# Apostila de Algoritmos II

Professor Ms. Eduardo Rosalém Marcelino
eduardormbr@gmail.com

(2009 - 2018)

Bibliografias utilizadas:


- MANZANO, J. A. N. G & OLIVEIRA, J. F. Algoritmos: Lógica para Desenvolvimento de Programação de Computadores. 14 ed. São Paulo:Érica, 2002.
- ROBINSON, Simon, Professional C# Programando – De programador para programador, São Paulo: Pearson Education, 2004.
- DAGHLIAN, Jacob, Lógica e Álgebra de Boole, Ed.Atlas, 4º Ed. - SP, 1995
- Apostila Técnicas de Programação I cedida pela Profa. Dra. Elisamara de Oliveira
- LIMA, EDWIN, C# e .Net para desenvolvedores / Edwin Lima, Eugênio Reis. – Rio de Janeiro: Campus, 2002
- Curso de C# - Módulo I Introdução ao .NET com C# - Carlos Vamberto

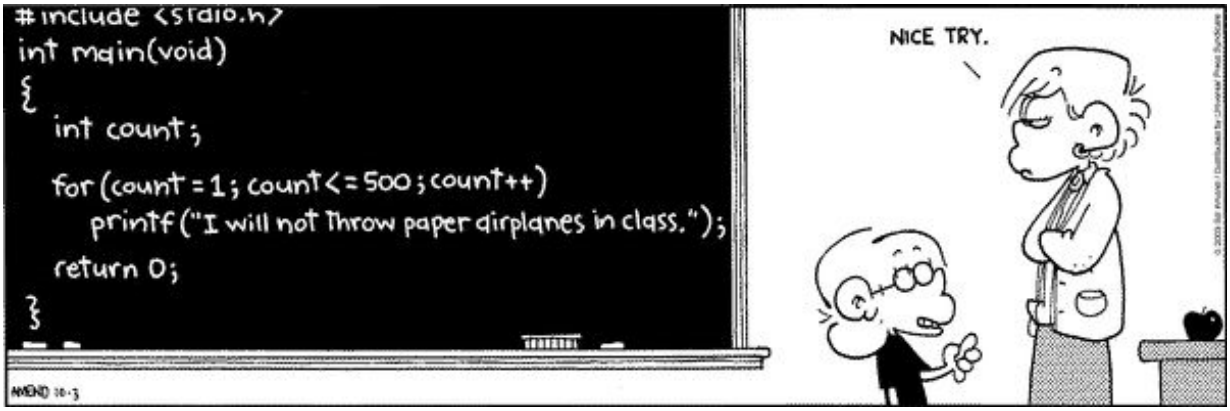

Bibliografias indicadas que possuímos em nosso acervo:


- Anita Lopes; Guto Garcia. Introdução à Programação. 500 algoritmos resolvidos. Elsevier, 2002
- Sandra Puga; Gerson Rissetti. Lógica de programação e estruturas de dados com aplicações em Java. Pearson Prentice Hall, 2009
- Ana Fernanda Gomes Ascencio. Lógica de Programação com Pascal. Makron Books, 1999.
- Victorine Viviane Mizrahi. Treinamento em Linguagem C. Pearson Prentice Hall, 2008.
- MANZANO, J. A. N. G & OLIVEIRA, J. F. Algoritmos: Lógica para Desenvolvimento de Programação de Computadores. 14 ed. São Paulo:Érica, 2002.


**ESCLARECIMENTOS**

Parte desta apostila foi criada utilizando-se como referência livros e apostilas cedidas por outros professores. Qualquer problema, por favor, entre em contato.

Esta apostila foi elaborada com o propósito de servir de apoio ao curso e não pretende ser uma referência completa sobre o assunto. Para aprofundar conhecimentos, sugerimos consultar livros da área.



