



## UNIDADE 1 – LÓGICA DE PROGRAMAÇÃO

---

Nesta primeira unidade faremos uma revisão em lógica de programação. Aprenderemos o que é um programa, como criamos um programa em C# e entre outros assuntos relacionados a programação.



## SUMÁRIO

---

UNIDADE 1 – lógica de programação .....	1
1 O que é um Programa?.....	3
1.1 Linguagem de Máquina .....	3
1.2 Linguagem de Programação .....	4
1.3 Compilador .....	5
1.4 Máquinas Virtuais.....	5
1.5 Exemplo de programa C# .....	8
1.6 Método Main - Ponto de Entrada.....	9
1.7 Exercícios de Fixação .....	10
1.8 Variáveis .....	11
1.8.1 Declaração .....	11
1.8.2 Inicialização .....	12
1.8.3 Tipos Primitivos .....	13
1.9 Operadores.....	14
1.9.1 Aritmético.....	14
1.9.2 Atribuição .....	15
1.9.3 Relacional .....	16
1.9.4 Lógico .....	17
1.10 IF-ELSE .....	17
1.11 WHILE .....	18
1.12 FOR .....	18
1.13 Exercícios de Fixação .....	19
1.14 Exercícios Complementares .....	24
1.15 A “Arte” de Debugar .....	26
1.15.1 Depuração .....	27
1.15.2 Immediate Window .....	27



# 1 O QUE É UM PROGRAMA?

---

Um dos maiores benefícios da utilização de computadores é a automatização de processos realizados manualmente por pessoas. Vejamos um exemplo prático:

Quando as apurações dos votos das eleições no Brasil eram realizadas manualmente, o tempo para obter os resultados era alto e havia alta probabilidade de uma falha humana. Esse processo foi automatizado e hoje é realizado por computadores. O tempo para obter os resultados e a chance de ocorrer uma falha humana diminuíram drasticamente.

Basicamente, os computadores são capazes de executar instruções matemáticas mais rapidamente do que o homem. Essa simples capacidade permite que eles resolvam problemas complexos de maneira mais eficiente. Porém, eles não possuem a inteligência necessária para definir quais instruções devem ser executadas para resolver uma determinada tarefa. Por outro lado, os seres humanos possuem essa inteligência. Dessa forma, uma pessoa precisa definir um roteiro com a sequência de comandos necessários para realizar uma determinada tarefa e depois passar para um computador executar esse **roteiro**. Formalmente, esses roteiros são chamados de **programas**.

Os programas devem ser colocados em arquivos no disco rígido dos computadores. Assim, quando as tarefas precisam ser realizadas, os computadores podem ler esses arquivos para saber quais instruções devem ser executadas.

## 1.1 LINGUAGEM DE MÁQUINA

Os computadores só sabem ler instruções escritas em **linguagem de máquina**. Uma instrução escrita em linguagem de máquina é uma sequência formada por “0s” e “1s” que representa a ação que um computador deve executar.



Figura 1: Código de máquina



Teoricamente, as pessoas poderiam escrever os programas diretamente em **linguagem de máquina**. Na prática, ninguém faz isso pois é uma tarefa muito complicada e demorada.

Um arquivo contendo as instruções de um programa em Linguagem de Máquina é chamado de **executável**.

## 1.2 LINGUAGEM DE PROGRAMAÇÃO

Como vimos anteriormente, escrever um programa em linguagem de máquina é totalmente inviável para uma pessoa. Para resolver esse problema, surgiram as linguagens de programação, que tentam se aproximar das linguagens humanas. Confira um trecho de um código escrito com a linguagem de programação C#:

```
1  class Program
2  {
3      References
4      static void Main(string[] args)
5      {
6          Console.WriteLine("Primeiro programa em C#! Aizá C# veio!!!");
7      }
8  }
```

Tabela 1: Código em C#

Por enquanto você pode não entender muito do que está escrito, porém fica bem claro que um programa escrito dessa forma fica bem mais fácil de ser lido.

Um arquivo contendo as instruções de um programa em linguagem de programação é chamado de **arquivo fonte**.



## 1.3 COMPILADOR

Por um lado, os computadores processam apenas instruções em linguagem de máquina. Por outro lado, as pessoas definem as instruções em linguagem de programação. Dessa forma, é necessário traduzir o código escrito em linguagem de programação por uma pessoa para um código em linguagem de máquina para que um computador possa processar. Essa tradução é realizada por programas especiais chamados compiladores.

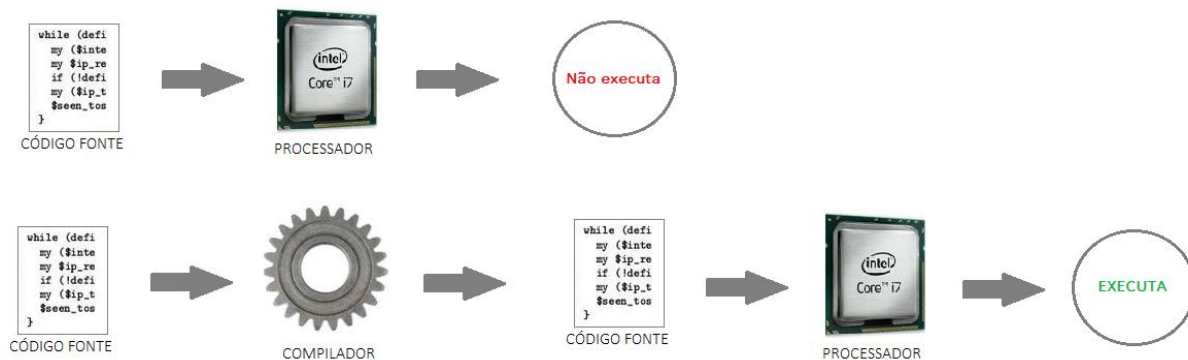


Figura 2: Processo de compilação de um programa

## 1.4 MÁQUINAS VIRTUAIS

Assim como as pessoas podem se comunicar através de línguas diferentes, os computadores podem se comunicar através de linguagens de máquina diferentes. A linguagem de máquina de um computador é definida pela **arquitetura do processador** desse computador. Há diversas arquiteturas diferentes (Intel, ARM, PowerPC, etc) e cada uma delas define uma linguagem de máquina diferente. Em outras palavras, um programa pode não executar em computadores com processadores de arquiteturas diferentes.

Os computadores são controlados por um **sistema operacional** que oferece diversas bibliotecas necessárias para o desenvolvimento das aplicações que podem ser executadas através dele. Sistemas operacionais diferentes (Windows, Linux, Mac OS X, etc) possuem bibliotecas diferentes. Em outras palavras, um programa pode não executar em computadores com sistemas operacionais diferentes.

