

# Apostila de Lógica de Programação em C#

Professor Ms. Eduardo Rosalém Marcelino

[eduardormbr@gmail.com](mailto:eduardormbr@gmail.com)

(2009 - 2016)

## 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

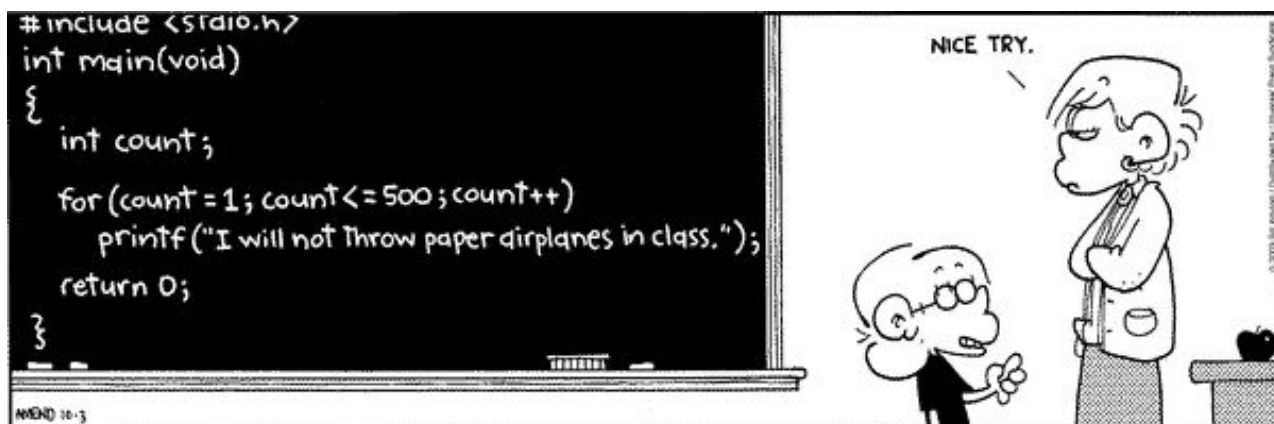
## Bibliografias indicadas que possuímos em nosso acervo:

- 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.

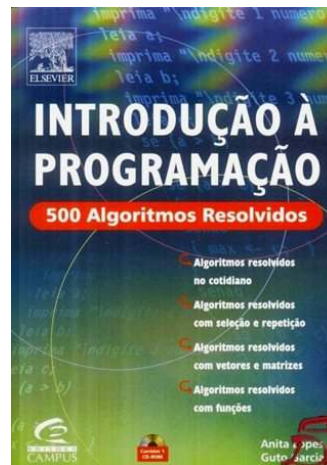


## Índice

Bem vindo ao jogo! .....	4
Álgebra Booleana .....	6
Lógica digital .....	6
Álgebra booleana .....	6
Proposições .....	7
Operação AND (conjunção) .....	7
Operação OR (disjunção inclusiva) .....	8
Operação NOT (negação) .....	8
Operação XOR (disjunção exclusiva) .....	9
O Conceito de Algoritmo .....	11
O que é um algoritmo? .....	11
Conceitos Básicos da Programação de Computadores .....	13
Gerações das linguagens .....	16
Linguagem de programação de primeira geração .....	16
Linguagem de programação de segunda geração .....	16
Linguagem de programação de terceira geração .....	17
Linguagem de programação de quarta geração .....	17
Linguagens Compiladas, Linguagens Interpretadas e Modelo Híbrido .....	18
Interpretadas .....	18
Compiladas .....	18
Híbridas .....	19
<b>Expressão de Algoritmos</b> .....	23
Expressão de Algoritmos através de Diagramas e Fluxogramas .....	23
Exemplo de um Fluxograma .....	24
Alguns símbolos utilizados em diagramas de bloco e fluxogramas: .....	25
Expressão de Algoritmos através de Pseudolinguagem .....	25
Expressão de Algoritmos através de Linguagem de Programação .....	26
Fluxogramas .....	29
Programação em C# .....	30
O que é .NET .....	30
Compilando programas .NET: introduzindo a linguagem intermediária MSIL (Microsoft Intermediate Language) .....	30
Gerenciamento da memória: introduzindo o GC (Garbage Collector) .....	31
Linguagens que suportam .NET .....	31
Principais vantagens da linguagem C# .....	32
Quando usar a .NET? .....	32
Configuração do Visual Studio .....	34
Principais teclas de atalho .....	34

	3
Estrutura básica de um programa em C#.....	35
Interagindo com o console.....	36
Principais Operadores.....	37
Variáveis.....	40
Convenção PascalCasing.....	40
Convenção camelCasing.....	40
Palavras reservadas:.....	40
Declarando variáveis.....	41
Atribuindo valor a variáveis.....	41
Tipos de variáveis.....	42
Manipulando valores numéricos.....	44
Conversões de tipos.....	45
Adicionando valor a uma variável.....	47
Diferença entre ++VAR e VAR++.....	48
Tipo de dado String.....	49
Concatenar.....	49
Tipo de dado Char.....	49
Propriedades e Métodos das Strings.....	50
Propriedade Length.....	50
Método Substring.....	50
Método IndexOf.....	51
Método Replace.....	51
Método Remove.....	52
Console.ReadLine().....	52
Exemplo completo:.....	52
Manipulando arquivos texto – parte 1.....	54
Escrevendo no arquivo.....	54
Lendo os dados do Arquivo texto.....	56
Constantes.....	57
Saindo explicitamente da aplicação:.....	57
Estrutura de Decisão (IF).....	58
Indentação.....	61
Depuração de Programas.....	63
Estrutura de Repetição FOR.....	64
Laços for infinitos.....	65
Laços for aninhados.....	65
Laços for em decremento.....	66
Estrutura de Repetição DO WHILE.....	66
Estrutura de Repetição WHILE.....	67
Vetores e Matrizes.....	68
Estrutura de decisão (SWITCH).....	70
Números randômicos.....	71
Estrutura de repetição foreach/in.....	72
Recebendo parâmetros na linha de comando.....	73
Estruturas de dados heterogêneas.....	74
Métodos.....	77
Declarando Métodos.....	77
Escopo das variáveis.....	79
Passando Parâmetros para Métodos por Valor e por Referência.....	80
A Palavra-Chave out.....	81
Controle de Exceção - Básico.....	83

# Bem vindo ao jogo!



## Conceitos iniciais

- **Lógica de programação** é a técnica de encadear pensamentos para atingir determinado objetivo. O aprendizado desta técnica é necessário para quem deseja trabalhar com desenvolvimento de sistemas e programas.
- **Algoritmo** é uma sequência de passos finitos com o objetivo de solucionar um problema.

Quando nós temos um problema, nosso objetivo é solucioná-lo.

Algoritmo não é a solução de um problema, pois, se assim fosse, cada problema teria um único algoritmo. Algoritmo é um conjunto de passos (ações) que levam à solução de um determinado problema, ou então é um caminho para a solução de um problema e, em geral, os caminhos que levam a uma solução são muitos.



O aprendizado de algoritmos não é uma tarefa muito fácil, só se consegue através de muitos exercícios. Este é o objetivo principal deste livro: possibilitar que você, a partir das soluções apresentadas, venha construir sua própria lógica de programação.



A maioria dos tratamentos só faz efeito se você ingerir o medicamento de forma sistematizada e contínua durante um período de tempo.

Tomar o medicamento todo **no último dia** do tratamento **não vai te curar**.

# Álgebra Booleana

Fonte adicional: <http://www.forumpcs.com.br/coluna.php?b=120830>

## Lógica digital

Todo o raciocínio lógico é baseado na tomada de uma decisão a partir do cumprimento de determinadas condições. Inicialmente tem-se os dados de entrada e uma condição (ou uma combinação de condições). Aplica-se a condição aos dados de entrada para decidir quais são os dados de saída.

A lógica digital não é diferente. Mas apresenta uma peculiaridade: trabalha apenas com variáveis cujos valores alternam exclusivamente entre dois estados e não admitem valores intermediários. Estes estados podem ser representados por "um" e "zero", "sim" e "não", "verdadeiro" e "falso" ou quaisquer outras grandezas cujo valor possa assumir apenas um dentre dois estados possíveis. Portanto, a lógica digital é a ferramenta ideal para trabalhar com grandezas cujos valores são expressos no sistema binário.

Em um computador, todas as operações são feitas a partir de tomadas de decisões que, por mais complexas que sejam, nada mais são que combinações das operações lógicas que veremos a seguir. Para tomadas de decisões mais complexas, tudo o que é preciso é combinar estas operações. E para isto é necessário um conjunto de ferramentas capaz de manejar variáveis lógicas. Esse conjunto de ferramentas é a chamada "Álgebra Booleana".

## Álgebra booleana

A álgebra booleana recebeu seu nome em homenagem ao matemático inglês George Boole, que a concebeu e publicou suas bases em 1854, em um trabalho intitulado "An Investigation of the Laws of Thought on Which to Found the Mathematical Theories of Logic and Probabilities". O trabalho, evidentemente, nada tinha a ver com computadores digitais, já que foi publicado quase um século antes que eles fossem inventados. Era meramente um tratado sobre lógica, um dos muitos exemplos em que os matemáticos se adiantam ao tempo e criam com décadas de avanço as bases abstratas para uma tecnologia de ponta que só vai ser "descoberta" muitos anos depois. De fato, foi somente em 1938 que Claude Shannon, um pesquisador do MIT, se deu conta que a lógica booleana era a ferramenta ideal para analisar circuitos elétricos baseados em relés, os antecessores imediatos dos computadores eletrônicos digitais à válvula – que por sua vez originaram os modernos computadores que empregam a eletrônica do estado sólido.

Não cabe aqui um estudo aprofundado da álgebra booleana. Por isso abordaremos apenas os conceitos fundamentais que nos permitirão mais tarde entender como eles serão utilizados internamente nos computadores. Mas para quem quiser se aprofundar no assunto, há farto material disponível tanto na literatura técnica especializada quanto na Internet. Aqui, repito, ficaremos apenas nos conceitos mais gerais.

A álgebra booleana é semelhante à álgebra convencional que conhecemos no curso secundário, o ramo da matemática que estuda as relações entre grandezas examinando as leis que regulam as operações e processos formais independentemente dos valores das grandezas, representadas por "letras" ou símbolos abstratos. A particularidade da álgebra booleana é que ela estuda relações entre variáveis lógicas que podem assumir apenas um dentre dois estados opostos, "verdadeiro" ou "falso", não admitindo nenhum valor intermediário.

Da mesma forma que a álgebra convencional, a álgebra booleana utiliza operações que são executadas com suas variáveis. A diferença é que estas operações somente podem agir sobre variáveis lógicas, portanto são operações lógicas.

As razões pelas quais a álgebra booleana é a ferramenta ideal para analisar problemas de lógica digital tornam-se evidentes assim que se tomam conhecimento de suas operações.

Da mesma forma que há apenas quatro operações fundamentais na aritmética, há apenas três operações fundamentais na álgebra booleana. Estas operações são **AND**, **OR** e **NOT**.

## Proposições

É uma sentença declarativa, afirmativa e que deve exprimir um pensamento de sentido completo, podendo ser escrita na forma simbólica ou na linguagem usual. Ex:

a)  $5 > 7$

b) O Canadá fica na América do Norte.

Dizemos que o valor lógico de uma proposição é verdade se a proposição é verdadeira; e falsidade se a proposição é falsa.

As proposições podem ser simples ou compostas. Proposição simples é a que não contém nenhuma outra proposição como parte integrante de si mesma. Geralmente são indicadas por letras minúsculas.

Ex:

a: Carlos é careca.

b: Corinthians é o melhor time do planeta.

A proposição composta é formada por duas ou mais proposições relacionadas pelos conectivos E, OU, XOR, etc., e geralmente são indicadas por letras maiúsculas.

a: O Brasil é lindo

b: O Brasil é o país do futebol

Ex:  $P = a \text{ E } b$  ou "O Brasil é lindo E O Brasil é o país do futebol."

Em algumas partes desta apostila, as proposições poderão também ser tratadas como variáveis.

## Operação AND (conjunção)

Operação AND pode ser aplicada a duas ou mais variáveis (ou proposições) que podem assumir apenas os valores "verdadeiro" ou "falso". A operação AND resulta "verdadeiro" se e apenas se os valores de ambas as variáveis (ex A e B) assumirem o valor "verdadeiro".

Tabela Verdade – AND (também conhecido por " $\wedge$ " "E" ".")

A	B	A AND B
V	V	V
V	F	F
F	V	F
F	F	F

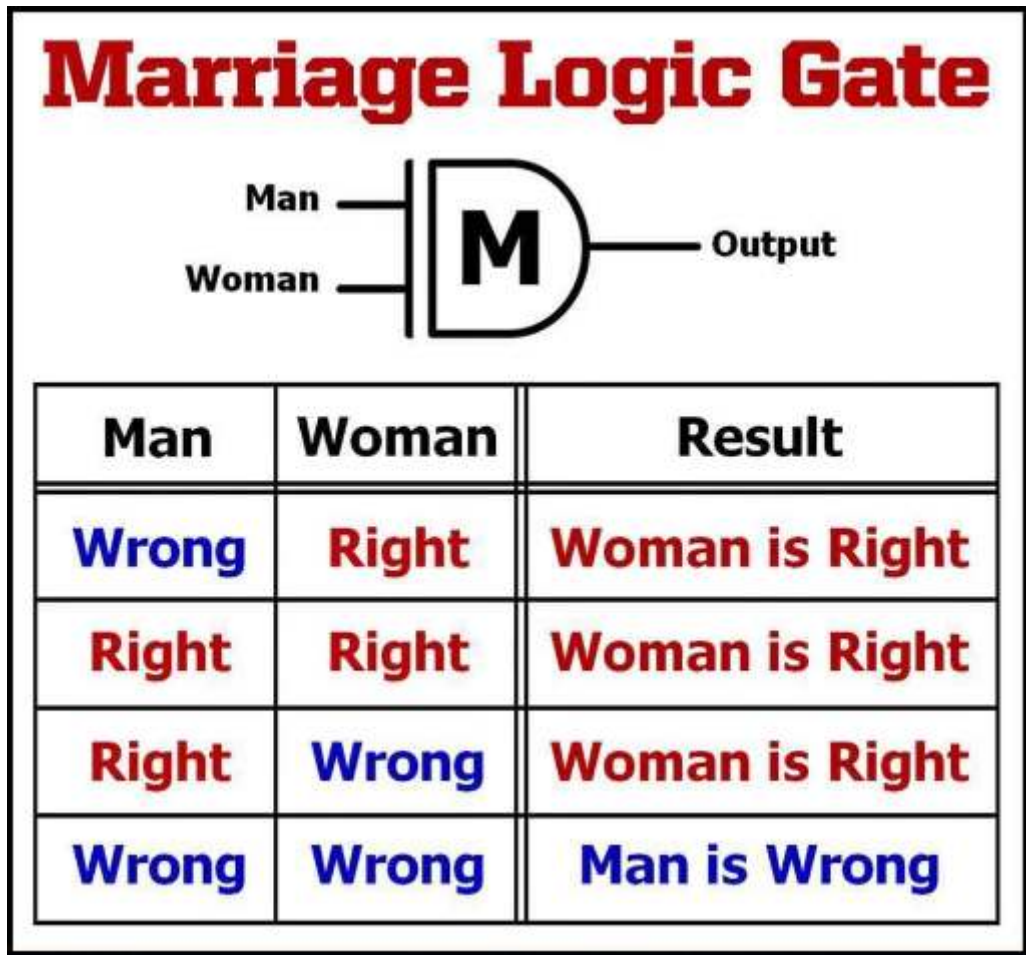
Onde V=verdadeiro e F=Falso e A e B são duas variáveis.

Ex: Se houver sol **e** se for feriado eu vou à praia.

Na frase acima, existem 2 variáveis, que são "se houver sol" e "se for feriado". Para que o sujeito vá à praia, **as duas** situações devem ocorrer, ou seja, devem ser verdadeiras (V). Observe que na frase há o operador AND (ou E).

Variável "Se houver sol" (A)	Variável "Se for feriado" (B)	Resultado "vou à praia" (A AND B)
V	V	V
V	F	F
F	V	F
F	F	F





## Operação OR (disjunção inclusiva)

Operação OR também pode ser aplicada a duas ou mais variáveis (que podem assumir apenas os valores "verdadeiro" ou "falso"). A operação OR resulta "verdadeiro" se o valor de qualquer uma das variáveis A ou B assumir o valor "verdadeiro".

Tabela Verdade – OR (também conhecido por "v" "OU" "+")

A	B	A OR B
V	V	V
V	F	V
F	V	V
F	F	F

Onde V=verdadeiro e F=Falso e A e B são duas variáveis.

Ex: Se houver sol **ou** se for feriado eu vou à praia.

Na frase acima, existem 2 variáveis, que são "se houver sol" e "se for feriado". Para que o sujeito vá à praia, **pelo menos uma** das duas situações deve ocorrer, ou seja, deve ser verdadeira (V). Observe que na frase há o operador OR ( OU ).

## Operação NOT (negação)

A operação NOT é unária, ou seja, aplicável a uma única variável. A operação NOT inverte o valor da variável. Ela resulta "verdadeiro" se a variável assume o valor "falso" e resulta "falso" se a variável assume o valor "verdadeiro". Pode ser representado também por "~".



Ex: seja uma variável A = "Vou à praia".

NOT A = Não vou à praia.

Se A = Verdadeiro, então NOT A = Falso.

## Operação XOR (disjunção exclusiva)

A operação, XOR ou "OR exclusivo" é um caso particular da função OR. Ela é expressa por:

A XOR B.