# Índice

# Métodos

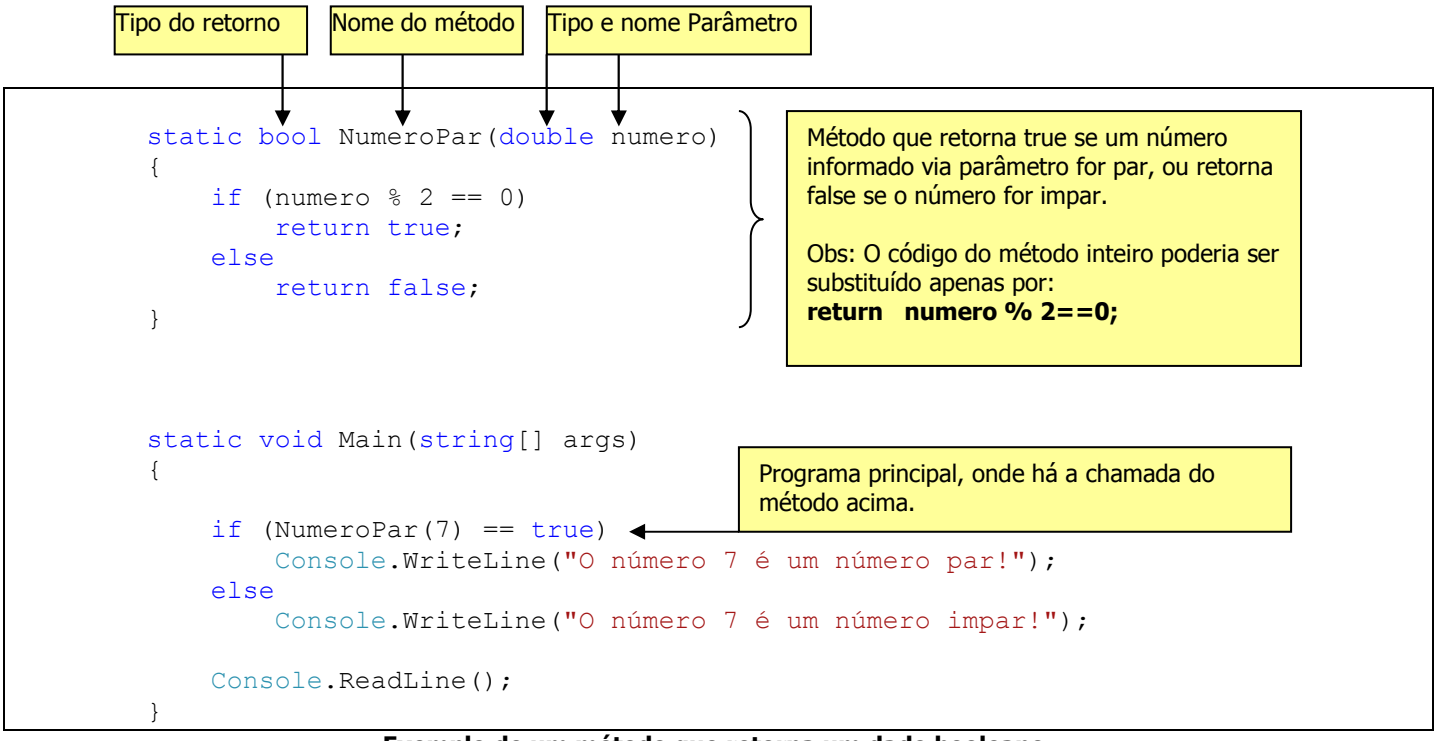
Os métodos em C# são conceitualmente similares a procedimentos e **funções** em outras linguagens de alto nível. Normalmente correspondem a "trechos" de código que podem ser chamados em um objeto específico (de alguma classe). Os métodos podem admitir parâmetros como argumentos, e seu comportamento depende do objeto ao qual pertencem e dos valores passados por qualquer parâmetro. Todo método em C# é especificado no corpo de uma classe. A definição de um método compreende duas partes: a assinatura, que define o nome e os parâmetros do método, e o corpo, que define o que o método realmente faz.

## Declarando Métodos

Em C#, a definição de um método é formada por único modificador de método (como é a acessibilidade do método), pelo tipo do valor de retorno seguido do nome do método, seguido de uma lista de argumentos de entrada incluída entre parênteses, seguida do corpo cio método incluído entre chaves.

```
[modificadores]  tipo_retorno    NomeMetodo( [parâmetros])
{
  Corpo do método
}
```

Cada parâmetro consiste no nome do tipo do parâmetro e no nome pelo qual ele é mencionado no corpo do método. Além disso, se o método retornar um valor, um comando **return** deverá ser usado com o valor de retorno para indicar o ponto de saída. Por exemplo:




**Exemplo de um método que retorna um dado booleano.**

- Se o método não retornar nada, nós especificamos o tipo de retorno como **void**, pois não podemos omitir o tipo de retorno.
- Se não houver nenhum argumento, precisamos também incluir um conjunto vazio de parênteses depois do nome do método. Nesse caso, a inclusão de um comando **return** é opcional - o método retornará automaticamente quando a chave de fechamento for alcançada.
- Você deve observar que um método poderá conter quantos comandos **return** forem necessários.
- Assim que o **return** for executado, o método é terminado.

```
static void ExibeTextoVermelho(string texto)
{
  ConsoleColor  cor = Console.ForegroundColor; // guarda a cor agual na variável cor
  Console.ForegroundColor = ConsoleColor.Red; // altera a cor atual para vermelho
  Console.WriteLine( texto ); // escreve o texto em vermelho
  Console.ForegroundColor = cor; // restaura a cor que estava antes de executar o método.
}


static void Main(string[] args)
{
  ExibeTextoVermelho("Este texto está sendo exibido em vermelho");
  Console.WriteLine("Pressione enter para terminar.");
  Console.ReadLine();
}
```

