

JAN/2018

CURSO DE FÉRIAS

BACK END

COM C#



Sumário

1 - Lógica de Programação	4
1.1 - As variáveis.....	4
1.2 - Modificadores	4
1.3 - Tipos de dados	5
1.4 - Variáveis Reference Types	7
1.5 - String.IsNullOrEmpty.....	11
1.6 - Contains	11
1.7 - Equals.....	12
1.8 - Remove	12
1.9 - Replace	12
1.10 - Substring	13
1.11 - ToLower.....	13
1.12 - ToUpper	13
1.13 - Split.....	14
1.14 - Operadores Relacionais	14
1.15 Operadores Lógicos.....	15
1.16 - If/else	15
1.17 - Operador ternário	17
1.18 - For	17
1.19 - Break	18
1.20 - While	18
1.21 - Continue.....	19
1.22 - Switch/Case.....	20
1.23 - Default.....	21
1.24 - Foreach.....	21
1.25 - Declaração de arrays.....	22
1.26 - Acesso aos dados do array	23
1.27 - Usando Expressões Lambda com LINQ.....	24
2 - Conceitos de C#.....	27
2.1 Introdução	27
2.2 Variáveis	27
2.3 Métodos	30
2.3.1 Sobrecarga.....	30
3.1 - Abstração	31

3.2 - Encapsulamento.....	32
3.3 - Exercícios.....	32
3.4 - Modificadores de Acesso (Visibilidade)	32
3.5 - Herança	32
3.6 - Exercícios.....	33
3.7 - Polimorfismo	34
3.8 - Exercícios.....	34
4 - Modelagem de Dados	34
4.1 - Preparação do Ambiente	34
4.2 - Microsoft SQL Server.....	35
4.3 - BANCO DE DADOS RELACIONAL.....	35
4.4 - SQL - STRUCTURED QUERY LANGUAGE	36
4.5 - INTEGRIDADE DE DADOS	36
4.6 - TRANSAÇÕES	36
4.7 - Abordagem do Modelo ER (entidade e relacionamento)	36
4.8 - ENTIDADE	37
4.9 –	37
RELACIONAMENTO	37
4.10 - CARDINALIDADE.....	37
4.11 - ATRIBUTO.....	38
4.12 - MODELO CONCEITUAL ER	38
4.13 - Composição de um Banco de Dados Relacional	39
4.14 - TABELA	39
4.15 - CHAVES.....	40
4.16 - DOMÍNÍOS E VALORES VAZIOS.....	40
4.17 - RESTRIÇÕES DE INTEGRIDADE.....	40
4.18 - TIPOS DE DADOS	41
4.19 - Normalização de dados.....	42
4.20 - Criação de um Banco de Dados Relacional com base a um modelo ER	43
4.21 - Linguagem de definição de dados (DDL).....	44
5 - Comandos Básicos para Manipulação de Dados.....	47
5.1 – INSERT	47
5.2 - UPDATE	48
5.3 – DELETE	49
5.4 – SELECT	49
5.5 - Conhecendo os Tipos de JOIN.....	50

5.6 - INNER JOIN	51
5.7 - LEFT OUTER JOIN	51
5.8 - RIGHT OUTER JOIN	51
5.9 - FULL OUTER JOIN	52
5.10 - Operadores de Comparação	52
5.11 - Operadores Lógicos.....	53
5.12 - Stored Procedures.....	53
5.13 - Functions	55
5.14 - Triggers.....	56
5.15 - Algumas Funções do T-SQL	57
6 - O que é Git?.....	57
6.1 - Repositórios	58
6.2 - Instalando o GIT	59
6.3 - Comandos Básicos.....	60
6.4 - Commitando e atualizando seus projetos	61
7.1 - Por que utilizar MVC?.....	65
7.2 - Exemplo do funcionamento do MVC	66
7.3 - ViewData, ViewBag e TempData.....	67
7.4 - Rotas.....	70
7.5 - Passando parâmetros aos Métodos.....	72

1 - Lógica de Programação

- Poder ser definida como um conjunto de técnicas para encadear pensamentos a fim de atingir determinados objetivos.
- Lógica de programação pode ser aplicada para qualquer linguagem.
- Principal objetivo é fazer você pensar de forma racional.
- Impossível ser um desenvolvedor de sistemas sem a lógica de programação.
- A lógica é o primeiro nível (Planejamento).
- Após ter a lógica, você pode aplicá-la na programação usando a linguagem desejada

1.1 - As variáveis

São utilizadas para armazenar informações na memória em tempo de execução da aplicação, isso significa que essas informações estarão disponíveis enquanto a aplicação estiver em execução e, mais precisamente, enquanto a classe ou método em que ela foi criada estiver em execução.

No C# toda variável deve ter: modificador de acesso, tipo de dados e nome.

A regra para nomear uma variável é que o nome dela sempre comece 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. Entende-se por palavras reservadas os comandos do C# e que são facilmente identificadas quando digitadas, por ficarem da cor azul. Exemplos de palavras que não podem ser utilizadas são: `if`, `for`, `while`, `string` e etc.

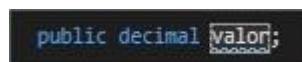


```
decimal valor;
```

- No exemplo, `decimal` é o tipo e `valor` é o nome da variável. Mas, e o modificador?
- Como dito anteriormente, toda variável deve ter o modificador de acesso.
- No exemplo acima não tem um modificador porque no C#, quando não há um modificador em uma variável é atribuído a ele o modificador padrão *private*.

1.2 - Modificadores

No caso das variáveis, os modificadores definem a visibilidade delas, se elas poderão ser acessadas por outras classes que não seja a sua própria, se serão acessadas somente por classes derivadas a classe que ela está e assim por diante, como mostra a seguir.



```
public decimal valor;
```

Modificador	Funcionamento
Public	O acesso não é restrito
Protected	O acesso é limitado às classes ou tipos derivados da classe que a variável está
Internal	O acesso é limitado ao conjunto de módulos(assembly) corrente.
Protected internal	O acesso é limitado ao conjunto corrente ou tipos derivados da classe recipiente
Private	O acesso é limitado a classe que a variável está

No C# existem os seguintes modificadores:

1.3 - Tipos de dados

- C# é uma linguagem de programação fortemente tipada, isso significa que todas as variáveis e objetos devem ter um tipo declarado. Os tipos de dados são divididos em value types e reference types.
- Os value types são derivados de System.ValueType e reference types são derivados de System.Object

Variáveis Value Type

- As variáveis value type contém dentro delas um valor, enquanto as reference type contém uma referência. Isso significa que se copiar uma variável do tipo value type para dentro de outra o valor é copiado e, se o mesmo for feito com uma do tipo reference type será copiado apenas a referência do objeto.

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

- a) Struct: é dividida em tipos numéricos, bool e estruturas personalizadas pelo usuário.
- Tipo bool representa um valor que pode ser verdadeiro ou falso, o seu valor padrão é false.

Os tipos numéricos são os presentes na tabela abaixo:

Tipo de dados	Intervalo
Byte	0...255
Sbyte	-128...127
Short	32.768...32.767
Ushort	0...65,535
Int	-2.147.483,648...2.147.483,647
UInt	0...4,294,967,295
Long	-9.223.372.036.854.755,808... -9.223.372.036.854.755,807
Ulong	0...18.446.744.073.709.551,615
Float	-3.402.823e38...3.402.823e32
Double	-1.797.693.134.862.32e308...1.79769313486232e308
Decimal	-79228162514264337593543950335...79228162514264337593543950335
Char	U+0000...U+ffff
Float	-3.402823e38...3.402823e38
Double	-1.79769313486232e308...1.79769313486232e308

b) **Enum**: permite criar um tipo que é formado por várias constantes.

- Normalmente é usada quando em algum momento existe a necessidade de um atributo que pode ter múltiplos valores, como por exemplo, em uma aplicação de venda que tem a tela de pedidos: cada pedido tem a sua situação que pode ser Aberto, Faturado e Cancelado. Para criar a classe Pedido, o tipo do atributo situação será int, conforme mostra a seguir:

```
public class Pedido
{
    public int numero;
    public DateTime dataHora;
    public int situacao;
}
```

Agora é necessária a criação do Enum Situacao. Um Enum é criado da mesma forma que se cria uma classe, mas ao invés de utilizar a palavra class usa-se enum, conforme mostra a seguir.

```
public enum Situacao
{
    Aberto,
    Faturado,
    Cancelado
}
```

O Enum por padrão retorna int e neste exemplo temos três opções, onde cada uma possui um índice. Então quando o valor de um Enum é atribuído a um outro atributo, o que é atribuído é seu índice, no caso do exemplo é atribuído o valor 1.

```
class Program
{
    static void Main(string[] args)
    {
        Pedido pedido = new Pedido();

        pedido.numero = 1;
        pedido.dataHora = DateTime.Now;
        pedido.situacao = (int)Situacao.Faturado;

        Console.WriteLine("Número do pedido: "
            + pedido.numero);
        Console.WriteLine("Número do pedido: "
            + pedido.dataHora);
        Console.WriteLine("Número do pedido: "
            + pedido.situacao);

        Console.ReadLine();
    }
}
```

1.4 - Variáveis Reference Types

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

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

```
string nome = "CursoSMN";
```


Para concatenar (juntar) uma ou mais strings é usando o sinal de +, como mostra a seguir

```
string a = "C#";  
string b = ".net";  
string c = a + b;
```

Outro recurso também é a possibilidade de se obter um determinado caractere de uma string, como mostra a seguir:

```
string a = "C#";  
string b = ".net";  
string c = a + b;  
char d = c[3];
```

- A variável d vai ficar com o valor da letra n que é a letra que está no índice 3, conforme informado no código.
- Também é possível fazer a mesma coisa com uma palavra, como mostra a seguir.

```
char d = "C#.net"[3];
```

Começando pelos métodos com membros estáticos, vamos falar do **Format**. Este método é uma mão na roda para os desenvolvedores que desejam formatar seus textos sem muitas preocupações, de uma forma rápida e simples.

Com o método Format criamos um padrão de formatação para a nossa string. Assim, dentro de nossa string podemos colocar alguns “tokens” que serão substituídos por parâmetros, que acrescentaremos ao final.

```
Console.WriteLine(String.Format("Id: {0}\nNome: {1}\nIdade: {2}", 1, "Gustavo Dantas", 20));
```

O método Format pós a primeira vírgula, como no código acima monta a lógica de substituir os “tokens” pelos respectivos parâmetros, que são passados logo acima. Vemos o resultado a seguir:

```
Id: 1  
Nome: Gustavo Dantas  
Idade: 20
```

Segue uma tabela com as formatações de números.

Especificador	Formato	Tipo	Saída Double(1000.00)	Saída int(-1000)
c	{0:c}	Moeda	R\$ 1.000,00	-R\$ 1.000,00
d	{0:d}	Decimal	System.FormatException	-1000
e	{0:e}	Expoente/Científico	1,000000e+003	-1,000000e003
f	{0:f}	Ponto fixo	1000,00	-1000
g	{0:g}	Geral	1.000,00	-1.000,00
n	{0:n}	Número	1.000,00	-1.000,00
r	{0:r}	Round trippable	1000	System.FormatException
x	{0:x4}	hexadecimal	System.FormatException	ffffc18

Abaixo tem uma tabela com as possibilidades de formatos de datas.

Especificador	Formato	Tipo	Saída
d	{0:d}	Data Curto	21/12/2017
D	{0:D}	Data Longa	Quinta-feira, 21 de dezembro de 2017
t	{0:t}	Hora curta	13:06
T	{0:T}	Hora longa	13:06:33
f	{0:f}	Data complete e hora	Quinta-feira, 21 de dezembro de 2017 13:07:30
F	{0:F}	Data complete e hora longa	Quinta-feira, 21 de dezembro de 2017 13:08:30
g	{0:g}	Data padrão e hora	21/12/2017 13:09
G	{0:G}	Data padrão e hora longa	21/12/2017 13:10:33
M	{0:M}	Dia / Mês	08 de julho

r	{0:r}	RFC1123 data escrita	Qui, 21 Dez 2018 13:11:40 GMT-3
s	{0:s}	Data Hora classificada	2017-12-21T 13:12:33
u	{0:u}	Hora Universal, local timezone	2017-12-21T 13:13:33z
Y	{0:Y}	Mês / Ano	Dezembro de 2017

Temos também formatações personalizadas para datas.

Especificador	Formato	Tipo	Saida
dd	{0:dd}	Dia	21
ddd	{0:ddd}	Nome curto do dia	Qui
dddd	{0:dddd}	Nome inteiro do dia	Quinta-feira
hh	{0:hh}	Hora com 2 dígitos	21
HH	{0:HH}	Hora com 2 dig{24h}	13
mm	{0:mm}	Minutos com 2 dígitos	Quinta-feira
MM	{0:MM}	Mês	12
MMM	{0:MMM}	Nome curto do mês	Dez
MMMM	{0:MMMM}	Nome inteiro do mês	Dezembro
ss	{0:ss}	Segundos	40
tt	{0:tt}	AM/PM	PM
yy	{0:yy}	Ano com 2 dígitos	2018
yyyy	{0:yyyy}	Ano com 4 dígitos	2018
:	{0:dd:MM:yyyy}	Separador	13:15:00
-	{0:dd-MM-yyyy}	Separador	21-12-2017

/	{0:dd/mm/yyyy}	Separador	21/12/2017
---	----------------	-----------	------------

Temos também formatação personalizada para números.

Especificador	Formato	Tipo	Saída(1000)
0	{0:00.000}	Zero de espaços reservados	1.234.560
#	{0:#.##}	Espepaço reservado para dígito	1234.56
.	{0:0.0}	Espaço reservado ponto decimal	1234.6
,	{0:0,0}	Separador de milhar	1,235
%	{0.0%}	Porcentagem	123456%

1.5 - String.IsNullOrEmpty

Muito útil para validar se o valor de uma variável é nulo ou vazio.

```
string nome = "";
if (String.IsNullOrEmpty(nome))
{
    Console.WriteLine("String vazia");
}
```

Simple assim, como ele é do tipo bool a verificação se torna bem simples. Caso desejássemos fazer a negativa do código acima, bastaria adicionar uma exclamação (!) antes da chamada ao método.

1.6 - Contains

Este método é parecido com o operador LIKE, da linguagem SQL. Sua função é verificar se uma determinada string contém o valor informado em seu parâmetro.

```
string nome = "CursoSMN";
if (nome.Contains("SMN"))
{
    Console.WriteLine("A string contem: SMN");
}
```

Veja o resultado a seguir:

```
A string contem: SMN
Pressione qualquer tecla para continuar. . .
```

1.7 - Equals

Este método é a forma elegante (boas práticas!) de usarmos o operador de comparação (==) em nossas instruções condicionais. Com ele verificamos, por exemplo, se uma string é igual a outra.

```
string nome = "CursoSMN";
if (nome.Equals("CursoSMN"))
{
    Console.WriteLine("O texto digitado esta correto");
}
```

Se a palavra for "CursoSMN" mostra a mensagem a seguir:

```
O texto digitado esta correto
Pressione qualquer tecla para continuar. . .
```

1.8 - Remove

O método Remove, como o próprio nome sugere, serve para remover os caracteres de uma determinada string. Em uma de suas sobrecargas ele pede como parâmetro o índice de início, do tipo int, o qual a partir dele tudo será removido até o final. Veja como remover "SMN C#", de "Curso SMN C#";

```
string nome = "Curso SMN C#";
Console.WriteLine(nome.Remove(5));
```

Assim, a partir do índice 5 (lembrando que sempre começa em 0) tudo é removido.

```
Curso
Pressione qualquer tecla para continuar. . .
```

1.9 - Replace

Outro método muito usado, o Replace é muito útil para substituir um trecho de uma string por outro valor.

```
string nome = "CursoSMN node";
if(nome.Equals("CursoSMN node"))
{
    Console.WriteLine(nome.Replace("node", "c#"));
}
```

Resultado do método Replace:

```
CursoSMN c#
Pressione qualquer tecla para continuar. . .
```

1.10 - Substring

Com este método conseguimos recuperar partes de uma string, apenas informando o índice de início (em uma das sobrecargas), semelhante a como foi feito com o método Remove. Veja como pegar trechos de uma frase qualquer,

```
string nome = "Curso SMN C#";  
Console.WriteLine(nome.Substring(6));
```

Resultado do método Substring:

```
SMN C#  
Pressione qualquer tecla para continuar. . .
```

1.11 - ToLower

Com este método conseguimos converter qualquer string para letras minúsculas. Veja como é bem simples seu uso:

```
string nome = "CURSO SMN C#";  
Console.WriteLine(nome.ToLower());
```

Fazendo a string ficar minúscula

```
curso smn c#  
Pressione qualquer tecla para continuar. . .
```

1.12 - ToUpper

Função oposta ao método ToLower, que faz toda a string ficar maiúscula. Veja o exemplo:

```
string nome = "curso smn c#";  
Console.WriteLine(nome.ToUpper());
```

Fazendo a string ficar maiúscula

```
CURSO SMN C#  
Pressione qualquer tecla para continuar. . .
```

1.13 - Split

Split – O Split é usado para dividir uma string em pequenos pedaços. Para isso, seu método retorna um array de strings, contendo as respectivas partes, definidas de acordo com a string passada como parâmetro, que na verdade funciona como o “agente” divisor da mesma.

De uma forma resumida pense na string “micro-ondas”. Supondo que desejamos dividi-la pelo hífen, teríamos que fazer a implementação ilustrado a seguir:

```
string texto = "micro-ondas";  
string[] retornoSplit = texto.Split('-');  
Console.WriteLine(String.Format("Posição 0: {0} \nPosição 1: {1}", retornoSplit[0], retornoSplit[1]));
```

Note que o método Split espera um char como parâmetro, por isso são usadas as aspas simples.

```
Posição 0: micro  
Posição 1: ondas  
Pressione qualquer tecla para continuar. . .
```

1.14 - Operadores Relacionais – Os operadores relacionais comparam dois valores e retornam um valor booleano (**true** ou **false**). Veja na tabela abaixo esses operadores:

Operador Relacional	Descrição
==	Igual a
!=	Diferente de
>	Maior que
<	Menor que
>=	Maior do que ou igual a
<=	Menor do que ou igual a

O exemplo a seguir ajuda a ilustrar o uso destes operadores:

```
int a = 10, b = 25, c = 50, d = 100;  
Console.WriteLine(a == d); //false  
Console.WriteLine(b != c); //true  
Console.WriteLine(a > b); //false  
Console.WriteLine(c < d); //true  
Console.WriteLine(c >= a); //true  
Console.WriteLine(d <= b); //false
```

1.15 Operadores Lógicos – Os operadores lógicos trabalham como operandos booleanos e seu resultado também será booleano (true ou false). Eles são usados somente em expressões lógicas, e são descritos na tabela a seguir:

Operador Lógico	Descrição
&&	AND = E
	OR = Ou
!	NOT = Não

Assim, em um teste lógico usando o operador && (AND), o resultado somente será verdadeiro (true) se todas as expressões lógicas forem avaliadas como verdadeiras.

