MINICURSO

Métodos

- Os métodos têm capacidade de receber parâmetros, processar alguma tarefa e retornar algum valor.

- Na programação não é uma boa prática repetirmos código, ao contrário, devemos reutilizar o máximo possível, para isso servem os métodos, que são blocos de código que podem ser chamados sempre que necessário.

- Um método deve ter um **tipo de retorno**, um **nome**, os **parâmetros** (se houver), o **corpo** (onde é processado o código) e o **retorno** (quando houver).

Métodos

```
tipoRetorno nomeMétodo() {
        corpoDoMétodo;
        retornoValor;
}
```

```
//Declaração do Método com tipo de retorno, nome e parâmetros
double Somar(double valor1, double valor2)
{
    //corpo do método
    double soma = valor1 + valor2;

    //Retorno do método
    return soma;
}
```

Métodos

```
modificadorAcesso tipoRetorno nomeMétodo() {
        corpoDoMétodo;
        retornoValor;
}
```

Modificadores

Modificadores de Acesso

- **public**: Acesso não é restrito.
- **private**: O acesso é limitado para o tipo de recipiente.

- **protected** : O acesso é limitado à classe ou tipos derivados da classe. (classes descendentes ou a própriar)
- **internal** : O acesso é limitado ao assembly atual. *(por projeto)*

```
//Declaração do Método com tipo de retorno, nome e parâmetros
public double Somar(double valor1, double valor2)
{
    //corpo do método
    double soma = valor1 + valor2;

    //Retorno do método
    return soma;
}
```

Modificadores

- **static -** é utilizado para para declarar um membro estático, que pertence a si mesmo, em vez de pertencer um objeto específico.

- **const -** é utilizado para variáveis que não podem ter o seu valor alterado.
  - *Utilizado para a criação de constantes.*

Tipos Enumerados

```
enum TipoDiretor {
     Marketing,
     RH,
     Comercial,
     Financeiro
}

// Instanciando
TipoDiretor tpDiretor = TipoDiretor.Comercial;

// Imprime 'Comercial'
Console.WriteLine(tpDiretor);
```

Exemplo de código!

Coleções ou Collections

- O que é uma coleção e qual a diferença entre uma coleção e um Array ?

- Uma coleção também armazena um conjunto de valores da mesma forma que um **array** a diferença é que uma coleção armazena os elementos como **Object**. Portanto as coleções tem capacidade de colecionar itens do tipo **Object**.

- As classes de coleções podem ser encontradas no namespace **System.Collections** e elas dão suporte a *pilhas, filas, listas, hash table, entre outras. (stacks, queues, lists, dictionary, ...)*

Coleções ou Collections

- **System.Collections.ArrayList**
- **System.Collections.Stack**
- **System.Collections.Queue**

- **System.Collections.Generic.List**
- **System.Collections.Generic.Dictionary**

- Obs.: Existem outros tipos de **Collections**, mas neste curso vamos abordar apenas os principais.

Vamos conhecer cada Collection em detalhes

ArrayList

- A classe ArrayList é um array dinâmico de objetos **heterogêneos**.

- O tamanho é gerenciado de forma dinâmica automaticamente.

- Em um um **array** podemos armazenar apenas objetos do mesmo tipo, já em um **ArrayList** podemos ter diferentes tipos de objetos, que são armazenados como tipos **Object**. Assim podemos ter um ArrayList que armazena *float, integer, string, etc.,* todos armazenados como **Object**.

Exemplo de código!

Queue

- Você pode comparar a classe **Queue** com as pessoas que estão esperando em uma fila.

- É **array** dinâmico de objetos **homogêneos** ou **heterogêneos**.

- A primeira pessoa que chega é a primeira a ser atendida, toda pessoa que chega é atendida na ordem de chegada.

- Este mecanismo é chamado **FIFO** : *first in first out* (primeiro a chegar primeiro a sair).

Exemplo de código!

Stack

- A classe **Stack** funciona basicamente como a classe Queue com uma diferença : O último objeto incluído na pilha é retornado primeiro.

- É **array** dinâmico de objetos **homogêneos** ou **heterogêneos**.

- Imagine uma pilha de papéis, o primeiro que você pega da pilha foi o último que foi colocado nela.

- Este mecanismo é conhecido como **LIFO** : **last in first out** (o último a entrar é o primeiro sair).

Exemplo de código!

List

- List: éa classe genérica que corresponde a ArrayList. É **array** dinâmico de objetos **homogêneos** ou **heterogêneos**.
  - **ArrayList** é **heterogênea.**
  - **List** é **homogênea.**

- A classe **List** da biblioteca genérica permite a criação de uma coleção para armazenamento de um conjunto de dados quaisquer.

- Permite informar o tipo de dados que queremos armazenar na coleção.

Exemplo de código!