A operação XOR resulta "verdadeiro" se e apenas se exclusivamente uma das variáveis A ou B assumir o valor "verdadeiro" (uma outra forma, talvez mais simples, de exprimir a mesma idéia é: a operação XOR resulta "verdadeiro" quando os valores da variáveis A e B forem diferentes entre si e resulta "falso" quando forem iguais).

Tabela Verdade – XOR

A	B	A XOR B
V	V	F
V	F	V
F	V	V
F	F	F

Onde V=verdadeiro e F=Falso e A e B são duas variáveis.

Ex: Para você chegar à saída, você **deve** escolher entre o caminho à direita ou o caminho à esquerda.

Na frase acima, existem 2 variáveis, que são "caminho à direita" e "caminho à esquerda". Para chegar à saída, apenas uma das variáveis deve ser verdadeira. Se ambas forem verdadeiras ou se ambas forem falsas, o resultado final será falso.

## Expressões Algébricas

As regras básicas da álgebra booleana são simples. As operações que utilizaremos são (NOT, AND, OR e XOR). Os valores possíveis, tanto para as variáveis quanto para as expressões, são apenas dois (V ou F). No entanto, expressões obtidas combinando operações que envolvem um grande número de variáveis podem atingir um grau de complexidade notável. Não obstante, sua avaliação é sempre feita decompondo-se a expressão em operações elementares respeitando-se a ordem de precedência indicada pelos parênteses, avaliando as operações elementares e combinando-se seu resultado. A avaliação pode ser trabalhosa, mas não difícil.

Ex:

Sendo A e B duas variáveis que possuem respectivamente os valores V e F, calcule as expressões abaixo:

Lembrando que:  $\wedge$  = E       $\vee$  = OU       $\sim$  = NOT

<b>A <math>\wedge</math> B</b>	<b>(A <math>\wedge</math> B) <math>\vee</math> B</b>	<b><math>\sim</math>( (B <math>\vee</math> A) <math>\wedge</math> A)</b>
V $\wedge$ F	( V $\wedge$ F) $\vee$ F	$\sim$ ( (F $\vee$ V) $\wedge$ V)
F	F $\vee$ F	$\sim$ ( V $\wedge$ V)
	F	$\sim$ V
		F

**Obs:** O operador E ( $\wedge$ ) tem prioridade sobre o operador OU ( $\vee$ ).

EX: Na expressão  $P \vee Q \wedge P$ , como não há parênteses, deve-se resolver primeiro o  $Q \wedge P$ .

## Exercícios:

Sejam P e Q e R variáveis cujos valores são respectivamente V, V e F, calcule as expressões abaixo:

<b>a)</b> $(P \wedge Q) \wedge \sim R$	<b>b)</b> $\sim(Q \wedge Q) \vee \sim(P \vee R)$	<b>c)</b> $P \wedge \sim((Q \vee \sim P) \wedge (R \text{ XOR } \sim P))$
<b>d)</b> $(Q \text{ XOR } R) \wedge \sim(R \vee Q)$	<b>e)</b> $(R \vee \sim R) \wedge \sim(\sim(R \text{ XOR } P) \wedge \sim Q)$	<b>f)</b> $\sim(P \wedge Q) \vee (\sim P \text{ XOR } Q) \wedge (R \vee \sim Q)$



Respostas:

<b>a)</b> $(P \wedge Q) \wedge \sim R$ $(V \wedge V) \wedge V$ $V \wedge V$ <b>V</b>	<b>b)</b> $\sim(Q \wedge Q) \vee \sim(P \vee R)$ $\sim(V \wedge V) \vee \sim(V \vee F)$ $\sim V \vee \sim V$ $F \vee F$ <b>F</b>	<b>c)</b> $P \wedge \sim((Q \vee \sim P) \wedge (R \text{ xor } \sim P))$ $V \wedge \sim((V \vee \sim V) \wedge (F \text{ xor } \sim V))$ $V \wedge \sim((V \vee F) \wedge (F \text{ xor } F))$ $V \wedge \sim(V \wedge F)$ $V \wedge V$ <b>V</b>
<b>d)</b> $(Q \text{ xor } R) \wedge \sim(R \vee Q)$ $(V \text{ xor } F) \wedge \sim(F \vee V)$ $V \wedge F$ <b>F</b>	<b>e)</b> $(R \vee \sim R) \wedge \sim(\sim(R \text{ xor } P) \wedge \sim Q)$ $(F \vee \sim F) \wedge \sim(\sim(F \text{ xor } V) \wedge \sim V)$ $V \wedge \sim(\sim V \wedge F)$ $V \wedge \sim(F \wedge F)$ $V \wedge V$ <b>V</b>	<b>f)</b> $\sim(P \wedge Q) \vee (\sim P \text{ xor } Q) \wedge (R \vee \sim Q)$ $\sim(V \wedge V) \vee (\sim V \text{ xor } V) \wedge (F \vee \sim V)$ $\sim V \vee V \wedge F$ $F \vee F$ <b>F</b>

# O Conceito de Algoritmo

Em nosso dia-a-dia executamos mecanicamente uma série de ações que são seguidas sequencialmente e que provocam o acontecimento de algo. Por exemplo, temos um trajeto frequente ao sairmos diariamente de casa em direção ao nosso trabalho ou à nossa universidade que, sempre que seguido, nos leva ao nosso destino. Isso é um algoritmo. Em outras palavras, um algoritmo descreve eventos com duração finita, que envolvem um conjunto de objetos cujas características podem ser alteradas, através de ações que ocorrem sequencialmente.

## O que é um algoritmo?

"Programar é construir algoritmos"

"Programa = Algoritmo + Estruturas de Dados"

"No processo de construção de programas a formulação do algoritmo e a definição das estruturas de dados estão intimamente ligadas"

Num algoritmo podem-se observar os seguintes aspectos:

- Ação: evento que ocorre num período de tempo finito
- Estado: propriedades de um objeto numa dada situação
- Processo: sequência temporal de ações
- Padrão de comportamento: toda vez que é seguido, um evento ocorre

➔ Exemplo de um algoritmo: Algoritmo para fazer " batatas fritas para o jantar "

*"Traga a cesta com batatas da despensa" ;*

*"Traga a panela do armário";*

*"Coloque óleo na panela";*

*Se "a roupa é clara"*  
     *então "coloque o avental";*

*Enquanto "nº de batatas é insuficiente para o número de pessoas" faça*  
     *"descasque as batatas";*

*"Pique as batatas";*

*"Esquente o óleo da panela";*

*"Frite as batatas na panela";*

*"Escorra o excesso de óleo das batatas fritas";*

*"Coloque as batatas fritas numa vasilha com papel absorvente".*



Apesar de muito simples, algumas observações importantes podem ser notadas neste algoritmo:

- Há um sequenciamento das ações, que estão separadas por um ponto-e-vírgula ";"
- Avental não é usado toda vez: existe um motivo para colocá-lo, ou seja, há uma condição para que o avental seja colocado;
- O número de batatas descascadas varia; a ação de "descascar uma batata" repete-se até que a condição de parada (ser suficiente para alimentar as pessoas que irão jantar) seja alcançada;
- A ordem das ações é importante: primeiro descasca-se a batata, pica-se a batata, para depois fritá-la...

➔ Exemplos de algoritmos conhecidos:

- Qual é o algoritmo que você descreve para vir estudar?
- Qual é o algoritmo para se fazer uma feijoada?

Apesar de receitas culinárias e trajetos rotineiramente percorridos encaixarem-se perfeitamente no conceito inicial de algoritmo, no nosso curso estamos interessados num tipo de algoritmo especial, que seja capaz de ser executado por um computador. Para tanto, é necessário que identifiquemos problemas do mundo real que possam ser traduzidos em ações primitivas finitas e dos quais se possa extrair um *padrão de comportamento*.

→ Qual o padrão de comportamento utilizado para gerar as sequências?

---

1, 5, 9, 13, 17, 21, 25 ...  
 1, 1, 2, 3, 5, 8, 13, 21, 34 ...  
 11, 21, 1211, 111221, 312211,...

---

Os dois exemplos anteriores são problemas do mundo real que, por serem finitos (ou para os quais se possa determinar uma condição de parada) e por possuírem um padrão de comportamento, podem ser resolvidos através de um programa de computador. No entanto, antes de se chegar ao programa de computador, o 1º passo é fazer um *algoritmo* que seja capaz de solucionar o problema em questão....

→ Definição de algoritmo:

*Um algoritmo é a descrição de um padrão de comportamento, expresso em termos de um repertório bem definido e finito de ações primitivas que podem ser executadas*

→ Num algoritmo distinguem-se claramente dois aspectos:

- Aspecto estático: texto
- Aspecto dinâmico: sua execução (a partir de valores iniciais)

O curso de Algoritmos é, na realidade, um curso de Programação de Computadores para alunos que iniciam cursos superiores na área de Informática. Para começarmos a construir algoritmos e fazermos nossos primeiros programas de computador, é necessário que o aluno domine uma série de conceitos básicos, que são apresentados a seguir, a começar pelo próprio computador!

## História das linguagens de programação

[http://pt.wikipedia.org/wiki/Linguagem\\_de\\_programa%C3%A7%C3%A3o](http://pt.wikipedia.org/wiki/Linguagem_de_programa%C3%A7%C3%A3o)

O primeiro trabalho de linguagem de programação foi criado por Ada Lovelace. A linguagem de programação ADA foi batizada em homenagem a esta primeira programadora.

O primeiro compilador foi escrito por Grace Hopper, em 1952, para a linguagem de programação A-0. A primeira linguagem de programação de alto nível amplamente usada foi Fortran, criada em 1954.



Grace Hopper em 1984

Em 1957 foi criada B-0, sucessora da A-0, que daria origem a Flow-Matic (1958), antecessor imediato de COBOL, de 1959. O COBOL foi uma linguagem de ampla aceitação para uso comercial. A linguagem ALGOL foi criada em 1958-1960. O ALGOL-60 teve grande influência no projeto de muitas linguagens posteriores.

A linguagem Lisp foi criada em 1958 e se tornou amplamente utilizada na pesquisa na área de ciência da computação mais proeminentemente na área de Inteligência Artificial. Outra linguagem relacionada ao campo da IA que surge em 1972 é a linguagem Prolog, uma linguagem do paradigma lógico.

A orientação a objetos é outro marco importante na história das linguagens de programação. A linguagem Simula introduz o conceito de classes. A linguagem Smalltalk expande o conceito de classes e se torna a primeira linguagem de programação que oferecia suporte completo à programação orientada a objetos. A linguagem C++ (originalmente conhecida como C com classes) populariza a orientação a objetos.

Diversas linguagens de programação surgiram desde então, grande parte orientadas a objetos. Entre estas incluem-se C#, VB.NET, Java, Object Pascal, Objective-C, PHP, Python e Ruby.



Niklaus Wirth em 2005. Criador da linguagem Pascal entre outras.

## Conceitos Básicos da Programação de Computadores

**Computador:** é uma máquina capaz de seguir uma espécie de algoritmo chamado *programa* que está escrito em linguagem de máquina.

**Linguagem de máquina:** internamente o computador executa uma série de instruções que ficam armazenadas em sua memória principal em código binário, ou seja, em linguagem de máquina (zeros (0) e uns (1) que são os dígitos binários ou *bits*).

**Linguagem de alto nível:** para escrever os programas de computador, os programadores utilizam linguagens que estão mais próximas da linguagem humana, que são chamadas de linguagens de alto nível ou simplesmente linguagens de programação. Exemplos de linguagens de alto nível são Pascal, C, C++, Basic, Fortran, dentre muitas outras.

**Linguagem de baixo nível (ou linguagens de montagem):** há programas de computador que precisam interferir diretamente no *hardware* da máquina para permitir a execução de funções específicas como as oferecidas por sistemas operacionais, por exemplo. Neste caso, os programadores utilizam as linguagens de montagem ou linguagens *assembly*, que estão muito próximas da linguagem de máquina e mais distantes das linguagens de programação, sendo por isso chamadas de linguagem de baixo nível.

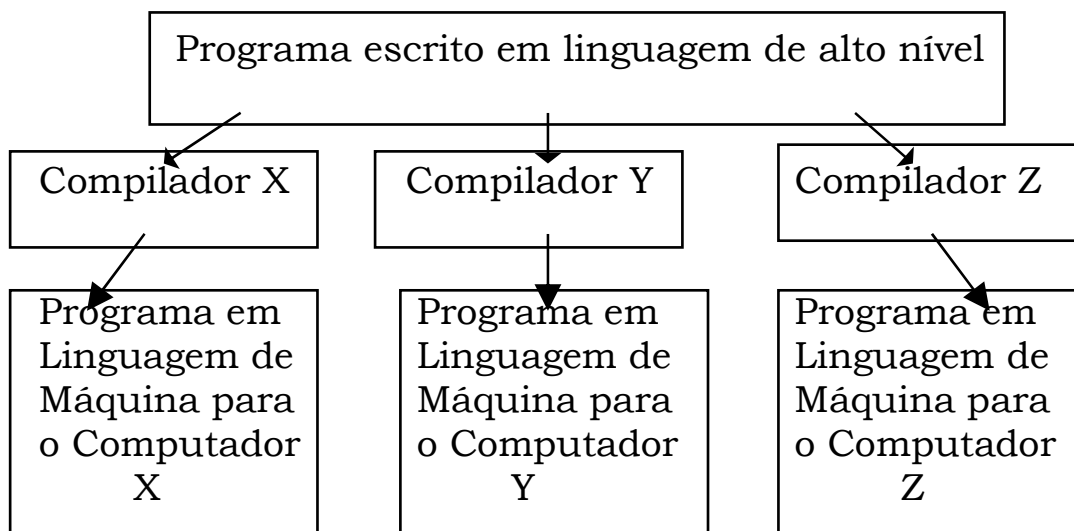
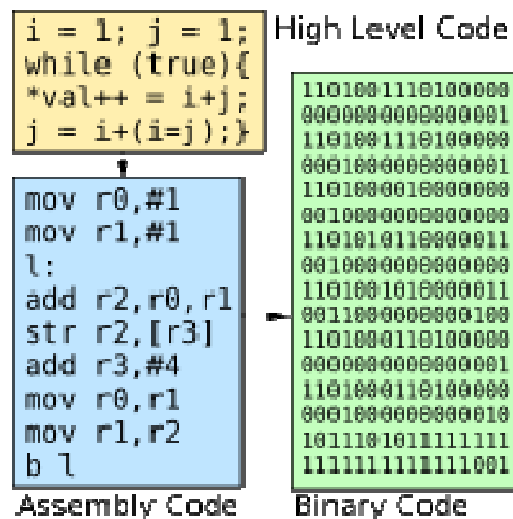
As instruções escritas nas linguagens de baixo nível são compostas, em sua maioria por código de máquina que é enviado e executado pelo processador. As instruções compreendem as características arquiteturais da máquina, seus algoritmos utilizam somente instruções diretas com processador. Para que isso seja possível essas linguagens devem conhecer os registradores que compõem o processador.

Nota sobre registradores: O registrador de uma CPU (unidade central de processamento) é uma unidade de memória capaz de armazenar n bits. Os registradores estão no topo da hierarquia de memória, sendo assim, são o meio mais rápido e caro de se armazenar um dado.

Lembrando que os registradores são circuitos digitais capazes de armazenar e deslocar informações binárias, e são tipicamente usados como um dispositivo de armazenamento temporário.

São utilizados na execução de programas de computadores, disponibilizando um local para armazenar dados. Na maioria dos computadores modernos, quando da execução das instruções de um programa, os dados são movidos da memória principal para os registradores. Então, as instruções que utilizam estes dados são executadas pelo processador e, finalmente, os dados são movidos de volta para a memória principal. Fonte: [http://pt.wikipedia.org/wiki/Registrador\\_\(inform%C3%A1tica\)](http://pt.wikipedia.org/wiki/Registrador_(inform%C3%A1tica))

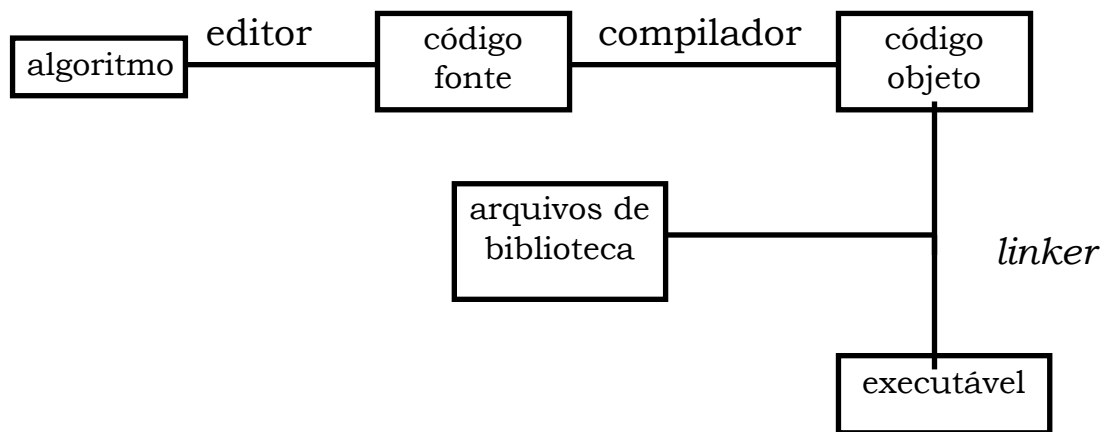
**Compilador:** permite que os programadores utilizem linguagens de alto nível para escrever os programas de computador, pois se encarrega de traduzi-los para linguagem de máquina. O compilador é um programa que traduz uma determinada linguagem de programação para linguagem de máquina. Desta forma, existem diversos compiladores específicos para cada uma das linguagens de programação e específicos para cada sistema operacional, conforme ilustra a figura abaixo. Um compilador C para o sistema Windows 98 (compilador X) é diferente de um compilador C para o sistema Unix (compilador Y), embora a linguagem de programação seja a mesma (linguagem de alto nível). O compilador de linguagem de montagem é chamado de *assembler*.



