Bloco: Desenvolvimento .NET
Disciplina: Fundamentos de Desenvolvimento com C#

BIBLIOGRAFIA BÁSICA:

- SHARP, John. Microsoft Visual C# 2013: Passo a Passo. Bookman, 2014.
- NAGEL, Christian. Professional C# 5.0 and .NET 4.5.1.John Wiley & Sons, 2014.
- SKEET, Jon. C# in Depth. Manning, 2014.

BIBLIOGRAFIA COMPLEMENTAR:

- ALBAHARI, Joseph. C# 5.0 in a Nutshell. O' Reilly, 2012.
- WAGNER, Bill.Effective C#: 50 Specific Ways to Improve Your C#. Addison-Wesley, 2010.
- BEVIS, Tony. C# Design Pattern Essentials.Ability First, 2012.
- MARTIN, Robert.Princípios, Padrões e Práticas Ágeis em C#.Bookman, 2011.
- OSHEROVE, Roy.The Art of Unit Testing with Examples in C#. Manning , 2014.tapa 04

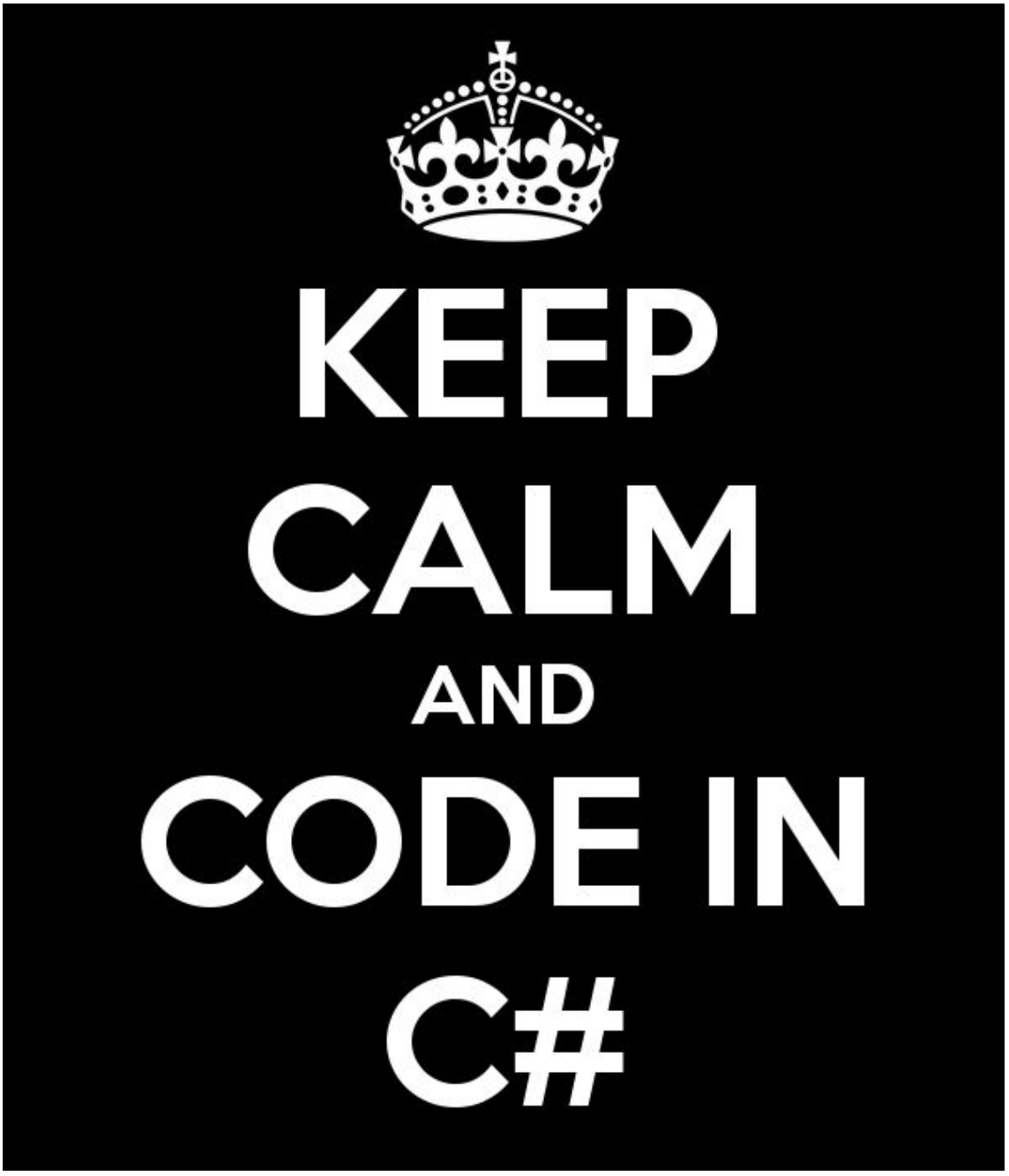
Disciplina: Fundamentos de Desenvolvimento com C#

a) ABERTURA:

Olá {GENERICO:type=aluno},

Seja bem-vindo à disciplina Fundamentos de Desenvolvimento com C#!

O objetivo desta disciplina é apresentar os conceitos básicos da construção de programas, utilizando a linguagem de programação C# e a plataforma .NET, através de uma abordagem prática, voltada para a construção de programas desde a primeira etapa da disciplina.



Prepare-se para entender mais sobre o .NET Framework e como você pode construir aplicativos desktop, mobile e web com ele. Você também vai conhecer o poderoso ambiente integrado de desenvolvimento, Visual Studio, e entender como ele pode facilitar o dia a dia do desenvolvedor.

Você será apresentado à linguagem de programação C# e verá como ela torna fácil e rápida a criação de aplicações. Nessa jornada, você conhecerá os tipos mais básicos de objetos e aprenderá como controlar o fluxo da sua aplicação através de seleção e/ou de iteração.

Além disso, você estudará como definir classes e instanciar objetos e, em pouco tempo, você será capaz de criar classes que, além de armazenar dados através de campos e propriedades, executam ações através de métodos. Você,

também, irá adquirir o conhecimento de como organizar todo o código escrito em bibliotecas para que seja possível a sua reutilização em diferentes projetos.

Em seguida, você entrará no mundo do desenvolvimento de software orientado a objetos. Irá entender, na prática, os importantes conceitos de encapsulamento, de herança e de polimorfismo.

Na reta final da disciplina, você irá entender mais sobre manipulação, leitura e escrita de dados.

Nesta disciplina, você vai desenvolver as seguintes competências:

1. Compreender os fundamentos de programação com C# .NET e Visual Studio.
2. Criar métodos, classes e implementar coleções de fontes.
3. Criar interfaces e hierarquias de classes usando herança.
4. Ler e escrever dados locais.

Etapa 01 - O .NET Framework e o Visual Studio.

Olá {GENERICO:type=aluno},

Já imaginou utilizar uma única linguagem de programação para criar tanto aplicativos desktop, quanto aplicativos web, ou aplicativos mobile? Isso é possível utilizando C# e o .NET framework!

Nesta etapa, você vai compreender o que é o .NET Framework e como ele nos ajuda a criar aplicativos desktop, web ou mobile utilizando a linguagem de programação C#.

Irá conhecer o poderoso ambiente integrado de desenvolvimento: Visual Studio.

Vamos lá.

>> Clique no link: Roteiro de aprendizagem para iniciar essa etapa.

Visão geral: .NET Framework

{GENERICO:type=aluno}, o .NET Framework é um ambiente de execução gerenciado que fornece uma variedade de serviços para seus aplicativos em execução. Ele possui dois componentes principais:

Common Language Runtime (CLR)

que é o mecanismo de execução que gerencia a execução de aplicativos, e

Biblioteca de Classes do .NET Framework

que fornece aos desenvolvedores código reutilizável e testado pronto para utilizar.

É fundamental que você acesse o recurso abaixo:

Introdução ao .NET Framework
https://docs.microsoft.com/pt-br/dotnet/framework/get-started/index

Visão geral: Visual Studio

{GENERICO:type=aluno}, o Visual Studio é um ambiente integrado de desenvolvimento. Com ele nós conseguimos escrever código de maneira eficiente e eficaz, depurar e testar com muita facilidade e produtividade.