Dictionary

- Corresponde a **HashTable**.
- É uma estrutura dinâmica e **heterogênea**.
- É uma lista indexada que nos permite armazenar pares de chave (**key**) + valor (**value**), sendo que estes podem ser de qualquer tipo.
- O melhor é que esta lista é indexada pela chave que você define, se tornando muito fácil e rápido localizar itens dentro dela.

Exemplo de código!

Biblioteca de coleções

- Namespace System.Collections
  - ArrayList
  - Queue
  - Stack
  - …
- Namespace System.Collections.Generic
  - List<T>
  - Dictionary<K, V>
  - …

Nullable Types

- Recurso da versão 2.0.
- Variáveis continuam a representar todos os valores do seu tipo, e mais um valor adicional *null*.
- Permite uma integração melhor com bancos de dados, que podem conter valores *null* em seus campos.
- Declaradas através da classe *Nullable*, ou através do operador *?* adicionado ao tipo à ser utilizado.

Nullable Types

- Podem ser declaradas de duas formas:
    - System.Nullable<T> variavel;
    - <T>? variavel;
- Onde T é o tipo da variável
- Ambas notações funcionam de maneira equivalente

Nullable Types

- Qualquer tipo por Valor pode ser usado como Nullable Type

```
int? i = 10;
double? x = 3.14;
bool? flag = null;
char? letra = 'a';
int?[] MeuArray = new int?[10];
```

- Os exemplos seguintes não são tipos por valor, portanto não são aceitáveis como Nullable Types

```
string? variavel = "tipos anulaveis";
object? umCliente = new object?();
```

Nullable Types

- Uma instância de um tipo anulável possui duas propriedades *read-only*
    - HasValue: do tipo *bool*, retorna verdadeiro quando uma variável contém um valor não-nulo.
    - Value: do tipo equivalente ao tipo anulável utilizado, se HasValue é true, então Value contém um valor significativo, senão, ao tentar acessar Value será gerada uma *exception.*

Nullable Types

```
int? clienteId = null;


if (clienteId.HasValue)
{
    Console.WriteLine(clienteId.Value);
    Console.WriteLine(clienteId);
}
else
    Console.WriteLine("Identificação Indefinida!");
```

```
// Exemplos de Conversões Explícitas
int? clienteId = null;

// Não compila
int x = clienteId;

// Compila, mas será gerada uma exception se x for null
int y = (int) clienteId;

// Compila, mas será gerada uma exception se x for null
int z = clienteId.Value;



//Conversao Implícita
int i = 12;   // Int, nao pode armazenar valores NULOS
int? j = 22;  // NULLABLE INT, pode armazenar valores INT
```

Nullable Types

- Operadores
  - Quaisquer operadores existentes para tipos por valor podem ser utilizados com Nullable Types.
  - O operador produzirá *null* se os operandos forem nulos, caso contrário, usará o valor contido para calcular o resultado.

```
int? x = 10;
x++;     // x agora é 11
x = x * 10; // x agora é 110

int? y = null;
x = x + y;  // x agora é nulo
x++; // x continua nulo
```

Nullable Types

- Operadores
  - Comparações: com dois *nullable types*, se o valor de um deles é *null*, e do outro não, o resultado sempre será falso.

```
int? num1 = 10;
int? num2 = null;

if (num1 >= num2)
    Console.WriteLine("num1 is greater than or equal to num2");
else
    // Este bloco é selecionado, mas num1 não é menor que num2.
    Console.WriteLine("num1 >= num2 returned false (but num1 < num2 also is false)");
```

Nullable Types

- Exceto quando utilizado o operador “!=”

```
if (num1 != num2)
{
    // Esta comparação é verdadeira, num1 e num2 nao são iguais.
    Console.WriteLine("Finally, num1 != num2 returns true!");
}
```

Nullable Types

- O operador *?? (null-coalescer)*
  - Define um valor padrão que é retornado quando um tipo anulável é atribuído à um tipo não-anulável.

```
int? x = null;

// Atribui o valor 0 à y se x for null
int y = x ?? 0;
```

  - Também pode ser utilizado com vários Nullable Types

```
int? x = null;
int? y = 0;
// Se x não for null, z = x.
// Se x for null, z = y.
int? z = x ?? y;
```

Tratamento de Exceções:

- Exceção
  - Qualquer evento não usual, errôneo ou não, detectável por hardware ou software que possa requerer um processamento especial.

Tratamento de Exceções:

- Exceções de Hardwre:
  - Operações de entrada e saída
  - Quando é erro de hardware
  - Ou quando é final de arquivo
  - Overflow de ponto flutuante
  - Exibição de erros / mensagens de diagnóstico
  - Após exibição o programa é finalizado

Tratamento de Exceções:

- Exceções de Software:
  - Detecção de erros de índices de faixas de matrizes
  - Situações em que precisa de um tratamento especifico antes de continuar
  - Opcional (C++ vs Csharp/Java)
  - Pode afetar performance

Tratamento de Exceções:

- Tratamento de Exceções
  - É o processamento especial que pode ser requerido quando uma exceção é detectada.

