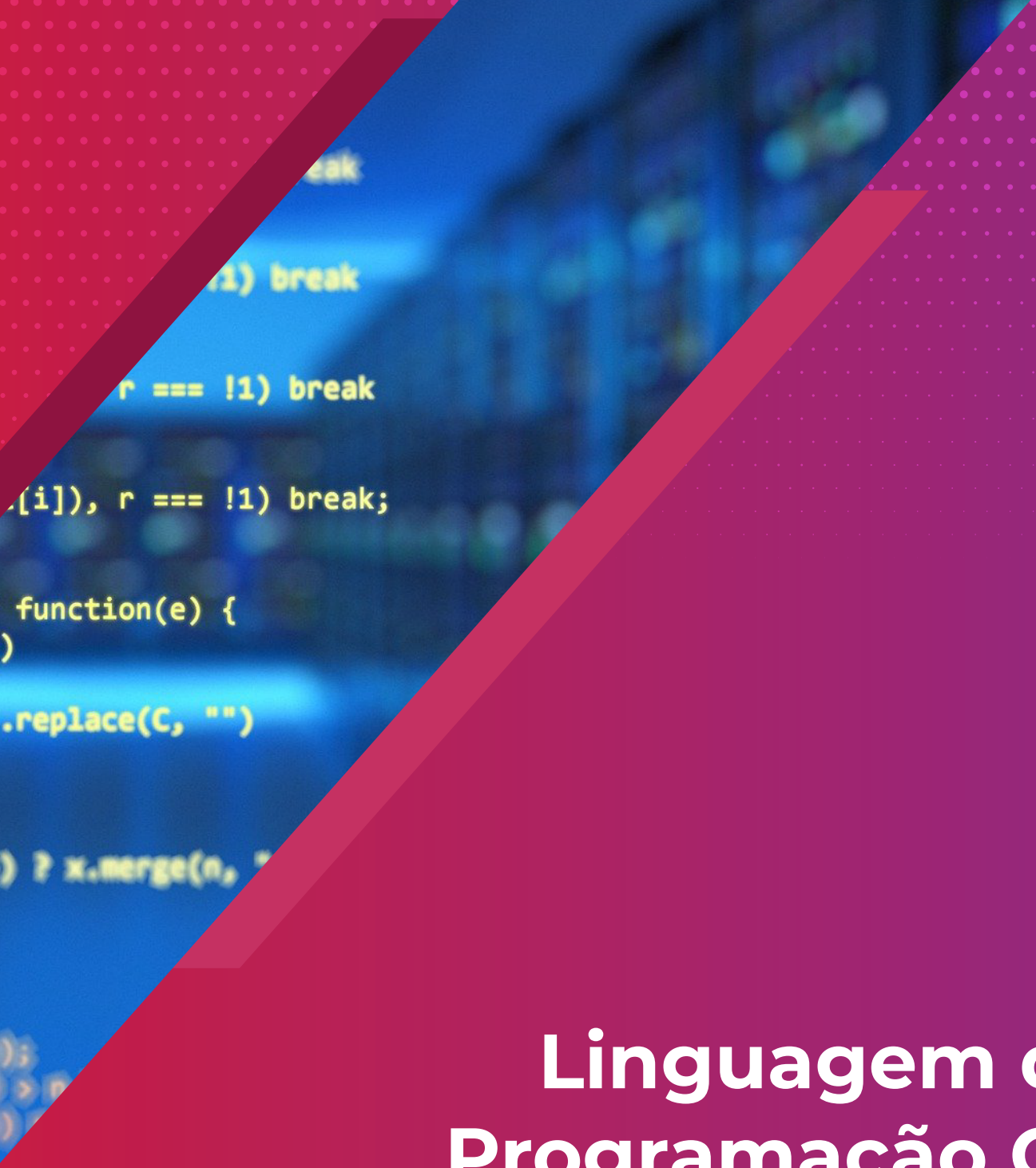


Linguagem de Programação C# Avançado

fundação bradesco | escola virtual

Sumário

Sumário

Apresentação

Bem-vindo(a) ao curso Linguagem de Programação C# - Avançado!

O objetivo deste curso é apresentar conceitos avançados sobre a utilização da linguagem de programação C#. Você verá assuntos relevantes para o seu aprendizado, como: **classes, métodos, programação orientada ao objeto, construtores, elementos estáticos, parâmetros, atributos e enumerações**.

O conteúdo é apresentado a partir de exemplos práticos para cada um dos temas citados, de modo que fique claro como as ferramentas ensinadas poderão ser utilizadas na hora da programação.

Antes de iniciar seu aprendizado, assista ao vídeo a seguir para uma introdução quanto ao que será visto ao longo do seu estudo.

Desejamos a você um bom curso!

Vídeo

Confira o vídeo de apresentação do curso.

Perdeu algum detalhe? Confira o que foi abordado no vídeo.

Olá! Bem-vindo(a) ao curso Linguagem de Programação C# - Avançado!

Aqui você verá os conceitos de temas como enumerações, parâmetros, classe e métodos. Conceitos fundamentais que o auxiliaram a utilizar a linguagem C# de modo avançado.

Além desses conceitos iniciais da etapa de linguagem C#, você também aprenderá alguns pontos interessantes sobre a programação orientada ao objeto, envolvendo assuntos bem conhecidos da área como herança, polimorfismo, classe abstrata e abstração.

E ao final, você verá sobre métodos e funções na utilização da linguagem C#.

Com isso, será possível identificar como cada conteúdo se encaixa e o levará a descobrir como esses conhecimentos farão diferença na sua vida profissional e pessoal como programador.

Vamos começar essa jornada?

Módulo 1

Conceitos básicos para aplicar a linguagem C# avançada

fundação bradesco | escola virtual

Conceitos básicos para aplicar a linguagem C# avançada

Neste módulo, você estudará sobre os conceitos fundamentais para que tenha capacidade de utilizar a linguagem C# de modo avançado. Tais conceitos são: **classes, métodos, construtores, parâmetros variáveis, enumerações, parâmetros por referência e parâmetro padrão**.

