Programação Orientada a Objetos em C#

Conceitos fundamentais e aplicações práticas para desenvolvimento de software moderno

Desenvolvimento de Sistemas Orientados a Objetos · 2025

Classes, Objetos e Construtores — Teoria

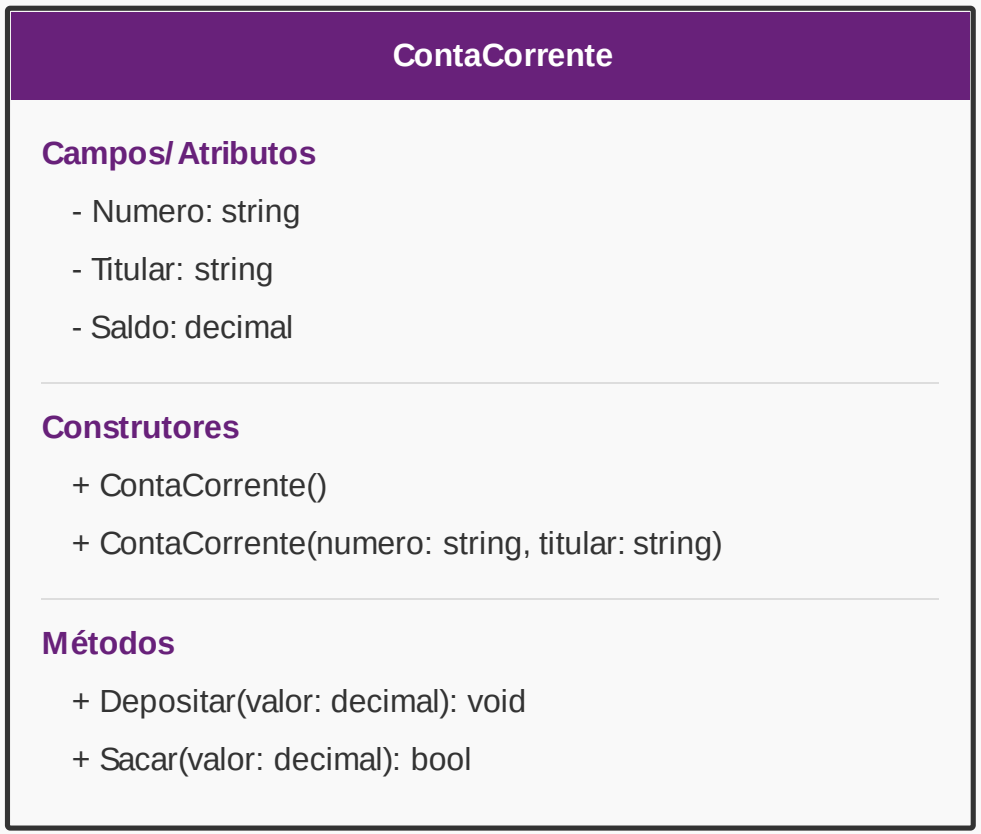
Conceitos de objeto e classe: Uma classe é um modelo/template que define propriedades e comportamentos. Um objeto é uma instância concreta de uma classe.

Membros de uma classe: Campos (variáveis), propriedades (com getters/setters) e métodos (funções).

Construtores: Métodos especiais chamados quando um objeto é criado, permitindo inicializar seus campos e estado.

Palavra-chave this: Referência ao objeto atual, útil para diferenciar campos da classe de parâmetros com mesmo nome.

Diagrama UML de Classe



Classes, Objetos e Construtores — Exercício Prático

Exercício: Criando uma Classe ContaCorrente

1. Crie a classe ContaCorrente com propriedades para Número, Titular e Saldo.
2. Adicione dois construtores: um padrão e outro que receba número e titular como parâmetros.
3. Instancie dois objetos ContaCorrente utilizando os diferentes construtores.

