



---

## Fundamentos de C#

Bootcamp Speed Wiz Dev

---

Samuel Alves dos Santos

2021

## **Fundamentos de C#**

### **Bootcamp Speed Wiz Dev**

Samuel Alves dos Santos

© Copyright do Instituto de Gestão e Tecnologia da Informação.

Todos os direitos reservados.

## Sumário

---

Capítulo 1.	O Que é C# e .NET .....	6
	Contextualizando C# e .NET .....	6
	Máquina Virtual .....	7
Capítulo 2.	O Ambiente de Desenvolvimento do C# .....	8
	Ambientes de Desenvolvimento Integrado (IDE) .....	8
	O Primeiro Programa em C# .....	9
Capítulo 3.	Variáveis e Tipos Primitivos .....	14
	Tipos Primitivos .....	15
	Tipos de Dados .....	19
	Enum (Enumeradores) .....	20
	Struct (Estruturas) .....	20
	Inferência de Tipos .....	22
	Operações com variáveis .....	23
	Variáveis do Tipo String .....	26
	Interpolação de Cadeia de Caracteres .....	26
	Comentários .....	27
Capítulo 4.	Estruturas de Controle .....	28
	If...Else .....	28
	Switch .....	29
Capítulo 5.	Estruturas de Repetição .....	31
	FOR .....	31

WHILE .....	32
DO...WHILE .....	33
FOREACH .....	33
Capítulo 6. Arrays.....	34
Capítulo 7. Classes e Objetos .....	37
Campos (Fields).....	38
Propriedades.....	38
Métodos .....	39
Sobrecarga .....	44
Construtores .....	46
Capítulo 8. Herança.....	48
Capítulo 9. Abstração .....	51
Classes Abstratas .....	51
Métodos Abstratos .....	51
Membros Virtuais .....	53
Classes Parciais .....	55
Classes Seladas .....	56
Capítulo 10. Polimorfismo.....	58
Polimorfismo com Herança .....	59
Polimorfismo com Classe Abstrata .....	59
Polimorfismo com Interfaces.....	60
Capítulo 11. Interfaces .....	61

Trabalhando com Interfaces .....	61
Capítulo 12. Listas 64	
O que são listas? .....	64
Inicializando Listas .....	65
Manipulando Listas .....	66
Capítulo 13. Membros Estáticos .....	68
Classes Estáticas .....	68
Campos Estáticos .....	70
Métodos Estáticos .....	70
Construtores Estáticos .....	71
Capítulo 14. Exceções .....	72
Referências .....	74

## Capítulo 1. O Que é C# e .NET

---

### Contextualizando C# e .NET

---

A linguagem C# desenvolvida pela Microsoft é parte do conjunto de ferramentas oferecidas na plataforma .NET. Além da simplicidade, é caracterizada pela robustez, totalmente orientada a objetos, fortemente tipada e altamente escalável. O C# oferece um grande escopo de possibilidades e projetos com a ideia de atender diversas plataformas a fim de permitir que uma mesma aplicação possa ser executada em diversos dispositivos de hardware.

A Microsoft possuía, no fim da década de 1990, diversas tecnologias e linguagens de programação para resolver problemas diferentes. Quando havia a necessidade de migrar um projeto para uma nova linguagem era necessário aprender os paradigmas, conceitos e bibliotecas que a envolviam. Para resolver esse tipo de problema, a Microsoft recorreu à linguagem Java, pois através dela era possível construir programas independentes do ambiente de execução, além do benefício de contar com bibliotecas para resolver problemas comuns e diferentes naturezas.

Durante a utilização do Java, a Microsoft identificou que havia problemas na comunicação com as bibliotecas existentes em código nativo. Então, optaram por criar uma nova e própria implementação do Java, denominada J++, que através de suas extensões proprietárias, solucionava problemas de comunicação com código nativo.

A J++ era uma versão da linguagem Java que só podia ser executada no ambiente Microsoft. Esse fato gerou problemas judiciais pela violação de licenciamento da Microsoft feito em relação a Sun. Então a Microsoft foi obrigada a repensar a estratégia e a empresa começou a trabalhar em uma nova plataforma chamada .NET, que seguia o mesmo conceito de trabalhar com diversas linguagens de programação compartilhando o mesmo conjunto de bibliotecas.

Com a evolução da .NET Framework, que antes era totalmente proprietário, agora é uma plataforma de código-aberto gratuita que provê ferramentas e guias para que possa ser usada a fim de criar uma grande variedade de aplicações web, gaming,

dispositivos móveis, desktops e internet das coisas (IoT). Liberado desde 2002 em sua primeira versão, na qual desenvolvedores e companhias a usavam para criar aplicações baseada em formulários e web. A presente estrutura *cross-platform* cria um forte contexto em aceitar múltiplas linguagens de programação com uma grande quantidade de bibliotecas, resultando em versatilidade, facilidade de uso, poder de ganho e, conseqüentemente, adoção entre empresas que buscam a proficiência do universo .NET.

## Máquina Virtual

---

Uma máquina virtual (VM) é uma emulação virtual de um sistema operacional de computador, porém com diferentes tipos de abstração de sistema. As máquinas virtuais do sistema existem como sistemas operacionais totalmente funcionais e são normalmente criadas como um substituto para o uso de uma máquina física. Em relação ao C#, existem máquinas virtuais de processo que situam entre o sistema operacional e aplicação como uma camada responsável por traduzir o que a aplicação deseja realizar com as respectivas chamadas do sistema operacional onde está executando no momento, além de empregar coleta de lixo, operações baseadas em pilha, segurança em nível de execução e tratamentos de exceções.

As aplicações são executadas sem envolvimento com sistema operacional, comunicando apenas com a camada intermediária (máquina virtual do C#), conhecida como *Common Language Runtime* (CLR).

O CLR trabalha com todas as linguagens da plataforma .Net e se situa totalmente isolado do sistema operacional, não afetando outras máquinas virtuais ou o sistema operacional em caso de falha. Mesmo trabalhando com diversas linguagens o CLR não executa diretamente código, ele precisa executar uma linguagem intermediária conhecida como *Common Intermediate Language* (CIL), que é um código binário que poderíamos chamar de código de máquina da plataforma, sendo gerado pelo compilador da linguagem.

## Capítulo 2. O Ambiente de Desenvolvimento do C#

---

### Ambientes de Desenvolvimento Integrado (IDE)

---

**Figura 1 - Logo Visual Studio.**



O Visual Studio é o ambiente utilizado para desenvolver código C# e outras diversas linguagens, oferecendo suporte para o desenvolvimento de aplicações desktop, mobile, web e IoT (internet das coisas) e é distribuído pela própria Microsoft. Você poderá usar o Visual Studio Community 2019, versão gratuita da ferramenta, para o desenvolvimento do projeto deste curso. Seu download pode ser realizado no site: <https://www.visualstudio.com/pt-br/downloads/>.

O .Net Framework será automaticamente instalado durante o processo de instalação do Visual Studio, permitindo que sua máquina esteja pronta para executar as aplicações escritas em C#.

Além do Visual Studio, existem outras IDEs que podem ser utilizadas para o desenvolvimento, como Visual Studio Code, Visual Studio for Mac ou Rider da JetBrains.

**Figura 2 - Logo Visual Studio Code.**





Apesar de ser um editor de textos para desenvolvedores, o Visual Studio Code (ou vscode) é tão completo que é frequentemente confundido como uma IDE. Criada pela Microsoft, o vscode é um editor open source multiplataforma e com diversos recursos para o desenvolvimento C#. Possui suporte nativo ao JavaScript, TypeScript, JSON, HTML, CSS e outras tecnologias, além disso, é possível instalar plugins para melhorar o suporte para outras tecnologias, como o próprio C#.

**Figura 3 - Logo Visual Studio for MAC.**



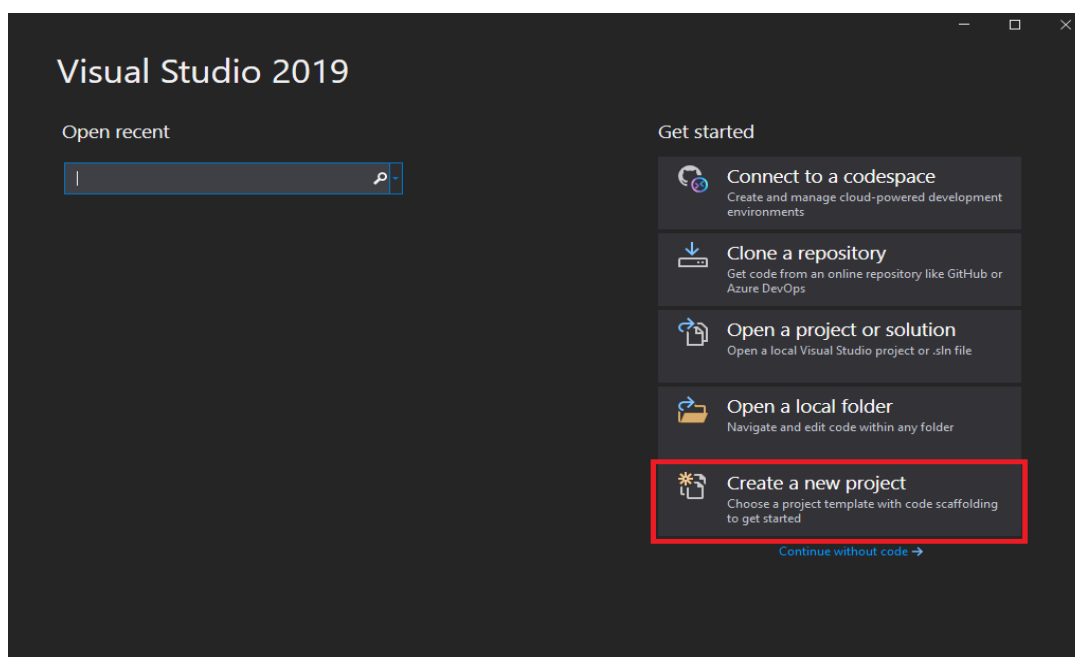
O Visual Studio for MAC é um ambiente de desenvolvimento integrado .NET no Mac que pode ser usado para editar, depurar e construir código e, em seguida, publicar um aplicativo. Além de um editor de código e depurador, inclui compiladores, ferramentas de autocompletar código, designers gráficos e recursos de controle de origem para facilitar o processo de desenvolvimento de software.

### O Primeiro Programa em C#

---

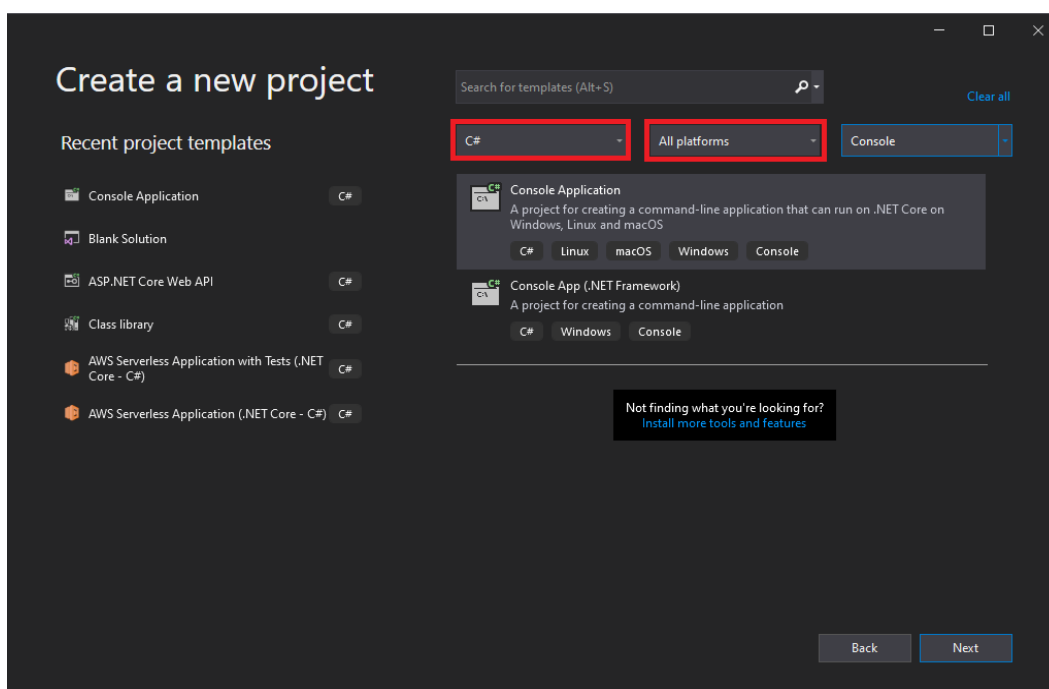
- Abra o Visual Studio.
- Na janela **Iniciar**, escolha **Criar um novo projeto**.