Já, se o operador usado for o || (OR), basta que apenas uma das expressões lógicas seja verdadeira para que o resultado também seja verdadeiro.

Completando, o operador lógico ! (NOT) é usado para gerar uma negação. Desta forma, é invertida toda a lógica da expressão.

Veja o exemplo e o resultado a seguir:

```
int a = 5, b = 10, c = 15, d = 20;

Console.WriteLine(a == 5 && d == 10); //false
Console.WriteLine(c < b || d == 20); //true
Console.WriteLine(!(b > a)); //false
```

1.16 - If/else

A estrutura condicional if/else é um recurso que indica quais instruções o sistema deve processar de acordo com uma expressão booleana. Assim, o sistema testa se uma condição é verdadeira e então executa comandos de acordo com esse resultado.

Sintaxe do if/else:

```
if (expressao_booleana)
{
    //codigo 1
}
else
{
    //codigo 2
}
```

Caso a expressão booleana seja verdadeira, as instruções entre chaves presentes no código 1 serão executadas; caso contrário, serão executadas as instruções presentes no código 2.

As chaves, ou delimitadores de bloco, têm a função de agrupar um conjunto de instruções. O uso desses delimitadores é opcional caso haja apenas uma linha de código. Ainda assim, seu uso é recomendado, pois essa prática facilita a leitura e manutenção do código, tornando-o mais legível.

else if

Complementar ao if/else temos o operador else if que traz uma nova condição a ser testada no caso de falha no teste da condição anterior.

Sintaxe do else if:

```
if (expressao_booleana_1)
{
    //codigo 1
}
else if (expressao_booleana_2)
{
    //código 2
}
else
{
    //codigo 3
}
```

Para demonstrar a estrutura condicional if/else, considere um programa que verifica se um aluno foi aprovado, se está em recuperação ou se foi reprovado. Para ser aprovado, o aluno precisa ter média maior ou igual (\geq) a 7. Para a recuperação, ele precisa ter média menor ($<$) que 7 e ($\&\&$) maior ou igual (\geq) a 5. Por fim, para ser reprovado, precisa ter média menor ($<$) do que 5.

Note que temos três condições a serem verificadas, o que nos leva à necessidade de uso do if, else if, else.

Exemplo de uso:

```
double media = 8;
if (media >= 7)
{
    Console.WriteLine("Aluno aprovado!");
}
else if (media < 7 && media >= 5)
{
    Console.WriteLine("Aluno em recuperação!");
}
else
{
    Console.WriteLine("Aluno reprovado!");
}
```

1.17 - Operador ternário

O operador ternário é um recurso para tomada de decisões com objetivo similar ao do if/else, mas que é codificado em apenas uma linha.

Sintaxe do operador ternário:

```
int valor1 = 10;
int valor2 = 10;

bool resultado = valor1 == valor2 ? true : false;
```

Para demonstrar um exemplo de uso do operador ternário, suponha que precisamos apresentar ao aluno o resultado com base na sua média. Para isso, criamos uma variável que recebe um texto padrão e que depois será incrementado para informar se o aluno está aprovado ou reprovado. Essa concatenação será feita a partir do resultado obtido com o operador ternário, que, neste caso, também nos devolve um texto.

Exemplo de uso:

```
double media = 8;
string resultado = "Olá aluno, voce foi ";
resultado += media >= 7 ? "aprovado" : "reprovado";
Console.WriteLine(resultado);
```

1.18 - For

Na expressão de iteração, indicamos **o que** deverá ocorrer após cada execução do corpo do laço de repetição. Ela sempre deverá ser processada após o corpo do laço ser executado, mas será a **última** execução da instrução do laço **for**.

```
//for(declaração e inicialização de variavel; condição; iteração;)
{
    //instrução do corpo do laço for;
}
```

Agora veja o seguinte exemplo, usando a sintaxe acima:

```
for (int a = 10; a > 5; a--)  
{  
    Console.WriteLine("CursoSMN");  
}
```

Onde:

- **int** refere-se ao tipo de variável que será declarada;
- **a = 10** refere-se às variáveis que estão sendo inicializadas;
- **a < 5** refere-se ao teste booleano que será feito na variável a;
- **a--** indica a iteração da variável a.

Após verificarmos todas essas informações, destacamos que, apesar de **não ser uma boa prática**, podemos deixar de declarar uma das três partes descritas no laço de repetição **for** (as duas primeiras no artigo anterior). Nesse caso, o laço será infinito, e não tendo seções de declaração e inicialização no laço de repetição **for** ele agirá como um laço de repetição **while**.

1.19 - Break

O comando **break** é usado em laços de repetição **while**, **do/while**, **for** e com os comandos **switch/case**. Quando usado em laço de repetição, causa uma interrupção imediata do mesmo, continuando a execução do programa na próxima linha após o laço. Isso ocorre caso a condição imposta seja atendida.

```
for (int i = 1; i < 10; i++)  
{  
    if (i == 5)  
    {  
        break; //interrompe o loop  
    }  
    Console.WriteLine("Valor de i = " + i);  
}  
Console.WriteLine("Próxima linha após o loop...");
```

1.20 - While

O **while** trata-se da estrutura de repetição mais utilizada quando programamos com C#. Com ela, enquanto a condição for verdadeira o bloco de código será executado. Primeiramente o sistema testa essa condição. Caso verdadeira, executa as linhas declaradas dentro do **while**; do contrário, sai do loop.

Sintaxe da estrutura de repetição **while**:

```
while (condicao)  
{  
    //bloco de código  
}
```

Exemplo prático

Para demonstrar o funcionamento do while, criamos um loop que imprimirá no console os valores de 1 a 10.

Exemplo de uso:

```
int i = 1;
while (i <= 10)
{
    Console.WriteLine(i);
    i++;
}
```

Break

Por padrão um loop só é finalizado quando a condição de parada é atendida. Porém é possível interromper a execução de um laço de repetição utilizando o comando break. Esse comando não espera nenhum parâmetro, portanto basta informar a palavra reservada break na linha em que o loop deve ser interrompido.

No código abaixo o laço deveria ser executado enquanto a variável i fosse menor que 10. Porém, usamos o comando break para interromper o loop caso i seja igual a 5. Neste caso, as instruções não serão executadas, então o resultado desse código será a impressão dos valores 1 a 4.

```
int i = 1;
while (i < 10)
{
    if (i == 5)
        break;
    Console.WriteLine(i);
    i++;
}
```

1.21 - Continue

Além do break há outro comando capaz de modificar o fluxo natural de execução de loop: o continue. Esse comando faz com que a execução da iteração/repetição seja finalizada e o fluxo seja direcionado novamente para o início do loop. Dessa forma a condição de parada será novamente avaliada e o loop será executado novamente.

```
int i = 0;
while (i < 10)
{
    i++;
    if (i % 2 == 0)
        continue;
    Console.WriteLine(i);
    i++;
}
```

1.22 - Switch/Case

Switch/case é uma estrutura de condição que define o código a ser executado com base em uma comparação de valores.

Para que isso fique mais claro, vejamos a sintaxe do switch/case:

```
switch(variavel_ou_valor)
{
    case false:
        //código 1
        break;

    case true:
        //código 2
        break;
}
```

Em switch (variável_ou_valor), definimos a variável ou valor que desejamos comparar. No primeiro case informamos se o valor declarado neste case for igual ao contido no switch, será executado. O mesmo comportamento se aplica ao segundo case. Ademais, caso o valor contido no switch não seja atendido em uma das condições, nenhum bloco será executado.

E o comando break? O comando break é utilizado para especificar a última linha de código a ser executada dentro da condição. Se não declarado, os códigos implementados dentro dos cases subsequentes serão executados.

Caso deseje executar o mesmo código para valores diferentes, você pode programar o switch/case da seguinte forma:

```
switch (variavel_ou_valor)
{
    case "Gustavo":
    case "Flavia":
    case "Alberto":
        //código 1
        break;

    case "Marcelo":
    case "Fabio":
    case "Daniel":
        //código 2
        break;
}
```

1.23 - Default

O operador default é utilizado quando precisamos definir um fluxo alternativo para as situações em que o valor contido no switch não seja atendido por nenhum dos **cases** especificados.

```
switch (variavel_ou_valor)
{
    case "Gustavo":
        //código 1
        break;
    case "Daniel":
        //código 2
        break;
    default:
        //código 3
        break;
}
```

Exemplo prático

Para demonstrar um exemplo de switch/case, considere um programa que informa ao usuário a quantidade de dias de um mês a partir do nome de um mês por ele informado.

Com esse intuito, a partir de uma variável do tipo string (**mês**) implementamos o código abaixo:

```
switch (mes)
{
    case "Janeiro":
    case "Março":
    case "Maio":
    case "Julho":
    case "Agosto":
    case "Outubro":
    case "Dezembro":
        Console.WriteLine("Este mês tem 31 dias");
        break;
    case "Fevereiro":
        Console.WriteLine("Este mês tem 28 ou 29 dias");
        break;
    default:
        Console.WriteLine("Este mês tem 30 dias");
        break;
}
```

1.24 - Foreach

O foreach é um recurso do C# que possibilita executar um conjunto de comandos para cada elemento presente em uma coleção (Array, List, Stack, Queue e outras). Portanto, diferentemente do while e do for, não precisamos definir uma condição de parada. Isso é definido de forma implícita, pelo tamanho da coleção.

Sintaxe da estrutura de repetição foreach:

```
foreach (string nome in viagem)
{
    //bloco de código
}
```

Exemplo prático

Considere que desejamos imprimir na tela todos os nomes presentes em um array. Para isso, em vez de criar um while e nos preocuparmos com a condição de parada, podemos fazer uso do foreach.

```
string[] nomes = {"Andre", "Bruna", "Carla", "Daniel"};
foreach (string nome in nomes)
{
    Console.WriteLine(nome);
}
```

1.25 - Declaração de arrays

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.

A declaração padrão para um array qualquer obedece a seguinte estrutura

```
//tipo[] nomeDoArray = new tipo[tamanho_do_array];
```

Por exemplo, para um array do tipo inteiro (int), com 10 posições, a declaração seria da seguinte forma:

```
int[] array = new int[10];
```

Inserção de dados no array

Existem duas formas básicas para inserirmos dados nos arrays em C#. A primeira é realizarmos isso durante a declaração do mesmo. Dessa forma, poderíamos ter a declaração das seguintes formas:

```
int[] array1 = new int[5] {1, 3, 7, 12, 8};
int[] array2 = {1, 3, 2, 7, 6};
```

No primeiro caso temos uma declaração explícita de um array que conterá cinco números inteiros. Aqui, se passarmos menos ou mais de cinco números entre as chaves haverá um erro de compilação, uma vez que sabemos que o array tem cinco posições.

No segundo caso, a declaração da quantidade de elementos no array é implícita e depende de quantos itens temos entre as chaves. Nesse exemplo passamos cinco números; assim, array2 possui cinco posições e não há possibilidade de erro de compilação.

Outra forma de inserirmos os dados no array é acessando o item através de seu índice. Por exemplo, podemos fazer uma declaração simples e então iterarmos sobre todos os índices do array, adicionando os valores específicos para cada um deles, como a seguir:

```
int[] array = new int[50];  
for (int i = 0; i < 50; i++)  
{  
    array[i] = i + i;  
}
```

1.26 - Acesso aos dados do array

O acesso aos dados do array é feito de forma extremamente simples, uma vez que estamos lidando com uma coleção indexada. Assim, basta acessarmos o índice específico e podemos obter o valor do elemento, como podemos ver abaixo:

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

Iterações sobre os arrays

Uma das principais características das coleções, em qualquer linguagem de programação, são as iterações que podem ser realizadas sobre elas. No caso dos arrays em C# isso não é diferente. Assim, 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 para nós. No código abaixo temos ambas as opções:

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


Exemplo prático

```
string[] pilotos = new string[4] { "Ayrton Senna", "Michael Schumacher", "Lewis Hamilton", "Alain Prost" };
Console.WriteLine(pilotos[3]);
Console.WriteLine();
pilotos[3] = "Rubens Barrichello";
foreach (string piloto in pilotos)
{
    Console.WriteLine(piloto);
}
```

1.27 - Usando Expressões Lambda com LINQ

LINQ é uma linguagem de consulta que foi introduzida a partir do .Net Framework 3.5 e é utilizada na manipulação de coleções, sejam elas vindas de um banco de dados, documentos XML, coleções de objetos, etc. Assim, podemos definir o LINQ como uma consulta que não tem dependência da fonte de dados.

Agora vamos criar um cenário para que possamos demonstrar a utilização do LINQ e a melhora quando utilizamos lambda junto a ele. Vamos criar uma lista do tipo pessoa; pessoa é um tipo de dado com as propriedades Nome, Idade, Cidade e Sexo.

```
public class Pessoa
{
    public string Nome { get; set; }
    public int Idade { get; set; }
    public string Cidade { get; set; }
    public string Sexo { get; set; }

    public Pessoa(string _Nome, int _Idade,
        string _Cidade, string _Sexo)
    {
        Nome = _Nome;
        Idade = _Idade;
        Cidade = _Cidade;
        Sexo = _Sexo;
    }

    public override string ToString()
    {
        return "Nome: " + Nome + " Idade: " + Idade + " Cidade: " + Cidade + " Sexo: " + Sexo;
    }
}
```

No mesmo arquivo Pessoa.cs vamos também implementar uma classe que é uma lista de pessoa. Essa classe conterá apenas um construtor sem parâmetros que servirá para alimentarmos nossa lista de pessoas, ou seja, ao instanciarmos essa lista ela já estará populada.

```

class IPessoa : List<Pessoa>
{
    public IPessoa()
    {
        this.Add(new Pessoa("JOSÉ SILVA", 21, "PIRACICABA", "M"));
        this.Add(new Pessoa("ADRIANA GOMES", 18, "CURITIBA", "F"));
        this.Add(new Pessoa("JOSÉ NOVAES", 17, "CAMPINAS", "M"));
        this.Add(new Pessoa("ANDRÉ NEVES", 50, "PIRACICABA", "M"));
        this.Add(new Pessoa("JONATHAN SANTOS", 45, "CAMPINAS", "M"));
        this.Add(new Pessoa("IVANILDA RIBEIRO", 29, "AMERICANA", "F"));
        this.Add(new Pessoa("BRUNA STEVAN", 28, "AMERICANA", "M"));
        this.Add(new Pessoa("ANDREIA MARTINS", 22,
            "RIO DE JANEIRO", "F"));
        this.Add(new Pessoa("ANTONIO DA SILVA", 30,
            "POÇOS DE CALDAS", "M"));
        this.Add(new Pessoa("JOSEFINO DANTAS", 48,
            "CAMPOS DO JORDÃO", "M"));
        this.Add(new Pessoa("HELENA MARIA GOMES", 25,
            "SÃO PAULO", "F"));
        this.Add(new Pessoa("DJONATHAN MASTRODI", 29,
            "PIRACICABA", "M"));
        this.Add(new Pessoa("BRUNO MARTINO", 55, "AMERICANA", "M"));
        this.Add(new Pessoa("ANA MARIA GONÇALVES", 60,
            "PRAIA GRANDE", "F"));
        this.Add(new Pessoa("MARCELA MARIA RIBEIRO", 85,
            "JOAO PESSOA", "F"));
        this.Add(new Pessoa("ISABELLA MENDES", 32, "PIRACICABA", "F"));
        this.Add(new Pessoa("IGOR MENDES", 22, "CAMPINAS", "M"));
        this.Add(new Pessoa("ANTONIO JOSÉ FARIAS", 21,
            "RIO DAS PEDRAS", "M"));
        this.Add(new Pessoa("JULIO VIEIRA", 19, "ANDRADINA", "M"));
        this.Add(new Pessoa("MARIO RIBEIRO", 13, "LIMEIRA", "M"));
    }
}

```

Agora que já criamos as classes necessárias, veremos a utilização de lambda com LINQ. A cada exemplo mostraremos a forma com e sem lambda para que você possa ver as diferenças entre os dois tipos.

Primeiro vamos utilizar a nossa lista de pessoas, porém deveremos retornar na tela somente as pessoas que a primeira letra do Nome é "A". Para isso devemos implementar a rotina utilizando Expressões Lambda.

Exemplo de lambda com LINQ

```

IPessoa _IPessoa = new IPessoa();

foreach (var item in _IPessoa.Where
    (p => p.Nome.StartsWith("A")))
{
    Console.WriteLine(item.ToString());
}

```

```

Nome: ADRIANA GOMES Idade: 18 Cidade: CURITIBA Sexo: F
Nome: ANDRÉ NEVES Idade: 50 Cidade: PIRACICABA Sexo: M
Nome: ANDREIA MARTINS Idade: 22 Cidade: RIO DE JANEIRO Sexo: F
Nome: ANTONIO DA SILVA Idade: 30 Cidade: POÇOS DE CALDAS Sexo: M
Nome: ANA MARIA GONÇALVES Idade: 60 Cidade: PRAIA GRANDE Sexo: F
Nome: ANTONIO JOSÉ FARIAS Idade: 21 Cidade: RIO DAS PEDRAS Sexo: M
Pressione qualquer tecla para continuar. . .

```

Exemplo com LINQ

```
IPessoa _IPessoa = new IPessoa();  
var pessoa = from p in _IPessoa  
              where p.Nome.StartsWith("A")  
              select p;  
  
foreach (var item in pessoa)  
{  
    Console.WriteLine(item.ToString());  
}
```

```
Nome: ADRIANA GOMES Idade: 18 Cidade: CURITIBA Sexo: F  
Nome: ANDRÉ NEVES Idade: 50 Cidade: PIRACICABA Sexo: M  
Nome: ANDREIA MARTINS Idade: 22 Cidade: RIO DE JANEIRO Sexo: F  
Nome: ANTONIO DA SILVA Idade: 30 Cidade: POÇOS DE CALDAS Sexo: M  
Nome: ANA MARIA GONÇALVES Idade: 60 Cidade: PRAIA GRANDE Sexo: F  
Nome: ANTONIO JOSÉ FARIAS Idade: 21 Cidade: RIO DAS PEDRAS Sexo: M  
Pressione qualquer tecla para continuar. . .
```

Exercícios

1. Criar um algoritmo que imprima a média aritmética entre os infinitos números informados pelo usuário.
2. Fazer um algoritmo que recebe uma palavra e verifica se ela é um palíndromo.
3. Faça um programa para calcular o MDC (máximo divisor comum) entre dois números.

