**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 aplica-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.



* No exemplo, *int*é o tipo e numero 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 e é atribuído a ele o modificador padrão *private*.

**1.2 - Modificadores**

No caso das varáveis, 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.



**No C# existem os seguintes modificadores:**

|  |  |
| --- | --- |
| **Modificador** | **Funcionamento** |
| Public | O acesso não é restrito |
| Protected | O acesso é limitado ás 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 |

**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.**

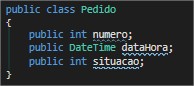
1. Struct: é dividida em tipos numéricos, bool e estruturas personalizadas pelo usuário.

* Tipo bool representa um valor que pode ser verdadeiro ou falso, o seu valor padrão é false.

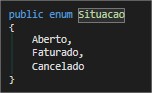
|  |  |
| --- | --- |
| **Tipo de dados** | **Intervalo** |
| Byte | 0...255 |
| Sbyte | -128...127 |
| Short | 32.768...32.767 |
| Ushort | 0...65,535 |
| Int | -2.147.483,648...2.147.483,647 |
| Uint | 0...4,294,967,295 |
| Long | -9.223.372.036.854.755,808... -9.223.372.036.854.755,807 |
| Ulong | 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.

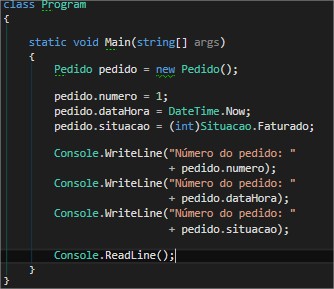
* Normalmente é usada quando em algum momento existe a necessidade de um atributo que pode ter múltiplos valores, como por exemplo, em uma aplicação de venda que tem a tela de pedidos: cada pedido tem a sua situação que pode ser Aberto, Faturado e Cancelado. Para criar a classe Pedido, o tipo do atributo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



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.



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 casodo exemplo éatribuído o valor 1.



**1.4 - Variáveis Reference Types**

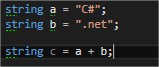
As variáveis reference type armazenam apenas a referência do objeto. Os tipos de referência são: class, interface, delegate, object, string e Array.

a) Tipo object: todos os tipos são derivados da classe Object, sendo assim é possível converter qualquer tipo para object.

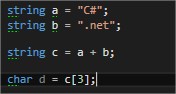
b) Tipo string: é utilizado para se armazenar caracteres e uma string deve estar entre aspas, como mostra a seguir.



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



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



* 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.



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.

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:







**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/Cientifico | 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 | fffffc18 |

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Especificador** | **Formato** | **Tipo** | **Saida** |
| 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.**

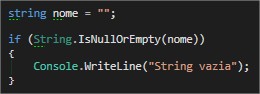
|  |  |  |  |
| --- | --- | --- | --- |
| **Especificador** | **Formato** | **Tipo** | **Saida** |
| dd | {0:dd} | Dia | 21 |
| ddd | {0:ddd} | Nome curto do dia | Qui |
| dddd | {0:dddd} | Nome inteiro do dia | Quinta-feira |
| hh | {0:hh} | Hora com 2 dígitos | 21 |
| HH | {0:HH} | Hora com 2 dig{24h} | 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 espeços reservados | 1.234.560 |
| # | {0:#.##} | Espepaço reservado para digito | 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**

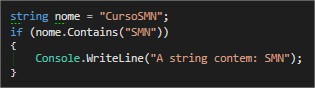
Muito útil para validar se o valor de uma variável é nulo ou vazio.



Simples 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.

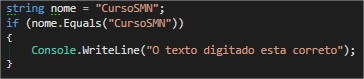


**O resultado pode ser a seguir:**

****

**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.



**Se a palavra for “CursoSMN“ mostra a mensagem a seguir:**



**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#”;

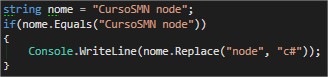
C:\Users\Gustavo\AppData\Local\Microsoft\Windows\INetCache\Content.Word\CursoSMN - Microsoft Visual Studio_5.jpg

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



**1.9 - Replace**

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

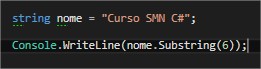


Resultado do método Replace:



**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,

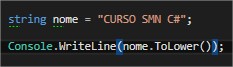


Resultado do método Substring:

****

**1.11 - ToLower**

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



**Fazendo a string ficar minúscula**

****

**1.12 - ToUpper**

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

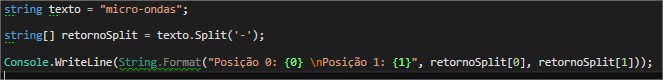


**Fazendo a string ficar maiúscula**



**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:

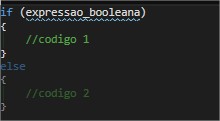
Note que o método Split espera um char como parâmetro, por isso são usadas as aspas simples.