**Figura 4 - Criação de um novo projeto no Visual Studio 2019.**



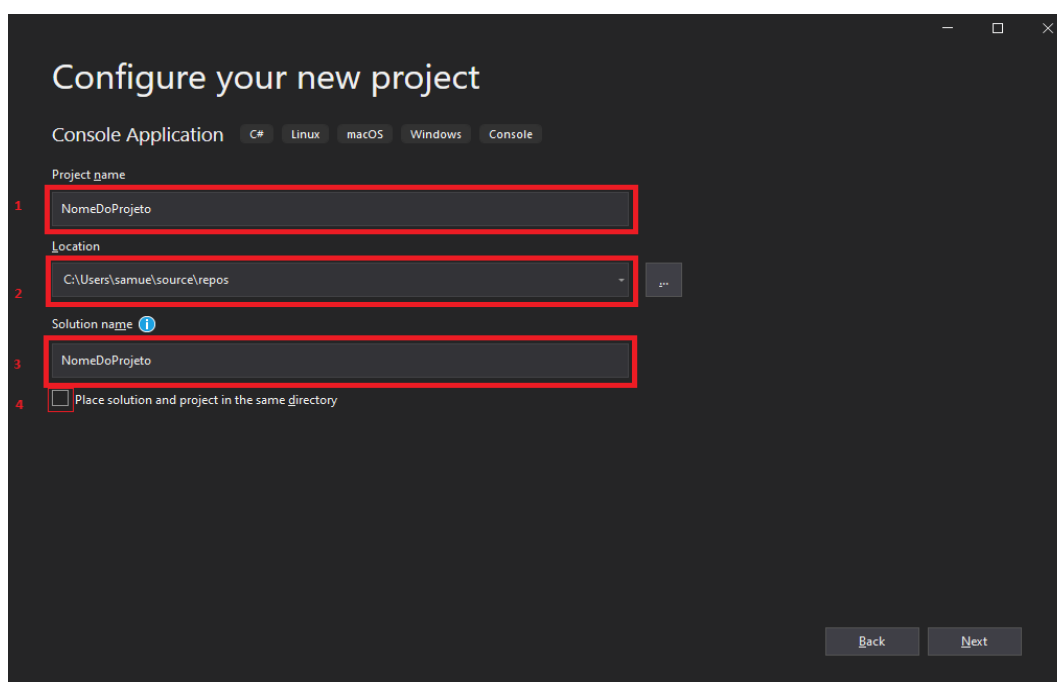
- Na janela ***Criar um novo projeto***, escolha **C#** na lista de idiomas, em seguida, escolha **Windows** na lista de plataformas e **Console** na lista de tipos de projeto.
- Depois de aplicar o idioma, a plataforma e os filtros de tipo de projeto, escolha o modelo de ***aplicativo de console*** e, em seguida, escolha ***Avançar***

**Figura 5 - Opções para criação de um novo projeto.**



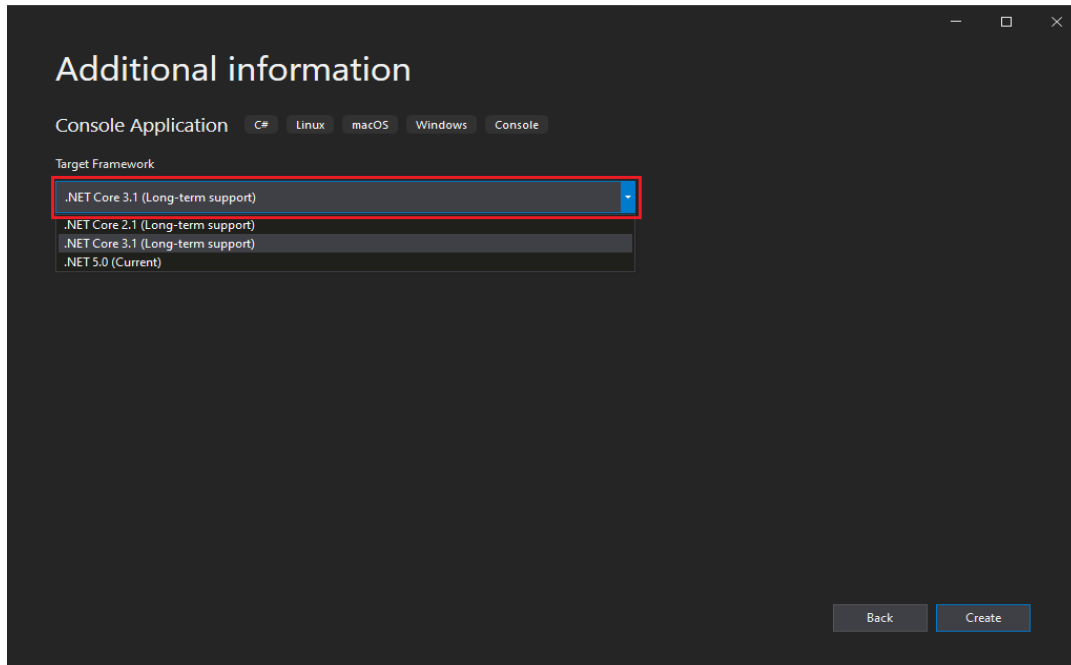
- Na janela **Configurar seu novo projeto**, digite o nome do projeto e escolha a localização. Defina o nome da solução (local onde se agrupam os projetos) e escolha se a solução e o projeto ficarão dentro do mesmo diretório através do checkbox na parte inferior da janela de diálogo. Em seguida, escolha **Avançar**.

**Figura 6 - Configurações de um novo projeto.**



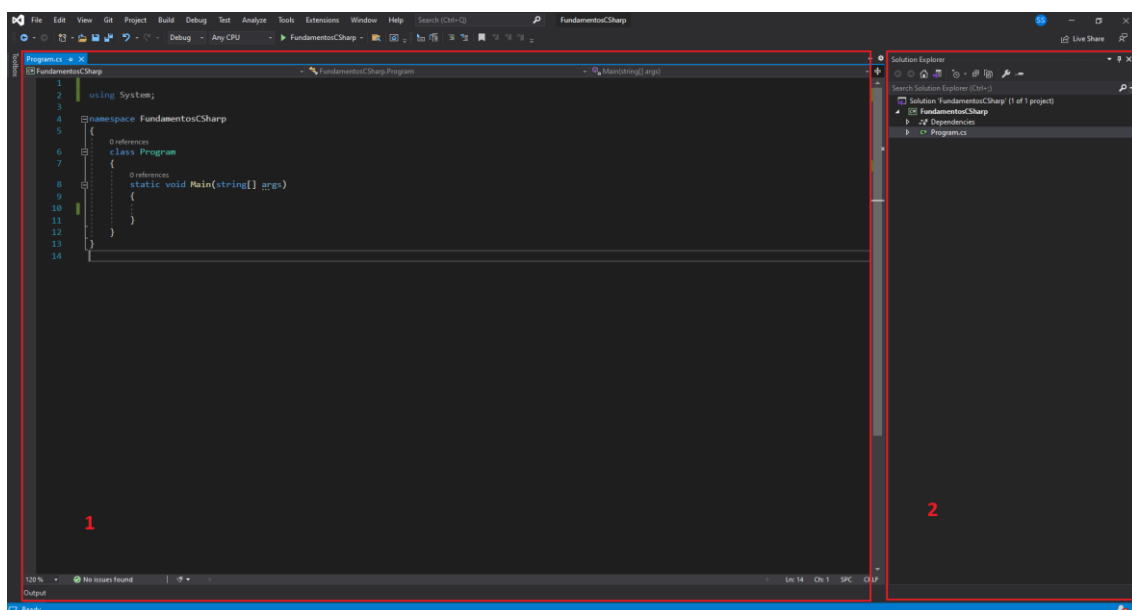
- Na janela **Informações Adicionais** escolha o framework e clique em **Criar** para finalizar a criação do projeto.

**Figura 7 - Informações Adicionais de um novo projeto.**



Após a criação do projeto, somos apresentados ao ambiente abaixo. Sendo a primeira área, onde o código será editado e a segunda área, o explorador da solução que nos permite localizar os arquivos e onde eles são alocados após a criação.

**Figura 8 - Visualização do Projeto no Visual Studio 2019.**



Em um projeto console temos a seguinte estrutura:

- Uma declaração **using** que nos permite referenciar bibliotecas no código atual. No caso apresentado, **System** se refere a uma biblioteca presente no próprio *framework*.
- Um **namespace** fornece uma maneira de manter um conjunto de nomes separado de outro. Os nomes de classe declarados em um namespace não entram em conflito com os mesmos nomes de classe declarados em outro.
- A declaração da classe **Program** como elemento principal de um projeto do tipo *console*.
- Um método estático **Main**, onde se concentra a estrutura do projeto.

**Figura 9 - Classe Principal do Projeto.**

```
using System;

namespace FundamentosCSharp
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
        }
    }
}
```

## Capítulo 3. Variáveis e Tipos Primitivos

Para armazenar dados, precisamos solicitar ao C# que reserve regiões de memória para guardar estas informações. Essas regiões de memória são conhecidas como variáveis.

As variáveis guardam informações de um tipo específico. Poderíamos, por exemplo, guardar um número inteiro representando o identificador de um cliente, um texto para representar um endereço ou um número real para representar o valor de um débito. Para utilizar uma variável, devemos primeiramente declará-la no contexto do programa.

No C# toda variável deve ter nome, tipo de dados e modificador de acesso. Entretanto, o modificador de acesso não é obrigatório, isto porque no C#, quando não há um modificador, é atribuído à variável o modificador padrão **private**. Visto isso, para declarar uma variável do tipo inteiro que representa o identificador de um cliente, utilizamos o seguinte código:

**Figura 10 - Exemplo de declaração de uma variável.**

```
int identificadorCliente;
```

Repare no **ponto-e-vírgula ( ; )** ao final da linha. Ao final de comandos da linguagem C#, precisamos do **ponto-e-vírgula ( ; )** para terminá-lo.

Além do tipo **int** (para representar inteiros), temos também os tipos **double** e **float** (para números reais), **string** (para textos), entre outros.

Depois de declarada, uma variável pode ser utilizada para armazenar valores. Por exemplo, para atribuir valor a variável **identificadorCliente** que declaramos anteriormente, utilizamos o seguinte código:

**Figura 11 - Exemplo de declaração e atribuição de valor a uma variável.**

```
int identificadorCliente;
identificadorCliente = 123456;
```

Lê-se "*identificadorCliente* recebe 123456". Quando, no momento da declaração da variável, sabemos qual será seu valor, podemos utilizar a seguinte sintaxe para declarar e atribuir o valor para a variável:

**Figura 12 - Exemplo de atribuição direta de valor a uma variável.**

```
int identificadorCliente = 123456;
```

## Tipos Primitivos

Em C#, toda variável tem um tipo e abaixo estão listados os tipos primitivos ou tipos valor (***value types***).

**Figura 13 - Tabela de Tipos Primitivos.**

Tipos primitivos do C#	Tipo Framework Class Library	Descrição
sbyte	System.SByte	Valor a 8 bits com sinal
byte	System.Byte	Valor a 8 bits sem sinal
short	System.Int16	Valor a 16 bits com sinal
ushort	System.UInt16	Valor a 16 bits sem sinal
int	System.Int32	Valor a 32 bits com sinal
uint	System.UInt32	Valor a 32 bits sem sinal
long	System.Int64	Valor a 64 bits com sinal
ulong	System.UInt64	Valor a 64 bits sem sinal
char	System.Char	Caracter Unicode a 16 bits
float	System.Single	IEEE 32 bits
double	System.Double	IEEE 64 bits
bool	System.Boolean	1 bit (True ou False)
decimal	System.Decimal	Float 128 bits (nao e primitivo no CTS)

As variáveis de tipo de valor contêm diretamente seus valores, o que significa que a memória é alocada embutida em qualquer contexto em que a variável é declarada.

Para atribuir um valor a uma variável, escreva o nome da variável, seguido pelo operador de atribuição ( = ) e então o valor, do mesmo tipo da variável, a ser atribuído. Veja os exemplos a seguir:

**Figura 14 - Outros exemplos de declaração e atribuição de variáveis.**

```
bool valido = true;

int idade = 28;

double latitude = -17.5635272;

decimal total = 12.35M;

char caracter = 'W';
```

#### ▪ Conversão de Valores:

Uma situação comum é a conversão de valores em uma aplicação. Por exemplo, quando um valor inteiro é passado para uma variável do tipo decimal.