Aconselho a utilizar o método de divisões sucessivas, pois você verá que a forma como costumamos resolver problemas matemáticos na mão nem sempre é o melhor caminho quando estamos codificando.

4. Fazer um programa para calcular a média aritmética entre todos os números PARES (informados pelo usuário, usuário escolhe quantos números quer informar).
5. Em uma matriz 3x3 desenhe um X
Exemplo: 101
 010
 101
Faça uma 5x5 também.

6. Dado um vetor qualquer sem valores repetidos, descubra qual é o índice do maior valor e o índice do menor valor.
Repare que eu quero o índice e não o valor daquele índice.

7. Dados dois strings (um contendo uma frase e outro contendo uma palavra), determine o número de vezes que a palavra ocorre na frase.

Exemplo:

Para a palavra ANA e a frase:

ANA E MARIANA GOSTAM DE BANANA

Temos a palavra 4 vezes na frase.

2 - Conceitos de C#

2.1 Introdução

A linguagem C# (lê-se “cêsharp”) foi criada juntamente com a arquitetura da plataforma .NET da Microsoft. A linguagem C# foi influenciada por várias linguagens, como como o JAVA e C++. É uma junção das principais vantagens dentre essas linguagens, melhorando suas implementações e adicionando novos recursos. C# foi feita a partir do zero para funcionar especialmente na plataforma .NET.

Sua sintaxe é simples e de fácil aprendizagem, bem parecida com a sintaxe de JAVA e C. É orientada a objetos. Seus métodos e classes não precisam ser declarados em ordem. Case-sensitive, diferencia letras maiúsculas de minúsculas.

É fortemente tipada, ou seja, para trabalhar com cada tipo de dados as variáveis tem o tipo declaradas o que ajuda a evitar erros oriundos de uma manipulação imprópria de tipo e/ou atribuições

Você pode usar C# para criar aplicativos de cliente do Windows, serviços Web XML, componentes distribuídos, aplicativos cliente-servidor, aplicativos de banco de dados e muitos outros.

2.2 Variáveis

Na maioria dos programas que desenvolvemos, não estamos interessados em mostrar apenas mensagens para os usuários. Queremos também armazenar informações.

Toda variável aloca uma quantidade de memória, é nesse espaço de memória que está armazenado o conteúdo da variável, internamente uma variável possui um ponteiro, o ponteiro para o sistema operacional é um endereçamento físico de memória, serve para localizar onde está armazenado tal dado.

Na linguagem de programação C# as variáveis são tipadas, ou seja, guardam informações de um tipo específico. Podemos, por exemplo, guardar um número inteiro(int) representando a idade, um

texto(string) para representar o nome da pessoa ou um número real(decimal) para representar o seu saldo no banco.

Para utilizar uma variável, devemos primeiramente declará-la no programa. Na declaração de uma variável, devemos dizer seu tipo (inteiro, texto ou real, por exemplo) e, além disso, qual é o nome que usaremos para referenciá-la no texto do programa.

Tipo NomeDaVariavel

```
1 int idade;
```

***use sempre ponto e vírgula no final de cada comando.**

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

```
1 int idade;  
2 decimal saldo;  
3 string nome;
```

Depois que for declarada, a variável pode ser utilizada para guardar valores. Por exemplo, se quiséssemos armazenar o valor 1 na variável idade que declaramos anteriormente, iríamos utilizar o seguinte código:

```
1 idade = 1;
```

Se soubermos qual será o valor da variável no momento de sua declaração, podemos utilizar a seguinte sintaxe para declarar e atribuir o valor para a variável:

```
1 int idade = 19;
```

Podemos executar operações com esses valores, por exemplo, vamos adicionar 15 reais ao saldo:

```
1 decimal saldo = 80.0;  
2 saldo = saldo + 15.0;
```

Nesse código, estamos guardando na variável saldo o valor de 80.0 (saldo antigo) mais 15.0 então seu valor final será de 95.0. Da mesma forma que podemos somar valores, podemos também fazer subtrações, multiplicações e divisões. Conheça os principais operadores e seus diferentes tipos:

```

1 //Operadores matemáticos
2 + (Adição)
3 - (Subtração)
4 * (Multiplicação)
5 / (Divisão)
6 % (Resto/Módulo)
7 //Usado para operações matemáticas
8
9 //Operadores de Atribuição
10 = (Atribuição simples)
11 += (Atribuição aditiva)
12 -= (Atribuição Subtrativa)
13 *= (Atribuição Multiplicativa)
14 /= (Atribuição de divisão)
15 %= (Atribuição de módulo)
16 //Um operador de atribuição serve para atribuímos um valor a uma variável.
17
18 //Operadores relacionais
19 == (Igualdade)
20 > (Maior)
21 < (Menor)
22 <= (Menor igual)
23 >= (Maior igual)
24 != (Diferente)
25 //Uma expressão relacional retorna um valor Booleano, verdadeiro ou falso,
26
27 //Operadores lógicos
28 && (E)
29 || (OU)
30 //Os operadores lógicos são usados para combinar comparações.

```

Podemos também manipular textos(strings). O operador de adição em strings executa uma ação diferente do que acontece com números, por exemplo, com números inteiros ele soma os dois números, já com strings ele concatena os textos, como podemos ver a seguir:

```

1 string a = 'Olá, ';
2 string b = 'amigo.';
3 string c = a+b;

```

***A saída seria 'Olá, amigo.'.**

Quando queremos documentar o significado de algum código dentro de um programa C#, podemos utilizar comentários. Para fazermos um comentário de uma linha, utilizamos o //. Tudo que estiver depois do // é considerado comentário e, por isso, ignorado pelo compilador da linguagem.

E se precisarmos de comentar mais de uma linha de código? Bom, para isso usamos o comentário de múltiplas linhas que é inicializado por um /* e terminado pelo */. Tudo que estiver entre a abertura e o fechamento do comentário é ignorado pelo compilador da linguagem:

```

1 /*
2 Comentário de
3 múltiplas linhas.
4 */

```

2.3 Métodos

Um método é um bloco de código que contém uma série de instruções. No C# todos os códigos são executados no contexto de um método, o método Main é o ponto de entrada de todos os aplicativos.

Criação de um método

Métodos são declarados em uma classe.

Quando declaramos um método começamos com seu nível de acesso:

- **Public:** acessível em qualquer lugar.
- **Private:** acessível somente na classe que foi declarada o método.
- **Protected:** acessível na própria classe ou classe derivada.

Após o nível de acesso declaramos o tipo de retorno do método. Quando ele não retorna nada usamos a palavra reservada **void**. Depois o nome do método com a primeira letra maiúscula, por padrão, e os parâmetros que o método recebe entre parênteses. E por fim instruções de código entre chaves(é o que o método irá fazer).

Exemplos:

Sem retorno:

```
1 public void Saudacao(){
2     Console.WriteLine("Hello, World!");
3 }
```

Com retorno e parâmetros:

```
1 public int Soma(int numero1, int numero2){
2     int soma= numero1+numero2
3     return soma;
4 }
```

2.3.1 Sobrecarga

Sobrecarga refere-se aos métodos de uma classe, sendo que os mesmos podem ser sobrecarregados em relação aos seus nomes, podendo diversos métodos possuir o mesmo nome, porém os tipos de dados da lista de parâmetros devem ser divergentes.

Exemplo:

```
1 public void Saudacao(){
2     Console.WriteLine("Hello, World!");
3 }
4 public void Saudacao(string saudacao){
5     Console.WriteLine(saudacao);
6 }
```


3 - Programação Orientada à Objetos – (POO)

A Programação Orientada à Objetos é atualmente um dos paradigmas mais utilizados por empresas e desenvolvedores no mundo. Esse modelo utilizado para o desenvolvimento de software revolucionou e facilitou a vida dos programadores em 101%.

A orientação à objetos é um paradigma que possui 4 pilares fundamentais, sendo eles:

- **Abstração**
- **Encapsulamento**
- **Herança**
- **Polimorfismo**

3.1 - Abstração

A **abstração** na orientação à objetos é onde tudo começa, é o ponto chave para que a abstração do mundo real seja transformada em um sistema. No caso da orientação à objeto, as abstrações são adotadas no formato de “classes”.

Uma **classe** é uma abstração de um objeto do mundo real para o software, uma classe possui atributos e métodos. O seguinte exemplo abaixo irá mostrar como funciona o processo de abstração:

Vamos considerar o seguinte problema:

1. Preciso criar uma classe que represente a abstração de um funcionário de uma empresa.
2. Preciso definir quais são os atributos e métodos dessa classe que representará o meu funcionário.

Um **funcionário** possui as seguintes características, que na orientação à objetos conhecemos como ‘atributos’: nome, idade, profissão, rg, cpf e entre outros.

Um **funcionário** pode executar as seguintes funções: marcar ponto, trabalhar, descansar e etc.

Essa é abstração de um **funcionário** passada para a orientação à objetos.

Funcionário
- Nome : string - Idade : int - Profissão : string - RG : string - CPF : string
+ marcarPonto() : void + trabalhar() : void + descansar() : void

3.2 - Encapsulamento

O *encapsulamento* é uma das principais técnicas que define a programação orientada a objetos. Se trata de um dos elementos que adicionam segurança à aplicação em uma programação orientada a objetos pelo fato de esconder as propriedades, criando uma espécie de caixa preta.

A maior parte das linguagens orientadas a objetos implementam o encapsulamento baseado em propriedades privadas, ligadas a métodos especiais chamados *getters* e *setters*, que irão pegar e alterar os valores de uma propriedade, respectivamente. Essa atitude evita o acesso direto a propriedade do objeto, adicionando uma outra camada de segurança à aplicação.

Para fazermos um paralelo com o que vemos no mundo real, temos o encapsulamento em outros elementos. Por exemplo, quando clicamos no botão ligar da televisão, não sabemos o que está acontecendo internamente. Podemos então dizer que os métodos que ligam a televisão estão encapsulados.

3.3 - Exercícios

1- Crie uma classe com o nome de “Funcionário” e implemente os atributos e métodos citados no diagrama acima.

2 – Após a criação da classe, no programa principal faça a instanciação de objetos da classe “Funcionário” e altere os seus atributos e faça a chamada dos seus métodos.

3.4 - Modificadores de Acesso (Visibilidade)

Uma parte também muito importante da programação orientada à objetos são os modificadores de acesso, ou também conhecidos como: “visibilidade de métodos e atributos”. Os modificadores de acesso têm a responsabilidade de dar “permissões” às respectivas classes que poderão acessar métodos e atributos de uma outra.

Os modificadores de acesso mais usados e as suas funcionalidades são os seguintes:

Public - Os métodos e atributos são acessíveis em qualquer classe.

Private - Os métodos e atributos são acessíveis somente dentro da própria classe.

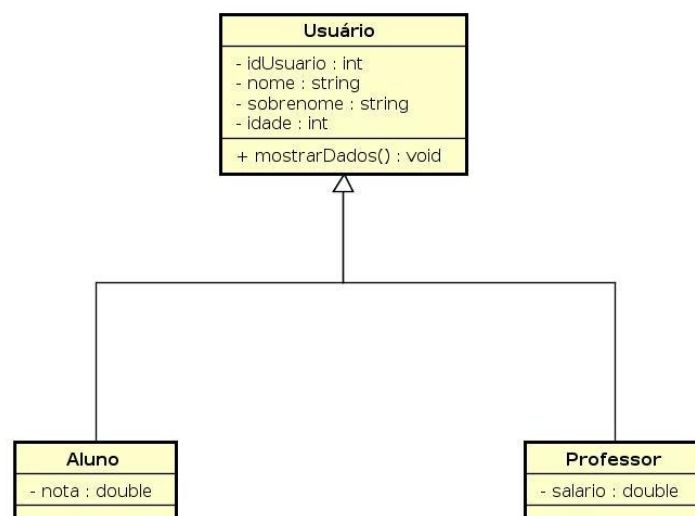
Protected - Os métodos e atributos são acessíveis somente dentro da própria classe ou por uma subclasse. Ou seja, por uma classe que herdou da mesma.

3.5 - Herança

O reuso de código é uma das grandes vantagens da programação orientada a objetos. Muito disso se dá por uma questão que é conhecida como *herança*. Essa característica otimiza a produção da aplicação em tempo e linhas de código.

Para entendermos essa característica, vamos imaginar uma família: a criança, por exemplo, está herdando características de seus pais. Os pais, por sua vez, herdam algo dos avós, o que faz com que

a criança também o faça, e assim sucessivamente. O objeto abaixo na hierarquia irá herdar características de todos os objetos acima dele, seus “ancestrais”. A herança a partir das características do objeto mais acima é considerada herança direta, enquanto as demais são consideradas heranças indiretas. Por exemplo, na família, a criança herda diretamente do pai e indiretamente do avô e do bisavô.



A imagem acima mostra um ciclo de **herança** representado por um diagrama de classes, onde três classes são definidas. Sendo elas: **Usuário**, **Aluno** e **Professor**. Onde a classe “**Usuário**” é a classe pai e “**Aluno**” e “**Professor**” são as classes filhas, também conhecidas como “*subclasses*”. Estas classes herdam métodos e atributos de sua classe pai, e também pode ter métodos e atributos próprios definidos dentro dela mesma.

3.6 - Exercícios

1- Crie uma classe abstrata com o nome de “**Eletrônico**”, uma classe com o nome de “**Rádio**”, uma classe com o nome de “**Televisão**” e implemente os atributos e métodos base na classe “**Eletrônico**”. Após, herde os mesmos nas classes “**Rádio**” e “**Televisão**” e crie atributos e métodos específicos para cada uma destas. Finalizada a construção dessas classes, no programa principal, instancie objetos dessas classes e faça a alteração dos atributos e a chamada dos métodos.

3.7 - Polimorfismo

Outro ponto essencial na programação orientada a objetos é o chamado polimorfismo. Na natureza, vemos animais que são capazes de alterar sua forma conforme a necessidade, e é dessa ideia que vem o polimorfismo na orientação a objetos. Como sabemos, os objetos filhos herdam as características e ações de seus “ancestrais”. Entretanto, em alguns casos, é necessário que as ações para um mesmo método seja diferente. Em outras palavras, o *polimorfismo* consiste na alteração do funcionamento interno de um método herdado de um objeto pai.

Exemplo:

Suponha que temos uma classe chamada “Aparelho Eletrônico”, nós sabemos existem vários tipos de aparelhos eletrônicos, certo? Um aparelho eletrônico pode ser um rádio, televisão, celular, câmera fotográfica e entre outros.

Agora, voltando um pouco atrás onde falamos que uma classe possui métodos e atributos, vamos considerar que a classe “Aparelho Eletrônico” seja a nossa classe genérica, ou seja: um aparelho eletrônico pode assumir diversas formas, das quais foram mencionadas anteriormente.

Agora, vamos definir que a nossa classe “Aparelho Eletrônico” possui um método chamado “ligar”. Como foi analisado antes, um aparelho eletrônico pode assumir diversas formas diferentes, e todos eles necessitam ser **ligados** para funcionar, certo? Mas nem todos eles são ligados da mesma forma. Exemplo: a forma de ligar um rádio pode ser a de ligar uma câmera fotográfica e entre outros, mas todos eles executam o método ligar.

Graças a esse tipo de abordagem, podemos reutilizar o método **ligar** da nossa classe genérica “Aparelho Eletrônico” em classes que assumem a forma de um aparelho eletrônico.

3.8 - Exercícios

1- Na classe “Eletrônico” implementada anteriormente, crie um método abstrato chamado “EmitirSom” e o sobrescreva nas classes “Rádio” e “Televisão”.

4 - Modelagem de Dados

4.1 - Preparação do Ambiente

INSTALAÇÃO SQL SERVER EXPRESS 2017

Execute o arquivo “... \1 - SQLSERVEREXPRESS2017\Express_PTB\SETUP” para instalar o SQL Server Express 2017;

Clique em “Nova instalação autônoma do SQL Server...”;

Clique em Aceitar e Avançar;

Desmarque “Serviços de Machine Learning” e clique em Avançar;

Clique em Avançar para mantermos a instância do SQL Server como SQLEXPRESS;

Mude o tipo de Inicialização do “SQL Server Browser” para Automática e clique em Avançar;

Selecione Modo Misto, digite uma Senha que irá usar para acessar o banco de dados e clique em Avançar;

Após a instalação Clique em Fechar;

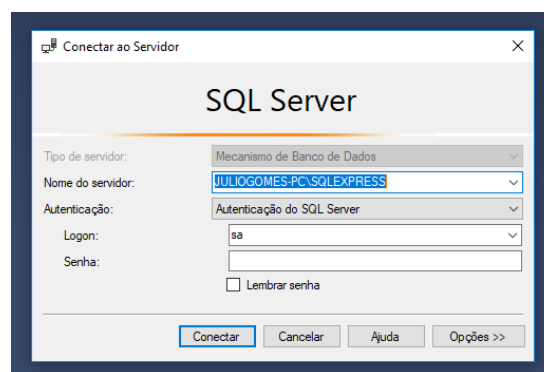
INSTALAÇÃO MANAGEMENT STUDIO 2017

Execute o arquivo “2 - MANAGEMENTSTUDIO2017” para instalar o Management Studio;

Clique em Instalar, aguarde a instalação e clique em Fechar;

Para iniciarmos a conexão com o servidor de Banco de Dados, abra o Management Studio, selecione a opção Autenticação do SQL Server, no Logon

Coloque “sa”, na Senha digite a que foi definida na instalação do SQL Server e Clique em Conectar.



4.2 - Microsoft SQL Server

O QUE É?

O Microsoft SQL Server é um sistema gerenciador de Banco de dados relacional (SGBD), criado parceria com a SYBASE em 1988. Parceria que terminou em 1994, onde a Microsoft deu continuidade as melhorias e manutenções do produto. Atualmente é um dos SGBDs mais utilizados no mundo, com versões gratuitas e pagas, tendo como concorrentes sistemas como Oracle e MySQL.

4.3 - BANCO DE DADOS RELACIONAL

Um banco de dados relacional é uma coleção de dados com relacionamentos predefinidos entre si. Esses itens são organizados como um conjunto de tabelas com colunas e linhas. As tabelas são usadas para reter informações sobre os objetos a serem representados no banco de dados. Cada coluna da tabela retém um determinado tipo de dado e um campo armazena o valor em si de um atributo. As linhas na tabela representam uma coleção de valores relacionados de um objeto ou uma entidade. Cada linha em uma tabela pode ser marcada com um único identificador chamado de chave principal. Já as linhas entre as várias tabelas podem ser associadas usando chaves estrangeiras. Esses dados podem ser acessados de várias formas diferentes.