Portanto, para determinar se um código em linguagem de máquina pode ou não ser executado por um computador, devemos considerar a arquitetura do processador e o sistema operacional desse computador.

Algumas bibliotecas específicas de sistema operacional são chamadas diretamente pelas instruções em linguagem de programação. Dessa forma, geralmente, o código fonte está “amarrado” a uma plataforma (sistema operacional + arquitetura de processador).



Figura 3: Ilustração mostrando que cada plataforma necessita de um executável específico.

Uma empresa que deseja ter a sua aplicação disponível para diversos sistemas operacionais (Windows, Linux, Mac OS X, etc), e diversas arquiteturas de processador (Intel, ARM, PowerPC, etc), terá que desenvolver versões diferentes do código fonte para cada plataforma (sistema operacional + arquitetura de processador). Isso pode causar um impacto financeiro nessa empresa que inviabiliza o negócio.



Para tentar resolver o problema do desenvolvimento de aplicações multiplataforma, surgiu o conceito de máquina virtual. Uma máquina virtual funciona como uma camada a mais entre o código compilado e a plataforma. Quando compilamos um código fonte, estamos criando um executável que a máquina virtual saberá interpretar e ela é quem deverá traduzir as instruções do seu programa para a plataforma.

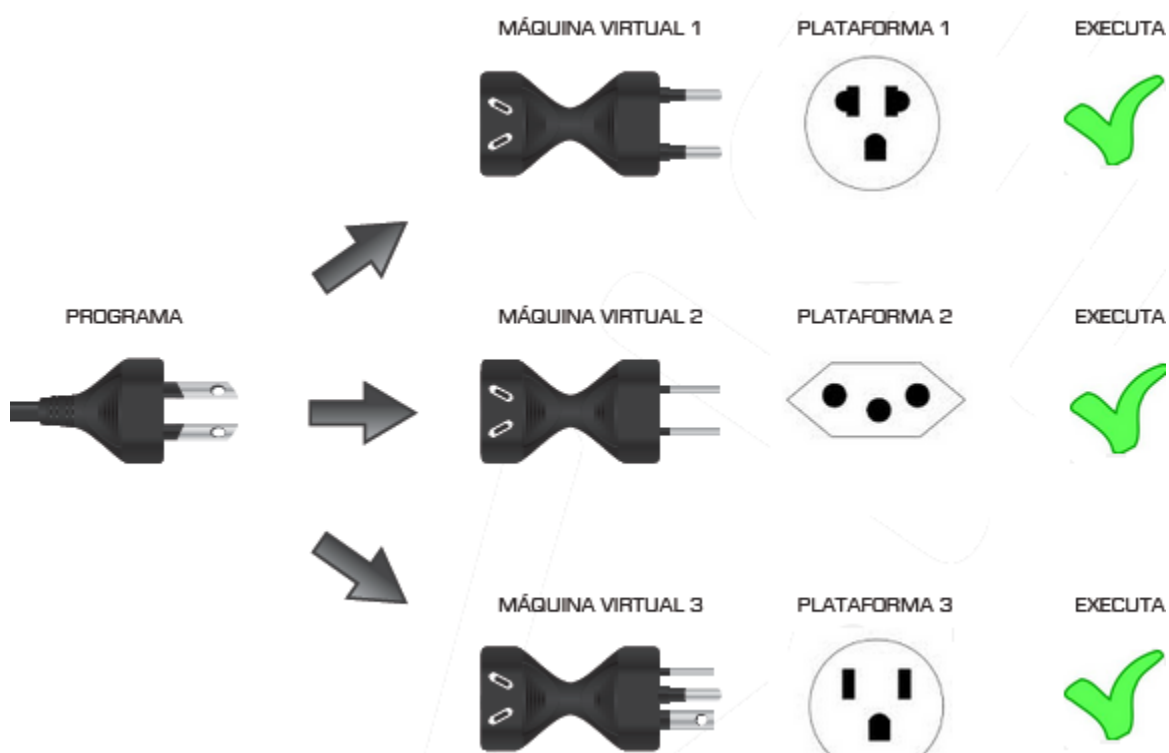


Figura 4: Ilustração do funcionamento da máquina virtual.

Tudo parece estar perfeito agora. Porém, olhando atentamente a figura acima, percebemos que existe a necessidade de uma máquina virtual para cada plataforma. Alguém poderia dizer que, de fato, o problema não foi resolvido, apenas mudou de lugar.

A diferença é que implementar a máquina virtual não é tarefa do programador que desenvolve as aplicações que serão executadas nela. A implementação da máquina virtual é responsabilidade de terceiros, que geralmente são empresas bem conceituadas ou projetos de código aberto que envolvem programadores do mundo inteiro. Como maiores exemplos podemos citar a Microsoft CLR (Common Language Runtime) e Mono CLR.

Uma desvantagem em utilizar uma máquina virtual para executar um programa é a diminuição de performance, já que a própria máquina virtual consome recursos do computador. Além disso, as instruções do programa são processadas primeiro pela máquina virtual e depois pelo computador.

Por outro lado, as máquinas virtuais podem aplicar otimizações que aumentam a performance da execução de um programa. Inclusive, essas otimizações podem considerar informações geradas durante a execução. São exemplos de informações geradas durante a execução: a quantidade de uso da memória RAM e do processador do computador, a quantidade de acessos ao disco rígido, a quantidade de chamadas de rede e a frequência de execução de um determinado trecho do programa. Algumas máquinas virtuais



identificamos trechos do programa que estão sendo mais chamados em um determinado momento da execução para traduzi-los para a linguagem de máquina do computador. A partir daí, esses trechos podem ser executados diretamente no processador sem passar pela máquina virtual. Essa análise da máquina virtual é realizada durante toda a execução. Com essas otimizações que consideram várias informações geradas durante a execução, um programa executado com máquina virtual pode até ser mais eficiente em alguns casos do que um programa executado diretamente no sistema operacional.

### Mais Sobre

Geralmente, as máquinas virtuais utilizam uma estratégia de compilação chamada **Just-in-time compilation (JIT)**. Nessa abordagem, o código de máquina pode ser gerado diversas vezes durante o processamento de um programa com o intuito de melhorar a utilização dos recursos disponíveis em um determinado instante da execução.

