**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();

}

}

* ***Composição:*** 
  + **Conceito:** é um conceito relacionado ao design orientado a objetos, onde uma classe pode ser composta de outras classes. Basicamente, em vez de herdar comportamentos e atributos de uma classe base (como na herança), uma classe “composta” inclui instâncias de outras classes como parte de sua definição. A composição permite que você construa uma relação "tem-um" entre as classes, onde uma classe contém outra como um atributo.
  + **O que é:** A composição é quando uma classe contém uma ou mais instâncias de outras classes como membros. Esse relacionamento implica que uma classe é formada por outras, ou seja, ela "tem" objetos de outras classes. O uso de composição é uma prática que promove modularidade e reutilização de código, pois permite que as classes sejam combinadas para criar funcionalidades mais complexas.
  + **Quando usar:** composição é ideal quando uma classe "tem" outra como parte dela. Use composição:

1. Quando a relação "tem-um" faz sentido. Exemplo: um Carro tem um Motor, um Funcionario tem um Endereco.
2. Quando você quer evitar herança excessiva. A herança pode criar estruturas complexas que são difíceis de manter, enquanto a composição permite maior flexibilidade.
3. Quando você deseja criar objetos complexos a partir de objetos menores e reutilizáveis.
   * **Vantagens:**
4. **Acoplamento mais baixo**: A composição cria um acoplamento mais solto entre as classes do que a herança. Isso permite que as classes sejam alteradas sem impactar diretamente outras partes do sistema.
5. **Reusabilidade**: Diferentes classes podem compartilhar os mesmos componentes (ou instâncias de outras classes), promovendo reuso de código.
6. **Manutenção**: Facilita a manutenção, uma vez que a lógica relacionada a cada componente está isolada em classes específicas.
   * **Desvantagens:**
7. **Complexidade no gerenciamento de dependências:** Como as classes compostas podem depender de várias outras classes, isso pode aumentar a complexidade ao gerenciar dependências, especialmente quando há muitos níveis de composição.
8. **Mais código a ser gerenciado:** Comparado à herança, pode parecer que há mais código para se gerenciar, uma vez que há instâncias de outras classes envolvidas e muitas vezes construtores maiores.
   * **Diferença entre Composição e Herança:**
9. **Composição**: Relacionamento "tem-um". Exemplo: um carro tem um motor, uma empresa tem funcionários.
10. **Herança:** Relacionamento "é-um". Exemplo: um Cachorro é um Animal, um Carro é um Veículo.
    * **Resumo:** Use composição sempre que quiser construir classes a partir de outras classes, onde a relação "tem-um" seja mais adequada do que "é-um". Composição é uma abordagem flexível, utilizada em cenários onde queremos montar objetos complexos a partir de objetos menores e mais simples, promovendo modularidade e reuso de código.
    * **Exemplo:**



* ***Herança:***
  + **Conceito:** **Herança** é um princípio fundamental de programação orientada a objetos que permite que uma classe derive características (atributos e métodos) de outra. A classe "filha" ou "derivada" herda os membros da classe "pai" ou "base", tornando possível o compartilhamento de código entre classes relacionadas.
  + **O que é:** Em C#, a herança é implementada criando uma relação de "é um" (ou seja, uma relação hierárquica) entre classes. Isso significa que uma classe derivada é uma especialização de uma classe base.

Por exemplo, se temos uma classe “Conta”, podemos criar uma classe derivada “ContaPoupança" que herda todas as características e comportamentos de “Conta”, mas possibilitando adicionarmos novos atributos e métodos específicos da classe “ContaPoupança”.

* + **Quando usar:** A herança é mais útil quando há uma clara relação hierárquica entre duas classes, e a classe derivada precisa de acesso aos métodos e propriedades da classe base. Geralmente, você deve considerar usar herança se:

1. As classes compartilham atributos e comportamentos (ex.: Carro e Moto são tipos de Veiculo).
2. Deseja promover reutilização de código, evitando duplicação.
3. A relação "é um" é verdadeira (ex.: um Cachorro é um tipo de Animal).