4.4 - SQL - STRUCTURED QUERY LANGUAGE

SQL, ou Structured Query Language, é a interface primária usada para comunicação com bancos de dados relacionais. O SQL tornou-se padrão do American National Standards Institute (ANSI) em 1986. O SQL padrão ANSI é aceito por todos os mecanismos conhecidos de banco de dados relacional, e alguns desses mecanismos também têm a extensão para SQL ANSI para apoiar a funcionalidade específica desse mecanismo. O SQL é usado para adicionar, atualizar ou excluir linhas de dados, recuperar subconjuntos de dados para processamento de transações e aplicações de análise, além de gerenciar todos os aspectos do banco de dados.

4.5 - INTEGRIDADE DE DADOS

Integridade dos dados é a completude, precisão e consistência gerais dos dados. Bancos de dados relacionais usam uma série de constraints para obrigar integridade de dados no banco de dados. Aí estão incluídas chaves primárias, chaves estrangeiras, constraint 'Not NULL', constraint 'Unique', constraint 'Default' e constraints 'Check'. Essas constraints de integridade ajudam na obrigatoriedade das regras dos negócios quanto aos dados nas tabelas, de forma a garantir a precisão e a confiabilidade dos dados. Além desses, a maioria dos bancos de dados relacionais também permite que um código personalizado seja incorporado em triggers executados com base em uma ação no banco de dados.

4.6 - TRANSAÇÕES

Uma transação de banco de dados consiste em uma ou mais instruções SQL executadas como uma sequência de operações que formam uma só unidade lógica do trabalho. As transações disponibilizam uma proposição "tudo ou nada", o que significa que a transação inteira deve estar completa como uma só unidade e ser gravada no banco de dados, caso contrário, nenhum componente individual da transação deverá passar. Em relação à terminologia do banco de dados, uma transação resulta em COMMIT ou ROLLBACK. Cada transação é tratada de forma coerente e confiável, independente de outras transações.

4.7 - Abordagem do Modelo ER (entidade e relacionamento)

Para a modelagem do banco de dados que iremos usar no curso, a primeira coisa que vamos fazer é a leitura e interpretação de um modelo conceitual do que virá a ser o banco de dados que vamos modelar. Nesse modelo conceitual chamado de "Modelo entidade-relacionamento", veremos uma notação gráfica onde teremos algumas regras e conceitos de como e vamos montar nossas tabelas e relacionamentos no banco de dados. Para isso devemos conhecer alguns conceitos.

4.8 - ENTIDADE

Uma entidade representa, no modelo conceitual, um conjunto de objetos (tabelas) os quais deseja-se guardar informações. Alguns exemplos de entidades poderiam ser produtos, pessoas, vendas, entre outros. No sistema que vamos desenvolver teremos as seguintes entidades, Cliente, Endereço, Venda e Produto. A representação gráfica de uma entidade é um retângulo como na figura abaixo.

4.9 –



RELACIONAMENTO

Os relacionamentos representam associações do mundo real entre uma ou mais entidades. Eles são



representados por um losango cujo ação de ligação é escrita

em seu interior. Exemplificando, o relacionamento comum entre médico e paciente seria Consulta. Veja a representação gráfica deste relacionamento citado, na figura abaixo.

Relacionamento um-para-um (1,1) – Cada ocorrência de uma entidade relaciona-se com uma e somente uma ocorrência da outra entidade.

Relacionamento um-para-muitos (1,n) – Uma ocorrência da entidade pai relaciona-se com muitas ocorrências da entidade filho, mas cada ocorrência da entidade filho somente pode estar relacionada com uma ocorrência da tabela pai.

Relacionamento muitos-para-muitos (n,n) – Apresenta em ambos os sentidos um ou mais relacionamento de um-para-muitos. No modelo relacional, que veremos a seguir, não é possível efetuar este tipo de relacionamento de forma direta. Neste caso, deve-se construir uma terceira tabela (tabela de detalhes). Essa tabela seria uma entidade que se relaciona com as duas entidades em um relacionamento de um-para-muitos.

4.10 - CARDINALIDADE

Em modelagem de dados a cardinalidade é um dos princípios fundamentais sobre relacionamento de um banco de dados relacional. Nela são definidos os graus de relação entre duas entidades

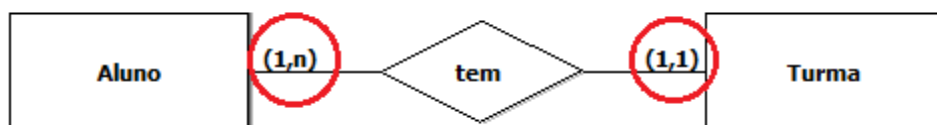
ou tabelas. Ou seja, ela define o número mínimo e máximo de ocorrências em um relacionamento.

Cardinalidade Mínima – Considera-se duas cardinalidades mínimas:

Opcional: Indica que o relacionamento é opcional. Representa-se pelo número 0.

Obrigatória: Indica que o relacionamento é obrigatório. Representa-se pelo número 1.

Cardinalidade Máxima – É o número máximo de ocorrências de entidade que são associadas a uma ocorrência da mesma ou de outra entidade através de um relacionamento, Apenas duas cardinalidades máximas são relevantes: a cardinalidade máxima **1** e a cardinalidade máxima **n** (muitos).

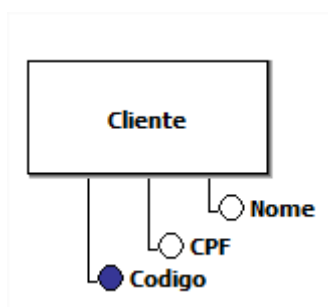


Na representação gráfica anterior, indica que um Aluno deve estar no mínimo em uma Turma e no máximo em uma Turma. Com isso, a cardinalidade de Turma em relação ao Aluno é de “um-para-um” (1,1). Do outro lado a cardinalidade de Aluno em relação a Turma é de “um-para-muitos” (1,n). Isso indica que uma Turma deve ter no mínimo um Aluno e no máximo muitos.

4.11 - ATRIBUTO

Atributos são características de entidades que oferecem detalhes descritivos sobre elas, é um dado que é associado a cada ocorrência de uma entidade ou de um relacionamento.

Atributos são representados graficamente conforme mostra a figura abaixo. A figura expressa que a cada ocorrência de Cliente é associado exatamente um CPF, um código e um nome.

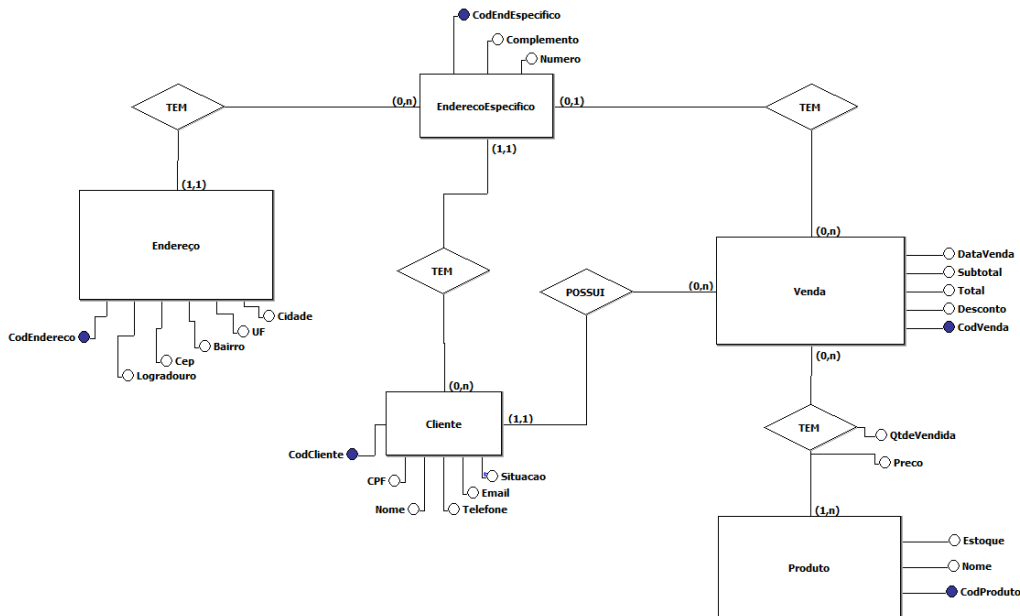


Na prática, atributos não são representados graficamente, para não sobrecarregar os diagramas, já que muitas vezes entidades possuem um grande número de atributos.

4.12 - MODELO CONCEITUAL ER

O banco de dados que vamos modelar, deverá atender a demanda de uma loja que tem um cadastro de clientes com seus respectivos endereços residenciais e faz a entrega dos produtos vendidos no endereço do cliente ou em outro endereço desejado. Para isso vamos analisar o diagrama e aproveitar para entender os conceitos apresentados anteriormente. Mais à frente iremos transformar

esse diagrama, da imagem seguinte, em um modelo lógico que represente as tabelas e estrutura do banco de dados que vamos modelar.



4.13 - Composição de um Banco de Dados Relacional

Apresentaremos agora um conjunto mínimo de conceitos sobre bancos de dados relacionais, com o objetivo de permitir o entendimento do Banco de Dados que iremos criar.

4.14 - TABELA

Um banco de dados relacional é composto por tabelas. Uma tabela é um conjunto não ordenado de linhas (tuplas). Cada linha é composta por uma série de campos (valor de atributo).

Cada campo é identificado por um nome de campo (nome de atributo). O conjunto de campos das linhas de uma tabela que possuem o mesmo nome formam uma coluna. Esses nomes de campos serão utilizados como referência para que possamos manipular os dados das tabelas.

Emp			
CódigoEmp	Nome	CodigoDepto	CategFuncional
E5	Souza	D1	C5
E3	Santos	D2	C5
E2	Silva	D1	C2
E1	Soares	D1	—

coluna (atributo)

nome do campo (nome do atributo)

linha (tupla)

valor de campo (valor de atributo)

4.15 - CHAVES

O conceito básico para estabelecer relações entre linhas de tabelas de um banco de dados relacional é o da *chave*. Em um banco de dados relacional, há vários tipos de chaves, vamos considerar apenas dois tipos, que serão usados no nosso banco de dados: *chave primária (PK)* e *chave estrangeira (FK)*.

Chave Primária (PK) – é uma coluna ou combinação de colunas cujos valores distinguem uma linha das demais dentro de uma tabela. A regra que deve ser obedecida com uma chave primária é a unicidade de valores nas colunas que compõem a chave. Torna-se uma chave composta a partir do momento que a formação da unicidade das colunas é dada a partir de uma combinação de duas ou mais colunas. A chave primária não pode ser composta por campo opcional, ou seja, campo que aceite nulo.

Chave Estrangeira (FK) – é uma coluna ou uma combinação de colunas, cujos valores aparecem necessariamente na chave primária de uma tabela. A chave estrangeira é o mecanismo que permite a implementação de relacionamentos em um banco de dados relacional.

4.16 - DOMÍNIOS E VALORES VAZIOS

Quando uma tabela do banco de dados é definida, para cada coluna da tabela deve ser especificado um conjunto de valores (alfanumérico, numérico, ...) que os campos da respectiva coluna podem assumir. Este conjunto de valores é chamado de domínio da coluna.

Além disso, deve ser especificado se os campos da coluna podem estar vazios (“null”) ou não. Estar vazio indica que o campo não recebeu nenhum valor de seu domínio. As colunas nas quais não são admitidos valores vazios são chamadas de colunas obrigatórias. As colunas nas quais podem aparecer campos vazios são chamadas de colunas opcionais.

4.17 - RESTRIÇÕES DE INTEGRIDADE

Um dos objetivos primordiais de um SGBD é a integridade de dados. Dizer que os dados de um banco de dados estão íntegros significa dizer que eles refletem corretamente a realidade representada pelo banco de dados e que são consistentes entre si. Para tentar garantir a integridade de um banco de dados existem as restrições de integridade. Uma restrição de integridade é uma regra de consistência de dados que é garantida pelo próprio SGBD. Costuma-se classificar as restrições nas seguintes categorias:

Integridade de domínio – Restrições deste tipo especificam que o valor de um campo deve obedecer a definição de valores admitidos para a coluna (o domínio da coluna). Nos SGBD relacionais comerciais, é possível usar apenas domínios pré-definidos (número inteiro, número real, alfanumérico de tamanho definido, data, ...). O usuário do SGBD não pode definir domínios próprios de sua aplicação (por exemplo, o domínio dos dias da semana ou das unidades da federação).

Integridade de vazio – Através deste tipo de restrição de integridade é especificado se os campos de uma coluna podem ou não ser vazios (se a coluna é obrigatória ou opcional). Como já foi mencionado, campos que compõem as chaves primárias sempre devem ser diferentes de vazio.

Integridade de chave – Trata-se da restrição que define que os valores da chave primária e alternativa devem ser únicos.

Integridade referencial – É a restrição que define que os valores dos campos que aparecem em uma chave estrangeira devem aparecer na chave primária da tabela referenciada.

As restrições dos tipos acima especificados devem ser garantidas automaticamente por um SGBD relacional, isto é, não deve ser exigido que o programador escreva procedimentos para garanti-las explicitamente. Há muitas outras restrições de integridade que não se encaixam em nenhuma das categorias acima e que normalmente não são garantidas pelo SGBD. Essas restrições são chamadas de restrições semânticas. Alguns exemplos de restrições deste tipo poderiam ser:

☐ Um empregado do departamento denominado “Finanças” não pode ter a categoria funcional “Engenheiro”.

☐ Um empregado não pode ter um salário maior que seu superior imediato.

4.18 - TIPOS DE DADOS

Os tipos de dados são classificados em diferentes categorias e permitem N formatos. Abaixo uma descrição de algumas categorias e de alguns tipos de dados do SQL Server:

Baseados em Caracteres:

Char (n) – Trata-se de um datatype que aceita como valor qualquer dígito, sendo que o espaço ocupado no disco é de um dígito por caractere. É possível utilizar até 8 mil dígitos.

Varchar (n) – Também aceita como valor qualquer dígito e o espaço ocupado em disco é de um dígito por caractere. Permite usar também no máximo 8 mil dígitos. A diferença para o Char, é que o Varchar geralmente é usado quando não sei o tamanho fixo de um campo.

Baseados em Caracteres Unicode:

Nchar (n) – Neste datatype, pode usar qualquer dígito, sendo ocupados 2 bytes a cada caractere. É possível usar até 8 mil bytes.

Nvarchar(n) – Igual ao tipo anterior, com a única diferença que uso esse tipo quando não sei o tamanho fixo de um campo. 2 bytes são ocupados a cada caractere. É possível usar até 8 mil bytes.

Baseados em Numéricos Inteiros:

Bigint – Aceita valores entre -2^{63} e $2^{63}-1$, sendo que esse datatype ocupa 8 bytes.

Int – Os valores aceitos aqui variam entre -2^{31} a $2^{31}-1$. Ocupa 4 bytes.

Smallint – Aceita valores entre -32768 até 32767 e ocupa 2 bytes.

Tinyint – Os valores aceitos aqui variam entre 0 e 255, ocupa apenas 1 byte.

Bit – É um tipo de dado inteiro (conhecido também como booleano), cujo valor pode corresponder a NULL, 0 ou 1. Podemos converter valores de string TRUE e FALSE em valores de bit, sendo que TRUE corresponde a 1 e FALSE a 0.

Baseados em Numéricos Exatos:

Decimal (P, S) – Os valores aceitos variam entre $-10^{38}-1$ e $10^{38}-1$, sendo que o espaço ocupado varia de acordo com a precisão. Se a precisão for de 1 a 9, o espaço ocupado é de 5 bytes. Se a precisão

é de 10 a 19, o espaço ocupado é de 9 bytes, já se a precisão for de 20 a 28, o espaço ocupado é de 13 bytes, e se a precisão for de 29 a 38, o espaço ocupado é de 17 bytes.

Numeric (P, S) – Considerado um sinônimo do datatype decimal, o numérico também permite valores entre $-10^{38}-1$ e $10^{38}-1$ e o espaço ocupado é o mesmo do anterior.

Baseados em Valores Numéricos Monetários:

Money – Este datatype aceita valores entre -2^{63} e $2^{63}-1$, sendo que 8 bytes são ocupados.

Baseados em Data e Hora:

Datetime – Permite o uso de valores entre 1/1/1753 e 31/12/9999. Este datatype ocupa 8 bytes e sua precisão atinge 3.33 milissegundos.

4.19 - Normalização de dados

Este processo baseia-se no conceito de forma normal. Uma forma normal é uma regra que deve ser obedecida por uma tabela para que esta seja considerada “bem projetada”. Há diversas formas normais, isto é, diversas regras, cada vez mais rígidas, para verificar tabelas relacionais. No caso deste trabalho, vamos considerar três formas normais. As formas normais são denominadas simplesmente primeira, segunda e terceira forma normal, abreviadamente 1FN, 2FN e 3FN.

1FN - 1ª Forma Normal

Todos os atributos de uma tabela devem ser atômicos, ou seja, a tabela não deve conter grupos repetidos e nem atributos com mais de um valor. Para deixar nesta forma normal, é preciso identificar a chave primária da tabela, identificar a(s) coluna(s) que tem dados repetidos e removê-la(s), criar uma nova tabela com a chave primária para armazenar o dado repetido e, por fim, criar uma relação entre a tabela principal e a tabela secundária. Por exemplo, considere a tabela Pessoas a seguir.

PESSOAS = {ID+ NOME + ENDERECO + TELEFONES}

Ela contém a chave primária ID e o atributo TELEFONES é um atributo multivalorado e, portanto, a tabela não está na 1FN. Para deixá-la na 1FN, vamos criar uma nova tabela chamada TELEFONES que conterá PESSOA_ID como chave estrangeira de PESSOAS e TELEFONE como o valor multivalorado que será armazenado.

PESSOAS = { ID + NOME + ENDERECO }

TELEFONES = { PESSOA_ID + TELEFONE }

2FN - 2ª Forma Normal

Antes de mais nada, para estar na 2FN é preciso estar na 1FN. Além disso, todos os atributos não chaves da tabela devem depender unicamente da chave primária (não podendo depender apenas de parte dela). Para deixar na segunda forma normal, é preciso identificar as colunas que não são funcionalmente dependentes da chave primária da tabela e, em seguida, remover essa coluna da tabela principal e criar uma nova tabela com esses dados. Por exemplo, considere a tabela ALUNOS_CURSOS a seguir.