**Exemplo de um método que não retorna nada.**

```
        static void LimpaTela()
        {
            Console.CursorLeft = 0;
            Console.CursorTop = 0;
            for (int linha = 0; linha <= 23; linha++)
            {
                for (int coluna = 0; coluna <= 78; coluna++)
                    Console.Write(" ");
                Console.WriteLine();
            }
            Console.CursorLeft = 0;
            Console.CursorTop = 0;
        }


        static void Main(string[] args)
        {
            ExibeTextoVermelho("xxxxxxxxxxxxxxxxx");
            Console.WriteLine("yyyyyyyyyyyyyyyyyy");
            Console.WriteLine("Pressione enter para limpar a tela!");
            Console.ReadLine();
            LimpaTela();
            Console.ReadLine();
        }
```

**Exemplo de um método que não retorna nada e não tem parâmetros.**

Sobre a palavra reservada **Static**, não se preocupe neste momento, pois este é um assunto que será visto mais adiante, em orientação a objetos. Um método (ou campo) estático está associado à definição de classe como um todo, não com alguma instância em particular daquela classe. Isso significa que eles serão chamados pela especificação do nome da classe e não do nome da variável.

# Escopo das variáveis

O escopo de uma variável é a região de código na qual a variável pode ser acessada. De forma geral, o escopo é determinado pelas seguintes regras:

1. Um campo (também conhecido como variável membro) de uma classe permanecerá no escopo pelo mesmo tempo em que a classe na qual está contido permanecer no escopo.
2. Uma variável local permanecerá no escopo até que uma chave indique o fim de uma instrução de bloco ou do método no qual ela foi declarada.
3. Uma variável local declarada em uma instrução for, while e ou outra semelhante permanecerá no escopo no corpo daquele laço.
4. Variáveis com o mesmo nome não poderão ser declaradas duas vezes no mesmo escopo.

```
class Program
{
    static int x = 10;

    static void Main(string[] args)
    {
        int y = 7;
        Console.WriteLine(x + y);
        Console.ReadLine();

    }
}
```

A variável "x" tem uma visibilidade "global" dentro da classe "Program", ou seja, ela é "visível" em todos os métodos criados dentro desta classe.
Novamente, não se preocupe com a declaração "static".

A variável "y" tem uma visibilidade "local" e só pode ser utilizada dentro do método Main (onde ela foi criada)

```
.....

if (x == 5)
{
   string nome = "cosmo";
   Console.WriteLine(nome);
}

nome = "wanda"; //vai dar erro de compilação, pois esta variável não existem mais!
```

**Como explicado no item 2, a variável "nome" só terá visibilidade dentro do "if" onde ela foi criada.**

# Controle de Exceção - Básico

Três tipos de erros podem ser encontrados em seus programas, são eles: erros de sintaxe, erros de Runtime e erros lógicos, vamos entender cada um deles.

**Erros de sintaxe ou erro de compilação:**

- Acontece quando você digita de forma errada uma palavra reservada ou comando do C#. Você não consegue executar seu programa quando tem esse tipo de erro no seu código.

**Erros de Runtime:**

- Acontecem quando o programa para de executar der repente durante sua execução, chamamos essa parada de exceção.

- Erros de runtime acontecem quando alguma coisa interfere na correta execução do seu código, por exemplo, quando seu código precisa ler um arquivo que não existe. Neste momento ele gera uma exceção e para bruscamente a execução. Esse tipo de erro pode e deve ser tratado.