****

**1.14 - 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:**

****

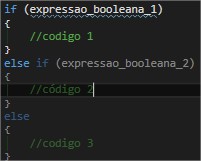
Caso a expressão booleana seja verdadeira, as instruções entre chaves presentes no código 1 serão executadas; caso contrário, serão executadas as instruções presentes no código 2.

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**

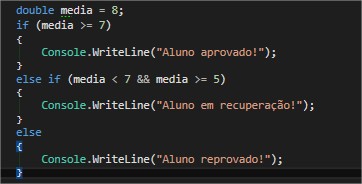
Complementar ao if/else temos o operador else if que traz uma nova condição a ser testada no caso de falha no teste da condição anterior.

Sintaxe do else if:

****

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 (>=) a 7. Para a recuperação, ele precisa ter média menor (<) que 7 e (&&) maior ou igual (>=) a 5. Por fim, para ser reprovado, precisa ter média menor (<) do que 5.

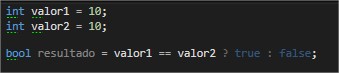
Note que temos três condições a serem verificadas, o que nos leva à necessidade de uso do if, else if, else.

**Exemplo de uso:**

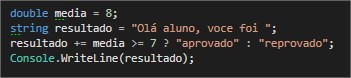
**1.15 - Operador ternário**

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.

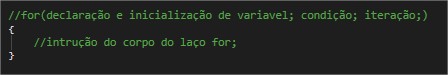
**Sintaxe do operador ternário:**



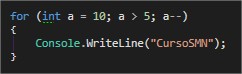
Para demonstrar um exemplo de uso do operador ternário, suponha que precisamos apresentar ao aluno o resultado com base na sua média. Para isso, criamos uma variável que recebe um texto padrão e que depois será incrementado para informar se o aluno está aprovado ou reprovado. Essa concatenação será feita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

  
**Exemplo de uso:**

**1.16 - 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**.

**Agora veja o seguinte exemplo, usando a sintaxe acima:**



Onde:

 - **int** refere-se ao tipo de variável que será declarada;

 - **a = 10** refere-se às variáveis que estão sendo inicializadas;

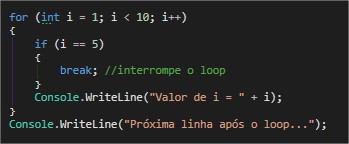
 - **a < 5** refere-se ao teste booleano que será feito na variável a;

 - **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**.

**1.17 - Break**

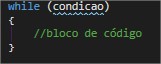
O comando **break** é usado em laços de repetição **while**, **do/while**, **for** e com os comandos **switch/case**. Quando usado em laço de repetição, causa uma interrupção imediata do mesmo, continuando a execução do programa na próxima linha após o laço. Isso ocorre caso a condição imposta seja atendida.



**1.18 - 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.

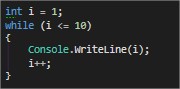
**Sintaxe da estrutura de repetição while:**



**Exemplo prático**

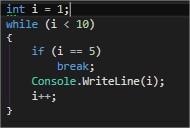
Para demonstrar o funcionamento do while, criamos um loop que imprimirá no console os valores de 1 a 10.

**Exemplo de uso:**



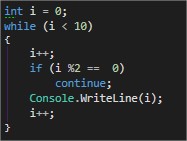
**1.19 - Break**

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.

**1.20 - Continue**

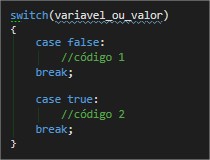
Além do break há outro comando capaz de modificar o fluxo natural de execução de loop: o continue. Esse comando faz com que a execução da iteração/repetição seja finalizada e o fluxo seja direcionado novamente para o início do loop. Dessa forma a condição de parada será novamente avaliada e o loop será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

****

**1.21 - Switch/Case**

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

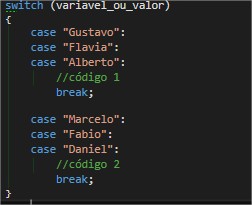
Para que isso fique mais claro, vejamos a sintaxe do switch/case:



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.

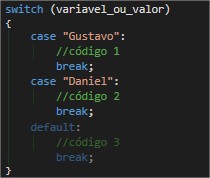
E o comando break? O comando break é utilizado para especificar a última linha de código a ser executada dentro da condição. Se não declarado, os códigos implementados dentro dos cases subsequentes serão executados.