ALUNOS_CURSOS = { ID_ALUNO + ID_CURSO + NOTA + DESCRICAO_CURSO }

Nessa tabela, o atributo DESCRICAO_CURSO depende apenas da chave primária ID_CURSO. Dessa forma, a tabela não está na 2FN. Para tanto, cria-se uma nova tabela chamada CURSOS que tem como chave primária ID_CURSO e atributo DESCRICAO retirando, assim, o atributo DESCRICAO_CURSO da tabela ALUNOS_CURSOS.

ALUNOS_CURSOS = {ID_ALUNO + ID_CURSO + NOTA}

CURSOS = {ID_CURSO + DESCRICAO}

3FN - 3ª Forma Normal

Para estar na 3FN, é preciso estar na 2FN. Além disso, os atributos não chave de umas tabelas devem ser mutuamente independentes e dependentes unicamente e exclusivamente da chave primária (um atributo B é funcionalmente dependente de A se, e somente se, para cada valor de A só existe um valor de B). Para atingir essa forma normal, é preciso identificar as colunas que são funcionalmente dependentes das outras colunas não chave e extraí-las para outra tabela. Considere, como exemplo, a tabela FUNCIONARIOS a seguir.

FUNCIONARIOS = { ID + NOME + ID_CARGO + DESCRICAO_CARGO }

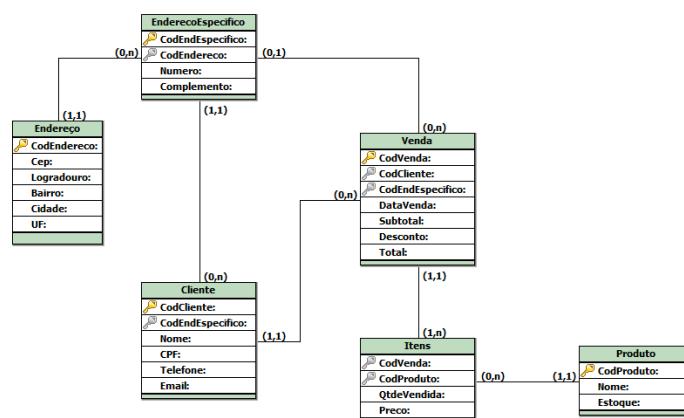
O atributo DESCRICAO_CARGO depende exclusivamente de ID_CARGO (atributo não chave) e, portanto, deve-se criar uma nova tabela com esses atributos. Dessa forma, ficamos com as seguintes tabelas:

FUNCIONARIOS = { ID + NOME + ID_CARGO }

CARGOS = { ID_CARGO + DESCRICAO }

4.20 - Criação de um Banco de Dados Relacional com base a um modelo ER

Com base no modelo ER demonstrado anteriormente e nos conceitos de banco de dados relacional descritos nessa apostila, serão abordados nos slides e explicações o processo de transformação de um modelo conceitual para uma abordagem relacional lógica, que será a representação gráfica do banco de dados que criaremos e que será utilizado nas próximas aulas do curso.



4.21 - Linguagem de definição de dados (DDL)

Permite ao usuário a definição da estrutura e organização dos dados armazenados e das relações existentes entre eles. A seguir teremos alguns comandos que irá auxiliar na criação das tabelas do nosso banco de dados.

CREATE DATABASE – Instrução usada para criar um novo banco de dados SQL.

```
CREATE DATABASE dbname;
```

DROP DATABASE – Instrução usada para descartar um banco de dados SQL.

```
DROP DATABASE dbname;
```

CREATE TABLE – Instrução usada para criar uma nova tabela em um banco de dados.

```
CREATE TABLE table_name (  
    column1 datatype,  
    column2 datatype,  
    column3 datatype,  
    ....  
);
```

DROP TABLE – Instrução usada para descartar uma tabela existente.

```
DROP TABLE table_name;
```

ALTER TABLE ADD Coluna – Instrução usada para adicionar uma coluna na tabela.

```
ALTER TABLE table_name  
ADD column_name datatype;
```

ALTER TABLE DROP COLUMN – Instrução usada para excluir uma coluna na tabela.

```
ALTER TABLE table_name  
DROP COLUMN column_name;
```

ALTER TABLE ALTER COLUMN – Instrução usada para alterar o nome e o tipo de dados de uma coluna na tabela.

```
ALTER TABLE table_name  
ALTER COLUMN column_name datatype;
```

RESTRIÇÕES SQL – Algumas restrições comumente usadas em SQL:

NOT NULL - Garante que uma coluna não pode ter um valor NULL

```
CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255) NOT NULL,  
    Age int  
);
```

UNIQUE - Garante que todos os valores em uma coluna sejam diferentes

```
CREATE TABLE Persons (  
    ID int NOT NULL UNIQUE,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int  
);  
  
ALTER TABLE Persons  
DROP CONSTRAINT UC_Person;  
  
ALTER TABLE Persons  
ADD UNIQUE (ID);  
  
ALTER TABLE Persons  
ADD CONSTRAINT UC_Person UNIQUE (ID,LastName);
```

PRIMARY KEY - Uma combinação de um NOT NULL e UNIQUE. Identifica exclusivamente cada linha em uma tabela

```
CREATE TABLE Persons (  
    ID int NOT NULL PRIMARY KEY,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int  
);  
  
ALTER TABLE Persons  
DROP CONSTRAINT PK_Person;  
  
ALTER TABLE Persons  
ADD PRIMARY KEY (ID);  
  
ALTER TABLE Persons  
ADD CONSTRAINT PK_Person PRIMARY KEY (ID,LastName);
```

FOREIGN KEY - Identifica exclusivamente uma linha / registro em outra tabela

```
CREATE TABLE Orders (  
    OrderID int NOT NULL PRIMARY KEY,  
    OrderNumber int NOT NULL,  
    PersonID int FOREIGN KEY REFERENCES Persons(PersonID)  
);  
  
ALTER TABLE Orders  
ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);  
  
ALTER TABLE Orders  
DROP CONSTRAINT FK_PersonOrder;
```

DEFAULT - Define um valor padrão para uma coluna quando nenhum valor é especificado

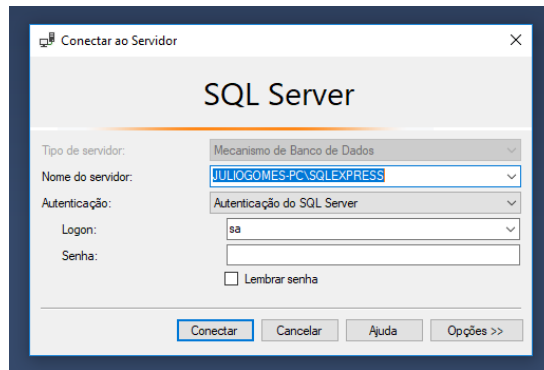
```
CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    City varchar(255) DEFAULT 'Sandnes'  
);  
  
ALTER TABLE Persons  
ALTER COLUMN City SET DEFAULT 'Sandnes';  
  
ALTER TABLE Persons  
ALTER COLUMN City DROP DEFAULT;
```

AUTO INCREMENT Field - Auto incremento permite que um número exclusivo seja gerado automaticamente quando um novo registro é inserido em uma tabela.

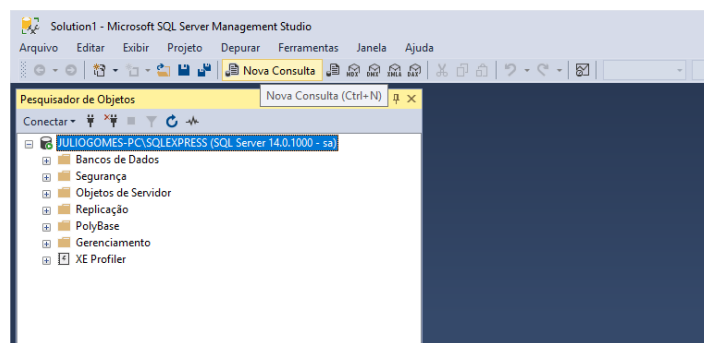
Muitas vezes, este é o campo de chave primária que gostaríamos de ser criado automaticamente sempre que um novo registro foi inserido.

```
CREATE TABLE Persons (  
    ID int IDENTITY(1,1) PRIMARY KEY,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int  
);
```

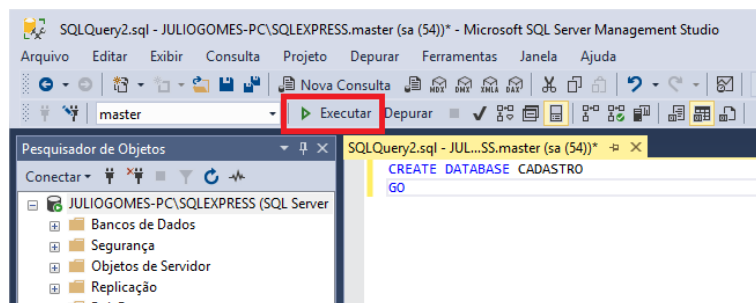
Para nos auxiliar conecte no Management Studio, como foi explicado no início dessa apostila.



Vamos iniciar criando um banco de dados usando T-SQL, então, com o Management Studio aberto, clique no botão New Query, como mostra a imagem:



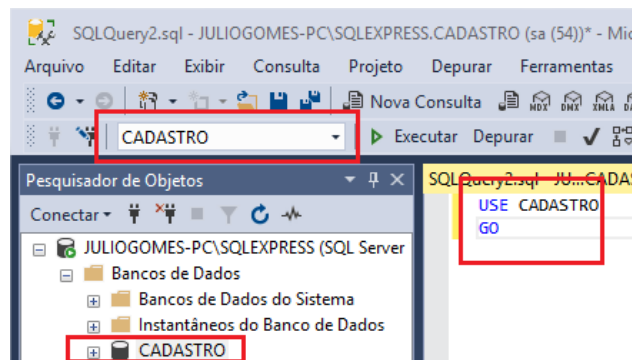
Em seguida, na parte central da interface aparecerá uma tela em branco, na qual você poderá digitar os códigos para criar o banco de dados, tabelas, efetuar consultas, etc. Nesse momento, vamos executar o código a seguir e pressionar F5 ou clicar em Executar, como mostra a imagem:



Com o banco criado, precisamos entrar no seu contexto para poder executar novos scripts dentro dele. No caso, precisamos acessá-lo para criar nossa tabela. Para isso, devemos executar o seguinte comando:

```
USE CADASTRO
GO
```

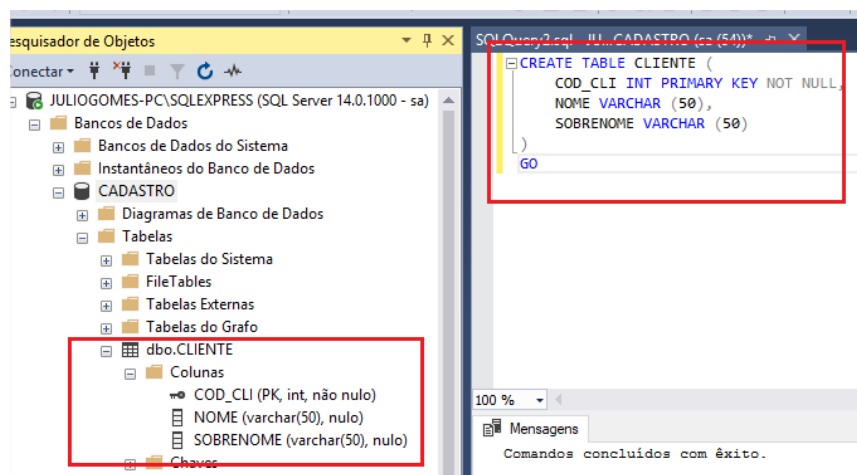
Para confirmar que a operação foi executada como esperado, veja na parte superior esquerda do Management Studio se o banco criado está em uso, como mostra a imagem:



O próximo passo será criar uma tabela, nesse caso contendo apenas três campos. Isso pode ser feito executando o script da imagem a seguir:

```
CREATE TABLE CLIENTE (  
    COD_CLI INT PRIMARY KEY NOT NULL,  
    NOME VARCHAR (50),  
    SOBRENOME VARCHAR (50)  
)  
GO
```

Em seguida, para verificar se a tabela foi criada, acesse a árvore de elementos do banco de dados no lado esquerdo, como vemos na próxima imagem:



5 - Comandos Básicos para Manipulação de Dados

5.1 – INSERT

O comando para inclusão de dados em uma determinada tabela do banco de dados é o INSERT, que possui a seguinte sintaxe:


```
INSERT INTO Nome_Tabela(Nome_Colunas)
VALUES(Valores)
```

Onde:

Nome_Tabela: Nome da tabela no banco de dados onde será inserido a informação.

Nome_Colunas: Colunas da tabela que receberão os valores, as especificações das colunas devem estar separadas por vírgula.

Valores: Valores que serão inseridos na tabela, esses valores serão inseridos na coluna corresponde a mesma ordem, os valores assim como as colunas devem estar separados por vírgula.

Vejamos abaixo um exemplo de utilização do comando INSERT

```
INSERT INTO Clientes(CPF, RG, Nome, Data_Nascimento, Data_Cadastro, Nome_Email)
VALUES(45104130843, 503708367, 'Bruno Alves', '1995-03-02', '2017-12-26', NULL)
```

Note que a quantidade de colunas e valores especificados é igual. Perceba também que podemos inserir valores NULL, sendo possível realizar esta atribuição apenas se a coluna da tabela aceite valores NULL.

5.2 - UPDATE

O comando para atualizar dados de uma determinada tabela do banco de dados é o UPDATE, que possui a seguinte sintaxe:

```
UPDATE Nome_Tabela
SET Campo = Valor
WHERE Condição
```

Onde:

Nome_Tabela: Nome da tabela no banco de dados que possui as informações que serão atualizadas.

Campo: Nome da coluna que será atualizada.

Valor: Novo valor que será setado na coluna especificada.

Where: Clausula que irá impedir que todos os registros da tabela sejam atualizados, apenas os registros que se encaixam na Condição imposta serão afetados.

No exemplo exposto abaixo a atualização está sendo feita na coluna Nacionalidade e o novo valor que será setado é brasileiro, entretanto esta atualização será feita apenas no cliente que possui o ID (Identificador) igual a 100.

```
UPDATE Clientes
SET Nacionalidade = 'Brasileiro'
WHERE Id_Cliente = 100
```

5.3 – DELETE

O comando para excluir dados de uma determinada tabela do banco de dados é o DELETE, que possui a seguinte sintaxe:

```
DELETE FROM Nome_Tabela  
WHERE Condição
```

Onde:

Nome_Tabela: Nome da tabela no banco de dados que possui as informações que serão excluídas.

Where: Clausula que irá impedir que todos os registros da tabela sejam excluídos, apenas os registros que se encaixam na Condição imposta serão afetados.

No exemplo exposto abaixo está sendo deletado da tabela Clientes todos os clientes que possuem o código do estado igual ao 10.

```
DELETE FROM Clientes  
WHERE Codigo_Estado = 10
```

5.4 – SELECT

O comando para selecionar dados de uma ou mais tabelas é o SELECT, que possui a seguinte sintaxe:

```
SELECT Nome_Colunas  
FROM Nome_Tabela  
WHERE Condição  
ORDER BY Nome_Coluna DESC
```

Onde:

Nome_Colunas: Colunas que serão exibidas após a execução do comando.

Nome_Tabela: Nome da tabela no banco de dados que possui as informações que serão selecionadas.

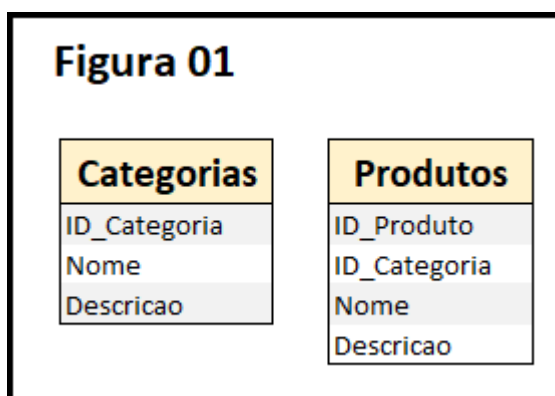
Where: Clausula que irá impedir que todos os registros da tabela sejam selecionados, apenas os registros que se encaixam na Condição imposta serão afetados.

Order By: Ordem em que os registros serão retornados pelo SELECT, a ordenação será feita pela coluna especificada na clausula, no exemplo acima utilizamos o comando adicional DESC (Descending) que indica que a ordenação será feita do maior valor para o menor valor contido na coluna especificada, podemos também utilizar o comando ASC (Ascending) que indica que a ordenação será feita do menor valor para o maior, caso não seja especificado o DESC ou ASC na clausula de ordenação será retornado do menor para o maior, o comando ORDER BY é de uso opcional.

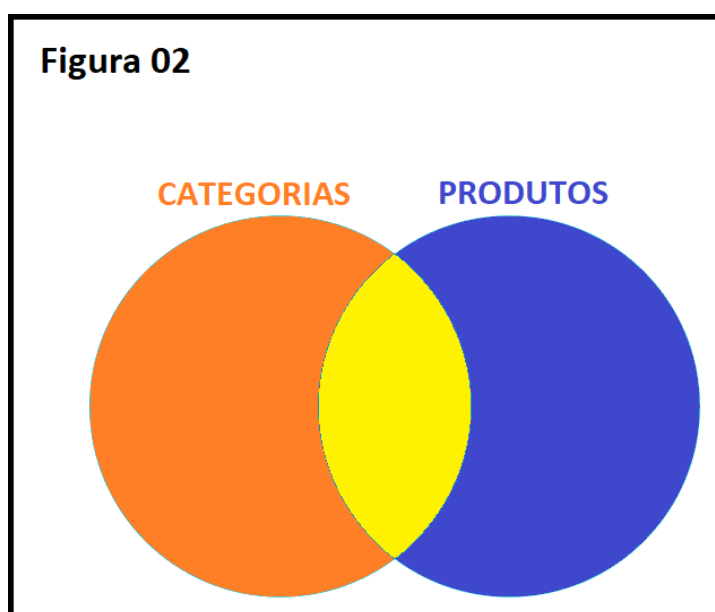
5.5 - Conhecendo os Tipos de JOIN

Join é uma ferramenta básica quando se trata de banco de dados relacional, é através dos tipos de join que conseguimos recuperar dados de diversas tabelas com apenas um único SELECT.

Para os exemplos a seguir vamos considerar duas tabelas, a tabela de Categorias e a tabela de Produtos, conforme ilustrado na **Figura 01**.