## 1.5 EXEMPLO DE PROGRAMA C#

Vamos criar um simples programa para entendermos como funciona o processo de compilação e execução. Utilizaremos a linguagem C#, que é amplamente adotada nas empresas, inclusive na NDDigital. Observe o código do exemplo de um programa escrito em C# que imprime uma mensagem na tela:

```
1  class OlaAcademia
2  {
3      // References
4      private static void Main(string[] args)
5      {
6          Console.WriteLine("Ola academia do programador!");
7      }
8  }
```

Tabela 1: Primeiro programa C#.

O código fonte C# deve ser colocado em arquivos com a extensão **.cs**. Agora, não é necessário entender todo o código do exemplo. Basta saber que toda aplicação C# precisa ter um método especial chamado **Main** para executar.

O próximo passo é compilar o código fonte, para gerar um executável que possa ser processado pela máquina virtual do .NET. O compilador padrão da plataforma .NET (**csc**) pode ser utilizado para compilar esse arquivo. O compilador pode ser executado pelo **terminal**.

```
c:\CSharp>C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe OlaAcademia.cs
Microsoft (R) Visual C# Compiler version 4.0.30319.18408
for Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.
```

Figura 5: Executar o compilador padrão da plataforma .NET





O código gerado pelo compilador .NET é armazenado em arquivos com a extensão .exe. No exemplo, o programa gerado pelo compilador é colocado em um arquivo chamado OlaMundo.exe e ele pode ser executado através de um terminal.

```
c:\CSharp>OlaAcademia.exe
Sejam bem vindos a Academia de Programação 2014!
```

Figura 6: Executando o primeiro programa em C#.

## Importante

Para compilar e executar um programa escrito em C#, é necessário que você tenha instalado e configurado em seu computador uma máquina virtual .NET. Versões mais recentes do Windows já possuem uma máquina virtual .NET instalada.

## 1.6 MÉTODO MAIN - PONTO DE ENTRADA

Para um programa C# executar, é necessário definir um método especial para ser o ponto de entrada do programa, ou seja, para ser o primeiro método a ser chamado quando o programa for executado. O método Main precisa ser **static** e seu tipo de retorno pode ser **void** ou **int**. Ele também pode declarar parâmetros para receber os argumentos passados pela linha de comando e deve ser inserido em uma classe C#.

Algumas das possíveis variações da assinatura do método Main:

```
1 static void Main ()
2 static int Main ()
3 static void Main ( string [] args )
4 static int Main ( string [] args )
```

Tabela 2: Variações da Assinatura do Método Main

Os parâmetros do método Main são passados pela linha de comando e podem ser manipulados dentro do programa. O código abaixo imprime cada parâmetro recebido em uma linha diferente.

```
1 class Program
2 {
3     References
4     private static void Main(string[] args)
5     {
6         for (int i = 0; i < args.Length; i++)
7         {
8             Console.WriteLine(args[i]);
9         }
10    }
11 }
```

Tabela 3: Imprimindo os parâmetros da linha de comando



Os parâmetros devem ser passados imediatamente após o nome do programa. A compilação e execução do programa é mostrada na figura abaixo.

```
c:\CSharp>C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe args.cs
Microsoft (R) Visual C# Compiler version 4.0.30319.18408
for Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.

c:\CSharp>args.exe "Thiago" "Sartor"
Thiago
Sartor
```

Figura 7: Imprimindo os parâmetros da linha de comando

## 1.7 EXERCÍCIOS DE FIXAÇÃO

- 1) Abra um terminal e crie uma pasta com o seu nome. Você deve salvar os seus exercícios nessa pasta.

```
c:\>md Thiago
```

Figura 8: Criando pasta de exercícios.

- 2) Dentro da sua pasta de exercícios, crie uma pasta para os arquivos desenvolvidos nesse capítulo chamada **ExerciciosFixacao**.

```
c:\Thiago>md ExercicioFixacao
```

Figura 9: Criando a pasta dos exercícios desse capítulo

- 3) Crie um programa que imprima uma mensagem na tela. Adicione o seguinte arquivo na pasta logica.

```
1 class OlaAcademia
2 {
3     References
4     private static void Main(string[] args)
5     {
6         Console.WriteLine("Ola academia do programador!");
7     }
8 }
```

Tabela 4: OlaAcademia.cs

Compile e execute a classe OlaAcademia.cs



## 1.8 VARIÁVEIS

Basicamente, o que um programa faz é manipular dados. Em geral, esses dados são armazenados em **variáveis** localizadas na memória RAM do computador. Uma variável pode guardar dados de vários tipos: números, textos, booleanos (verdadeiro ou falso), referências de objetos. Além disso, toda variável possui um nome que é utilizado quando a informação dentro da variável precisa ser manipulada pelo programa.

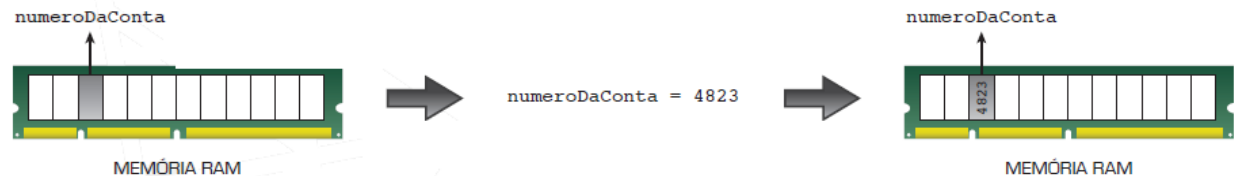


Figura 10: Processo de atribuição do valor numérico 4823 à variável numeroDaConta.

### 1.8.1 Declaração

Na linguagem de programação C#, as variáveis devem ser declaradas para que possam ser utilizadas. A declaração de uma variável envolve definir um nome único (identificador) dentro de um escopo e um tipo de valor. As variáveis são acessadas pelos nomes e armazenam valores compatíveis com o seu tipo.

```
1 // Uma variável do tipo int chamada numeroDaConta .
2 private int numeroDaConta;
3
4 // Uma variável do tipo double chamada precoDoProduto .
5 private double precoDoProduto;
```

Tabela 5: Declaração de Variáveis

### Mais Sobre

Uma linguagem de programação é dita **estaticamente tipada** quando ela exige que os tipos das variáveis sejam definidos antes da compilação. A linguagem C# é uma linguagem estaticamente tipada.

Uma linguagem de programação é dita **fortemente tipada** quando ela exige que os valores armazenados em uma variável sejam compatíveis com o tipo da variável. A linguagem C# é uma linguagem fortemente tipada.