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



**1.22 - Default**

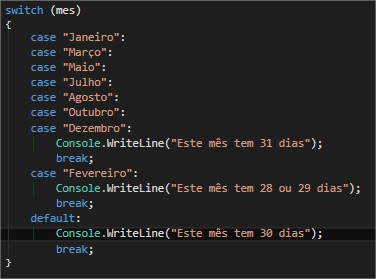
O operador default é utilizado quando precisamos definir um fluxo alternativo para as situações em que o valor contido no switch não seja atendido por nenhum dos **cases** especificados.



**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.

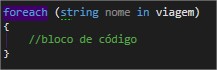
Com esse intuito, a partir de uma variável do tipo string (**mês**) implementamos o código abaixo:



**1.23 - Foreach**

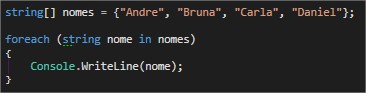
O foreach é um recurso do C# que possibilita executar um conjunto de comandos para cada elemento presente em uma coleção (Array, List, Stack, Queue e outras). Portanto, diferentemente do while e do for, não precisamos definir uma condição de parada. Isso é definido de forma implícita, pelo tamanho da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

Sintaxe da estrutura de repetição foreach:



**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.



**1.24 - 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



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



Inserção de dados no array

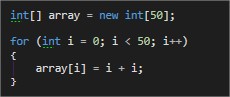
Existem duas formas básicas para inserirmos dados nos arrays em C#. A primeira é realizarmos isso durante a declaração do mesmo. Dessa forma, poderíamos ter a declaração das seguintes formas:



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:



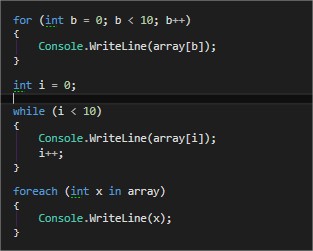
**1.25 - Acesso aos dados do array**

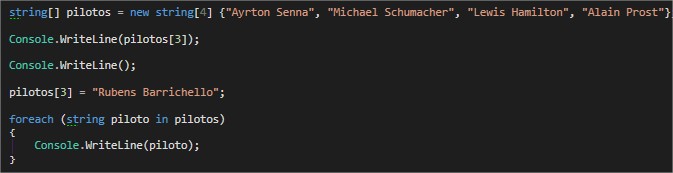
O acesso aos dados do array é feito de forma extremamente simples, uma vez que estamos lidando com uma coleção indexada. Assim, basta acessarmos o índice específico e podemos obter o valor do elemento, como podemos ver abaixo:



**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:

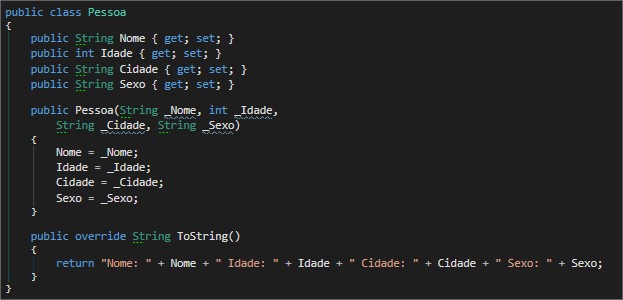


**Exemplo prático**

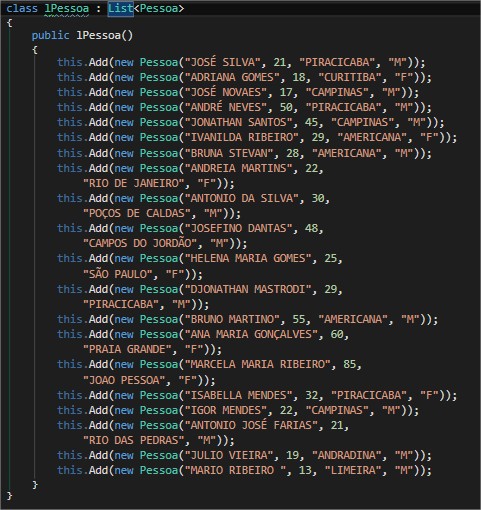
**1.26 - Usando Expressões Lambda com LINQ**

LINQ é uma linguagem de consulta que foi introduzida a partir do .Net Framewok 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.



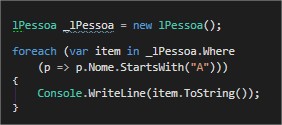
No mesmo arquivo Pessoa.cs vamos também implementar uma classe que é uma lista de pessoa. Essa classe conterá apenas um construtor sem parâmetros que servirá para alimentarmos nossa lista de pessoas, ou seja, ao instanciarmos essa lista ela já estará populada.

