testes específicos para garantir que as exceções são lançadas corretamente ao tentar adicionar/remover quantidades inválidas (zero ou negativo).
    4. Teste o cenário de tentativa de remoção de mais itens do que os disponíveis no estoque, validando a EstoqueInsuficienteException.
- **Entregáveis:**

1. Código-fonte da classe EstoqueProduto e da exceção EstoqueInsuficienteException.
2. Código-fonte da classe de testes.
3. Relatório de cobertura de código, visando **100% de cobertura de linhas e ramos**.

Tarefa 4: Processador de Transações Bancárias

- **Contexto:** Você está trabalhando no backend de um banco digital e precisa implementar a lógica de processamento de transações para uma conta corrente, garantindo que nenhuma regra de negócio seja violada.
- **Requisitos Funcionais:**
    1. Crie uma classe ContaCorrente com um construtor que define o saldoInicial e um limiteChequeEspecial.
    2. O construtor deve lançar IllegalArgumentException se o limite do cheque especial for um valor positivo (deve ser zero ou negativo).
    3. Crie um método sacar(double valor). O saque só pode ser efetuado se saldo - valor >= limiteChequeEspecial. Se o saque for bem-sucedido, o método retorna true. Caso contrário, retorna false sem alterar o saldo.
    4. Crie um método depositar(double valor). O depósito deve ser bem-sucedido e atualizar o saldo.
    5. Ambos os métodos sacar e depositar devem lançar IllegalArgumentException se o valor for menor ou igual a zero.
    6. Adicione um método getSaldo() para consultar o saldo atual.
- **Requisitos de Teste:**
    1. Teste o fluxo padrão: depósito seguido de um saque válido.
    2. Teste um saque que utiliza parte do limite do cheque especial.
    3. Teste um saque que atinge exatamente o limite do cheque especial.
    4. Valide a falha de um saque que ultrapassaria o limite do cheque especial, garantindo que o saldo permaneça inalterado.
    5. Crie testes para os caminhos de exceção: tentar sacar ou depositar valores negativos ou zero.

- **Entregáveis:**
    1. Código-fonte da classe ContaCorrente.
    2. Código-fonte da classe de testes.
    3. Relatório de cobertura de código, com o objetivo de atingir **100% de cobertura de linhas e ramos**.

Tarefa 5: Validador de DTO (Data Transfer Object) de Inscrição

- **Contexto:** Em uma API REST, o endpoint de cadastro de novos usuários recebe um objeto JSON (DTO) com os dados do usuário. Antes de processar e salvar no banco de dados, é crucial validar se todos os dados atendem às regras de negócio da aplicação para garantir a integridade e segurança dos dados.
- **Requisitos Funcionais:**
  1. Crie uma classe de modelo UserRegistrationDTO contendo os campos: username (String), email (String), password (String), e birthDate (String no formato "yyyy-MM-dd").
  2. Desenvolva uma classe de serviço UserValidator com um método validate(UserRegistrationDTO dto).
  3. O método validate deve retornar um objeto ValidationResult, que contém um booleano isValid e uma List<String> de mensagens de erro.
  4. **Regras de Validação:**
     - username: Não pode ser nulo e deve ter entre 3 e 20 caracteres.
     - email: Deve estar em um formato de e-mail válido (use uma expressão regular simples).
     - password: Deve ter no mínimo 8 caracteres, contendo pelo menos uma letra maiúscula, uma letra minúscula e um número.
     - birthDate: O usuário deve ter pelo menos 18 anos de idade com base na data atual.
  5. O validador deve acumular todos os erros encontrados, não parar no primeiro.
- **Requisitos de Teste:**
  1. Crie um teste para o "caminho feliz", com um DTO perfeitamente válido, esperando
     um ValidationResult com isValid como true e lista de erros vazia.

2. Implemente um teste unitário para **cada regra de validação individualmente**. Por exemplo, um teste para um e-mail inválido, um para senha fraca, um para menor de idade, etc.

3. Crie um teste que envie um DTO com múltiplos campos inválidos e verifique se o ValidationResult contém todas as mensagens de erro correspondentes.