É fundamental que você acesse os recursos abaixo:

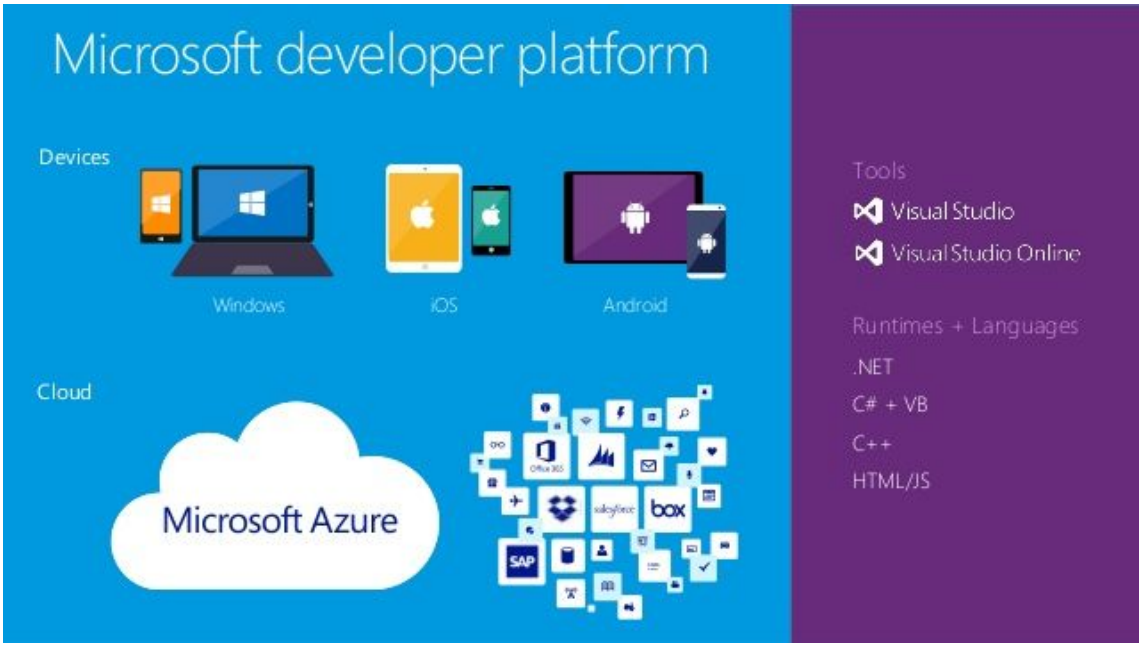
Para conhecer melhor o Visual Studio, acesse:

Visual Studio IDE
https://docs.microsoft.com/pt-br/visualstudio/ide/visual-studio-ide

Documentação do Visual Studio
https://docs.microsoft.com/pt-br/visualstudio/

Página de download do Visual Studio
https://www.visualstudio.com/pt-br/

.NET Framework + Visual Studio



Ref.: MasterGrade IT

{Aluno}, utilizando o Visual Studio como IDE e o .NET Framework como ambiente de execução, você pode criar quaisquer tipos de aplicativos: desde apps mobile, até páginas web, incluindo aplicativos para desktop.

Dentro do Visual Studio você encontra templates de todos esses tipos de projetos, o que facilita ainda mais o início da construção de uma solução.

É fundamental que você acesse os recursos abaixo:

Saiba mais, acesse:

Soluções e projetos no Visual Studio
https://docs.microsoft.com/pt-br/visualstudio/ide/solutions-and-projects-in-visual-studio

Introdução ao desenvolvimento com o Visual Studio
https://docs.microsoft.com/pt-br/visualstudio/ide/get-started-developing-with-visual-studio

Chegou a hora de você mostrar o que aprendeu. Acesse o link abaixo e faça a avaliação.

Teste Objetivo – 1A

Nessa etapa, você desenvolveu a seguinte competência:

Compreender os fundamentos de programação com C# .NET e Visual Studio.

- Visão geral do desenvolvimento de aplicações com .NET e Visual Studio.

Etapa 02 - A Criação de Programas Simples com C#

Olá {GENERICO:type=aluno},

Quando estamos aprendendo uma nova linguagem de programação, geralmente, o primeiro programa que escrevemos é o famoso Hello World!

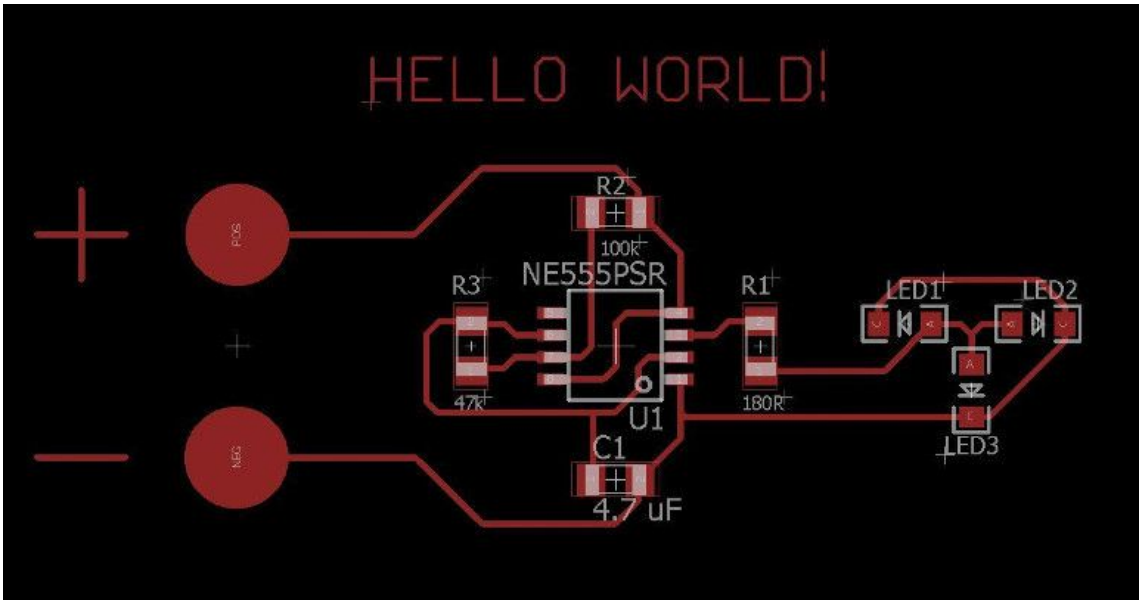
Nesta etapa, você vai criar um Hello World! e outros programas simples utilizando o Visual Studio, o .NET Framework e C#, como linguagem de programação.

Está preparado para seus primeiros passos no mundo da programação?

Vamos lá?

>> Clique no link: Roteiro de aprendizagem para iniciar essa etapa.

Um Programa Simples com C#: Hello World



Ref.: Support

{GENERICO:type=aluno}, quando estamos aprendendo uma nova linguagem de programação, geralmente, o primeiro programa que escrevemos é o famoso Hello World!

Esse programa é bem simples, apenas exibe na tela um texto, mas nos ajuda muito a começar a compreender a estrutura da linguagem que estamos aprendendo e algumas de suas características.

```
// A Hello World! program in C#.
using System;
namespace HelloWorld
{
    class Hello
    {
        static void Main()
        {
            Console.WriteLine("Hello World!");

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
```

```
        }
    }
}
```

Modificando nosso programa simples: Hello {GENERICO:type=aluno}!

{GENERICO:type=aluno}, para ficarmos um pouco mais confortáveis com a estrutura do programa que acabamos de ver, vamos fazer algumas alterações neles. O objetivo é que com essas alterações, um usuário possa digitar seu nome, pressionar a tecla enter, e ver uma mensagem de saudação com seu nome.

Esse programa também é bastante simples, mas além de exibir na tela um texto, ele também obtém o nome do usuário.

```
// A Hello World! program in C#.
using System;
namespace HelloWorld
{
    class Hello
    {
        static void Main()
        {
            Console.WriteLine("Digite seu nome:");

            var name = Console.ReadLine();
            Console.WriteLine($"Hello, {name}!");

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}
```

Para o nosso programa ficar um pouco mais “profissional”, poderíamos pedir para o usuário tanto o seu nome como o seu sobrenome.

```
// A Hello World! program in C#.
using System;
namespace HelloWorld
{
```

```
    class Hello
    {
        static void Main()
        {
            Console.WriteLine("Digite seu nome:");
            var firstName = Console.ReadLine();

            Console.WriteLine("Digite seu sobrenome:");
            var lastName = Console.ReadLine();

            Console.WriteLine($"Hello {firstName} {lastName}!");

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}
```

Você conseguiria criar um programa simples que exibe na tela o resultado da soma de dois números?