Está pronto? Então vamos lá!

Classes

O primeiro conceito que você verá é a **classe**, o qual é um bloco de construção básico na linguagem C#.

Ao criar uma classe, estabelecemos um "tipo". A partir dessa etapa, criamos uma variável.

Para que você veja na prática, utilizamos no código abaixo, o exemplo da criação de uma classe com nome "Carro" e, posteriormente, um novo objeto "C1" para essa classe.

```
01. class Carro
02. {
03. ....'Atributos'
04.
05. ....'Algoritmos'
06. }
07.
08. Carro C1 = new Carro();
```

Acompanhe o passo a passo na sequência, para entender o que foi realizado:

Passo 1

Após criar a classe, inserimos dentro dela seus atributos, ou seja, de forma mais direta, os dados como: cor, marca, nome, potência etc. Temos, também, os comportamentos, os quais receberão os dados e farão as funções desejadas.

Passo 2

A partir do comando *new*, é elaborado um novo objeto que utilizará a classe em questão, como modelo para a sua criação, fazendo uso dos seus atributos que, apesar de serem parte da classe, cada objeto terá um atributo diferente.

Simples, não é mesmo?

Agora que apresentamos como criar uma classe, vamos conhecer os métodos!

Métodos

Após estudar sobre as classes, vamos agora para o conceito de método. Para que você entenda melhor as suas características e possíveis aplicações, assista ao vídeo a seguir!

Vídeo

Confira o vídeo sobre o conceito de método.

Perdeu algum detalhe? Confira o que foi abordado no vídeo.

Olá, vamos falar um pouco sobre os métodos?

Até este ponto você conheceu o conceito de classe, o modo como ela é formulada e compreendeu como ela é usada.

Veremos agora o conceito de métodos e sua aplicação dentro do contexto da Linguagem de Programação C#.

Os métodos são responsáveis por determinar os comportamentos que serão realizados na classe. Ao criar métodos que realizarão as atividades desejadas, utilizamos sempre a primeira letra maiúscula, sendo que as entradas desse método ficarão entre parênteses.

Observe que criamos um método com o nome “Multiplicar”, o qual receberá duas variáveis do tipo *float* com nome “x” e “y”.

Veja que definimos também o tipo da variável que será retornada ao término da resolução das atividades do método, colocando antes do nome da classe.

Isto é, a variável retornada será do tipo *float*.

```
01. float Multiplicar(float x, float y)
02. {
03.
04. return x*y;
05.
06. }
```

É importante que você saiba que os métodos não precisam, obrigatoriamente, fazer o retorno de alguma variável, podendo ser de um tipo que não faz retorno, ou seja, não necessita enviar nada que será utilizado em outra função, mas poderá realizar comandos dentro do seu escopo, como resolução de equações, plotar na tela resultados, dentre outros.

Nesse caso, em vez de colocar o tipo da variável antes do nome do método, indicamos o nome "*void*" (vazio), que é uma característica de métodos que não fazem retorno de variáveis. Observe no exemplo, o método com nome "Entrada" não faz retorno.

```
01. void Entrada()
02. {
03. }
```

Com o que aprendeu até aqui, que tal agora praticar criando os seus métodos: com retorno e sem retorno de variáveis?

Com o que você aprendeu até aqui, podemos analisar um exemplo inicial para fixar melhor o conceito sobre métodos.

Vídeo

Acompanhe o que preparamos, assistindo a mais um vídeo.

Perdeu algum detalhe? Confira o que foi abordado no vídeo.

Olá, agora que você já conhece o conceito de métodos e como eles determinam os comportamentos na classe, chegou o momento de praticar. Acompanhe!

Neste código criamos uma classe com o nome “Aluno”.

Dentro dela, inserimos alguns atributos, como nome, idade e série.

Observe que, antes da determinação do tipo dos atributos, existe a palavra *public*, que faz esses atributos se tornarem visíveis para todos.

```
Class Aluno
{
public  string Nome;
public int Idade;
public int Serie;

}
```

Vamos ver mais um exemplo, neste outro código, observe que dentro do *main* é estabelecido um objeto para a classe “Aluno”.

Como explicamos anteriormente, ao criarmos um objeto, ele fará uso dos atributos da classe, mas cada objeto terá seus atributos independentes.

Note que, estabelecemos o objeto “aluno1”. Em seguida, realizamos a determinação dos seus atributos e apresentamos cada um deles.

Observe que, como resultado, teremos o nome João, idade 13 e série 6.

```
static void Main(string[] args)

{
Aluno aluno1 = new Aluno();
aluno1.Nome = "João";
alunol.Idade = 13;
alunoi.Serie - 6;
Console.WriteLine($"0 aluno {aluno1.Nome} tem {aluno1.
Idade) anos e pertence ao {aluno1.Serie} ano");

}
```

Vale destacar que outra forma de montagem do código pode ser feita com a apresentação do aluno ainda na classe. Veja o código na sequência.

```
class Aluno
{
public string Nome;
public int Idade;
public int Serie;

1 referência
public string Retorno()
{
return string.Format($"O aluno {Nome} tem {Idade}
anos e pertence ao {Serie} ano");
}

1 referência
public void RetornoConsole()
{
Console.WriteLine(Retorno());
}
```

Veja que aqui foi criado um método com nome “Retorno”, o qual retornará à apresentação do aluno na classe.

Perceba que esse método não necessita ter como entrada os parâmetros do aluno, pois já tem acesso aos atributos. Observe que o resultado será: O aluno João tem 13 anos e pertence ao 6º ano.

Criamos, ainda, outro método com nome “RetornoConsole”, que fará a apresentação na tela.

Já no *main*, é necessário apenas fazer a chamada do método “RetornoConsole”.

```
class Program
{
0 referências
static void Main(string[] args)
{
Aluno aluno1 = new Aluno();
aluno1. Nome = “João”;
aluno1.Idade = 13;
aluno1. Serie = 6;

aluno1. RetornoConsole();

}
```

Agora é com você, tente recriar os exemplos apresentados aqui e veja os resultados.