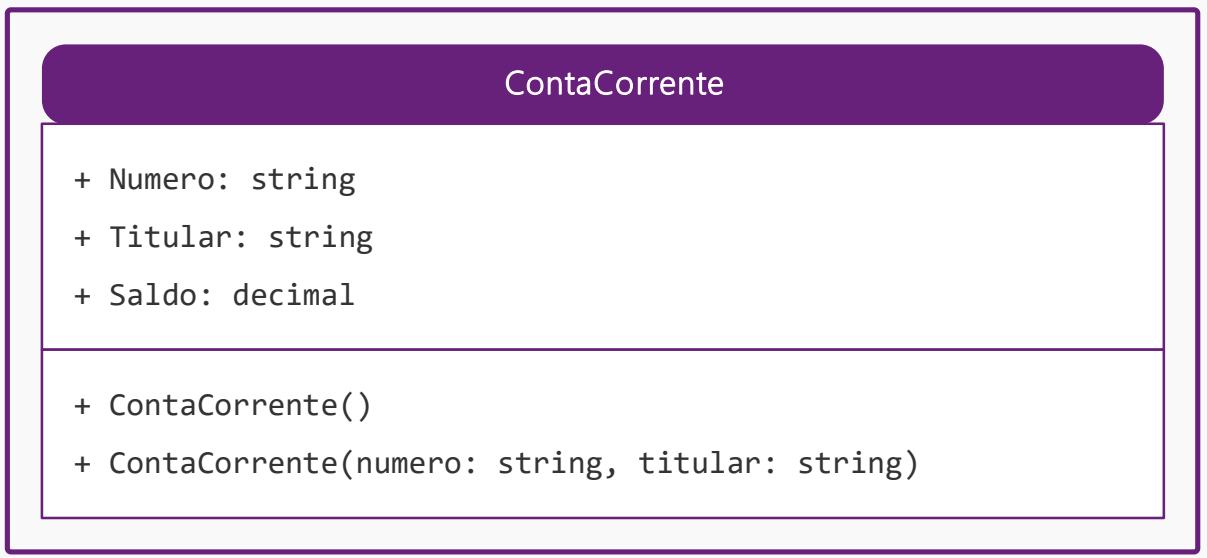
Dica de implementação:

Use a palavra-chave this no construtor com parâmetros para referenciar os campos da classe e diferenciá-los dos parâmetros recebidos.

Entrega do exercício:

Crie um arquivo ContaCorrente.cs com sua implementação e outro arquivo Program.cs demonstrando o uso da classe.

Estrutura Esperada



Exemplos de uso esperados:

- Criar uma conta usando o construtor padrão e definir suas propriedades
- Criar uma conta usando o construtor com parâmetros
- Mostrar informações das contas criadas

Saída esperada do programa:

```
Conta 1: João Silva, Saldo: 0
Conta 2: Maria Santos, Saldo: 0
```

Encapsulamento e Propriedades — Teoria

Modificadores de acesso: Controlam a visibilidade e o acesso aos membros da classe.

- public: Acesso sem restrições, de qualquer lugar.
- private: Acesso somente dentro da própria classe.
- protected: Acesso na própria classe e classes derivadas.

Propriedades (get/set): Métodos especiais que fornecem acesso controlado aos campos privados, possibilitando validação e lógica adicional.

Auto-Properties: Sintaxe simplificada que cria automaticamente um campo privado de suporte. Exemplo: public string Nome { get; set; }

Vantagens do encapsulamento: Protege os dados, garante consistência, permite validação e facilita manutenção futura do código.

Encapsulamento em Ação

public

```
// Acesso direto (sem encapsulamento)
public decimal Saldo;
// Qualquer código externo pode modificar
conta.Saldo = -1000; // Permitido 😲
```

private + Propriedades

```
// Campo encapsulado
private decimal _saldo;
// Propriedade com validação
public decimal Saldo {
get { return _saldo; }
private set { /* validação aqui */ }
}
// Métodos controlados
public void Depositar(decimal valor) {
if (valor > 0)
_saldo += valor;
}
```

Encapsulamento e Propriedades — Exercício Prático

Exercício: Encapsulando o campo Saldo

1. Modifique a classe ContaCorrente tornando o campo Saldo privado (acessível apenas dentro da classe).
2. Crie um método Depositar que receba um valor e adicione ao saldo (com validação).
3. Implemente um método Sacar que subtraia um valor do saldo apenas se houver saldo suficiente.
4. Crie uma propriedade de leitura para o Saldo que permita consultar, mas não modificar diretamente o valor.

Benefícios do encapsulamento:

Ao encapsular o saldo, você protege a integridade dos dados garantindo que todas as modificações passem por validações e regras de negócio definidas na classe.