### Mais Sobre

Em geral, as linguagens de programação possuem convenções para definir os nomes das variáveis. Essas convenções ajudam o desenvolvimento de um código mais legível. Na convenção de nomes da linguagem C#, os nomes das variáveis devem seguir o padrão **camel case** com a primeira letra minúscula. Esse padrão também é conhecido como **lower camel case**. Veja alguns exemplos:

- nomeDoCliente
- numeroDeAprovados



A convenção de nomes da linguagem C# pode ser consultada na seguinte url: [http://msdn.microsoft.com/en-us/library/xzf533w0\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/xzf533w0(v=vs.71).aspx).

A declaração de uma variável pode ser realizada em qualquer linha de um bloco. Não é necessário declarar todas as variáveis no começo do bloco como acontece em algumas linguagens de programação.

```
1 // Declaração com Inicialização
2 int numero = 10;
3
4 // Uso da variável
5 Console.WriteLine(numero);
6
7 // Outra Declaração com Inicialização
8 double preco = 137.6;
9
10 // Uso da variável
11 Console.WriteLine(preco);
```

Tabela 6: Declarando em qualquer linha de um bloco.

Não podemos declarar duas variáveis com o mesmo nome em um único bloco ou escopo pois ocorrerá um erro de compilação.

```
1 // Declaração
2 int numero = 10;
3
4 // Erro de Compilação
5 int numero = 10;
```

Tabela 7: Duas variáveis com o mesmo nome no mesmo bloco.

### 1.8.2 Inicialização

Toda variável deve ser inicializada antes de ser utilizada pela primeira vez. Se isso não for realizado, ocorrerá um erro de compilação. A inicialização é realizada através do operador de atribuição =. Esse operador guarda um valor em uma variável.



```
1 // Declarações
2 int numero;
3 double preco;
4
5 // Inicialização
6 numero = 10;
7
8 // Uso Correto
9 Console.WriteLine(numero);
10
11 // Erro de compilação
12 Console.WriteLine(preco);
```

Tabela 8: Inicialização

### 1.8.3 Tipos Primitivos

A linguagem C# define um conjunto de tipos básicos de dados que são chamados **tipos primitivos**. A tabela abaixo mostra os oito tipos primitivos da linguagem C# e os valores compatíveis.

Tipo	Descrição	Tamanho ("peso")
sbyte	Valor inteiro entre -128 e 127 (inclusivo)	1 byte
byte	Valor inteiro entre 0 e 255 (inclusivo)	1 byte
short	Valor inteiro entre -32.768 e 32.767 (inclusivo)	2 bytes
ushort	Valor inteiro entre 0 e 65.535 (inclusivo)	2 bytes
int	Valor inteiro entre -2.147.483.648 e 2.147.483.647 (inclusivo)	4 bytes
uint	Valor inteiro entre 0 e 4.294.967.295 (inclusivo)	4 bytes
long	Valor inteiro entre -9.223.372.036.854.775.808 e 9.223.372.036.854.775.807 (inclusivo)	8 bytes
ulong	Valor inteiro entre 0 e 18.446.744.073.709.551.615 (inclusivo)	8 bytes
float	Valor com ponto flutuante entre $1,40129846432481707 \times 10^{-45}$ e $3,40282346638528860 \times 10^{38}$ (positivo ou negativo)	4 bytes

Tabela 9: Tipos primitivos de dados em C#.



Tipo	Descrição	Tamanho ("peso")
double	Valor com ponto flutuante entre $4,94065645841246544 \times 10^{-324}$ e $1,79769313486231570 \times 10^{308}$ (positivo ou negativo)	8 bytes
decimal	Valor com ponto flutuante entre $1,0 \times 10^{-28}$ e $7,9 \times 10^{28}$ (positivo ou negativo)	16 bytes
bool	true ou false	1 bit
char	Um único caractere Unicode de 16 bits. Valor inteiro e positivo entre 0 (ou '\u0000') e 65.535 (ou '\uffff')	2 bytes

Tabela 10: Tipos primitivos de dados em C#.

### Importante

Nenhum tipo primitivo da linguagem C# permite o armazenamento de texto. O tipo primitivo **char** armazena apenas um caractere. Quando é necessário armazenar um texto, devemos utilizar o tipo **string**. Contudo, é importante salientar que o tipo string **não** é um tipo primitivo.

## 1.9 OPERADORES

Para manipular os valores das variáveis de um programa, devemos utilizar os operadores oferecidos pela linguagem de programação adotada. A linguagem C# possui diversos operadores e os principais são categorizados da seguinte forma:

- Aritmético (+, -, \*, /, %)
- Atribuição (=, +=, -=, \*=, /=, %=)
- Relacional (==, !=, <, <=, >, >=)
- Lógico (&&, ||)

### 1.9.1 Aritmético

Os operadores aritméticos funcionam de forma muito semelhante aos operadores na matemática.

Os operadores aritméticos são:

- Soma +
- Subtração -
- Multiplicação \*
- Divisão /
- Módulo %



```
1  int umMaisUm = 1 + 1;           // umMaisUm = 2
2  int tresVezesDois = 3 * 2;      // tresVezesDois = 6
3  int quatroDivididoPor2 = 4 / 2; // quatroDivididoPor2 = 2
4  int seisModuloCinco = 6 % 5;    // seisModuloCinco = 1
5  int x = 7;
6  x = x + 1 * 2;                  // x = 14
7  x = x - 4;                      // x = 10
8  x = x / (6 - 2 + (3 * 5) / (16 - 1)); // x = 2
```

Tabela 11: Exemplo de uso dos operadores aritméticos.

### Importante

O módulo de um número  $x$ , na matemática, é o valor numérico de  $x$  desconsiderando o seu sinal (valor absoluto). Na matemática expressamos o módulo da seguinte forma:

$$|-2| = 2.$$

Em linguagens de programação, o módulo de um número é o resto da divisão desse número por outro. No exemplo acima, o resto da divisão de 6 por 5 é igual a 1. Além disso, lemos a expressão  $6\%5$  da seguinte forma: seis módulos cinco.

### Importante

As operações aritméticas em C# obedecem as mesmas regras da matemática com relação à precedência dos operadores e parênteses. Portanto, as operações são resolvidas a partir dos parênteses mais internos até os mais externos, primeiro resolvemos as multiplicações, divisões e os módulos. Em seguida, resolvemos as adições e subtrações.

## 1.9.2 Atribuição

Nas seções anteriores, já vimos um dos operadores de atribuição, o operador `=` (igual). Os operadores de atribuição são:

- Simples `=`
- Incremental `+=`
- Decremental `-=`
- Multiplicativa `*=`
- Divisória `/=`
- Modular `%=`



```
1  int valor = 1;      // valor = 1
2  valor += 2;         // valor = 3
3  valor -= 1;         // valor = 2
4  valor *= 6;         // valor = 12
5  valor /= 3;         // valor = 4
6  valor %= 3;         // valor = 1
```

Tabela 12: Exemplo de uso dos operadores de atribuição.