4. Teste os casos de borda: um usuário que faz 18 anos hoje, um username com exatamente 3 caracteres.

- **Entregáveis:**

  1. Código-fonte das classes UserRegistrationDTO, UserValidator e ValidationResult.

  2. Código-fonte completo da classe de testes.

  3. Relatório de cobertura de código, visando **100% de cobertura de linhas e ramos**.

Tarefa 6: Serviço de Gerenciamento de Configuração da Aplicação

- **Contexto:** Toda aplicação de sistema precisa carregar configurações de diversas fontes (arquivos, variáveis de ambiente). Um componente centralizado que carrega, valida e fornece essas configurações com valores padrão é essencial para a robustez do sistema.
- **Requisitos Funcionais:**
  1. Crie uma classe AppConfigService que é inicializada com um Map<String, String> representando as chaves e valores de configuração.
  2. Implemente os seguintes métodos para obter valores de configuração:
     - getString(String key, String defaultValue): Retorna o valor da chave ou o padrão se a chave não existir.
     - getInt(String key, int defaultValue): Retorna o valor da chave convertido para inteiro. Se a chave não existir, retorna o padrão. Se o valor não for um inteiro válido, lança InvalidConfigurationException.
     - getBoolean(String key, boolean defaultValue): Retorna true para valores como "true", "1", "yes". Retorna o padrão para chaves ausentes.
  3. Implemente um método require(String key) que retorna o valor da chave, mas lança MissingConfigurationException se a chave não existir.
- **Requisitos de Teste:**
  1. Teste o carregamento de cada tipo de dado (String, int, boolean) com sucesso.
  2. Teste o comportamento de cada método get quando a chave **não existe**, garantindo que o valor padrão seja retornado.
  3. Teste o método getInt com um valor que não pode ser convertido para número (ex: "abc"), verificando se a InvalidConfigurationException é lançada.

4. Teste o método require para uma chave existente e para uma chave ausente, validando a exceção no segundo caso.

- **Entregáveis:**

  1. Código-fonte da classe AppConfigService e das exceções customizadas.
  2. Código-fonte da classe de testes.
  3. Relatório de cobertura de código, com o objetivo de atingir **100% de cobertura**.

Tarefa 7: Gerenciador de Estado de Assinatura de Usuário

- **Contexto:** Em um serviço de SaaS (Software as a Service), o estado da assinatura de um usuário (ativa, cancelada, vencida) dita o acesso aos recursos. É necessário um gerenciador que aplique as regras de transição de estado de forma segura.
- **Requisitos Funcionais:**

  1. Crie uma classe Subscription com os atributos status (um enum com valores ACTIVE, CANCELED, EXPIRED) e expirationDate (Data).
  2. Crie uma classe de serviço SubscriptionManager.
  3. Implemente os seguintes métodos que recebem um objeto Subscription e o modificam:
     - cancel(Subscription sub): Só pode cancelar uma assinatura ACTIVE. Se bem-sucedido, muda o status para CANCELED. Se o status já for CANCELED ou EXPIRED, lança InvalidStateException.
     - renew(Subscription sub, Date newExpirationDate): Só pode renovar uma assinatura ACTIVE ou EXPIRED. Atualiza a data de expiração e define o status como ACTIVE. Lança InvalidStateException se a assinatura estiver CANCELED.
     - checkStatus(Subscription sub, Date currentDate): Se o status for ACTIVE e a currentDate for posterior à expirationDate, muda o status para EXPIRED.
- **Requisitos de Teste:**

  1. Teste cada transição de estado válida: de ACTIVE para CANCELED, de ACTIVE para EXPIRED (via checkStatus), de EXPIRED para ACTIVE (via renew).
  2. Teste cada tentativa de transição de estado **inválida**, garantindo que a InvalidStateException seja lançada (ex: tentar cancelar uma assinatura já cancelada).

3. Teste o método checkStatus com datas diferentes: um dia antes, no dia e um dia depois da expiração, validando que o status só muda quando esperado. A passagem da currentDate como parâmetro é fundamental para a testabilidade.

- **Entregáveis:**

  1. Código-fonte das classes Subscription, SubscriptionManager, o enum de status e a exceção.
  2. Código-fonte da classe de testes.
  3. Relatório de cobertura de código, visando **100% de cobertura**.