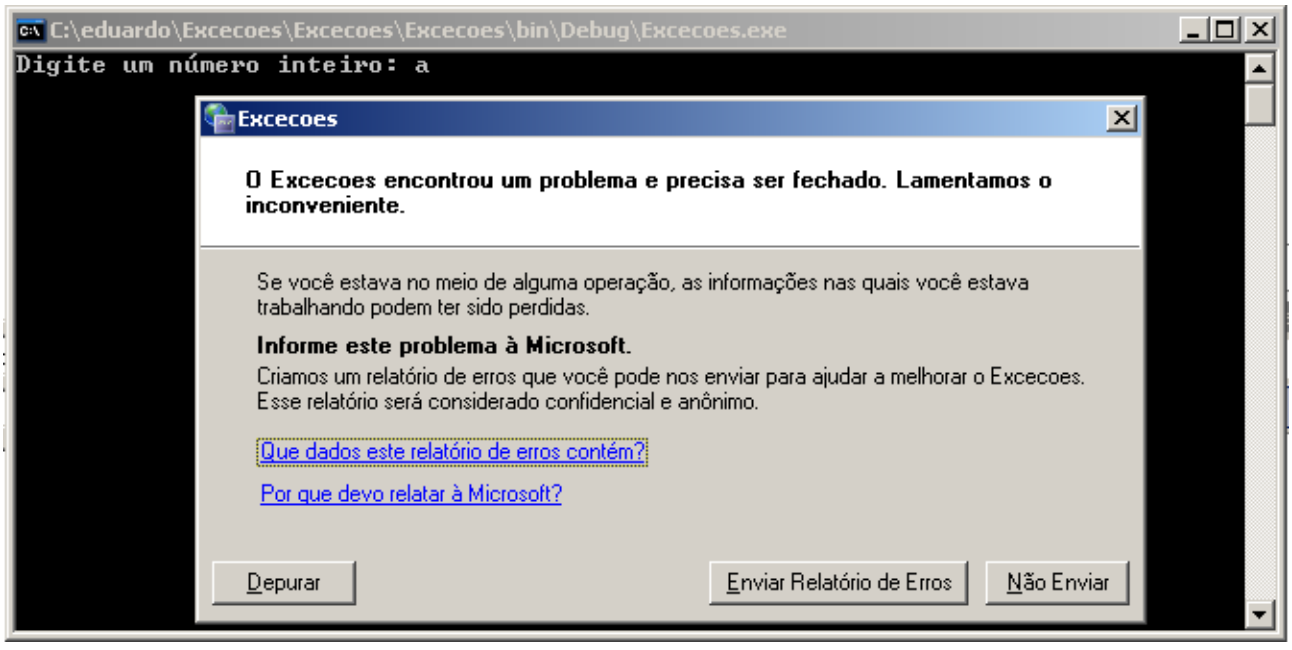
**Erros lógicos:**

- Esse é o tipo de erro mais difícil de ser tratado. É um erro humano. O código funciona perfeitamente, mas o resultado é errado. Exemplo, uma função que deve retornar um valor, só que o valor retornado está errado, o erro neste caso se encontra na lógica da função que está processando o cálculo. A grosso modo é como se o seu programa precise fazer um cálculo de 2 + 2 em que o resultado certo é 4 mas ele retorna 3. Quando é uma conta simples é fácil de identificar mas e se o cálculo for complexo.

O tratamento de exceção é um mecanismo capaz de dar robustez a uma aplicação, permitindo que os erros sejam manipulados de uma maneira consistente e fazendo com que a aplicação possa se recuperar de erros, se possível, ou finalizar a execução quando necessário, sem perda de dados ou recursos.

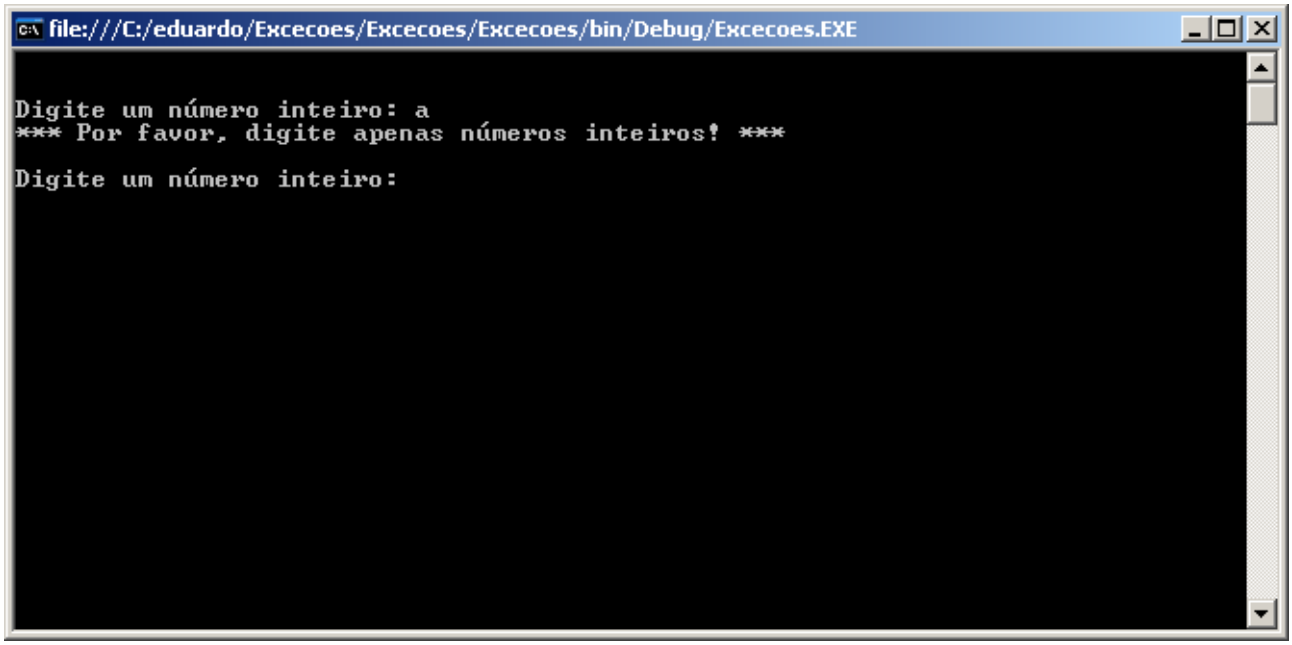
Para que uma aplicação seja segura, seu código necessita reconhecer uma exceção quando ela ocorrer e responder adequadamente a esta. Se não houver tratamento consistente para uma exceção, será exibida uma mensagem padrão descrevendo o erro e todos os processamentos pendentes não serão executados. Uma exceção deve ser respondida sempre que houver perigo de perda de dados ou de recursos do sistema.