Após assistir aos vídeos, não podemos deixar de inserir um elemento importante nesse contexto: **os construtores**, assunto que você conhecerá melhor no próximo tópico.

Construtores

Quando nos referimos a **construtores**, analisamos um método que faz um laço entre classe e objeto. Observe o código abaixo que traz o funcionamento do construtor.

```
class Carro
{
public string Nome;
public string Marca;
public double Potencia;
1 referência
public Carro(string nome, string marca, double potencia)
{
Nome = nome;
Marca = marca;
Potencia = potencia;
}
1 referência
public Carro()
{
}
}
0 referências


class Construtor
{

0 referências
static void Main(string[] args)
{
var carro = new Carro();
carro.Nome = "Hilux";
carro.Marca = "Toyota";
carro.Potencia = 3.1;
Console.WriteLine($"{ carro.Nome} { carro.Marca} { carro.
Potencia}");
var carro2 = new Carro("Celta","Chevrolet",1.0);
Console.WriteLine($"{carro2.Nome} {carro2.Marca} {carro2.
Potencia}");
}
```

Fizemos a criação de um objeto por método tradicional, realizando a determinação dos parâmetros um a um, assim como está no início do nosso *main*, mas considerando a criação do construtor "Carro" para receber os parâmetros. Nesse caso, os atributos do construtor iniciam com letra minúscula, tornando-o diferente dos atributos já determinados anteriormente.

Portanto, é possível compreender que o segundo carro é chamado pelo construtor. Então, temos duas formas de trabalhar, sendo que o construtor fez exatamente o papel esperado: a ligação entre classe e objeto.

Para dar continuidade aos seus estudos, no próximo tópico, você verá sobre os elementos estáticos.

Elementos estáticos

Você conhece os elementos estáticos? Sabe como eles funcionam? Acompanhe o slide na sequência para entender como são aplicados!

O conceito de **estático** se aplica tanto aos métodos quanto aos atributos. Ao analisarmos os exemplos dos códigos utilizados até aqui, percebemos a presença da palavra *static*, que representa os elementos estáticos.

#PraCegoVer

Um homem e uma mulher estão em frente a uma tela de um computador. Ele está à direita com uma camisa azul apontando para tela e ela está à esquerda com uma camisa preta digitando, sendo que ambos estão comentando sobre códigos de programação.

#PraCegoVer

As mãos de um programador, segurando uma caneta, interagindo com a tela de um notebook que contém linhas de código de programação.

O fato de termos um método estático nos faz com que seja possível obter o acesso diretamente, sem precisarmos criar uma nova instância, ou seja, não será necessária a criação de uma instância para a chamativa da classe.

Verifique no código a seguir que conseguimos chamar a classe “MetodosEstaticos” no *main*, para a realização da multiplicação sem a necessidade de termos que criar um objeto.

Elementos estáticos

```
class MetodosEstaticos
{
1 referência
public static int somar (int x, int y)
{
return x + y;
}
1 referência
public static int multiplicar (int x, int y)
{
return x * y;
}
0 referências
static void Main(string[] args)
{
var result = MetodosEstaticos.somar (3, 3);
Console.WriteLine("A soma é igual a: {0}", result);

Console.WriteLine(MetodosEstaticos.multiplicar(3, 4));
}
```

Agora, observe no próximo código que, se excluirmos a determinação *static* do método multiplicar, será necessária a criação de uma instância. Nesse caso, devemos criar *calc* para ter acesso ao método "multiplicar".

Criando uma instância

```
class MetodosEstaticos
{
1 referência
public static int somar (int x, int y)
{
return x + y;
}
1 referência
public static int multiplicar (int x, int y)
{
return x * y;
}
0 referências
static void Main(string[] args)
{
var result = MetodosEstaticos.somar (3, 3);
Console.WriteLine("A soma é igual a: {0}", result);

MetodosEstaticos cal = new MetodosEstaticos();
Console.WriteLine(MetodosEstaticos.multiplicar(3, 4));
}
```

O mesmo conceito se aplica aos atributos estáticos, em que, caso seja atribuída a característica de estático ao atributo, este passará a se referir à classe como um todo, deixando de ser uma instância específica.

Depois de você entender os conceitos de elementos estáticos, principalmente, em relação à sua capacidade de dispensar a criação de uma instância para a chamativa da classe, você estudará no próximo tópico sobre os parâmetros variáveis.

Parâmetros variáveis

No contexto da Linguagem C#, quando estamos elaborando uma linha de código, existem os **parâmetros**, isto é, elementos que passam a obter um valor pelos programadores, os quais podem detalhar os parâmetros separadamente para cada método diferente, no caso, os **argumentos**, ou seja, o valor atribuído a um parâmetro no momento em que se chama o procedimento.

Já os **parâmetros variáveis** trazem a possibilidade de passarmos quantos parâmetros desejarmos, e a linguagem agrupará essas variáveis.

Veja no código abaixo que utilizamos a função da linguagem C#, denominada de *params*, para estabelecermos a criação desse parâmetro variável. Além disso, incluímos o *foreach* com o objetivo de realizar a varredura enquanto houver nomes disponíveis para expor na tela.

Parâmetros variáveis

```
public static void Despedida (params string [] alunos)
{
foreach (var aluno in alunos)
{
Console.WriteLine("Tchau {0}, até a próxima aula", aluno);
}
}

0 referências
static void Main()
{
Despedida("Joao", "Thiago", "Antonio");
}
```

Nesse sentido, enviamos para o método “Despedida” mais de uma variável, sendo que todas foram agrupadas em “alunos”.