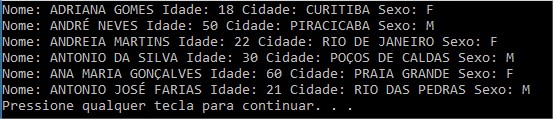


Agora que já criamos as classes necessárias, veremos a utilização de lambda com LINQ. A cada exemplo mostraremos a forma com e sem lambda para que você possa ver as diferenças entre os dois tipos.

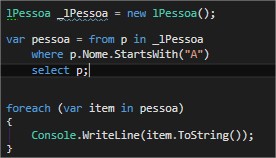
Primeiro vamos utilizar a nossa lista de pessoas, porém deveremos retornar na tela somente as pessoas que a primeira letra do Nome é “A”. Para isso devemos implementar a rotina utilizando Expressões Lambda.

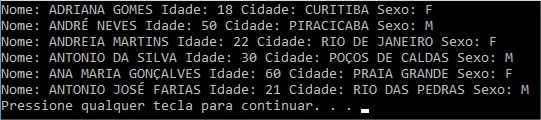
**Exemplo de lambda com LINQ**





**Exemplo com LINQ**





**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 maiusculas de minusculas.

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



**\*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.



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, iriamos utilizar o seguinte código:  

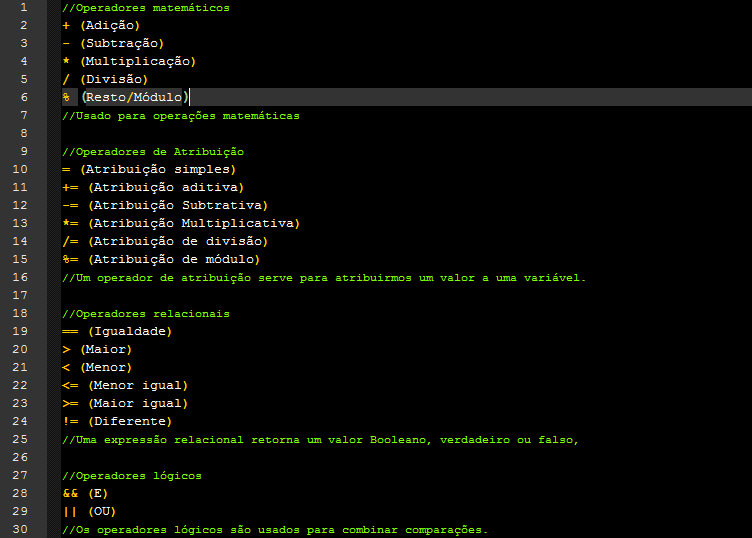

Se soubermos qual será o valor da variável no momento de sua declaração, podemos utilizar a seguinte sintaxe para declarar e atribuir o valor para a variável:

****

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



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:



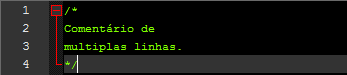
Podemos também manipular textos(strings). O operador de adição em strings executa uma ação diferente do que acontece com números, por exemplo, com números inteiros ele soma os dois números, já com strings ele concatena os textos, como podemos ver a seguir:



**\*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:



**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.

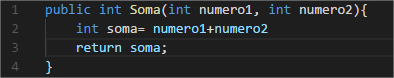
Após o nível de acesso declaramos o tipo de retorno do método. Quando ele não retorna nada usamos a palavra reservada ***void.***  Depois o nome do método com a primeira letra maiúscula, por padrão, e os parâmetros que o método recebe entre parênteses. E por fim instruções de código entre chaves(é o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

Exemplos:

Sem retorno:



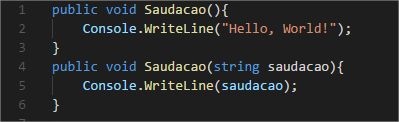
Com retorno e parâmetros:



**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:



**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 possuí atributos e métodos. O seguinte exemplo abaixo irá mostrar como funciona o processo de abstração:

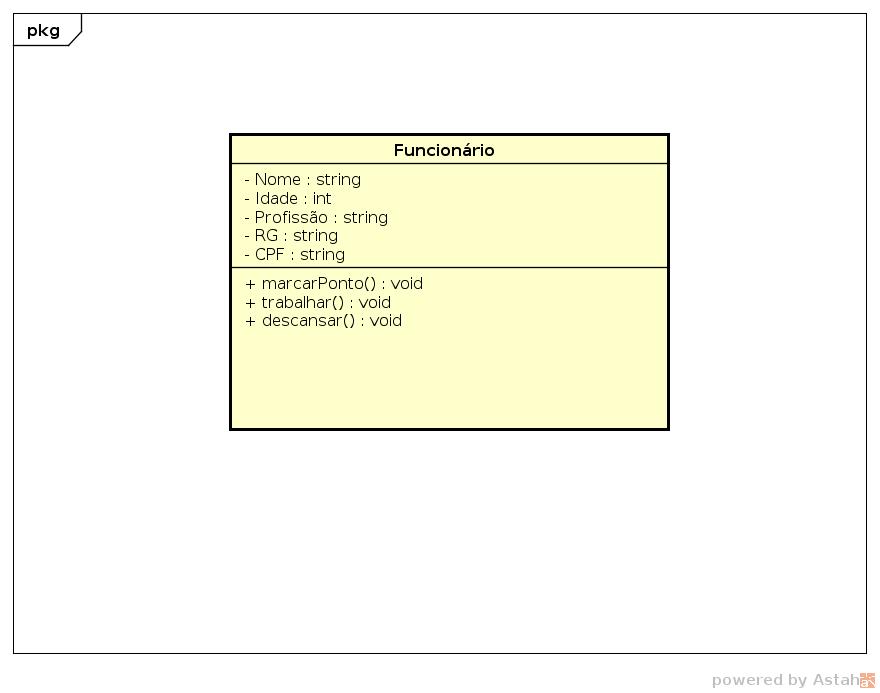
Vamos considerar o seguinte problema:

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.



**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.

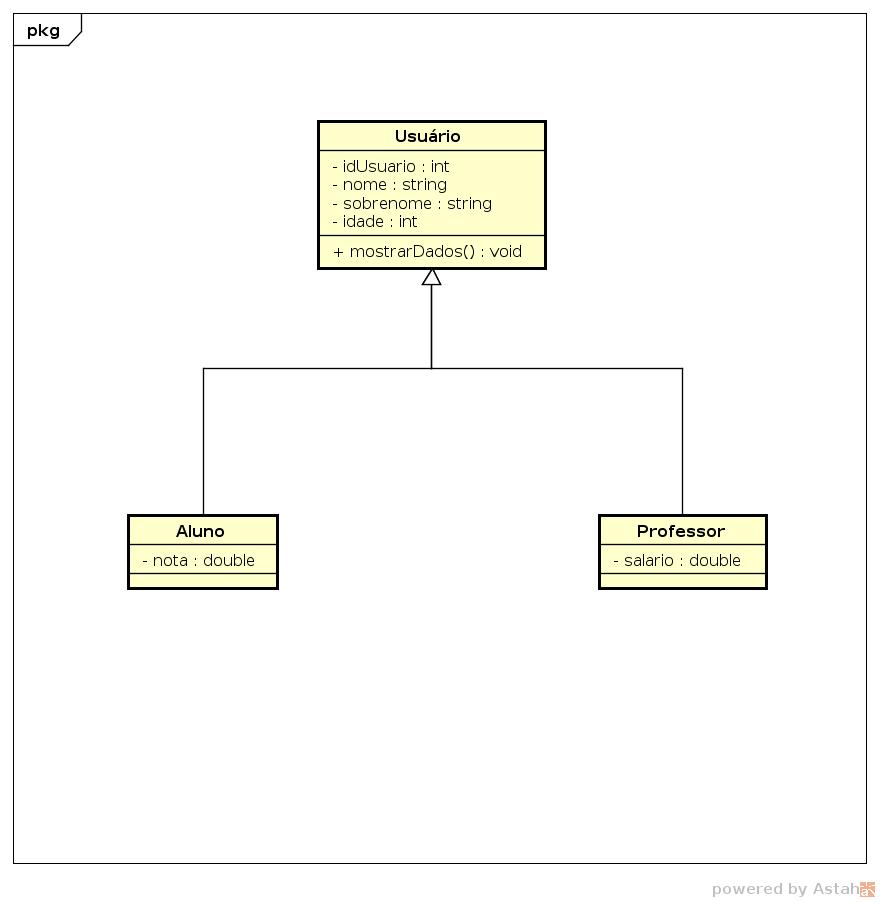
**Private -** Os métodos e atributos são acessíveis somente dentro da própria classe.