O exemplo abaixo ilustra uma exceção que ocorreu em um programa que esperava uma entrada de um valor inteiro, mas foi digitada uma letra:



O ideal seria o tratamento destes erros, evitando a perda de dados ou a necessidade de encerrar a aplicação. Além de tratar o erro, a rotina de tratamento de erros poderia enviar ao usuário uma mensagem em português, mais significativa. A forma mais simples para responder a uma exceção é garantir que algum código limpo é executado. Este tipo de resposta não corrige o erro, mas garante que sua aplicação não termine de forma instável. Normalmente, usa-se este tipo de resposta para garantir a liberação de recursos alocados, mesmo que ocorra um erro.

O tratamento mais simples seria uma simples mensagem ao usuário com a proposta para ele tentar efetuar novamente a operação que tenha causado o erro, conforme podemos ver no exemplo abaixo:




## BLOCOS PROTEGIDOS

Bloco protegido é uma área em seu código que está "protegido" de exceções. Se o código não gerar nenhuma exceção, ele prossegue com o programa. Caso ocorra uma exceção, então ele cria uma resposta a este insucesso.

Quando se define um bloco protegido, especifica-se respostas a exceções que podem ocorrer dentro deste bloco. Se a exceção ocorrer, o fluxo do programa pula para a resposta definida, e após executá-la, abandona o bloco. Um bloco protegido é um grupo de comandos com uma seção de tratamento de exceções.

O exemplo abaixo verifica se foi digitado apenas números. Caso seja digitado qualquer outro caractere, uma mensagem é exibida:

```
try
{

      Console.Write("\n\nDigite um número inteiro: ");
      numero = Convert.ToInt32(Console.ReadLine());
}
catch
{
      Console.Write("Digite apenas números inteiros!");
}
```

O comando `Try{ }` define o bloco protegido.

Se alguma exceção ocorrer ali, o fluxo de execução é transferido para o bloco `catch { }`

O fluxo de execução **só** será transferido para o `catch` se ocorrer uma exceção no bloco `try`.

O exemplo abaixo verifica se o usuário digitou um número inteiro válido e, caso não o tenha feito, o programa ficará solicitando o número até que o usuário o informe corretamente.

```
static void Main(string[] args)
{
	int numero;
	bool correto;

	do
	{
		try
		{
			Console.Write("\n\nDigite um número inteiro: ");
			numero = Convert.ToInt32(Console.ReadLine());
			correto = true;
		}
		catch
		{
			correto = false;
			Console.Write("*** Digite apenas números inteiros! *** ");
		}
	}
	while (correto == false);
}
```

Caso o usuário digite uma letra, a linha **correto = true;** não será executada pois a exceção fará o fluxo ser direcionado para o bloco **catch**.

# Estruturas de dados heterogêneas

http://msdn.microsoft.com/pt-br/library/vstudio/ah19swz4.aspx
http://msdn.microsoft.com/pt-br/library/vstudio/ms173109%28v=vs.100%29.aspx

Os tipos de variáveis que apresentamos até agora (ex: int, float, Double, string, etc.) permitem armazenar apenas 1 tipo de dado. Um a estrutura de dados heterogênea é um tipo de dado criado pelo usuário. Este novo tipo de dado pode ser utilizado para armazenar, em uma única variável, tipos de dados diferentes. Em C#, o tipo **struct** é utilizado para criar um novo tipo de dado, definido pelo usuário.

Um tipo **struct** é um tipo de valor normalmente usado para encapsular pequenos grupos de variáveis relacionadas, como as coordenadas de um retângulo ou as características de um item em um inventário.

Imagine ter que desenvolver um sistema acadêmico. Neste sistema deverão ser armazenados dados de alunos, professores, disciplinas, notas, etc. Neste exemplo, existirão diversas variáveis que terão nomes muito parecidos.