Resultado do algoritmo

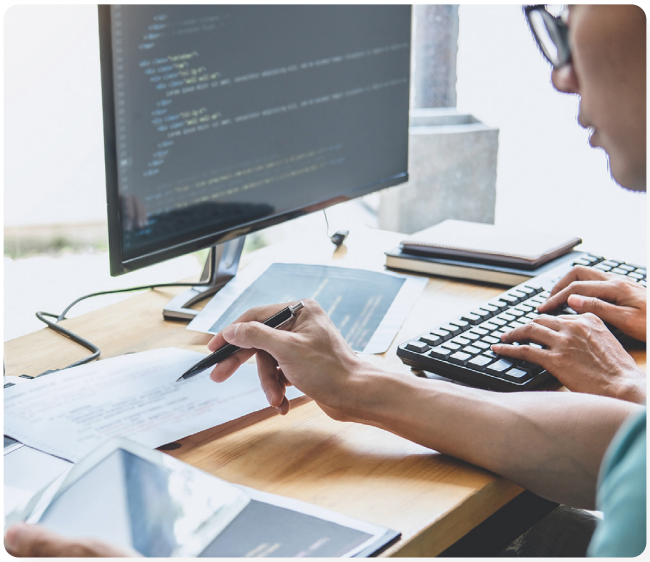
O resultado desse algoritmo é mostrado abaixo. Observe que todas as variáveis criadas no *main* foram apresentadas na tela.

```
Tchau Joao, até a próxima aula
Tchau Thiago, até a próxima aula
Tchau Antonio, até a próxima aula
```

Além dos parâmetros variáveis, ainda temos os tipos *get* e *set*, que você estudará na sequência. Vamos lá!

Get e set

Até esse ponto você aprendeu que a criação de atributos do tipo *public* faz com que eles sejam visíveis para todos. O contrário também é válido, ou seja, se criarmos atributos do tipo *private*, eles serão visíveis apenas para o método.



#PraCegoVer
Mesa de trabalho com uma tela de notebook que duas pessoas estão em uma mesa com seus computadores executando linhas de programação, um dos usuário está com um notebook e outro está digitando em um computador de mesa.

- A vantagem de utilização de atributos do tipo *private* é evitar que sejam modificados. Para termos acesso a eles, devem ser criados métodos.
- Deve ser estabelecido um atributo para alterar o tipo *set*, enquanto o método *get* nos ajudará na leitura do valor do atributo. Assim, cada atributo terá um método *get* e um método *set*.

No código a seguir, note como o termo é feito para a utilização dos métodos *get* e *set*, conforme você estudou até o momento.

Método *get* e *set*

```
public class Carro
{
private string Marca;
private string Nome;
private double Potencia;

1 referência public Carro(string marca, string nome, double
potencia)
{
Marca - marca;
Nome = nome;
Potencia - potencia;
}
0 referências
public Carro()
{

}
1 referencia
public string GetMarca()
{
return Marca;
}
0 referências
public void SetMarca(string marca)
{
Marca = marca;
}
1 referência
public string GetNome()
{
return Nome;
}
```

Perceba que foi criada a classe "Carro", com os atributos do tipo *private* — que serão acessados pelos métodos *get* e *set* — e, posteriormente, foi criado o construtor. No código, demonstramos a criação dos métodos *get* e *set* para o atributo "Marca", mas o procedimento é realizado para os três atributos.

Neste tópico, você estudou sobre um tipo específico de parâmetro variável, o *get* e *set*, cuja função é atribuir a característica "*private*" a um método específico. Já a seguir, estudaremos as enumerações, *enums*, recurso usado para as variáveis incomuns presentes na linguagem C#.

Enumerações

Em determinados momentos, é necessário realizarmos a representação de alguma variável que não se encaixa nos tipos comuns, como *string* ou *int*. Dessa forma, criamos os ***enums***. Como a própria abreviação nos revela, serve para indicar enumerações de atributos. Observe no código!

Enumeração

```
class Enum
{
2 referências
public enum Genero { Filme, Serie, Documentario};

0 referências
public class Filme
{
public string Nome;
public Genero TipoVideo;
}

0 referências
static void Main()
{
int cod = (int)Genero.Serie;
Console.WriteLine(cod);

}
```

Note que ele apresenta um exemplo de enumeração realizada para a designação de um vídeo. É importante observar que, no *main*, é feita a conversão de *string* para *int*. O valor que retornará para a tela será a posição do tipo “serie”, tendo valor igual a “1”, uma vez que a enumeração começa a contar de “0”.

Depois de estudar sobre as enumerações, as quais estão relacionadas a variáveis incomuns, você verá os conceitos de *structs* e classes.

Structs e classes

Antes de continuarmos nossos estudos, é importante reforçarmos a diferença entre dois conceitos: *struct* e classe.

Isso se justifica, porque ambos influenciam na elaboração das linhas de código, principalmente, quando você, futuro programador, estiver trabalhando com a criação de estrutura de dados.

Sabendo disso, agora, vamos entender a principal diferença entre *struct* e classe!

Struct
A atribuição é sempre feita por valor, e não por referência.
Classe
A atribuição é realizada por referência.

No código a seguir, foram criadas uma *struct* e uma classe com os mesmos parâmetros, a fim de verificarmos na prática essa diferença. Observe com atenção!

Struct e classe (comparação)

```
class StructClasse
{
2 referencias
public struct PontoS
{
public int X;
public int Y;
}
2 referências
public class Pontoc
{
public int X;
public int Y;
}
0 referencias
public static void Main()
{
PontoS ponto = new Pontos { X = 5, Y = 3 };
PontoS ponto2 = ponto; // Copiar através do valor;
ponto.X = 3;

Console.WriteLine("Ponto: {0}", ponto.x);
Console.WriteLine("Ponto 2: {0} ", ponto2.x);

Pontoc ponto3 = new Pontoc { x = 6, Y = 9 };
Pontoc ponto4 = ponto3;
ponto3.X = 3;

Console.WriteLine("Ponto 3 = {0} , Ponto 4 - {1}", ponto3.x,
ponto4.x);
}
```

Note que, ao fazer a resolução do algoritmo, no caso da *struct*, os valores não são iguais, ou seja, mesmo que tenhamos criado uma cópia do ponto, ao modificar um, não se observará a mesma modificação no outro. Esse fato, por outro lado, já não é observado para o caso da utilização da classe.