**Figura 15 - Conversão de valores.**

```
int numero = 1234;

decimal numero2 = numero;
```

Para esse tipo de conversão, damos o nome de conversão implícita que é aquela em que não é necessário nenhum código especial, pois se trata de uma conversão segura, sem risco de perda de dados.

Existem outros tipos de conversões de dados, vejamos alguns deles:



- **Conversão explícita (casts):** esse tipo de conversão necessita de um operador. É realizado quando há a necessidade de se converter um valor e pode ocorrer perda de informações. Para fazer este **casting**, deve ser informado o tipo entre parênteses na frente da variável que será convertida.

**Figura 16 - Conversão explícita.**

```
double numero = -35.70;
int novoNumero = (int)numero;
```

- **Conversões com classes auxiliares:** são as conversões em que se utiliza a classe **System.Convert** a fim de converter para o tipo desejado ou o método **Parse** dos tipos de dados.

**Figura 17 - Conversão com classes auxiliares.**

```
int identificador = Convert.ToInt32("1234");
```

- **Conversões definidas pelo usuário:** são conversões realizadas por métodos criados pelos desenvolvedores que contém o código que converte as informações para a maneira desejada.

#### ▪ Nome de variáveis:

A regra para nomear uma variável é sempre começar por uma letra ou `_`. No meio do nome podem-se usar números, mas não se devem usar caracteres especiais e também não pode ser uma palavra reservada. Exemplos de palavras que não podem ser utilizadas são: *if*, *for*, *while*, *string* etc.

#### ▪ Constantes:

Constantes têm características semelhantes às variáveis, entretanto possuem valores que são definidos em tempo de compilação e não podem ser

alterados. Para declarar uma constante, usamos a palavra reservada **const** antes de informar seu tipo e seu nome.

**Figura 18 - Exemplo de declaração de constante.**

```
const double medida = 10.5;
```

#### ▪ Modificadores de Acesso:

Os modificadores de acesso são palavras-chave usadas que definem a visibilidade de um membro ou de um tipo. No C# existem os seguintes modificadores:

1. *public*
2. *protected*
3. *internal*
4. *private*

Os níveis de acessibilidade podem ser especificados usando os modificadores de acesso, são eles:

- **public**: O acesso não é restrito.
- **protected**: O acesso é limitado à classe que a contém ou aos tipos derivados da classe que a contém.
- **internal**: O acesso é limitado ao assembly atual.
- **protected internal**: O acesso é limitado ao assembly ou tipos atuais derivados da classe que a contém.
- **private**: O acesso é limitado ao tipo recipiente.
- **private protected**: O acesso é limitado à classe que a contém ou aos tipos derivados da classe que a contém dentro do assembly atual.

## Tipos de Dados

---

Em C#, as variáveis e objetos devem ter um tipo declarado, por isso trata-se de uma linguagem **fortemente tipada**. O valor atribuído a uma variável deve corresponder com o seu tipo declarado.

Os tipos de dados são divididos em **value types** (Tipos Valor) e **reference types** (Tipos Referência). Os **value types** são derivados de **System.ValueType** e **reference types** são derivados de **System.Object**.

As variáveis **value type**, como o nome propriamente sugere, contêm dentro delas um valor, enquanto as **reference type** contêm uma referência. Isso significa que se copiar uma variável do tipo **value type** para dentro de outra variável, o valor é copiado. No caso da variável do tipo **reference type** será copiado apenas a referência do objeto.

### ▪ Variáveis Tipo Valor (**Value Type**):

Dentro de **Value Type** existem duas categorias: *struct* e *enum*.

- **Struct**: é dividida em tipos numéricos, bool e estruturas personalizadas pelo usuário.
- **Enum**: permite criar um tipo que é formado por várias constantes. Geralmente é usada quando existe a necessidade de um atributo que pode ter múltiplos valores.

### ▪ Variáveis Tipo Referência (**Reference Type**):

As variáveis **reference type** armazenam apenas a referência do objeto. Os tipos de referência são: *class*, *interface*, *delegate*, *object*, *string* e *Array*.

- **Tipo object**: todos os tipos são derivados da classe *Object*, sendo assim é possível converter qualquer tipo para *object*.
- **Tipo string**: é utilizado para se armazenar caracteres e uma *string* deve estar entre aspas.

## Enum (Enumeradores)

A palavra-chave **enum** é usada para declarar um tipo distinto que consiste em um conjunto de constantes, ou seja, uma enumeração. O **Enum** é um tipo de valor e não pode herdar ou ser herdado. Os itens de uma enumeração devem ser separados por vírgula:

**Figura 19 - Exemplo de Enumeração.**

```
2 references
enum Nivel
{
    Baixo,
    Medio,
    Alto
}

0 references
static void Main(string[] args)
{
    Nivel nivel = Nivel.Alto;
    Console.WriteLine(nivel);
}
```

## Struct (Estruturas)

Em C#, **structs** são tipos de valor que representam estruturas de dados. Ele pode conter um construtor parametrizado, construtor estático, constantes, campos, métodos, propriedades, indexadores, operadores, eventos e tipos aninhados. *Struct* pode ser usado para conter pequenos valores de dados que não requerem herança, por exemplo, pontos de coordenadas, pares de valores-chave e estrutura de dados complexa.

Uma estrutura é declarada usando a palavra-chave **struct**. O exemplo a seguir declara uma estrutura **Medidas** que contém as propriedades *altura* e *largura*.

Figura 20 - Exemplo de Struct.

```

2 references
struct Medidas
{
    public int altura;
    public int largura;
}

0 references
static void Main(string[] args)
{
    Medidas medidas = new Medidas();
    Console.WriteLine(medidas.altura);
    Console.WriteLine(medidas.largura);
}

```

Uma struct pode conter propriedades, propriedades auto-implementadas, métodos etc., da mesma forma que as classes. A principal diferença entre estruturas e classes é que não podemos usar uma estrutura como base de outras estruturas ou classes para herança.

Figura 21 - Exemplo de Struct.

```

3 references
struct Medidas
{
    2 references
    public int Altura { get; set; }
    2 references
    public int Largura { get; set; }

    0 references
    public Medidas(int altura, int largura)
    {
        Altura = altura;
        Largura = largura;
    }

    0 references
    public void DefinirValorInicial()
    {
        Altura = 0;
        Largura = 0;
    }
}

```

## Inferência de Tipos

Também conhecida como **target typing**, a inferência de tipos permite que o compilador decida o tipo de dado pelo contexto da expressão. No exemplo abaixo, o compilador sabe que **saudacao** é uma variável **string**, pois o valor atribuído a ela é uma sequência de caracteres.

**Figura 22 - Exemplo de inferência de tipos.**

```
var saudacao = "Oi";
```

Quando esse tipo não pode ser inferido, o tipo da variável deve ser especificado. Veja o exemplo abaixo:

**Figura 23 - Exemplo de especificação de tipo.**

```
string saudacao = null;
```

### ▪ Inferência de tipos em “expressões new”:

Em versões anteriores ao C# 9.0, a inferência de tipos era realizada fazendo o uso da palavra-chave **var**, limitando seu uso às declarações das variáveis/objetos. Agora, esse recurso foi estendido à cláusula **new**.

Para declarar um novo objeto em C#, a sintaxe usada é **new T()**, onde T é o tipo do objeto. No C# 9.0, se o tipo do objeto estiver sendo especificado, ele pode ser omitido da expressão **new**, pois o compilador será capaz de inferir qual construtor deve ser invocado baseado no tipo do objeto. É importante ressaltar que se o construtor recebe algum parâmetro, este deve ser informado na criação da instância.

**Figura 24 - Exemplos de inferência de tipos com expressão new.**

```
Pedido pedido = new();  
Pedido pedido = new("Compras");
```

Como o tipo do objeto precisa ser especificado na expressão, não é possível declarar um novo objeto usando **new** e se estiver ocultando o tipo através da palavra-chave var. O exemplo a seguir irá gerar um erro de compilação:

**Figura 25 - Erro na inferência de tipos.**

```
var pedido = new("Compras");
```

Para corrigir o erro acima, pode-se reescrever a expressão através de uma das seguintes formas:

**Figura 26 - Declaração do objeto Pedido.**

```
Pedido pedido = new("Compras");  
  
var pedido = new Pedido("Compras");
```

## Operações com variáveis

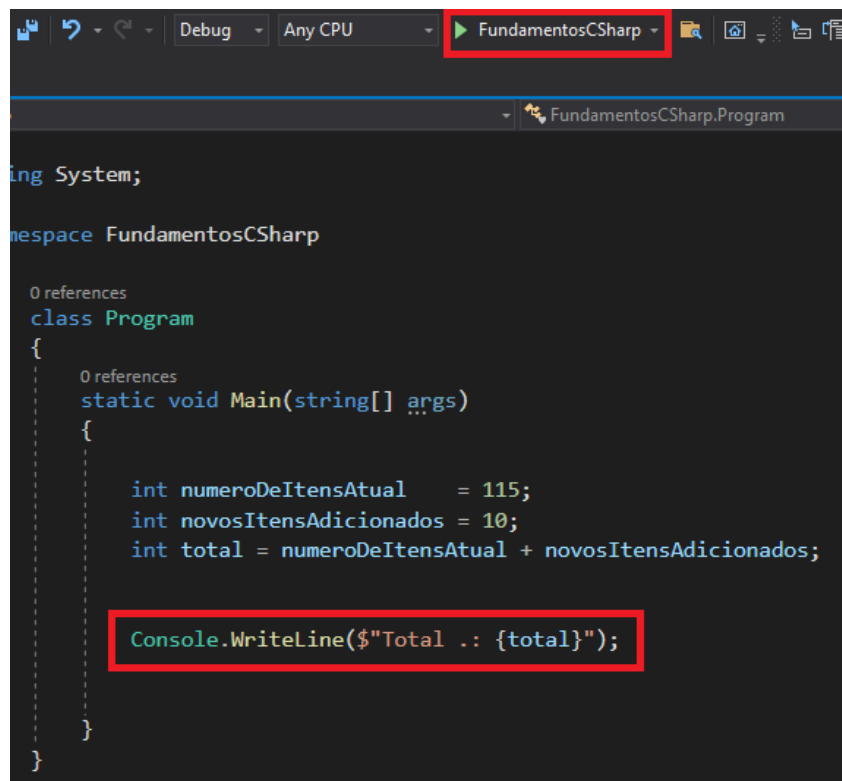
Agora que já sabemos como guardar informações no programa, estamos interessados em executar operações nesses valores. Pode ser interessante, por exemplo, para um cliente saber qual é a quantidade de itens no carrinho de compras após a adição de mais 10 itens. Para isso, realizamos a seguinte operação:

**Figura 27 - Exemplo de operação com variáveis.**

```
int numeroDeItensAtual    = 115;
int novosItensAdicionados = 10;
int total = numeroDeItensAtual + novosItensAdicionados;
```

Nesse código estamos guardando na variável **total**, o valor dos itens atuais (quantidade anterior a soma) mais o número de itens adicionados, sendo o valor final igual a 125. Depois de realizar a operação, queremos mostrar para o usuário qual é o total atual de itens adicionados.

**Figura 28 - Exemplo de programa para operação de soma com variáveis.**



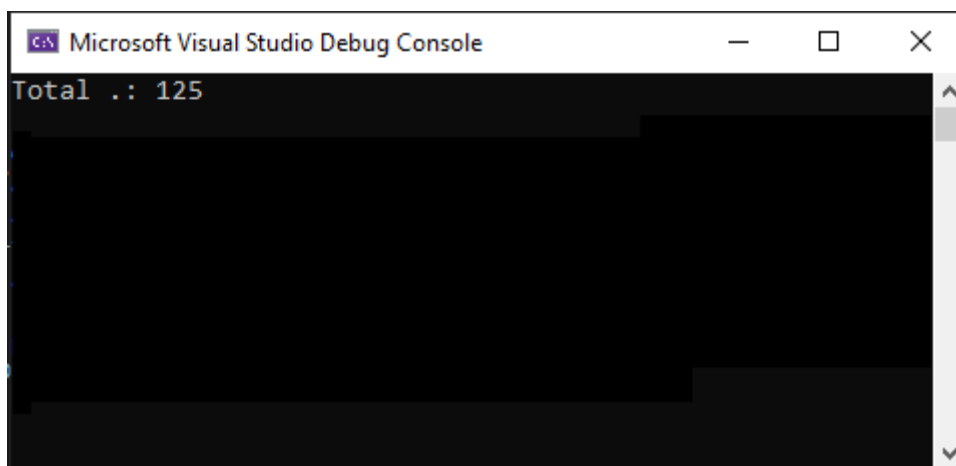
```
using System;

namespace FundamentosCSharp
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            int numeroDeItensAtual    = 115;
            int novosItensAdicionados = 10;
            int total = numeroDeItensAtual + novosItensAdicionados;

            Console.WriteLine($"Total .: {total}");
        }
    }
}
```



**Figura 29 - Resultado da execução da aplicação da figura 28.**



Também é possível substituir a variável **total** pela variável **numeroDeltensAtual**, a fim de que esta receba o valor da soma. As expressões abaixo representam formas equivalentes de realizar a soma das variáveis:

**Figura 30 - Equivalência de operação com variáveis.**

```
numeroDeItensAtual = numeroDeItensAtual + novosItensAdicionados;
numeroDeItensAtual += novosItensAdicionados;
```

Quando o compilador do C# encontra a expressão “numeroDeltensAtual += novosItensAdicionados”, a linha é interpretada da mesma forma como vimos anteriormente, isto é, “numeroDeltensAtual = numeroDeltensAtual + novosItensAdicionados”.