É fundamental que você acesse os recursos abaixo:

Saiba mais em:

Programa Simples com C#: Hello World
https://msdn.microsoft.com/pt-br/library/k1sx6ed2.aspx

Estrutura geral de um programa em C#
https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/inside-a-program/general-structure-of-a-csharp-program

Se você quiser complementar seus conhecimentos, acesse o recurso abaixo:

C# Hello World para Iniciantes
https://www.youtube.com/watch?v=rKQYRKO5izM

Chegou a hora de você mostrar o que aprendeu. Acesse o link abaixo e faça a avaliação.

Teste Objetivo – 2A

Teste de Performance - TP1

Nessa etapa, você desenvolveu as seguintes competências e suas subcompetências:

Compreender os fundamentos de programação com C# .NET e Visual Studio.

- Compreender a criação de programas simples com C# (Hello World!)

Etapa 03 - O Uso de Variáveis e os Tipos Básicos de Objetos

Olá {GENERICO:type=aluno},

Na etapa anterior, você criou um Hello World! e outros programas simples utilizando o Visual Studio, o .NET Framework e C#, como linguagem de programação.

Você sabe como o computador armazena cada pedacinho de informação que utilizamos nos nossos programas? Vamos conhecer um pouco um pouco mais sobre variáveis e seus diferentes tipos básicos!

Nesta etapa, você vai compreender a declaração e o uso de variáveis, assim como conhecer alguns tipos básicos de objetos que o .NET Framework disponibiliza.

>> Clique no link: Roteiro de aprendizagem para iniciar essa etapa.

Visão geral: Variáveis

{GENERICO:type=aluno}, em programação, uma variável é um objeto capaz de reter e representar um valor ou uma expressão.

As variáveis são espaços reservados na memória do computador, geralmente, são associadas a nomes, que as identificam durante o desenvolvimento, e possuem um tipo, que indica o que pode ser armazenado naquele espaço de memória.

É fundamental que você acesse os recursos abaixo:

Saiba mais, acesse:

Tipos (Guia de Programação em C#)
https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/types/

Learning About Built-In Types and Variables
https://www.microsoft.com/net/tutorials/csharp/getting-started/types-variables

Tipos de valor e tipos de referência

{GENERICO:type=aluno}, existem dois tipos distintos de variáveis em C#:

Os tipos de valor	Os tipos de referência

As variáveis que são tipos de valor armazenam dados e as variáveis que são tipos de referência armazenam referências aos dados reais. O que chamamos de objetos, na verdade são os tipos de referência.

```
// variável de um tipo de valor
int i = 5;

// variável de um tipo de referência
object o = new object();
```

É possível converter um tipo de valor em um tipo de referência, e também é possível converter um tipo de referência, que seja um tipo de valor, em um tipo de valor. Essas conversões são chamadas de boxing e unboxing.

```
// variável de um tipo de valor
int i = 5;

// boxing
object o = i;
```

```
// unboxing
i = o as int;
```

Os tipos de referência podem não fazer referência a nenhum dado, quando isso acontece atribuímos nulo a essa variável.

É fundamental que você acesse os recursos abaixo:

Tipos de valor (Referência de C#)
https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/value-types

Tipos de referência (Referência de C#)
https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/reference-types

Conversões boxing e unboxing (Guia de Programação em C#)
https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/types/boxing-and-unboxing

Tipos internos do C#

{GENERICO:type=aluno}, no C# existe um conjunto padrão de tipos, alguns deles para representar valores numéricos, expressões booleanas, caracteres de texto e outros tipos de dados.

Os tipos de valor internos do C# são: bool, byte, sbyte, char, decimal, double, float, int, uint, long, short e ushort. Os tipos de referência internos do C# são: object e string.

Cada um desses tipos possui seu correspondente no namespace System. O int do C# é apenas um alias para o tipo System.Int32 e o tipo float do C# é apenas um alias para o tipo System.Single, por exemplo.

É fundamental que você acesse os recursos abaixo:

Tabela de tipos internos (Referência de C#)
https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/built-in-types-table

Tabela de tipos integrais (Referência de C#)
https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/integral-types-table

Tabela de tipos de ponto flutuante (Referência de C#)
https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/floating-point-types-table

O tipo string

{GENERICO:type=aluno}, alguns tipos representam uma sequência elementos de outros tipos. Um exemplo desses tipos são as strings.

As strings representam uma sequência de zero ou mais caracteres, ou seja, uma sequência de elementos do tipo char. O conteúdo das strings é imutável, uma vez criada, seu valor não pode ser alterado.

Strings são tipos de referência, enquanto não atribuímos um valor a ela, seu valor é nulo. Para criar uma string vazia, o valor “” ou String.Empty deve ser atribuído a ela.

É fundamental que você acesse os recursos abaixo:

string (Referência de C#)
https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/string

Working with Strings
https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/strings/

O tipo System.DateTime

{GENERICO:type=aluno}, o tipo System.DateTime é o que utilizamos quando precisamos trabalhar com tempo.

Com ele você consegue acesso estático ao relógio do sistema operacional que pode ser utilizado para obtermos a data e hora atual.

É fundamental que você acesse o recurso abaixo:

Working with Dates and Times
https://docs.microsoft.com/pt-br/dotnet/api/system.datetime?view=netframework-4.7.1

Programação em C# - Declaração e Atribuição de Variáveis
https://www.youtube.com/watch?v=7lLIdUaDLJA

Chegou a hora de você mostrar o que aprendeu. Acesse o link abaixo e faça a avaliação.

Teste Objetivo – 3A

Nessa etapa, você desenvolveu as seguintes competências e suas subcompetências:

Compreender os fundamentos de programação com C# .NET e Visual Studio.

- Compreender a declaração e o uso de variáveis e os tipos básicos de objetos.

Etapa 04 – O Controle do Fluxo de Aplicações: Seleção e Iteração

Olá {GENERICO:type=aluno},

Quando estamos construindo uma aplicação utilizando uma linguagem de programação, raramente, queremos que ela execute os comandos que escrevemos no código de maneira estritamente sequencial. Na maioria das aplicações, em muitos momentos, gostaríamos de poder escolher entre uma ou outra sequência de comandos dada uma determinada condição. Outra situação muito comum quando construímos aplicações é querermos que uma sequência de comandos seja executada um determinado número de vezes.

Essas alterações no fluxo da aplicação podem ser controladas através de estruturas de seleção e de iteração. As estruturas de seleção mais comuns são o if-else e o switch-case e as estruturas de iteração mais comuns são o for, o foreach e o while.

>> Clique no link: Roteiro de aprendizagem para iniciar essa etapa.

As expressões lógicas

{GENERICO:type=aluno}, antes de entendermos melhor como podemos controlar o fluxo das nossas aplicações, vamos conhecer as expressões lógicas.

As expressões lógicas são compostas de operadores e operandos. Depois de avaliadas, essas expressões resultam em um valor do tipo bool, ou seja, resultam em true ou false.

Existem dois tipos de operadores para as expressões lógicas: operadores de comparação e operadores lógicos.

Os operadores igual (==), diferente (!=), maior que (>), menor que (<), maior que ou igual (>=) e menor ou igual (<=) são operadores de comparação. Já os operadores e (&&), ou (||), negação (!) e ou exclusivo (^) são operadores lógicos.

Operadores de Comparação

==	Igual
!=	Diferente
>	Maior que
<	Menor que
>=	Maior que ou igual
<=	Menor que ou igual

Operadores Lógicos

&&	E
||	Ou
!	Negação
^	Ou exclusivo

Ao aplicar os operadores lógicos, podemos obter os resultados abaixo:

true && true	true
true && false	false
false && true	false
false && false	false
true || true	true
true || false	true
false || true	true
false || false	false
!true	false
!false	true
true ^ true	false
true ^ false	true
false ^ true	true
false ^ false	false

Link

Os fluxos de seleção

{GENERICO:type=aluno}, agora que entendemos o que são as expressões lógicas e sabemos que elas resultam em valores booleanos, temos tudo o que precisamos para realizar fluxos de seleção em nossos programas.