Por exemplo, se utilizar a variável **código** para definir o código do aluno, que nome de variável utilizar para definir o código do professor? E da disciplina? E do curso? Da Turma? Poderíamos fazer da seguinte maneira:

```
int codigo_curso, codigo_aluno, codigo_disciplina, codigo_turma;
```

O problema será gerenciar este monte de variáveis com nomes parecidos, fora que corremos o risco de algum dia ser criada, por exemplo, uma variável **endereco** e neste caso ficaremos sem saber de quem é o endereço (aluno? Professor?).

Uma das vantagens das estruturas (**structs**) é justamente agrupar informações que são correlacionadas.

Vamos resolver o problema acima, mapeando as variáveis de aluno, professor e disciplina em estruturas:

```
struct Aluno
{
    public int codigo;
    public string nome;
    public string cpf;
    public double mensalidade;
}


struct Professor
{
    public int codigo;
    public string nome;
    public string cpf;
}


struct Disciplina
{
    public int codigo;
    public string nome;
}
```

Para utilizar as estruturas, primeiro precisamos criar uma variável para depois preencher os seus campos!

```
Aluno aluno1 = new Aluno();
aluno1.codigo = 7;
aluno1.nome = "Cosmo";
aluno1.cpf = "123.456.789-09";
aluno1.mensalidade = 756.33;

Aluno aluno2 = new Aluno();
aluno2.codigo = 20;
aluno2.nome = "Wanda";
aluno2.cpf = "223.457.721-33";
aluno2.mensalidade = 800.00;

Professor p1 = new Professor();
p1.codigo = 1;
p1.nome = "Girafales";
p1.cpf = "777.777.777-77";
```

O exemplo abaixo cria um tipo de dado chamado **Pessoa**. Este novo tipo de dado é composto por 2 campos. Um campo chamado **codigo** do tipo inteiro e um campo chamado **nome** do tipo string.

```
namespace Estrut_heterogeneas_1
{
    class Estruturas_heterogeneas
    {
        // Estrura para armazenar dados heterogêneos.
        struct Pessoa
        {
            public int codigo;
            public string nome;
        }

        static void Main(string[] args)
        {
            Pessoa p1 = new Pessoa();

            p1.codigo = 5;
            p1.nome = "Daniela da Silva";

            Console.WriteLine("{0}  - {1}", p1.codigo, p1.nome);

            Console.ReadLine();
        }
    }
}
```

A declaração do novo tipo deve ser fora do método **main**.

**Public** indica a visibilidade da estrutura dentro do programa

Os campos de uma estrutura são acessados após o ".".

Ex: dado.codigo = 5

Observe no programa acima que foi criado um novo tipo de dado, o tipo **Pessoa**. As variáveis criadas a partir deste tipo irão possuir 2 campos (codigo e nome).

Esse novo tipo de dado pode ser utilizado também para criar vetores. Sendo assim, é possível guardar mais de uma informação em cada célula de um vetor. Ex:

```
namespace Estrut_heterogeneas_1
{
    class Estruturas_heterogeneas
    {
        // Estrutura para armazenar dados heterogêneos no vetor.
        struct Pessoa
        {
            public int    codigo;
            public string nome;
        }

        static void Main(string[] args)
        {

            Pessoa[] Vetor = new Pessoa[5];


            // leitura dos dados do vetor
            for (int i = 0; i < Vetor.Length; i++)
            {
                Vetor[i] = new Pessoa();


                do // validação do código
                {
                    Console.WriteLine("Informe um código maior que zero.");
                    Vetor[i].codigo =Convert.ToInt16( Console.ReadLine());
                }
                while (Vetor[i].codigo <= 0);


                do // validação do nome
                {
                    Console.WriteLine("Agora informe o nome");
                    Vetor[i].nome = Console.ReadLine();
                }
                while (Vetor[i].nome.Trim().Length == 0);
            }


            // impressão em vídeo dos dados lidos
            Console.WriteLine("\n\nDADOS CADASTRADOS:\n");
            for (int i = 0; i < Vetor.Length; i++)
            {
              Console.WriteLine("Cód.: {0} - Nome: {1}", Vetor[i].codigo, Vetor[i].nome);
            }

            Console.ReadLine();
        }
    }
}
```

No vetor, as informações serão armazenadas da seguinte maneira:

| | | |
|---|---|---|
| 0 | Código:<br>Nome: | 1<br>Ana |
| 1 | Código:<br>Nome: | 2<br>Daniela |
| 2 | Código:<br>Nome: | 3<br>Cláudia |
| 3 | Código:<br>Nome: | 4<br>Bruna |
| 4 | Código:<br>Nome: | 5<br>Paula |

Observe que cada célula possui 2 campos.