Da mesma forma, podemos subtrair valores (operador -), podemos também fazer multiplicações (operador \*), divisões (operador /) e obter o resto da divisão (operador %). Além do operador +=, tem-se também os operadores -= (para subtrações), \*= (para multiplicações) e /= (para divisões).

## Variáveis do Tipo String

Já vimos os tipos primitivos existentes para diferentes propósitos. Podemos agora destacar que o C# possui um tipo específico para armazenar textos. O tipo **string** é responsável por representar valores de cadeias de caracteres.

**Figura 31 - Exemplo de declaração de string.**

```
string mensagem = "Fundamentos CSharp";
```

Para realizar a junção de duas ou mais strings, podemos utilizar o operador **+**. Essa operação é conhecida como concatenação de strings.

**Figura 32 - Concatenação de strings.**

```
string mensagem = "Fundamentos CSharp" + " " + "2021";
```

## Interpolação de Cadeia de Caracteres

Interpolação de cadeia de caracteres ou interpolação de strings baseia-se no recurso de formatação composta e fornece uma sintaxe mais legível e conveniente para incluir resultados da expressão formatada em uma cadeia de caracteres de resultado.

A interpolação de cadeia de caracteres pode ser obtida prefixando a string com um caractere **\$**. Veja abaixo:

**Figura 33 - Exemplos de Interpolação de Strings.**

```
string cliente = "Renato Andrade";
int idade = 42;
Console.WriteLine(cliente + "tem " + idade + " anos");
Console.WriteLine($"{cliente} tem {idade} anos");
```

Nos dois casos, os resultados do Console.WriteLine são iguais. Entretanto, enquanto no primeiro a concatenação é feita tradicionalmente usando duas variáveis, no segundo modo a concatenação é feita usando interpolação de cadeia de caracteres. Note que na segunda opção as variáveis são injetadas diretamente na string, cercado-as com chaves `{ }`.

As variáveis usadas não precisam ser tipos simples, sendo possível usar propriedades de objetos complexos da mesma forma de variáveis simples. No exemplo abaixo temos a criação de uma instância da classe Cliente que possui nome e idade como propriedades. A seguir, tem-se como se dá a interpolação de cadeia de caracteres através das propriedades deste objeto.

**Figura 34 - Interpolação de Strings através de Propriedades de Objetos.**

```
var cliente = new Cliente("Renato Andrade", 42);

Console.WriteLine($"{cliente.Nome} tem {cliente.Idade} anos");
```

## Comentários

Dentro do C#, algumas vezes pode ser necessário documentar o que estamos fazendo ou o propósito do que estamos construindo através de comentários e podemos fazer isso de duas formas:

**Figura 35 - Comentário de Linha Única.**

```
//1. Comentário em linha unica
```

**Figura 36 - Comentário de Múltiplas Linhas**

```
/*
    2.
    Comentário em
    múltiplas
    linhas.
*/
```

## Capítulo 4. Estruturas de Controle

Estruturas de Controle são usadas quando uma aplicação precisa tomar uma determinada decisão mediante o valor de uma expressão ou variável para definir um caminho durante a execução do software.

### If...Else

Em C#, um exemplo de estrutura condicional simples utilizando a construção **if** pode ser visto no exemplo abaixo:

**Figura 37 - Estrutura IF simples.**

```
if (condicao)
{
    //Execute alguma ação se a condição for verdadeira.
}
```

Vamos imaginar que queremos adicionar mais um item a um carrinho de compras, mas essa ação somente será possível se houver mais itens disponíveis deste produto. Para tal ação precisamos fazer a execução condicional de código.

**Figura 38 - Exemplo IF simples.**

```
int computadoresDisponiveis = 10;

if (computadoresDisponiveis > 0)
{
    novosItensAdicionados += 1;
    computadoresDisponiveis -= 1;
}
```

Estendendo um pouco mais a implementação acima podemos ainda ter uma ação se a condição não for atendida, sendo realizada através da construção **else**, na qual poderíamos informar ao usuário a indisponibilidade do produto.

Figura 39 - Exemplo IF...ELSE.

```
if (computadoresDisponiveis > 0)
{
    novosItensAdicionados += 1;
    computadoresDisponiveis -= 1;
}
else
{
    Console.WriteLine("Produto indisponível.");
}
```

Existem outros operadores de comparação que podemos utilizar em condições como menor (<), menor ou igual (<=), maior ou igual (>=), igual (==) e diferente (!=). Podemos negar uma condição de um *if* utilizando o operador *!* na frente da condição. Podemos também verificar se duas condições são verdadeiras ou se pelo menos uma delas é verdadeira usando o operador *&&* (*AND*) para fazer um “e lógico” e através do operador *||* (*OR*) para fazer um “ou lógico”.

## Switch

A instrução ***switch*** pode ser usada em vez da instrução ***if...else*** quando você deseja testar uma variável em três ou mais condições.

A sintaxe usa a palavra-chave ***switch*** que contém uma ***expressão*** ou uma variável dentro de colchetes. O resultado dessa expressão de correspondência ou de uma variável será testado em relação às condições especificadas como ***casos***, dentro das chaves ***{}***.

Um caso deve ser especificado com o valor constante exclusivo e termina com dois pontos (***:***). Cada caso inclui uma ou mais instruções que serão executadas se o valor do ***“case”*** e o valor da ***“condição switch”*** forem iguais.

A instrução **switch** também pode conter um rótulo padrão opcional. O rótulo padrão será executado se nenhum caso for válido. As palavras-chave **break**, **return** ou **goto** devem ser usadas para sair do **switch**

O exemplo a seguir demonstra uma instrução **switch** simples.

**Figura 40 - Exemplo de Switch.**

```
int valorAtual = 10;

switch (valorAtual)
{
    case 5:
        Console.WriteLine("O valor da variável corresponde a : 5.");
        break;
    case 10:
        Console.WriteLine("O valor da variável corresponde a : 10.");
        break;
    case 15:
        Console.WriteLine("O valor da variável corresponde a : 15.");
        break;
    default:
        Console.WriteLine("Valor desconhecido.");
        break;
}
```

Acima, a instrução **switch(valorAtual)** inclui a variável **valorAtual** cujo valor será combinado com o valor de cada caso determinado pela palavra-chave **case**.

A instrução switch acima contém três casos com valores constantes 5, 10 e 15. Ela também contém o rótulo **default**, que será executado se nenhum dos valores de caso corresponder à variável analisada. Cada caso inclui uma instrução a ser executada.

No exemplo, o valor da variável **valorAtual** é 10 e corresponde ao segundo caso, então a saída seria imprimir no console da aplicação a mensagem “O valor da variável corresponde a: 10”.

## Capítulo 5. Estruturas de Repetição

As estruturas de repetição são usadas para controlar a execução de códigos repetidamente enquanto uma condição for atendida. Uma estrutura de repetição é também conhecida como **Loop**.

Veja os tipos de estrutura de repetição que vamos aprender neste artigo:

- for
- while
- do...while
- foreach

### FOR

O primeiro tipo de *loop* que vamos estudar é o **for**. O loop **for** trabalha verificando uma condição e executando um bloco de código enquanto essa condição for verdadeira.

**Figura 41 - Exemplo de estrutura do FOR.**

```
for (int i = 0; i < 10; i++)
{
    //instruções.
}
```

No código acima, declaramos uma variável do tipo inteiro (int) e a inicializamos com o valor 0 (zero). Em seguida, a condição verifica se a variável *i* é menor ou igual a 10. Por último, temos o incremento desta variável.

O funcionamento é simples. No exemplo, todo o código dentro do bloco **for** será executado dez vezes. Lembrando que a variável **i** é uma representação para índice que pode ser redefinida.

## WHILE

Embora com o mesmo objetivo da estrutura de repetição **for**, o loop **while** é mais simples de ser entendido. Na estrutura **while** usa-se a condição que queremos testar e o bloco será executado enquanto a condição for verdadeira. Abaixo a sintaxe de uma estrutura **while**:

Figura 42 - Estrutura do WHILE.

```
while (condicao)
{
    //instruções.
}
```

Em uma estrutura **while**, não esqueça de incrementar ou se assegurar que sua condição pode ser falsa em algum momento, pois caso contrário entrará no que chamamos de **loop infinito**, ou seja, seu programa nunca terá fim.

Figura 43 - Exemplo da estrutura de repetição WHILE.

```
while (contador < 10)
{
    Console.WriteLine($" Contador atual .: {contador}");
    contador++;
}
```



## DO...WHILE

Podemos traduzir "**do**" para "**faça**", ou seja, faça as instruções enquanto (**while**) uma expressão for verdadeira. O **do...while** é igual ao loop **while**, exceto que executa o bloco de código pelo menos uma vez. Exemplo:

**Figura 44 - Exemplo da estrutura DO...WHILE.**

```
var contador = 15;

do
{
    //instruções.
}
while (contador <= 10);
```

No exemplo acima, para o valor inicial do contador, a condição do **while** será falsa. Portanto, o bloco de instruções **do** será executado apenas uma vez.

## FOREACH

O loop **foreach** é usado para percorrer listas. Essa estrutura opera sobre Arrays e Coleções. Veja um exemplo:

**Figura 45 - Exemplo da estrutura FOREACH.**

```
string[] linguagens = { "C#", "Typescript", "F#", "Python" };

foreach (string linguagem in linguagens)
{
    Console.WriteLine(linguagem);
}
```

## Capítulo 6. Arrays

*Arrays* são coleções de dados extremamente importantes em qualquer linguagem de programação. Sua grande vantagem está no fato de serem estruturas indexadas, ou seja, os itens dos *arrays* podem ser acessados através de índices, o que garante grande velocidade para essas operações.

A declaração de arrays é algo relativamente simples no C#. No entanto, é importante destacar que é essencial sabermos exatamente o tamanho que o array terá, o que acaba sendo uma limitação em alguns casos. Por exemplo, para um array do tipo *double*, com 15 posições, a declaração seria feita da seguinte forma:

**Figura 46 - Exemplo de declaração de array.**

```
double[] array = new double[15];
```

Existem duas formas básicas para inserirmos dados nos *arrays* em C#. A primeira, a declaração explícita, onde o tamanho do array é informado. Neste exemplo temos um *array* que conterá seis números do tipo *double*. Aqui, se passarmos menos ou mais de seis números entre as chaves haverá um erro de compilação, uma vez que sabemos que o array tem seis posições.

**Figura 47 - Exemplo de declaração e atribuição explícita de array.**

```
double[] array = new double[6] { 0, 14, 6, 9.5, 10.3, 23 };
```

A segunda forma seria declará-lo de forma implícita. Nesta oculta-se o tamanho do *array*, pois este estará implícito na quantidade de itens entre as chaves. Neste exemplo passamos cinco números; assim, o *array* possui quatro posições e não há possibilidade de erro de compilação.

**Figura 48 - Exemplo de declaração e atribuição implícita de array.**

```
double[] array = { 0, 14, 6, 9.5 };
```

Poderíamos inserir dados no array acessando o item através de seu índice.

**Figura 49 - Exemplo de atribuição de valor através de índice.**

```
int capacidade = 25;
int[] array = new int[capacidade];

for (int i = 0; i < capacidade; i++)
{
    array[i] = i + 1;
}
```

O acesso aos dados do *array* é feito de forma simples, uma vez que estamos lidando com uma coleção indexada. Os índices de um array se iniciam na posição 0, isto é, para recuperar o primeiro elemento faríamos **array[0]**. Assim, basta acessarmos o índice específico e podemos obter o valor do elemento, como vemos abaixo:

**Figura 50 - Exemplo de acesso a dados através do índice do array.**

```
int capacidade = 25;
int[] array = new int[capacidade];

for (int i = 0; i < capacidade; i++)
{
    array[i] = i + 1;
}

Console.WriteLine(array[10]);
```

É importante, entretanto, dizer que esse tipo de acesso pode ser problemático caso se tente acessar uma posição inexistente no array. Por exemplo, se tivermos um array com cinco posições (os índices serão de 0 a 4) e tentarmos acessar o índice 5, isto é, a sexta posição do array, então o código irá levantar uma exceção. Portanto, é fundamental assegurar que estamos acessando um índice válido.