Os compiladores são específicos para cada linguagem e para cada sistema operacional



O ciclo completo da elaboração do algoritmo à execução de um programa de computador pode ser visto na figura abaixo. Cada um dos componentes deste ciclo é explicado a seguir.



Do algoritmo à execução de um programa de computador

**Algoritmo:** estrutura do programa; instruções que descrevem a lógica do programa

**Editor de texto:** permite que o código fonte do programa seja editado em um arquivo-fonte. Alguns compiladores têm editores com ambiente de programação integrados, como é o caso do Turbo Pascal e do Turbo C.

**Código fonte:** conjunto de comandos escritos na linguagem de programação escolhida (como Pascal ou C). O código fonte fica armazenado no arquivo-fonte em formato *ASCII*. (O arquivo-fonte possui a extensão relativa à linguagem de programação usada, por exemplo, *.pas* (Pascal), *.C* (C), *.cpp* (C++)).

**Compilador:** lê o código fonte do programa e cria um outro arquivo em linguagem binária ou de máquina.

**Código objeto:** arquivo com o programa em linguagem de máquina (o arquivo-objeto possui a extensão *.obj* para a maioria das linguagens de programação).

**Arquivos de biblioteca:** contém funções já compiladas que podem ser utilizadas no programa.

**Linker:** cria um programa executável a partir de arquivos-objeto e dos arquivos de biblioteca.

**Código executável:** programa que pode ser executado no computador (o arquivo-executável possui a extensão *.exe*).



## Gerações das linguagens

Fonte: [http://pt.wikipedia.org/wiki/Linguagem\\_de\\_programa%C3%A7%C3%A3o](http://pt.wikipedia.org/wiki/Linguagem_de_programa%C3%A7%C3%A3o)

### Linguagem de programação de primeira geração

A primeira geração, ou 1GL, é o código de máquina. Essas linguagens utilizam código de máquina nativo do microprocessador, é a única que um microprocessador pode compreender. As composições são códigos com caracteres de controle, não pode ser escrito ou lido por um editor de texto. É uma linguagem raramente usada diretamente por uma pessoa. Nela utiliza-se apenas 0 (zero) e 1 (um) para programar softwares.

Originalmente nenhum tradutor foi usado para compilar ou montar as linguagens de primeira geração. As instruções de programação eram submetidas através dos interruptores localizados no painel frontal de sistemas de computadores. O principal benefício na primeira geração é que o código escrito pelo usuário é muito rápido e eficiente, desde que este código seja diretamente executado pela CPU. O surgimento de linguagens de programação com maior capacidade de abstração e menor ocorrência de erros levaram a linguagem de máquina a cair em desuso rapidamente, sendo usada hoje apenas de modo indireto (através de tradutores).

### Linguagem de programação de segunda geração

A segunda geração, ou 2GL, é uma linguagem que embora não seja uma linguagem nativa do microprocessador, como é o caso da 1GL, um programador que a utilize deve compreender as características arquiteturais do microprocessador, deve entender o funcionamento dos registradores e instruções destes.

Uma representante das linguagens de baixo nível 2GL é a linguagem Assembly. Não é nativa do microprocessador para pertencer a 1GL, no entanto as características da arquitetura do microprocessador devem ser conhecidas pelo programador que pretende trabalhar com ela. As instruções que são executadas na linguagem de máquina normalmente são representadas por uma sequência de números binários ou hexadecimais.

O programa debug da plataforma Microsoft que roda sob o DOS pode ser usado para editar código em linguagem de máquina. (DCA 800, 2004, p 3-5).

```

100.          ;-----
101.          ; zstr_count:
102.          ; Counts a zero-terminated ASCII string to determine its size
103.          ; in:  eax = start address of the zero terminated string
104.          ; out: ecx = count = the length of the string
105.
106.          zstr_count:          ; Entry point
107. 00000030 B9FFFFFF      mov  ecx, -1      ; Init the loop counter, pre-decrement
108.                                     ; to compensate for the increment
109.          .loop:
110. 00000035 41          inc  ecx          ; Add 1 to the loop counter
111. 00000036 803C0800      cmp  byte [eax + ecx], 0      ; Compare the value at the string's
112.                                     ; [starting memory address Plus the
113.                                     ; loop offset], to zero
114. 0000003A 75F9          jne  .loop      ; If the memory value is not zero,
115.                                     ; then jump to the label called '.loop',
116.                                     ; otherwise continue to the next line
117.          .done:
118.                                     ; We don't do a final increment,
119.                                     ; because even though the count is base 1,
120.                                     ; we do not include the zero terminator in the
121.                                     ; string's length
122. 0000003C C3          ret      ; Return to the calling program

```

Exemplo de código Assembly (2GL)

## Linguagem de programação de terceira geração

Uma linguagem de terceira geração (3GL, em inglês) é uma linguagem de programação projetada para ser facilmente entendida pelo ser humano, incluindo coisas como variáveis com nomes. Um exemplo disso seria: `COMPUTE COMISSAO = VENDA * 0,5`

Fortran, ALGOL e COBOL são algumas das primeiras linguagens desse tipo. A maioria das linguagens "modernas" (BASIC, C, C++) são de terceira geração. A maioria das linguagens de terceira geração suportam programação estruturada.

## Linguagem de programação de quarta geração

As linguagens de programação de quarta geração, ou 4GL em sua abreviatura de origem inglesa, são linguagens de programação de alto-nível com objetivos específicos, como o desenvolvimento de softwares comerciais de negócios. Elas permitem o programador especificar o que deve ser feito visando um resultado imediato.

O termo 4GL foi usado primeiramente por James Martin em seu livro publicado em 1982 "Applications Development Without Programmers" para se referir a estas linguagens não-procedimentais e de alto-nível. Alguns acreditam que a primeira 4GL foi uma linguagem chamada Ramis, desenvolvida por Gerald C. Cohen na empresa Mathematica (uma companhia de software matemáticos). Cohen deixou Mathematica e fundou a Information Builders, para criar uma 4GL similar, chamada FOCUS.

A principal diferença entre as linguagens de terceira e quarta geração, é que estas primeiras são linguagens procedurais que descrevem como fazer algo, enquanto a 4GL descreve o que você quer que seja feito.

Uma 4GL que se popularizou foi a linguagem SQL (Structured Query Language), que se tornou um padrão para manipulação e consulta de bancos de dados, sendo hoje em dia muito usada em conjunto com as linguagens de terceira geração.

Linguagens de quarta geração tem sido frequentemente comparadas a Linguagens de domínio específicos (DSLs). Alguns pesquisadores afirmam que as 4GLs são um subconjunto de DSLs.<sup>1 2</sup>

### Algumas linguagens de quarta geração bem-sucedidas

- Database query languages
- FOCUS
- Oracle PL/SQL
- NATURAL
- Progress 4GL
- SQL
- Criação de GUI
- 4th Dimension (Software)
- Delphi programming language
- eDeveloper
- MATLAB's GUIDE
- Progress 4GL AppBuilder
- Revolution programming language
- Visual Basic's form editor
- Windows Forms (part of the .NET Framework)
- OpenROAD
- Powerbuilder

# Linguagens Compiladas, Linguagens Interpretadas e Modelo Híbrido

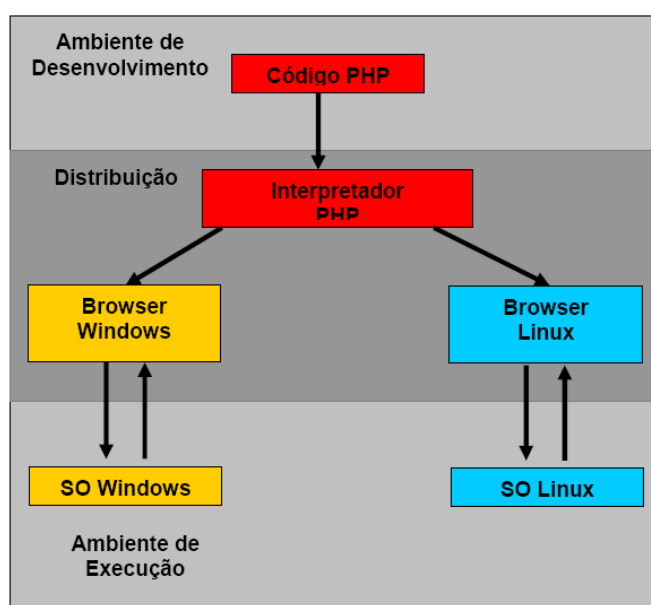
Fonte: <http://rcs.comp.googlepages.com/POO-Capitulo02-ParadigmasdeProgramao.pdf>

Linguagens de programação são comumente divididas em linguagens **interpretadas** e **compiladas**. Já existem algumas de nova geração com conceito **híbrido**.

## Interpretadas

Como o próprio nome diz, são interpretadas linha a linha em tempo de execução. Nessa categoria normalmente o código é armazenado como texto puro sendo transformado em instruções apenas quando são executados, dessa forma, os códigos são expostos a possíveis indivíduos mal-intencionados.

**Têm-se como exemplos de linguagens interpretadas: Perl, ASP (Active ServerPages), JavaScript, PHP e Basic.**



Esquema da execução de um código fonte PHP. O código passa pelo interpretador e é enviado aos browsers escritos para os dois Sistemas Operacionais (Windows e Linux)

## Compiladas

O compilador traduz o programa fonte apenas uma vez para linguagem compilada (executável) não importando quantas vezes o programa irá ser executado. No processo de compilação, o código fonte é submetido à análise sintática, léxica e semântica. Caso algum erro seja encontrado, o arquivo executável não é gerado, e os erros são apontados pelo compilador. Muitos erros são eliminados durante o processo de compilação como, por exemplo, os seguintes erros sintáticos:

- Caracteres inválidos;
- Nomes de variáveis, métodos e classes inválidas;
- Seqüência de comandos inválidos.

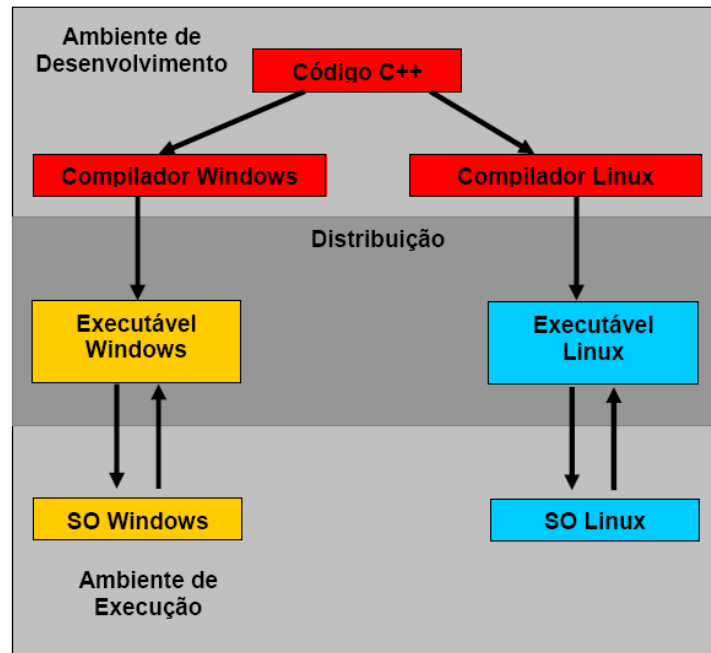
Além de erros semânticos, incluindo:

- Tipos e quantidade de parâmetros, retorno de funções etc;
- Atribuição de um valor alfanumérico para uma variável inteira.

Por outro lado, erros lógicos não são capturados no processo de compilação, gerando algum tipo de erro apenas ao ser executado, como por exemplo:

- Divisão por zero, operadores logicamente errados, etc;

**Como exemplos dessa categoria de linguagens têm-se o Pascal e C/C++.**



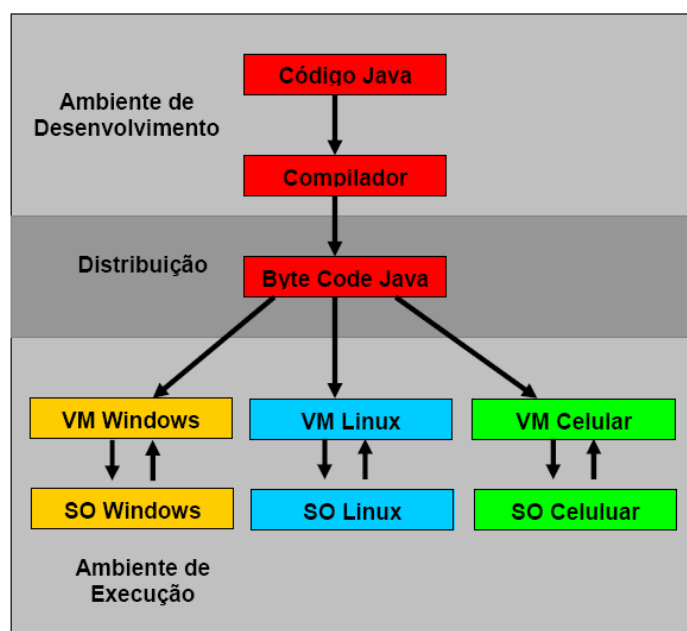
Esquema de criação de um programa em C++. O mesmo código-fonte passa por dois compiladores (um para Windows e outro para Linux) e dá origem a dois tipos de executáveis – um para cada Sistema Operacional

## Híbridas

As plataformas que implantam o conceito híbrido têm como objetivo a segurança das verificações existentes em um processo de compilação e a portabilidade dos ambientes interpretados. O processo adotado para implementação do modelo híbrido baseia-se na utilização de uma representação intermediária denominada *bytecode*, que é gerada pelo compilador e interpretada no momento da execução. A plataforma Java utiliza essa abordagem, contendo os seguintes passos em seu processo de desenvolvimento.

- Arquivos com os códigos-fonte são armazenados como texto simples (.java);
- Após a compilação dos fontes (.java), são gerados os bytecodes (.class);
- Os bytecodes (.class) são utilizados para executar a aplicação, sendo interpretados pela Máquina Virtual Java (JVM).

**Outros ambientes também adotam essa codificação intermediária por meio de bytecodes, como é o caso do Microsoft.NET.**



Esquema de criação de um programa escrito em Java. O Código-Fonte passa pelo compilador e dá origem a um único Byte Code. Este Byte Code é interpretado pelas Máquinas Virtuais Java (VMs) instaladas em cada um dos três tipos de Sistema Operacional – Windows, Linux e Celular. Cada VM foi escrita exclusivamente para aquele Sistema Operacional em particular.

## Linguagens Não Estruturadas

São linguagens mais sofisticadas e flexíveis do que as de baixo nível. A flexibilidade se dá porque seus comandos não estão tão vinculados ao processador e sistemas utilizados, isso torna seu uso possível em diferentes plataformas. As linguagens não estruturadas representam um considerável avanço de qualidade no que diz respeito à programação quando comparadas com as linguagens de baixo nível.

Caracteriza-se pela não modularização, ou seja, não dá pra dissolver um programa escrito nestas linguagens em funções, procedimentos, (functions, procedures) para controle do fluxo programa. O código componente do programa é constituído por uma sequência de comandos ou declarações que modificam dados acessíveis a todos os pontos do programa.

O processo de elaboração nestas linguagens tem várias desvantagens no caso de programas grandes, sobretudo se um encadeamento de código se torna necessário em localizações diferentes. Neste caso, essa sequência necessária em outro ponto deve ser copiada para o local para poder ser usada.

Algumas representantes dessas linguagens são: a batch do MS-DOS (arquivo.bat), e o BASIC.

Um exemplo de arquivo batch pode ser visto no quadro abaixo:

```
01  @ECHO OFF
02  GOTO :Inicio
03  :Erro
04  ECHO Ocorreu um Erro de nível 6!
05  GOTO Fim
06  :Inicio
07  PRG
08  IF ERRORLEVEL 6
09  GOTO Erro
10  :Fim
11  @ECHO ON
```

Exemplo de código de linguagem não estruturada.

Os códigos das linguagens não estruturadas não são muito intuitivos. Essas linguagens tornaram-se pouco a pouco em desuso com a introdução de linguagens estruturadas mais sofisticadas, tais como as linguagens procedurais, funcionais e as orientadas a objetos

## Programação em Linguagem Estruturada

As linguagens modeladas em estruturas foram desenvolvidas entre 1974 e 1986. A característica principal desse estilo de programação em estruturas é exatamente a permissão do uso de estruturas “structs” no caso da linguagem “C”, ou “records” no caso da linguagem “Pascal”.

O conceito central das linguagens estruturadas é decompor uma codificação extensa e de complexidade acentuada em muitas partes menores e independentes para facilitar o trabalho e o entendimento. Ainda se



destacam a maior capacidade de programação procedural e estrutural dos dados. Essas linguagens possibilitaram a criação de sistemas distribuídos, incorporação de recursos inteligentes sem muita requisição de hardware.

Os mecanismos de controle das linguagens estruturadas diferem das linguagens não estruturadas em termos de promoção de testes condicionais (if then else), e em termos de controle de repetição de blocos de código (for, while, do), também podem fazer seleções alternativas como o switch - case.

A programação estruturada decompõe o código do programa em módulos com funções e ou procedimentos. Os procedimentos não retornam valor enquanto as funções retornam algum valor sendo que em ambos (procedimentos e funções) podem conter dentro de seus escopos outros procedimentos ou funções. Blocos com comandos BEGIN e END podem ser codificados em qualquer parte do programa.

