**TÓPICOS IMPORTANTES EM C#**

* ***Enumeração(Enum):***
  + **O que é:** é um tipo de dado especial que permite definir um conjunto de constantes nomeadas, facilitando o trabalho com valores que têm significado simbólico em vez de usar números ou strings "soltos" no código. A enum é usada para representar estados, opções ou categorias de maneira mais clara e legível.
  + **Quando usar:** As enumerações são úteis quando você tem um conjunto fixo de opções ou estados que representam algo concreto. O *enum* em C# é projetado para retornar a representação textual (nome do membro) quando usado em um contexto como Console.WriteLine ou ToString(). Se você quiser o número associado ao membro, você pode fazer um cast explícito:
  + **Vantagens:**
  1. **Legibilidade**: Em vez de usar valores numéricos ou constantes que podem não ser claros, você usa nomes simbólicos que têm um significado semântico mais forte.
  2. **Manutenção**: As enums centralizam as opções, facilitando a modificação e manutenção do código.
  3. **Segurança de tipo:** Como uma enum é um tipo de dado, você tem a garantia de que só poderá atribuir valores válidos a variáveis do tipo enum.
  + **Resumo:** Enums são uma excelente maneira de lidar com grupos de valores fixos, tornando o código mais claro, seguro e fácil de manter. Você deve usá-los sempre que tiver um conjunto definido de valores que representam algo significativo no seu sistema, como estados, opções ou categorias.
  + **Exemplo:**

enum StatusPedido

{

Pendente,

Processando,

Enviado,

Entregue,

Cancelado

}

class Pedido

{

public StatusPedido Status { get; set; }

public void AvancarStatus()

{

if (Status < StatusPedido.Cancelado)

{

Status++;

}

}

public void ExibirStatus()

{

Console.WriteLine($"O status atual do pedido é: {Status}");

}

}

class Program

{

static void Main(string[] args)

{

Pedido pedido = new Pedido();

pedido.Status = StatusPedido.Pendente;

pedido.ExibirStatus();

pedido.AvancarStatus();

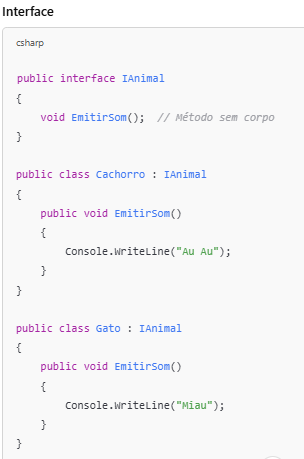
pedido.ExibirStatus();

}

}

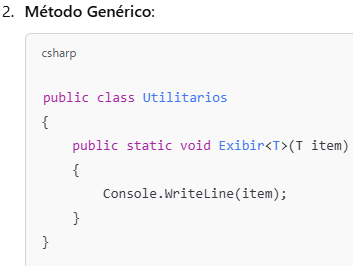
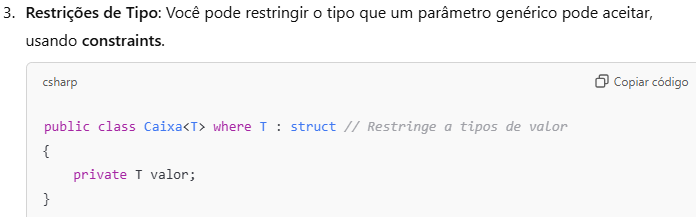
* ***Classe abstrata e interface:***
  + **Conceito:**
    - **Interface**: Serve para definir um contrato de comportamento que diversas classes podem implementar, independentemente de onde elas estejam na hierarquia de classes. Um contrato, se você usa o interface você tem que implementar o método.
    - **Classe Abstrata**: Serve como uma base para outras classes que compartilham uma implementação comum. Reaproveitamento de método e atributos.
  + **Quando usar:** 
    - **Interface:** Ideal quando você quer definir um comportamento comum que classes completamente diferentes precisam compartilhar, sem uma implementação compartilhada. Comportamentos comum.
    - **Classe Abstrata:** Ideal quando várias classes relacionadas compartilham um comportamento ou estado comum e você quer definir parte da lógica base. Base comum com lógica.
  + **Vantagens:** 
    - **Interface:** 
      * Flexibilidade e Polimorfismo: permitem que classes totalmente diferentes implementem o mesmo conjunto de métodos,
      * Independência de Herança: são ideais quando você quer definir um comportamento comum para classes que não têm uma relação hierárquica (herança) direta.
      * Implementação Múltipla: Uma classe pode implementar várias interfaces,
      * Orientação a Contrato: atuam como contratos puros: qualquer classe que implementa uma interface garante que fornecerá uma implementação para seus métodos, o que aumenta a previsibilidade do comportamento do código.
      * Ideal para APIs e Testes: são ideais para definir APIs, pois permitem que diferentes implementações sigam a mesma estrutura;
    - **Classe Abstrata:**
      * Compartilhamento de Implementação: permitem definir métodos concretos que podem ser reutilizados pelas classes derivadas, evitando a repetição de código
      * Uso de Campos e Construtores: podem ter campos e construtores, permitindo a definição de um estado compartilhado entre as classes derivadas.
      * Facilidade de Expansão com Métodos Concretos: você pode adicionar novos métodos com implementação padrão sem quebrar o contrato com as classes derivadas.
      * Base para Relacionamento Hierárquico: são úteis para estruturar uma hierarquia de classes com uma relação “é um”, onde várias subclasses se relacionam por meio de características comuns.
      * Ideal para Modelos Mais Estruturados: Quando várias classes compartilham lógica e características próximas, uma classe abstrata é mais adequada.
  + **Resumo:**
    - **Interface**: Define um contrato de métodos e propriedades sem implementação, focada em comportamento comum para classes não relacionadas. Permite múltiplas implementações, favorecendo flexibilidade e polimorfismo.
    - **Classe Abstrata:** Define uma base para classes relacionadas, podendo ter métodos implementados e campos. Útil para compartilhar lógica e estado entre subclasses em uma hierarquia organizada.
  + **Exemplo:**