As instruções acima poderiam ser escritas de outra forma:

```
1  int valor = 1;      // valor = 1
2  valor = valor + 2;   // valor = 3
3  valor = valor - 1;   // valor = 2
4  valor = valor * 6;   // valor = 12
5  valor = valor / 3;   // valor = 4
6  valor = valor % 3;   // valor = 1
```

Tabela 13: O mesmo exemplo anterior, usando os operadores aritméticos.

Como podemos observar, os operadores de atribuição, com exceção do simples (=), reduzem a quantidade de código escrito. Podemos dizer que esses operadores funcionam como “atalhos” para as operações que utilizam os operadores aritméticos.

### 1.9.3 Relacional

Muitas vezes precisamos determinar a relação entre uma variável ou valor e outra variável ou valor. Nessas situações, utilizamos os operadores relacionais. As operações realizadas com os operadores relacionais devolvem valores do tipo primitivo bool. Os operadores relacionais são:

- Igualdade ==
- Diferença !=
- Menor <
- Menor ou igual <=
- Maior >
- Maior ou igual >=

```
1  bool t = false;
2  t = (valor == 2); // t = true
3  t = (valor != 2); // t = false
4  t = (valor < 2);  // t = false
5  t = (valor <= 2); // t = true
6  t = (valor > 1);  // t = true
7  t = (valor >= 1); // t = true
```

Tabela 14: Exemplo de uso dos operadores relacionais em C#.





### 1.9.4 Lógico

A linguagem C# permite verificar duas ou mais condições através de operadores lógicos. Os operadores lógicos devolvem valores do tipo primitivo bool. Os operadores lógicos são:

- “E” lógico &&
- “OU” lógico ||

```
1  int valor = 30;
2  bool teste = false;
3  teste = valor < 40 && valor > 20;    // teste = true
4  teste = valor < 40 && valor > 30;    // teste = false
5  teste = valor > 30 || valor > 20;    // teste = true
6  teste = valor > 30 | valor < 20;    // teste = false
7  teste = valor < 50 & valor == 30;   // teste = true
```

Tabela 15: Exemplo de uso dos operadores lógicos em C#.

### 1.10 IF-ELSE

O comportamento de uma aplicação pode ser influenciado por valores definidos pelos usuários. Por exemplo, considere um sistema de cadastro de produtos. Se um usuário tenta adicionar um produto com preço negativo, a aplicação não deve cadastrar esse produto. Caso contrário, se o preço não for negativo, o cadastro pode ser realizado normalmente.

Outro exemplo, quando o pagamento de um boleto é realizado em uma agência bancária, o sistema do banco deve verificar a data de vencimento do boleto para aplicar ou não uma multa por atraso.

Para verificar uma determinada condição e decidir qual bloco de instruções deve ser executado, devemos aplicar o comando **if**.

```
1  if (preco < 0)
2  {
3      Console.WriteLine("O preço do produto não pode ser negativo");
4  }
5  else
6  {
7      Console.WriteLine("Ola academia do programador!");
8  }
```

Tabela 16: Comando If

O comando **if** permite que valores booleanos sejam testados. Se o valor passado como parâmetro para o comando **if** for true, o bloco do **if** é executado. Caso contrário, o bloco do **else** é executado.

O parâmetro passado para o comando **if** deve ser um valor booleano, caso contrário o código não compila. O comando **else** e o seu bloco são opcionais.



## 1.11 WHILE

Em alguns casos, é necessário repetir um trecho de código diversas vezes. Suponha que seja necessário imprimir 10 vezes na tela a mensagem: “Bom Dia”. Isso poderia ser realizado colocando 10 linhas iguais a essa no código fonte:

```
1 Console.WriteLine("Bom dia!");
```

Tabela 17: “Bom dia”

Se ao invés de 10 vezes fosse necessário imprimir 100 vezes, já seriam 100 linhas iguais no código fonte. É muito trabalhoso utilizar essa abordagem para esse problema. Através do comando **while**, é possível definir quantas vezes um determinado trecho de código deve ser executado pelo computador.

```
1 int contador = 0;
2
3 while (contador < 100)
4 {
5     Console.WriteLine("Bom dia!");
6     contador ++;
7 }
```

Tabela 18: Comando com while

A variável **contador** indica o número de vezes que a mensagem “Bom Dia” foi impressa na tela. O operador **++** incrementa a variável **contador** a cada rodada. O parâmetro do comando **while** tem que ser um valor booleano. Caso contrário, ocorrerá um erro de compilação.

## 1.12 FOR

O comando **for** é análogo ao **while**. A diferença entre esses dois comandos é que o **for** recebe três argumentos.

```
1 for (int contador = 0; contador < 100; contador++)
2 {
3     Console.WriteLine("Bom dia! :)");
4 }
```

Tabela 19: Comando for



## 1.13 EXERCÍCIOS DE FIXAÇÃO

Crie uma pasta com o nome Academia do Programador (Escolha um caminho fácil), a partir de agora utilizaremos o Visual Studio 2013, então siga os seguintes passos:

1º Passo: Vá no menu, FILE/New/ Project:

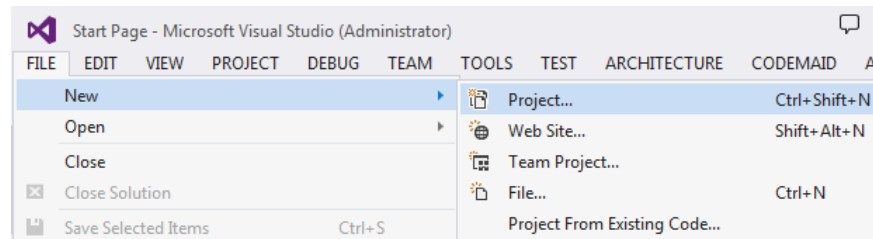


Figura 11: Tutorial Visual Studio 2013

2º Passo: Crie uma solução com o nome “AcademiaProgramador”, não esqueça de escolher o caminho de onde a solução irá ficar, no nosso caso dentro da pasta que você criou anteriormente.