Até aqui, você conheceu a diferença entre *struct* e classe na linguagem C#. Conforme falamos, saber a distinção entre ambos é relevante, uma vez que isso influencia na criação de linhas de códigos as quais possibilitam a elaboração e o uso de estruturas de dados mais eficazes e eficientes.

A seguir, você verá sobre a maneira pela qual é possível aplicar a referência dentro do contexto de parâmetros.

Vamos lá?

Parâmetros por referência

Neste tópico, veremos como utilizar uma **referência** ou cópia de um valor, o que dependerá da maneira como você deseja criar a chamada da variável.

Observe o código com atenção.

```
1 referência
public static void AlterarRef (int numero)
{
numero = numero + 500;
}
1 referência
public static void AlterarOut(int numero)
{
numero = numero + 50;
}

0 referências
public static void Main()
{
int x = 5;
AlterarRef(x);
Console.WriteLine(x);

int y = 8;
AlterarOut(y);
Console.WriteLine(y);

}
```

Note que os valores que utilizamos ao fazer a chamativa dos métodos faz uma cópia das variáveis “x” e “y”. Com isso, ela não será alterada. Logo, os valores impressos na tela serão “5” e “8”, pois não foi transmitida a referência, mas sim, a cópia da variável.

```
public static void AlterarRef (ref int numero)
{
numero = numero + 500;
}
1 referência public static void AlterarOut(int numero)
{
numero = 0;
numero = numero + 50;
}
O referências
public static void Main()
{
int x = 5;
AlterarRef(ref x);
Console.WriteLine(x);

int y = 8;
AlterarOut(y);
Console.WriteLine(y);

}
```

Observe, ainda, que existe a possibilidade de fazermos a passagem das variáveis como referências. Verifique que a variável “x” está sendo passada como referência, portanto, os valores que serão impressos na tela ao compilar o algoritmo serão “505” e “8”.

Assim como temos o parâmetro por referência, também existe o parâmetro padrão, que você estudará a seguir.

Parâmetro padrão

O **parâmetro padrão**, como o próprio nome indica, está relacionado ao padrão que será atribuído para determinada variável, caso nenhum valor seja concedido a ela. Vamos analisar um exemplo?

Veja o caso a seguir:

```
01. public static double Multiplica(double x = 3.4 , double y
= 3){
02. return x * y;
03. }
```

Nesse exemplo, foram inseridos valores padrões para as variáveis “x” e “y”, os quais são iguais a “3,4” e “3”, respectivamente. Esses valores podem ser modificados em algum momento, mas, caso não haja alterações durante o decorrer do código, o valor padrão será mantido.

Atenção

Lembre-se! A utilização desses valores padrões não se aplica para o uso de “*ref*” ou “*out*”.

Parabéns! Você chegou ao final do módulo 1.

Neste módulo, você conheceu os conceitos fundamentais para a aplicação da linguagem C#, que são relevantes para o seu futuro profissional, como, por exemplo, na programação orientada a objetos, a qual é muito usada em situações de desenvolvimentos de softwares personalizados de acordo com as futuras demandas.

A respeito desses conceitos iniciais, vamos estudar um pouco mais sobre a programação orientada a objeto? Será o assunto do próximo módulo.

Módulo 2

Programação orientada a objeto

fundação bradesco | escola virtual

Programação orientada a objeto

Neste módulo, serão apresentados os conceitos relacionados à linguagem C# ligados à programação orientada a objetos, a qual é muito importante quando você estiver atuando como programador, sobretudo, ao criar softwares personalizados.

Partindo disso, você estudará sobre **abstração e herança, polimorfismo, e encapsulamento, classe abstrata e classe *sealed***.

Vamos começar?!

Abstração e herança

Para iniciar seus estudos sobre abstração e herança, ouça nosso *podcast* para obter maiores informações quanto aos conceitos.

Podcast

Confira o ***podcast*** sobre abstração e herança.

Perdeu algum detalhe? Confira o que foi abordado no *podcast*.

Olá! Tudo bem? Vamos falar um pouco sobre a **Abstração** e a **Herança**? Embora pareçam dois conceitos fáceis de entender, na prática eles podem exigir certa atenção da nossa parte para não haver confusão.

Por isso, é preciso ter cuidado no momento de aplicar cada um deles na construção da linguagem de programação.

O conceito de abstração se refere ao processo de abstrair a ideia do projeto e trazer ela para o código.

Para deixar mais fácil a explicação sobre a abstração, imagine um sistema de registro de todas as escolas de um determinado estado, onde, desse sistema, você precisa abstrair as características necessárias de cada escola e trazer para o código. Como por exemplo: nome, quantidade de funcionários, verbas enviadas para a escola, entre outros aspectos.

Já a herança é uma característica relacionada com a possibilidade de fazer a reutilização de códigos, mas atenção, devemos sempre evitar o autoplágio.

De maneira prática, podemos dizer que a Herança é a possibilidade de uma classe herdar os atributos de outra, que pode ser chamada de superclasse.

Aliás, uma característica da linguagem C# é que uma classe só pode ter herança de outra classe.

Assim fica mais prático de compreender e entender a diferença entre a Abstração e a Herança, não é mesmo?

Observe no código, o conceito inicial do processo de utilização de herança de determinada classe.

Herança (parte 1)

```
public class Professor
{

protected readonly float salariomaximo;
float salarioatual;

1 referência
public Professor(float salarioMaximo)
{
salariomaximo = salarioMaximo;
}

2 referências
protected float Modificarsalario(float Modf)
{
float novoSalario = salarioatual + Modf;

if (novoSalario < 0)
{
salarioatual = 0;
}
else if (novoSalario > salariomaximo)
{
salarioatual = salariomaximo;
}
else
salarioatual = novoSalario;
}
return salarioatual;
}

2 referências
public float AumentaSalario()
{
return Modificarsalario(+500);
}

0 referências
public float DiminuiSalario()
{
return Modificarsalario(-500);
}
```

Esse projeto consiste em um algoritmo para controle de salário dos professores. Note que, inicialmente, foi criada a variável "salariomaximo" com característica "*readonly*", ou seja, que não pode ser modificada. Já o "salarioatual" serve para fazer a atualização do valor do salário do professor.