**Figura 51 - Exemplo de acesso a um índice inexistente no array.**

```
int[] array = new int[5] { 0, 1, 2, 3, 4 };  
  
Console.WriteLine(array[5]);
```

Uma das principais características das coleções são as iterações que podem ser realizadas sobre elas. Para iterarmos sobre os arrays, existem basicamente duas opções: loops em que acessamos os índices no array diretamente (while ou for) ou o loop foreach, que faz isso internamente.

**Figura 52 - Exemplos de iterações sobre arrays/coleções.**

```
int[] array = new int[10];  
  
for (int i = 0; i < 10; i++)  
{  
    Console.WriteLine(array[i]);  
}  
  
int j = 0;  
while (j < 10)  
{  
    Console.WriteLine(array[j]);  
    j++;  
}  
  
foreach (int x in array)  
{  
    Console.WriteLine(x);  
}
```

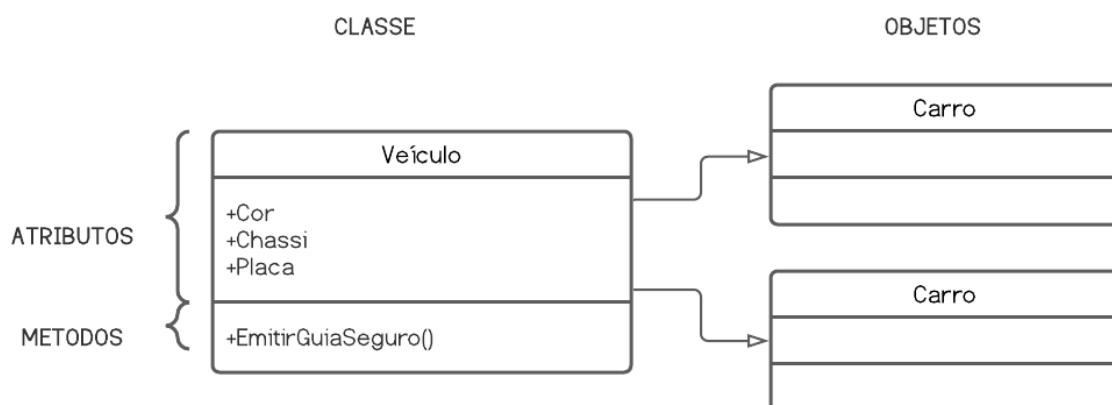
## Capítulo 7. Classes e Objetos

Nesta importante etapa abordaremos um conceito geral de Classes e Objetos na linguagem C#, aproveitando para descrever os tipos mais importantes para nosso aprendizado. Na **Programação Orientada a Objetos** (POO) tudo é baseado em **classes** e **objetos**. Estes conceitos são universais e imprescindíveis, qualquer que seja a linguagem em que seja aplicada.

A POO procura estabelecer um modelo de programação que aproxima o desenvolvedor do mundo real. Uma **classe** é uma unidade do sistema e dentro dela são definidos **campos** e **métodos**, que são respectivamente as informações que uma classe pode armazenar e ações que elas podem desempenhar. Para ficar mais fácil o entendimento, um atributo possui as mesmas funcionalidades de uma variável, assim como um método o mesmo que um procedimento ou função.

Por sua vez, o **objeto** é a **classe materializada**, ou como dizemos, uma **instância** da classe. É importante saber que enquanto existe apenas uma definição de classe, podem existir diversos objetos baseados numa classe.

**Figura 53 - Representação de Classe e Objeto.**



## Campos (Fields)

---

Os campos são responsáveis por guardar informações referentes à classe. Quando um objeto é criado a partir de uma classe, nesse momento os campos passam a ter informações específicas assumindo estados especiais, ou seja, de uma forma geral esse é o momento em que são atribuídos os valores. Existem campos do tipo que permitem acesso externo ao da classe sendo os “Públicos”, os que não permitem sendo os “Privados” e os que permitem somente a visibilidade nas subclasses sendo denominados como “Protegidos”.

**Figura 54 - Exemplo de Campos (Fields).**

```
public string nome;
private string endereco;
```

## Propriedades

---

Uma propriedade é um membro que fornece um mecanismo flexível para ler, escrever ou calcular o valor de um campo particular.

Portanto, envolvemos campos (fields) com Propriedades para nos dar a capacidade de alterar a implementação sem quebrar o código e para permitir um maior controle sobre os membros da classe que desejamos expor ou ocultar.

**Figura 55 - Exemplo de Propriedades.**

```
private string endereco;

public string Endereco
{
    get
    {
        return endereco;
    }
    set
    {
        endereco = value;
    }
}
```

Observe que declaramos a propriedade com get e set onde:

get - significa que estamos dando acesso de leitura a essa propriedade;

set - significa que estamos dando acesso de escrita a essa propriedade;

O encapsulamento é um dos pilares da orientação a objetos. Por meio dele é possível simplificar bastante a programação, bem como proteger informações sigilosas ou sensíveis.

## Métodos

Os métodos são procedimentos ou funções que realizam as ações próprias do objeto. Assim, os métodos são as ações que o objeto pode realizar.

Um método pode ser definido usando o seguinte modelo:

**[modificador de acesso] [tipo de retorno] Nome do método ([tipoParametro nomeParametro])**

Vejamos cada item:

**[modificador de acesso]** – Determina a visibilidade de uma variável ou um método de outra classe.

**[tipo de retorno]** – Um método pode retornar um valor. O tipo de retorno é o tipo de dados do valor que o método retorna. Se o método não estiver retornando valores, o tipo de retorno será inválido.

**[Nome do método]** – Identificador exclusivo é case sensitive (diferença maiúsculas de minúsculas). Não pode ser o mesmo que qualquer outro identificador declarado na classe.

**[Lista de parâmetros]** – Incluídos entre parênteses, os parâmetros são usados para passar e receber dados de um método. Os parâmetros são opcionais; ou seja, um método pode não ter parâmetros.

**Corpo do método** – contém o conjunto de instruções necessárias para completar a atividade requerida.

**Figura 56 - Exemplo de Método sem retorno (void).**

```
public void MetodoSemRetorno(int parametro1, string parametro2)
{
    //Ações a serem executadas no método.
}
```

**Figura 57 - Exemplo de Método com retorno.**

```
public string MetodoComRetorno(int parametro1, string parametro2)
{
    //Ações a serem executadas no método.
    return parametro2 + parametro1;
}
```



Existem quatro diferentes formas de passar parâmetros para um método no C#, são elas:

- Value
- Out
- Ref
- Params

### Parâmetro Value

**Value** é o tipo de parâmetro padrão no C#, ou seja, se o parâmetro não tiver nenhum modificador ele é um parâmetro value por default e seu valor real é passado para a função. Isso significa que qualquer mudança no valor do parâmetro é feita localmente na função e não é devolvida para o código que chamou a função.

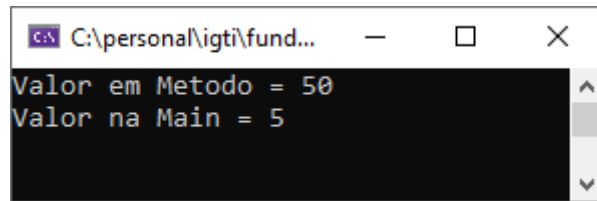
Figura 58 - Exemplo de Parâmetro Value.

```
1 reference
static void MetodoExemplo(int valor)
{
    valor = 50;
    // Aqui a saída será 50
    Console.WriteLine("Valor em Metodo = " + valor);
}

0 references
static void Main(string[] args)
{
    int valor = 5;
    MetodoExemplo(valor);

    //Aqui a saída será 5, perceba que o valor não foi modificado
    Console.WriteLine("Valor na Main = " + valor);
    Console.Read();
}
```

**Figura 59 - Saída da Aplicação.**



### Parâmetro Ref

Parâmetros **ref** são parâmetros de entrada e saída, o que significa que eles podem ser usados tanto para passar um valor para uma função quanto receber de volta esse valor de uma função. Parâmetros ref são criados precedendo um tipo de dado com o modificador ref. Sempre que um parâmetro ref é passado é uma referência da variável que é passada para a função.

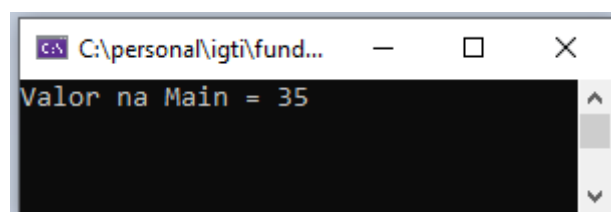
**Figura 60 - Exemplo de Parâmetro Ref.**

```
1 reference
static void MetodoExemplo(ref int valor)
{
    valor += 30;
}

0 references
static void Main(string[] args)
{
    int valor = 5;
    MetodoExemplo(ref valor);

    Console.WriteLine("Valor na Main = " + valor);
    Console.Read();
}
```

**Figura 61 - Saída da Aplicação.**



## Parâmetro Out

O parâmetro **out** significa um parâmetro de referência. Assim, os parâmetros de saída são semelhantes aos parâmetros de referência, exceto pelo fato de que eles transferem dados para fora do método. A palavra-chave **out** força os argumentos a serem passados por referência.

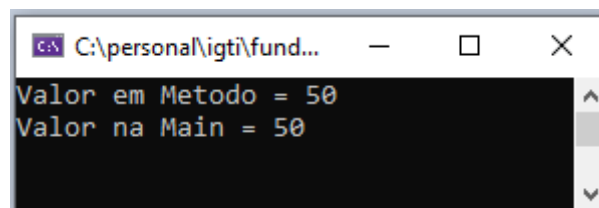
Figura 62 - Exemplo de Parâmetro Out.

```
1 reference
static void MetodoExemplo(out int valor)
{
    valor = 50;
    Console.WriteLine("Valor em Metodo = " + valor);
}

0 references
static void Main(string[] args)
{
    int valor = 5;
    MetodoExemplo(out valor);

    Console.WriteLine("Valor na Main = " + valor);
    Console.Read();
}
```

Figura 63 - Saída da Aplicação.



## Parâmetro Params

Um array de **parâmetros** permite passar um número variável de argumentos para um método. Ele é indicado usando a palavra-chave **params**.

**Figura 64 - Exemplo de Parâmetro params.**

```

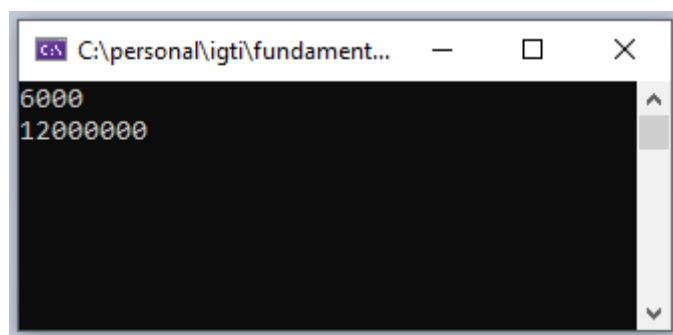
2 references
static int Multiplicar(params int[] parametros)
{
    int valor = 1;

    foreach (int parametro in parametros)
    {
        valor = valor * parametro;
    }
    return valor;
}

0 references
static void Main(string[] args)
{
    Console.WriteLine(Multiplicar(10, 20, 30));
    Console.WriteLine(Multiplicar(10, 20, 30, 40, 50));
    Console.Read();
}

```

**Figura 65 - Saída da Aplicação.**



## Sobrecarga

Sobrecarregar métodos significa ter vários métodos com nomes iguais, mas assinaturas diferentes. A assinatura de um método no C# é dada pela junção do seu nome e seus parâmetros de entrada.

Tomando como base o método abaixo, sua assinatura seria `Calcular(int altura, int largura)`, o que nos permitiria ter outros métodos com o nome `Calcular`, desde que os mesmos contenham parâmetros diferentes.

**Figura 66 - Assinatura do método Calcular.**

```
0 references
public void Calcular(int altura, int largura)
{
    //
}
```

Os métodos abaixo **NÃO SÃO** permitidos, pois possuem assinaturas iguais.

**Figura 67 - Sobrecarga inválida.**

```
0 references
public void Calcular(int altura, int largura)
{
    //
}

// Não permitido
0 references
public void Calcular(int largura, int altura)
{
    //
}

// Não permitido
0 references
public double Calcular(int x, int y)
{
    //
}
```

Mesmo mudando o retorno do método, ainda assim ocorre um erro de compilação, pois o retorno não faz parte da assinatura.

