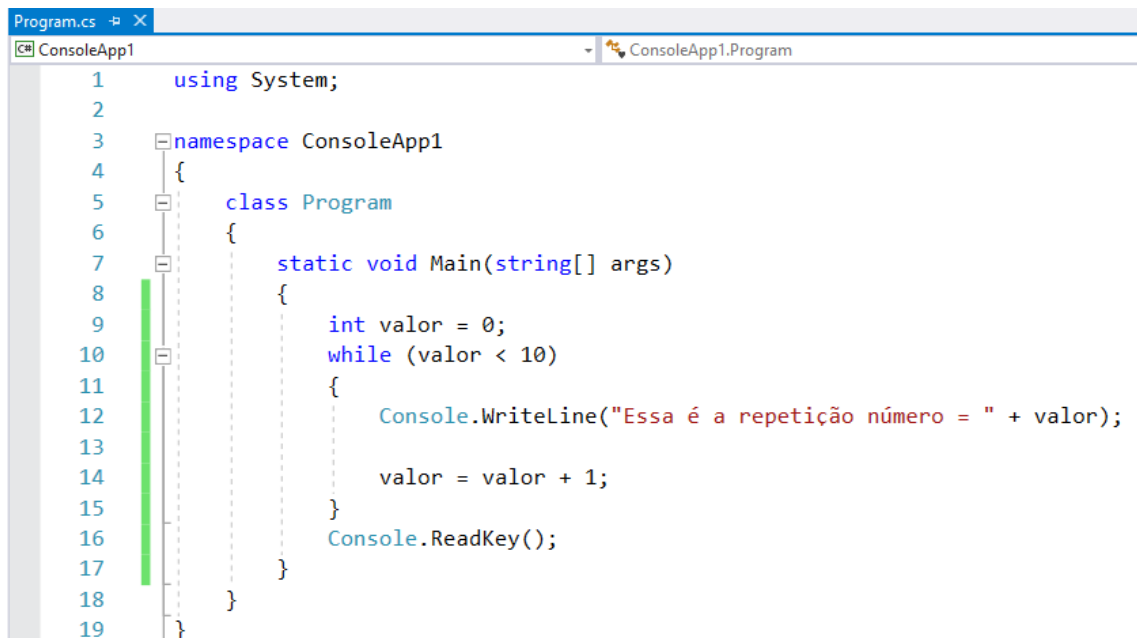


Abordaremos os seguintes tópicos na aula de hoje:

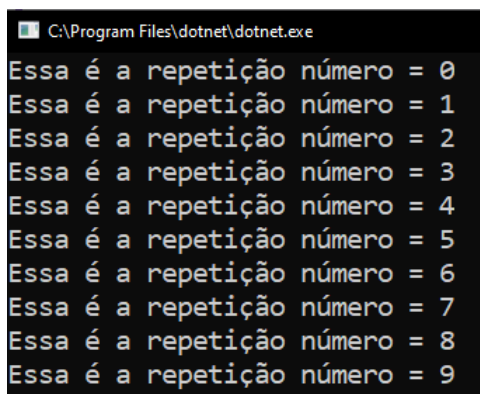
- 1 - Estruturas de repetição
 - 1.1 - WHILE
 - 1.2 - DO WHILE
 - 1.3 - FOR
 - 1.4 - ARRAY
 - 1.5 - FOREACH
- 2 - ADO .Net
- 3 - CRUD com SQL Server usando console

1.1 - WHILE

A estrutura de repetição while serve para executarmos o código enquanto ele for verdadeiro.



```
1  using System;
2
3  namespace ConsoleApp1
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              int valor = 0;
10             while (valor < 10)
11             {
12                 Console.WriteLine("Essa é a repetição número = " + valor);
13
14                 valor = valor + 1;
15             }
16             Console.ReadKey();
17         }
18     }
19 }
```



```
C:\Program Files\dotnet\dotnet.exe
Essa é a repetição número = 0
Essa é a repetição número = 1
Essa é a repetição número = 2
Essa é a repetição número = 3
Essa é a repetição número = 4
Essa é a repetição número = 5
Essa é a repetição número = 6
Essa é a repetição número = 7
Essa é a repetição número = 8
Essa é a repetição número = 9
```

Na maioria das vezes o código é escrito dessa forma:
Note que ao invés de utilizarmos a expressão $i = i + 1$, usaremos `i++`.
Esse operador utilizado é chamado de operador de incremento.

```
Program.cs*  X
ConsoleApp1  ConsoleApp1.Program

1  using System;
2
3  namespace ConsoleApp1
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              int i = 0;
10             while (i < 10)
11             {
12                 // Console.WriteLine("i = " + i);
13                 Console.WriteLine("i = {0}", i);
14
15                 // Interpolação de valores com { }
16
17                 // i = i + 1;
18                 i++;
19             }
20             Console.ReadKey();
21         }
22     }
23 }
24 }
```

Aqui foi utilizado a interpolação de valores com os símbolos `{ }`.

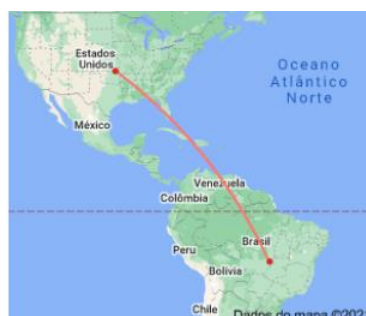
```
C:\Program Files\dotnet\dotnet.exe
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
```

Podemos fazer agora o exercício da tabuada.
E depois das milhas acumuladas.

A cada viagem aos EUA, ganho 400 milhas para serem utilizadas em qualquer outra viagem.