É importante notar que o uso excessivo da herança pode tornar o código mais complexo e difícil de manter, então é preferível aplicá-la apenas quando a relação entre as classes é clara.

* + **Definições:**
    - Relação “é um”;
    - Generalização/especialização;
    - Superclasse (classe base) / Subclasse (classe derivada);
    - Herança/extensão;
    - Herança é uma associação entre classes (e não entre objetos)
    - **Palavra-chave “base”**: Permite que a classe derivada acesse membros da classe base, útil para chamadas ao construtor da classe base ou acesso a métodos.
  + **Vantagens:**
    - **Reutilização de código:** A herança permite que classes derivadas aproveitem e estendam o código da classe base, evitando duplicação e facilitando a manutenção.
    - **Organização e estrutura:** Com herança, é possível criar uma hierarquia lógica de classes, tornando o código mais compreensível e organizado.
    - **Facilidade de extensão:** A herança facilita a criação de classes especializadas que adicionam funcionalidades específicas a partir de uma classe base.
  + **Resumo:** é uma técnica de reutilização de código que permite criar hierarquias entre classes. Uma classe derivada herda os métodos e atributos de uma classe base, promovendo a reutilização e a organização do código. Ela é útil quando há uma relação hierárquica clara e quando a especialização da classe base em subclasses faz sentido. É recomendada para simplificar o desenvolvimento e a manutenção de classes relacionadas.
* ***Polimorfismo:***
  + **Conceito:** é um princípio da programação orientada a objetos que permite que métodos de classes diferentes, mas relacionadas, respondam de forma diferente à mesma chamada. Em outras palavras, polimorfismo permite o uso de uma única interface para representar múltiplos comportamentos de objetos.
  + **O que é:** Polimorfismo pode ser obtido principalmente através de duas técnicas:

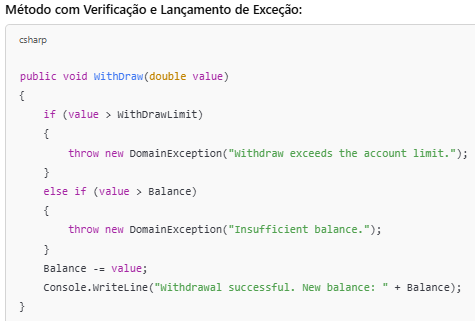
1. **Polimorfismo de Sobrecarga (ou Estático):** Quando um método é sobrecarregado na mesma classe, ou seja, existem várias versões do mesmo método, cada uma com uma lista de parâmetros diferente.
2. **Polimorfismo de Substituição (ou Dinâmico):** É quando métodos em classes derivadas substituem (ou “sobrescrevem”) métodos da classe base. Isso é obtido com a palavra-chave virtual no método da classe base e override no método da classe derivada.
   * **Quando usar:** O polimorfismo é útil quando se quer tratar objetos de várias classes derivadas da mesma maneira, mas mantendo comportamentos específicos de cada um. Pode ser usado em situações como:
3. **Interfaces e classes abstratas:** Quando você deseja que classes derivadas tenham implementações específicas, mas quer tratá-las de forma uniforme.
4. Métodos que precisam ter comportamento personalizado em classes derivadas Como métodos de cálculo, exibição, entre outros.
5. **Sobrecarga de métodos:** Quando métodos precisam aceitar diferentes tipos ou números de parâmetros na mesma classe.
   * **Definições:**
     + **Polimorfismo de Sobrecarga (Estático):** Implementado com métodos com o mesmo nome e diferentes parâmetros dentro da mesma classe.
     + **Polimorfismo de Substituição (Dinâmico):** Realizado através de métodos virtual e override, permitindo que classes derivadas implementem sua própria versão de um método da classe base.
     + **Interface:** Em C#, interfaces ajudam a definir um conjunto de métodos que classes devem implementar, promovendo o polimorfismo.
     + **Classe Abstrata:** Uma classe que não pode ser instanciada diretamente e contém métodos abstract que devem serimplementados em classes derivadas, ajudando a garantir que todas as subclasses compartilhem métodos específicos.
   * **Vantagens:**
     + **Flexibilidade e extensibilidade:** Facilita a criação de código mais flexível, permitindo adicionar novos comportamentos a classes derivadas sem alterar a estrutura da classe base.
     + **Uniformidade no tratamento de objetos:** É possível usar referências de classe base para manipular objetos de classes derivadas, simplificando o código e melhorando a legibilidade.
     + **Redução de código duplicado:** Com o polimorfismo, podemos definir um comportamento geral na classe base e apenas ajustar o necessário em classes derivadas, reduzindo a duplicação de código.
   * **Resumo:** O polimorfismo em C# permite que classes derivadas redefinam o comportamento de métodos da classe base, promovendo reutilização, flexibilidade e tratamento uniforme de objetos. Ele pode ser implementado de forma estática (sobrecarga de métodos) ou dinâmica (substituição de métodos), e é especialmente útil ao trabalhar com hierarquias de classes, interfaces e classes abstratas.