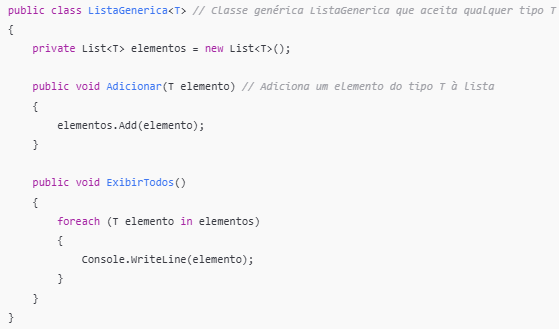


* ***Generics:***
  + **Conceito:** C# são uma maneira de criar classes, métodos e estruturas que funcionam com qualquer tipo de dado, sem precisar especificar o tipo exato no momento da implementação. Eles permitem que você crie um código mais flexível e reutilizável, lidando com diferentes tipos de dados de forma segura em tempo de compilação.
  + **Quando usar:**
    - Quando queremos evitar duplicação de código que trabalha com diferentes tipos de dados.
    - Deseja melhorar a performance evitando conversões de tipo (casting) e boxing/unboxing.
    - Quer garantir segurança de tipo (type safety) sem sacrificar a flexibilidade do código.
  + **Como usar:** Para definir uma classe, método, ou estrutura genérica, usa-se parâmetros de tipo. Por convenção, T é usado como nome do parâmetro de tipo genérico.

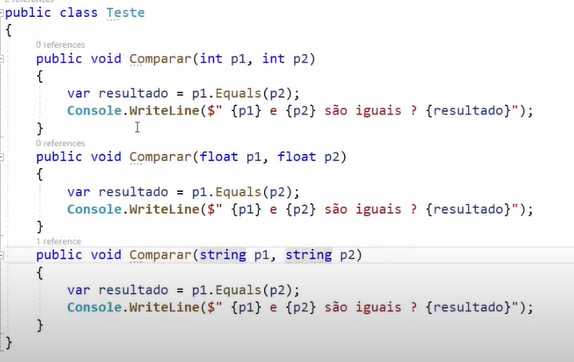


* + **Vantagens:**
    - Segurança de Tipo: Permite detecção de erros em tempo de compilação, evitando erros de tipo.
    - Reutilização de Código: Evita duplicação de código, uma vez que a mesma classe ou método pode ser usado com diferentes tipos.
    - Performance: Evita a necessidade de casting ou boxing/unboxing, que podem impactar a performance.
  + **Resumo:** Generics permitem criar código flexível e seguro para qualquer tipo, mantendo a segurança de tipo e a performance. Ao usá-los, é possível reduzir a duplicação e tornar o código mais limpo e reutilizável.
  + **Desvantagens:** Usar T pode ser desnecessário se você sabe que a lista precisa manipular apenas um tipo específico de dado. Nesse caso, definir a lista diretamente com o tipo específico pode tornar o código mais simples e fácil de entender para um leitor futuro.
  + **Exemplo:**
    - Fazer a lista genérica permite que você crie listas para qualquer tipo (int, string, double, etc.) sem precisar escrever classes separadas para cada tipo. Assim, a mesma classe ListaGenerica<T> pode ser usada com qualquer tipo de dado, economizando tempo e reduzindo a duplicação de código.



* + **Outro exemplo:** 
    - Muita repetição de código:



* + - Uso eficiente do Generics