Tarefa 8: Parser de Arquivo de Log Simples

- **Contexto:** Um sistema gera logs em um formato de texto específico. Uma ferramenta de monitoramento precisa de um parser que leia uma linha de log e a transforme em um objeto estruturado para análise posterior.
- **Requisitos Funcionais:**
    1. Crie uma classe LogEntry com os campos: timestamp (Data/Hora), level (String, ex: "INFO", "ERROR"), e message (String).
    2. Crie uma classe LogParser com um método parse(String logLine).
    3. O formato da linha de log é: [YYYY-MM-DDTHH:mm:ss] LEVEL: Mensagem de log.
        - Exemplo: [2025-10-27T10:00:00] INFO: User logged in successfully.
    4. O método parse deve retornar um objeto LogEntry preenchido.
    5. Se a linha não corresponder ao formato esperado (faltando colchetes, formato de data inválido, etc.), o método deve lançar LogParsingException.
- **Requisitos de Teste:**
    1. Teste o parsing de linhas válidas com diferentes níveis de log (INFO, WARN, ERROR).
    2. Teste linhas com mensagens contendo caracteres variados.
    3. Crie testes para cada possível formato inválido: data incorreta, sem o nível de log, sem os colchetes, linha vazia ou nula. Em todos esses casos, verifique se LogParsingException é lançada.
    4. Teste um caso de borda onde a mensagem de log contém a palavra "ERROR:", para garantir que o parser não se confunda.
- **Entregáveis:**

1. Código-fonte das
classes LogEntry, LogParser e LogParsingException.
2. Código-fonte da suíte de testes.
3. Relatório de cobertura de código, com meta de **100%**.

Tarefa 9: Serviço de Cálculo de Carrinho de Compras com Vouchers

- **Contexto:** Em um sistema de e-commerce, o cálculo final do carrinho de compras deve aplicar descontos de vouchers. A lógica precisa lidar com diferentes tipos de vouchers (percentual vs. valor fixo) e garantir que as regras sejam aplicadas corretamente.
- **Requisitos Funcionais:**
    1. Crie uma classe CartItem com price e quantity.
    2. Crie uma classe Voucher com type (enum: PERCENTAGE, FIXED_AMOUNT) e value (double).
    3. Desenvolva um serviço CartCalculator com um método calculateTotal(List<CartItem> items, Voucher voucher).
    4. **Lógica de Cálculo:**
        - Primeiro, calcule o subtotal (soma de price * quantity de todos os itens).
        - Se o voucher não for nulo, aplique o desconto:
            - FIXED_AMOUNT: Subtraia o value do voucher do subtotal.
            - PERCENTAGE: Calcule o desconto (subtotal * value / 100) e subtraia do subtotal.
        - O total final nunca pode ser menor que zero. Se o desconto for maior que o subtotal, o total deve ser zero.
    5. O método deve lançar IllegalArgumentException se a lista de itens for nula.
- **Requisitos de Teste:**
    1. Teste o cálculo com uma lista de itens, mas sem voucher (voucher nulo).
    2. Teste com um voucher de valor fixo que resulta em um total positivo.
    3. Teste com um voucher de valor percentual.

4. Teste o caso de borda onde o desconto de valor fixo é **maior** que o subtotal, esperando um resultado final de 0.
5. Teste com uma lista de itens vazia, esperando um resultado de 0.
6. Teste passando uma lista de itens nula, esperando IllegalArgumentException.

- **Entregáveis:**
  1. Código-fonte das classes CartItem, Voucher, enum e CartCalculator.
  2. Código-fonte da classe de testes.
  3. Relatório de cobertura de código, visando **100%**.

Tarefa 10: Gerador de Slug para URL Amigável

- **Contexto:** Um blog ou CMS precisa converter títulos de artigos em "slugs" amigáveis para URLs (ex: "Olá Mundo em 2025!" se torna "ola-mundo-em-2025"). Esse componente é fundamental para SEO e legibilidade.
- **Requisitos Funcionais:**
    1. Crie uma classe SlugGenerator com um método generate(String title).
    2. **Regras de Conversão:**
        - Converter toda a string para minúsculas.
        - Substituir caracteres acentuados por seus equivalentes sem acento (ex: 'á' -> 'a', 'ç' -> 'c').
        - Substituir qualquer caractere que não seja letra (a-z), número (0-9) ou hífen por um hífen.
        - Substituir múltiplos hífens consecutivos por um único hífen.
        - Remover hífens no início ou no final da string.
    3. Se o título for nulo ou vazio, o método deve retornar uma string vazia.
- **Requisitos de Teste:**
    1. Teste um título simples como "Meu Primeiro Artigo".
    2. Teste um título com acentuação e caracteres especiais, como "Acentuação e Pontuação: Guia Completo!".
    3. Teste um título com múltiplos espaços e hífens, como "Teste -- com -- hifens --- extras".
    4. Teste títulos que resultariam em hífens no início ou fim após a limpeza, como "!Importante!" ou " Artigo ".
    5. Teste com entradas nulas e vazias.
- **Entregáveis:**
    1. Código-fonte da classe SlugGenerator.

2. Código-fonte da classe de testes.

3. Relatório de cobertura de código, com meta de **100% de cobertura**.