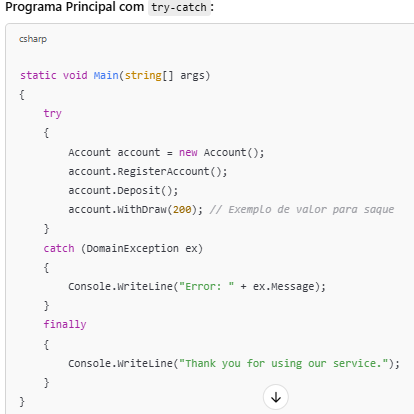
* ***Tratamento de exceções:*** 
  + **Conceito:** Tratamento de exceções é o processo de capturar e lidar com erros em tempo de execução para evitar que o programa encerre inesperadamente. Isso melhora a robustez e a confiabilidade da aplicação.
  + **O que é:** 
    - **2 forma principais de tratar exceções no C#:**
    - **Tratamento com Exceções Padrão usando try, catch, finally:** A forma mais comum de tratamento de exceções em C# é usando exceções padrão do .NET, como ArgumentNullException, etc. Elas são úteis para lidar com erros gerais no fluxo de execução. Para isso, usamos o recurso try, catch e finally para montar a verificação da exceção.
    - **Tratamento com Exceções Personalizadas usando throw e if:** As exceções personalizadas são úteis para erros específicos do domínio da aplicação. Nesses casos, criamos uma nova classe de exceção que herda de Exception ou uma de suas subclasses (ApplicationException, etc.), como sua DomainException. Com a classe criada vamos no método ou classe que queremos fazer a exceção e definimos a regra usando um if e lançando a exceção.
  + **Quando usar:** 
    - **Exceções padrões:** Use exceções padrão para erros genéricos ou erros comuns já contemplados pelo .NET, como o acesso a índices fora do array (IndexOutOfRangeException) ou leitura de arquivos inexistentes (FileNotFoundException).
    - **Exceções personalizadas:** Use exceções personalizadas para tratar erros específicos da lógica do programa, como "SaldoInsuficienteException" em um sistema bancário, ou "PedidoInvalidoException" em um sistema de pedidos.
  + **Definições:** 
    - **Sempre tentar usar o tipo específico de exceção no catch.**
    - **Try:** Delimita o bloco de código que pode causar uma exceção. Caso uma exceção ocorra, o fluxo de execução é transferido para o bloco catch correspondente
    - **Catch:** Captura e trata exceções lançadas no bloco try. Cada catch pode capturar tipos específicos de exceção, permitindo tratamento personalizado para cada tipo de erro.
    - **Finally:** Executado após o bloco try e catch, independentemente de uma exceção ter ocorrido ou não. Geralmente usado para liberar recursos, como fechar arquivos ou conexões de banco de dados.
    - **Exceções específicas no catch:** Sempre que possível, capture tipos específicos de exceções para diferenciar o tratamento de cada tipo de erro, melhorando a precisão do tratamento de exceções.
    - **Exceções Personalizadas:** Para tratar erros específicos do programa, crie uma classe de exceção personalizada herdando de ApplicationException ou outra subclasse de Exception.
  + **Vantagens:** 
    - **Melhor controle de erros:** O código pode prever, capturar e tratar erros em vez de encerrar inesperadamente.
    - **Organização:** Permite separar a lógica do programa do tratamento de erros, facilitando a manutenção e a leitura do código.
    - **Exceções personalizadas:** Ajudam a fornecer informações de erro mais específicas e detalhadas para o usuário, relacionadas diretamente ao contexto do programa.
  + **Como criar uma exceção padrão:** 
    - **Definição de Exceções Padrão:**
      * As exceções padrão são classes de erro que já vêm incluídas na biblioteca do .NET, como ArgumentNullException, InvalidOperationException, FormatException, entre outras.
      * Elas são usadas para representar erros comuns que podem ocorrer durante a execução de um programa e são úteis para fornecer informações sobre o que deu errado em um contexto específico.
    - **Tratamento da Exceção no Método:**
      * Ao escrever um método que pode falhar devido a condições esperadas, você deve estar preparado para capturar e lidar com essas exceções.
      * Utilize um bloco try para tentar executar a lógica que pode gerar uma exceção. Se uma exceção padrão ocorrer, use um bloco catch correspondente para capturar e lidar com essa exceção.
      * Por exemplo, ao tentar converter uma entrada do usuário para um número, você pode usar int.Parse() e, caso a entrada não seja um número válido, isso lançará uma FormatException.
    - **Tratamento da Exceção no Programa Principal:**
      * No ponto onde o método é chamado (geralmente no Main), utilize um bloco try-catch para tratar exceções que podem ser lançadas pelas operações dentro do método.
      * Se um erro ocorrer (como uma FormatException ao tentar converter uma string em número), o bloco catch captura a exceção e permite que você forneça uma mensagem ou tratamento adequado ao usuário.
    - **Exemplo:**



