



Avaliação

Instruções iniciais.

Pensando em oferecer mais flexibilidade, segurança e transparência, o banco atualizou a forma de funcionamento de suas contas. Agora, cada conta possui um limite definido, permitindo ao cliente realizar saques mesmo quando o saldo não é suficiente, desde que não ultrapasse o limite negativo estabelecido no momento da abertura da conta. Além disso, toda operação de saque, depósito ou tentativa de saque acima do limite deve gerar um registro de operação, que ficará disponível para consulta por meio da emissão de extratos individuais por conta, por cliente e gerais do banco.

Sua tarefa nesta avaliação será implementar essa nova versão da conta, garantindo que:

- o limite seja corretamente considerado em operações de saque;
- operações que ultrapassem o limite gerem um registro de falha;
- depósitos e transferências também gerem registros apropriados;
- extratos possam exibir todas as operações (inclusive falhas), sempre em ordem decrescente de data e hora.

Organização de arquivos

Para garantir boa organização e modularidade do projeto, todas as classes desta avaliação devem ser implementadas em arquivos separados.

Crie os seguintes arquivos TypeScript:

- operacao.ts: Contém a classe Operacao, representa qualquer ação realizada em uma conta (depósito, saque, transferência ou falha);
- conta.ts, cliente.ts e banco.ts: transfira suas implementações prévias para que cada classe fique em um arquivo;

- `testes.ts` (já fornecido): arquivo que contém testes para validar a implementação. O aluno não deve alterar este arquivo. Apenas executá-lo.

Todos esses arquivos devem ser salvos em um repositório ao final e enviados ao professor.

Dinâmica e organização

Os alunos devem realizar a prova em dupla permutando o controle do computador a cada intervalo estipulado pelo professor. Ao final de cada intervalo, um alarme soará e o membro que estava “digitando” deve ceder o espaço ao outro aluno.

Ao final da avaliação, um dos alunos deve criar um repositório com os arquivos e disponibilizá-los todos os arquivos como resposta à avaliação.

Observações:

Presume-se honestidade acadêmica;

A interpretação das questões faz parte da avaliação.

Questões

1) Classe Operação:

Crie uma classe `Operacao`, para representar ações realizadas em uma conta. Ela deve conter os atributos privados inicializados no construtor:

- `_id` (number): identificador único da operação;
- `_conta` (`Conta`): referência à conta onde a operação ocorreu;
- `_tipo` (string): tipo da operação. Os valores possíveis são:
 - "CRÉDITO" — quando um depósito ou crédito de transferência for realizado;
 - "DÉBITO" — quando um saque ou débito de transferência for realizado;
 - "FALHA" — quando uma tentativa de saque ou transferência ultrapassar o limite permitido.
- `_valor` (number): valor monetário envolvido na operação;
- `_descricao` (string): texto explicativo sobre a operação (veja abaixo);
- `_dataHora` (`Date`): data e hora exatas em que a operação foi criada.

A classe deve disponibilizar os métodos:

- get id()
- get conta()
- get tipo()
- get valor()
- get descricao()
- get dataHora()
- set descricao(texto: string)

O atributo descrição deve apresentar uma descrição coerente com o tipo e o contexto. Abaixo seguem algumas sugestões:

- "Depósito na conta XXXXX";
- "Saque na conta XXXXX";
- "Transferência para conta XXXXX";
- "Transferência recebida da conta XXXXX";
- "Saque não autorizado: limite de saldo excedido";
- "Falha na transferência: saque não autorizado (limite excedido)".

2) Alterações na Classe Conta: Limite e Registro de Operações

2.1) Crie os seguintes atributos:

- `_operacoes: Operacao[]` — armazena o histórico de operações;
- `_limite: number` — definido no construtor; representa quanto a conta pode ficar negativa;
- `_idOperacaoAtual: number` — contador interno para gerar IDs únicos. Deve funcionar como um *autoincremento* (ex.: iniciar em 1 e aumentar a cada operação criada).

2.2) Métodos devem retornar operações

Os métodos abaixo devem retornar instâncias de Operacao:

- `sacar(valor: number): Operacao`
- `depositar(valor: number): Operacao`
- `transferir(contaDestino: Conta, valor: number): Operacao[]`
(um array com uma ou duas operações)

2.3) Regra do limite

A conta pode ficar negativa até o valor do limite. O saque somente deve ocorrer se:

$$\text{saldoApós} = \text{saldoAtual} - \text{valor} \geq -\text{_limite}$$

Caso contrário:

- o saldo não é alterado;
- uma operação do tipo "FALHA" deve ser retornada.

Exemplos:

- Limite: 200, Saldo: 100, Saque: 150 → permitido (saldo final: -50);
 - Limite: 100, Saldo: -50, Saque: 100 → recusado (saldo ficaria -150).
-

2.4) Registro das operações

Toda operação criada deve ser colocada na primeira posição do array `_operacoes` usando `unshift()`. Além disso

Saque

- Permitido → retornar operação "DÉBITO";
- Recusado → retornar operação "FALHA".