O relacionamento entre as duas tabelas pode ser conferido na **Figura 02**.



A área em laranja representa todas as categorias que foram cadastradas, porém ainda não foram vinculadas a nenhum tipo de produto.

A área em azul representa todos os produtos que foram cadastrados, porém que não possuem vínculo com nenhuma categoria.

A área em amarelo representa a intersecção das duas entidades, ou seja, representa todas as categorias que foram cadastradas e que estão vinculadas a um produto ou todos os produtos cadastrados e que estão vinculados a uma categoria.

A partir de agora vamos conhecer os quatro tipos de join mais utilizados, lembrando que os exemplos a seguir serão baseados nas tabelas de Categorias e Produtos.

5.6 - INNER JOIN

Para selecionar do banco de dados todos os produtos que possuem um tipo de categoria, ou seja, todos os produtos representados pela área em amarelo na **Figura 02**, vamos utilizar o comando INNER JOIN, conforme ilustrado na imagem abaixo.

```
SELECT p.ID_Produto,
       p.ID_Categoria,
       p.Nome,
       p.Descricao,
       c.Nome,
       c.Descricao
FROM Produtos p
     INNER JOIN Categorias c
       ON c.ID_Categoria = p.ID_Categoria
```

5.7 - LEFT OUTER JOIN

Para selecionar todos os produtos cadastrados que não foram vinculados a nenhum tipo de categoria, ou seja, todos os produtos representados pela área azul da **Figura 02**, vamos utilizar o comando LEFT OUTER JOIN, conforme ilustrado na imagem abaixo.

```
SELECT p.ID_Produto,
       p.Nome,
       p.Descricao
FROM Produtos p
     LEFT OUTER JOIN Categorias c
       ON c.ID_Categoria = p.ID_Categoria
WHERE c.ID_Categoria IS NULL
```

5.8 - RIGHT OUTER JOIN

Para selecionar todas as categorias representadas pela área em laranja da **Figura 02**, ou seja, todas as categorias que não possuem vínculo com nenhum produto vamos utilizar o comando RIGHT OUTER JOIN, conforme ilustrado na imagem abaixo.

```
SELECT c.ID_Categoria,
       c.Nome,
       c.Descricao
FROM Produtos p
     RIGHT OUTER JOIN Categorias c
       ON c.ID_Categoria = p.ID_Categoria
WHERE p.ID_Produto IS NULL
```

5.9 - FULL OUTER JOIN

Para seleccionar todos os registros de ambas as tabelas, independente dos vínculos entre as duas entidades, ou seja, todos os registros das áreas em laranja, amarelo e azul da **Figura 02**, vamos utilizar o comando FULL OUTER JOIN, conforme ilustrado na imagem abaixo.

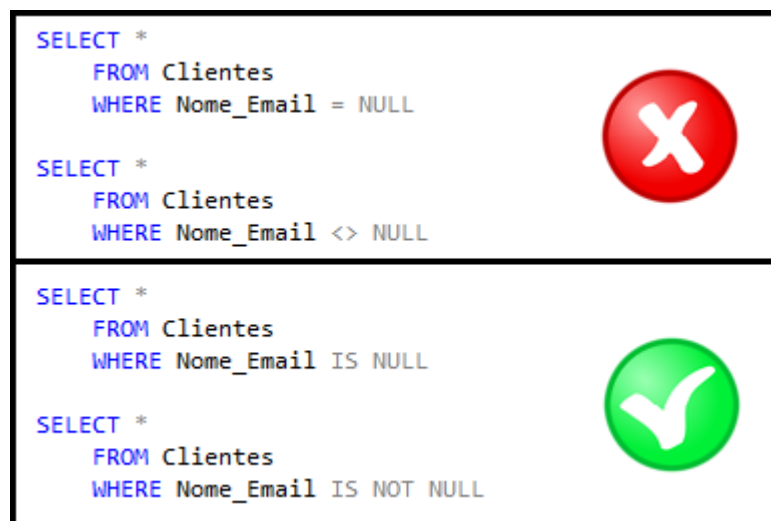
```
SELECT p.ID_Produto,
       p.Nome,
       p.Descricao,
       c.ID_Categoria,
       c.Nome,
       c.Descricao
FROM   Produtos p
       FULL OUTER JOIN Categorias c
       ON c.ID_Categoria = p.ID_Categoria
```

5.10 - Operadores de Comparação

Os operadores de comparação testam a condição imposta e com base no resultado retorna TRUE ou FALSE. Os operadores de comparação podem ser usados em todas as expressões exceto em expressões de dados text, ntext ou image. A tabela a seguir lista os operadores de comparação aceitos na linguagem T-SQL.

Operador	Significado
=	Igual a
>	Maior que
<	Menor que
>=	Maior que ou Igual a
<=	Menor que ou Igual a
<>	Diferente de

DICA: Ao realizar comparações com valores NULL, lembre-se que devemos utilizar os comandos IS NULL, para verificar se o campo está NULL, ou o comando IS NOT NULL, para verificar se o campo não está NULL, conforme ilustrado na imagem abaixo.



5.11 - Operadores Lógicos

Os operadores lógicos testam a legitimidade de algumas condições. Os operadores lógicos, como os operadores de comparação, retornam um tipo de dados Boolean com valor TRUE, FALSE ou UNKNOWN.

Abaixo podemos conferir os operadores lógicos aceitos pela linguagem T-SQL.

Operador	Significado
ALL	TRUE se tudo em um conjunto de comparações for TRUE.
AND	TRUE se as duas expressões booleanas forem TRUE.
ANY	TRUE se qualquer conjunto de comparações for TRUE.
BETWEEN	TRUE se o operando estiver dentro de um intervalo.
EXISTS	TRUE se uma subconsulta tiver qualquer linha.
IN	TRUE se o operando for igual a um de uma lista de expressões.
LIKE	TRUE se o operando corresponder a um padrão.
NOT	Inverte o valor de qualquer outro operador booliano.
OR	TRUE se qualquer expressão booliana for TRUE.
SOME	TRUE se algum conjunto de comparações for TRUE.

5.12 - Stored Procedures

Procedimentos Armazenados ou Stored Procedure é uma coleção de comandos em SQL (Structured Query Language) para otimização de banco de dados, elas também nos permitem o encapsulamento de tarefas repetitivas.

As Stored Procedures aceitam parâmetros de entrada para que o processamento seja de acordo com a necessidade, elas podem ou não retornar algum tipo de valor.

Abaixo podemos conferir a sintaxe de como criar uma Stored Procedure.

```
CREATE PROCEDURE Nome_Procedure
    @Parametros int

AS

BEGIN

    -- Coleção de comandos em SQL

END
```

Onde:

Nome_Procedure: Nome da procedure que será criada no banco de dados.

@Parametros: Lista dos parâmetros que serão aceitos pela procedure, note que o nome do parâmetro precisar ser iniciado com o caractere @, caso seja necessário adicionar outros parâmetros basta separa-los por virgula.

Perceba também que o código que irá compor a Stored Procedure precisa estar entre as declarações BEGIN e END.

Após criada a Stored Procedure podemos executa-la conforme ilustrado na imagem abaixo.

```
EXEC Nome_Procedure
```

Para informar os parâmetros na execução da procedure podemos fazê-lo de duas maneiras, sendo elas:

1 – Atribuindo os valores na mesma ordem que a Stored Procedure os recebe.

```
EXEC Sel_Clientes 45104130843, 'A'
```

No exemplo acima o primeiro parâmetro aceito pela Stored Procedure será atribuído o valor 45104130843, e o segundo parâmetro o caractere A.

2 – Definindo o valor de cada parâmetro.

```
EXEC Sel_Clientes @CPF = 45104130843,
    @Indicador_Ativo = 'A'
```

No exemplo acima o parâmetro de nome @CPF será atribuído o valor 45104130843 e o parâmetro @Indicador_Ativo será atribuído o valor A.

Caso seja necessário recuperar o retorno da execução da Stored Procedure podemos fazê-lo de acordo com a imagem abaixo.

```
DECLARE @Retorno int
EXEC @Retorno = Nome_Procedure
```

Note que foi preciso criar uma variável que será atribuído o valor retornado pela Stored Procedure, vale lembrar que a definição do tipo da variável criada deve ser do mesmo tipo de retorno do procedimento, ou seja, no exemplo acima a Stored Procedure irá retornar um inteiro.

Podemos também alterar ou excluir uma procedure já existente no banco de dados, podemos conferir a sintaxe destes comandos nas imagens abaixo.

```
/* Para alterar uma procedure. */  
ALTER PROCEDURE Nome_Procedure  
  
/* Para excluir uma procedure do banco de dados. */  
DROP PROCEDURE Nome_Procedure
```

5.13 - Functions

Funções em T-SQL são procedimentos que retornam valores ou tabelas, e diferentes das Stored Procedure não são pré compiladas no banco de dados, isso quer dizer que toda vez que uma função é acionada ela é compilada pelo banco de dados, executada e retornado um valor ou tabela.

Abaixo podemos conferir a sintaxe de como criar uma função.

```
CREATE FUNCTION Nome_Function  
(  
    @Parametros int  
)  
  
RETURNS int  
  
BEGIN  
  
    RETURN 10  
  
END
```

Onde:

Nome_Function: Nome da função que será criada no banco de dados.

@Parametros: Lista dos parâmetros que serão aceitos pela função, note que o nome do parâmetro precisar ser iniciado com o caractere @, caso seja necessário adicionar outros parâmetros basta separa-los por virgula.

Returns: Tipo de dado que a função ao ser executada irá retornar.

Perceba que o código que irá compor a função precisa estar entre as declarações BEGIN e END.

Após criada a função podemos executa-la conforme ilustrado na imagem abaixo.

```
SELECT [dbo].[Nome_Function](Parametros)
```

Note que para acionar uma função devemos utilizar do comando SELECT, e para passar os parâmetros devemos informa-los entre parêntese, e caso seja mais de um parâmetro separa-los por virgula.

Ao acionar uma função devemos passar todos os parâmetros que a função aceita sem exceção, e na mesma ordem.

Podemos também alterar ou excluir uma função já existente no banco de dados, podemos conferir a sintaxe destes comandos nas imagens abaixo.

```
/* Para alterar uma função. */  
ALTER FUNCTION Nome_Function  
  
/* Para excluir uma função do banco de dados.  
DROP FUNCTION Nome_Function
```

5.14 - Triggers

Triggers (gatinho em português), são disparadas mediante alguma ação, geralmente essas ações que acionam as triggers são operações que ocorrem nas tabelas por meio de alguma inserção, atualização ou exclusão (INSERT, UPDATE ou DELETE).

Um gatilho está relacionado a uma tabela, e ao ocorrer algumas das operações citadas acima sobre a tabela, a trigger é automaticamente acionada.

A sintaxe para criar um trigger é a seguinte:

```
CREATE TRIGGER Nome_Gatilho  
ON Nome_Tabela  
FOR INSERT, UPDATE, DELETE  
  
AS  
  
BEGIN  
    -- Corpo do gatilho.  
  
END
```

Onde:

Nome_Gatilho: Nome da trigger que será criada no banco de dados.

Nome_Tabela: Tabela a qual o gatilho estará ligado, para ser disparado mediante as ações de inserção, atualização ou exclusão.

For: Ação sobre a tabela que irá disparar o gatilho.

Perceba que o código que irá compor a trigger precisa estar entre as declarações BEGIN e END.

Podemos também alterar ou excluir uma trigger já existente no banco de dados, podemos conferir a sintaxe destes comandos nas imagens abaixo.

```

/* Para alterar uma trigger */
ALTER TRIGGER Nome_Trigger

/* Para excluir uma trigger */
DROP TRIGGER Nome_Trigger

```

Exercício

Crie todas as Stored Procedure do sistema que será desenvolvido, as Stored Procedures criadas deverão estar relacionadas dentro de um mesmo arquivo com o nome de **CursoFerias.sql**.

5.15 - Algumas Funções do T-SQL

Na tabela abaixo podemos visualizar algumas funções disponíveis para utilização na linguagem T-SQL.

Função	Ação
GETDATE	Retorna a data atual, incluindo hora, minuto, segundo e milissegundo.
YEAR	Retorna o ano da data especificada.
MONTH	Retorna o mês da data especificada.
DAY	Retorna o dia da data especificada.
DATEADD	Retorna uma nova data, com base no intervalo e data base especificados.
DATEDIFF	Retorna a diferença entre duas datas, com base no intervalo e data inicio e final especificados.
SUM	Retorna a somatória de todos os valores na expressão.
COUNT	Retorn o número de itens de um grupo.
MAX	Retorna o valor máximo na expressão.
MIN	Retorna o valor mínimo na expressão.
AVG	Retorna a média dos valores em um grupo.
ISNULL	Retorna o primeiro valornão NULLO da expressão.
UPPER	Retorna uma expressão de caracteres convertidos em letras maiúsculas.
LOWER	Retorna uma expressão de caracteres convertidos em letras minúsculas.
CAST / CONVERT	Converte uma expressão de um tipo de dados para outro.

6 - O que é Git?

Git é um controlador de versões. Essa ferramenta possibilita várias pessoas trabalharem no mesmo projeto simultaneamente editando, excluindo e criando sem alterar o projeto master(projeto principal).

Se não existisse um sistema controlador de versões, imagine o caos entre duas pessoas abrindo o mesmo arquivo ao mesmo tempo. Uma das aplicações do git é justamente essa, permitir que um arquivo possa ser editado ao mesmo tempo por pessoas diferentes. Por mais complexo que isso seja, ele tenta manter tudo em ordem para evitar problemas para nós desenvolvedores.

Outro fator importante do git é a possibilidade de criar, a qualquer momento, vários **branch** no seu projeto. Suponha que o seu projeto seja um site html, e você deseja criar uma nova seção no seu código HTML, mas

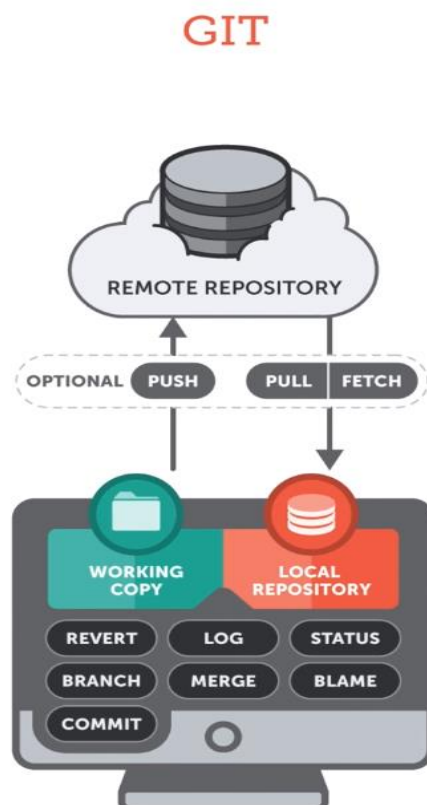
you don't want to make this available to anyone but you and your team of developers, summarizing, you want to change your project, but you don't want this to go to production, so you create a new branch (as if it were a mirror copy) and then you work only on this branch, until you get all the details right. After that, you just make a **merge** back to your original branch. Don't worry, we'll explain all these terms for you.

6.1 - Repositórios

Repository is where the files of a project are, like a folder for example. If you have a set of files that are part of a project, they must be located inside a folder in common, which is your repository.

Repositories are initially located locally on your computer, in some folder that contains your project. Maybe for someone who is working alone it will be sufficient, since there won't be interaction with another collaborator on the project, but if more than one person is working on a project, it is necessary a remote repository, or in other words, one that is not on your computer.

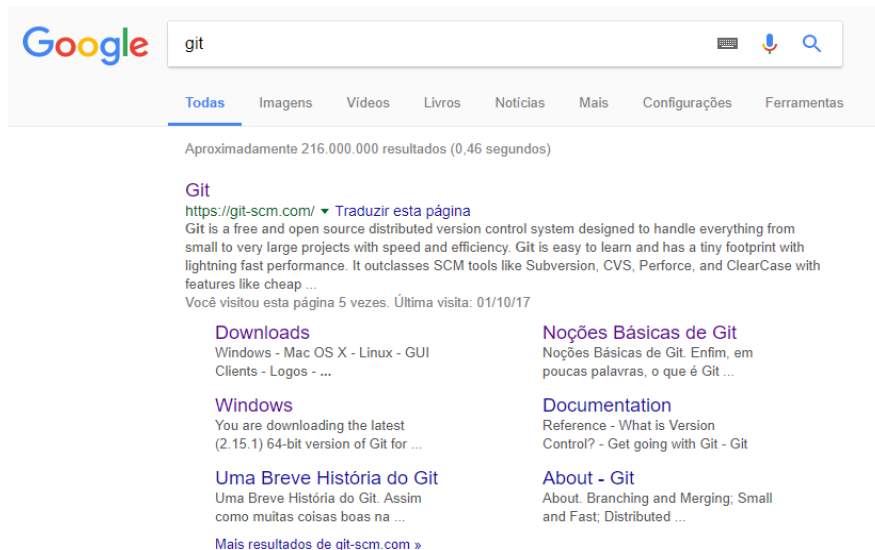
Remote repositories are located on sites like GitHub for example. The purpose of working in a remote repository is to always be up to date, together with other collaborators, either on the same branch or on others. Thus the changes or creations that you make, will be available for the other collaborators and vice-versa.



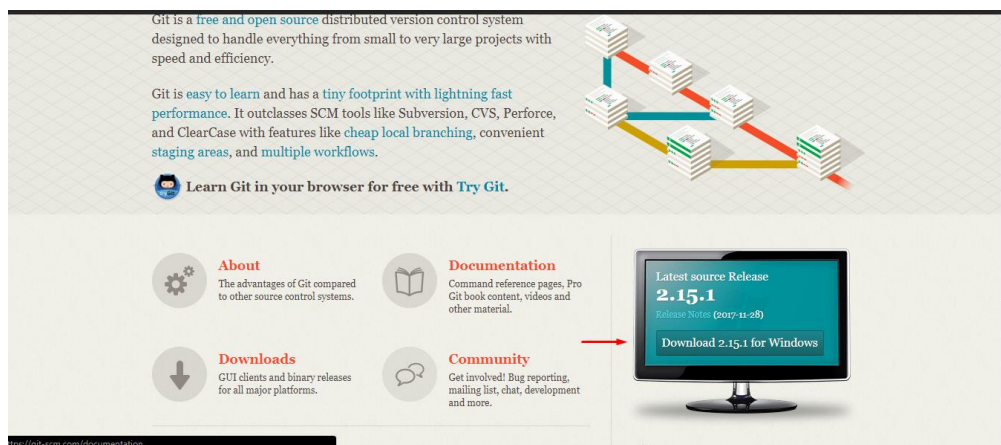
GitHub nada mais é do que um serviço web onde você pode guardar seus projetos em um repositório remoto gratuito.