São representantes de linguagens estruturadas também conhecidas como de terceira geração, as linguagens COBOL, Basic, C e Pascal.

O quadro a seguir mostra o código de um programa codificado em linguagem estruturada em pascal:

```

01  program numerosPares;
02  var n1, n2: integer;
03  begin
04      writeln('Digite dois Inteiros para Exibir Uma Lista de Números: ');
05      readln(n1, n2 );
06      if (n1 mod 2) <> 0 then
07          n1 := n1 + 1;
08      while n1 <= n2 do
09          begin
10              writeln(n1, ' - ');
11              n1 := n1 + 2;
12          end;
13      writeln('Fim da Lista de Números');
14  end.
```

*Exemplo de um programa em linguagem estruturada.*

Para conseguirmos construir programas de computador, é necessário cumprir as 5 etapas básicas da programação, das quais a confecção do algoritmo é extremamente importante.

As 5 etapas da programação de computadores são:

1. Identificação de um problema do mundo real
2. Confecção do algoritmo
3. Teste do algoritmo
4. Confecção do programa de computador
5. Execução do programa no computador

Para realizarmos este ciclo de etapas com sucesso, quer dizer, para conseguirmos fazer programas de computador com qualidade e confiáveis, é muito importante a utilização de técnicas programação como a programação estruturada.

A programação estruturada é uma metodologia de projeto e desenvolvimento, que pretende:

- facilitar a escrita;
- facilitar o entendimento;
- permitir a verificação;
- facilitar a alteração e a manutenção dos programas de computador

*O principal objetivo da metodologia de programação estruturada é reduzir a complexidade*

*"A arte de programar consiste na arte de organizar e dominar a complexidade."*

## Programação Orientada a Objetos

A problemática relacionada à solução de problemas tem sido bastante abordada e teve uma visível diminuição com o pensamento dos envolvidos na tarefa evolutiva das linguagens de programação, o lançamento de uma nova linguagem tem representado muitos avanços, em determinados limites, em relação às anteriores.

A técnica de POO é uma forma de programar na qual o desenvolvedor escreve o código de forma que este fique organizado em classes que descrevem objetos. Os objetos contêm características tais como propriedades e métodos que executam ações através de interações entre si. (FARINELLI, 2007 p. 4).

Uma linguagem é caracterizada como Orientada a Objetos quando atende a estes quatro tópicos:

- Abstração
- Encapsulamento
- Herança
- Polimorfismo

Maria pergunta ao Zé:  
"- Moooooooooor, o que é leptospirose?"  
E o Zé responde na lata:  
"- Fofa, é uma doença que ataca os  
usuários de lépitópi. É transmitida pela  
urina do mouse..."



# Expressão de Algoritmos

Os algoritmos podem ser expressos através de diagramas, através de pseudo-linguagens ou através da própria linguagem de programação. Vamos examinar cada uma destas três opções e, no final deste capítulo, vamos optar por uma delas para que possamos começar a fazer nossos primeiros algoritmos!

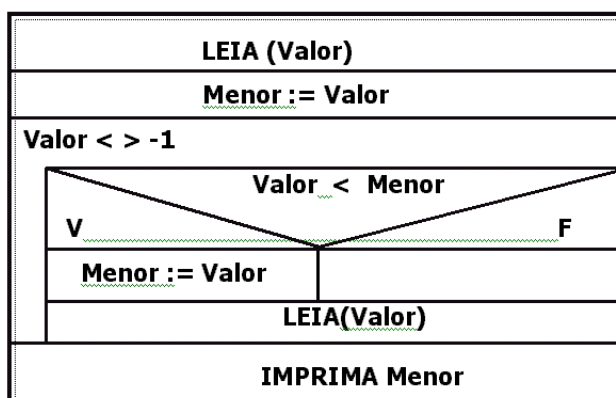
## Expressão de Algoritmos através de Diagramas e Fluxogramas

A utilização de diagramas para a expressão de algoritmos foi bastante utilizada até a década de 1980. Diagramas de Chapin e fluxogramas foram os principais métodos utilizados, então. Nos métodos baseados em diagramas, uma grande variedade de formas geométricas como quadrados, retângulos, hexágonos, pentágonos, etc., são utilizadas para representar as instruções de leitura e impressão de dados, assim como os comandos condicionais, de repetição, etc. Além disso, uma série de regras para a disposição dessas formas e/ou setas para representar o sequenciamento das instruções fazem parte desses métodos.

Apesar de terem sido utilizados largamente pelas primeiras gerações de programadores, estes métodos apresentam uma série de inconveniências como:

- ✓ O programador tem que memorizar todas as formas geométricas e conhecer as regras de inter-relacionamento entre elas;
- ✓ O programador perde um tempo considerável para fazer e refazer desenhos, tendo que possuir diversas réguas com os símbolos dos diagramas;
- ✓ Para algoritmos muito grandes, os desenhos começam a ocupar muitas páginas, tornando impraticável a visualização de toda a solução;
- ✓ A memorização de regras de expressão desvia a atenção do programador que não tem apenas de se concentrar na lógica do problema, como seria desejável, mas tem as preocupações adicionais de elaborar desenhos e consultar regras e regras...

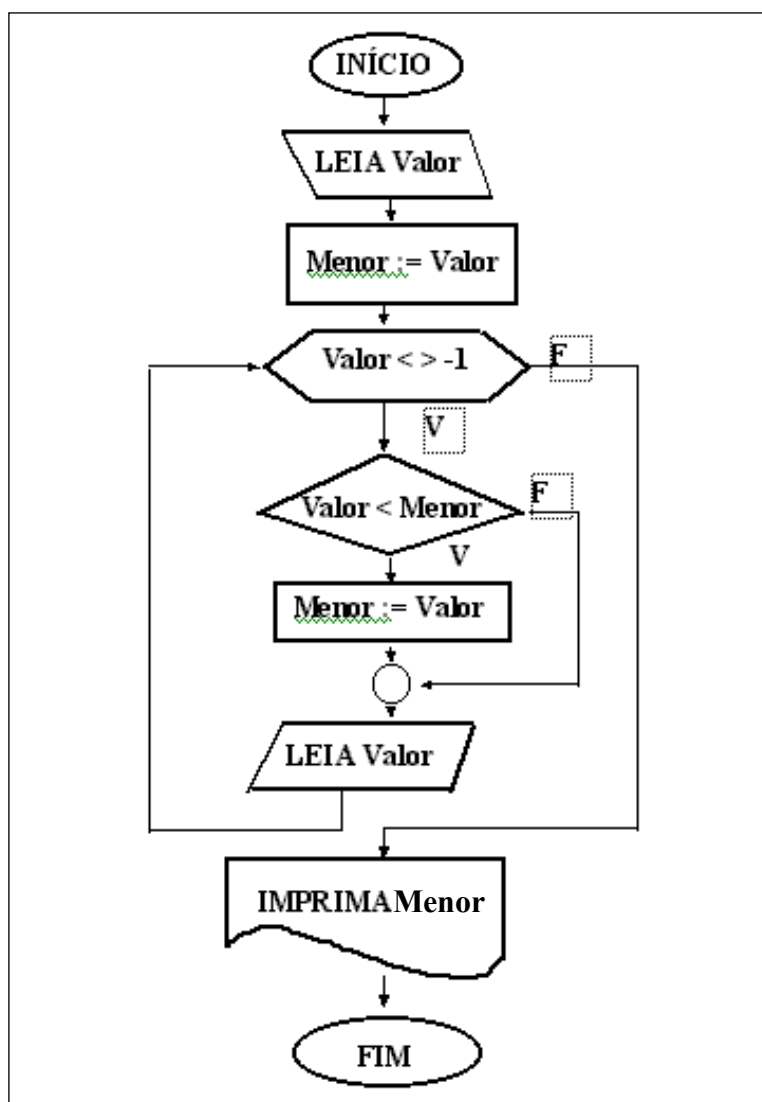
Além de todos os inconvenientes citados, que de longe esgotam os seus problemas, os métodos baseados em diagramas se distanciam muito do alvo da programação que é a expressão da lógica algorítmica na própria linguagem de programação! Além disso, esses métodos não são nem um pouco intuitivos... veja a figura abaixo e tente descobrir o que o programa faz... sem muito esforço, se for possível:



Algoritmo expresso através de um diagrama de Chapin (desenvolvido por Nassi Shneiderman e ampliado por Ned Chapin) também conhecido como diagrama N-S.

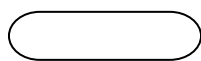
## Exemplo de um Fluxograma

A figura abaixo mostra um exemplo de um algoritmo expresso através de um fluxograma. Observe a existência de diferentes formas geométricas para as diferentes instruções e o uso de setas para representar o fluxo dos dados. Você seria capaz de dizer o que este algoritmo faz? Quais as variáveis que ele utiliza e de que tipo básico elas são? (Dica: este algoritmo faz a mesma coisa que o expresso pelo diagrama de Chapin mostrado na figura anterior).

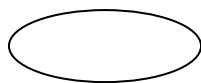


Algoritmo expresso através de um Fluxograma

## Alguns símbolos utilizados em diagramas de bloco e fluxogramas:



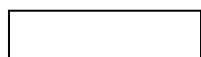
**Terminal:** Indica o início e o fim do diagrama.



**Terminal:** idem acima.



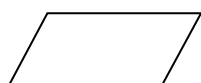
**Seta:** Indica o sentido do fluxo.



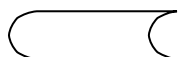
**Processamento:** Indica algum tipo de cálculo/processamento.



**Teclado:** utilizado para informar dados ao computador.



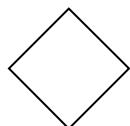
**Teclado:** idem acima.



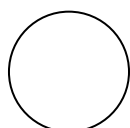
**Vídeo:** utilizado para exibir algo no vídeo.



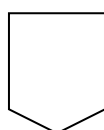
**Vídeo:** idem acima.



**Decisão:** utilizado para indicar uma decisão a ser tomada, indicando os desvios.



**Conector:** utilizado quando é preciso particionar o diagrama. Quando ocorrer mais de uma partição, é colocada uma letra ou número para dentro do símbolo de conexão para identificar os pares de ligação.



**Conector:** específico para indicar conexão do fluxo em outra página.

## Expressão de Algoritmos através de Pseudolinguagem

Uma pseudolinguagem é uma notação para expressão de algoritmos para ser utilizada nas 3 primeiras etapas da programação: *identificação de um problema do mundo real, confecção e teste do algoritmo*. É apresentada na forma de português estruturado. Embora seja uma pseudolinguagem, possui estrutura, sintaxe e semântica semelhantes às de uma linguagem de programação.

**A principal diferença entre a pseudolinguagem e a linguagem de programação é que a primeira não possui um compilador. Isso significa que é possível expressar o raciocínio algorítmico utilizando-se uma pseudolinguagem, mas o programa não pode ser executado no computador.**

São muitas as vantagens de se utilizar uma pseudolinguagem para escrever algoritmos:

- ✓ É uma linguagem independente de máquina; o programador pensa somente no problema a ser resolvido sem se preocupar com possíveis restrições do compilador ou do *hardware* (computador);
- ✓ O programador tem que conhecer a sintaxe, a semântica e a estrutura da pseudolinguagem, mas tem total liberdade para criar novos comandos ou usar instruções em alto nível (ou em forma de frases): por um lado ele vai se acostumando com a rigidez da sintaxe das linguagens de programação, mas por outro lado, tem a liberdade de expressar seu raciocínio sem esbarrar em limitações de contexto;
- ✓ Uma vez estando pronto o algoritmo na pseudolinguagem, a sua implementação no computador (etapas 4 e 5 da programação: *confecção e execução do programa*) fica muito facilitada, pois toda a lógica já foi desenvolvida e testada e somente uma tradução para a linguagem de programação-alvo se faz necessária.

Veja o exemplo de um algoritmo escrito em pseudolinguagem:

```

INÍCIO
VARIÁVEIS
S, C, I, A, MD:Real;

S ← 0;
C ← 0;
PARA I de 1 ATÉ 10 FAÇA PASSO 1
    Escreva "Digite um número: ";
    LEIA A;
    SE A ≥ 0 ENTÃO
        S ← S + A;
        C ← C + 1;
    FIM SE;
FIM PARA;

MD ← S / C;
ESCREVER ("A média é: ", MD);

FIM

```

Algoritmo expresso em pseudolinguagem (<https://pt.wikipedia.org/wiki/Pseudoc%C3%B3digo>).

## Expressão de Algoritmos através de Linguagem de Programação

Expressar um algoritmo através de uma linguagem de programação é o objetivo, a meta do programador, com certeza. Uma linguagem de programação permite, além da expressão do raciocínio algorítmico, a sua execução no computador (por causa do compilador, com já aprendemos). Existem diversas linguagens de programação. Cada uma pode ser mais adequada à resolução de problemas específicos, e recebem alguns rótulos por isso, como linguagens para aplicações científicas, linguagens para desenvolvimento de software básico, linguagens para utilização intensiva de recursos gráficos, para manipulação de bancos de dados, programação para Internet, etc, etc. Há também as linguagens para o ensino de programação! Veja a figura a seguir. É um código escrito em Pascal.



```

PROGRAM MenorValor;
  {Este programa seleciona o menor número em uma sequência de
  números inteiros}

  VAR  Valor, Menor: INTEGER;

  BEGIN
    WRITE('Forneça um número inteiro: ');
    READLN(Valor);
    Menor := Valor;
    WHILE Valor <> -1 DO
      BEGIN
        IF Valor < Menor THEN
          Menor := Valor;
        WRITE('Forneça um número inteiro (flag=-1): ');
        READLN(Valor);
      END; {while}
    WRITELN('O menor valor lido foi: ', Menor);
  END.

```

#### Algoritmo expresso na Linguagem de Programação Pascal

A figura acima mostra um algoritmo expresso na linguagem de programação Pascal, ou seja, já é um programa de computador. Observe a estrutura do programa: começa com PROGRAM, depois tem a seção VAR na qual as variáveis são declaradas e depois vêm as instruções contidas entre o BEGIN (início) e o END. (fim.). Você seria capaz de dizer o que este algoritmo faz? Quais as variáveis que ele utiliza e de que tipo básico elas são? (Dica: este programa faz a mesma coisa que os exemplos anteriores).

Agora observe o exemplo de um programa escrito na linguagem de programação C.

```

/* Este programa seleciona o menor número em uma sequência de
   números inteiros */

#include <stdio.h>

int    valor, menor;

void main (void)
{
  printf ("\n Forneça um número inteiro:   ");
  scanf ("%i", &valor);
  menor = valor;
  while (valor != -1)
  {
    if (valor < menor)
      menor = valor;
    printf("\nForneça um número inteiro (-1 para terminar):  ");
    scanf ("%i", &valor);
  }
  printf ("O menor valor lido foi: %i ", menor);
}

```

#### Algoritmo expresso na Linguagem de Programação C

A figura acima mostra um algoritmo expresso na linguagem de programação C, ou seja, também é um programa de computador. Observe a estrutura do programa: começa com um comentário, depois tem a inclusão de uma biblioteca de funções, depois a declaração das variáveis e em seguida vem a seção `main` na qual as instruções do programa estão contidas entre o `{` (início) e o `}` (fim). Você seria capaz de dizer o que este algoritmo faz? Quais as variáveis que ele utiliza e de que tipo básico elas são? (Dica: este programa faz a mesma coisa que o expresso pelo diagrama de Chapin, pelo fluxograma e pelo programa Pascal mostrados anteriormente).

O programa traz um comentário dizendo explicitamente o que ele faz. Portanto, garanto que você conseguiu responder à primeira pergunta. Com relação às variáveis que o programa utiliza e seu tipo básico, observe a declaração `"int valor, menor;"` no programa. Há duas variáveis ali declaradas: `valor` e `menor`... o tipo básico é `int` ... um pouquinho de imaginação e `int` = inteiro!

Que tal você comparar o código C com o código Pascal? Semelhanças? Qual dos dois lhe pareceu mais fácil de entender?

Bem, se sua resposta foi o código Pascal, digo-lhe que o Pascal tem o "rótulo" de linguagem mais adequada ao ensino de programação... Se sua resposta foi o código C, saiba que é uma linguagem que foi criada para ser utilizada por programadores experientes... Mas... se você teve dificuldade de entender ambos os códigos, não se preocupe! O curso está apenas começando e há muito que se aprender de lógica de programação! Garanto que no final do curso este programa vai parecer muito, muito fácil para você!

Apesar do Pascal ter se consagrado como uma linguagem adequada para o ensino da programação aos estudantes que a iniciam, ou seja, como primeira linguagem de programação, a linguagem C, antes restrita à comunidade científica, ganhou uma popularidade inquestionável na década de 1990, que se estende aos dias atuais, pois é a base de novas linguagens e paradigmas. Em função disto, a linguagem C passou a ser alvo do interesse dos estudantes. A resistência em utilizá-la como primeira linguagem de programação pelos professores se deveu, em parte, ao fato de seus criadores, Dennis Ritchie e Brian Kernighan, terem afirmado que *"C retém a filosofia básica de que os programadores sabem o que estão fazendo"*<sup>1</sup>. Bem, se considerarmos que programadores iniciantes não têm condições de saber com segurança o que estão fazendo, exatamente pelo fato de serem inexperientes, a adoção da linguagem seria realmente questionável.