Depósito

- Retornar operação "CRÉDITO";

Transferência

- Se o saque da conta de origem falhar:
 - não depositar na conta destino;
 - retornar apenas a operação "FALHA".
- Se o saque for bem-sucedido:
 - registrar DÉBITO ("Transferência para ") no array da conta de origem (`this`);
 - registrar "CRÉDITO" no array na conta de destino;

- retornar ambas as operações em um array.
-

Observação importante

A cada alteração, rode os testes fornecidos para validar a implementação. Ative também as três primeiras questões de teste.

3) Ajustes na Classe Banco: Registro Global e Emissão de Extratos

Implemente as alterações abaixo na classe Banco, garantindo que todas as operações realizadas nas contas também sejam registradas e possam ser consultadas posteriormente.

3.1) Atributos privados obrigatórios

Crie (ou confirme a existência) destes atributos privados:

- `_contas: Conta[]` — lista de contas do banco.
- `_clientes: Cliente[]` — lista de clientes cadastrados.
- `_operacoes: Operacao[]` — registro global contendo todas as operações realizadas no banco, inclusive operações de falha.
- `_idClienteAtual: number` — contador interno para IDs de clientes.
- `_idContaAtual: number` — contador interno para IDs de contas.

Os atributos de ID devem funcionar como autoincremento, de forma semelhante ao mecanismo demonstrado em sala.

3.2) Registro global de operações

Toda vez que uma operação for criada por uma conta, o banco deve registrá-la em `_operacoes`. Ou seja, após chamar `conta.sacar(...)`, `conta.depositar(...)`, `conta.transferir(...)` o banco deve:

- receber a operação (ou array de operações) da classe conta em questão;
- armazená-la na primeira posição do array `_operacoes`;
- Falhas de saque e falhas de transferência também devem ser armazenadas.

3.3) Métodos de operações no banco

Ajuste os seguintes métodos:

- `sacar(numeroConta: string, valor: number): void`
 - Deve chamar o método `sacar()` da conta e registrar a operação retornada no array global do banco.
- `depositar(numeroConta: string, valor: number): void`
 - Deve chamar o método `depositar()` da conta e registrar a operação retornada.
- `transferir(numeroOrigem: string, numeroDestino: string, valor: number): void`
 - Deve chamar `transferir()` da conta de origem e registrar o array retornado (uma operação de falha ou duas de sucesso).

3.4) Emissão de Extratos

Implemente os métodos abaixo para consulta dos registros.

a) `consultarExtratoConta(numeroConta: string): Operacao[]`

- Retorna todas as operações da conta informada;
- Pode reutilizar diretamente o array `_operacoes` lido através de um “get” implementado na conta;
- Deve vir em ordem decrescente (garantida ao inserir sempre no início do array).

b) `consultarExtratoCliente(cpf: string): Operacao[]`

- Deve buscar todas as contas associadas ao cliente;
- Coletar as operações de todas elas;
- Retornar um único array com todas as operações em ordem decrescente;

c) `consultarExtratoGeral(): Operacao[]`

- Retorna diretamente o array `_operacoes`.

- Deve vir em ordem decrescente (garantida ao inserir sempre no início do array).
 - Por óbvio, contém todas as operações realizadas no banco de todas as contas de todos os clientes: depósitos, saques, transferências e falhas
-

Execução dos testes e visão orientada a objetos

Além das classes Cliente, Conta, Operacao e Banco, esta avaliação inclui também uma classe de testes (por exemplo, TestesBancoLimite).

Essa classe de testes também é orientada a objetos:

- Possui atributos privados, como:
 - o banco (`_banco`),
 - os pontos obtidos (`_pontosObtidos`)
 - e o total de pontos (`_pontosTotais`);
- Possui métodos privados, responsáveis por:
 - configurar o banco e zerar seu estado (`iniciarBanco()`);
 - criar clientes e contas dentro do banco;
 - executar cada teste específico (Teste 1, Teste 2, ..., Teste 5);
- Possui um método público, normalmente chamado `executar()`, que:
 - chama internamente os métodos de teste,
 - soma a pontuação obtida,
 - e exibe no final uma nota simulada de 0 a 10.

Para utilizar essa classe:

1. Certifique-se de que as chamadas dos testes no método `executar()` estão descomentadas (por exemplo: `this.teste1Deposito();`,
`this.teste2SaqueComLimite();` etc.).
2. Compile/execute o projeto normalmente.
3. Ao rodar a classe de testes, você verá no console:

- quais testes passaram ou falharam;
- quantos pontos foram obtidos em cada teste;
- e uma estimativa da nota final (0 a 10) de acordo com as funcionalidades implementadas.

Importante:

- a conta é um objeto,
 - o banco é um objeto,
 - o cliente é um objeto,
 - a operação é um objeto
- e o próprio conjunto de testes também é um objeto.**

Ou seja, toda a lógica da prova — do sistema bancário até a correção automática — está sendo construída com classes, objetos, atributos e métodos, reforçando na prática os conceitos de Programação Orientada a Objetos.