Estrutura da Classe (UML)

ContaCorrente

- _saldo: decimal
+ Numero: string
+ Titular: string
+ Saldo: decimal (somente leitura)

+ ContaCorrente(numero: string, titular: string)
+ Depositar(valor: decimal): bool
+ Sacar(valor: decimal): bool

Regras de Implementação:

- O método Depositar deve validar se o valor é positivo
- O método Sacar deve verificar se há saldo suficiente
- Ambos os métodos devem retornar um bool indicando sucesso/falha

Dica de Teste:

Crie uma instância da classe e teste as operações de depósito e saque com valores válidos e inválidos para verificar se o encapsulamento está funcionando corretamente

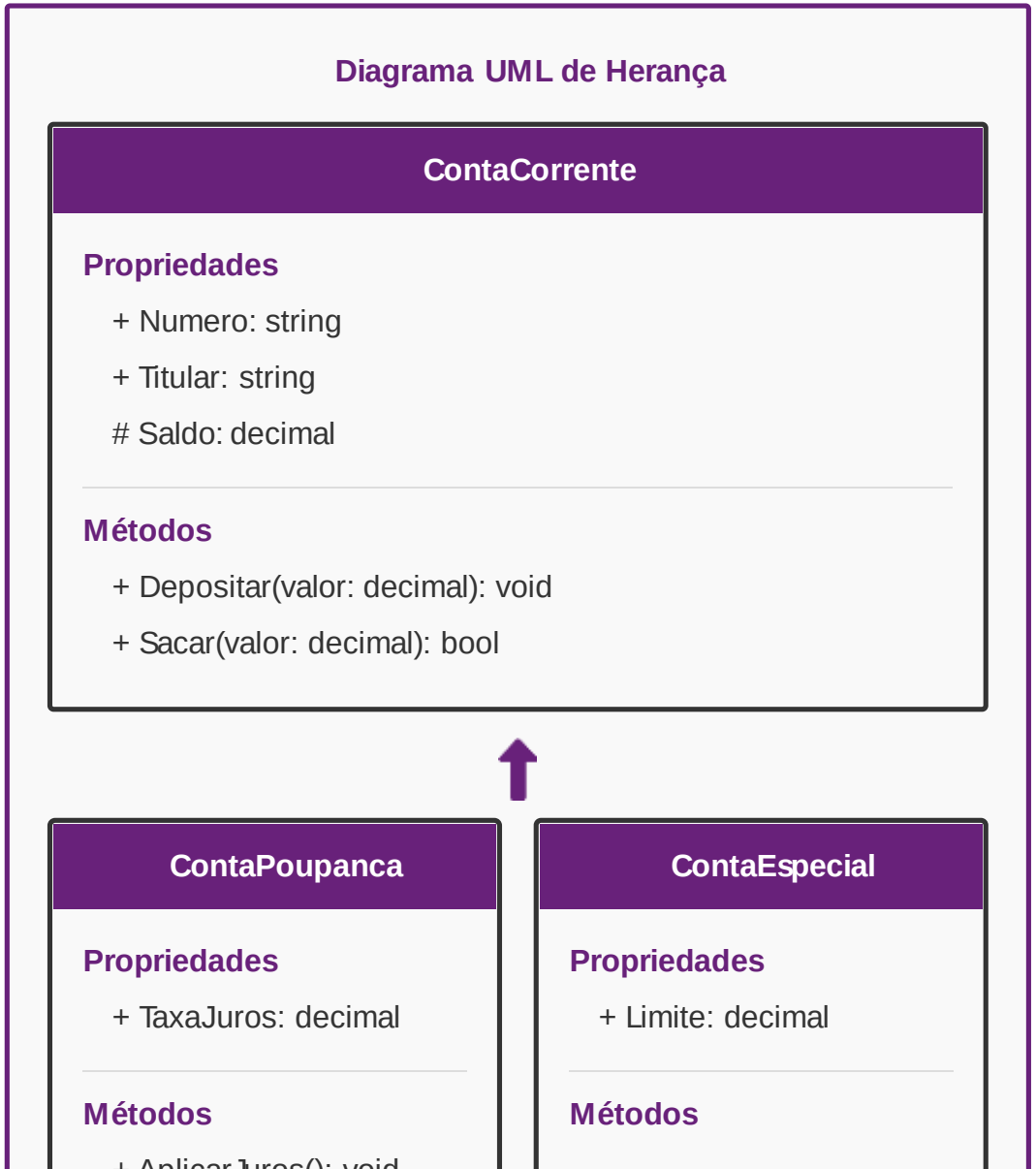
Herança e Polimorfismo (I) — Teoria

Conceitos de herança: Mecanismo que permite criar uma nova classe (derivada) a partir de uma classe existente (base), herdando seus campos, propriedades e métodos.