Agora temos alguns exemplos válidos. Mesmo os parâmetros tendo nomes iguais, seus tipos são diferentes, o que permite o método ser sobrecarregado.

**Figura 68 - Sobrecarga válida do método Calcular.**

```

0 references
public void Calcular(int altura, int largura)
{
    //
}

0 references
public void Calcular(int altura, int largura, int x, int y)
{
}

0 references
public void Calcular(Retangulo retangulo)
{
}

```

## Construtores

Construtores são métodos executados na criação da classe. As regras para definição de um construtor são:

- O construtor deve ter o mesmo nome da classe;
- Não tem tipo de retorno;
- É executado apenas um, apenas uma vez, no momento da criação do objeto;
- Não pode ser chamado diretamente;

Os construtores são como os métodos convencionais, podem ser sobrecarregados e conter parâmetros. É nele que indicamos os valores dos campos de uma classe. Esses valores podem ser internos (no código) ou externos (passados por parâmetros).

Figura 69 - Exemplo de construtores.

```

4 references
public class ClasseExemplo
{
    int atributo1, atributo2;

    1 reference
    public ClasseExemplo()
    {
    }

    0 references
    public ClasseExemplo(int atributo1, int atributo2)
    {
        this.atributo1 = atributo1;
        this.atributo2 = atributo2;
    }

    0 references
    public void ExibeValores()
    {
        Console.WriteLine($"O valor 1 é : {atributo1} | O valor 2 é : {atributo2}");
    }
}

```

Figura 70 - Exemplo de criação de objeto.

```

0 references
static void Main(string[] args)
{
    ClasseExemplo c1 = new ClasseExemplo();
    ClasseExemplo c2 = new ClasseExemplo(10, 20);

    c1.ExibeValores();
    c2.ExibeValores();
}

```

## Capítulo 8. Herança

---

O conceito mais importante da POO pode ser considerado a herança, pois através deste pilar extraímos a maior parte das vantagens, principalmente quando nos referimos ao reaproveitamento das características com base em uma superclasse (classe ancestral).

O significado em si nos permite levantar a ideia de reaproveitamento, onde em uma situação natural sabemos que um filho herda características do pai, levando em consideração uma perspectiva mais biológica.

Esse é o mesmo conceito utilizado quando visualizamos o reaproveitamento de características de nossas classes. Analisando o mesmo exemplo, um filho para existir precisa de um pai, conseqüentemente este filho poderá ser pai e partir dele gerando um filho e assim sucessivamente. Poderíamos chegar ao infinito, mas não precisamos, o que é realmente importante é que construção de um filho se torna bem mais simples com a existência de um pai.

Seguindo para um outro exemplo. Construimos inicialmente um carro do início, mas seria algo extremamente complicado se tivéssemos que repetir o processo para um novo carro, pois teríamos que conceituar todo o seu funcionamento, desde o motor, toda a parte elétrica e assim por diante. Isso seria de fato um problema e obviamente não faríamos, mas sim reaproveitaríamos muito do que já foi feito, por exemplo, o próprio motor, todo o câmbio, toda a parte elétrica, mudaríamos apenas aquilo que fosse necessário e único. Neste ponto estamos então herdando vários conceitos de certa classe e aproveitando-os para construir uma nova, implementando novas e até alterando algumas de suas características. Podemos dizer que essa economia de trabalho é significativa para o processo, dependendo do que se quer construir ou herdar.

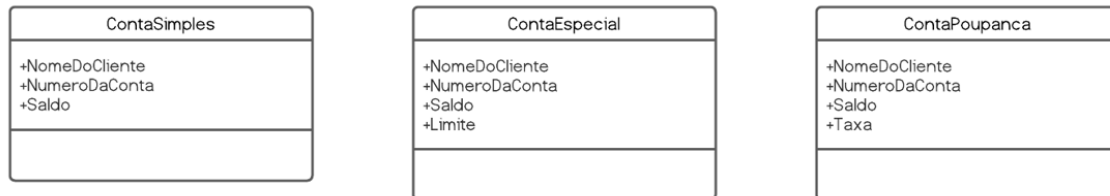
Vamos criar um exemplo onde em um sistema bancário tivéssemos que modelar as seguintes classes para permitir a inclusão de diferentes tipos de contas:

- Conta Simples



- Conta Especial
- Conta Poupança

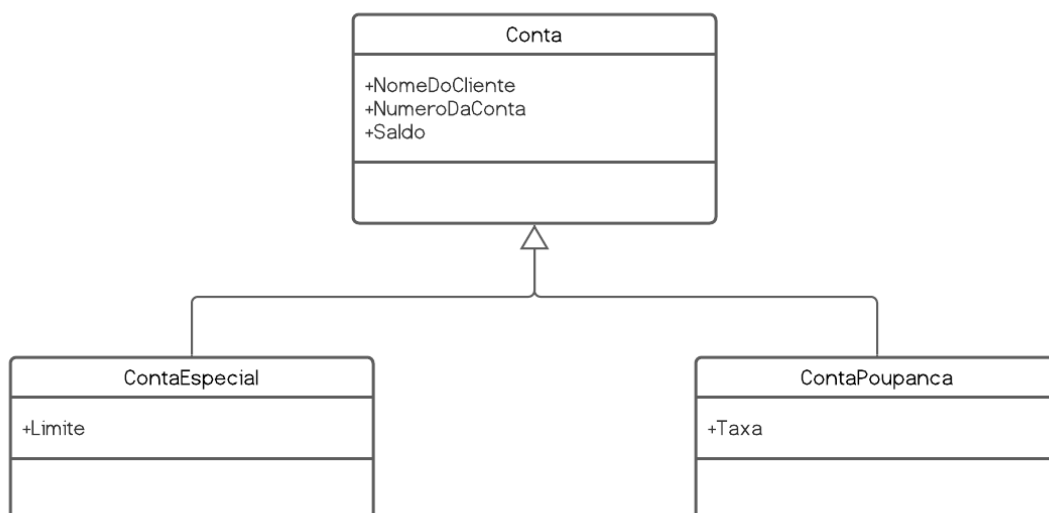
**Figura 71 - Exemplo de Classes com características similares.**



As classes acima possuem os mesmos membros “Nome do cliente”, “nº da conta” e “Saldo” em comum e não haveria motivo para redundância. Com o conceito de herança em mente poderíamos então tomar a classe Conta Simples como a classe base, pois esta possui os três membros comuns a todas as outras, e daí as outras classes simplesmente herdariam, adicionando ou implementando somente as informações específicas ao seu propósito.

O desenho abaixo demonstra as duas situações:

**Figura 72 - Representação de Herança.**



Para representar herança em C#, usamos a estrutura:

*ClasseFilha : ClassePai*

No exemplo abaixo temos ContaPoupanca : Conta, ou seja, a classe ContaPoupanca herda da classe Conta.

**Figura 73 - Exemplo de Aplicação com Herança.**

```
public class Conta
{
    public string NomeDoCliente { get; set; }

    public int NumeroDaConta { get; set; }

    public decimal Saldo { get; set; }
}

public class ContaEspecial : Conta
{
    public decimal Limite { get; set; }
}

public class ContaPoupanca : Conta
{
    public decimal Taxa { get; set; }
}

static void Main(string[] args)
{
    Conta conta = new Conta();
    conta.NomeDoCliente = "Conta base";

    ContaEspecial contaEspecial = new ContaEspecial();
    contaEspecial.NomeDoCliente = "Conta Especial";
    contaEspecial.Limite = 100.0m;

    ContaPoupanca contaPoupanca = new ContaPoupanca();
    contaPoupanca.NomeDoCliente = "Conta Poupanca";
    contaPoupanca.Taxa = 100.0m;
}
```

## Capítulo 9. Abstração

### Classes Abstratas

A classe abstrata é um tipo de classe que somente pode ser herdada e não instanciada. Pode-se dizer que este tipo de classe é uma classe conceitual que pode definir funcionalidades para que as suas subclasses (classes que herdam desta classe) possam utilizá-las ou implementá-las. Membros em uma classe abstrata declarados como abstratos devem ser obrigatoriamente implementados nas subclasses. A palavra-chave **abstract** é usada para definir classes e membros abstratos.

### Métodos Abstratos

Os métodos abstratos estão presentes somente em classes abstratas e são aqueles que não possuem um corpo com implementação.

**Figura 74 - Exemplo de classe e métodos abstratos.**

```

3 references
public abstract class Veiculo
{
    protected bool MotorLigado;
    protected bool MotorPartidaAcionado;
    protected bool InjetadoCombustivel;

    0 references
    public Veiculo()
    {
        Console.WriteLine("Eu sou um veiculo");
    }

    2 references
    public abstract void Ligar();
    1 reference
    public abstract void Acelerar();
    1 reference
    public abstract void Frear();
}

```

Para sobrescrever os métodos da classe base em uma subclasse é necessário utilizarmos a palavra-chave **override**.

**Figura 75 - Exemplo do uso de override.**

```
public class Carro : Veiculo
{
    0 references
    public Carro()
    {
        Console.WriteLine("Eu sou um carro");
    }

    1 reference
    public override void Acelerar()
    {
        throw new NotImplementedException();
    }

    1 reference
    public override void Frear()
    {
        throw new NotImplementedException();
    }

    1 reference
    public override void Ligar()
    {
        InjetadoCombustivel = true;
        MotorLigado = true;
        Console.WriteLine("O carro esta ligado.");
    }
}
```

Por meio da palavra-chave **override** (que apenas pode ser utilizada em métodos e atributos virtuais) sobrescrevemos o método **Ligar**, **Acelerar** e **Frear**, criado na classe **Carro**. Um método abstrato é implicitamente virtual

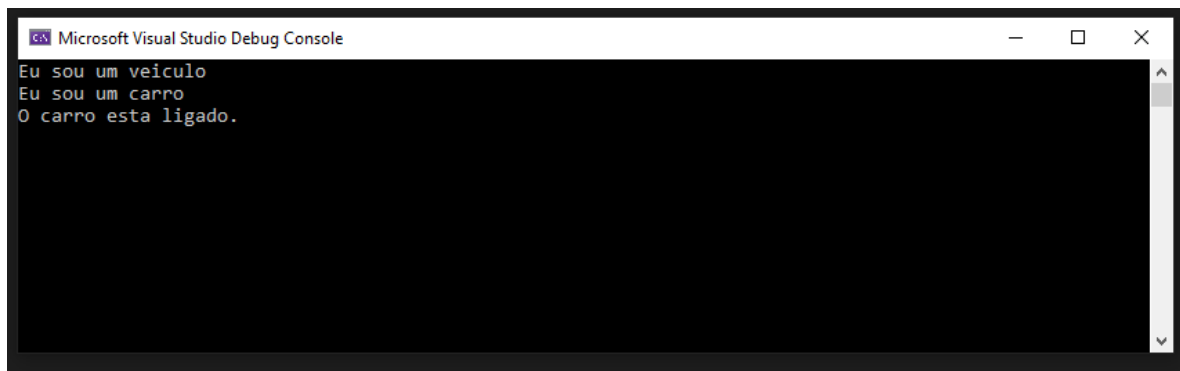
Para identificar o comportamento do método que foi implementado na subclasse podemos codificar da seguinte forma:

**Figura 76 - Exemplo de classe Main.**

```
0 references
static void Main(string[] args)
{
    Veiculo carro = new Carro();
    carro.Ligar();
}
```

Executando o projeto, o console imprime o seguinte resultado:

**Figura 77 - Exemplo de execução.**



Após a execução do método **carro.Ligar()**, podemos observar que a saída corresponde à implementação da subclasse **Carro**, dessa forma o comportamento da superclasse foi sobrescrito. Esse processo acontece em tempo de execução e é um típico comportamento de polimorfismo ao qual veremos no próximo capítulo.

## Membros Virtuais

A palavra-chave **virtual** é usada para modificar uma declaração de método, propriedade, indexador ou evento e permitir que ela seja substituída em uma classe derivada. No exemplo abaixo temos um método virtual que apresenta uma funcionalidade e este pode ser totalmente substituível nas classes herdeiras.

Figura 78 - Exemplo de classe pai com método virtual.

```
1 reference
public class Heroi
{
    2 references
    public virtual void MostrarMinhaHabilidade()
    {
        Console.WriteLine("Eu tenho superforça");
    }
}
```

Figura 79 - Exemplo de classe filha com método virtual.

```
0 references
public class HomemAranha : Heroi
{
    2 references
    public override void MostrarMinhaHabilidade()
    {
        base.MostrarMinhaHabilidade();

        Console.WriteLine("Eu tenho agilidade");
    }
}
```

Figura 80 - Exemplo de classe Main.

```
0 references
static void Main(string[] args)
{
    Heroi heroi = new HomemAranha();
    heroi.MostrarMinhaHabilidade();
}
```

**Figura 81 - Exemplo de execução.**



Dessa forma os membros marcados como virtuais também permitem uma reimplementação característica para a finalidade da funcionalidade.