**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 destes. 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”.

**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.

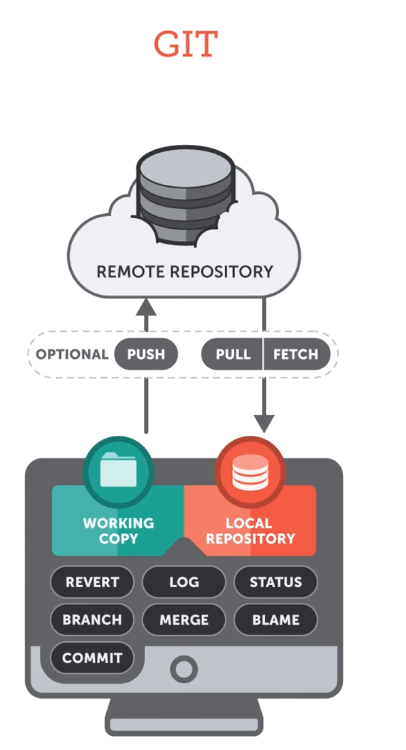
Outro fator importante do git é a possibilidade de criar, a qualquer momento, vários **branch** no seu projeto. Suponha que o seu projeto seja um site html, e você deseja criar uma nova seção no seu código HTML, mas você não quer disponibilizar isso para mais ninguém a não ser você e seu time de desenvolvedores, resumindo, você quer alterar seu projeto, mas não quer que isso suba para produção, então você cria um novo branch (como se fosse uma cópia espelho) e então trabalha apenas nesse branch, até acertar todos os detalhes dele. Depois é só você fazer um **merge** de volta do seu branch até o projeto original. Calma não se assuste, iremos explicar todos esses termos para vocês.

**6.1 - Repositórios**

Repositório é onde estão os arquivos de um projeto, como uma pasta por exemplo. Se você possui um conjunto de arquivos que fazem parte de um projeto, eles devem estar alocados dentro de uma pasta em comum, que é o seu repositório.

Os repositórios primeiramente estão localmente no seu computador, em alguma pasta que contenha seu projeto. Talvez para alguém que esteja trabalhando sozinho somente ele será suficiente, pois não teria a interação de outro colaborador no projeto, mas se mais de uma pessoa estiver em um projeto é necessário um repositório remoto, ou seja, que não está no seu computador.

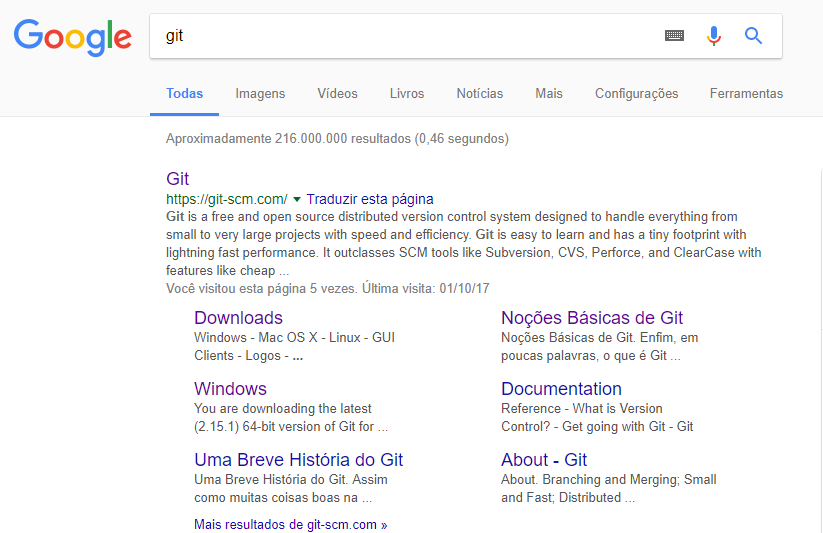
Repositórios remotos ficam em sites como GitHub por exemplo. O propósito de trabalhar em um repositório remoto é estar sempre o atualizando, em conjunto com outros colaboradores, seja no mesmo branch ou em outros. Assim as alterações ou criações que você realizar, estarão disponíveis para os demais colaboradores e vice-versa.



O 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**

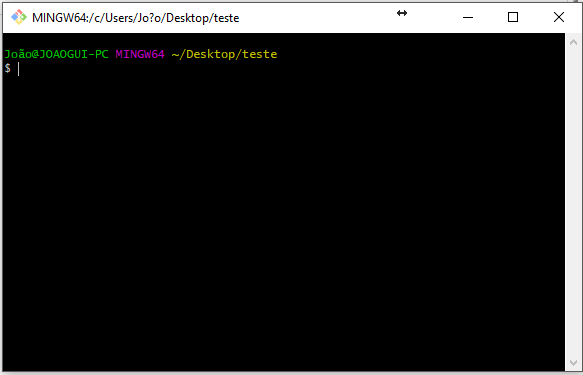
1. Abra seu navegador e pesquise por “GIT” no google.



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



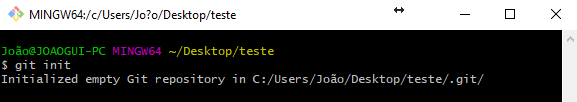
1. Depois de instalado crie uma nova pasta no seu Desktop, dentro dessa pasta clique com o botão direito do mouse e selecione a opção “git bash here” e o terminal do git vai abrir.