# Passagem de Parâmetros por Valor e por Referência

Geralmente, os argumentos (as informações entre parênteses na declaração do método) podem ser passados aos métodos por **referência** ou por **valor**.

- Uma variável passada por **referência** a um método será afetada por quaisquer alterações que o método chamado fizer nela.
- Uma variável passada por **valor** para um método não será alterada pelas alterações que ocorrerem no corpo do método. Isso ocorre porque o método se refere às variáveis originais quando elas são passadas por referência, mas apenas às cópias das variáveis quando elas são passadas por valor.

## A Palavra-Chave ref

Para tipos de dado mais complexos, a passagem por referência é mais eficiente em decorrência da grande quantidade de dados que devem ser copiados quando se passa por valor.

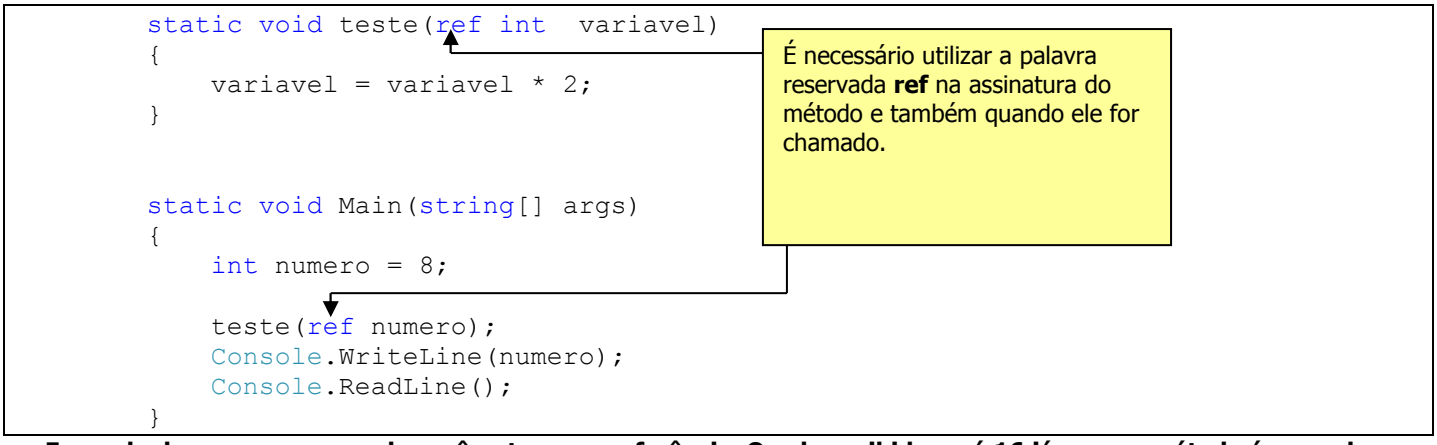
No C#, todos os parâmetros são passados por valor, a menos que solicitemos especificamente que isso não seja feito. No entanto, o tipo de dado do parâmetro também determinará o efetivo comportamento de quaisquer parâmetros que sejam passados a um método. Como os tipos referência contêm apenas uma referência ao objeto, eles ainda passarão apenas essa referência para o método. Os tipos valor, ao contrário, contêm realmente o dado, de forma que uma cópia do próprio dado será passada para o método.

- Um **int**, por exemplo, é passado por valor para uma método, e quaisquer alterações que esse método fizer no valor desse **int** não alterará o valor do objeto **int** original.
- Inversamente, se um **array** ou qualquer tipo referência, como uma classe, for passado para um método e o método alterar um valor naquele **array**, o novo valor será refletido no objeto **array** original.

```
static void teste(int variavel)
{
    variavel = variavel * 2;
}


static void Main(string[] args)
{
    int numero = 8;
    teste(numero);
    Console.WriteLine(numero);
    Console.ReadLine();
}
```

**Exemplo de uma passagem de parâmetros por valor. O valor exibido será 8 já que seu conteúdo será copiado no método teste.**

Esse comportamento é padrão. Porém, nós podemos fazer com que parâmetros de valor sejam passados por referência. Para fazer isso, usamos a palavra-chave ref. Se um parâmetro for passado a um método e o argumento de entrada para aquele método for anteposto com a palavra-chave **ref**, qualquer alteração que o método faça na variável afetará o valor do objeto original:




```
static void teste(ref int  variavel)
{
    variavel = variavel * 2;
}


static void Main(string[] args)
{
    int numero = 8;

    teste(ref numero);
    Console.WriteLine(numero);
    Console.ReadLine();
}
```

**Exemplo de uma passagem de parâmetros por referência. O valor exibido será 16 já que no método é passado o endereço da variável numero, e não o seu valor.**

**Obs**: Se for utilizada a passagem por referência, a chamada do método não poderá ser feita com **literais** ou **constantes**. Ex:

```
teste( 9 );
```

O código acima está incorreto pois a declaração do método usa a palavra **ref**, que EXIGE a passagem de um valor por referência, ou seja, um endereço de uma variável.