## Classes Parciais

Vamos pensar em um cenário onde a equipe de desenvolvimento está trabalhando em um projeto e dois programadores precisam trabalhar na classe `Cliente`. Caso o cenário seja muito complexo e difícil de realizar uma junção posterior, cada programador pode trabalhar em um arquivo separado, declarando o modificador ***partial*** que o compilador saberá que se trata da mesma classe e irá tratar como uma só.

**Figura 82 - Exemplo de classe parcial.**

```
1 reference
public partial class Cliente
{
    0 references
    public int Identificador { get; set; }
    1 reference
    public string Endereco { get; private set; }
}

1 reference
public partial class Cliente
{
    0 references
    public void AtualizaEndereco(string novoEndereco)
    {
        Endereco = novoEndereco;
    }
}
```

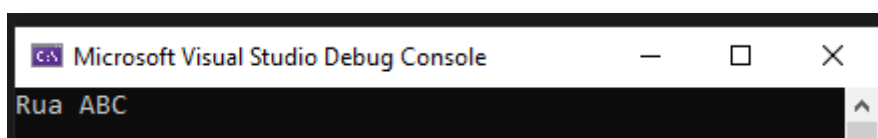
Elas se tornarão uma só com as duas propriedades e o método.

**Figura 83 - Exemplo de método Main.**

```
0 references
static void Main(string[] args)
{
    Cliente cliente = new Cliente();
    cliente.Identificador = 12345;
    cliente.AtualizaEndereco("Rua ABC");

    Console.WriteLine(cliente.Endereco);
}
```

**Figura 84 - Exemplo de execução.**



## Classes Seladas

A classe selada, por sua vez, impede que outras classes herdem dela. É o oposto da abstrata, onde precisa de herdeiros para fazer sentido. Pensamos em uma aplicação onde temos várias classes gerando uma hierarquia de heranças. Uma Superclasse que é herdada por uma subclasse (filha) que é herdada por uma terceira... podemos ir longe com isso. Para que isso não ocorra, marcamos a classe com o modificador **sealed**. Chamamos uma classe selada de classe pronta, pois não será herdada e não terá sobrescrita de métodos.



Figura 85 - Exemplo de classe selada.

```

1 reference
public class Cliente
{
    1 reference
    public string Endereco { get; private set; }
    0 references
    public void AtualizaEndereco(string novoEndereco)
    {
        Endereco = novoEndereco;
    }
}

0 references
public sealed class ClienteEspecial : Cliente
{
    0 references
    public string Bonus { get; set; }
}

```

Figura 86 - Exemplo de erro em herança com classe selada.

```

0 references
public class NovoClienteEspecial : ClienteEspecial
{
    0 references
    public string Limite { get; set; }
}

```

class FundamentosCSharp.ClienteEspecial

CS0509: 'NovoClienteEspecial': cannot derive from sealed type 'ClienteEspecial'

Show potential fixes (Alt+Enter or Ctrl+.)

**Observação:** Mensagem de erro informa que a derivação não é permitida.

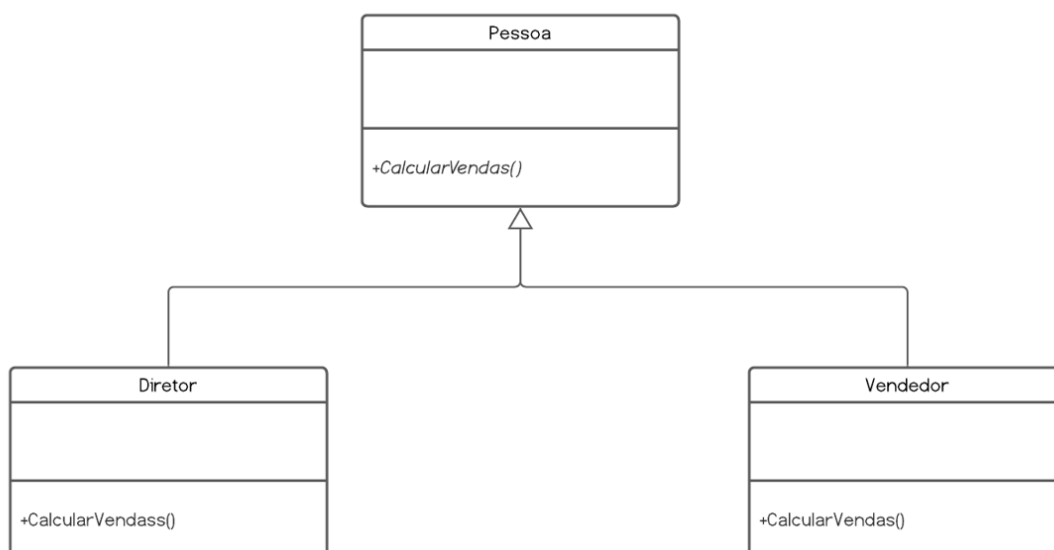
## Capítulo 10. Polimorfismo

Definimos Polimorfismo como um princípio a partir do qual as classes derivadas de uma única classe base são capazes de invocar os métodos que, embora apresentem a mesma assinatura, comportam-se de maneira diferente para cada uma das classes derivadas.

O Polimorfismo é um mecanismo por meio do qual selecionamos as funcionalidades utilizadas de forma dinâmica por um programa no decorrer de sua execução. Com o Polimorfismo, os mesmos atributos e objetos podem ser utilizados em objetos distintos, porém, com implementações lógicas diferentes.

Podemos dizer que uma classe chamada **Vendedor** e outra chamada **Diretor** podem ter como base uma classe chamada **Pessoa**, com um método chamado **CalcularVendas**. Se esse método (definido na classe base) se comportar de maneira diferente para as chamadas feitas a partir de uma instância de Vendedor e para as chamadas feitas a partir de uma instância de Diretor, ele será considerado um método polimórfico, ou seja, um método de várias formas.

**Figura 87 - Representação de herança e polimorfismo.**



Assim podemos ter na classe **Vendedor** o método **CalcularVendas**:

**Figura 88 - Exemplo do método CalcularVendas na classe Vendedor.**

```
0 references
public decimal CalcularVendas()
{
    decimal valorUnitario = 50;

    decimal produtosVendidos = 1500;

    return valorUnitario * produtosVendidos;
}
```

E podemos ter na classe **Diretor** o método **CalcularVendas** da seguinte forma:

**Figura 89 - Exemplo do método CalcularVendas na classe Diretor.**

```
0 references
public decimal CalcularVendas()
{
    decimal valorUnitario = 150;

    decimal produtosVendidos = 3800;

    decimal taxaAdicional = 100;

    return taxaAdicional + (valorUnitario * produtosVendidos);
}
```

## Polimorfismo com Herança

---

No polimorfismo a utilização da herança permite que os membros da classe base possam ser sobrescritos, possibilitando a vinculação dinâmica. A máquina virtual (CLR) determina qual método deve ser chamado em tempo de execução. Este procedimento é realizado através da palavra-chave virtual adicionada ao método, habilitando assim a sobrescrita em classes derivadas.

## Polimorfismo com Classe Abstrata

---

Sabemos que não é possível extrair uma instância de uma classe abstrata, então recursos como interfaces e heranças imprimem a aplicação de programação polimórfica, que por sua vez é feita através do uso de classes abstratas onde os membros marcados como abstratos forçam a implementação.

As classes abstratas são diferentes das interfaces, pois além de fornecer a assinatura para os métodos abstratos que devem ser implementados, outros membros podem ser implementados (não marcados como abstratos). Esse tipo de classe pode incluir outros membros de dados, como propriedades, eventos e métodos. Os métodos podem ser marcados como virtuais ou como abstratos, ou ser completamente implementados.

## Polimorfismo com Interfaces

---

A mesma interface pode ser implementada por várias classes ou uma classe pode implementar várias interfaces. A interface representa um contrato descrevendo métodos e tipos de parâmetros que cada membro do método precisa receber e retornar, a implementação ou o corpo ficam para as classes que implementam a interface definida.

Assim, cada classe que implementa a interface pode assumir um comportamento completamente diferente. A assinatura do método é a mesma para todas as classes, diferindo apenas a implementação.

## Capítulo 11. Interfaces

---

As interfaces são muito parecidas com classes abstratas e compartilham o fato de que nenhuma instância delas pode ser criada. No entanto, as interfaces são ainda mais conceituais do que as classes abstratas, uma vez que nenhum corpo de método é permitido. Portanto, uma interface é como uma classe abstrata com nada além de métodos abstratos e, como não há métodos com código real, não há necessidade de campos.

Entretanto, em interfaces as propriedades são permitidas, bem como indexadores e eventos. Uma interface pode ser vista como um contrato, onde uma classe que a implementa é necessária para implementar todos os métodos e propriedades. A diferença mais importante é que, embora o C# não permita herança múltipla, é permitido a implementação de várias interfaces.

### Trabalhando com Interfaces

---

Vejamos agora um exemplo claro do uso de interfaces:

Figura 90 - Exemplo de interface.

```

1 reference
interface IAnimal
{
    3 references
    string Nome { get; set; }
    1 reference
    string QuemSouEu();
}

1 reference
public class Cachorro: IAnimal
{
    private string nome;

    3 references
    public string Nome
    {
        get { return nome; }
        set { nome = value; }
    }

    0 references
    public Cachorro(string nome)
    {
        this.Nome = nome;
    }

    1 reference
    public string QuemSouEu()
    {
        return "Ola, eu sou um cachorro e meu nome é " + this.Nome;
    }
}

```

Vamos começar onde declaramos a interface. Como você pode ver, a única diferença de uma declaração de classe é a palavra-chave usada, **interface** ao invés de **class**. O nome da interface é prefixado com a letra I - este é simplesmente um padrão de codificação, e não um requisito.

Você pode dar qualquer nome para suas interfaces, mas como elas são usadas como classes em seu código, é interessante usar o prefixo I para diferenciá-las. No exemplo, temos a interface **IAnimal**.

Declaramos a propriedade **Nome**, que possui as palavras-chave **get** e **set**, tornando-a uma propriedade de leitura e escrita. Você também notará a falta de modificadores de acesso (público, privado, protegido etc.), porque eles não são permitidos em uma interface. Todos são públicos por padrão. Também declaramos o método **QuemSouEu**.

A implementação dos membros é feita por uma **classe concreta** ou **struct** que implementa a interface. Na linguagem C#, a sintaxe usada para indicar a utilização de uma interface é colocar dois pontos após o nome da classe concreta que vai implementar a interface seguido do nome da interface. No exemplo acima temos **Cachorro : IAnimal** (**Cachorro = classe concreta** **IAnimal = interface**).

## Capítulo 12. Listas

### O que são listas?

C# tem uma variedade de classes para lidar com listas. Eles implementam a interface **ICollection** e a implementação mais popular é a lista genérica, frequentemente referida como **List<T>**. O **T** especifica o tipo de objetos contidos na lista, que tem o benefício adicional de que o compilador irá verificar e certificar-se de que você apenas adiciona objetos do tipo correto à lista.

**List** é muito parecido com a classe **ArrayList**, que era a escolha preferida de List antes das listas genéricas compatíveis com C#. Portanto, você também verá que List pode fazer muitas das mesmas coisas que um Array (que também implementa a interface ICollection), mas em muitas situações, List é mais simples e fácil de trabalhar. Por exemplo, você não precisa criar uma lista com um tamanho específico - em vez disso, você pode apenas criá-la e o .NET irá expandi-la automaticamente para caber na quantidade de itens conforme você os adiciona.