**6.3 - Comandos Básicos**

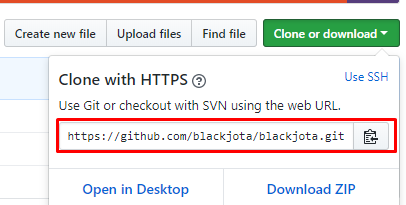
Clonando e criando novos projetos

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

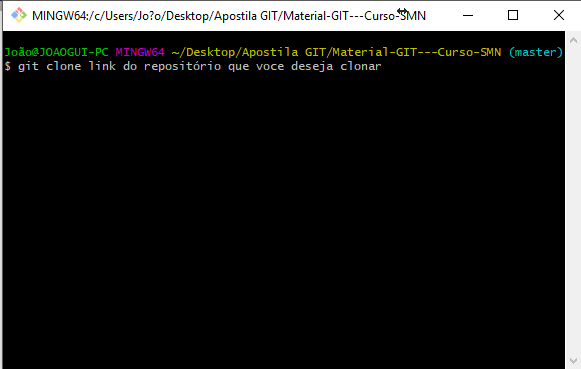


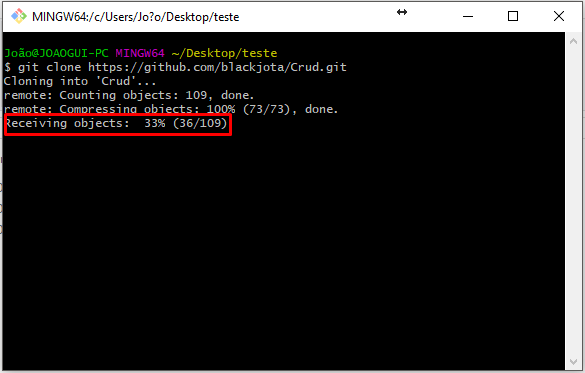
**git clone-** O git clone como o próprio nome já diz, clona o repositório desejado para utilizar.

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.

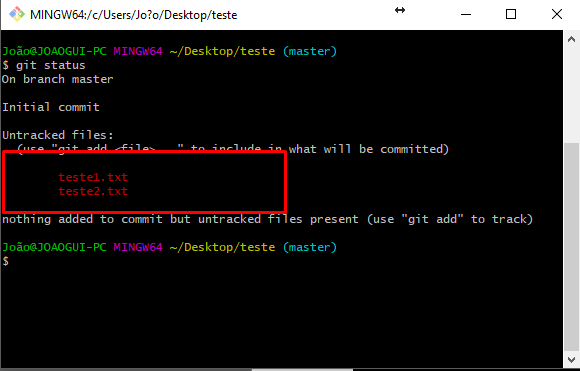




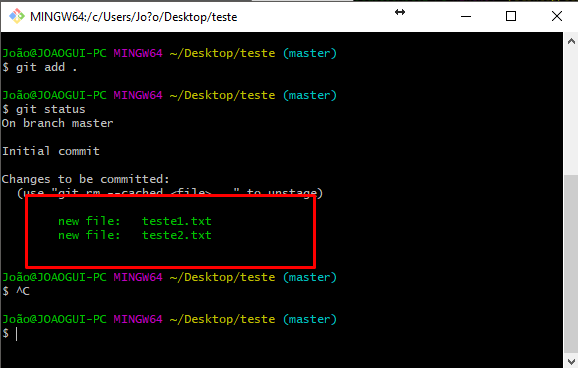
Pronto, você clonou o repositório para a sua máquina.

**6.4 - Commitando e atualizando seus projetos**

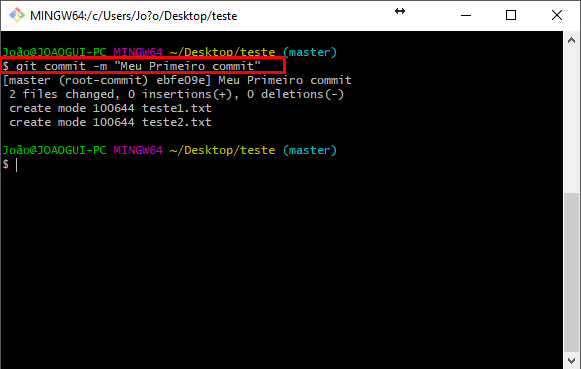
**git status –**  mostra todos os arquivos que foram modificados no seu projeto em vermelho.



**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.



**git commit -m “Descrição” -** Este comando tem a função de salvar suas modificações, e serão inseridas no histórico de commit do projeto para que 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.