6.2 - Instalando o GIT

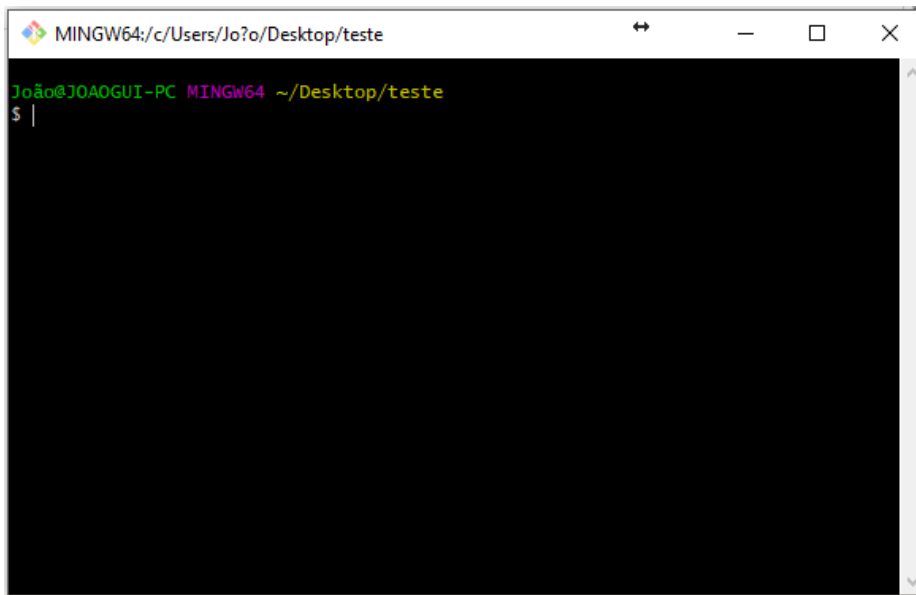
- 1- Abra seu navegador e pesquise por “GIT” no google.



- 2- Faça o download de acordo com seu sistema operacional.



- 3- Depois de instalado crie uma nova pasta no seu Desktop, dentro dessa pasta clique com o botão direito do mouse e selecione a opção “git bash here” e o terminal do git vai abrir.



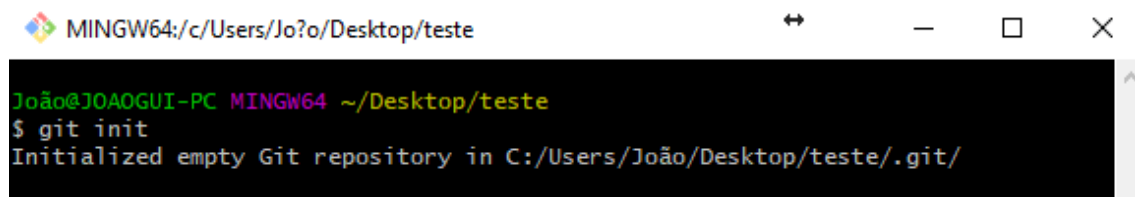
```
MINGW64:/c:/Users/João/Desktop/teste

João@JOÃOGUI-PC MINGW64 ~/Desktop/teste
$ |
```

6.3 - Comandos Básicos

Clonando e criando novos projetos

git init - O git init inicia um repositório vazio, ou reinicia um existente.

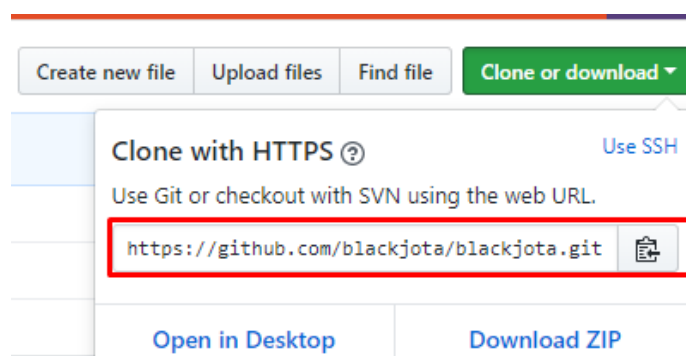


```
MINGW64:/c:/Users/João/Desktop/teste

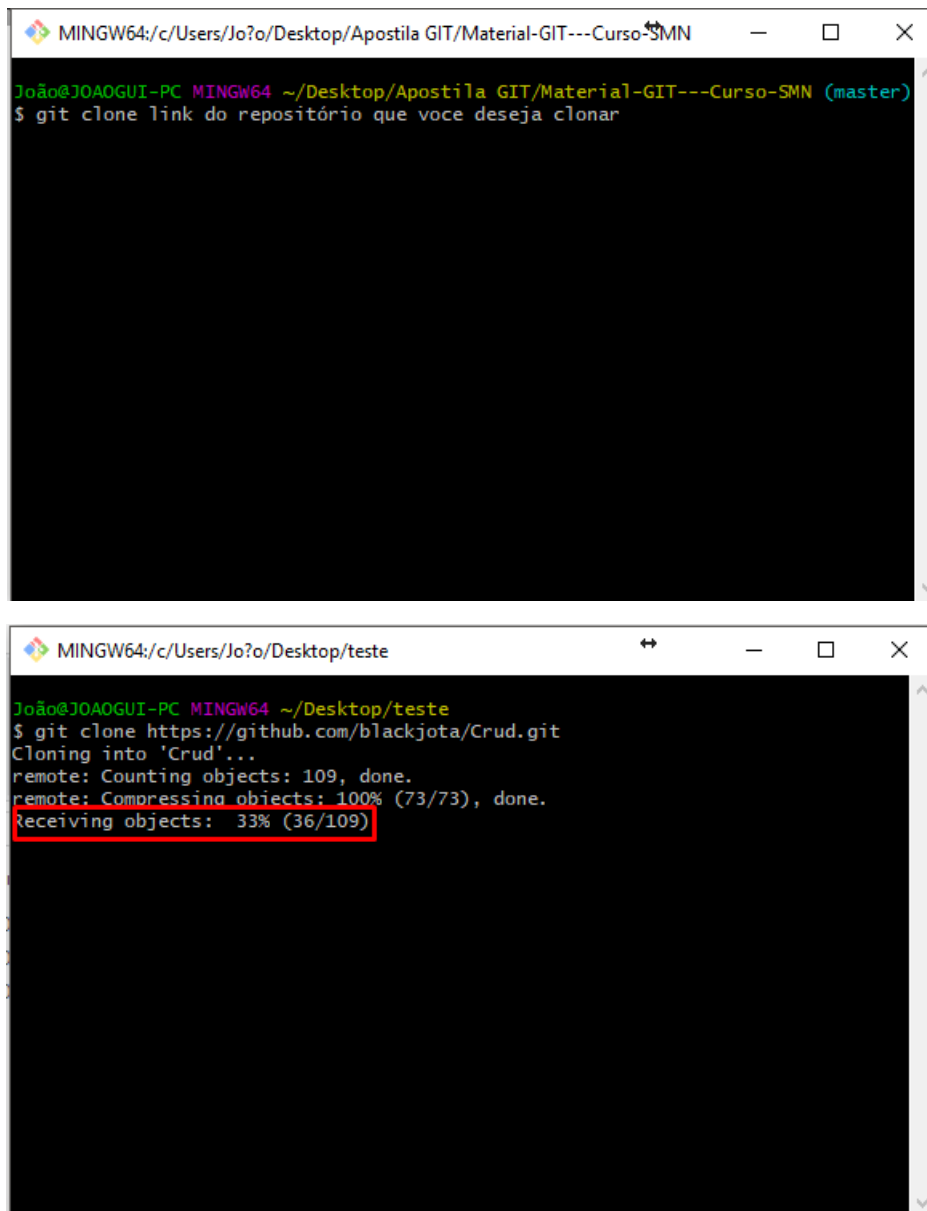
João@JOÃOGUI-PC MINGW64 ~/Desktop/teste
$ git init
Initialized empty Git repository in C:/Users/João/Desktop/teste/.git/
```

git clone- O git clone como o próprio nome já diz, clona o repositório desejado para utilizar.

No GitHub abra o repositório que deseja clonar e cline em “Clone or download” e copie o link que vai estar na caixa de texto como na imagem abaixo.



No seu terminal git, digite git clone depois cole o link do repositório desejado (botão direito e PASTE) pressione enter e espere até o processo acabar.



The image consists of two screenshots of a Windows command prompt window. The top screenshot shows the user at a prompt in the directory 'c:/Users/Jo?o/Desktop/Apostila GIT/Material-GIT---Curso-SMN'. The user has entered the command 'git clone link do repositório que voce deseja clonar'. The bottom screenshot shows the user at a prompt in the directory 'c:/Users/Jo?o/Desktop/teste'. The user has entered the command 'git clone https://github.com/blackjota/Crud.git'. The output shows the cloning progress: 'Cloning into 'Crud'...', 'remote: Counting objects: 109, done.', 'remote: Compressing objects: 100% (73/73), done.', and 'Receiving objects: 33% (36/109)'. The last line is highlighted with a red box.

```
MINGW64:/c:/Users/Jo?o/Desktop/Apostila GIT/Material-GIT---Curso-SMN
João@JOAOGUI-PC MINGW64 ~/Desktop/Apostila GIT/Material-GIT---Curso-SMN (master)
$ git clone link do repositório que voce deseja clonar

MINGW64:/c:/Users/Jo?o/Desktop/teste
João@JOAOGUI-PC MINGW64 ~/Desktop/teste
$ git clone https://github.com/blackjota/Crud.git
Cloning into 'Crud'...
remote: Counting objects: 109, done.
remote: Compressing objects: 100% (73/73), done.
Receiving objects: 33% (36/109)
```

Pronto, você clonou o repositório para a sua máquina.

6.4 - Commitando e atualizando seus projetos

git status – mostra todos os arquivos que foram modificados no seu projeto em vermelho.

```
MINGW64:/c/Users/Jo?o/Desktop/teste

João@JOAOGUI-PC MINGW64 ~/Desktop/teste (master)
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <files>" to include in what will be committed)

teste1.txt
teste2.txt

nothing added to commit but untracked files present (use "git add" to track)

João@JOAOGUI-PC MINGW64 ~/Desktop/teste (master)
$
```

git add --all ou git add . - Adiciona seus arquivos editados em uma “caixa”, que significa que estão prontos para serem commitados. Se não quiser commitar todos os arquivos, basta digitar `git add “nome do arquivo editado que deseja commitar”` sem aspas. E se dermos um “`git status`” novamente, o nome dos arquivos vai estar verde.

```
MINGW64:/c/Users/Jo?o/Desktop/teste

João@JOAOGUI-PC MINGW64 ~/Desktop/teste (master)
$ git add .

João@JOAOGUI-PC MINGW64 ~/Desktop/teste (master)
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <files>" to unstage)

new file:   teste1.txt
new file:   teste2.txt

João@JOAOGUI-PC MINGW64 ~/Desktop/teste (master)
$ ^C

João@JOAOGUI-PC MINGW64 ~/Desktop/teste (master)
$ |
```

git commit -m “Descrição” - Este comando tem a função de salvar suas modificações, e serão inseridas no histórico de commit do projeto para que sua equipe saiba o que você fez e aí vem o ‘-m’ que serve para colocar uma mensagem descrevendo seu commit como vemos no exemplo abaixo.

```
MINGW64:/c:/Users/Jo?o/Desktop/teste
João@JOAOGUI-PC MINGW64 ~/Desktop/teste (master)
$ git commit -m "Meu Primeiro commit"
[master (root-commit) ebfe09e] Meu Primeiro commit
2 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 teste1.txt
create mode 100644 teste2.txt
João@JOAOGUI-PC MINGW64 ~/Desktop/teste (master)
$ |
```

git push - O push como o nome em inglês já diz “Empurrar” simplesmente empurra a “caixa” em que você adicionou os arquivos pra commitar para o repositório que você está trabalhando. Lembre-se **Depois de um commit sempre tem um git push, para enviar seu commit para o repositório.**

git pull – Com o comando pull você pega todos os commits (modificações) que seu time fez no projeto para a sua máquina. **Lembre-se antes de um commit sempre de um git pull git pull assim você evita perder modificações de outras pessoas.**

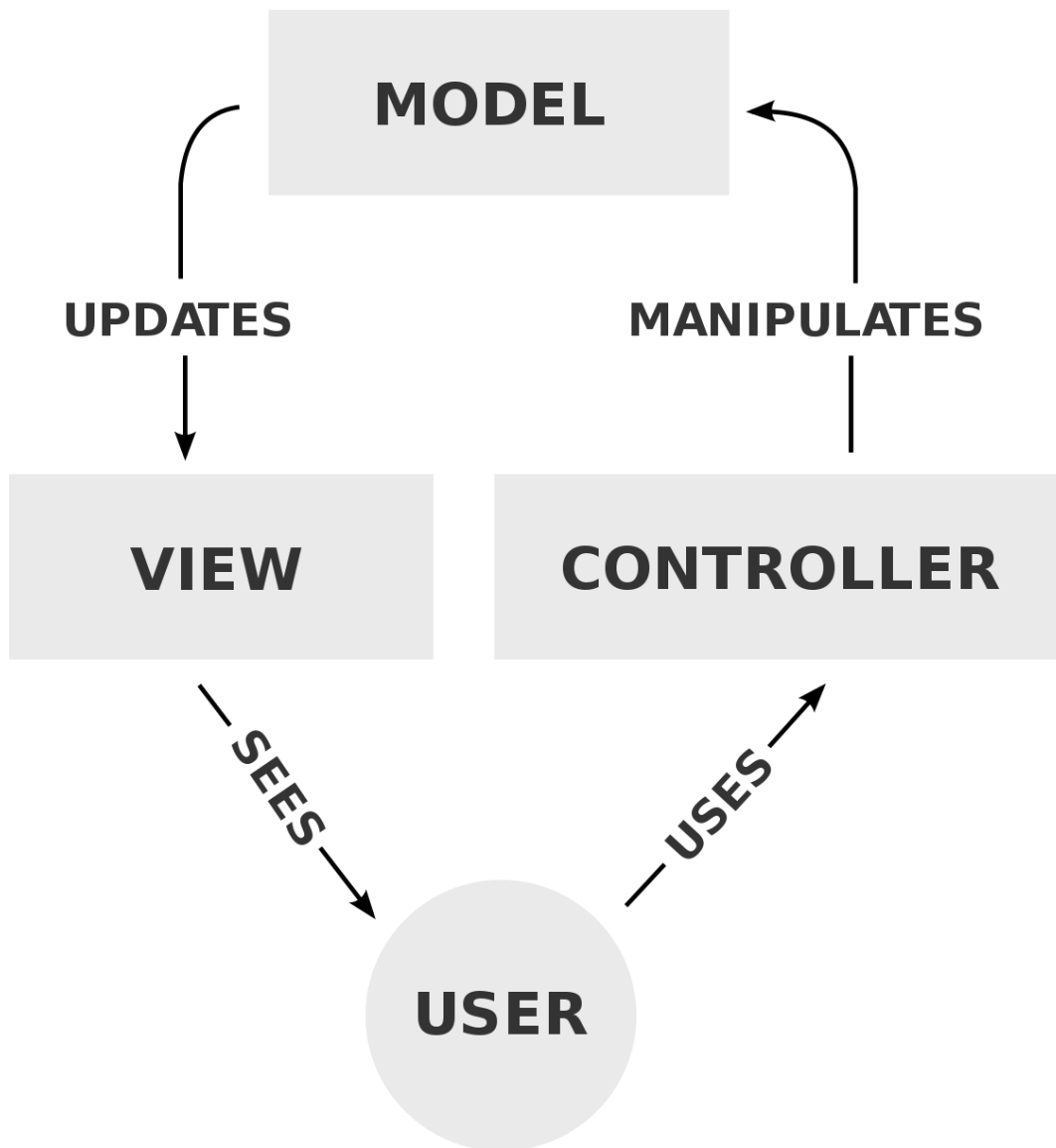
Criando branches e falando sobre merges

git checkout -b - O checkout sai do seu branch atual e o -b cria um novo branch, resumindo esse comando cria um novo branch sai do seu branch atual e entra no novo branch.

git merge- O merge nada mais é do que fazer a junção das suas alterações que você fez no branch com a master como está na imagem abaixo:



7 - ASP NET MVC



O padrão arquitetônico MVC (Model-View-Controller) separa um aplicativo em três componentes principais: modelo, exibição e controlador. A estrutura ASP.NET MVC oferece uma alternativa ao padrão Web Forms do ASP.NET para criar aplicativos Web. A estrutura ASP.NET MVC é uma estrutura de apresentação leve e altamente testável que (à semelhança dos aplicativos baseados em Web Forms) é integrada aos recursos ASP.NET existentes, como páginas mestras e autenticação baseada em associação. A estrutura MVC é definida no assembly **System.Web.Mvc**.

O MVC é um padrão de design padrão que muitos desenvolvedores conhecem. Alguns tipos de aplicativos Web irão se beneficiar da estrutura MVC. Outros vão continuar usando o padrão de

aplicativo ASP.NET tradicional que é baseado em Web Forms e postbacks. Outros tipos de aplicativos Web irão combinar as duas abordagens; uma abordagem não exclui a outra.

O padrão MVC ajuda a criar aplicativos que separam os diferentes aspectos do aplicativo (lógica de entrada, lógica de negócio e lógica da IU), enquanto fornece um acoplamento flexível entre esses elementos. O padrão especifica onde cada tipo de lógica deve ficar localizado no aplicativo. A lógica da IU fica na exibição. A lógica de entrada fica no controlador. A lógica de negócios fica no modelo. Essa separação ajuda a administrar a complexidade quando você cria um aplicativo, porque ela permite que você se concentre em um aspecto da implementação por vez. Por exemplo, você pode se concentrar na exibição sem depender da lógica de negócios.

O acoplamento flexível entre os três componentes principais de um aplicativo MVC também promove o desenvolvimento paralelo. Por exemplo, um desenvolvedor pode trabalhar na exibição, um segundo desenvolvedor pode trabalhar na lógica do controlador e um terceiro desenvolvedor pode se concentrar na lógica de negócios do modelo.

7.1 - Por que utilizar MVC?