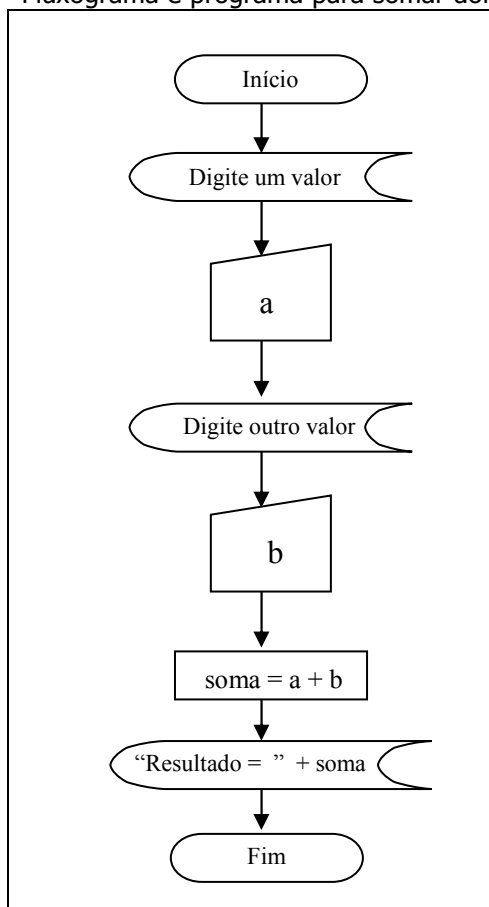
No entanto, a utilização da Informática evoluiu de tal forma que já no ensino médio, em muitas escolas brasileiras, são ensinadas aos estudantes noções básicas de programação. Além disso, a adoção de um método adequado e coerente para o ensino da programação de computadores pode atuar como um facilitador do processo de aprendizagem, permitindo que uma linguagem mais complexa, como o C, possa ser ensinada sem oferecer obstáculos à aprendizagem.

---

<sup>1</sup> *Computação e Programação em C*, de Brian Kernighan e Dennis Ritchie, Prentice Hall, 1978.

## Fluxogramas

Fluxograma e programa para somar dois números e exibir o resultado.



### Exercícios

Resolva as equações abaixo utilizando fluxograma:

1 - ) Calcule o valor de R, exibindo-o.

$$R = B^2 - (C * D)$$

2-) Calcule o valor de R, exibindo-o.

$$X = B^3 * H$$

$$R = X / J$$

# Programação em C#

## O que é .NET

.NET é uma plataforma de software que conecta informações, sistemas, pessoas e dispositivos. A plataforma .NET conecta uma grande variedade de tecnologias de uso pessoal, de negócios, de telefonia celular a servidores corporativos, permitindo assim, o acesso rápido a informações importantes onde elas forem necessárias e imprescindíveis.

Desenvolvido sobre os padrões de Web Services XML, o .NET possibilita que sistemas e aplicativos, novos ou já existentes, conectem seus dados e transações independente do sistema operacional(SO) instalado, do tipo de computador ou dispositivo móvel que seja utilizado e da linguagem de programação que tenha sido utilizada na sua criação.

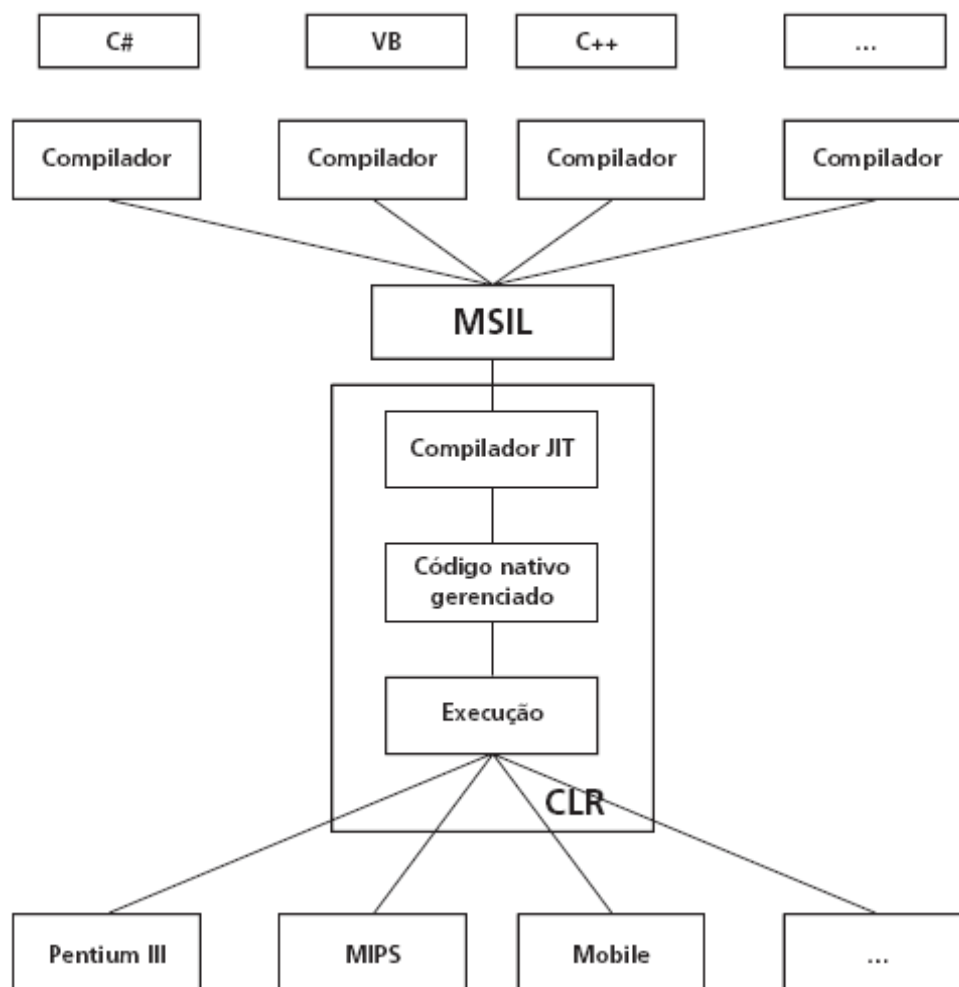
O .NET é um "ingrediente" sempre presente em toda a linha de produtos da Microsoft, oferecendo a capacidade de desenvolver, implementar, gerenciar e usar soluções conectadas através de Web Services XML, de maneira rápida, barata e segura. Essas soluções permitem uma integração mais ágil entre os negócios e o acesso rápido a informações a qualquer hora, em qualquer lugar e em qualquer dispositivo.

## Compilando programas .NET: introduzindo a linguagem intermediária MSIL (Microsoft Intermediate Language)

A MSIL – ou simplesmente IL – é a linguagem intermediária para qual é interpretado qualquer programa .NET, independente da linguagem em que este for escrito. Essa tradução é feita para código intermediário (como em JAVA com os byte codes) sintaticamente expresso na IL. Por sua vez, qualquer linguagem .NET compatível, na hora da compilação, gerará código IL e não código assembly específico da arquitetura do processador onde a compilação do programa é efetuada, conforme aconteceria em C++ ou Delphi, por exemplo. E por que isso? Isso acontece para garantir duas coisas: a independência da linguagem e a independência da plataforma (arquitetura do processador).

A MSIL é a linguagem intermediária para qual é interpretado qualquer programa .NET na hora da compilação, independente da linguagem em que este for escrito.

Pelo dito acima, podemos afirmar que .NET, apesar de inicialmente estar sendo desenhada para a plataforma Microsoft, é uma arquitetura portátil tanto em termos de linguagem de programação quanto no nível da arquitetura do processador, dado que o código gerado pode ser interpretado para a linguagem assembly da plataforma host na hora da execução, sem necessidade de recompilação de código-fonte.



Processo de execução de uma aplicação.

## Gerenciamento da memória: introduzindo o GC (Garbage Collector)

O gerenciamento da memória é efetuado pelo runtime, permitindo que o desenvolvedor se concentre na resolução do seu problema específico. O que diz respeito ao sistema operacional, como o gerenciamento da memória, é feito pelo runtime. Como isso é efetuado? À medida que uma área de memória é necessária para alocar um objeto, o GC ou coletor de lixo (Garbage Collector) realizará essa tarefa, assim como a liberação de espaços de memória que não estiverem mais em uso.

Para os que não trabalham com linguagens de programação como C ou C++, que permitem o acesso direto à memória heap via ponteiros, essa é uma das maiores dores de cabeça que os programadores sofrem, ora por fazer referência a espaços de memória que não foram alocados, ora porque estes espaços já foram liberados anteriormente; é exatamente esse tipo de erro que o coletor de lixo nos ajuda a evitar. O gerenciamento da memória, quando efetuado diretamente pelo programador, torna os programas mais eficientes em termos de desempenho, mas ao mesmo tempo o penaliza, obrigando-o a alocar e desalocar memória quando assim é requerido. A .NET permite que o programador faça esse gerenciamento também, o que é chamado de "unsafe code" (código inseguro); entretanto, por default, o GC é o encarregado dessa tarefa, e o contrário não é recomendado.

## Linguagens que suportam .NET

Dentre as linguagens que suportam .NET podemos citar:

- C# (é claro!)

- C++
- Visual Basic
- Jscript
- Cobol
- Small Talk
- Perl
- Pascal
- Phyton
- Oberon
- APL
- Haskell
- Mercury
- Scheme
- CAML
- OZ

### **Principais vantagens da linguagem C#:**

- Clareza, simplicidade e facilidade: C# é clara, simples, fácil de aprender, mas nem por isso menos poderosa.
- Completamente orientada a objetos: C#, diferentemente de muitas linguagens existentes no mercado, é completamente orientada a objetos. Em C#, tudo é um objeto.
- Não requer ponteiros para gerenciar a memória: C# não requer ponteiros para alocar/desalocar memória heap. Esse gerenciamento, como dissemos acima, é feito pelo GC (Garbage Collector).
- Suporta interfaces, sobrecarga, herança, polimorfismo, atributos, propriedades, coleções, dentre outras características essenciais numa linguagem que se diz orientada a objetos.

### **Quando usar a .NET?**

Como consequência do que foi dito acima, a .NET se adapta perfeitamente ao desenvolvimento do seguinte tipo de aplicações:

- Aplicações clientes de front end
- Aplicações de middleware: Web services, aplicações do lado servidor (ASP.NET, SOAP, Web Services e XML)
- Aplicações para internet: a .NET fornece bibliotecas especializadas para o desenvolvimento de aplicações para Internet suportando os protocolos mais comuns: FTP, SMTP, HTTP, SOAP etc.
- Aplicações gráficas: via a biblioteca GDI+, a .NET dá suporte completo a esse tipo de aplicações.
- Acesso a bancos de dados via ADO.NET: ADO.NET é uma evolução da tecnologia ADO usada amplamente no desenvolvimento de sistemas para bancos de dados. Entretanto, novas características são encontradas nessa nova biblioteca, como manipulação de dados na aplicação cliente, como se esta estivesse sendo manipulada no servidor. Isso implica em aplicações connectionless (sem conexão) com vistas a não degradar o desempenho do servidor de banco de dados, quando este está servindo milhares de conexões simultaneamente.
- Aplicações multitarefa: a biblioteca System.Thread dá suporte ao desenvolvimento de aplicações multitarefa.

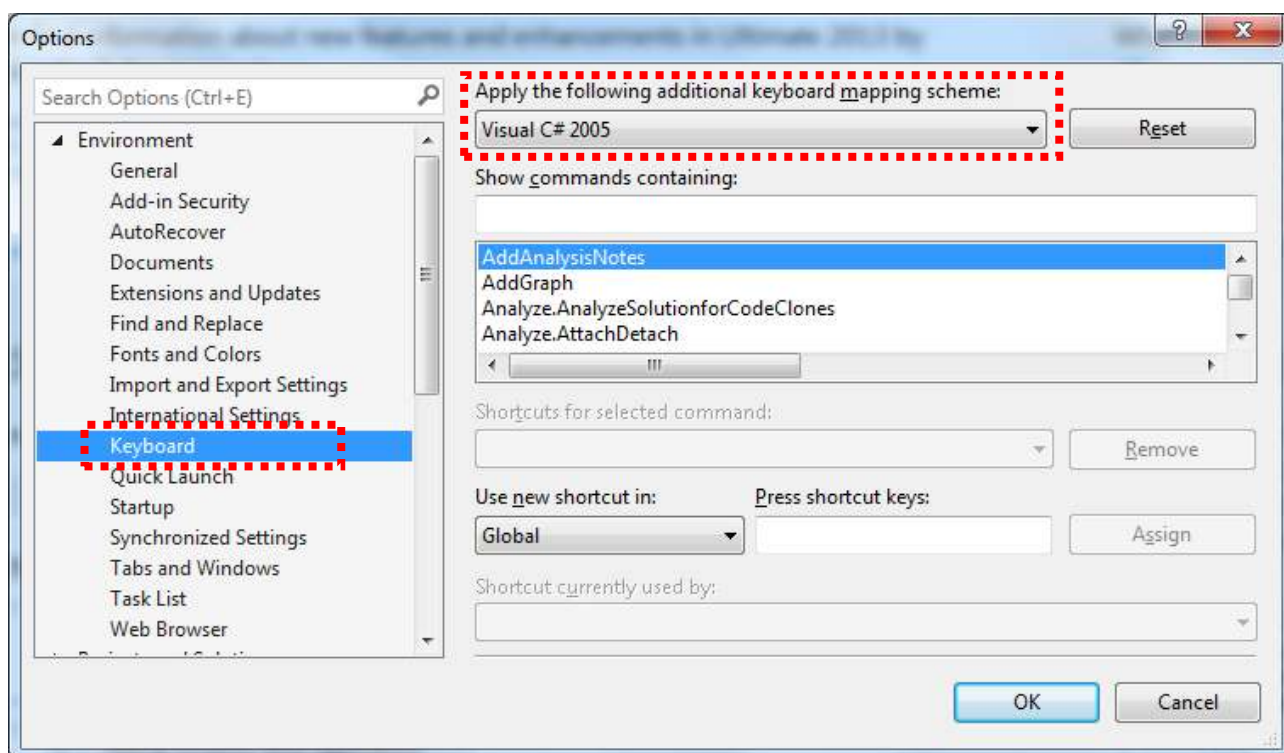




## Configuração do Visual Studio

As teclas de atalho utilizadas nesta apostila são referentes a seguinte configuração de teclado:

Obs: para acessar a tela abaixo, acesse o menu Tools → Options.



## Principais teclas de atalho

Teclas de atalho que podem ser usadas quando o Visual Studio tela estiver configurada como na imagem acima

Descrição	Tecla de Atalho
Janela de propriedades do objeto	F4
Compilar	F6
Executar o programa	F5
Ir para o código fonte do formulário	F7
Retornar ao formulário	Shift + F7
Adicionar namespace	Alt + Shift + F10
Criar propriedade	CTRL + R + E
Depurar sem entrar nos métodos	F10
Depurar entrando nos métodos	F11
Adicionar Break Point	F9
Pesquisar pelo nome de um arquivo no projeto	CTRL + ,
Pesquisar no conteúdo de um arquivo	CTRL + F
Pesquisar no conteúdo de todos os arquivos do projeto	CTRL + SHIFT + F
Identar o código (não pode estar com erro de sintaxe)	CTRL + E + D ou CTRL + K + D

## Estrutura básica de um programa em C#

O pequeno trecho de código a seguir implementa o clássico programa "Olá mundo":

```
using System;
class AppPontoNet
{
    static void Main( )
    {
        // escrevendo no console
        Console.WriteLine("Olá mundo em C#");
        //Aguarda o usuário pressionar Enter para finalizar o programa.
        Console.ReadLine( );
    }
}
```

### O Cabeçalho do programa

A primeira linha do nosso programa, que escreve no console "Olá mundo em C#", contém a informação do namespace System, que contém as classes primitivas necessárias para ter acesso ao console do ambiente .NET. Para incluir um namespace em C#, utilizamos a cláusula **using** seguida do nome do namespace.

### A declaração de uma classe

O C# requer que toda a lógica do programa esteja contida em classes. Após a declaração da classe usando a palavra reservada **class**, temos o seu respectivo identificador. Para quem não está familiarizado com o conceito de classe, apenas adiantamos que uma classe é um tipo abstrato de dados que no paradigma de programação orientada a objetos é usado para representar objetos do mundo real. No exemplo acima, temos uma classe que contém apenas o método Main( ) e não recebe nenhum parâmetro.

### O Método Main( )

Todo programa C# deve ter uma classe que defina o método Main( ), que deve ser declarado como estático usando o modificador **static**, que diz ao runtime que o método pode ser chamado sem que a classe seja instanciada. É através desse modificador que o runtime sabe qual será o ponto de entrada do programa no ambiente Win32, para poder passar o controle ao runtime .NET. O "M" maiúsculo do método Main é obrigatório, e seu valor de retorno void significa que o método não retorna nenhum valor quando é chamado.

Algumas variantes do método Main( ):

// Main recebe parâmetros na linha de comando via o array args

```
static void Main(string[ ] args)
{
    //corpo do método
}
```

// Main tem como valor de retorno um tipo int

```
static int Main( )
{
    // corpo do método
}
```

A forma do método Main( ) a ser usada vai depender dos seguintes fatores:

- O programa vai receber parâmetros na linha de comando? Então esses parâmetros serão armazenados no array args.
- Quando o programa é finalizado, é necessário retornar algum valor ao sistema? Então o valor de retorno será do tipo int.

Alguns últimos detalhes adicionais

- Blocos de código são agrupados entre chaves { }.
- Cada linha de código é separada por ponto-e-vírgula.

- O C# é sensível ao contexto, portanto int e INT são duas coisas diferentes.
- É uma boa prática sempre declarar uma classe onde todos os aspectos inerentes à inicialização da aplicação serão implementados, e obviamente, que conterà o método Main( ) também.
- 

### Comentários

Os comentários de linha simples começam com duas barras //.