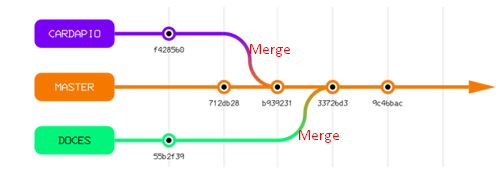
**git push -** O push como o nome em inglês já diz “Empurrar” simplesmente empurra a “caixa” em que você adicionou os arquivos pra commitar para o repositório que você está trabalhando. Lembre-se **Depois de um commit sempre tem um git push, para enviar seu commit para o repositório.**

**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 branchs 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 funcionado. 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**

* Similaridades entre ViewData e ViewBag

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

Uma imagem contendo texto

Descrição gerada com alta confiança

View

Uma imagem contendo interior

Descrição gerada com alta confiança

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**

**Uma imagem contendo captura de tela, texto

Descrição gerada com alta confiança**

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**

Uma imagem contendo parede, interior, monitor, tela

Descrição gerada com alta confiança

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).

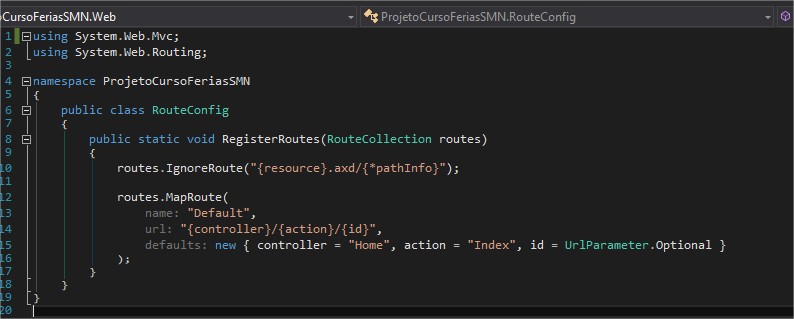
**7.4 - Rotas**

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.

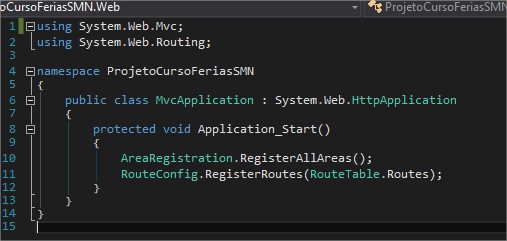


Na imagem acima podemos ver a classe RouteConfig citada acima, ela tem inicialmente a rota padrão nomeada “Default” que diz que as URL’s serão entendidas como:

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.

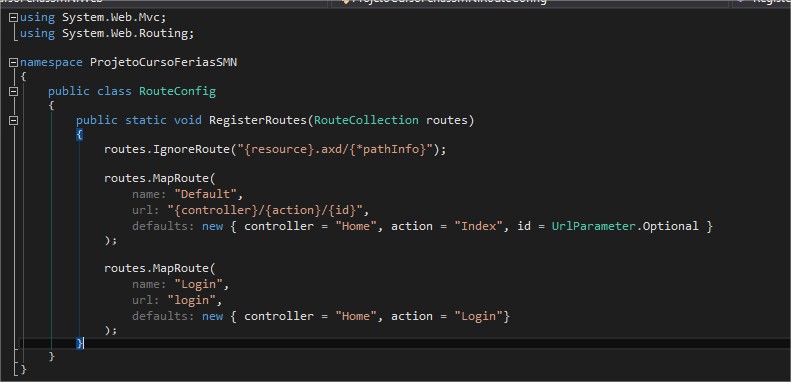
O método estático RegisterRoute, que é definido no arquivo RouteConfig.cs, é chamado a partir do Arquivo Global.asax.cs



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.

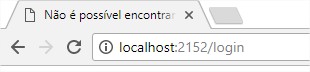
Voltando ao assunto principal, vamos configurar uma nova rota, vamos configurar essa rota para montar uma URL amigável que redireciona o usuário a tela de login.

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



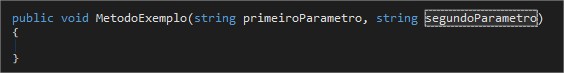
Na imagem, na nova rota que criamos, o parâmetro name define um nome para a nossa rota, cada rota personalizada criada por você deve ter um nome diferente, o segundo parâmetro passado no MapRoute foi a url, nela nós definimos a URL que irá direcionar o usuário a tela de login, por último, no parâmetro 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.

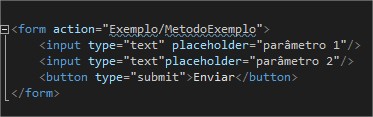


**7.5 - Passando parâmetros aos Métodos**

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:



E o seguinte código HTML:



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



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.