Observe os pontos principais quanto ao que foi realizado no projeto mencionado. Vamos conhecê-los!

▼ Ponto 1

Foi realizada a formação de um construtor, que recebe o valor do salário máximo.

▼ Ponto 2

Criou-se um método responsável por fazer a modificação do salário.

▼ Ponto 3

Foi acrescentada uma espécie de saturador, em que, determinando-se um teto máximo e um valor mínimo para o salário, este não poderá ser negativo, nem ultrapassar o teto máximo.

Além de tudo isso, foram criados os métodos responsáveis por fazer o aumento ou a diminuição do salário, intitulados "AumentaSalario" e "DiminuiSalario", respectivamente.

Agora, veja mais uma parte do código:

Herança (parte 2)

```
public class Prof : Professor
{

1 referência
public Prof() : base (1400)
{
0 referências
public static void Main()
{
Console.WriteLine("Professor 1: ");
Prof profi = new Prof();
Console.WriteLine (prof1. AumentaSalario());
Console.WriteLine(prof1. AumentaSalario());

}
```

Note que foi criada uma classe “Prof” que herdará os métodos da superclasse “Professor”. Posteriormente, no *main*, criamos um objeto para a classe que herdou os métodos. Em seguida, utilizamos os métodos herdados com o objeto criado.

Na prática, os conceitos de abstração e herança ficam mais fáceis de serem entendidos, não é mesmo? Assim como a técnica utilizada, que o ajudará muito no desenvolvimento de seus projetos. Com o objetivo de complementar seus estudos, no próximo tópico apresentaremos o polimorfismo e o encapsulamento. Confira!

Polimorfismo e encapsulamento

O Polimorfismo e o Encapsulamento almejam tornar o futuro software não só mais flexível, como também facilita a sua modificação e criação de novas implementações.

Iniciando pelo Polimorfismo, pode-se dizer que ele se subdivide em estático ou dinâmico. Vamos entender a diferença entre os dois e ainda conhecer a definição do que é o encapsulamento, muito utilizado também no dia a dia da profissão!

▼ Polimorfismo estático

Observamos dois métodos com o mesmo nome dentro de uma mesma classe.

▼ Polimorfismo dinâmico

Consiste em sistemas que contêm classes com heranças, fazendo a instância de objetos a partir da superclasse.

▼ Encapsulamento

Consiste na possibilidade de "esconder" parte do código para evitar o acesso de outra pessoa que não seja o programador, ficando até mais fácil realizar modificações no código sem causar problemas.

Temos disponíveis quatro tipos de visibilidade. No caso do *public*, está relacionado ao que pode ser visto e acessado por todos. Já *internal* é aquilo que pode ser acessado apenas dentro do projeto. O *protected* está ligado ao que é passado por herança. Por fim, o *private* é aquilo que pode ser visto apenas pela classe.

Para entender o conceito de encapsulamento na prática, observe na parte 1 do código abaixo, a criação de uma variável do tipo private, a qual será acessada apenas pela classe responsável. Utilizaremos, também, os métodos get e set estudados anteriormente, demonstrando uma relação entre os conteúdos.

Encapsulamento (parte 1)

```
public class Mensagem
{
private String Texto;

1 referência
public void Exibir()
{
Console.WriteLine(this.Texto);
}

0 referências
public String getTexto()
{
return this.Texto;
}
1 referência
public void setTexto(String txt)
{
this. Texto = txt;
}
```

Na parte 2 do código, note que o acesso aos métodos da classe é realizado a partir de objetos criados nessa mesma classe.

Encapsulamento (parte 2)

```
static void Main(string [] args)
{
Mensagem txt1, txt2;

txt1 = new Mensagem();
txt1.setTexto("CURSO AVANÇADO DE C#");
txt1. Exibir();

}
```

Após estudar sobre o **Polimorfismo estático e dinâmico**, bem como o **Encapsulamento** a partir de exemplos, você pode notar o quanto eles facilitam as possíveis modificações do software que será desenvolvido por você, como também o tornam mais eficaz no momento do uso.

Todavia, aliados a esses dois conceitos, você verá no próximo tópico sobre as classes abstrata e *sealed*.

Classes abstrata e *sealed*

Conforme você estudou, a classe é um bloco de construção básico na linguagem C#. Ela pode ser do tipo **abstrata** somente se houver outra classe que a herde, uma vez que esse tipo não pode ser instanciado. Entenda melhor com os conceitos abaixo!

#PraCegoVer

Mesa de trabalho com uma tela de notebook que contém vários códigos de programação rodeada por vasos de plantas.

- A classe abstrata deve ser criada caso o programador tenha a intenção de elaborar outras classes que herdem os métodos da primeira.
- A classe abstrata pode ter todos os seus métodos abstratos.

No código a seguir, veja na prática a criação de uma classe abstrata com o nome “Telefone” e a criação, logo na sequência, de uma classe de nome “Xiaomi”, que herda os métodos da classe “Telefone”. No caso, a configuração *override* é utilizada quando desejamos sobrescrever um método.

Classe abstrata

```
public abstract class Telefone
{
1 referência
public abstract string Tipo();
}

0 referências
public class Xiaomi : Telefone
{
1 referência
public override string Tipo()
{
return "Note 8 Pro";
}
```

Já a classe do tipo ***sealed*** é utilizada quando desejamos que seja proibido haver herança. A configuração é realizada conforme o exemplo a seguir, fazendo com que a classe "Escola" esteja selada.

```
01. sealed class Escola {
02. }
```

Parabéns! Você chegou ao final do módulo 2.