Os fluxos de seleção podem ocorrer de duas formas:

Com blocos if-else, ou	Com blocos switch-case

Blocos if-else

Nos blocos if-else, avaliamos uma expressão lógica. Caso o valor resultante da expressão seja true executamos um trecho de código, caso contrário executamos outro trecho de código.

```
if (expressaoLogica)
{
    // Se a expressaoLogica for verdadeira executamos o trecho de código entr
e essas chaves
}
else
{
    // Se a expressaoLogica não for verdadeira executamos o trecho de código
entre essas chaves
}
```

No caso de não querermos executar nenhum trecho de código no caso de a expressão lógica não ser verdadeira, podemos ter um bloco if, sem o subsequente else.

```
if (expressaoLogica)
{
    // Se a expressaoLogica for verdadeira executamos o trecho de código entr
e essas chaves
}
```

Blocos switch-case

Quando utilizamos o switch-case, queremos executar trechos diferentes de código de acordo com o valor de uma determinada variável. O switch-case é muito utilizado quando temos uma série de casos para valores conhecidos.

```
 switch (condicao)
{
    case caso1:
        // Se a condicao for igual ao caso1 executamos esse trecho de código
entre o case e o break
        break;
    case caso2:
        // Se a condicao for igual ao caso2 executamos esse trecho de código
entre o case e o break
        break;
    default:
        // Se a condicao diferente de todos os casos acima executamos esse tr
echo de código
}
```

É fundamental que você acesse o recurso abaixo:

Instruções de Seleção
https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/selection-statements

Os fluxos de iteração

{GENERICO:type=aluno}, como mencionamos no início da etapa, algumas vezes desejamos que algum trecho de código seja executado um determinado número de vezes.

Duas formas de criarmos fluxos de iteração são com as estruturas de repetição:

while	for

while

Com a estrutura de repetição while, podemos executar um trecho de código enquanto uma condição continuar verdadeira. Normalmente, a cada execução do trecho de código, alguma lógica pode alterar um valor tornando a condição falsa, possibilitando o fim da repetição. Outra maneira de finalizar a repetição é utilizando um break.

```
var i = 10;
while (i > 0)
{
    Console.WriteLine(i);

    if (i == 5)
    {
        break;
    }

    i--;
}
```

for

Na estrutura de repetição for, podemos ter, uma variável que será alterada a cada passo da execução, uma condição que será avaliada a cada passo da execução, e uma operação que será realizada no final de cada passo da execução.

```
var count = 10;
for (int i = 0; i < count; i++)
{
    Console.WriteLine(i);
}
```

É fundamental que você acesse o recurso abaixo:

Instruções de Iteração
https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/iteration-statements

Arrays e coleções

{GENERICO:type=aluno}, arrays e coleções são tipos criados para possibilitar o armazenamento de múltiplas variáveis de um mesmo tipo.

Quando instanciamos um array, devemos informar um tamanho fixo. Podemos declarar um array e inicializá-lo na mesma linha de código e assim não precisamos definir seu tamanho. Um array que não foi instanciado ou inicializado tem o valor nulo.

```
// Array com valor nulo
int[] primeiroArray;

// Array instanciado com tamanho 3
int[] segundoArray = new int[3];

// Array inicializado com 3 elementos
int[] terceiroArray = new int[] { 1, 2, 3 };
```

Para acessarmos os elementos do array devemos lembrar que a primeira posição é a posição de índice zero.

```
int[] meuArray = new int[] { 1, 2, 3, 4, 5 };

// Imprime no console o elemento 1
Console.WriteLine(meuArray[0]);
```

Uma das coleções mais utilizadas no .NET é o tipo List. Diferente do array, o tipo List não possui um tamanho fixo, o que o torna um pouco mais flexível que o array.

```
List<int> meuList = new List<int>();

meuList.Add(1);
meuList.Add(2);
meuList.Add(3);
meuList.Add(4);
meuList.Add(5);

// Um list com os elementos 1, 2, 3, 4 e 5
```

Para percorrermos os elementos de arrays e coleções, temos um fluxo de iteração diferente do que vimos anteriormente. Esse fluxo de iteração pode ser

definido com a estrutura de iteração foreach, que executa um trecho de código para cada elemento do enumerável.

```
var numeros = new List<string>() { "Um", "Dois", "Três" };

foreach (var numero in numeros)
{
    // Imprime no console cada um dos numeros do List
    Console.WriteLine(numero);
}
```

É fundamental que você acesse os recursos abaixo:

Working with Arrays and Collections
https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/concepts/collections

Link

Se você quiser complementar seus conhecimentos, acesse os recursos abaixo:

Programação em C# - Operadores Relacionais e Lógicos

Programação em C# - Estruturas de Decisão Condicional Simples e Composta
https://www.youtube.com/watch?v=TkmTlg9rEMI

Programação em C# - Declaração Condicional Switch
https://www.youtube.com/watch?v=NpG6jpWBV6E

Programação em C# - Estrutura de Repetição While
https://www.youtube.com/watch?v=Cdf-d7_1c5g&t=13s

Programação em C# - Estrutura de Repetição For
https://www.youtube.com/watch?v=a4g7av96p7E

Programação em C# - Arrays - Declarando e Instanciando
https://www.youtube.com/watch?v=doD391y7fWM

Chegou a hora de você mostrar o que aprendeu. Acesse o link abaixo e faça a avaliação.

Teste Objetivo – 4A

Nessa etapa, você desenvolveu as seguintes competências e suas subcompetências:

Compreender os fundamentos de programação com C# .NET e Visual Studio.

- Compreender a declaração e o uso de variáveis e os tipos básicos de objetos.
- Compreender o controle do fluxo de aplicações: seleção e iteração.

Etapa 05 – Os Métodos e seus Modificadores de Acesso

Olá {GENERICO:type=aluno},

Você sabe o que é um método? É hora de começar a descobrir!

Nesta etapa, você vai compreender os métodos como o primeiro mecanismo de organização e reaproveitamento de código. Vai compreender como declarar e chamar métodos. Também vai começar a entender um conceito muito importante, o de modificadores de acesso.

>> Clique no link: Roteiro de aprendizagem para iniciar essa etapa.

Declaração e Utilização de Métodos

{GENERICO:type=aluno}, métodos são uma das unidades mais básicas de organização de código. Utilizando métodos, podemos agrupar um conjunto de instruções que se repete com frequência em nosso código, e que façam sentido serem chamadas sequencialmente, para reduzir a quantidade de código duplicado e aumentar a reutilização do código.

Um método tem um nome e um corpo. É importante que o nome descreva exatamente o que o método faz. O corpo do método é composto pelo conjunto de instruções que serão executadas quando o método for executado. Os métodos também podem receber dados necessários para realizar seu processamento e podem retornar alguma informação resultante do processamento feito.

Nós utilizamos métodos desde o primeiro programa que escrevemos.

Ao escrevermos na saída padrão utilizando a instrução Console.WriteLine, estamos utilizando um método cujo nome é WriteLine. Esse método, da maneira que utilizamos, espera receber uma string como parâmetro e o resultado de sua execução é a exibição da string informada na saída padrão do sistema operacional.

```
// A Hello World! program in C#.
using System;
namespace HelloWorld
{
    class Hello
    {
        static void Main()
        {
            Console.WriteLine("Hello World!");

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}
```

Assim seria a declaração de um método Soma, que recebe dois números inteiros, um como a, outro como b, e retorna outro número inteiro como o resultado.

```
static int Soma(int a, int b)
{
    return a + b;
}
```

Como pudemos ver acima, para retornarmos alguma informação sobre o resultado de um processamento de um método, utilizamos a palavra reservada return.

A utilização de um método é um processo bem simples. Temos apenas que chamá-lo pelo nome, passando os parâmetros necessários para a sua execução. Veja o exemplo abaixo:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HelloMethods
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 3;
            int y = 5;

            int s = Soma(x, y);
        }

        static int Soma(int a, int b)
        {
            return a + b;
        }
    }
}
```

É fundamental que você acesse os recursos abaixo:

Saiba mais, acesse:

Métodos (Guia de Programação em C#)

https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/methods

Passagem de parâmetros

{GENERICO:type=aluno}, vimos que os métodos possuem uma assinatura e um corpo que contém as instruções que serão executadas quando o método for chamado. Na assinatura do método, temos seu nome e parâmetros.

Os parâmetros podem ser passados tanto por valor quanto por referência. A passagem de parâmetros por referência faz com que o valor daquele

parâmetro, ao ser alterado dentro do método, continue alterado quando a execução do método acaba. Para que ocorra essa passagem de parâmetro por referência utilizamos as palavras reservadas ref e out.

