FMX Academy

INFORMAÇÕES		
MÓDULO	AULA	INSTRUTOR
01 Lógica de Programação e C# Básico	07 - Abstração: Interfaces e Classes Abstratas	Guilherme Paracatu

1. Revisão rápida e dúvidas

Recapitulando Git e GitHub!

Quem lembra qual o commando para iniciar um repositório?

Quem consegue explicar o que é Git?

Quem consegue explicar o que é GitHub?

2. Abstração: Focando no Essencial

Objetivo: Apresentar o conceito de Abstração e sua importância na POO, focando no "o quê" em vez do "como".

2.1. O Que é Abstração?

Agora, o quarto pilar: **Abstração**. Pensem em um mapa. Um mapa rodoviário não mostra cada árvore, cada casa. Ele mostra apenas as informações essenciais para dirigir: estradas, cidades, talvez alguns pontos de interesse. Ele **abstrai** os detalhes desnecessários.

No contexto da POO, **Abstração** é o processo de **identificar as características e comportamentos essenciais de um objeto ou sistema**, enquanto **ignora os detalhes de implementação menos importantes**. É focar no 'o quê' (o que algo faz) e adiar o 'como' (como ele faz).

É como um contrato: 'Você precisa ser capaz de fazer X e Y', mas não diz 'como' você fará X e Y.

2.2. Por Que Abstrair?

Por que nos preocupamos em abstrair? Para simplificar a complexidade! Em sistemas grandes, não queremos que cada parte do código precise saber todos os mínimos detalhes de como outra parte funciona. Queremos apenas que ela saiba o que esperar (o 'contrato').

Vantagens da Abstração:

- **Simplificação:** Reduz a complexidade, tornando o código mais fácil de entender e gerenciar.
- **Flexibilidade:** Permite trocar implementações sem afetar o código que usa a abstração.
- **Segurança:** Esconde detalhes internos que não precisam ser expostos.
- **Colaboração:** Diferentes times podem trabalhar em implementações diferentes de um mesmo contrato."
  - **Analogia:** "Pensem na interface de um aplicativo de smartphone. Todos os apps de mensagens (WhatsApp, Telegram) têm um botão para 'enviar mensagem'. Vocês não precisam saber como cada um 'envia a mensagem' por dentro (detalhes da implementação da rede, criptografia). Basta saber que existe uma função 'enviar mensagem'. Esse 'enviar mensagem' é uma abstração.

3. Interfaces: O Contrato Puro

Objetivo: Compreender o que são Interfaces, como declará-las, implementá-las e seu papel no polimorfismo e na flexibilidade do código.

3.1 – O Que é uma Interface?

- Em C#, a principal forma de criar abstração é através de **Interfaces**. Pensem numa Interface como um **contrato**, uma **lista de regras** ou uma **especificação**.
- Uma interface define um **conjunto de métodos e propriedades** que uma classe deve implementar. Ela diz: 'Se você quiser ser do Tipo X, você DEVE ter

esses métodos e propriedades'. Mas ela **não fornece a implementação** desses métodos; apenas a sua 'assinatura' (nome, parâmetros, tipo de retorno).

Características Chave:

- Começam com a letra I por convenção (ex: IImprimivel).
- Contêm apenas **assinaturas** de métodos, propriedades, eventos, etc. (sem corpo, sem lógica).
- **Não podem ser instanciadas** (não podem criar new IImprimivel()).
- Uma classe pode **implementar múltiplas interfaces** (diferente da herança de classes, onde só se pode herdar de uma).
- Todos os membros de uma interface são implicitamente public e abstract (não precisam ser declarados como tal).

3.2 - Declarando e Implementando Interfaces

Vamos criar um exemplo: queremos que diferentes objetos possam ser 'imprimíveis'.

- **Exemplo de Código (Interface IImprimivel.cs):**

```
// Declaração da interface - com 'I' no início por convenção
public interface IImprimivel
{
    // Assinatura de um método (sem corpo!)
    void Imprimir();

    // Assinatura de uma propriedade (somente get)
    string ObterConteudoImpressao { get; }
}
```

Agora, vamos fazer algumas classes **implementarem** essa interface. Elas serão **obrigadas** a seguir o contrato!