4.544 mi

Distância de Brasil até Estados Unidos



Exercício da tabuada.

```
Program.cs  X
ConsoleApp1 ConsoleApp1.Program
1 using System;
2
3 namespace ConsoleApp1
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             int valor = 1;
10
11             Console.Write("Entre com o valor da tabuada: ");
12             int valTab = int.Parse(Console.ReadLine());
13
14             while (valor < 10)
15             {
16                 // Console.WriteLine(valTab + " X " + valor + " = " + valTab*valor);
17                 Console.WriteLine("{0} X {1} = {2}", valTab, valor, valTab*valor);
18
19                 valor = valor + 1;
20             }
21             Console.ReadKey();
22         }
23     }
24 }
```

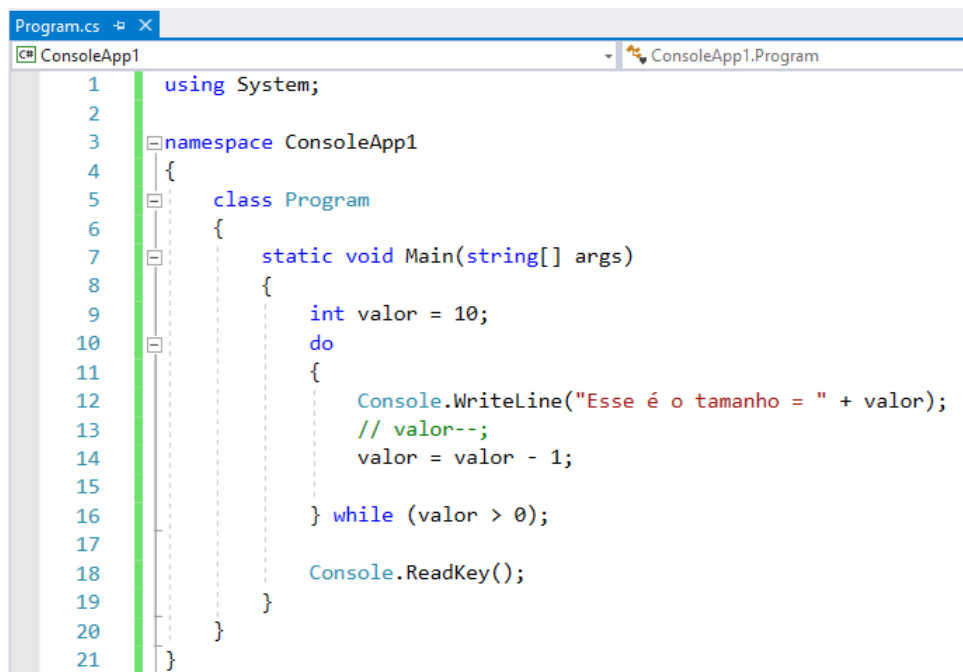
Exercício das milhas ganhas.

```
Program.cs  X
ConsoleApp1 ConsoleApp1.Program Main(string[] args)
1 using System;
2
3 namespace ConsoleApp1
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             int milhasGanhas = 400;
10             int contador = 1;
11
12             Console.Write("Entre com a qtde de viagens: ");
13             int qteViagem = int.Parse(Console.ReadLine());
14
15             while (qteViagem > contador)
16             {
17                 Console.WriteLine("Milhas da viagem " + contador + " = " + contador*milhasGanhas);
18
19                 // contador++;
20                 contador = contador + 1;
21             }
22             Console.ReadKey();
23         }
24     }
25 }
```

1.2 – DO WHILE

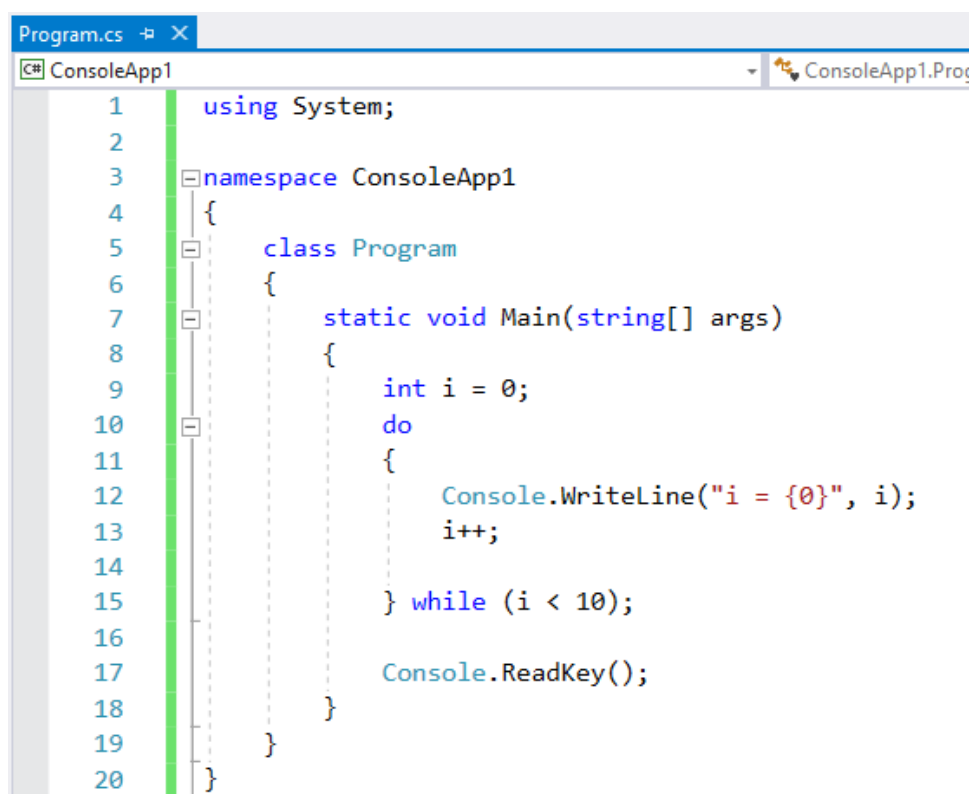
Também temos a estrutura de repetição do while. Essa repetição é utilizada quando precisamos que seja executada a condição ao menos **uma** única vez.

Ou podemos dizer que sempre a condição será executada, e somente então a condição será verificada.



```
Program.cs  X
[ConsoleApp1] ConsoleApp1.Program
1  using System;
2
3  namespace ConsoleApp1
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              int valor = 10;
10             do
11             {
12                 Console.WriteLine("Esse é o tamanho = " + valor);
13                 // valor--;
14                 valor = valor - 1;
15             } while (valor > 0);
16
17             Console.ReadKey();
18         }
19     }
20 }
21 }
```

Também na maioria das vezes o código é escrito dessa forma:
Note que ao invés de utilizarmos a expressão $i = i - 1$, usaremos $i--$.
Esse operador utilizado é chamado de operador de decremento.