Comentários em bloco são feitos usando os terminadores /\* (de início) e \*/ (de fim).

```
/*
    Este é um comentário de bloco
    Segue o mesmo estilo de C/C++
*/
```

## Interagindo com o console

Praticamente toda linguagem de programação oferece meios de interagir com o console, para ler ou escrever na entrada (geralmente o teclado) e saída padrão (normalmente o vídeo em modo texto). Em C#, temos uma classe chamada Console no namespace System, a qual oferece uma série de métodos para interagir com a entrada e saída padrão. Vejamos alguns exemplos:

```
public class LeNome
{
    static void Main()
    {
        char c;
        string nome;
        // Escreve no console sem retorno de carro
        Console.Write("Digite seu nome: ");

        // Lê uma string do console. <Enter> para concluir
        nome = Console.ReadLine();

        // Escreve uma linha em branco
        Console.WriteLine();

        // Escreve uma string no console
        Console.WriteLine("Seu nome é: {0} \nPressione Enter " +
                           "para finalizar.", nome);

        // aguarda o usuário pressionar enter p/ sair do programa
        Console.ReadLine();
    }
}
```

Como você pode ver no exemplo acima, para escrever no console usamos os métodos:

- Console.Write( ), para escrever uma string sem retorno de carro;
- Console.WriteLine( ), para escrever uma string com retorno de carro. Essa string pode ser parametrizada, o que significa que o conteúdo de variáveis pode ser mostrado no console. As variáveis a serem mostradas começam a partir do segundo parâmetro e são separadas por vírgula. Na string do primeiro parâmetro elas são representadas por números inteiros, a começar por zero, encerrados entre terminadores de início "{" e de fim "}".

Exemplo:

```
Console.WriteLine("var1: {0}, var2: {1}, var3: {2}", var1, var2, var3);
```

Para ler dados da entrada padrão, usamos os seguintes métodos:

- Read( ), para ler um caractere simples;
- ReadLine( ) para ler uma linha completa, conforme mostrado no exemplo acima.

Formatando a saída padrão

A formatação da saída padrão é feita usando os chamados "caracteres de escape" (veja a tabela abaixo). Vejamos um exemplo:

\t = TAB

\n = quebra de linha e retorno de carro (CR LF)

```
Console.WriteLine( "var1: {0} \t var2: {1}\t var3: {2}\n", var1, var2, var3);
```

Caractere de Escape - Significado

\n Insere uma nova linha

\t TAB

\a Dispara o som de um alarme sonoro simples

\b Apaga o caractere anterior da string que está sendo escrita no console (backspace)

\r Insere um retorno de carro

\0 Caractere NULL (nulo)

## Principais Operadores

<i>Operadores Aritméticos</i>	
Operador	Uso
+	Soma tipos numéricos. Também é usado para concatenar strings
-	Efetua a diferença de tipos numéricos
/	Efetua a divisão de tipos numéricos
*	Efetua o produto de tipos numéricos
%	Retorna o resíduo da divisão

<b>Operadores Unários</b>	
<b>Operador</b>	<b>Uso</b>
<b>+</b>	Especifica números positivos
<b>-</b>	Especifica números negativos
<b>!</b>	Negação booleana
<b>~</b>	Complemento bit a bit
<b>++ (pré)</b>	Incremento pré-fixado. Primeiro é efetuado o incremento e depois a variável é avaliada. Exemplo: ++x
<b>++ (pós)</b>	Incremento pós-fixado. Primeiro a variável é avaliada e depois é efetuado o incremento. Exemplo: x++
<b>-- (pré)</b>	Decremento pré-fixado. Primeiro é efetuado o decremento e depois a variável é avaliada. Exemplo: --x
<b>-- (pós)</b>	Decremento pós-fixado. Primeiro a variável é avaliada e depois é efetuado o decremento. Exemplo: x--

<b>Operadores Relacionais</b>	
<b>Operador</b>	<b>Uso</b>
<b>&lt;</b>	Condição "menor que" para tipos numéricos
<b>&gt;</b>	Condição "maior que" para tipos numéricos
<b>&gt;=</b>	Condição "maior ou igual que" para tipos numéricos
<b>&lt;=</b>	Condição "menor ou igual que" para tipos numéricos
<b>is</b>	Compara em tempo de execução se um objeto é compatível como um tipo qualquer. Esse assunto será abordado mais adiante
<b>as</b>	Efetua mascaramento de tipos e caso este falhe, o resultado da operação será null. Esse assunto será abordado mais adiante

<b>Operadores de Igualdade</b>	
<b>Operador</b>	<b>Uso</b>
<b>==</b>	Avalia a igualdade de dois tipos
<b>!=</b>	Avalia a desigualdade de dois tipos

<b>Operadores Condicionais</b>	
<b>Operador</b>	<b>Uso</b>
<b>&amp;&amp;</b>	Operador lógico AND usado para comparar expressões booleanas
<b>  </b>	Operador lógico OR usado para comparar expressões booleanas
<b>^</b>	Operador lógico XOR usado para comparar expressões booleanas

<b><i>Operadores de Atribuição</i></b>	
<b>Operador</b>	<b>Uso</b>
<b>=</b>	Atribuição simples. No caso de atribuição entre objetos, referências são atribuídas e não valores
<b>*=</b>	Multiplicação seguida de atribuição. Exemplo: $x*= 10$ que é equivalente a $x = x*10$
<b>/=</b>	Divisão seguida de atribuição. Exemplo: $x/= 10$ que é equivalente a $x = x/10$
<b>%=</b>	Resíduo seguido de atribuição. Exemplo: $x%= 10$ que é equivalente a $x = x \% 10$
<b>+=</b>	Soma seguida de atribuição. Exemplo: $x += 10$ que é equivalente a $x = x + 10$
<b>-=</b>	Subtração seguida de atribuição. Exemplo: $x -= 10$ que é equivalente a $x = x - 10$

## Variáveis

Nomeando uma variável:

A documentação do Microsoft .Net Framework dá as seguintes recomendações para a nomeação das variáveis:

- Evite usar underline;
- Não crie variáveis que apenas se diferenciem apenas pela sua forma. Exemplo: *minhaVariavel* e outra chamada *MinhaVariavel*;
- Procure iniciar o nome com uma letra minúscula;
- Evite usar todas as letras maiúsculas;
- Quando o nome tiver mais que uma palavra, a primeira letra de cada palavra após a primeira deve ser maiúscula (conhecido como notação camelCase);

### Convenção PascalCasing

Para usar a convenção PascalCasing para nomear suas variáveis, capitalize o primeiro caractere de cada palavra. Exemplo:

```
void InitializeData();
```

A Microsoft recomenda usar o PascalCasing quando estiver nomeando classes, métodos, propriedades, enumeradores, interfaces, constantes, campos somente leitura e namespaces.

### Convenção camelCasing

Para usar esse tipo de convenção, capitalize a primeira letra de cada palavra menos da primeira. Como o exemplo:

```
int loopCountMax;
```

A Microsoft recomenda usar essa convenção na nomeação de variáveis que definem campos e parâmetros.

Para maiores informações sobre convenção de nomes pesquise "Naming Guidelines", na documentação do Visual Studio.

### Palavras reservadas:

A linguagem C# reserva setenta e cinco palavras para seu próprio uso. Estas palavras são chamadas de palavras reservadas e cada uma tem um uso particular. Palavras reservadas também não são permitidas como nome de variáveis. Segue uma lista que identifica todas estas palavras:

abstract	as	base	Bool
break	byte	case	Catch
char	checked	class	Const
continue	decimal	default	Delegate
do	double	else	Enum
event	explicit	extern	false
finally	fixed	float	for
foreach	goto	if	implicit
in	int	interface	internal
is	lock	long	namespace
new	null	object	operator
out	override	params	private



protected	public	readonly	ref
return	sbyte	sealed	short
sizeof	stackalloc	static	string
struct	switch	this	throw

Lista de palavras reservadas

No painel de código do Visual Studio .NET as palavras reservadas são identificadas pela cor de letra azul.

## Declarando variáveis

Antes de usar uma variável é necessário declará-la. Neste momento alocamos espaço para esta variável na memória e dizemos que tipo de dado pode ser armazenado nela. O tipo de dado indica qual o tamanho do espaço vai ser reservado para a variável.

O C# pode armazenar diferentes tipos de dados: como inteiros, números de ponto flutuante, textos e caracteres. Assim que declaramos uma variável precisamos identificar que tipo de dado ela armazenará.

Declaramos especificando o tipo de dado seguido do nome da variável como no exemplo:

```
int contador;
```

Esse exemplo declara uma variável chamada contador do tipo integer. Ou seja ela deverá armazenar números inteiros, mais a frente estudaremos melhor o que armazenar em cada tipo de dado.

Podemos também declarar múltiplas variáveis de uma vez, fazemos isso da seguinte maneira:

```
int contador, numeroCarro;
```

Estamos declarando nesse exemplo duas variáveis do tipo int, uma chamada contador e a outra numeroCarro.

## Atribuindo valor a variáveis

Depois de declarar sua variável você precisa atribuir um valor a ela. No C# você não pode usar uma variável antes de colocar um valor nela, isso gera um erro de compilação.

Exemplo de como atribuir um valor a uma variável:

```
int numeroFuncionario;  
  
numeroFuncionario = 23;
```

Primeiro nos declaramos nossa variável do tipo integer. Depois atribuímos o valor 23 a ela. Entendemos pelo sinal de igual como recebe. Assim numeroFuncionario recebe 23.

Podemos também atribuir um valor a variável quando a declaramos, dessa forma:

```
int numeroFuncionario = 23;
```

Isso faz a mesma coisa que o exemplo anterior, só que tudo em uma linha.

Mais um exemplo:

```
char letraInicial = 'M';
```

## Tipos de variáveis

A seguinte tabela mostra os tipos do C# com sua referencia no Framework.

Os tipos da tabela abaixo são conhecidos como tipos internos ou Built-in.

C# Type	.NET Framework type
bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
object	System.Object
short	System.Int16
ushort	System.UInt16
string	System.String

Cada tipo no C# é um atalho para o tipo do Framework. Isso quer dizer que se declararmos a variável desta forma:

```
string nome;
```

ou dessa forma

```
System.String nome;
```

teremos o mesmo resultado. O atalho serve apenas para facilitar na hora de desenvolver a aplicação.

A seguinte tabela mostra os tipos de variáveis e os valores possíveis de se armazenar em cada uma delas.

C# Type	Valores possíveis de se armazenar
bool	Verdadeiro ou Falso (Valores booleandos)
byte	0 a 255 (8 bits)
sbyte	-128 a 127 (8 bits)
char	Um caractere (16 bits)

decimal	$\pm 1.0 \times 10^{-28}$ a $\pm 7.9 \times 10^{28}$ (128 bits)
double	$\pm 5.0 \times 10^{-324}$ a $\pm 1.7 \times 10^{308}$ (64 bits)
float	$\pm 1.5 \times 10^{-45}$ a $\pm 3.4 \times 10^{38}$ (32 bits)
int	-2,147,483,648 a 2,147,483,647 (32 bits)
uint	0 a 4,294,967,295 (32 bits)
long	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807 (64 bits)
ulong	0 a 18,446,744,073,709,551,615 (64 bits)
object	Qualquer tipo.
short	-32,768 a 32,767 (16 bits)
ushort	0 a 65,535 (16 bits)
string	Seqüência de caracteres (16 bits por caractere)

Todos os tipos na tabela com exceção dos tipos object e string são conhecidos como tipos simples.

Para retornar o tipo de qualquer variável do C# você pode usar o método GetType(); Como no exemplo:

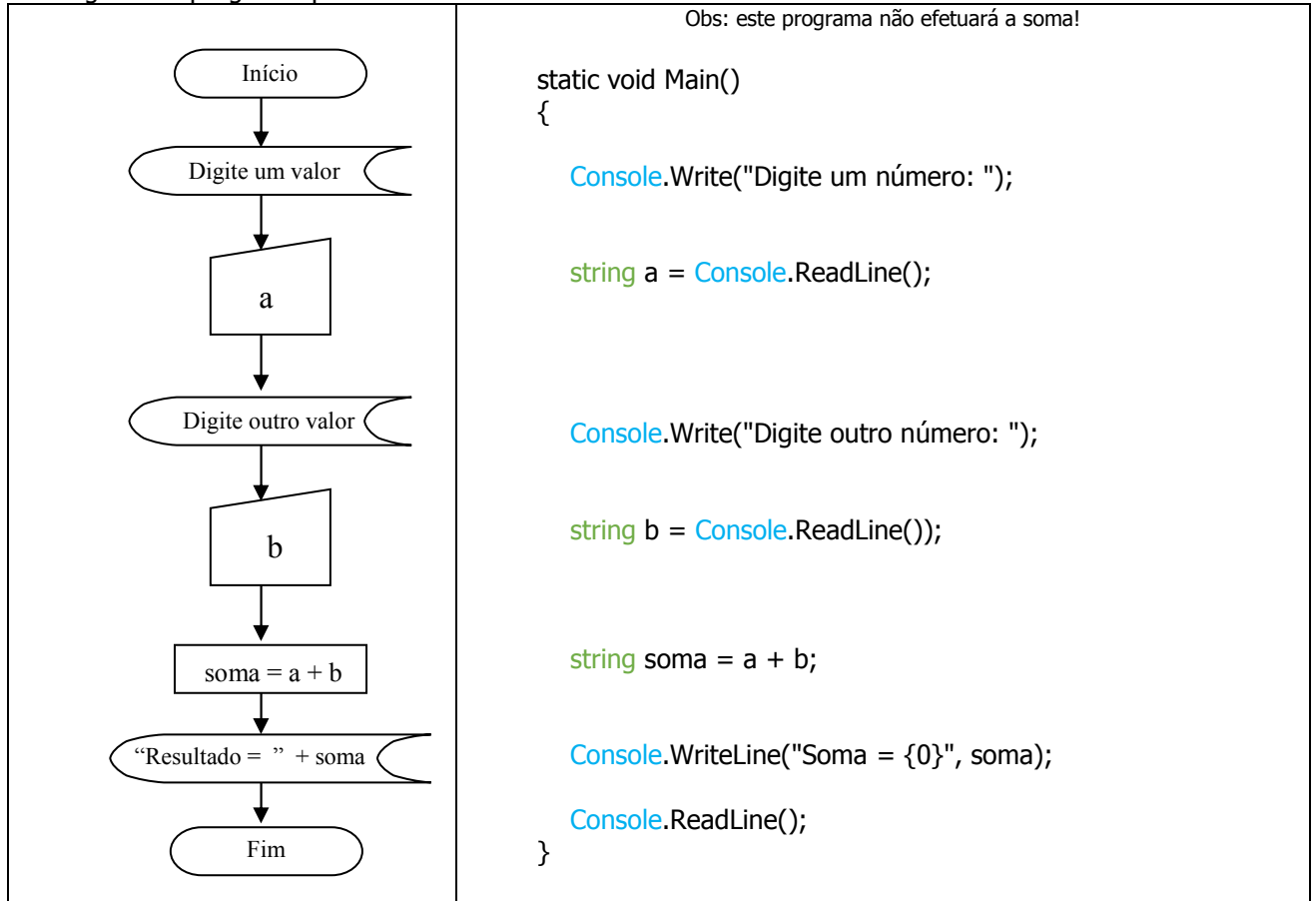
```
Console.WriteLine(minhaVariavel.GetType());
```

Isso retornaria o tipo da variável minhaVariavel.

Para maiores informações sobre tipos de variáveis consulte a documentação do Visual Studio por "data types".

## Manipulando valores numéricos

Fluxograma e programa para somar dois números e exibir o resultado.



O programa acima tem por objetivo somar dois números e exibir o resultado. Porém, o programa traduzido para C# não irá realizar a operação aritmética.

Imagine que o usuário informe os seguintes valores:

a= "7"

b = "5"

soma = "75"

O correto seria o resultado ser 12, porém ele concatenou os valores no lugar de somá-los.

Vamos ter que trabalhar com valores numéricos para que seja possível efetuar as operações aritméticas.

## Conversões de tipos

C#, Java, Pascal, C, C++ são exemplos de linguagens fortemente tipadas, onde a declaração do tipo é obrigatória. Todas as variáveis têm um tipo específico que tem que ser explicitado na criação da mesma.

EX:

```
int a = 7;  
String nome;  
double salario;  
char letra;
```

Isso implica que para realizar operações as variáveis devem ser de mesmo tipo ou, pelo menos, tipos compatíveis.

EX:

```
int a, b, c;  
  
a = 7;  
b = 5;  
c = a + b;
```

Na operação acima, o valor de “c” será 12 pois o tipo de dado utilizado foi o int (inteiro) e a operação irá realizar a soma entre os valores.

O exemplo a seguir também está correto, pois apesar dos tipos de dado diferentes (int e double), eles são numéricos e existe uma compatibilidade entre eles.

```
int a = 7, b = 5;  
double c;  
  
c = a + b;
```

O tipo de dado double serve para armazenar números inteiros e também fracionados, portanto ele comporta valores armazenados em uma variável do tipo int.

Agora, como vimos anteriormente, o tipo de dado string até armazena números, mas eles são representados como texto e, portanto, ao somarmos eles serão concatenados. Ex:

```
string a, b, c;  
a = "7";  
b = "5";  
c = a + b;
```

No exemplo acima, o valor de “c” será “75”, afinal de contas eles são strings e sua soma resulta nos valores concatenados. Mas, e se precisarmos que o resultado seja 12? Neste caso, precisaremos converter os valores antes de efetuar a operação. O resultado da operação também será atribuído a uma variável do tipo int.

```
string a, b;  
int c;  
a = "7";  
b = "5";  
c = Convert.ToInt32(a) + Convert.ToInt32(b);
```

Agora, vamos analisar o seguinte exemplo: (o exemplo abaixo está incompleto):

```
string a, b, c;
a = "7";
b = "5";
c = Convert.ToInt32(a) + Convert.ToInt32(b);
```

A classe Convert possui diversos métodos para realizar conversões. ToInt32() é um método "tenta" converter "algo" para um inteiro de 32 bits. "Tenta" por que pode ser que não seja possível converter. Por exemplo, o string "7" pode ser convertido para o inteiro 7, já o string "Corinthians" não pode ser convertido para nenhum número inteiro (embora seja 10 ☺).