Classe base e derivadas: A classe base (superclasse) define características comuns. As classes derivadas (subclasses) herdam essas características e adicionam suas próprias.

Sintaxe em C#: class ClasseDerivada : ClasseBase - O símbolo ":" indica herança em C#.

Palavra-chave base: Referência à classe base, permite acessar membros da classe pai e chamar seu construtor (base()).



Herança e Polimorfismo (I) — Exercício Prático

Exercício: Criando Classes Derivadas de ContaCorrente

1. Crie a classe ContaPoupanca que herda de ContaCorrente e adicione uma propriedade Taxa para rendimentos.

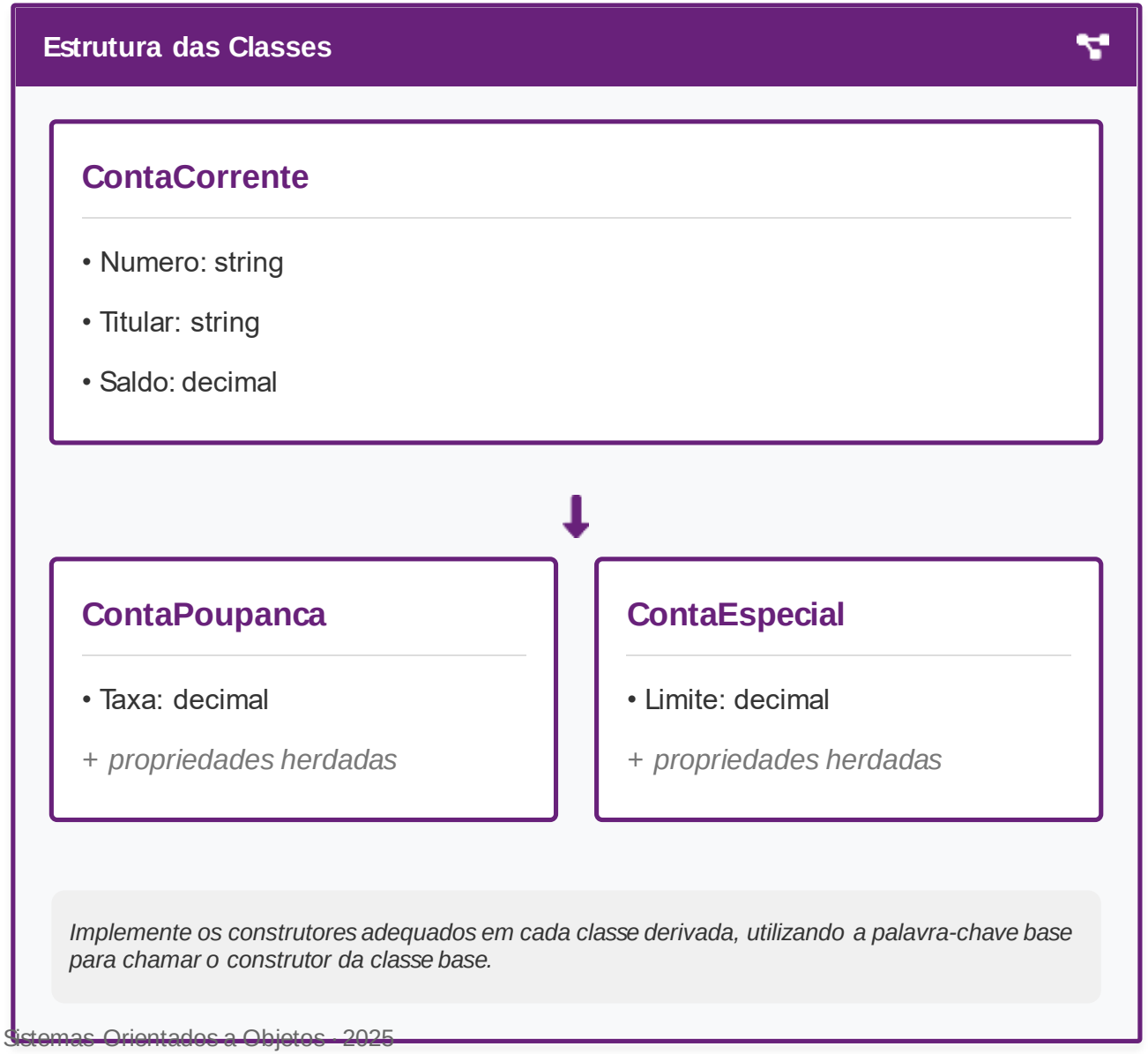
2. Crie a classe ContaEspecial que herda de ContaCorrente e adicione uma propriedade Limite para permitir saldo negativo.