```
static void Soma(int a, int b, out int s)
{
    s = a + b;
}
```

Nessa outra implementação do método Soma, estamos passando o parâmetro s, por referência, para ser preenchido dentro do método, ao invés de ser retornado.

É fundamental que você acesse os recursos abaixo:

Saiba mais, acesse:

Passando parâmetros (Guia de Programação em C#)

https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/passing-parameters

Passando matrizes com o uso de ref e out (Guia de Programação em C#)

https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/arrays/passing-arrays-using-ref-and-out

Passando parâmetros de tipo de valor (Guia de Programação em C#)

https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/passing-value-type-parameters

Passando parâmetros de tipo de referência (Guia de Programação em C#)

https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/passing-reference-type-parameters

Tipos de retorno

{GENERICO:type=aluno}, além de termos nome e parâmetros na assinatura dos métodos, definimos também os tipos que são retornados por aqueles métodos. Existem métodos que não retornam valor algum, e isso é indicado utilizando a palavra reservada void no lugar do tipo de retorno na assinatura do método.

Se o tipo de retorno não for void, o método poderá retornar um valor utilizando a palavra reservada return. A palavra return interrompe a execução do método.

```
static int Soma(int a, int b)
{
    return a + b;

    // Não será executado
    a = 3;
}
```

No exemplo acima, a instrução a = 3 não será executada pois o return encerra a execução do método.

É fundamental que você acesse os recursos abaixo:

Saiba mais, acesse:

Métodos (Guia de Programação em C#)

https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/methods

Métodos estáticos

{GENERICO:type=aluno}, os métodos que viemos utilizando possuem uma particularidade: o modificador static em sua assinatura. Esse modificador indica que o método pode ser chamado mesmo quando nenhuma instância da classe a que ele pertence foi criada.

É fundamental que você acesse os recursos abaixo:

Saiba mais, acesse:

Classes static e membros de classes static (Guia de Programação em C#)

https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members

Exceções: try, catch e finally

{GENERICO:type=aluno}, exceções são erros que os programas lançam sempre que operações não desejadas são realizadas. Por exemplo, se você tentar dividir algum número por zero em uma aplicação em C#, seu programa irá lançar uma exceção para te avisar que aquela é uma operação não desejada, a DivideByZeroException. Quando um programa lança uma exceção, seu fluxo normal é interrompido.

Utilizamos um bloco **try**, **catch** e (algumas vezes) **finally** quando suspeitamos que um trecho de código pode lançar uma exceção.

```
double Divisao(double a, double b)
{
    try
    {
        return a / b;
    }
    catch (System.Exception ex)
    {
        throw ex;
    }
}
```

No bloco **try** colocamos o trecho de código que suspeitamos que pode lançar uma exceção, no exemplo acima, a divisão entre dois números. Caso o denominador da divisão seja o número 0, aquele trecho de código irá lançar uma exceção, que será capturada pelo bloco catch.

No bloco **catch** temos algumas opções. Uma delas é lançar a exceção adiante, utilizando o throw. Nesse caso, estamos contando que quem utiliza o nosso método será o responsável por tratar o erro ocorrido. Outra opção é tratarmos o erro ocorrido. Essa opção pode ocorrer de diversas formas. Dependendo do contexto da sua aplicação, tratar um erro pode ser exibir uma mensagem amigável para o usuário, realizar um log da operação ocorrida e de seus parâmetros, ou retornar um valor padrão para o usuário.

Quando ocorre uma exceção no nosso código, o fluxo da aplicação é interrompido na linha daquela instrução, e caso esse erro tenha ocorrido dentro de um bloco try, o fluxo da aplicação é redirecionado para o bloco catch. Algumas vezes, o trecho de código executado até o momento do erro precisa ser desfeito, ou os recursos utilizados até ali precisam ser liberados. Para isso utilizamos o bloco finally.

```
double Divisao(double a, double b)
{
```

```
    try
    {
        return a / b;
    }
    catch (System.Exception ex)
    {
        throw ex;
    }
    finally
    {
        // As intruções aqui são sempre executadas;
    }
}
```

O bloco **finally** é executado sempre que o fluxo do código entrar em um bloco try, dado que o bloco finally tenha sido definido, mesmo que haja um return no try ou no catch, ou você tenha feito um throw no catch.

É importante ressaltar que os blocos try, catch e finally devem ser utilizados quando temos situações em que não é possível retornar um resultado válido. De um modo geral, é preferível verificar e tratar a condição de erro com um bloco de seleção if-else.

```
double Divisao(double a, double b)
{
    if (a < 0 && b == 0)
    {
        return Double.NegativeInfinity;
    }
    else if (b == 0)
    {
        return Double.PositiveInfinity;
    }
    else
    {
        return a / b;
    }
}
```

É fundamental que você acesse os recursos abaixo:

Saiba mais, acesse:

try-catch-finally (Referência de C#)

https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/try-catch-finally

Chegou a hora de você mostrar o que aprendeu. Acesse o link abaixo e faça a avaliação.

Teste Objetivo – 5A

Nessa etapa, você desenvolveu as seguintes competências e suas subcompetências:

Criar métodos, classes e implementar coleções de fontes.

- Compreender como declarar e chamar métodos.
- Compreender o conceito de escopo e os modificadores de acesso.
- Compreender a diferença entre os modificadores static, constant e read only.

Etapa 06 – A Criação de Classes e de Bibliotecas

Olá {GENERICO:type=aluno},

Você já percebeu que ao declararmos uma variável, definimos o seu tipo. O mesmo ocorre quando criamos um método, definimos os tipos dos parâmetros de entrada e o tipo que aquele método retorna. Isso ocorre porque C# é uma linguagem de programação fortemente tipada, ou seja, todas as variáveis têm um tipo específico.

Nesta etapa, você vai compreender como definir uma classe e como instanciar objetos daquele tipo, ou seja, do tipo daquela classe. Também vai

compreender o conceito de escopo e vai entender o que são os modificadores de acesso. Vai entender a diferença entre os modificadores static, constant e readonly.

Vai aprender a criar bibliotecas para maximizar o reuso de código e poder compartilhar o mesmo conjunto de classes, ou métodos, com diferentes projetos.

>> Clique no link: Roteiro de aprendizagem para iniciar essa etapa.

Definição de Classes e Instanciação de Objetos

{GENERICO:type=aluno}, em C#, classes são as definições dos tipos que utilizamos nos nossos programas. Nas classes, definimos, que campos, propriedades e métodos aquele determinado tipo vai ter, e de onde poderemos acessar cada um daqueles campos, propriedades e métodos.

Um objeto, em C#, é uma instância de uma classe, ou seja, utilizando uma classe podemos criar diversos objetos naquele mesmo molde. Costumamos dizer, que as propriedades que definimos em uma classe representam o estado de um objeto e que os métodos que definimos em uma classe representam seu comportamento.

Quando estamos desenvolvendo um programa, a grande maioria dos arquivo .cs que criamos, será para definirmos uma classe.

Imagine que estamos querendo criar uma representação de um carro no nosso programa. Para isso criaremos uma classe chamada Carro, como abaixo:

```
public class Carro
{

}
```

Para instanciarmos um objeto do tipo carro no nosso programa, devemos utilizar a palavra reservada new, como na seguinte linha de código:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HelloMethods
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            var meuCarro = new Carro();
        }
    }

    public class Carro
    {

    }
}
```

Com isso criamos um objeto do tipo Carro que associamos à variável meuCarro.

Repare que após o new, chamamos a classe Carro, como se fosse um método. Ao chamarmos Carro(), estamos chamando o construtor da classe Carro.

O construtor é como um método que pode ser definido com o mesmo nome da classe e que é responsável pela inicialização de seus campos ou propriedades. Se não definirmos um construtor para a nossa classe, o .NET define um construtor padrão, sem parâmetros para a nossa classe automaticamente.

É fundamental que você acesse os recursos abaixo:

Saiba mais, acesse:

Classes (Guia de Programação em C#)

https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/classes

Conceitos e Exemplos – Instanciação: Estrutura da Linguagem

http://www.devmedia.com.br/conceitos-e-exemplos-instanciacao-estrutura-da-linguagem/18817

Programação em C# - Classes - Criando e Instanciando - 28

https://www.youtube.com/watch?v=NY2ENg4ejXk

Definições de Escopo e Modificadores de Acesso

{GENERICO:type=aluno}, nas etapas anteriores já definimos diferentes escopos.