Conforme mencionado, o **T** significa tipo e é usado para especificar o tipo de objetos que você deseja que a lista contenha. Nosso primeiro exemplo mostra como criar uma lista que deve conter strings:

**Figura 91 - Criação de lista do tipo String.**

```
List<string> listaDeStrings = new List<string>();
```

Isso cria uma lista vazia, mas é muito fácil adicionar coisas a ela com o método Add:

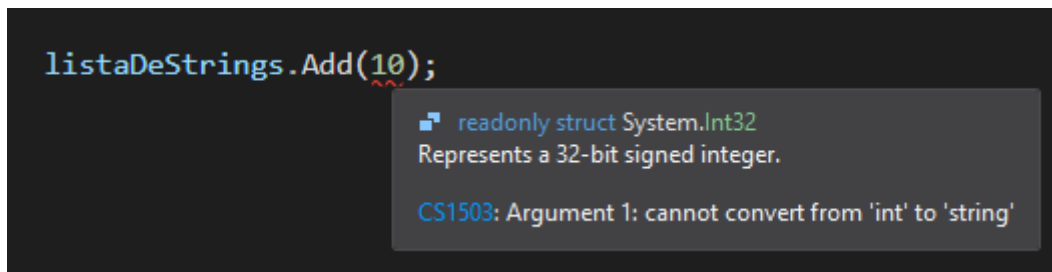
**Figura 92 - Adicionando elemento a lista.**

```
listaDeStrings.Add("Um novo item");
```



Ao tentar adicionar algo que não seja uma string, o compilador irá apresentar imediatamente um erro sobre isso:

**Figura 93 - Erro ao adicionar um inteiro numa lista do tipo String.**



## Inicializando Listas

No exemplo acima acabamos de criar uma lista e, em seguida, adicionamos um item a ela. No entanto, o C# permite que você crie uma lista e adicione itens a ela dentro da mesma instrução, usando uma técnica chamada inicializadores de coleção. Vamos ver como isso é feito:

**Figura 94 - Declaração e inicialização de lista.**

```
List<string> listaDeLinguagens = new List<string>{ "C#", "Javascript", "Python" };
```

A sintaxe é bastante simples: antes do ponto e vírgula final usual, temos um conjunto de chaves, que por sua vez contém uma lista dos valores que queremos apresentar na lista desde o início. Visto que esta é uma lista de strings, os objetos iniciais que fornecemos devem, é claro, ser do tipo string. No entanto, o mesmo pode ser feito para uma lista de outros tipos, mesmo que estejamos usando nossas próprias classes, como veremos na seção “Manipulando Listas”.

## Manipulando Listas

Existem várias maneiras de trabalhar com os itens de uma lista genérica e, para mostrar uma delas, veja o exemplo:

**Figura 95 - Criação e inicialização de lista do tipo `Usuario`.**

```

5 references
public class Usuario
{
    4 references
    public string Nome { get; set; }
    4 references
    public int Idade { get; set; }
}

0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        List<Usuario> usuarios = new List<Usuario>()
        {
            new Usuario() { Nome = "Renato Silva", Idade = 42 },
            new Usuario() { Nome = "Joana dos Santos", Idade= 34 },
            new Usuario() { Nome = "Pedro Garcia", Idade= 8 },
        };

        for (int i = 0; i < usuarios.Count; i++)
        {
            Console.WriteLine(usuarios[i].Nome + " tem " + usuarios[i].Idade + " anos");
        }
        Console.ReadKey();
    }
}

```

Já tentamos adicionar um único item a uma lista, mas existem mais opções para fazer isso. Em primeiro lugar, você pode inserir um item em vez de adicioná-lo através do método ***Insert***. A diferença é que enquanto o método ***Add*** sempre adiciona ao final da lista, o método ***Insert*** permite que você insira um item em uma posição específica.

**Figura 96 - Inserindo elementos na lista.**

```
List<string> catalogoDeNomes = new List<string>()
{
    "Felipe Couto"
};

// Insere no topo (index 0)
catalogoDeNomes.Insert(0, "Barbara Pereira");
// Insert no meio (index 1)
catalogoDeNomes.Insert(1, "Gustavo Alves");
```

Começamos a lista com apenas um item, em seguida inserimos mais dois itens, primeiro no topo da lista e depois no meio. O primeiro parâmetro do método Insert é o índice onde queremos inserir o item. Porém, tenha cuidado - uma exceção será lançada se você tentar inserir um item no índice 3, se a lista tiver menos itens!

Existem uma gama de possibilidades de formas que podemos manipular as listas e você pode verificar a documentação da Microsoft para conhecê-las.

## Capítulo 13. Membros Estáticos

Em C#, estático significa algo que não pode ser instanciado. Não é possível criar um objeto de uma classe estática e não se pode acessar membros estáticos usando um objeto.

Classes, variáveis, métodos, propriedades, operadores, eventos e construtores C# podem ser definidos como estáticos usando a palavra-chave **static**.

### Classes Estáticas

Aplique o modificador estático - static - depois do modificador de acesso e antes do nome da classe para torná-la estática. O seguinte define uma classe estática com campos e métodos estáticos.

**Figura 97 - Exemplo de classe estática.**

```

4 references
public static class Calculadora
{
    private static int ResultStorage = 0;

    public static string Tipo = "Aritmética";

    1 reference
    public static int Soma(int num1, int num2)
    {
        return num1 + num2;
    }

    1 reference
    public static void Armazenar(int resultado)
    {
        ResultStorage = resultado;
    }

    0 references
    public static void Imprimir()
    {
        Console.WriteLine(ResultStorage);
    }
}

```

A classe Calculadora mostrada acima é estática. Todos os membros dela também são estáticos. Você não pode criar um objeto da classe estática; portanto, os membros da classe estática podem ser acessados diretamente usando um nome de classe como **NomeDaClasse.NomeDoMembro**, conforme mostrado abaixo:

**Figura 98 - Exemplo de acesso aos métodos de uma classe estática.**

```
0 references
static void Main(string[] args)
{
    var resultado = Calculadora.Soma(10, 25); // chamando método estático.
    Calculadora.Armazenar(resultado);

    var tipoCalculadora = Calculadora.Tipo; //acessando variável estático.
    Calculadora.Tipo = "Geografia";        // atribuir valor para uma variavel estática.
}
```

### Regras para classe estática:

- Classes estáticas não podem ser instanciadas.
- Todos os membros de uma classe estática devem ser estáticos; caso contrário, o compilador fornecerá um erro.
- Uma classe estática pode conter variáveis estáticas, métodos estáticos, propriedades estáticas, operadores estáticos, eventos estáticos e construtores estáticos.
- Uma classe estática não pode conter membros e construtores de instância.
- Indexadores e destruidores não podem ser estáticos.
- var não pode ser usado para definir membros estáticos. Você deve especificar um tipo de membro explicitamente após a palavra-chave estática.
- Classes estáticas são classes lacradas e, portanto, não podem ser herdadas.
- Uma classe estática não pode herdar de outras classes.
- Os membros da classe estática podem ser acessados usando NomeDaClasse.NomeDoMembro.

- Uma classe estática permanece na memória durante o tempo de vida do domínio do aplicativo no qual o programa reside.

## Campos Estáticos

---

Os campos estáticos em uma classe não estática podem ser definidos usando a palavra-chave **static**. Os campos estáticos de uma classe não estática são compartilhados por todas as instâncias. Portanto, as alterações feitas por uma instância refletem nas outras.

**Figura 99 - Exemplo de campo estático em classe não estática.**

```
1 reference
public class Contador
{
    public static int contadorInterno = 0;

    0 references
    public Contador()
    {
    }
}
```

## Métodos Estáticos

---

Pode-se definir um ou mais métodos estáticos em uma classe não estática. Os métodos estáticos podem ser chamados sem criar um objeto. Não é possível chamar métodos estáticos usando um objeto da classe não estática.

Os métodos estáticos só podem chamar outros métodos estáticos e acessar membros estáticos. Não é possível acessar membros não estáticos da classe nos métodos estáticos.

**Figura 100 - Exemplo de método estático.**

```
static int contador = 0;
string nome = "Qualquer Nome";

0 references
static void Main(string[] args)
{
    contador++; // Podem acessar campos estáticos.
    Exibir("Fundamentos CSharp"); // Podem chamar métodos estáticos.

    nome = "New Demo Program"; //Erro: não pode acessar mebros nao-estáticos.
    OutroMetodoExibir("Fundamentos C#"); //Erro: não pode chamar métodos nao-estáticos.
}

1 reference
static void Exibir(string mensagem)
{
    Console.WriteLine(mensagem);
}

1 reference
public void OutroMetodoExibir(string mensagem)
{
    Console.WriteLine(mensagem);
}
```

### Regras para métodos estáticos:

- Os métodos estáticos podem ser definidos usando a palavra-chave static antes de um tipo de retorno e depois de um modificador de acesso.
- Os métodos estáticos podem ser sobrecarregados, mas não podem ser substituídos.
- Os métodos estáticos podem conter variáveis estáticas locais.
- Os métodos estáticos não podem acessar ou chamar variáveis não estáticas, a menos que sejam explicitamente passados como parâmetros.

### Construtores Estáticos

Uma classe não estática pode conter um construtor estático sem parâmetros. Ele pode ser definido com a palavra-chave estática e sem modificadores de acesso como público, privado e protegido.

## Capítulo 14. Exceções

Exceções na aplicação devem ser tratadas para evitar que o programa trave ou que haja resultado inesperado. C# fornece suporte integrado para lidar com a exceção usando blocos **try**, **catch** e **finally**.

**Figura 101 - Estrutura para tratativa de exceção.**

```
try
{
    // coloque o código aqui que pode gerar exceções.
}
catch
{
    // trata a exceção aqui.
}
finally
{
    // código de limpeza final
}
```

Bloco **try**: Qualquer código suspeito que possa levantar exceções deve ser colocado dentro de um bloco try {}. Durante a execução, se ocorrer uma exceção, o fluxo do controle salta para o primeiro bloco catch correspondente.

Bloco **catch**: O bloco catch é um bloco manipulador de exceção onde você pode executar alguma ação, como registrar e auditar uma exceção. O bloco catch recebe um parâmetro de um tipo de exceção com o qual você pode obter os detalhes de uma exceção.

Bloco **finally**: O bloco finally sempre será executado, independentemente de haver ou não uma exceção. Normalmente, um bloco finally deve ser usado para liberar recursos, por exemplo, para fechar qualquer fluxo ou objetos de arquivo que foram abertos no bloco try.

O seguinte trecho pode lançar uma exceção se você inserir um caractere não-numérico.



**Figura 102 - Exemplo de tratativa da exceção.**

```
try
{
    Console.WriteLine("Enter um número: ");

    var numero = int.Parse(Console.ReadLine());

    Console.WriteLine($"Quadrado de {numero} é {numero * numero}");
}
catch
{
    Console.WriteLine("Um erro ocorreu.");
}
finally
{
    Console.WriteLine($"Tente um número diferente.");
}
```

No exemplo acima, envolvemos esse código dentro de um bloco try. Se ocorrer uma exceção dentro de um bloco try, o programa irá pular para o bloco catch. Dentro de um bloco catch, exibimos uma mensagem para instruir o usuário sobre seu erro e no bloco finally exibimos uma mensagem sobre o que fazer após executar um programa.

## Referências

---

C SHARP. In: **WIKIPÉDIA**, a enciclopédia livre. Flórida: Wikimedia Foundation, 2021. Disponível em: [https://pt.wikipedia.org/w/index.php?title=C\\_Sharp&oldid=61871023](https://pt.wikipedia.org/w/index.php?title=C_Sharp&oldid=61871023). Acesso em: 18 out. 2021.

C# - Static Class, Methods, Constructors, Fields. **Tutorials Teacher**, 28 jun. 2020. Disponível em: <https://www.tutorialsteacher.com/csharp/csharp-static>. Acesso em: 18 out. 2021.

C# - Struct. **Tutorials Teacher**, 25 jun. 2020. Disponível em: <https://www.tutorialsteacher.com/csharp/csharp-struct>. Acesso em: 18 out. 2021.

C# Enum. **W3 Schools**, c1999-2021. Disponível em: [https://www.w3schools.com/cs/cs\\_enums.php](https://www.w3schools.com/cs/cs_enums.php). Acesso em: 18 out. 2021.

COLLECTIONS: Lists. **C# Tutorial**, c2007-2021. Disponível em: <https://csharp.net-tutorials.com/collections/lists/>. Acesso em: 18 out. 2021.

DANTAS, Cleber. Série aprenda C# - Estrutura de Repetição. **Linha de Código**, c2021. Disponível em: <http://www.linhadecodigo.com.br/artigo/1177/serie-aprenda-csharp-estrutura-de-repeticao.aspx>. Acesso em: 18 out. 2021.

EXCEPTION Handling in C#. **Tutorials Teacher**, c2020. Disponível em: <https://www.tutorialsteacher.com/csharp/csharp-exception-handling>. Acesso em: 18 out. 2021.

SILVA, Marcos César. Linguagem C# - Criando Classes e Objetos. **The Club**. Disponível em: <http://theclub.com.br/Restrito/Revistas/201208/ling1208.aspx>. Acesso em: 18 out. 2021.

VÁSQUEZ, Tomás. Introdução. **Curso Online de C# .NET**, 10 out. 2009. Disponível em: [http://www.tomasvasquez.com.br/cursocsharp/programacao\\_orientada\\_objetos\\_ne/t/introducao-2/](http://www.tomasvasquez.com.br/cursocsharp/programacao_orientada_objetos_ne/t/introducao-2/). Acesso em: 18 out. 2021.