3. Utilize a palavra-chave base para chamar o construtor da classe base em ambas as classes derivadas.

Dica de implementação:

Lembre-se de que a herança permite reutilizar o código da classe base. Use base() para aproveitar os construtores já implementados na classe ContaCorrente.

Estrutura das Classes



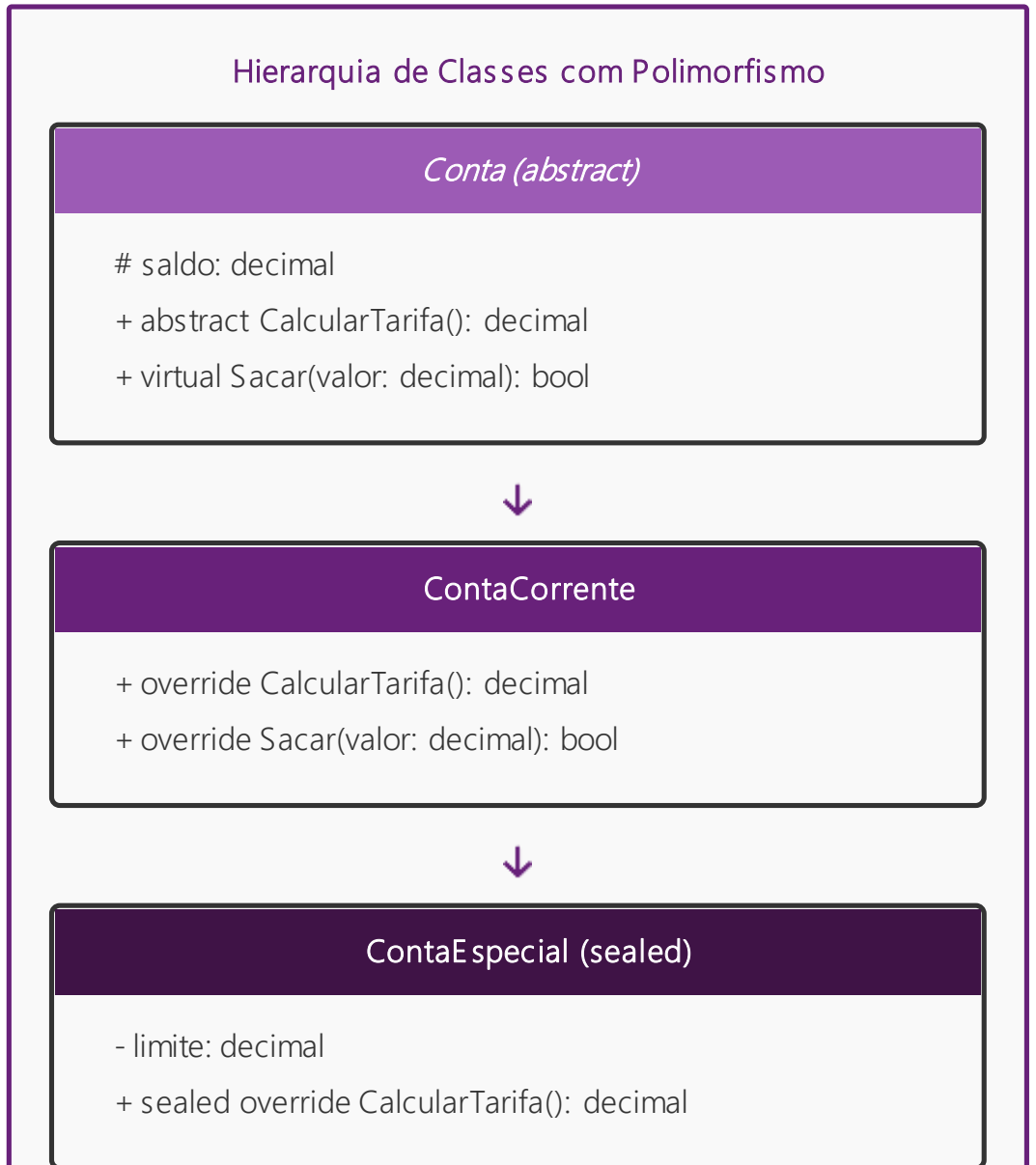
Herança e Polimorfismo (II) — Teoria

Métodos virtuais e override: Métodos marcados como virtual podem ser sobrescritos nas classes derivadas usando override, permitindo diferentes implementações do mesmo método.