- **Exemplo de Código (Implementação em Documento.cs e Foto.cs):**

```
// Documento.cs
   public class Documento : IImprimivel // Implementa a interface
  {
      public string Titulo { get; set; }
      public string Conteudo { get; set; }

      public Documento(string titulo, string conteudo)
      {
          Titulo = titulo;
          Conteudo = conteudo;
      }

      // Implementação do método da interface
      public void Imprimir()
      {
          Console.WriteLine($"\n--- Imprimindo Documento: {Titulo} ---");
          Console.WriteLine(Conteudo);
          Console.WriteLine("-----------------------------------");
      }

      // Implementação da propriedade da interface
      public string ObterConteudoImpressao
      {
          get { return $"Documento: {Titulo}\n{Conteudo}"; }
      }
  }
```

```
// Foto.cs
public class Foto : IImprimivel // Implementa a interface
{
    public string NomeArquivo { get; set; }
    public string Resolucao { get; set; }

    public Foto(string nomeArquivo, string resolucao)
    {
        NomeArquivo = nomeArquivo;
        Resolucao = resolucao;
    }

    // Implementação do método da interface
    public void Imprimir()
    {
        Console.WriteLine($"\n--- Imprimindo Foto: {NomeArquivo} ---");
        Console.WriteLine($"Resolução: {Resolucao}");
        Console.WriteLine("--------------------------------");
    }

    // Implementação da propriedade da interface
    public string ObterConteudoImpressao
    {
        get { return $"Foto: {NomeArquivo} ({Resolucao})"; }
    }
}
```

3.3 - Interfaces e Polimorfismo: O Poder dos Contratos

"Aqui está a mágica! Graças às interfaces, podemos usar o Polimorfismo para tratar objetos de tipos diferentes de forma uniforme, desde que eles implementem a mesma interface."

- **Exemplo de Código (Polimorfismo com Interface no Program.cs):**

```
// Program.cs

    class Program

    {

        static void Main(string[] args)

        {

            // Criando objetos de classes que implementam IImprimivel

            Documento relatorio = new Documento("Relatório de Vendas", "Vendas do mês de Julho...");

            Foto selfie = new Foto("minha_selfie.jpg", "1920x1080");

            // Podemos colocar objetos de tipos diferentes em uma lista da Interface!

            List<IImprimivel> filaDeImpressao = new List<IImprimivel>();

            filaDeImpressao.Add(relatorio);

            filaDeImpressao.Add(selfie);


            Console.WriteLine("\n--- Fila de Impressão (Polimorfismo!) ---");

            foreach (var item in filaDeImpressao)

            {

                // Chama o método Imprimir() de cada item polimorficamente

                item.Imprimir();

            }

            // Podemos verificar se um objeto implementa uma interface
```

```
            if (relatorio is IImprimivel)
            {
                Console.WriteLine("Relatório implementa IImprimivel.");
            }
        }
    }
```

3.4 - Vantagens das Interfaces

- **Contrato Explícito:** Garante que classes sigam um padrão de comportamento.
- **Acoplamento Fraco:** Seu código depende do contrato (interface), não da implementação concreta. Facilita a substituição de componentes.
- **Herança Múltipla de Comportamento:** Uma classe pode implementar várias interfaces (diferente da herança de classes).
- **Testabilidade:** Facilita a criação de 'mocks' (objetos de teste) para testes unitários.
- **Extensibilidade:** Você pode adicionar novas implementações de uma interface sem alterar o código que usa essa interface.

3.4 - Interfaces no Dia a Dia do C#

- **Injeção de Dependência:** Em projetos maiores, as interfaces são a base para a **Injeção de Dependência**, que é uma técnica super importante para construir sistemas modulares e testáveis. Vocês vão injetar a interface (IMensageiro), e não a implementação específica (WhatsAppMensageiro), para que seu código seja mais flexível e fácil de testar.

- **APIs e Frameworks .NET:** Muitas interfaces no próprio .NET (ex: IEnumerable, IDisposable, IComparable). Quando vocês usam foreach em uma lista, é porque ela implementa IEnumerable! Essa interface garante que elas podem ser 'enumeradas', ou seja, percorridas item por item.

4 Classes Abstratas: A Ponte entre Abstração e Implementação

4.1 - O Que é uma Classe Abstrata?

Objetivo: Compreender o que são Classes Abstratas, como declará-las, herdá-las e suas vantagens e casos de uso.

- "Se as Interfaces são contratos puros, as **Classes Abstratas** são um meio-termo entre uma interface e uma classe concreta (normal)."