Entre parênteses vai o que você quer converter! Convert.ToInt32( )  
-----^

O caso é que o exemplo acima ainda contém um erro. Sabe qual é?

O problema é que, como dissemos no início do tópico, C# é uma linguagem fortemente tipada, o que significa que se você está fazendo uma atribuição, os tipos de dados devem ser compatíveis, e neste caso "string" e "int" não são.

inteiro

c = Convert.ToInt32(a) + Convert.ToInt32(b);

string

Sendo assim, precisaremos converter o resultado da operação aritmética para string antes de atribuir para "c".

Para simplificar o processo, vamos criar uma nova variável (soma) que irá armazenar primeiro o resultado da soma, para então a convertermos para string:

```
string a, b, c;
a = "7";
b = "5";
int soma = Convert.ToInt32(a) + Convert.ToInt32(b);
c = Convert.ToString(soma);
```

OBS: Para converter para string, há um jeito mais fácil! No lugar de usar o Convert.ToString(), pode ser colocado .ToString() na frente das variáveis. EX:

```
c = soma.ToString();
```

Para quem conhece Java, há outras formas de converter dados em C# que lembra muito o Java. EX:

```
int numero = int.Parse("7");
```

Aqui estamos convertendo valores fixos. Agora, imagine que seja necessário converter um valor digitado pelo usuário. Ex:

```
int a = Console.ReadLine();
```

O exemplo acima irá gerar um erro de compilação já que o inteiro (a) e o string (console.ReadLine() ) são incompatíveis. Lembre-se: o **Console.ReadLine()** é usado para ler uma informação que o usuário digitou e **ele sempre irá lhe devolver essa informação no formato string**.

Para converter, utilize o seguinte código:

```
int a = Convert.ToInt32( Console.ReadLine() );
```

Segue abaixo um exemplo completo de um programa que soma dois números e exibe o resultado:

```
static void Main(string[] args)
{
    Console.Write("Digite um número: ");
    int a = Convert.ToInt32( Console.ReadLine() );

    Console.Write("Digite outro número: ");
    int b = Convert.ToInt32( Console.ReadLine() );

    int soma = a + b;

    Console.WriteLine("Soma = {0}", soma);
    Console.ReadLine();
}
```

## Adicionando valor a uma variável

É muito comum precisarmos adicionar ou subtrair valores de uma variável usando no calculo o valor que já esta armazenado na mesma.

O código seguinte declara uma variável do tipo *integer* chamada *contador* e armazena o valor 2 nesta variável, depois incrementa o valor 40:

```
int contador;

contador = 2;

contador = contador + 40;
```

No final do código acima a variável *contador* tem qual valor?

A resposta é 42, claro, criamos a variável, adicionamos o valor 2 nela e após, pegamos o valor dela (que era 2) e adicionamos 40, e armazenamos o valor na mesma.

Preste atenção na seguinte linha de código:

```
contador = contador + 40;
```

Perceba que para somar o valor a variável precisamos repetir o nome da variável.

Podemos fazer da seguinte forma também em C#:

```
contador += 40;
```

Isso teria o mesmo resultado e é uma maneira mais elegante.  
Você pode subtrair também valores, como o exemplo:

```
contador -= 23;
```

Isso subtrairia 23 do valor da variável.

Na verdade você pode fazer isso com todos os operadores aritméticos, como multiplicação e divisão também.

## Diferença entre ++VAR e VAR++

Ambos servem para incrementar em 1 o valor de uma variável numérica. Porém, existem diferenças no momento de testar o valor da variável. O mesmo vale para outros operadores, como -VAR ou VAR--.

EX:

```
int n = 4;
```

```
Console.Write( n++ ); // será exibido 4 na tela, pois primeiro a variável é "usada" para só depois ser incrementada.
```

```
int n = 4;
```

```
Console.Write( ++n ); // será exibido 5 na tela, pois primeiro a variável é incrementada para só depois ser "usada".
```





## Tipo de dado String

As variáveis do tipo string aceitam todos os tipos de caracteres, incluindo letras e números. Seu conteúdo é delimitado por aspas duplas.

As variáveis do tipo string são indexadas, que significa que você pode acessar os elementos da string utilizando para isso os caracteres [].

EX:

```
string nome = "ANA MARIA";
```

Caractere	A	N	A		M	A	R	I	A
Posição	0	1	2	3	4	5	6	7	8

Para exibir um determinado caractere do nome, faça:

```
Console.Write( nome[0] ); //será exibido A
Console.Write( nome[4] ); //será exibido M
Console.Write("Letras {0}{1}{2}", nome[0], nome[4], nome[8]); // AMA
```

## Concatenar

Concatenar é a operação de somar duas ou mais strings. EX:

```
string partel = "Brasil é o ";
string parte2 = "país do futebol";
string parte3 = partel + parte2;

Console.WriteLine(parte3); // Brasil é o país do futebol
```

```
string partel = "8";
string parte2 = "2";

Console.WriteLine(partel + parte2); // será exibido 82
```

## Tipo de dado Char

Uma variável do tipo `char` aceita apenas 1 caractere e seu conteúdo deve ser delimitado por aspas simples ('). Uma variável do tipo char sempre deve ter um valor, mesmo que seja vazio (um espaço).

EX: `char letra = 'A';`

Cada posição da string é do tipo char. Sendo assim, pode-se fazer algo do tipo:

```
char letra = nome[0];
Console.WriteLine("Letra = {0}", letra); // será exibido Letra = A
```

## Propriedades e Métodos das Strings

As variáveis do tipo string possuem uma série de características (propriedades) e comandos (métodos) que podem ser executadas sobre ela. EX:

### Propriedade Length

- Esta propriedade retorna um número inteiro que indica a quantidade de caracteres (incluindo espaços) que há na variável string.

EX:

```
string nome = "Ana Silvia"; // possui 10 caracteres
//           0123456789

int quantidade = nome.Length;

Console.Write("O nome possui {0} caracteres",
             quantidade); // será exibido 10

Console.Write("Última letra do nome {0}",
             nome[quantidade -1] ); // será exibido: a
```

### Método Substring

Permite copiar uma parte de uma string. O método tem várias variantes, as principais são:

**Substring (int inicio)** - copia o texto todo a partir da posição inicial informada.

**Substring (int inicio, int quantidade)** - copia uma determinada quantidade de caracteres do texto a partir da posição inicial informada.

EX:

```
string nome = "Ana Silvia"; // possui 10 caracteres
//           0123456789

//significa: copie a partir da posição Zero, 3 caracteres
//o que resultará em Ana
string primeiroNome = nome.Substring(0, 3);
Console.WriteLine(primeiroNome); // será exibido: Ana

//significa: copie tudo a partir da posição 4
Console.WriteLine( nome.Substring(4) ); // será exibido: Silvia
```

## Método IndexOf

Retorna a posição (inteiro) de um determinado valor char ou string dentro de uma string. Caso ele não encontre o que está procurando será devolvido -1.

EX:

```
string nome = "Ana Silvia"; // possui 10 caracteres
//           0123456789

Console.WriteLine(nome.IndexOf('n')); // será exibido 1
Console.WriteLine(nome.IndexOf('a')); // será exibido 2
Console.WriteLine(nome.IndexOf('A')); // será exibido 0
Console.WriteLine(nome.IndexOf(' ')); // será exibido 3
Console.WriteLine(nome.IndexOf('Z')); // será exibido -1

// observe que ao procurar por string deve-se delimitar usando "
Console.WriteLine(nome.IndexOf("Si")); // será exibido 4
```

## Método Replace

O método replace substitui todas as ocorrências de um determinado valor char ou string por outro char/string dentro de uma string.

EX:

```
string data = "25-10-2012";
Console.WriteLine( data.Replace('-', '/') ); // será exibido 25/10/2012
Console.WriteLine(data); // será exibido 25-10-2012
```

O comando acima não alterou a data original, apenas devolveu uma nova data com os caracteres trocados. Se o seu objetivo for alterar a data original, deve se fazer assim:

```
data = data.Replace('-', '/');
Console.WriteLine(data); // será exibido 25/10/2012
```

```
string CPF = "123.456.789-09";

CPF = CPF.Replace(".", "");
CPF = CPF.Replace("-", "");
Console.WriteLine(CPF); // será exibido 12345678909
```

Observe que acima foi utilizada uma string "." e não o char '.'. Isso foi feito já que se desejava substituir o "." por nada ("") e neste caso uma variável do tipo char não aceita vazio.

O comando acima também poderia ser feito assim:

```
CPF = CPF.Replace(".", "").Replace("-", "");
```

## Método Remove

O método remove apaga (remove) uma determinada parte da string, devolvendo uma versão modificada da string informada, sem a parte removida.

EX:

```
string data = "25-10-2012";
Console.WriteLine(data.Remove(6, 2)); // será exibido 25-10-12
Console.WriteLine(data); // será exibido 25-10-2012
Console.WriteLine(data.Remove(2)); // será exibido 25
```

O comando acima não alterou a data original. Se o seu objetivo for alterar a data original, deve se fazer assim:

```
data = data.Remove(6, 2);
Console.WriteLine(data); // será exibido 25-10-12
```

## Console.ReadLine()

Este comando é o responsável por solicitar que o usuário digite alguma informação. Toda informação devolvida por ele é do tipo string, mesmo que o usuário digite apenas números. Seu equivalente no fluxograma é o símbolo:



EX:

```
Console.Write("Por favor, informe seu nome completo: ");
string nome = Console.ReadLine();
```



O comando `Console.ReadLine()` irá aguardar o usuário digitar algo e pressionar ENTER. Então, ele irá retornar o valor lido e irá atribuir este valor na variável nome.

## Exemplo completo:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```

public class StringApp
{
    static void Main()
    {
        string nome;
        char letra;

        Console.Write("Por favor, informe seu nome completo: ");
        nome = Console.ReadLine();

        Console.WriteLine("Seu nome tem {0} letras ", nome.Length);
        Console.WriteLine("Ele inicia com a letra [ {0} ] e termina com [ {1} ]",
            nome[0], nome[nome.Length - 1]);
        Console.WriteLine("Nome todo em maiúsculo: {0} ", nome.ToUpper());
        Console.WriteLine("Nome todo em minúsculo: {0} ", nome.ToLower());
        Console.WriteLine("Apenas as 3 primeiras letras do nome: {0}", nome.Substring(0, 3));

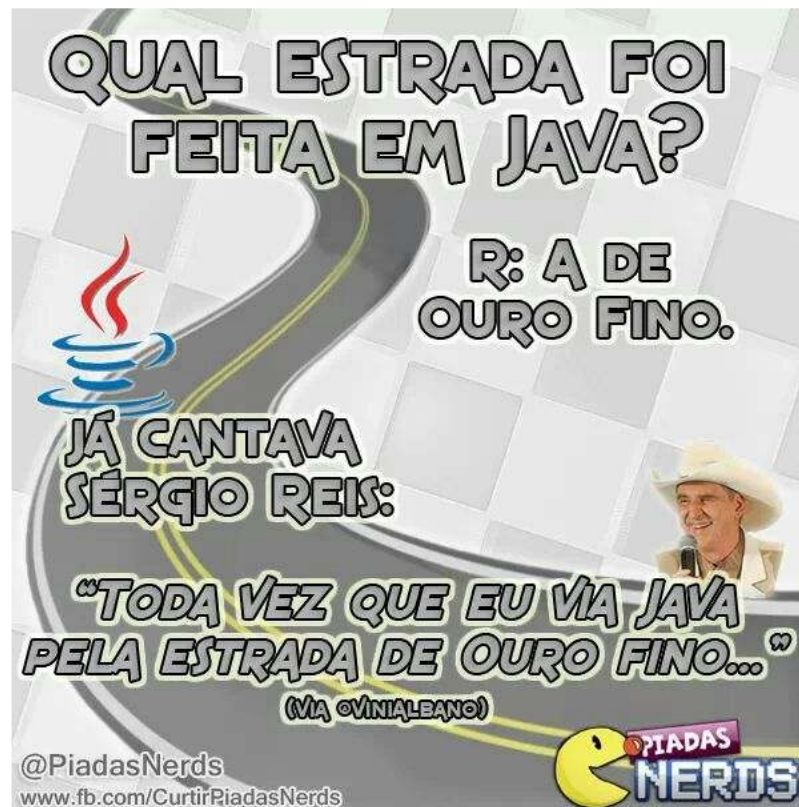
        // replace: substitui uma letra por outra
        // a partir de agora, o nome foi alterado!!!!
        nome = nome.Replace(' ', '_');
        Console.WriteLine("Veja como ficou seu nome depois de aplicar o replace: {0}", nome);

        Console.Write("Digite uma letra para pesquisar em seu nome: ");
        letra = Console.ReadLine()[0];

        // indexOf pesquisa a primeira ocorrência de um caractere dentro de uma string.
        // se não encontrar, retorna -1
        Console.WriteLine("A primeira ocorrência da letra [{0}] está na posição: {1} ",
            letra, nome.IndexOf(letra));

        Console.WriteLine("\n\nPressione ENTER para terminar.");
        Console.ReadLine();
    }
}

```



## Manipulando arquivos texto – parte 1

### Escrevendo no arquivo

Arquivos texto são arquivos de computador que podem ser lidos em qualquer editor de texto. Geralmente possuem a extensão .TXT. Exemplo: dados.txt, informações.txt e senhas.txt (!!!).

Arquivos texto podem ser lidos e escritos em C# de algumas maneiras, vamos explicar aqui a mais simples. Os dados em um arquivo texto são manipulados em C# como strings. Portanto, tudo lá é uma string e podemos utilizar os métodos que já vimos anteriormente para manipular este tipo de dado.

Veja o programa abaixo:

```
using System;
using System.Text;
using System.IO;

namespace ManipulandoArquivoTexto
{
    class Program
    {
        static void Main(string[] args)
        {
            string nome;
            int idade;

            Console.WriteLine("Digite seu nome: ");
            nome = Console.ReadLine();

            Console.WriteLine("Digite sua idade: ");
            idade = Convert.ToInt32(Console.ReadLine());

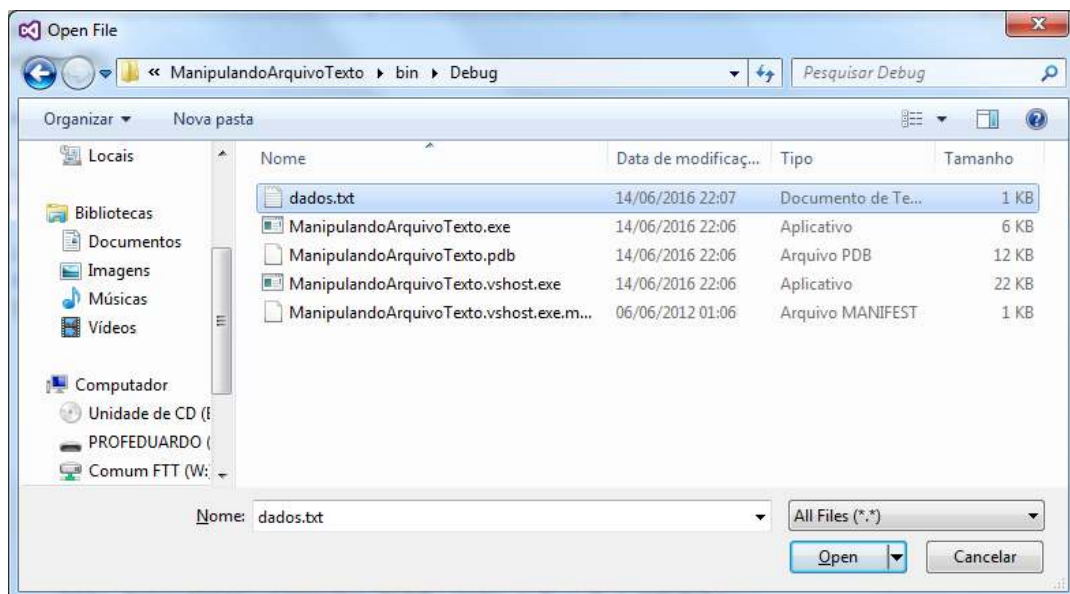
            File.AppendAllText("dados.txt", nome + " - " + idade);

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

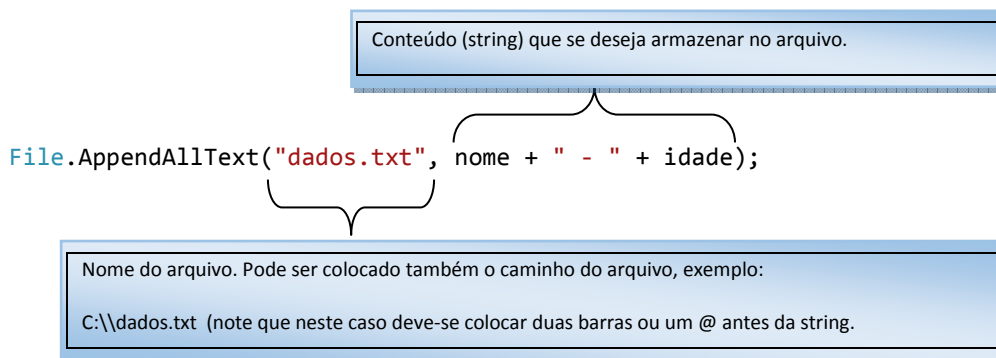
É necessário adicionar esta linha para que você possa utilizar classe `File`.