Com o aumento da complexidade dos sistemas/sites desenvolvidos hoje, essa arquitetura tem como foco dividir um grande problema em vários problemas menores e de menor complexidade. Dessa forma, qualquer tipo de alterações em uma das camadas não interfere nas demais, facilitando a atualização de layouts, alteração nas regras de negócio e adição de novos recursos. Em caso de grandes projetos, o MVC facilita muito a divisão de tarefas entre a equipe.

Abaixo serão listadas algumas das vantagens em utilizar MVC em seus projetos:

- Facilita o reaproveitamento de código;
- Facilidade na manutenção e adição de recursos;
- Maior integração da equipe e/ou divisão de tarefas;
- Diversas tecnologias estão adotando essa arquitetura;
- Facilidade em manter o seu código sempre limpo;

Definição das camadas

A estrutura MVC inclui os seguintes componentes:

- **Modelos(Models)**. Os objetos de modelo são as partes do aplicativo que implementam a lógica para o domínio de dados do aplicativo. Muitas vezes, os objetos de modelo recuperam e armazenam o estado do modelo em um banco de dados. Por exemplo, um objeto Product pode recuperar informações de um banco de dados, operar nele e, em seguida, gravar informações atualizadas de volta em uma tabela de Produtos em um banco de dados do SQL Server.

Em aplicativos pequenos, o modelo, muitas vezes, é uma separação conceitual em vez de física. Por exemplo, se o aplicativo apenas ler um conjunto de dados e enviá-lo para exibição, o aplicativo não terá uma camada de modelo físico nem classes associadas. Nesse caso, o conjunto de dados assume a função de um objeto de modelo.

O modelo (Model) é utilizado para manipular informações de forma mais detalhada, sendo recomendado que, sempre que possível, se utilize dos modelos para realizar consultas, cálculos e todas as regras de negócio do nosso site ou sistema.

- **Exibições(Views)**. As exibições são os componentes que exibem a interface do usuário (IU) do aplicativo, ou seja, as telas. Normalmente, esta IU é criada a partir dos dados do modelo. Um

exemplo seria uma exibição de edição de uma tabela de Produtos que mostra caixas de texto, listas suspensas e caixas de seleção com base no estado atual de um objeto Product. A visão (View) é responsável por tudo que o usuário final visualiza, toda a interface, informação, não importando sua fonte de origem.

- **Controladores(Controllers).** Os controladores são os componentes que lidam com a interação do usuário, trabalham com o modelo e, finalmente, selecionam uma exibição de renderização que mostra essa IU(View). Em um aplicativo MVC, a exibição só mostra informações; o controlador manipula e responde à entrada e à interação do usuário. Por exemplo, o controlador manipula valores da cadeia de consulta e passa esses valores ao modelo, que por sua vez pode usar estes valores para consultar o banco de dados.

7.2 - Exemplo do funcionamento do MVC

Com a teoria já compreendida, vamos imaginar a seguinte situação: Você desenvolveu um site, e esse site possui uma tela de login, onde o usuário digita seu login e sua senha, após a autenticação, caso ocorra tudo certo, o usuário acessa a área restrita do site, caso contrário é redirecionado novamente para a página de login repassando uma mensagem que a combinação de usuário e senha é inválida.

Conseguiu imaginar essa situação? Beleza... Agora veja como isso acontece caso você ainda não tenha adotado a arquitetura MVC em seu site: Primeiramente, o usuário preenche o formulário com seu login e sua senha e pressiona o botão “Logar”.

Depois disso, o formulário envia essas informações para um arquivo onde, no mesmo arquivo, você executa as seguintes etapas:

1. Armazena em variáveis os dados digitados pelo usuário;
 2. Montam um comando SQL para selecionar o usuário;
 3. Verifica se retornou alguma informação;
- Se retornar alguma informação, armazena o usuário em uma sessão e redireciona para a área restrita;
 - Se não retornar nenhuma informação, redireciona para a página de login com uma mensagem notificando que a combinação digitada é inválida;

Aparentemente está tudo ok, tudo funcionando. Mas veremos agora como funcionaria se o seu site estivesse utilizando a arquitetura MVC: Os passos seguem os mesmos, primeiramente, o usuário preenche o formulário com seu login e sua senha e pressiona o botão “Logar”. Agora veremos algumas mudanças.

Depois disso, o formulário envia essas informações para uma controladora, e essa controladora realizará as seguintes etapas:

1. A controladora (controller) carrega um modelo (model), e executa um método que realiza a validação;

2. No modelo (model) são executadas as seguintes tarefas:

- Armazena as informações digitadas pelo usuário;
 - Realiza a consulta no banco retornando verdadeiro (true) em caso de sucesso (usuário existe e a senha bate), ou falso (false) no caso de a combinação das informações digitadas serem inválidas;

3. A controladora (controller) verifica o que o modelo retornou;

- Se retornar verdadeiro (true) armazena as informações em uma sessão e redireciona o usuário para visão (view) da área restrita;
- Se retornar falso (false) redireciona o usuário de volta para a tela (view) de login repassando a mensagem que a combinação digitada é inválida;

Agora você pode ficar se perguntando, mas do modo que eu faço também funciona? Pode ser que sim, mas imagine ter que alterar a regra de negócio. Antes de utilizar MVC você precisaria abrir o arquivo que realiza todas as tarefas e localizar a sua regra, para depois alterar. No caso do MVC, você já sabe onde se encontra as suas regras de negócio, então você vai direto ao arquivo. Por isso que a maioria dos frameworks já vem com sua estrutura de diretórios pronta, facilitando a localização dos arquivos.

7.3 - ViewData, ViewBag e TempData

ViewData e ViewBag são similares nas seguintes características:

- São utilizadas para persistir dados entre a Controller e a View correspondente.
- A duração “tempo de vida” é apenas entre o envio através da Controller e a exibição na View, depois disso tornam-se nulas novamente.
- No caso de um redirect se tornam nulas.

Diferenças entre ViewData e ViewBag:

ViewData	ViewBag
É um dicionário de objetos derivado de ViewDataDictionary e é acessível utilizando strings como chaves.	É uma propriedade dinâmica baseada na funcionalidade “dynamic” do C# 4.0

Requer typecasting (conversão) quando associada a tipos complexos.

Não necessita de conversão para tipos complexos.

Exemplos de aplicação:

Controller

```
1 public class HomeController : Controller
2 {
3     public ActionResult Index()
4     {
5         // Meu dado de tipo complexo
6         var func = new Funcionario
7         {
8             Nome = "Eduardo Pires",
9             Idade = 31
10        };
11
12        // Propriedades "Dinâmicas"
13        ViewBag.Funcionario = func;
14
15        // Modo tradicional
16        ViewData["Funcionario"] = func;
17
18        return View();
19    }
20 }
```

View

```
1 @model ProjetoModelo.Models.Funcionario;
2
3 @{
4     ViewBag.Title = "Exemplo ViewData ViewBag";
5
6     // Necessita de TypeCasting
7     var viewDataVariavel = ViewData["Funcionario"] as
Funcionario;
8
9     // Não necessita de TypeCasting
10    var viewBagVariavel = ViewBag.Funcionario;
11 }
```

Resumindo, ViewData e ViewBag possuem a mesma proposta, porém o ViewBag está disponível a partir do ASP.Net MVC 3, enquanto o ViewData existe desde a primeira versão.

O ViewData é um wrapper, uma implementação do ViewBag, pois utiliza o ViewBag internamente, portanto:

Por este motivo ViewData é mais rápido que o ViewBag, porém essa diferença de velocidade é mínima, não é necessário deixar de usar o ViewBag por este motivo.

TempData

- TempData assemelha-se mais a uma sessão de servidor, porém de curta duração.
- Possui um tempo de vida maior que o ViewBag e ViewData, o TempData perdura desde sua criação até que seja chamado, ou seja, quando houver um request da informação do TempData, ele se tornará nulo novamente.
- Uma informação em TempData criada em um Controller persiste após um redirect entre actions (apenas um) e pode ser exibido em sequência em uma View (muito usado em tratamento de erros).
- Caso não seja chamado o TempData pode manter o estado de seus dados até que a sessão do usuário se encerre.
- É utilizado para compartilhar informações entre Controllers.
- O TempData salva suas informações no SessionState do servidor.
- Após a leitura os dados do TempData são marcados para deleção, ou seja, no final do request todos os dados marcados serão deletados.
- É um benefício quando necessário transmitir um volume de informações entre as Controllers sem se preocupar em zerar os valores, pois o TempData automaticamente faz isso.

Exemplo de aplicação

- Controller

```
1 public class HomeController : Controller
2 {
3     [HttpPost]
4     public ActionResult CriarFuncionario(Candidato cd)
5     {
6         // Meu dado de tipo complexo
7         var func = new Funcionario
8         {
9             Nome = cd.Nome,
10            Idade = cd.Idade
11        };
12
13        // Persistir dados até o próximo request.
14        TempData["Funcionario"] = func;
15
16        // Redirect entre Controllers
17        return RedirectToAction("CriarBeneficiosFuncionario");
18    }
19
20    [HttpGet]
21    public ActionResult CriarBeneficiosFuncionario()
22    {
23        // Validando se está vazio
24        if (TempData["Funcionario"] != null)
25        {
26            // Necessário TypeCasting para tipos complexos.
27            var func = TempData["Funcionario"] as Funcionario;
28        }
29
30        return View();
31    }
32 }
```

Neste exemplo pudermos entender que o propósito do TempData é compartilhar dados entre Controllers, portanto sua duração persiste até que a informação seja lida. Outro detalhe é sempre checar se o TempData não está nulo.

Caso você queira manter o dado de um TempData mesmo após a leitura, basta chamar o método Keep(), assim o dado será persistido novamente até a próxima requisição.

- **View**

```
1 // Mantendo o dado do TempData até a próxima leitura (requisição).
2 TempData.Keep("Funcionario");
3
4 // Removendo o dado do TempData desta e da próxima requisição.
5 TempData.Remove("Funcionario");
```

Recomenda-se utilizar sempre ViewBag e ViewData para transferência de dados entre Controller e View. O TempData em Views é recomendado no caso de um dado necessitar ser redirecionado entre Actions e posteriormente ser exibido numa View (ViewBag e ViewData são anulados em redirects).

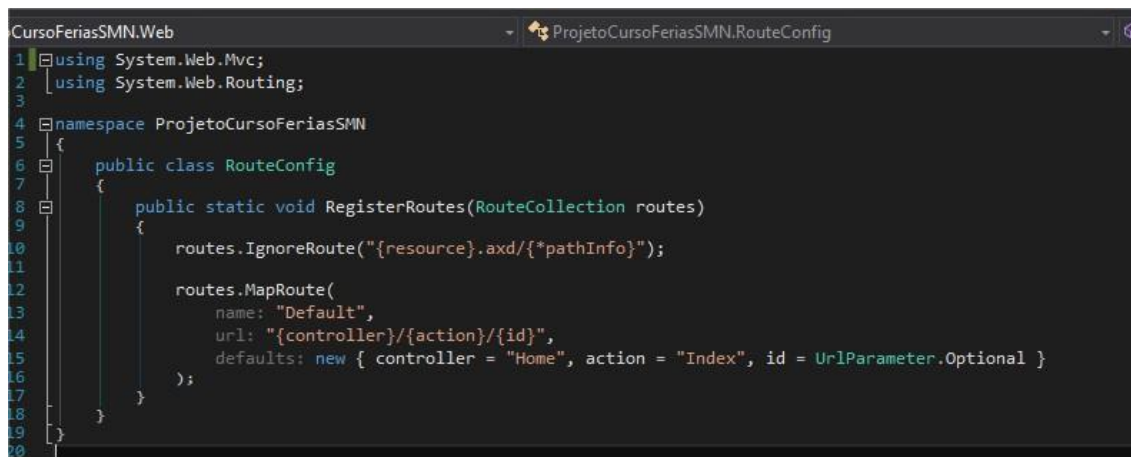
7.4 - Rotas

Antes do surgimento do MVC Framework, o ASP.NET assumiu que havia uma ligação direta entre URLs solicitadas e os arquivos no disco rígido do servidor, cabendo a ele receber o pedido do navegador e entregar a saída do arquivo correspondente.

Essa abordagem funciona muito bem para Web Forms, mas não para uma aplicação MVC, onde os pedidos são processados por **Actions** em classes de **Controller**. Não existe uma correlação um-para-um para os arquivos no disco. Esse sistema de rotas tem duas funções:

- Examinar uma URL de entrada para o Controller e a Action a que se destinam.
- Gerar URLs de saída, que é o HTML renderizado a partir das nossas Views.

As rotas padrões definidas no arquivo RouteConfig.cs fazem com que as URLs sejam entendidas como Controller/Action, mas podemos configurar, definir novas rotas personalizadas que nos permitem dizer que uma URL irá acessar a Action do Controller que desejarmos, criando assim URL's amigáveis para o usuário.



```

1 using System.Web.Mvc;
2 using System.Web.Routing;
3
4 namespace ProjetoCursoFeriasSMN
5 {
6     public class RouteConfig
7     {
8         public static void RegisterRoutes(RouteCollection routes)
9         {
10             routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
11
12             routes.MapRoute(
13                 name: "Default",
14                 url: "{controller}/{action}/{id}",
15                 defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
16             );
17         }
18     }
19 }
20

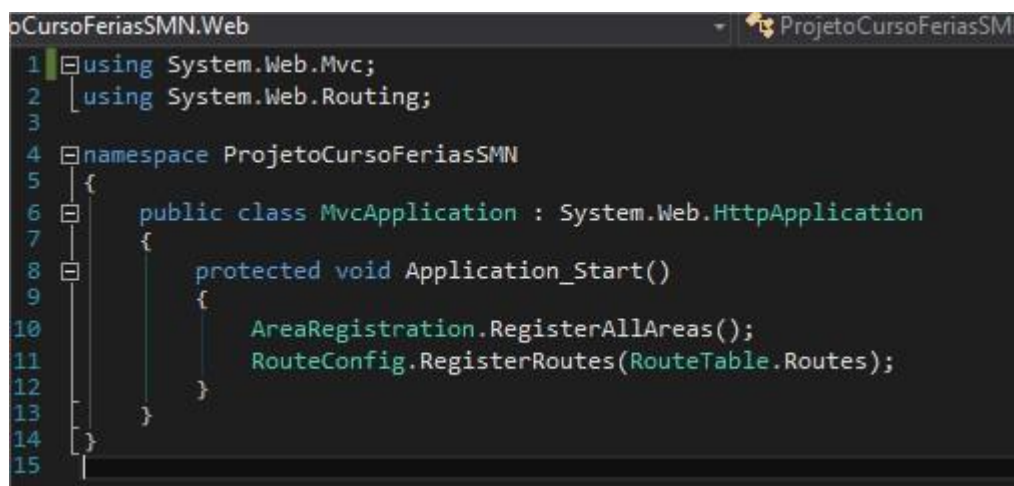
```

Na imagem acima podemos ver a classe RouteConfig citada acima, ela tem inicialmente a rota padrão nomeada “Default” que diz que as URL’s serão entendidas como:

Controller / Action que será acessada / ID(parâmetro opcional)

O Controller padrão é o Home, e a Action padrão é a Index, mas se você acessar uma URL dizendo que quer acessar um Controller diferente, mas sem especificar uma Action esta continuará por padrão sendo a Index.

O método estático RegisterRoute, que é definido no arquivo RouteConfig.cs, é chamado a partir do Arquivo Global.asax.cs



```

1 using System.Web.Mvc;
2 using System.Web.Routing;
3
4 namespace ProjetoCursoFeriasSMN
5 {
6     public class MvcApplication : System.Web.HttpApplication
7     {
8         protected void Application_Start()
9         {
10             AreaRegistration.RegisterAllAreas();
11             RouteConfig.RegisterRoutes(RouteTable.Routes);
12         }
13     }
14 }
15

```

O método Application_Start é chamado pela plataforma ASP.NET sempre quando a aplicação for iniciada. Sendo assim, podemos usar o Global.asax.cs para definir configurações necessárias para a execução da aplicação, como o método RouteConfig.RegisterRoutes que é uma instância da classe RouteCollection.

Voltando ao assunto principal, vamos configurar uma nova rota, vamos configurar essa rota para montar uma URL amigável que redireciona o usuário a tela de login.

O processo é muito simples, basta seguir o exemplo da rota Default do RouteConfig.


```

using System.Web.Mvc;
using System.Web.Routing;

namespace ProjetoCursoFeriasSMN
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

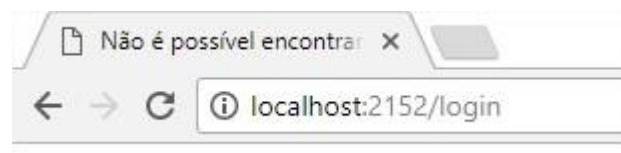
            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
            );

            routes.MapRoute(
                name: "Login",
                url: "login",
                defaults: new { controller = "Home", action = "Login" }
            );
        }
    }
}

```

Na imagem, na nova rota que criamos, o parâmetro name define um nome para a nossa rota, cada rota personalizada criada por você deve ter um nome diferente, o segundo parâmetro passado no MapRoute foi a url, nela nós definimos a URL que irá direcionar o usuário a tela de login, por último, no parâmetro defaults nós definimos o Controller que será acessado e a Action que será executada ao tentar acessar a rota definida na url.

Com esse trecho de código feito, quando o usuário tentar acessar a url abaixo ela irá nos direcionar ao Controller Home e executar a Action Login.



7.5 - Passando parâmetros aos Métodos

A passagem de parâmetros para os Controllers pode ser feita de várias formas diferentes, mas a principal é através da propriedade name dos inputs. Vamos supor o seguinte método:

```

public void MetodoExemplo(string primeiroParametro, string segundoParametro)
{
}

```

E o seguinte código HTML:

```

<form action="Exemplo/MetodoExemplo">
    <input type="text" placeholder="parâmetro 1"/>
    <input type="text" placeholder="parâmetro 2"/>
    <button type="submit">Enviar</button>
</form>

```

O resultado deste código é este formulário simples com os 2 campos que queremos enviar ao nosso método:



parâmetro 1 parâmetro 2 Enviar

O botão enviar irá executar o “MetodoExemplo” no nosso Controller, mas como faremos para que os dois parâmetros que estamos recebendo cheguem ao método?

A resposta é simples, como foi citado no primeiro parágrafo deste tópico, basta dar aos inputs a propriedade name, passando como valor desta propriedade o nome da variável correspondente no Controller.

Isto quer dizer que para que o nosso primeiro parâmetro do método “MetodoExemplo”, no caso “primeiroParametro” do tipo string chegue até o método, o nosso primeiro input precisa ter a propriedade name atribuída como “primeiroParametro”, a mesma coisa vale para o segundo parâmetro.