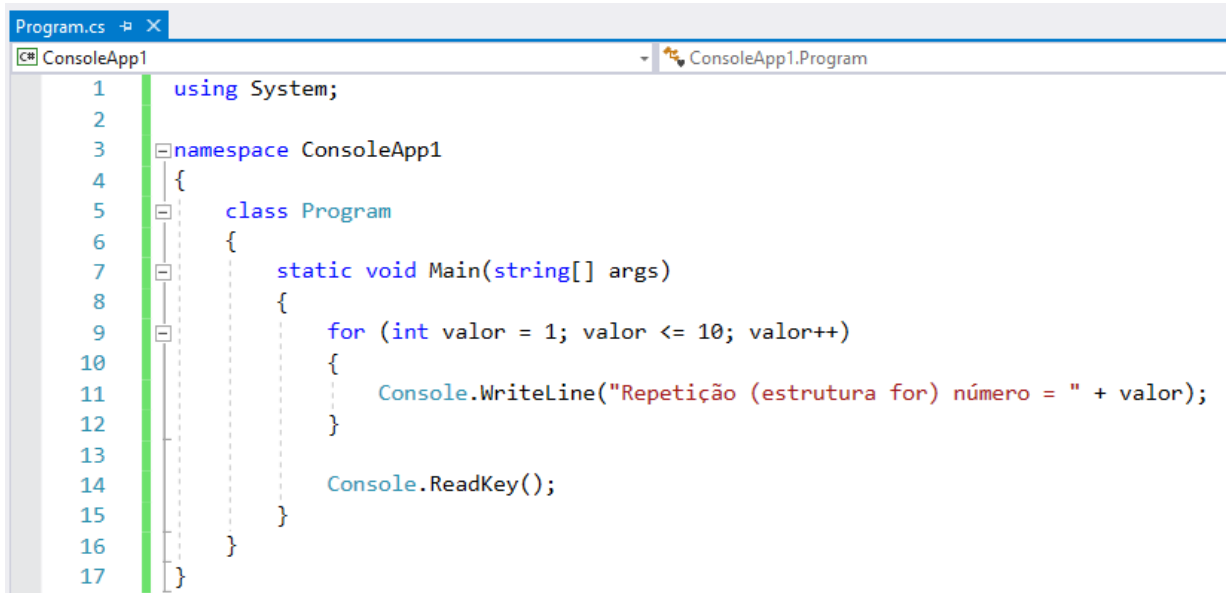
```
Program.cs  X
[ConsoleApp1] ConsoleApp1.Prog
1  using System;
2
3  namespace ConsoleApp1
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              int i = 0;
10             do
11             {
12                 Console.WriteLine("i = {0}", i);
13                 i++;
14             } while (i < 10);
15
16             Console.ReadKey();
17         }
18     }
19 }
20 }
```

Exemplo do paciente na emergência, que precisa ser atendido, e depois tem um tratamento que pode ser feito ou não pelo plano de saúde. (Fisioterapia ou cirurgia estética, ou outro cuidado).

```
Program.cs  Program.cs
ConsoleApp1 ConsoleApp1
3 namespace ConsoleApp1
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             int diasDeInter = 1;
10
11
12             Console.WriteLine("Plano de Saude - Dias de Internação");
13             Console.Write("Entre com a qtde. autorizada: ");
14             int qtdeAutoriz = int.Parse(Console.ReadLine());
15             do
16             {
17                 Console.WriteLine("Atender o paciente - dia " + diasDeInter );
18                 diasDeInter = diasDeInter + 1;
19
20             } while (qtdeAutoriz > diasDeInter );
21
22             Console.ReadKey();
23         }
24     }
25 }
```

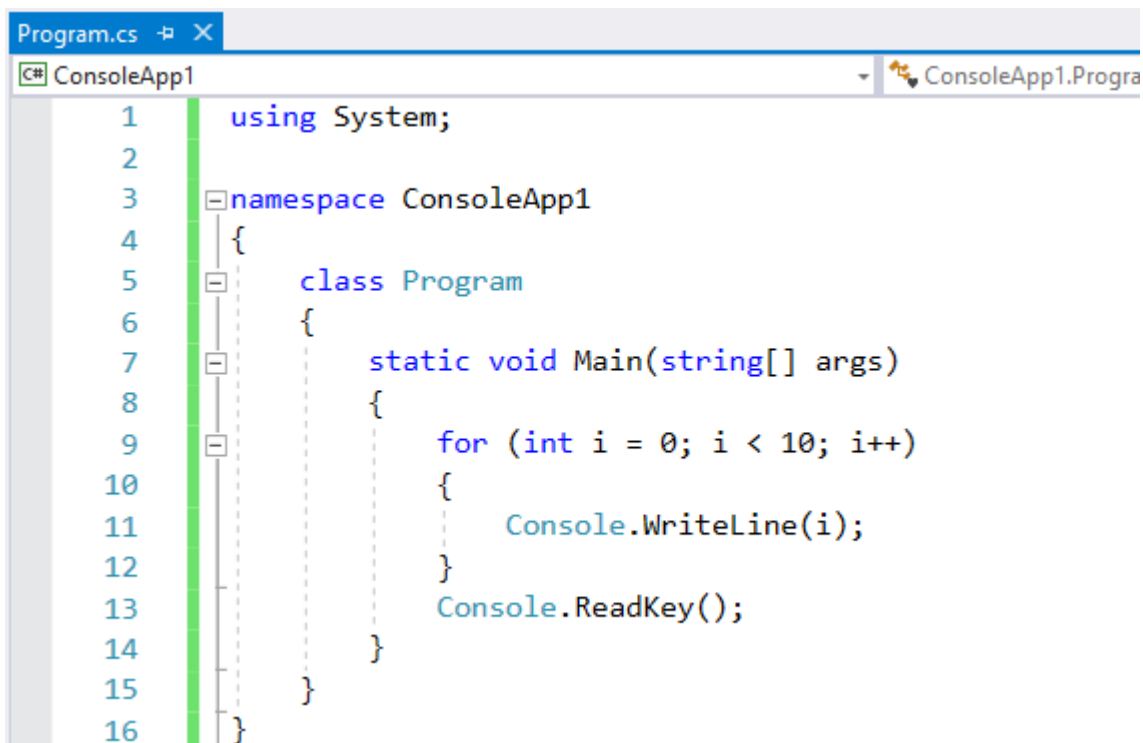
1.3 – FOR

A estrutura de repetição for tem o seguinte aspecto:



```
Program.cs [X]
ConsoleApp1 ConsoleApp1.Program
1 using System;
2
3 namespace ConsoleApp1
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             for (int valor = 1; valor <= 10; valor++)
10             {
11                 Console.WriteLine("Repetição (estrutura for) número = " + valor);
12             }
13
14             Console.ReadKey();
15         }
16     }
17 }
```