- "Uma classe abstrata **não pode ser instanciada diretamente** (não pode criar new ClasseAbstrata()). Ela serve como uma **classe base** para outras classes."

- "Ela pode conter:
  - **Membros Abstratos:** Métodos ou propriedades sem implementação (como na interface), que **devem** ser implementados pelas classes derivadas.
  - **Membros Concretos:** Métodos ou propriedades com implementação (como em classes normais), que são herdados pelas classes derivadas.
  - **Campos e Construtores:** Pode ter campos (variáveis de instância) e construtores (que são chamados pelas classes derivadas via base())."

- "**Uso:** Ideal para modelar a relação 'é um tipo de' (herança), mas quando você quer fornecer uma **implementação base** para alguns comportamentos, enquanto deixa outros comportamentos para as classes filhas decidirem."

4.2 - Declarando Classes e Membros Abstratos

Vamos criar um exemplo de FormaGeometrica.

- "Observem a palavra-chave **abstract**. Ela aparece na declaração da classe e nos membros que não têm implementação."

Exemplo de Código (Classe Abstrata FormaGeometrica.cs):

```
// Classe abstrata: não pode ser instanciada diretamente
    public abstract class FormaGeometrica
```

```
    {
        public string Cor { get; set; }

        // Construtor (será chamado pelas classes derivadas)
        public FormaGeometrica(string cor)
        {
            Cor = cor;
            Console.WriteLine($"\nForma Geométrica de cor {Cor} criada.");
        }

        // Método abstrato: deve ser implementado pelas classes derivadas
        // Não tem corpo, apenas a assinatura.
        public abstract double CalcularArea();

        // Método concreto (com implementação)
        // Este método é herdado e pode ser usado diretamente pelas classes filhas.
        public void ExibirCor()
        {
            Console.WriteLine($"A cor desta forma é: {Cor}");
        }
    }
```

4.3. Implementando Classes Abstratas (Herança)

Agora, vamos criar classes que **herdam** de FormaGeometrica. Elas serão obrigadas a implementar o método abstrato CalcularArea().

Exemplo de Código (Classes Derivadas Circulo.cs e Retangulo.cs):

```
// Circulo.cs

    public class Circulo : FormaGeometrica // Herda da classe abstrata
    {
        public double Raio { get; set; }
```

```
        public Circulo(string cor, double raio) : base(cor) // Chama construtor
da base
        {
            Raio = raio;
        }

        // Implementação OBRIGATÓRIA do método abstrato (override é necessário)
        public override double CalcularArea()
        {
            return Math.PI * Raio * Raio;
        }
    }

    // Retangulo.cs
    public class Retangulo : FormaGeometrica // Herda da classe abstrata
    {
        public double Largura { get; set; }
        public double Altura { get; set; }

        public Retangulo(string cor, double largura, double altura) : base(cor)
        {
            Largura = largura;
            Altura = altura;
        }

        // Implementação OBRIGATÓRIA do método abstrato
        public override double CalcularArea()
        {
            return Largura * Altura;
        }
    }
```

Exemplo de Código (Uso no Program.cs):

```
    class Program
    {
        static void Main(string[] args)
        {
            // NÃO PODE: FormaGeometrica formaGenerica = new
FormaGeometrica("verde"); // ERRO!
            // Uma classe abstrata não pode ser instanciada diretamente.

            // Podemos instanciar as classes CONCRETAS (que não são abstratas)
            Circulo circuloVermelho = new Circulo("Vermelho", 5.0);
            Retangulo retanguloAzul = new Retangulo("Azul", 10.0, 4.0);

            // Polimorfismo: Podemos criar uma lista do tipo da classe
abstrata!
```

```
            List<FormaGeometrica> minhasFormas = new List<FormaGeometrica>();
            minhasFormas.Add(circuloVermelho);
            minhasFormas.Add(retanguloAzul);

            Console.WriteLine("\n--- Calculando Áreas (Polimorfismo com
Abstração!) ---");
            foreach (var forma in minhasFormas)
            {
                forma.ExibirCor(); // Método concreto herdado de
FormaGeometrica
                Console.WriteLine($"Área calculada:
{forma.CalcularArea():F2}"); // Método abstrato sobrescrito (polimorfismo)
            }
        }
    }
```

4.4 – Vantagens das Classes Abstratas

A classe abstrata é ótima quando vocês têm uma hierarquia de herança forte (relação 'é um tipo de'), e vocês querem que as classes filhas compartilhem alguns comportamentos já implementados, mas também sejam obrigadas a implementar outros comportamentos específicos.