Tratamento de Exceções:

- Exceções podem ocorrer em momentos que não são pré-determinados.
- São mecanismos que permitem ao software reagir de forma determinada a certos erros em tempo de execução

Tratamento de Exceções:

- Quando o tratamento de exceções é incluído, um subprograma pode terminar de duas maneiras:
  - quando sua execução estiver completa
  - ou quando ela encontrar uma exceção.

Tratamento de Exceções:

- Finalização
  - Em algumas situações é necessário completar alguma computação independentemente de como a execução do subprograma termina. A habilidade de especificar tal computação é chamada de finalização.
- Onde a exceção continua, se é que continua, após o tratador de exceção completar sua execução?

Tratamento de Exceções:

```
try
{
    // Funcoes a serem executadas
}
catch (Exception e)
{
    // Este bloco é executado no momento em que ocorre uma exceção,
    // ou seja, captura os erros do bloco TRY
    // Utilização: tratar o erro, "lancar" a excecao, informar o usuário ou gerar um log
}
finally
{
    // Este bloco é sempre executado, tenha uma exceção sido capturada, ou nao
    // Utilização: Liberacao de recursos, fechamento de arquivos, desconectar do servidor
}
```

```
1 reference
static float CalcularMedia2(int valor, int quantidade)
{
    float media;
    float[] vet = new float[5];

    try
    {
        media = valor / quantidade;
        System.Console.WriteLine(media);
    }
    catch (DivideByZeroException ex)
    {
        throw new DivideByZeroException("Division by zero");
    }

    try
    {
        vet[5] = media;
    }
    catch (IndexOutOfRangeException ex)
    {
        throw new IndexOutOfRangeException("Index is out of range");
    }
    return 0; //Apenas um exemplo. Sei que este retorno não faz sentido :D
}
```

```
0 references
public static void ExecutarAlgumaCoisa()
{
    try
    {
        //Alguma coisa
    }
    catch (InvalidCastException e)
    {
        //...
    }
    catch (IndexOutOfRangeException e)
    {
        //...
    }
    catch (IOException e)
    {
        //...
    }
    catch (Exception e)
    {
        //...
    }
}
```

```
static double MeuMetodo(int vlr1, int vlr2)
{
    double resultado = 0;

    try
    {
        // Tratar erro previsto?
        // Permite customizar a mensagem
        if (vlr2 == 0)
            throw new DivideByZeroException("Erro divisão por zero.");
        else
            resultado = vlr1/vlr2;
    }
    catch (DivideByZeroException e)
    {
        Console.WriteLine(e.Message);
    }
    catch (Exception e)
    {
        Console.WriteLine("Erro detectado: " + e.Message);

        // Erro nao possui tratamento?
        // Pode lançar outra excecao que nao é tratada, causando a finalização do aplicativo
        throw;
    }

    return resultado;
}
```

Tratamento de Exceções:

```
// Exemplo Try-Catch-Finally
0 references
static void ExemploDivisao1(int vlr1, int vlr2)
{
    double resultado = 0;

    try
    {
        resultado = vlr1 / vlr2;
    }
    catch (DivideByZeroException e)
    {
        Console.WriteLine(e.Message);
    }
    finally
    {
        Console.WriteLine("Resultado: " + resultado);
    }
}
```

Propagação de Exceções:

- Uma exceção pode ser tratada fora do método onde ela ocorreu.

- Exceções podem se propagar pela hierarquia de métodos até que elas sejam tratadas ou quando elas atingem o método main.

```
1 reference
public static void ThrowTest()
{
    try
    {
        TryCast();
    }
    catch (Exception ex)
    {
        //Exceção: System.InvalidCastException
        Console.WriteLine("Levantada a exceção: {0}.", ex.GetType());
        throw;
    }
}

1 reference
public static void TryCast()
{
    string s = "Alguma coisa";
    object obj = s;
    //Tentará realizar um cast
    int result = (int)obj;
}
```

Tratamento de Exceções:

- A informação da exceção pode ser passada para o tratador?

```
1 reference
public static void TestDataException()
{
    try
    {
        TryCast();
    }
    catch (Exception e)
    {
        e.Data["ExtraInfo"] = "Informações do TryCast().";
        e.Data.Add("MoreExtraInfo", "Mais Informações do TryCast().");
        e.Data.Add("MoreExtraInfo2", "Mais Informações do TryCast()2.");
        //..
        //..
        //..
        foreach (DictionaryEntry item in e.Data)
        {
            Console.WriteLine(item.Value);
        }
        throw;
    }
}
```

MINICURSO