A estrutura for é mais usualmente vista dessa forma.



```
Program.cs [X]
ConsoleApp1 ConsoleApp1.Progra
1 using System;
2
3 namespace ConsoleApp1
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             for (int i = 0; i < 10; i++)
10             {
11                 Console.WriteLine(i);
12             }
13             Console.ReadKey();
14         }
15     }
16 }
```

Faremos a tabuada para ver como se comporta a estrutura for.

```
Program.cs  X
ConsoleApp1  ConsoleApp1.Program

1  using System;
2
3  namespace ConsoleApp1
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.Write("Digite a tabuada: ");
10             int tab = int.Parse(Console.ReadLine());
11
12             for (int contador = 0; contador < 10; contador++)
13             {
14                 Console.WriteLine("{0} X {1} = {2}", tab, contador, tab*contador);
15             }
16             Console.ReadKey();
17         }
18     }
19 }
20
```

Faremos uma estrutura de repetição for com uma estrutura de decisão if.

```
Program.cs  X
ConsoleApp1  ConsoleApp1.Program

1  using System;
2
3  namespace ConsoleApp1
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Veja seus descontos");
10             Console.Write("Até quantas peças pretende levar hoje? ");
11             int qtdeRoupa = int.Parse(Console.ReadLine());
12             double vlrRoupa = 100;
13             double desc = 5.0 / 100.00;
14             double descMax = 0.25;
15
16             for (int cont = 1; cont <= qtdeRoupa; cont++)
17             {
18                 if (cont <= 5)
19                 {
20                     Console.WriteLine("Valor R${0} desconto = R${1}", vlrRoupa * cont, vlrRoupa * cont * (desc * cont));
21                 }
22                 else
23                 {
24                     Console.WriteLine("Valor R${0} desconto = R${1}", vlrRoupa * cont, vlrRoupa * cont * descMax);
25                 }
26             }
27             Console.ReadKey();
28         }
29     }
30 }
31
```

1.4 - ARRAY

Arrays são estruturas de dados que agrupam dados do mesmo tipo

```
int[] distancias;  
distancias = new int[8];
```

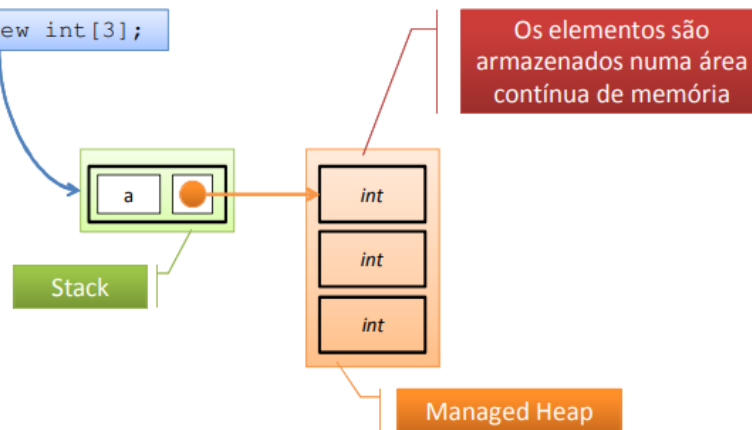
Array de **int** com 8 posições

```
double[] notas = new double[5];
```

Array de **double** com 5 posições

Arrays são reference types alocados no managed heap

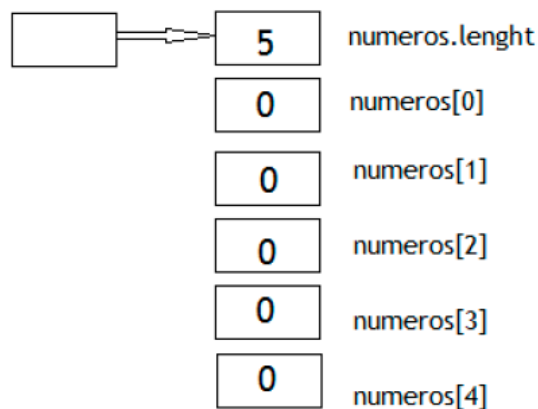
```
int[] a = new int[3];
```



Declaração: **int[] variavel = new int [n];**

Representação na memória: array de uma dimensão

int[] números = new int [5]



A inicialização de arrays pode ser feita de várias formas

```
int[] array = new int[5];
```

```
int[] array = new int[2]{ 1, 2 };
```

```
int[] array = new int[] { 1, 2 };
```

```
int[] array = { 1, 2 };
```

Arrays podem ter uma ou mais dimensões

```
int[,] array = new int[3, 4];
```

2 dimensões, 3x4

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 2 | 5 | 3 | 6 |
| 1 | 4 | 9 | 0 | 2 |
| 2 | 1 | 7 | 8 | 9 |

```
array[1, 3] = 9;
```

```
int[,] array = {  
    { 2, 5, 3, 6 },  
    { 4, 9, 0, 2 },  
    { 1, 7, 8, 9 }  
};
```

Temos então:

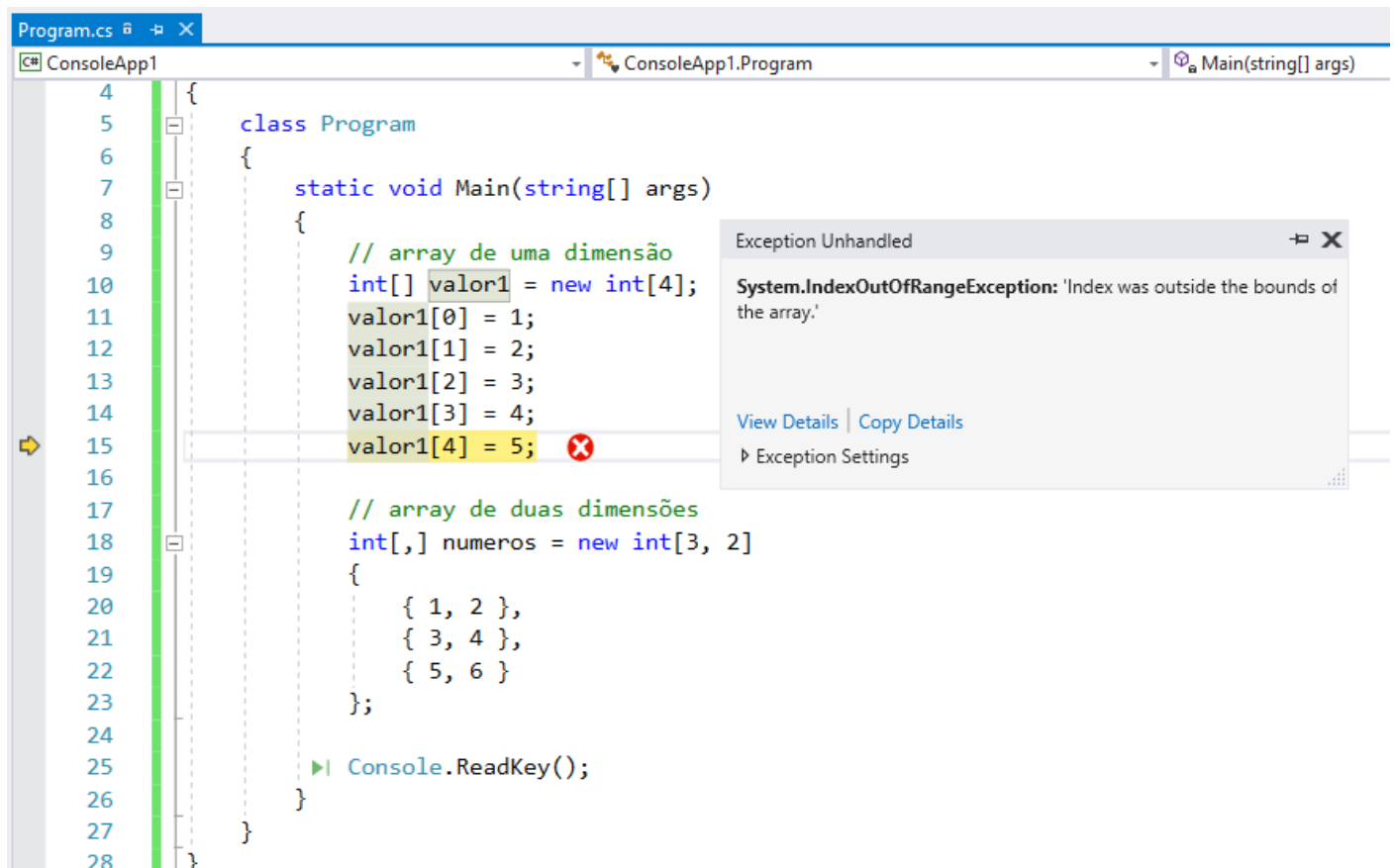
1ª linha ou linha índice 0

2ª linha ou linha índice 1

1ª coluna ou coluna índice 0

2ª coluna ou coluna índice 1

Posso até tentar colocar um elemento a mais no meu array porém ele não irá compilar.



```
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             // array de uma dimensão
10            int[] valor1 = new int[4];
11            valor1[0] = 1;
12            valor1[1] = 2;
13            valor1[2] = 3;
14            valor1[3] = 4;
15            valor1[4] = 5;
16
17            // array de duas dimensões
18            int[,] numeros = new int[3, 2]
19            {
20                { 1, 2 },
21                { 3, 4 },
22                { 5, 6 }
23            };
24
25            Console.ReadKey();
26        }
27    }
28 }
```

Exception Unhandled

System.IndexOutOfRangeException: 'Index was outside the bounds of the array.'

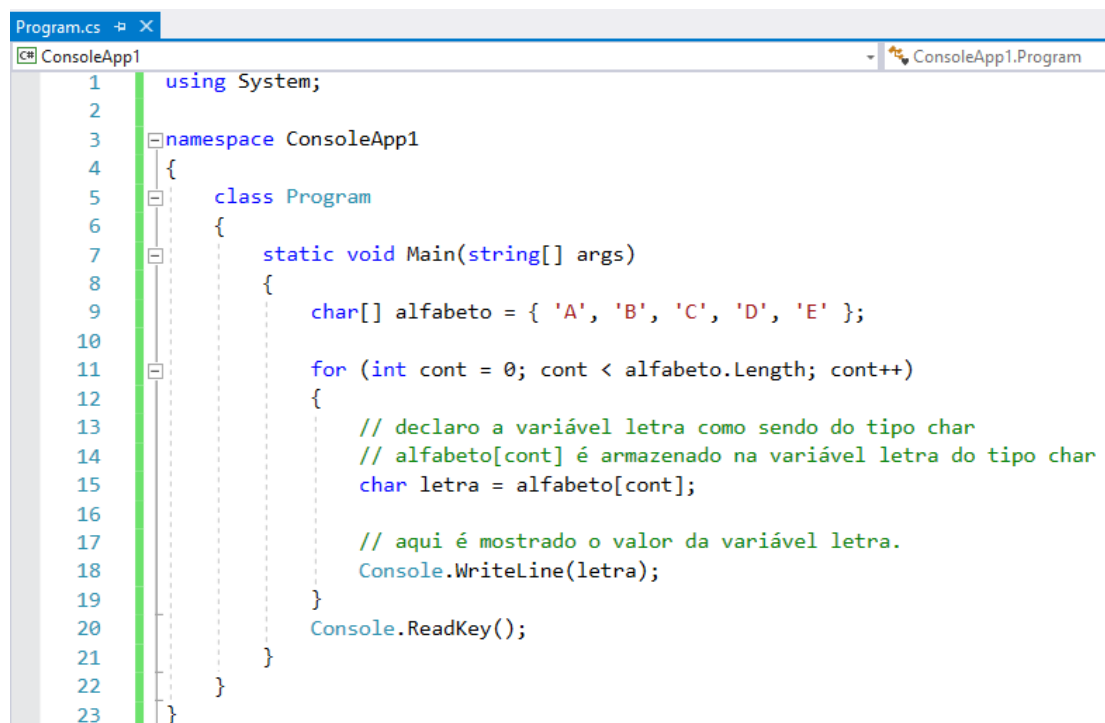
[View Details](#) | [Copy Details](#)

[Exception Settings](#)

1.5 - FOREACH

É usado para percorrer um array.