Classes e métodos abstratos: Classes com abstract não podem ser instanciadas diretamente e podem conter métodos abstratos (sem implementação) que devem ser implementados pelas classes derivadas.

Sealed classes: Classes marcadas com sealed não podem ser herdadas. Métodos sealed em classes derivadas não podem ser sobrescritos em subclasses.

Polimorfismo: Permite tratar objetos de classes derivadas através de referências da classe base, chamando automaticamente a implementação correta do método override.



Herança e Polimorfismo (II) — Exercício Prático

Exercício: Implementação Polimórfica de CalcularTarifa()

1. Implemente o método CalcularTarifa() na classe ContaCorrente como virtual.
2. Sobrescreva (override) o método nas classes ContaPoupanca e ContaEspecial com lógicas distintas.
3. Demonstre o polimorfismo chamando o método através de referências à classe base.
4. Defina uma lógica de negócio específica para cada tipo de conta:

   ContaCorrente: tarifa fixa
   ContaPoupanca: tarifa reduzida
   ContaEspecial: tarifa proporcional ao limite

Conceito-chave:

O polimorfismo permite que objetos de diferentes classes derivadas respondam de maneira única quando acessados através da mesma interface (classe base). Use virtual na base e override nas derivadas.

Espaço para Implementação

Estrutura Básica:

ContaCorrente (Classe Base)

Implemente o método virtual:

- Defina um método public virtual decimal CalcularTarifa()
- Retorne um valor fixo de tarifa

ContaPoupanca (Classe Derivada)

Sobrescreva com uma lógica diferente:

- Use public override decimal CalcularTarifa()
- Implemente uma tarifa mais baixa que a conta corrente comum

ContaEspecial (Classe Derivada)

Sobrescreva com uma lógica dependente do limite:

- Use public override decimal CalcularTarifa()
- Calcule a tarifa com base na propriedade Limite