* + **Como criar uma exceção personalizada:** 
    - **Definição de Exceção Personalizada:**
      * Defina uma classe de exceção personalizada, como DomainException, que herda de ApplicationException ou diretamente de Exception.
      * Isso é útil para criar exceções específicas para o domínio da aplicação, com mensagens relevantes para o contexto do erro.
    - **Lançamento da Exceção no Método:**
      * No método em que deseja fazer a verificação, utilize uma estrutura condicional (if) para verificar cada situação específica. Se uma condição indicar um erro, use throw new DomainException("Mensagem") para lançar a exceção.
      * A mensagem passada para DomainException deve ser clara e informar qual condição não foi atendida ou qual erro ocorreu.
    - **Tratamento da Exceção no Programa Principal:**
      * No ponto onde o método é chamado (geralmente no Main ou em algum método que gerencie a execução do programa), utilize um bloco try-catch.
      * O bloco try tenta executar o código que pode lançar exceções. Se uma exceção de DomainException for lançada, o bloco catch correspondente a essa exceção a captura.
    - **Exemplo:**







* + **Diferença entre try e throw:** 
    - **Try:** O try-catch é completamente funcional por si só. Ele é usado para capturar e tratar exceções, independentemente de onde essas exceções surgem. Elas podem ocorrer naturalmente durante a execução (como uma divisão por zero ou um acesso a um índice inválido de uma lista) sem necessidade de um throw.
    - **Throw:** throw é usado para lançar ou relançar exceções, ele precisa de um try-catch em algum lugar no código para capturar e tratar a exceção que ele lança. Sem um try-catch para interceptá-la, o throw causará uma interrupção no programa, encerrando-o e exibindo o erro diretamente na saída de erro. O throw pode ser usado para alterar ou personalizar o comportamento quando uma exceção de sistema é lançada. O throw é obrigatório para exceções personalizadas e opcional para exceções de sistema.
  + **OBS/Importante:** Exceções personalizadas são um recurso importante em cenários que demandam maior clareza em erros específicos ou quando estamos lidando com regras de negócio complexas, mas é verdade que, em muitos sistemas, as exceções padrões do .NET são suficientes e podem simplificar o código.
  + **Resumo:** Tratamento de exceções em C# é o processo de capturar e gerenciar erros durante a execução, utilizando blocos try, catch e finally para exceções padrão e o comando throw para lançar exceções, seja de sistema ou personalizadas, garantindo que o programa lide adequadamente com situações de erro e mantenha sua estabilidade.