A linguagem C# torna o comportamento mais explícito (evitando assim, supõe-se, os bugs) ao requerer o uso da palavra-chave **ref** quando um método é invocado.

**Obs**: Qualquer variável deverá ser inicializada antes de ser passada para um método, quer ela seja passada por valor ou por referência.

## A Palavra-Chave out

Em linguagens C#, é comum as funções poderem retornar mais de um valor em uma simples rotina. Isso é obtido por meio dos parâmetros de saída — pela atribuição de valores de saída a variáveis que foram passadas ao método por referência. Muitas vezes, os valores iniciais das vaiáveis passadas por referência não têm importância. Esses valores serão sobrescritos pela função, que pode até nunca chegar a examiná-los.
Seria conveniente se pudéssemos usar a mesma convenção em C#, mas, como você deve se lembrar, o C# requer que as variáveis sejam inicializadas com algum valor antes se serem referenciadas. Embora pudéssemos inicializar nossas variáveis de entrada com valores insignificantes antes de sua passagem dentro da função, o que dará a elas o valor real, essa prática parece desnecessária, e pior, confusa.  Contudo, há um meio de acabar com a insistência dos compiladores c# sobre os valores iniciais dos argumentos de entrada.

É por meio da palavra chave **out**. Quando o argumento de entrada de um método é anteposto com a palavra chave **out**, esse método pode receber uma variável que não foi inicializada de forma alguma. A variável é passada por referência, assim qualquer mudança que o método faz na variável persistirá quando o controle retornar ao método chamado.

```
        static void MaiorMenor(int[] vetor, out int maior, out int menor)
        {
            int i;
            maior = vetor[0];
            menor = vetor[0];
            for (i = 1; i < vetor.Length; i++)
            {
                if (vetor[i] > maior)
                    maior = vetor[i];
                else if (vetor[i] < menor)
                    menor = vetor[i];
            }
        }


        static void Main(string[] args)
        {
            int[] vetor = new int[3];
            int maior, menor;

            vetor[0] = 7;
            vetor[1] = 3;
            vetor[2] = 5;

            MaiorMenor(vetor, out maior, out menor);

            Console.WriteLine("Maior valor: {0} \nMenor valor: {1}", maior, menor) ;
            Console.ReadLine();
        }
```

**Exemplo de um método que retorna dados nos parâmetros "maior" e "menor".**

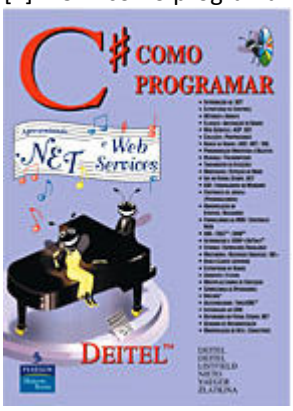
```
static void Main(string[] args)
{
   Console.Write("Digite um número: ");
   string numeroStr = Console.ReadLine();

   int numeroInt;
   if (int.TryParse(numeroStr, out numeroInt))
       Console.WriteLine("Consegui converter para inteiro. Valor Convertido: {0}", numeroInt);
   else
       Console.WriteLine("Não consegui converter {0} para inteiro.", numeroStr);

   Console.ReadLine();
}
```

**Exemplo de um método tenta converter para inteiro um valor digitado pelo usuário.**

# Programação Orientada a Objetos

- Classes e Objetos
- Atributos
- Propriedades
- Métodos
- Encapsulamento

Vide material na biblioteca virtual nos seguintes livros:

<table>
<tr>
<td>[1] – Microsoft Visual C# 2005 – Passo a Passo – John Sharp<br> 
</td>
<td>[2] – C# - como programar - Deitel<br> 
</td>
<td>[3] – Aprenda programação orientada a objetos em 21 dias – Anthony Sintes<br> 
</td>
</tr>
<tr>
<td>[4] – Java - como programar - Deitel<br> 
</td>
<td>[5] – Fundamentos do Desenho Orientado a Objeto com Uml - Page-Jones, Meillir<br> 
</td>
<td></td>
</tr>
</table>

Um resumo destes livros pode ser encontrado no material “Referências POO.PDF”
Páginas (**vide páginas indicadas no leitor de arquivos PDF e não as páginas indicadas no documento**)

Páginas 4 a 15 (classe, objeto, abstração, atributos, métodos)
Páginas 23 a 32 (modificadores de acesso, encapsulamento, getters e setters, propriedades)
Página 37 (Referências, Tipos de Valor e Tipos de Referência)
Páginas 41 a 42 Métodos e Classes Estáticos, enumeradores