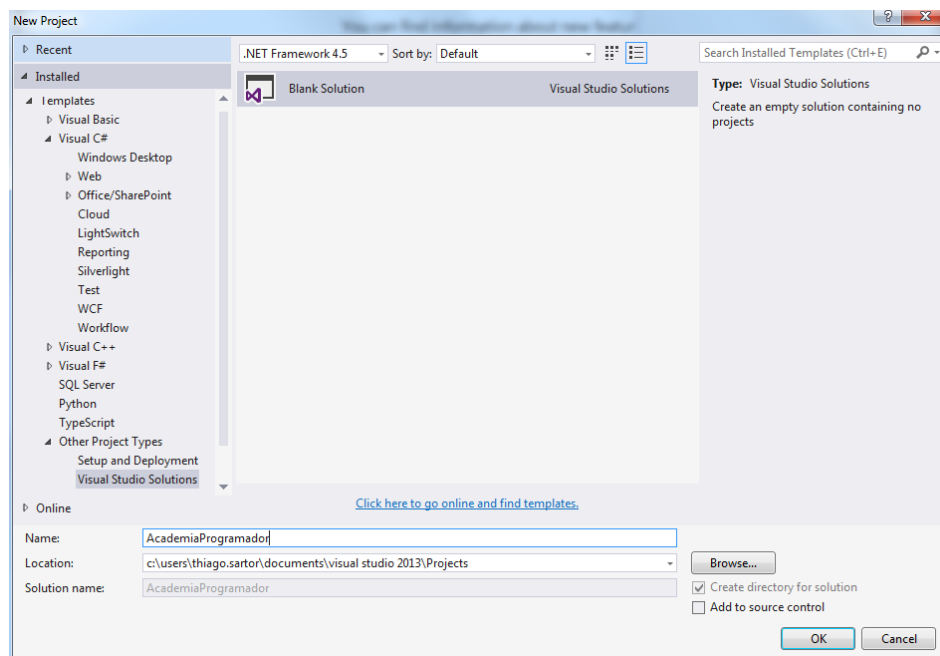


Figura 12: Criando uma solução em branco.

3º Passo: No menu superior em **VIEW/Solution Explorer**, você poderá ver a sua solução em branco.

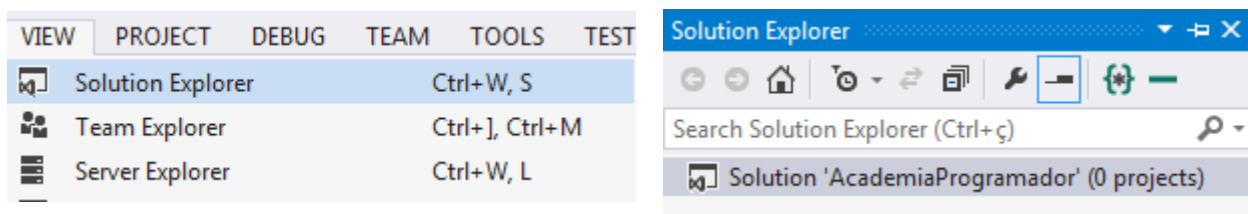


Figura 12: Abrindo o Solution Explorer



4º Passo: Clique com o botão direito do mouse na sua solução, vá em **Add/New Project**.

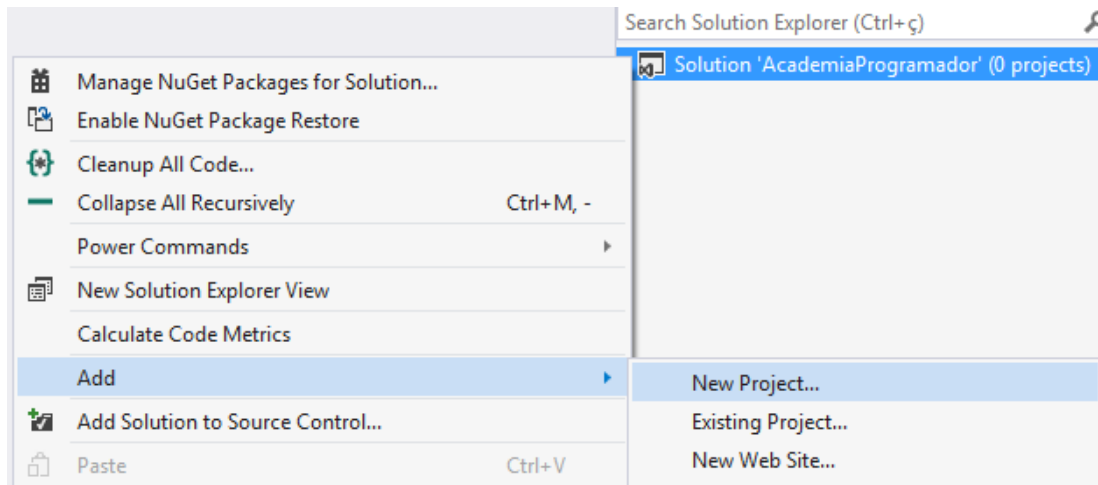


Figura 13: Adicionando um novo projeto

Abrirá uma janela, no canto esquerdo selecione **Visual C#**. Na parte central da tela selecione um projeto do tipo **ConsoleApplication**, coloque o nome do projeto como **Unidade1** e clique em **Ok**.