Quando utilizamos um bloco if/else, ou um bloco foreach, ou um bloco try/catch, se declaramos uma variável entre as chaves ({}) de cada um desses blocos, essa variável só pode ser utilizada ali dentro, ou seja, aquele é o escopo daquela variável.

Quando definimos um método, se declaramos uma variável dentro dele, aquela variável é criada toda vez que o método é chamado e é destruída quando aquele método termina a sua execução, caso essa variável não seja retornada pelo método.

Alguns objetos que criamos, tem seu tempo de vida relacionado à classe em que foi criado. Em geral, os campos e as propriedades de uma classe só existem enquanto o objeto que os contém também existe.

Vamos modificar um pouco o código do nosso primeiro programa, o Hello World!:

```
using System;

namespace ConsoleApplication
{
    public class Program
    {
        private int contador = 0;
        public void Main()
        {
            int max = 10;
            for (int i = 0; i < max; i++)
            {
                Console.WriteLine("Hello World!");
            }
            contador += max;
        }
    }
}
```

Neste código modificado temos exemplos dos 3 tipos de escopo de variável descritos acima. O tempo de vida da variável contador é o mesmo tempo de vida da classe Program, ou seja, enquanto um objeto do tipo Program estiver “vivo” assim também estará o contador. Toda vez que chamarmos o método Main criaremos uma variável max que utilizamos dentro desse método para

atualizar a variável contador. Também temos a variável i, que existe somente dentro do bloco for.

É fundamental que você acesse o recurso abaixo:

Saiba mais, acesse:

Modificadores de acesso (Referência de C#)

https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/access-modifiers

Repare também nas palavras reservadas public e private antes da declaração da classe Program e antes da declaração da variável contador. Essas palavras reservadas são modificadores de acesso.

Além das palavras reservadas public e private, também temos as palavras reservadas internal e protected para modificar o acesso de classes, métodos ou variáveis.

O modificador de acesso public indica que aquele código pode ser acessado e utilizado por qualquer outro trecho de código. O modificador de acesso private indica que aquele código pode ser acessado apenas por trechos de código dentro da mesma classe em que o primeiro foi declarado.

É fundamental que você acesse o recurso abaixo:

Saiba mais, acesse:

public (Referência de C#)

https://msdn.microsoft.com/pt-br/library/yzh058ae.aspx

private (Referência de C#)

https://msdn.microsoft.com/pt-br/library/st6sy9xe.aspx

Voltando ao nosso exemplo, em qualquer lugar do código, em outras classes, por exemplo, poderíamos acessar e criar uma outra instância da classe Program. Já a variável contador não estaria acessível para quem estivesse criando esses objetos, essa é uma variável de uso interno da classe.

Os outros modificadores de acesso, o protected e o internal, tornam o código mais acessível do que se fosse modificado pelo private, porém menos acessível do que se fosse modificado pelo public.

O modificador de acesso protected é utilizado quando estamos lidando com herança, um conceito que veremos na próxima etapa. Ele permite que uma classe que herda de outra possa acessar o código modificado pelo protected porém nenhum outro código externo possa.

É fundamental que você acesse o recurso abaixo:

Saiba mais, acesse:

protected (Referência de C#)

https://msdn.microsoft.com/pt-br/library/bcd5672a.aspx

O modificador de acesso internal permite que o código possa ser acessado por qualquer outro código dentro de um mesmo assembly.

É fundamental que você acesse o recurso abaixo:

Saiba mais, acesse:

internal (Referência de C#)

https://msdn.microsoft.com/pt-br/library/7c5ka91b.aspx

static, const e readonly

{GENERICO:type=aluno}, existem outros modificadores que são bastante utilizados em programas em C#. São eles o **static**, o **const**, e o **readonly**.

O **modificador static** pode ser utilizado em classes, campos, métodos, propriedades, operadores, eventos e construtores. O código modificado pela palavra reservada static é acessado diretamente pelo seu nome. No caso de uma classe, esta não pode ser instanciada. No caso de membros de uma classe, estes são acessados a partir do nome da classe e não de sua instância.

É fundamental que você acesse o recurso abaixo:

Saiba mais, acesse:

static (Referência de C#)

https://msdn.microsoft.com/pt-br/library/98f28cdx.aspx

O **modificador const** indica que um campo é constante, ou seja, não pode ser modificado durante a execução do programa. Um campo const só pode ser inicializado na sua declaração.

É fundamental que você acesse o recurso abaixo:

Saiba mais, acesse:

const (Referência de C#)

https://msdn.microsoft.com/pt-br/library/e6w8fe1b.aspx

O **modificador readonly** indica que uma variável não poderá ser modificada durante a execução do programa mas diferente de um campo com o modificador const, essa variável, modificada pelo readonly, pode ser inicializada na sua declaração ou no construtor da classe a que pertence.

É fundamental que você acesse o recurso abaixo:

Saiba mais, acesse:

readonly (Referência de C#)

https://msdn.microsoft.com/pt-br/library/acdd6hb7.aspx

Bibliotecas e a Reutilização de Código

{GENERICO:type=aluno}, assim como foi destacado na etapa anterior, é importante pensar em classes, assim como em métodos, como meios que temos para reutilizar código. Ao percebermos que uma funcionalidade se repete devemos transformar aquela funcionalidade em um método ou em uma classe. Podemos pensar as classe como um conjunto de métodos, campos e propriedades relacionados, que gostaríamos de utilizar, na maior parte das vezes, em conjunto.

Também podemos pensar as bibliotecas da mesma maneira. As bibliotecas são pedaços de código, em geral um conjunto de arquivos contendo diversas classes relacionadas que gostaríamos de reutilizar. Podemos organizar as

classes internas a uma biblioteca em namespaces para segmentar um pouco mais.

O .NET Framework nos fornece diversas bibliotecas prontas com diferentes namespaces para diversas funcionalidades. Por exemplo, dentro do namespace System temos classes fundamentais e classes base que definem os tipos de dados de referência e valor comumente usados como o Console, Int32, Double e String.

É fundamental que você acesse o recurso abaixo:

Saiba mais, acesse:

Biblioteca de classes .NET Framework

https://msdn.microsoft.com/pt-br/library/gg145045(v=vs.110).aspx

Desenvolvendo Bibliotecas de Classe

https://msdn.microsoft.com/pt-br/library/gg420958.aspx

Utilizando bibliotecas de classes no .Net Framework

http://www.devmedia.com.br/utilizando-bibliotecas-de-classes-no-net-framework/27493

C# Tipos de Projetos - Criando Um Biblioteca de Classes - Arquivo Dll

https://www.youtube.com/watch?v=OtqX1u4alLs

Chegou a hora de você mostrar o que aprendeu. Acesse o link abaixo e faça a avaliação.

Teste Objetivo – 6A

Teste de Performance – TP3

Nessa etapa, você desenvolveu as seguintes competências e suas subcompetências:

Criar métodos, classes e implementar coleções de fontes

- Compreender a definição de classes e instanciação de objetos.
- Compreender o conceito de escopo e os modificadores de acesso.
- Compreender a diferença entre os modificadores static, constant e readonly.
- Compreender a criação de bibliotecas de classes para compartilhar código.

Etapa 07 – As Hierarquias de Classes e as Interfaces

Olá {GENERICO:type=aluno},

Os conceitos de métodos e os conceitos de classes nos ajudam na organização do código que estamos desenvolvendo. Quando percebemos que um trecho útil de código se repete por várias vezes em nosso código podemos extraí-lo para um método. Quando temos um conjunto de métodos que precisam manter um estado para fazer sentido e que expressam algum comportamento de modo coletivo podemos criar uma classe.

Com o uso de interfaces, quando programamos utilizando o C# como linguagem de programação, nossa aplicação ficará mais flexível e mais fácil de realizar alterações.

Nesta etapa, você vai aprender como utilizar interfaces e classes abstratas para desenvolver código pensando em abstrações, e não nas implementações das estruturas de dados que irá utilizar. Você vai entender mecanismos que permitem esconder membros de uma classe ou sobrescrever esses membros. E também, como você pode prevenir que outra classe herde de uma classe que criou.

Vamos lá.

>> Clique no link: Roteiro de aprendizagem para iniciar essa etapa.