Nele, você estudou sobre os conceitos de **abstração e herança, polimorfismo, encapsulamento, classe abstrata e classe *sealed***.

Agora, para que seus estudos fiquem completos, vamos aprender sobre expressões?

Na programação, elas são muito relevantes, porque auxiliam na construção das operações, viabilizando a aplicação da Linguagem C#, contribuindo diretamente para o seu futuro profissional.

Desta forma, acompanhe o último módulo a seguir.

Módulo 3

Expressões

Expressões

Nesse último módulo, você aprenderá um conjunto de expressões essenciais para a aplicação da linguagem C#. Nesse sentido, vamos conhecer os métodos e funções dentro do uso dessa linguagem, envolvendo o estudo de LAMBDA, *delegates*, LINQ e extensões. Vamos lá!

LAMBDA

Considerando a linguagem LAMBDA, iremos analisar as suas funções principais: a ***action*** e ***func***. Você sabe qual é a diferença entre elas? Observe!

Action
Tipo de função que não tem retorno.
Func
Tipo de função que apresenta retorno.

A seguir, temos um exemplo desses tipos de funções na prática:

Funções

```
static void Main(string[] args)
{
Action apresentaConsole = () =>
{
Console.WriteLine("Funções e Métodos ");
};

apresentaConsole();

Func<int> sorteio = () =>
{
Random random = new Random();
return random.Next(1, 101);
};

Console.WriteLine(sorteio());

}
```

Note que, inicialmente, com a criação de uma *action*, entre parênteses, ficarão os parâmetros que são desejados passar para a função. A continuação da *action* pode ser compreendida melhor ao analisar as seguintes informações:

Utilizamos o símbolo **=>** para indicar que, após os parâmetros, virá o corpo da função. Esse símbolo é chamado de ***arrow***.

No corpo, como não há retorno, apresentamos na tela uma frase. Em seguida, é feita a chamada dessa função.

#PraCegoVer

Tela aproximada de um computador com parte de um código de programação desfocado.

Posteriormente, foi criada a função do tipo *func* para realizar o sorteio de um número. O retorno dessa função é do tipo *int*. Na sequência, utilizamos a função *random*, que sorteará um número de 1 a 100. Por fim, o número sorteado é apresentado na tela.

Desta forma, você pôde compreender mais sobre a relevância da linguagem LAMBDA, sobretudo associada aos conceitos de *action* e *func*, os quais são essenciais para operações randômicas nas linhas de código.

Delegate

Agora, falaremos sobre a função *delegate*. Ela se caracteriza como uma referência para um ou mais métodos, a qual é utilizada para a comunicação entre objetos de forma flexível e extensível. Observe o exemplo abaixo onde é criado um *delegate* no código.

Delegate

```
namespace Delegate
{

delegate double BinaryNumericOperation (double ni, double);
O referências
class Program
{

1 referência
class Calculadora
{

0 referências
public static double Max(double X, double Y)
{
if (X > Y)
{
return X;
}
else
{
return Y;
}
}
1 referência
public static double soma(double X, double Y)
{
return X + Y;
}
1 referências
static void Main(string[] args)
{
double a = 10;
double b = 30;

BinaryNumericOperation operacao = Calculadora. soma;
Console.WriteLine(operacao(a,b));
}
}
```

Nesse exemplo, é possível verificar que, dentro do "*namespace*" e antes da classe, foi criado o *delegate* como uma operação numérica binária. Isto é, será referência para uma operação que receberá dois números do tipo *double* (n1, n2).

Depois disso, dentro do *main* do programa, declara-se um objeto que receberá a classe calculadora com o método de soma, sendo possível fazer a chamada apenas com o "op" enviando as variáveis desejadas para a realização da operação. Lembramos de que é importante fazer o teste realizando o exemplo e as modificações para verificar o funcionamento.

Como pode ver, a função *delegates* auxilia na interpretação do código criado, uma vez que o torna menos confuso, facilitando, assim, a sua leitura.

LINQ

Por fim, você estudará sobre a última função, o *Language Integrated Query* (LINQ). Ele pode ser traduzido como uma consulta integrada à linguagem, diz respeito a tecnologias baseadas na integração de funcionalidade, consultando diretamente a linguagem C#.

> No LINQ, as operações são chamadas diretamente a partir das coleções. Em caso de consulta ao banco de dados, o compilador irá nos auxiliar a escrever corretamente o que precisarmos.

Assista ao vídeo, que traz um exemplo de LINQ de forma bem detalhada e explicativa, a fim de não restarem dúvidas!

Vídeo

Confira o vídeo sobre um exemplo de LINQ de forma bem detalhada.

Perdeu algum detalhe? Confira o que foi abordado no vídeo.

Olá!

Para criar a função LINQ (do inglês *Language Integrated Query*) primeiramente você deve criar uma fonte com os dados.

Observe que o LINQ é inserido no código utilizando o *using* logo no início do algoritmo. O código é utilizado para encontrar os números pares da fonte em que foi criada, fornecendo parâmetros.

Nesse caso, é criada a fonte "números" com alguns algarismos numéricos.

```
using System;
using System.Linq;

namespace Ling
{

0 referências
class Program
{
0 referências
static void Main(String[] args) {
int[] numeros = new int[] { 4, 5, 7, 8 };

var operacao = numeros. Where(x => x % 2 == 0);

foreach (int x in operacao)
{
Console.WriteLine(x);
}
}
}
}
```

Depois disso, é realizada uma operação para buscar os números que, divididos por dois, tenham resto igual à zero. Em outras palavras, procuramos os números que sejam divisíveis por dois, os quais são armazenados na variável "x". Em seguida, utilizamos o *foreach* para que cada valor de "x" seja plotado na tela.

Agora é com você! Pratique estes exemplos e confirme se o resultado que surgirá na tela são os números 4 e 8

Agora que você conheceu sobre o *Language Integrated Query* (LINQ), podemos seguir em frente. Na sequência, focaremos nas extensões e em sua funcionalidade.