- **Combinação de Abstração e Implementação:** Permite definir um "esqueleto" com partes implementadas e partes que devem ser implementadas.

- **Reutilização de Código:** Oferece implementação padrão para métodos comuns.

- **Hierarquia de Tipos:** Reforça a relação 'é um tipo de' e permite polimorfismo.

- **Força a Implementação:** Garante que subclasses implementem certos comportamentos críticos.

4.5 – Interfaces vs. Classes Abstratas: Quando Usar Qual?

Essa é uma pergunta muito comum e importante! Interfaces e Classes Abstratas parecem semelhantes, pois ambas lidam com abstração e polimorfismo. Mas elas têm propósitos e regras diferentes.

Regra Geral (Diretriz):

Use Interfaces quando:

- Você quer definir um **contrato de comportamento** que classes diferentes (que podem ou não estar na mesma hierarquia de herança) devem seguir. (Ex: IImprimivel, ICadastravel).
- Precisa de **"herança múltipla" de comportamento** (uma classe pode "ser" muitas coisas diferentes).
- Você quer ter um **acoplamento muito baixo** entre o código que usa a abstração e a implementação real.

Use Classes Abstratas quando:

- Você tem uma forte relação **"é um tipo de"** (herança) e quer fornecer uma **implementação base** para alguns comportamentos comuns, deixando outros para as classes filhas.
- Quer compartilhar **campos (estado)** e **construtores** entre as classes derivadas.
- Você tem métodos que são comuns, mas outros que *precisam* ser especializados por cada filho.

Analogia Final:

- **Interface:** A Constituição Federal. Ela dita as regras e direitos básicos que *qualquer cidadão* (classe) deve seguir ou ter. Não diz *como* você faz isso, só que você precisa fazer.

- **Classe Abstrata:** O Regulamento Interno de uma Empresa. Ele já tem algumas regras gerais implementadas (horário de trabalho), mas deixa outras para cada departamento (filho) definir (como organizar as pastas).

5 Exercício prático final

Desafio Final: Sistema de Notificações.

Instruções para o Exercício:

1. Crie um novo projeto de console chamado SistemaNotificacoes.
2. Crie uma **Interface INotificavel.cs**:
   - void EnviarNotificacao(string mensagem)
3. Crie uma **Classe Abstrata NotificadorBase.cs**:
   - public string Remetente { get; set; }
   - protected NotificadorBase(string remetente)
   - public void LogarNotificacao(string mensagem) (método concreto)
   - public abstract void ConfigurarCredenciais() (método abstrato)
4. Crie uma **Classe Concreta EmailNotificador.cs** que herda de NotificadorBase e implementa INotificavel:
   - public string EmailDestino { get; set; }
   - Construtor que recebe remetente, emailDestino.
   - override void ConfigurarCredenciais(): Simplesmente exibe "Configurando credenciais de e-mail..."
   - void EnviarNotificacao(string mensagem): Exibe "Email de [Remetente] para [EmailDestino]: [mensagem]" e chama LogarNotificacao.
5. Crie uma **Classe Concreta SmsNotificador.cs** que herda de NotificadorBase e implementa INotificavel:
   - public string NumeroTelefone { get; set; }
   - Construtor que recebe remetente, numeroTelefone.
   - override void ConfigurarCredenciais(): Exibe "Configurando credenciais de SMS..."
   - void EnviarNotificacao(string mensagem): Exibe "SMS de [Remetente] para [NumeroTelefone]: [mensagem]" e chama LogarNotificacao.
6. No Program.cs (Main):
   - Crie uma List<INotificavel>.
   - Adicione um EmailNotificador e um SmsNotificador à lista.
   - Para cada notificador na lista, chame ConfigurarCredenciais() e EnviarNotificacao("Mensagem de teste!").

6 Dúvidas e próximos passos

6.1 - O que vimos!

Hoje, destrinchamos a **Abstração**, um conceito que nos permite projetar softwares mais flexíveis e poderosos através de **Interfaces** (os contratos) e **Classes Abstratas** (os esqueletos com implementações parciais).

Lembrem-se: **Encapsulamento**, **Herança**, **Polimorfismo** e **Abstração** são as ferramentas que transformam vocês de 'codificadores' em 'arquitetos de software'. Eles permitem que vocês criem sistemas mais robustos, modulares, e fáceis de escalar e manter.