Antes, precisamos entender como percorrer um array utilizando a estrutura de repetição **for**.



```
1 using System;
2
3 namespace ConsoleApp1
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             char[] alfabeto = { 'A', 'B', 'C', 'D', 'E' };
10
11            for (int cont = 0; cont < alfabeto.Length; cont++)
12            {
13                // declaro a variável letra como sendo do tipo char
14                // alfabeto[cont] é armazenado na variável letra do tipo char
15                char letra = alfabeto[cont];
16
17                // aqui é mostrado o valor da variável letra.
18                Console.WriteLine(letra);
19            }
20            Console.ReadKey();
21        }
22    }
23 }
```

Note que utilizamos o atributo Length (comprimento) do array alfabeto, para saber esse valor e percorrer o array.

Colocaremos o foreach logo abaixo para compararmos

```
Program.cs  [X]
ConsoleApp1 ConsoleApp1.Program
1  using System;
2
3  namespace ConsoleApp1
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              char[] alfabeto = { 'A', 'B', 'C', 'D', 'E' };
10
11              for (int cont = 0; cont < alfabeto.Length; cont++)
12              {
13                  // declaro a variável letra como sendo do tipo char
14                  // alfabeto[cont] é armazenado na variável letra do tipo char
15                  char letra = alfabeto[cont];
16
17                  // aqui é mostrado o valor da variável letra.
18                  Console.WriteLine(letra);
19              }
20
21              // foreach = para cada
22              foreach (char letra in alfabeto)
23              {
24                  Console.WriteLine(letra);
25              }
26
27              Console.ReadKey();
28          }
29      }
30  }
```

O foreach tem algumas limitações:

Ele somente itera do início até o final, não é possível iterar em apenas uma parte do array. Apenas é possível iterar na ordem crescente. Não funciona na ordem decrescente.

Percorrendo matrizes.

Utilizando o foreach é simples. No caso do for usamos o GetLength.

Para saber qual o tamanho da linha (quantas linhas) utilizarei o atributo matriz.GetLength(0), pois necessito saber o tamanho da primeira linha que é a primeira dimensão.

O mesmo farei para saber o tamanho (quantas colunas) da coluna. Utilizarei o atributo matriz.GetLength(0), pois necessito saber o tamanho da segunda linha que é a segunda dimensão.

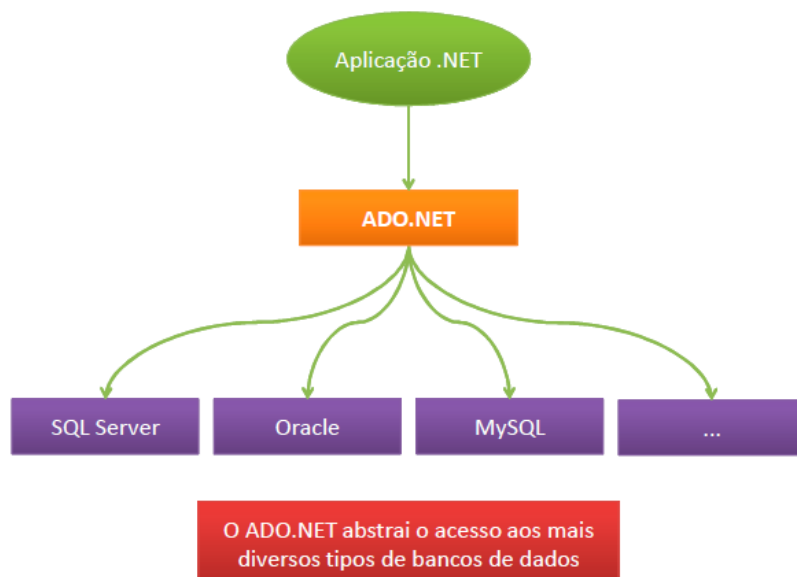
Vamos utilizar a estrutura de repetição foreach para percorrer e mostrar os valores que estão entre 100 e 400.

```
Program.cs  X
C# ConsoleApp1 ConsoleApp1.Program
1 using System;
2
3 namespace ConsoleApp1
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             int[] preco = { 300, 700, 800, 200, 500, 600, 100, 400 };
10
11             foreach (int orcamento in preco)
12             {
13                 if (orcamento <= 400 && orcamento >= 100 )
14                 {
15                     Console.WriteLine(orcamento);
16                 }
17             }
18             Console.ReadKey();
19         }
20     }
21 }
```

2 - ADO .Net

A comunicação da aplicação com o baco de dados nem sempre é simples.

Na plataforma .Net existe um componente que faz essa ponte entre a aplicação e o banco de dados que é o ADO.NET



Todo o trabalho de conexão, select, insert e outras operações ficam sob a responsabilidade do ADO.NET, deixando o trabalho do programador muito mais simplificado.

A grande vantagem é que se você fez sua aplicação utilizando o SQL Server e depois precisa mudar para Oracle, fica muito mais fácil. Apenas algumas configurações e o ADO.NET continuará fazendo todo esse trabalho de conexão

Um data provider é responsável por permitir a comunicação com determinado SGBD

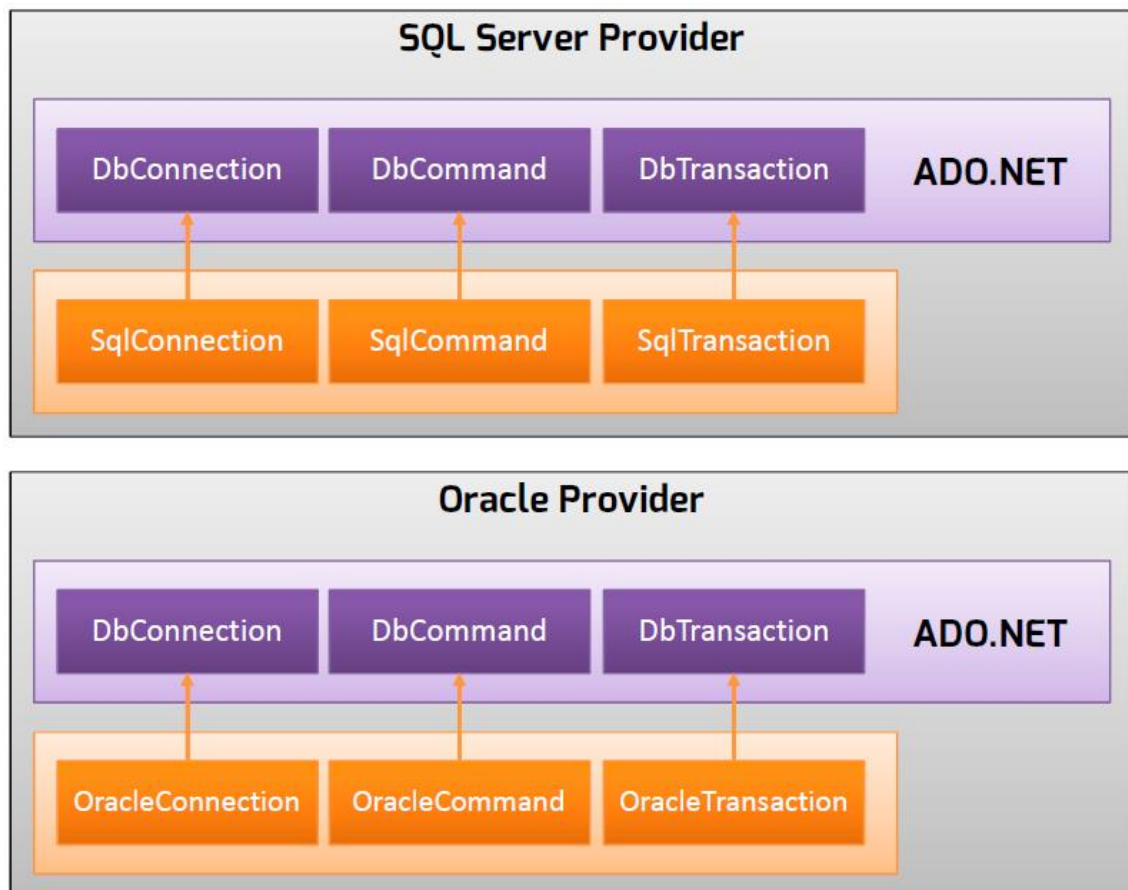
- Cada SGBD tem um provider específico.