Extensões

O objetivo das **extensões** é adicionar possibilidades a métodos que já existiam, ou seja, criarmos uma forma a partir daquela já existente. Para que você compreenda melhor o tema, analise o exemplo apresentado no código a seguir.

Extensão

```
namespace Extensao
{

O referências
public static class calculadoraExtensao
{
2 referências
public static double Soma (this double ni, double n2){
return nl + n2;
}
}

O referências
class Program
{
O referências
static void Main(String[] args) {

double x = 3;

Console.WriteLine(x.Soma (7));
Console.WriteLine(9.2. Soma (7));

}
}
}
```

Vamos entender o que foi feito nesse exemplo? Observe as informações para saber!

▼ Passo 1

Inicialmente, foi estabelecida uma classe para representar a extensão (com nome de calculadora) e adicionado um método de soma nessa classe para realizar a operação citada entre dois números.

▼ Passo 2

Perceba que, no parâmetro, foi inserido o termo ***this***, o qual representa a extensão. No *main* do nosso algoritmo, realizamos tipos como se fossem funções do próprio C#. Além disso, podemos utilizar números literais, como é o caso do valor “9.2”.

▼ Passo 3

Por fim, acrescentamos novos métodos para o cálculo da nossa calculadora (como subtração e multiplicação), a fim de verificar o funcionamento. O resultado ao compilar sugere “10” e “16,2”.

Assim fica mais fácil identificarmos o que se trata uma extensão, não é? Desse modo, você poderá colocar seus conhecimentos adquiridos em prática de maneira tranquila e sem grandes problemas!

Fechamento

Parabéns!

Você finalizou o curso avançado de C#. Esperamos que tenha aprimorado seus conhecimentos a respeito da linguagem.

Agora é com você, procure aplicar os aprendizados adquiridos. Observe que a linguagem C# apresenta uma quantidade vasta de ferramentas disponíveis para o desenvolvimento de algoritmos. Lembre-se sempre de buscar novas ferramentas e manter-se atualizado.

Até a próxima!

Referências

DIMES, T. **Programação em C# para iniciantes**. [*S. l.*]: Babelcube Inc., 2016. v. 3.

LIMA, E.; REIS, E. **C# e .Net para desenvolvedores**. Rio de Janeiro: Campus, 2002. Disponível em: http://www.etelg.com.br/paginaete/downloads/informatica/apostila2.pdf. Acesso em: 16 abr. 2021.

SAADE, J. **C# - guia do programador**. São Paulo: Novatec, 2011.

SILVA, L. F. da. **Desenvolvimento de software II C# programação em camadas**. Joinville: Clube de Autores, 2015.

TAVARES, N. S.; DIONYSIO, R. C. C.; SANTOS JR., C. I. dos. **C#**: introdução a programação orientada a objetos. Joinville: Clube de Autores, 2009. v. 1.

FREEPIK. **Fundo de tecnologia abstrata de código de programação do desenvolvedor de software e script de computador Foto Premium**. Freepik 2021. Disponível em: https://br.freepik.com/fotos-premium/fundo-de-tecnologia-abstrata-de-codigo-de-programacao-do-desenvolvedor-de-software-e-script-de-computador_2109654.htm#page=1&query=programa%C3%A7%C3%A3o&position=40. Acesso em 30 de junho de 2021.

_____. **Programador em desenvolvimento team development website design e codificação de tecnologias trabalhando no escritório da empresa de software Foto Premium**. Freepik 2021. Disponível em:https://br.freepik.com/fotos-premium/programador-em-desenvolvimento-team-development-website-design-e-codificacao-de-tecnologias-trabalhando-no-escritorio-da-empresa-de-software_5429218.htm#page=2&query=programa%C3%A7%C3%A3o&position=44. Acesso em 30 de junho de 2021.

_____. **Desenvolvedor de software codificando javascript no laptop Foto Premium**. Freepik 2021. Disponível em: https://br.freepik.com/fotos-premium/desenvolvedor-de-software-codificando-javascript-no-laptop_13486390.htm#query=programa%C3%A7%C3%A3o&position=25. Acesso em 30 de junho de 2021.

_____. **Escrevendo códigos e digitando tecnologia de código de dados, programador colaborando trabalhando em projeto de web site em desenvolvimento de software em computador desktop da empresa, programação com html, php e javascript. Foto Premium**. Freepik 2021. Disponível em: https://br.freepik.com/fotos-premium/escrevendo-codigos-e-digitando-tecnologia-de-codigo-de-dados-programador-colaborando-trabalhando-em-projeto-de-web-site-em-desenvolvimento-de-software-em-computador-desktop-da-empresa-programacao-com-html-php-e-javascript_12951900.htm#page=2&query=programming++coding++developer&position=28. Acesso em 30 de junho de 2021.

_____. **Código html na tela do laptop, plantas verdes na mesa, escritório aconchegante Foto Premium**. Freepik 2021. Disponível em: https://br.freepik.com/fotos-premium/codigo-html-na-tela-do-laptop-plantas-verdes-na-mesa-escritorio-aconchegante_12586698.htm#page=2&query=programming++coding++developer&position=45. Acesso em 30 de junho de 2021.

_____.**Código javascript em um monitor Foto Premium**.Freepik2021.Disponívelem: https://br.freepik.com/fotos-premium/codigo-javascript-em-um-monitor_8406217.htm#page=7&query=programming+coding+developer&position=37. Acesso em 30 de junho de 2021.

_____. **Código php em um monitor Foto Premium**. Freepik 2021. Disponível em: https://br.freepik.com/fotos-premium/codigo-php-em-um-monitor_8406218.htm#page=7&query=programming+coding+developer&position=38. Acesso em 30 de junho de 2021.

_____. **Freepik.**Recursos gráficos para todos. Freepik, 2021. Disponível em: https://br.freepik.com/. Acesso em: 21 de out. de 2021

fundação bradesco | escola virtual