Herança e Polimorfismo

{GENERICO:type=aluno}, o conceito de herança diz respeito à relação hierárquica entre classes. Se temos duas classes no nosso sistema, uma classe representando Pessoa e outra representando Aluno, e sabemos que o Aluno é uma Pessoa, ou seja, possui estado e comportamento comum com Pessoa, então a classe Aluno pode herdar de Pessoa. Com isso evitamos duplicar todo o código comum que aquelas classe teriam.

```
class Pessoa
{
    public string CPF { get; set; }
}

class Aluno : Pessoa
{
    public string Matricula { get; set; }
}
```

No exemplo acima, a classe aluno herda da classe pessoa. Com isso, evitamos escrever, novamente, que aluno possui CPF. O fato de pessoa ter CPF e de aluno herdar de pessoa faz com que aluno também tenha CPF.

Existem casos de classes que servem apenas para obrigar que as classes que herdarem dela respeitem uma API determinada. Essas classes são chamadas de classes abstratas e são modificadas pelo modificador abstract. Existe outro tipo de entidade que serve para obrigar que as classes que herdam dela respeitem uma API determinada. Essas entidades são chamadas interfaces.

```
abstract class Cachorro
{
    public abstract void Brincar();
}

interface IVoador
{
    void Voar();
}
```

O conceito de polimorfismo diz respeito às muitas formas que uma classe pode ter. Muitas vezes queremos que a classe derivada seja tratada como a classe

base. Outras vezes, as classes base podem definir métodos virtuais para que as classes derivadas possam substituí-los.

Caso a classe que herda de outra tenha comportamento diferente para um método específico, a linguagem nos permite sobrescrever esse método utilizando a palavra reservada override desde que esse método seja virtual. Um método de uma classe que é herdada, ou seja, de uma classe pai, pode ser modificado pelo operador virtual para permitir que as classes filhas sobrescrevam aquele método.

Uma classe derivada pode fazer com que um método virtual que ela implementa não seja substituído no caso de alguma outra classe herdar dela. Isso é possível com a utilização do operador sealed.

É fundamental que você acesse o recurso abaixo:

Saiba mais, acesse:

Herança (Guia de Programação em C#)

https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/inheritance

Polimorfismo (Guia de Programação em C#)

https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/polymorphism

Classes e membros de classes abstract e sealed (Guia de Programação em C#)

https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/abstract-and-sealed-classes-and-class-members

Classes abstratas e interfaces

{GENERICO:type=aluno}, as interfaces servem, de maneira geral, para estabelecer um contrato. Quando uma classe implementa uma interface, garantimos que ela apresenta o comportamento desejado, aquele estabelecido através do contrato.

```
interface IVoador
{
    void Voar();
}

class Aguia : IVoador
```

```
{
    public void Voar()
    {
        Console.WriteLine("Sou uma águia e sei voar muito bem!");
    }
}

class Galinha : IVoador
{
    public void Voar()
    {
        Console.WriteLine("Sou uma galinha e não sei voar direito!");
    }
}
```

Com as interfaces podemos reduzir o acoplamento do nosso código. Seu uso aumenta também a testabilidade do código, pois fica fácil substituir um objeto real que implementa a interface, por um objeto falso que também implementa a mesma interface para simularmos um comportamento na hora dos testes.

As classes abstratas também nos dão uma flexibilidade parecida, porém elas nos permitem implementar partes do código que gostaríamos de usar como base de algum contrato.

```
abstract class Cachorro
{
    public virtual void Brincar()
    {
        Console.WriteLine("Sou um cachorro e gosto de brincar!");
    }
}

class Labrador : Cachorro
{
    public override void Brincar()
    {
        Console.WriteLine("Sou um labrador e gosto muito de brincar!");
    }
}
```

Em ambos os casos não podemos instanciar diretamente uma classe abstrata ou uma interface.

```
// Error CS0144  Cannot create an instance of the abstract class or interface
 'Cachorro'
var cachorro = new Cachorro();

// Error CS0144  Cannot create an instance of the abstract class or interface
 'IVoador'
var voador = new IVoador();
```

É fundamental que você acesse o recurso abaixo:

Saiba mais, acesse:

Classes e membros de classes abstract e sealed (Guia de Programação em C#)

https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/abstract-and-sealed-classes-and-class-members

interface (Referência de C#)

https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/interface

Curso C# - Aula 9 - Trabalhando com Herança, Classes abstratas, e Interfaces.

https://www.youtube.com/watch?v=3IDrQlvuBJ0

Classe Abstrata e Interface

https://www.youtube.com/watch?v=npj3BxjRkJY

Metodos e Classes Abstratas

https://www.youtube.com/watch?v=gIIp8a2FKLQ

Chegou a hora de você mostrar o que aprendeu. Acesse o link abaixo e faça a avaliação.

Teste Objetivo – 7A

Nessa etapa, você desenvolveu as seguintes competências e suas subcompetências:

Criar interfaces e hierarquias de classes usando herança.

- Compreender o que é uma interface.
- Compreender como estender classes.

Etapa 08 – O Projeto e a Extensão de Classes

Olá {GENERICO:type=aluno},

Já vimos como criar classes simples, como criar interfaces que atuam como contratos entre as classes, e como criar classes abstratas que obrigam que algum comportamento seja implementado mas que também podem fornecer alguma implementação já pronta.

Nesta etapa, você vai entender mecanismos que permitem esconder membros de uma classe ou sobrescrever esses membros. E também, como você pode prevenir que outra classe herde de uma classe que criou.

Vamos lá.

>> Clique no link: Roteiro de aprendizagem para iniciar essa etapa.

Sobrescrever ou esconder membros

{GENERICO:type=aluno}, podemos sobrescrever membros quando implementamos um método virtual de uma classe base utilizando o modificador override. É uma maneira de especificar um comportamento mais específico para uma classe derivada.

Podemos esconder membros quando implementamos um método que não está definido como virtual em uma classe base, ou não está especificado como override, utilizando o modificador new.

```
public class ClasseBase
{
```

```
    public virtual void Metodo1()
    {
        Console.WriteLine("ClasseBase Metodo1");
    }

    public void Metodo2()
    {
        Console.WriteLine("ClasseBase Metodo2");
    }
}

public class ClasseDerivada : ClasseBase
{
    public override void Metodo1()
    {
        Console.WriteLine("ClasseDerivada Metodo1");
    }

    public new void Metodo2()
    {
        Console.WriteLine("ClasseDerivada Metodo2");
    }
}
```

Se utilizarmos a classe acima da seguinte maneira:

```
ClasseDerivada derivada = new ClasseDerivada();

derivada.Metodo1();
derivada.Metodo2();
```

O retorno será exatamente o que esperamos:

ClasseDerivada Metodo1

ClasseDerivada Metodo2

Se armazenamos a classe derivada em uma variável do tipo ClasseBase:

```
ClasseBase classeBase = new ClasseDerivada();

classeBase.Metodo1();
classeBase.Metodo2();
```

Qual seria o retorno? O mesmo que tivemos acima? Não.

ClasseDerivada Metodo1

ClasseBase Metodo2

Geralmente, esconder membros, é uma prática que devemos evitar. Ao esconder membros podemos ter efeitos inesperados.

É fundamental que você acesse o recurso abaixo:

Saiba mais, acesse:

override (Referência de C#)

https://msdn.microsoft.com/pt-BR/library/ebca9ah3.aspx

Quando usar as palavras-chave override e new (Guia de Programação em C#)

https://msdn.microsoft.com/pt-BR/library/ms173153.aspx

Programação em C# - Métodos Virtuais e Override - Herança

https://www.youtube.com/watch?v=gvOsmf_DGVA&t=16s

Prevenindo a herança

{GENERICO:type=aluno}, temos um mecanismo para evitar que uma classe seja utilizada como uma classe base para outras. Utilizando o operador sealed evitamos que outras classes herdem dessa classe que está selada.

Esse operador sealed também pode ser utilizado em métodos para evitar que eles continuem sendo sobrescritos.

```
public class ClasseBase
{
    public virtual void Metodo1()
    {
        Console.WriteLine("ClasseBase Metodo1");
    }
}

public sealed class ClasseDerivada1 : ClasseBase
{
    public override void Metodo1()
    {
        Console.WriteLine("ClasseDerivada1 Metodo1");
    }
}

public class ClasseDerivada2 : ClasseBase
{
```

```
    public sealed override void Metodo1()
    {
        Console.WriteLine("ClasseDerivada2 Metodo1");
    }
}
```

No exemplo acima, nenhuma outra classe pode derivar da ClasseDerivada1. Mas outras classes poderão herdar da ClasseDerivada2, porém, caso isso aconteça, essas classes não poderão sobrescrever o método 1 da ClasseDerivada2.