A plataforma .NET já traz alguns providers nativos

- Microsoft SQL Server
- ODBC
- OLE DB

Os providers estão disponíveis no assembly System.Data.dll

As classes que fornecem herança para os componentes ADO.NET é que possibilitam a conexão com o banco de dados. Estão dentro do Provider e podem ser fornecidos pelo ADO.NET ou pelo fabricante do SGBD (Banco de Dados).



Um provider factory permite criar objetos de um provider específico.

```
DbProviderFactory factory =  
    DbProviderFactories.GetFactory("System.Data.SqlClient");
```

Nome do provider

```
DbConnection conn = factory.CreateConnection();  
DbCommand cmd = factory.CreateCommand();
```

Código independente
de provider

```
SqlConnection conn = new SqlConnection();  
SqlCommand cmd = new SqlCommand();
```

Código específico do
SQL Server

Nesse último grupo de comandos, não utilizei a provider factory e deixei a conexão totalmente dependente do fabricante do banco de dados.

A recomendação é que utilize a factory, pois deixa as classes mais abstratas e não tão específicas.

Na ConnectionString é que tem informações do meu banco de dados, se está na rede, se está na internet, se é um servidor local.

Tenho que dizer o nome do banco de dados que vou me conectar. No nosso caso Initial Catalog=testedb.

A autenticação que é Integrated Security=True, para servidor local.

Quando defino a ConnectionString já estou apto a abrir a conexão com o banco de dados, para isso chamo o método Open(); A partir desse momento tenho uma conexão aberta com o banco de dados e já posso trocar informações, como: inserir dados, atualizar dados e demais operações.

```
using (DbConnection conn = factory.CreateConnection())  
{  
    conn.ConnectionString = @"Data Source=(local)\SQLEXPRESS;  
    Initial Catalog=testedb;Integrated Security=True";  
    conn.Open();  
}
```

Bloco using

Detalhe importante é: sempre que abrir uma conexão com o banco de dados temos que fechá-la. Essa conexão utiliza muitos recursos e quando existem muitas conexões abertas há uma queda de desempenho na aplicação.

Outra forma é utilizar o bloco try..finaly. Dessa forma você estará garantindo que a conexão pelo método Close(); com o banco de dados está sendo fechada.

```
DbConnection conn = factory.CreateConnection();
try
{
    conn.Open();
}
finally
{
    conn.Close();
}
```

Bloco try..finaly

Cada provider define um formato e informações para a connection string. Consulte a documentação

A classe SqlConnectionStringBuilder pode ser usada para facilitar a criação da connection string

```
SqlConnectionStringBuilder sb = new SqlConnectionStringBuilder();
sb.DataSource = @"(local)\SQLEXPRESS";
sb.InitialCatalog = "testedb";
sb.IntegratedSecurity = true;

conn.ConnectionString = sb.ConnectionString;
```

A classe SqlConnectionStringBuilder fornece as classe de DataSource que indica onde está o banco de dados, nesse caso um servidor local. define o banco de dados onde será feita a conexão que é a InitialCatalog. Estabelece a segurança através do IntegratedSecurity.

Externalizando Dados do Provider

Deixar informações sobre o provider diretamente no código não é uma boa prática.

O nome do provider e a connection string podem ir para um arquivo de configuração, por exemplo: App.config.

```
<configuration>
  <appSettings>
    <add key="provider" value="System.Data.SqlClient" />
  </appSettings>

  <connectionStrings>
    <add name="db" connectionString="Data Source=(local)\SQLEXPRESS;
      Initial Catalog=testedb;Integrated Security=True" />
  </connectionStrings>
</configuration>
```

A classe ConfigurationManager é utilizada para ler as informações

- Assembly: System.Configuration.dll
- Namespace: System.Configuration

```
String provider = ConfigurationManager.AppSettings["provider"];
```

```
String cn = ConfigurationManager.ConnectionStrings["db"].ConnectionString;
```

Criando Comandos

Depois de estabelecida a conexão, o próximo passo é a criação de comandos, que serão executados no banco de dados

- INSERT, DELETE, UPDATE, SELECT

Um objeto do tipo DbCommand é utilizado:

```
DbCommand cmd = factory.CreateCommand();  
cmd.Connection = conn;  
cmd.CommandText = "SELECT nome FROM contato";
```

A conexão e a query devem ser associadas ao comando

A execução de um comando é feita através dos seguintes métodos

| Método | Tipo de Comando | Retorno |
|--------------------------|------------------------|---------------------|
| <i>ExecuteNonQuery()</i> | INSERT, UPDATE, DELETE | <i>int</i> |
| <i>ExecuteReader()</i> | SELECT | <i>DbDataReader</i> |
| <i>ExecuteScalar()</i> | SELECT | <i>object</i> |

Para inserir dados no banco de dados, utilizo o `ExecuteNonQuery` e para consulta o `ExecuteReader`.