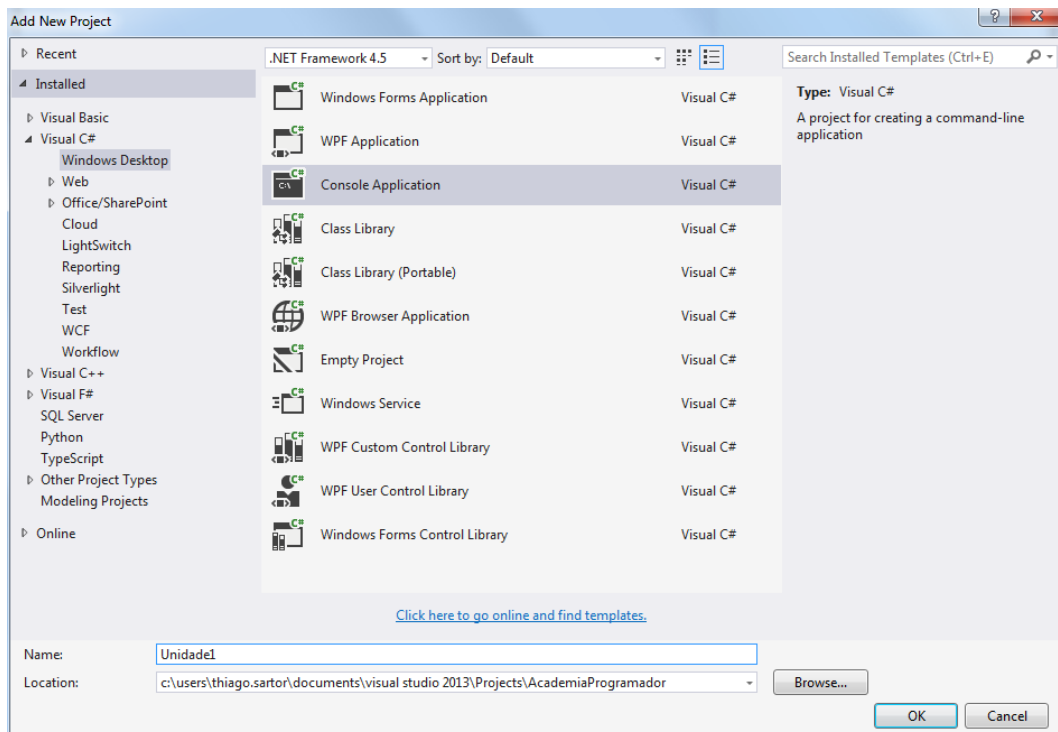


Figura 14: Criando um ConsoleApplication



O seu projeto **Unidade1** tem a classe **Program.cs** com o método principal **Main**. Está pronto para ser usada.

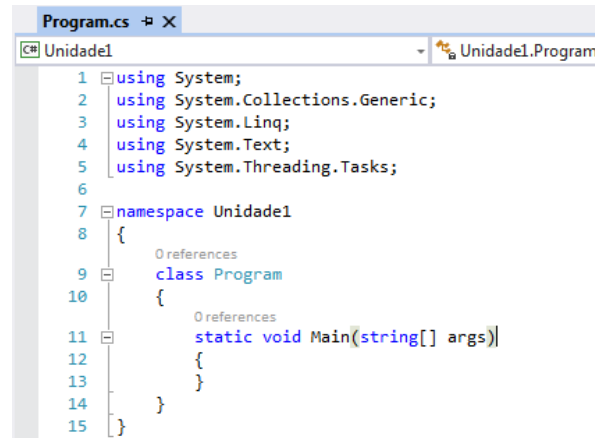


Figura 15: Classe Program.cs

5º Passo vá até sua **Solution Explorer**, clique com o botão direito no projeto **Unidade1**, vá até **Add/New Folder**, e arraste a classe Program.cs para dentro dessa pasta, é aí onde iremos colocar nossos exercícios.

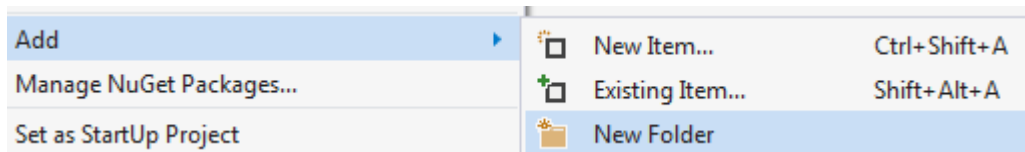


Figura 16: Criando uma pasta de Exercícios

Ela ficará dessa forma:

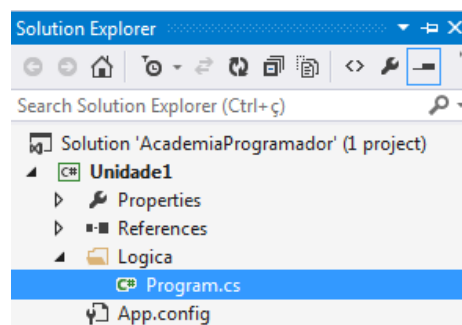


Figura 17: Adicionando a classe na pasta.



- 4) Imprima na tela o seu nome 100 vezes. Adicione na pasta **logica** a seguinte classe:

```
1 public class ImprimeNome
2 {
3     O references
4     private static void Main(string[] args)
5     {
6         for (int contador = 0; contador < 100; contador++)
7         {
8             Console.WriteLine("Thiago Sartor");
9         }
10    Console.ReadKey(); // ou Console.ReadLine()
11 }
```

Tabela 20: Criando a classe do primeiro exercício

Compile e execute a classe **ImprimeNome**:

```
Thiago Sartor
Thiago Sartor
Thiago Sartor
Thiago Sartor
```

Figura 18: Terminal com a saída

- 5) Imprima na tela os números de 0 até 99. Adicione na pasta **logica** a seguinte classe:

```
1 public class ImprimeNumeros
2 {
3     O references
4     private static void Main(string[] args)
5     {
6         for (int contador = 0; contador < 100; contador++)
7         {
8             Console.WriteLine(contador);
9         }
10    Console.ReadKey(); // ou Console.ReadLine()
11 }
```

Tabela 21: Código C# contador até 0 a 99.

Compile e execute a classe **ImprimeNumeros**:

```
96
97
98
99
```

Figura 19: Terminal com a saída



- 6) Faça um programa que percorra todos os números de 1 até 100. Para os números ímpares, deve ser impresso um "\*", e para os números pares, deve ser impresso dois "\*\*". Veja o exemplo abaixo:

```
*  
**  
*  
**  
*  
**
```

```
1 public class ImprimePadrao1  
2 {  
3     References  
4     private static void Main(string[] args)  
5     {  
6         for (int contador = 1; contador <= 100; contador++)  
7         {  
8             int resto = contador % 2;  
9             if (resto == 1)  
10            {  
11                Console.WriteLine("*");  
12            }  
13            else  
14            {  
15                Console.WriteLine("**");  
16            }  
17        }  
18    }  
19 }
```

Tabela 22: Código C# imprime padrão 1

Compile e execute a classe **ImprimePadrao1**:

```
*  
**  
*  
**  
*  
**  
*  
**
```

Figura 20: Terminal Padrão 1

- 7) Faça um programa que percorra todos os números de 1 até 100. Para os números múltiplos de 4, imprima a palavra "PI", e para os outros, imprima o próprio número. Veja o exemplo abaixo:

```
1  
2  
3  
PI  
5
```



6  
7  
PI

```
1 public class ImprimePadrao2
2 {
3     References
4     private static void Main(string[] args)
5     {
6         for (int contador = 1; contador <= 100; contador++)
7         {
8             int resto = contador % 4;
9             if (resto == 0)
10            {
11                Console.WriteLine("PI");
12            }
13            else
14            {
15                Console.WriteLine(contador);
16            }
17        }
18    }
19 }
```

Tabela 23: Código C# imprime padrão 2

Compile e execute a classe **ImprimePadrao2**.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
PI
5
6
7
PI
```

Figura 21: Terminal padrão 2

## 1.14 EXERCÍCIOS COMPLEMENTARES

1) Crie um programa que imprima na tela um triângulo de “\*”. Veja o exemplo abaixo:

```
*
**
***
****
```





\*\*\*\*\*

2) Crie um programa que imprima na tela vários triângulos de “\*”. Observe o padrão abaixo.

\*

\*\*

\*\*\*

\*\*\*\*

\*

\*\*

\*\*\*

\*\*\*\*

3) Os números de Fibonacci são uma sequência de números definida recursivamente. O primeiro elemento da sequência é 0 e o segundo é 1. Os outros elementos são calculados somando os dois antecessores.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233...

Crie um programa para imprimir os 30 primeiros números da sequência de Fibonacci.

4) Use seus conhecimentos para criar um programa que mostre um menu de atalho para os 5 padrões que acabamos de fazer. Exemplo:

```
      - MENU -
1 - Padrão 1
2 - Padrão 2
3 - Padrão 3
4 - Padrão 4
5 - Padrão 5
_
```

Figura 22: Terminal menu gerador



Se digitar o número 1, ele automaticamente tem de executar o código para o padrão 1, e assim sucessivamente.

Abaixo está o “esqueleto” da sua classe:

```
1 public class GeradordePadroes
2 {
3     Oreferences
4     private static void Main(string[] args)
5     {
6         int opc = 1; // Temos que atribuir o valor 1 na variável , para poder entrar no laço de repetição
7
8         while (opc != 0)
9         {
10             // Coloque o código do menu aqui .
11
12             string valorTela = Console.ReadLine();
13             opc = Convert.ToInt32(valorTela);
14
15             if (opc == 1)
16             {
17                 // Código para o Padrão 1
18             }
19             else if (opc == 2)
20             {
21                 // Código para o Padrão 2
22             }
23             else if (opc == 3)
24             {
25                 // Código para o Padrão 3
26             }
27             else if (opc == 4)
28             {
29                 // Código para o Padrão 4
30             }
31             else if (opc == 5)
32             {
33                 // Código para o Padrão 5
34             }
35         }
36     }
37 }
```

Tabela 24: GeradorDePadroes.cs

## 1.15 A “ARTE” DE DEBUGAR

Como assunto complementar dessa unidade, veremos o que é o Debug, uma funcionalidade essencial do Visual Studio e “amigo” do programador.

A capacidade de depurar (ou "debugar") programas é fundamental para o bom desenvolvimento de software. Existem excelentes recursos de depuração, permitindo ao desenvolvedor saber muito bem o que está acontecendo com o seu aplicativo durante a execução, de forma a eliminar erros.



### 1.15.1 Depuração

A qualquer momento que você está trabalhando no editor de código, você pode conjuntar um ponto de interrupção em uma linha de código, pressionando **F9**. Quando você pressiona **F5** para executar seu aplicativo no depurador do Visual Studio, o aplicativo irá parar nessa linha e você pode examinar como o valor de qualquer variável ou modo de exibição ou quando a execução se liberta de um loop, depurar a código uma linha em um time pressionando **F10** ou conjunto de pontos de parada adicionais.

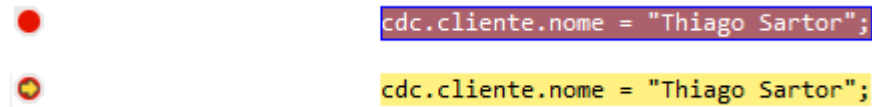


Figura 23: Break points

### 1.15.2 Immediate Window

É utilizada para depurar e avaliar expressões, executar instruções, imprimir valores de variáveis, e assim por diante. Ele permite que você insira expressões a serem avaliados ou executados pela linguagem de desenvolvimento durante a depuração. Para exibir a janela imediata, abra um projeto para edição, em seguida, escolher o Windows a partir do Debug menu e selecione Immediate, ou pressione CTRL + ALT + I.

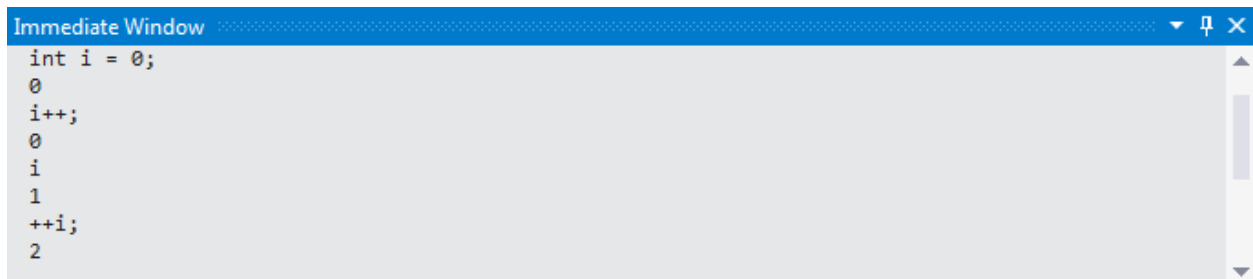


Figura 24: Immediate Window

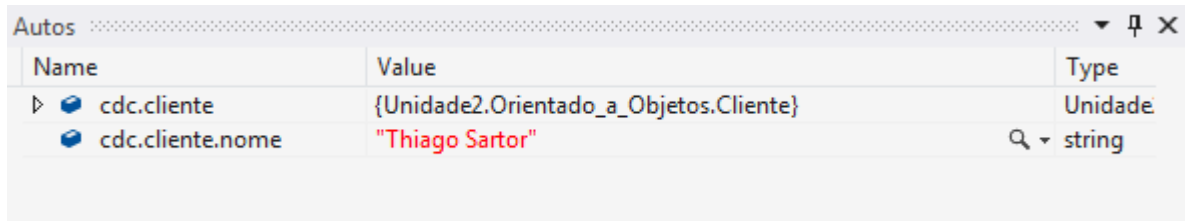
**Lembrando** que para você ter acesso as essas funcionalidades, o código deve estar em execução.

Outra janela importante é a Autos, que é semelhante à o teste de mesa tão tradicional na programação. Ela exibe variáveis utilizadas na instrução atual e da declaração anterior.

A declaração atual é a declaração no local de execução atual (a afirmação de que será executado ao lado, se a execução continua). O depurador identifica essas variáveis para você automaticamente, daí o nome da janela.



Variáveis de estrutura e de matriz tem um controle de árvore que você pode usar para exibir ou ocultar os elementos. Veja a imagem abaixo:



Name	Value	Type
cdc.cliente	{Unidade2.Orientado_a_Objeto.Cliente}	Unidade
cdc.cliente.nome	"Thiago Sartor"	string

Figura 25: Autos

Existem várias funcionalidades no menu Debug, você podem explorar e ver a qual delas é melhor para solucionar o seu problema.



Bons Estudos!