É fundamental que você acesse o recurso abaixo:

Saiba mais, acesse:

sealed (Referência de C#)

https://msdn.microsoft.com/pt-Br/library/88c54tsw.aspx

Classes e membros de classes abstract e sealed (Guia de Programação em C#)

https://msdn.microsoft.com/pt-BR/library/ms173150.aspx

Trabalhando com classes abstratas, classes seladas e Interfaces (Abstract class Sealed Class Interface - Classe Abstrata Classe Selada Interface)

http://www.devmedia.com.br/trabalhando-com-classes-abstratas-classes-seladas-e-interfaces-abstract-class-sealed-class-interface-classe-abstrata-classe-selada-interface/9548

aula 3340 c# classes e metodos Sealed Selada Finais

https://www.youtube.com/watch?v=5M2nzahmalw

Sealed Classes and Method in C#

https://www.youtube.com/watch?v=RswIBB3KVTQ

Estendendo classes sem utilizar herança

{GENERICO:type=aluno}, outro mecanismo muito poderoso em C# são os métodos de extensão. Com eles podemos ampliar as funcionalidades de classes sem ter que utilizá-las como classe base de uma implementação nossa. Veja o exemplo abaixo:

```
public static class Extensao
{
    public static bool EmailValido(this string email)
    {
        return Regex.IsMatch(email, @"[a-zA-Z0-9\.-_] +@[a-zA-Z0-9\.-_]+");
    }
}
```

No exemplo acima, estamos ampliando a funcionalidade da classe string. Estamos incluindo na classe string um método para a verificar se a string representa um email válido.

```
string email = "umteste@qualquer.com";

if (email.EmailValido())
{
    // Lógica no caso do email ser válido.
}
```

É fundamental que você acesse o recurso abaixo:

Saiba mais, acesse:

Métodos de extensão (Guia de Programação em C#)

https://msdn.microsoft.com/pt-br/library/bb383977.aspx

Como implementar e chamar um método de extensão personalizado (Guia de Programação em C#)

https://msdn.microsoft.com/pt-BR/library/bb311042.aspx

Dicas e utilidades – Métodos de extensão C#

http://ninjadevspace.com/dicas-e-utilidades-parte-1metodos-de-extensao-c/

Programação em C# - Classes - Métodos Estáticos

https://www.youtube.com/watch?v=bK75BjnwoIs&t=1s

Programação em C# - Métodos - Modificadores de Acesso

https://www.youtube.com/watch?v=RN-YEQ6d-fs&t=34s

Criando Módulos em C# (Métodos de Extensão)

https://www.youtube.com/watch?v=TSRnJMd5Yo4

Chegou a hora de você mostrar o que aprendeu. Acesse o link abaixo e faça a avaliação.

Teste Objetivo – 8A

Teste de Performance – TP4

Nessa etapa, você desenvolveu as seguintes competências e suas subcompetências:

Criar interfaces e hierarquias de classes usando herança.

- Compreender como estender classes.
- Compreender como esconder ou sobrescrever membros.
- Compreender como prevenir a herança.

Etapa 09 – O Sistema de Arquivos e os Fluxos de Dados

Fonte da imagem: Banas Qualidade

Olá {nome do aluno}.

Os tipos necessários para a leitura e escrita de dados locais no .NET Framework estão no namespace System.IO. Esses tipos permitem a leitura e escrita, tanto síncrona quanto assíncronamente, de fluxo de dados ou arquivos.

Nesta etapa, você vai aprender um pouco sobre os principais métodos de manipulação de arquivos e diretórios, e também de fluxos de dados. Também vai aprender um pouco sobre codificação de texto.

Vamos lá?!

>> Clique no link: Roteiro de aprendizagem para iniciar essa etapa.

1. Arquivos e diretórios

Um arquivo é uma coleção de bytes, que contém um nome e está ordenada, em um meio de armazenamento persistente. Quando trabalhamos com arquivos também trabalhamos com diretórios.

Podemos criar um arquivo e um diretório utilizando o trecho de código a seguir:

```
string diretorio = @"C:\MeuDiretorio\";
System.IO.Directory.CreateDirectory(diretorio);

string nomeArquivo = "MeuArquivo.txt";
string caminhoArquivo = System.IO.Path.Combine(diretorio, nomeAr
quivo);

using (var fluxoArquivo = System.IO.File.Create(caminhoArquivo))
{
    for (byte i = 0; i < 100; i++) { fluxoArquivo.WriteByte(i);
}
```

```
}
```

Observe que esse trecho de código pode gerar algumas exceções. Veja a seguir:

- Caso o nome do diretório que está sendo criado contenha caracteres ilegais ou esteja em branco um ArgumentException será gerado;
- Caso o nome do diretório seja null um ArgumentNullException será gerado;
- Caso o nome do diretório seja muito longo um PathTooLongException será gerado.

Esses são apenas alguns exemplos de exceções que podem ocorrer na criação de diretórios e arquivos.

Podemos copiar os arquivos de um diretório de origem para um diretório de destino utilizando o trecho de código a seguir:

```
var diretorioOrigem = new DirectoryInfo(@"C:\Origem\");
var arquivos = diretorioOrigem.GetFiles();

foreach (var arquivo in arquivos)
{
    var caminho = Path.Combine(@"C:\Destino\", arquivo.Name);
    arquivo.CopyTo(caminho, false);
}
```

Observe que esse trecho de código não está verificando se os diretórios origem ou destino existem antes de executar a operação de cópia. É boa prática fazê-lo.

Também podemos utilizar as classes de manipulação de arquivos e diretórios junto com o LINQ para obter uma lista de diretórios cuja criação tenha ocorrido após uma determinada data.

```
var programas = new DirectoryInfo(@"C:\Program Files\");
var dataCriacao = new DateTime(2015, 01, 01);

var diretorios = from diretorio in programas.EnumerateDirectories()
                 where diretorio.CreationTimeUtc < dataCriacao
                 select new
                 {
                     Programa = diretorio
                 };

foreach (var diretorio in diretorios)
```

```
{
    Console.WriteLine($"{diretorio.Programa.Name}");
}
```

É fundamental que você acesse os recursos abaixo:

Sistema de arquivos e o Registro Sistema de arquivos e o Registro (Guia de Programação em C#)

https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/file-system/index

Trabalhando com arquivos e diretórios em C#

http://www.linhadecodigo.com.br/artigo/3684/trabalhando-com-arquivos-e-diretorios-em-csharp.aspx

Como copiar, excluir e mover arquivos e pastas (Guia de Programação em C#)

https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/file-system/how-to-copy-delete-and-move-files-and-folders

Criação e Manipulação de Arquivos e Diretórios

https://www.youtube.com/watch?v=3QGUW5vrPnY

2. Fluxos de dados

Fluxos de dados são sequências de bytes que podem ser utilizados para leitura ou escrita em diversos meios, sejam eles disco ou memória. Existem diversos tipos de fluxos de dados para os diferentes meios, como memória, arquivo, rede, e pipes.

Podemos utilizar fluxos de dados para ler de um arquivo, por exemplo.

```
using (var fluxoLeitura = new StreamReader("MeuArquivo.txt"))
{
    Console.WriteLine(fluxoLeitura.ReadToEnd());
}
```

No exemplo acima estamos assumindo que existe um arquivo chamado MeuArquivo.txt na mesma pasta da aplicação que executa esse trecho de código.

Podemos utilizar fluxos de dados para escrever em um arquivo como no trecho de código a seguir:

```
```

```
using (var fluxoEscrita = new StreamWriter("MeuArquivo.txt"))
{
    for (var i = 1; i <= 100; i++)
    {
        fluxoEscrita.WriteLine($"Linha {i}");
    }
}
```

Para escrevermos e lermos caracteres de uma string também podemos utilizar um fluxo de dados.

```
var str = "Uma string qualquer";
var arrayStr = new char[str.Length];
using (var leitorString = new StringReader(str))
{
    leitorString.Read(arrayStr, 4, 10);
    Console.WriteLine(arrayStr);
}
```

É fundamental que você acesse os recursos abaixo:

File and Stream I/O

https://msdn.microsoft.com/en-us/library/k3352a4t.aspx

C# - StreamReader, StringReader e TextReader. Qual a diferença ? Qual usar ?

http://www.macoratti.net/14/04/c_cnio1.htm

Nessa etapa, você desenvolveu as seguintes competências e suas subcompetências:

Ler e escrever dados locais.

- Compreender como trabalhar com arquivos e diretórios.
- Compreender como trabalhar com streams.