Demonstração do Polimorfismo

Use a classe base para referenciar objetos de tipos derivados e

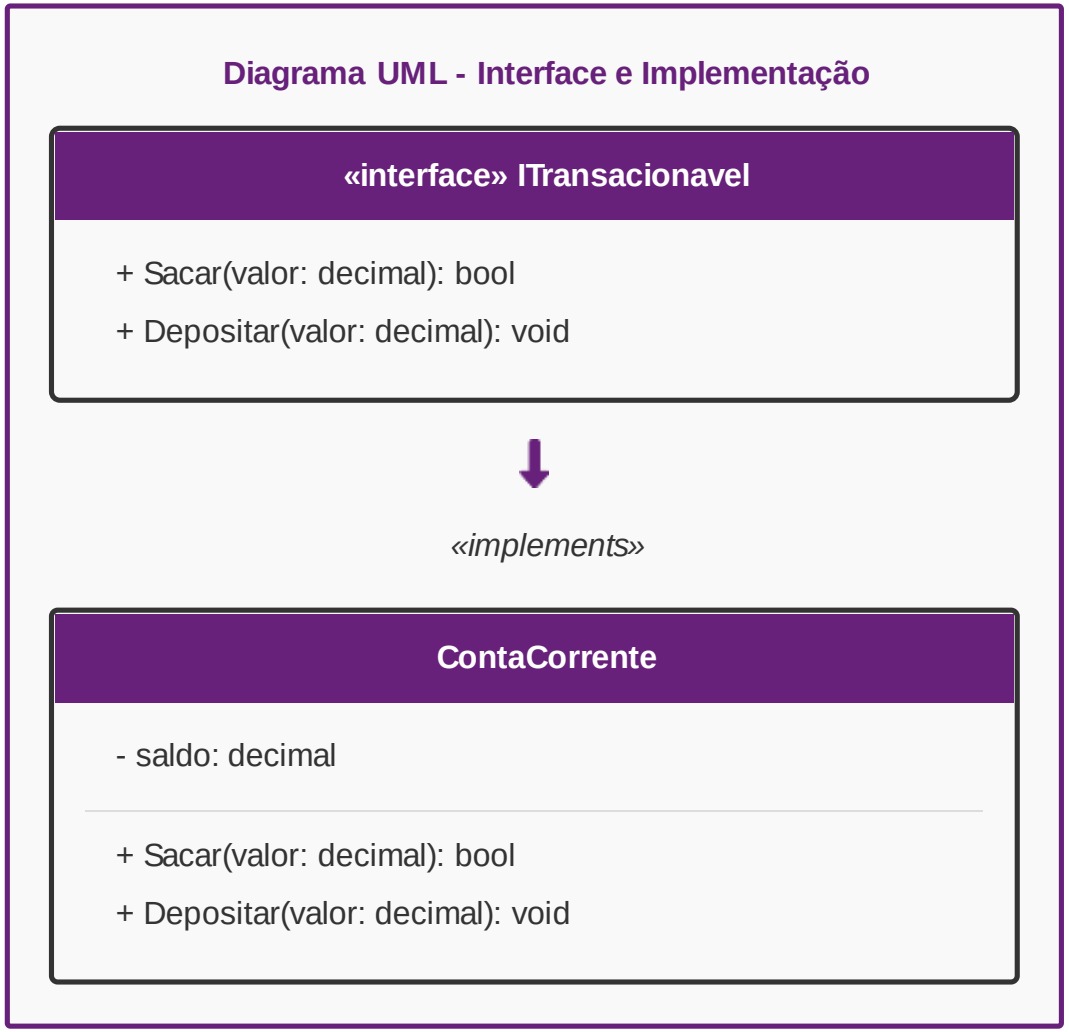
Interfaces e Acoplamento — Teoria

Interface como contrato: Define um conjunto de métodos e propriedades que uma classe deve implementar. Estabelece um "contrato" que a classe concreta deve cumprir.

Implementação de múltiplas interfaces: Uma classe pode implementar várias interfaces simultaneamente, superando a limitação da herança única em C#.

Inversão de dependência: Princípio que sugere depender de abstrações (interfaces), não de implementações concretas, reduzindo o acoplamento entre componentes.

Baixo acoplamento: Interfaces permitem que componentes interajam sem conhecer detalhes internos uns dos outros, facilitando manutenção e testes.



Interfaces e Acoplamento — Exercício Prático

Exercício: Criando e Implementando Interfaces

1. Crie a interface ITransacionavel com os métodos Sacar e Depositar.
2. Implemente a interface na classe ContaCorrente (e suas classes derivadas).
3. Adicione validações específicas para cada tipo de conta nas implementações dos métodos.

Benefício do uso de interfaces:

Com interfaces, você pode trabalhar com diferentes tipos de conta de forma polimórfica. Isso permite que um método aceite qualquer objeto ITransacionavel, independente da classe concreta.

Orientações para o Exercício

- A interface deve declarar apenas as assinaturas dos métodos, sem implementação.
- O método Sacar() deve retornar um bool indicando se a operação foi bem-sucedida.
- O método Depositar() deve ser void e aceitar um valor decimal como parâmetro.
- Lembre-se que as validações serão diferentes para cada tipo de conta:
  - ContaCorrente: validar saldo disponível
  - ContaPoupanca: validar saldo e considerar taxa
  - ContaEspecial: considerar limite disponível além do saldo
- Crie uma classe para testar o polimorfismo, que utilize as contas apenas através da interface ITransacionavel.

Conclusão e Referências

Resumo dos Principais Conceitos

Classes e Objetos: Classes definem o modelo de dados, enquanto objetos são instâncias com estado e comportamento.

Encapsulamento: Oculta implementação interna, expondo apenas o necessário através de interfaces bem definidas (propriedades e métodos).

Herança: Permite que classes derivadas herdem características de classes base, promovendo reuso de código.

Polimorfismo: Capacidade de tratar objetos derivados como objetos de sua classe base, permitindo comportamentos distintos.

Interfaces: Contratos que classes implementam, definindo comportamento sem especificar implementação.

Dominar estes conceitos fundamentais de POO permite o desenvolvimento de sistemas mais organizados, flexíveis e reutilizáveis em C#, facilitando a manutenção e evolução do software.

Referências Úteis

Microsoft Learn
Documentação Oficial C#

W3Schools
Tutoriais de C#

Exercícios C#
Exercícios Práticos

GitHub
Exemplos .NET