O laço de repetição `while` procura por registros e extrai os dados.

```
...  
cmd.CommandText = "INSERT INTO contato(nome) VALUES ('Maria')";  
int num = cmd.ExecuteNonQuery();
```

Número de
registros afetados

```
...  
cmd.CommandText = "SELECT nome, idade, FROM contato";  
using (DbDataReader result = cmd.ExecuteReader())  
{  
    while (result.Read())  
    {  
        string nome = (string)result["nome"];  
        int idade = (int)result["idade"];  
    }  
}
```

Loop enquanto
houver registros

Extrai os dados do
data reader

É possível também usar índices
com base na posição da coluna

Agora vejamos a consulta que resulta em apenas um único registro utilizando o `ExecuteScalar()`;

`DBNull` é uma classe que representa um valor nulo vindo do banco de dados.

```
...  
cmd.CommandText = "SELECT MAX(idade) FROM contato";  
object obj = cmd.ExecuteScalar();  
  
if (!Convert.IsDBNull(obj))  
{  
    int max = (int)obj;  
}
```

Retorna o valor da primeira
coluna do primeiro registro

Verifica se o valor é nulo
antes de fazer o casting

```
if (count == DBNull.Value)  
{  
    ...  
}
```

Outra forma de
checar a nulidade

Podemos tratar esse valor nulo como retornando a mensagem: não existem registros.

Transações

- Uma transação é uma operação atômica
 - Ou ela executa por completo, ou não executa
 - Não existe a possibilidade de ela executar apenas parcialmente

O exemplo mais clássico de transação é uma transferência bancária de um valor:

- Duas operações
 - Saque na conta de origem
 - Depósito na conta de destino
- Ambas precisam executar de forma atômica

No ADO.NET, uma transação é iniciada através do método `BeginTransaction()`, da classe `DbConnection`

```
DbTransaction transaction = conn.BeginTransaction();
```

Objeto que representa
a transação

A transação termina com um `commit` ou um `rollback`.

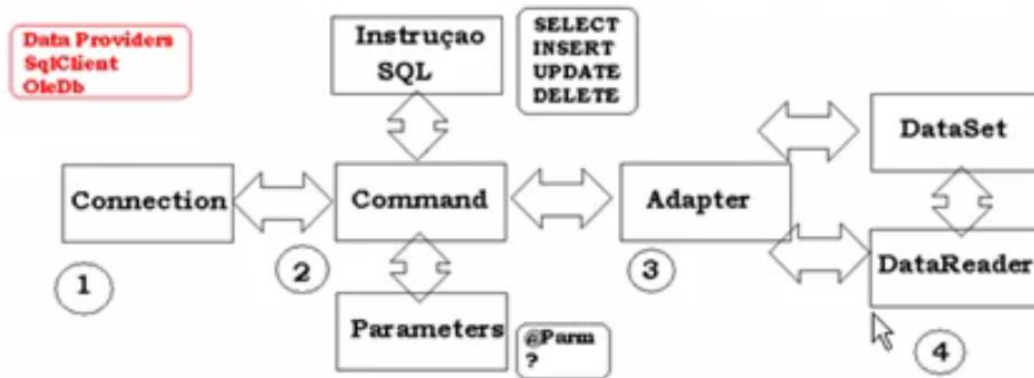
```
transaction.Commit();
```

Efetiva a transação

```
transaction.Rollback();
```

Desfaz a transação

ADO .NET - OBJETOS



Primeiro temos a Connection que será permitida a conexão, depois temos o Command que será em cima de uma instrução SQL ou de parâmetros. O adapter que fará a conexão entre os comandos e os DataSet e os DataReader

Roteiro básico

Definir uma **string** de conexão: (Representa informações sobre o caminho do banco de dados, usuário, senha, servidor, etc).

Criar um objeto **Connection** com a base de dados usando a string de conexão; esse objeto permite estabelecer a conexão com a base de dados.

Definir uma instrução SQL ou stored procedure (**Select, Insert, Delete, Update**)

Criar um objeto **Command** e vincula-lo ao objeto **Connection** e a uma instrução SQL (Stored Procedure).

Definir os parâmetros via propriedade **Parameters**;

Abrir a conexão com a base de dados.

Executar a instrução SQL e obter um **DataReader** ou **DataSet** ou o **número de linhas afetadas**.

O objeto Command pode executar instruções **SELECT, INSERT, UPDATE** ou **DELETE** e *Stored Procedures*

Para executar uma instrução **SELECT** pode-se invocar o método **ExecuteReader()** sobre o objeto **Command**, o qual retorna um **DataReader**.

Para executar um comando (*instrução SQL*) que não retorna resultados, tal como:

- **INSERT, UPDATE** ou **DELETE**, invoca-se o método **ExecuteNonQuery()**.

Estes comandos não retornam registros, simplesmente modificam dados e retornam o número de linhas (*registros*) afetadas.

Exemplo

SQL Server – System.Data.SqlClient

Com o namespace System.Data.SqlClient terei acesso às classes para fazer a conexão com o SQL Server, aos data providers que permitem a conexão com o SQL Server.

Faremos o seguinte:

Definimos a string de conexão com:

```
String strCon = "Data Souce=server;Initial Catalog=database;Persist  
Securit Info=True;User ID=as;Password=xxx"
```

Depois criamos a conexão:

```
SqlConnection con = new SqlConnection(strCon);
```

Em seguida definimos um comando SQL

```
String SQL = "delete from produtos where id=@id";
```

@id é o parâmetro da instrução.

Criaremos o comando SQL com:

```
SqlCommand cmd = new SqlCommand(SQL, con);
```

Passaremos os parâmetros de instrução que serão executados

```
Cmd.Parameters.AddWithValue("@id", txtID.Text);
```

O parâmetro txtID.Text está pegando o valor de uma caixa de texto do usuário e o ID seria o código de identificação do produto.

Abriremos a conexão com:

```
con.Open();
```

Executamos o método:

```
cmd.ExecuteNonQuery();
```

E fechamos a conexão:

```
con.Close();
```

3 - CRUD com SQL Server usando console

Vídeo serve passo a passo para criar uma conexão com o banco de dados via console.

C# Console ADO NET DATABASE CONNECTIVITY 2016 09 12 - 1910 03

<https://www.youtube.com/watch?v=xIKLfeS2kiE>

Código está no arquivo:

BD-02-conecta-completo.cs

Será comentado na aula