A classe `File` possui diversos métodos para manipular arquivo texto. O método `AppendAllText` escreverá em um arquivo. Se o arquivo já existir, ele irá adicionar nova informação no final do mesmo, caso contrário um novo arquivo será criado.

Execute o programa acima. Note que ao final de sua execução, um arquivo chamado dados.txt será criado na pasta onde o seu programa está sendo executado, veja o arquivo na imagem abaixo:

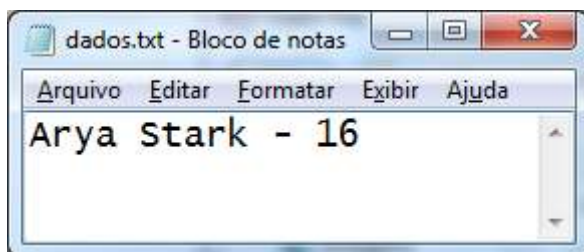


O arquivo poderá ser aberto em qualquer editor de texto (wordpad, notepad, notepad++, etc..). A sintaxe do comando AppendAllText é:



Se o arquivo não existir, o mesmo será criado. O `AppendAllText` não apaga o conteúdo do arquivo caso o mesmo já exista. Se quiser sobrescrever o conteúdo do arquivo, utilize o método `WriteAllText`. A Sintaxe é a mesma.

Exemplo do conteúdo do arquivo:



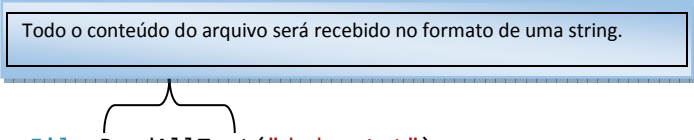
Obs: Para evitar problemas com acentuação, recomendo o uso de um padrão de codificação (Encoding). Mais detalhes podem ser consultados em: [https://msdn.microsoft.com/pt-br/library/system.text.encoding\(v=vs.110\).aspx](https://msdn.microsoft.com/pt-br/library/system.text.encoding(v=vs.110).aspx)

```
string conteudo = nome + " - " + idade;

File.AppendAllText("dados.txt", conteudo, Encoding.UTF8);
```

## Lendo os dados do Arquivo texto.

Para ler os dados do arquivo, também existem diversas formas. Iremos abordar aqui o formato mais simples, onde iremos receber o conteúdo do arquivo no formato de uma string:



Todo o conteúdo do arquivo será recebido no formato de uma string.

```
string texto = File.ReadAllText("dados.txt");  
Console.WriteLine("Conteúdo do arquivo: {0}", texto);
```

Como não foi informado o local do arquivo dados.txt, o sistema irá procurar na mesma pasta onde o programa está sendo executado.

Se necessário, é possível informar a pasta:

```
string texto = File.ReadAllText("c:\\dados.txt", Encoding.UTF8);
```

Também é possível utilizar o encoding na leitura do arquivo:

```
string texto = File.ReadAllText("dados.txt", Encoding.UTF8);
```



## Constantes

Ao antepor a palavra-chave **const** durante a declaração e inicialização de uma variável, ela se torna uma constante. Como insinua o nome, um constante é uma variável cujo valor não pode ser alterado ao longo de sua existência:

```
const int A = 100; // Este valor não pode ser alterado em tempo de execução
```

- Procure criar constantes sempre em CAIXA ALTA

As constantes têm as seguintes características:

- Elas devem ser inicializadas quando declaradas e, depois que um valor lhes for atribuído, não poderá mais ser alterado.
- O valor de uma constante deve ser computável em tempo de compilação. Então, nós não podemos inicializar uma constante com um valor proveniente de uma variável. Se você precisar fazer isso, use um campo somente leitura.
- As constantes são sempre estáticas. Note, porém, que não temos (e, na realidade, não é permitido) incluir o modificador **static** na declaração de constante.

Há três vantagens pelo menos no uso de constantes (ou variáveis somente leitura) em seus programas:

- As constantes tornam seus programas mais fáceis de ler, substituindo "números mágicos" e "strings mágicas" por nomes legíveis cujos valores são fáceis de entender.
- As constantes tornam seus programas mais fáceis de modificar. (Por exemplo, você tem uma constante Impostovenda em um de seus programas C# e a essa constante é atribuído o valor de 6%. Se a taxa de imposto de Impostovenda mudar tempos depois, você pode modificar o comportamento de todos os cálculos de imposto simplesmente atribuindo um valor novo à constante; você não tem de procurar em todo o seu código o valor 0.06 e modificar cada um, supondo que tivesse encontrado todos.)
- Com as constantes é mais fácil evitar erro sem seus programas. Se você tentar atribuir outro valor a uma constante em algum lugar em seu programa, depois que já tiver atribuído um valor, o compilador sinalizará o erro.

## Saindo explicitamente da aplicação:

A instrução abaixo faz com que o programa termine e retorne ao sistema operacional o código de erro informado entre parênteses.

```
Environment.Exit(0);
```

Exemplo:

```
static void Main(string[] args)
{
    Console.WriteLine("O comando Environment.Exit(0) fecha o programa imediatamente");
    Console.ReadLine();
    Environment.Exit(0); // 0 (zero) é o código de erro passado para o Sist. Oper.
    Console.WriteLine("Esta linha nunca será exibida");
}
```

Você também pode usar o comando **return** dentro do método Main para sair da aplicação.

Exemplo:

```
static void Main(string[] args)
{
    return; // mais sobre o comando return será explicado no tópico de Métodos!
    Console.WriteLine("Esta linha nunca será exibida");
}
```

## Estrutura de Decisão (IF)

O **if** avalia uma expressão lógica booleana e qualquer outro tipo será acusado como erro pelo compilador. Se o resultado for verdadeiro, o bloco de código dentro do if será executado; caso contrário, o controle é passado para a próxima declaração após o if. Os projetistas de C# optaram por aceitar unicamente expressões booleanas no if para evitar escrever código com semântica obscura e propensa a resultados inesperados.

A declaração **if** tem três formas básicas:

1.

```
if (expressão booleana)
{
    Declaração
}
```

2.

```
if (expressão booleana)
{
    Declaração
}
else
{
    Declaração
}
```

3.

```
if (expressão booleana)
{
    Declaração
}
else if (expressão booleana)
{
    Declaração
}
else if (expressão booleana)
{
    Declaração
}
else
{
    Declaração
}
```

Vejamos alguns exemplos:

```
static void Main(string[] args)
{
    int a = 0;
    int b = 0;
    if (a < b)
    {
        Console.WriteLine("B é maior");
    }
    else
    {
        Console.WriteLine("A é maior");
    }
}
```

O exemplo anterior é sintaticamente correto, mas o que aconteceria se  $a=b$ ? O nosso código passaria a dar a resposta incorreta, porque ele não avalia a condição  $a=b$ , o que torna o nosso código inconsistente. Vamos reescrevê-lo de forma que a condição de igualdade seja avaliada:

```

static void Main(string[] args)
{
    int a = 0;
    int b = 0;
    if (a < b)
    {
        Console.WriteLine("B é maior");
    }
    else if (a > b)
    {
        Console.WriteLine("A é maior");
    }
    else
    // e finalmente a condição de igualdade deveria ser satisfeita
    {
        Console.WriteLine("A é igual a B");
    }
}

```

O uso de chaves {} é opcional e está condicionado a execução de mais de uma instrução. No exemplo abaixo, poderemos retirar as chaves, pois caso a condição do if seja verdadeira, apenas uma instrução deverá ser executada. O mesmo acontece com o else.

```

static void Main(string[] args)
{
    int a = 0;
    int b = 0;
    if (a < b)
        Console.WriteLine("B é maior");
    else
        Console.WriteLine("A é maior");
}

```

Veja o código abaixo. Tente escrever essa sequência de if's de outra forma que produza o mesmo resultado.

```

double n1 = Convert.ToDouble( Console.ReadLine());
double n2 = Convert.ToDouble(Console.ReadLine());
double media = (n1 + n2) / 2;

if (media < 3)
    Console.WriteLine("Reprovado.");
else
{
    if (media >= 3 && media < 7)
        Console.WriteLine("Recuperação");
    else
        Console.WriteLine("Aprovado");
}

```

Atenção ao comando If. Como ele se baseia em expressões lógicas, ele não precisa avaliar a expressão inteira para decidir se executará ou não a instrução.

Se na expressão forem utilizados operadores lógicos E (&&) e se o primeiro elemento da expressão resultar em false, os demais não serão avaliados. EX:

```
int a = 7;
int b = 5;
int c = 10;
```

```
if ( a == 8 && a > b && c > b )
{
    Console.WriteLine("Teste");
}
```

Como **a** é diferente de 8, as demais expressões sequer serão avaliadas e o if não será executado. Isso por que foi utilizado o operador E onde todas as proposições lógicas DEVEM ser verdadeiras para que o resultado seja verdadeiro.

$F \text{ e } V \text{ e } V \Rightarrow F$

```
int a = 7;
int b = 5;
int c = 10;
```

```
if ( a == 8 || a > b || c > b )
{
    Console.WriteLine("Teste");
}
```

Apesar de **a** ser diferente de 8, as demais expressões ainda precisam ser avaliadas pois foi utilizado o operador OU, onde basta uma das proposições ser verdadeira para que o resultado seja verdadeiro. Neste caso, o if será executado.

$F \text{ ou } V \text{ ou } V \Rightarrow V$

## Indentação

Origem: Wikipédia, a enciclopédia livre. <http://pt.wikipedia.org/wiki/Indenta%C3%A7%C3%A3o>

Outras fontes de consulta:

<http://www.criarprogramas.com/2011/04/indentao-do-codigo-fonte-boas-prticas-de-programao/>

<http://trialforce.nostalgia.eng.br/?p=450>

Em ciência da computação, indentação (reco, neologismo derivado da palavra em inglês indentation, também encontram-se as formas idantação e endentação[1]) é um termo aplicado ao código fonte de um programa para indicar que os elementos hierarquicamente dispostos têm o mesmo avanço relativamente à posição (x,0).

Na maioria das linguagens a indentação tem um papel meramente estético, tornando a leitura do código fonte muito mais fácil (read-friendly), porém é obrigatória em outras. Python, Occam e Haskell, por exemplo, utilizam-se desse recurso tornando desnecessário o uso de certos identificadores de blocos ("begin" e/ou "end").

A verdadeira valia deste processo é visível em arquivos de código fonte extensos, não se fazendo sentir tanto a sua necessidade em arquivos pequenos (relativamente ao número de linhas). Para qualquer programador, deve ser um critério a ter em conta, principalmente, por aqueles que pretendam partilhar o seu código com outros. A indentação facilita também a modificação, seja para correção ou aprimoramento, do código fonte.

Existem centenas de estilos de indentação, mas, basicamente, consiste na adição de tabulações no início de cada linha na quantidade equivalente ao número de blocos em que cada linha está contida.

O visual studio identa automaticamente pressionando-se CTRL+E+D ou CTRL +K+D. Exemplos:

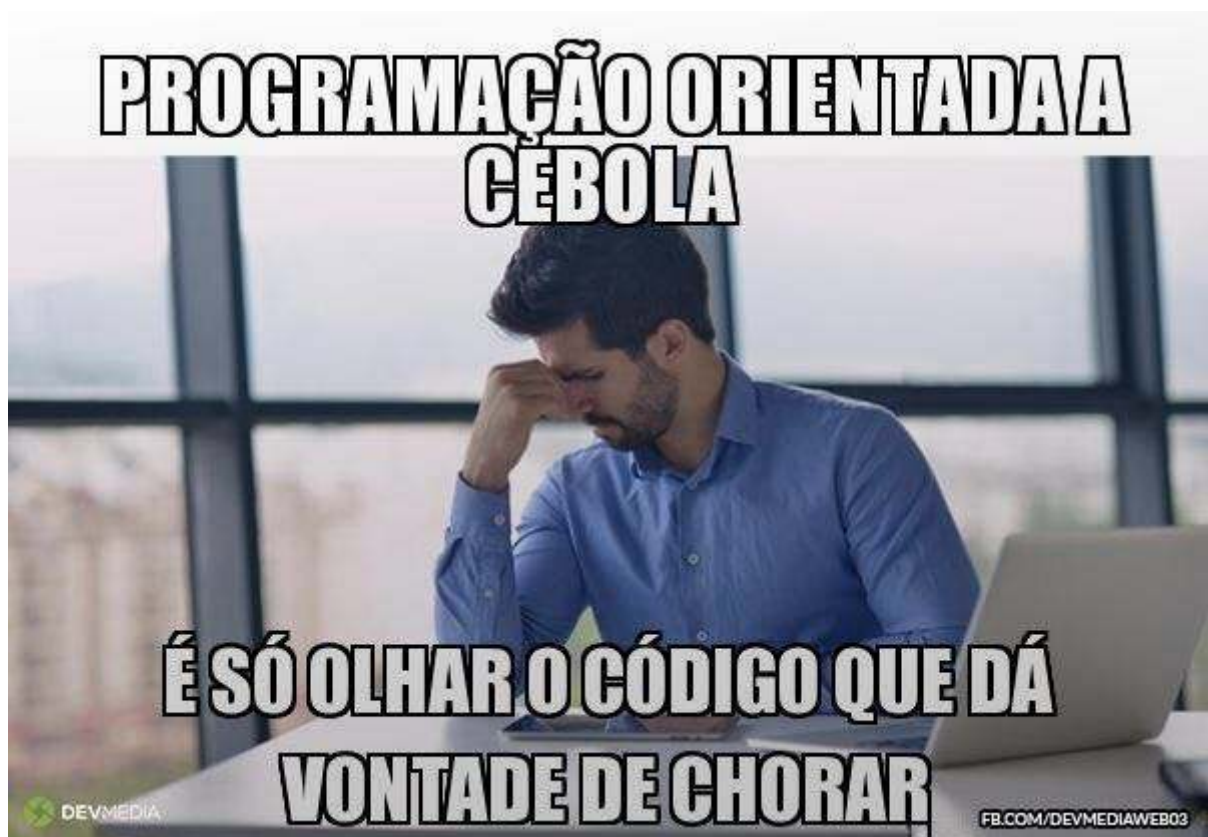
```
static void Main(string[] args)
{
    double n1, n2;
    Console.WriteLine("Digite a nota 1: ");
    n1 = Convert.ToDouble(Console.ReadLine());

    Console.WriteLine("Digite a nota 2: ");
    n2 = Convert.ToDouble(Console.ReadLine());

    double media = (n1 + n2) / 2;

    if (media < 4)
    {
        Console.WriteLine("Reprovado!");
    }
    else if (media < 7)
    {
        Console.WriteLine("Recuperação!");
    }
    else
    {
        Console.WriteLine("Aprovado!");
    }
}
```

```
static void Main(string[] args)
{
    string texto = Console.ReadLine();
    int soma = 0;
    char x;
    for (int n = texto.Length - 1; n >= 0; n--)
    {
        x = texto.ToUpper()[n];
        if (x == 'A' || x == 'E')
            soma += 10;
        soma++;
        if (x == 'I' || x == 'O')
        {
            soma = soma + 20;
            if (n < texto.Length - 2)
                soma = soma + (soma % 2);
        }
        if (x == 'U')
            soma--;
    }
    Console.WriteLine(soma);
    Console.ReadLine();
}
```



## Depuração de Programas

A depuração é o processo de executar um programa passo a passo (linha a linha), sendo possível visualizar (e até alterar) os valores das variáveis durante o processo. Este processo é muito interessante para um melhor entendimento do algoritmo e para encontrar problemas.

Links interessantes sobre o assunto:

<http://pt.wikipedia.org/wiki/Depura%C3%A7%C3%A3o>

<http://pt.wikipedia.org/wiki/Depurador>

Para depurar um programa no Visual Studio, utilize as teclas:

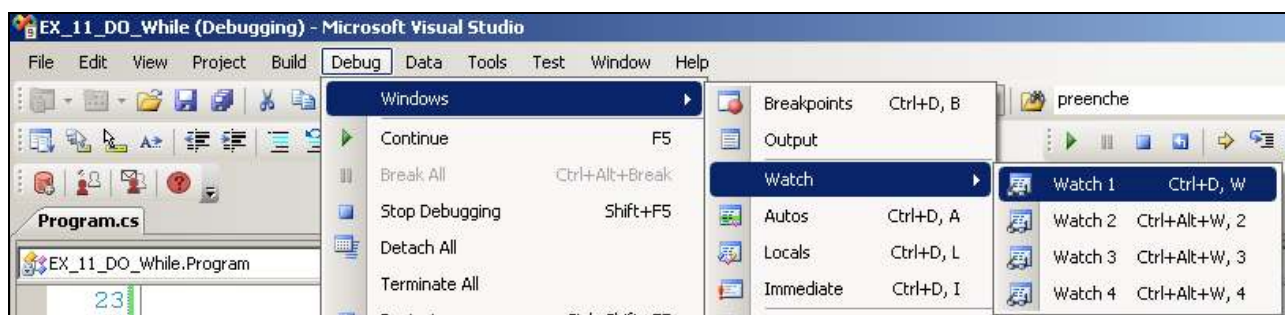
F10 – Executa o programa linha a linha, mas não entra nos métodos (porém os executa)

F11 – Executa o programa linha a linha, mas entrando nos métodos

F9 – Coloca um Break Point na linha selecionada.

F5 – Executa o programa

Durante o processo de depuração, é possível analisar o valor das variáveis. Para isso, você deve deixar o mouse sobre uma variável, ou utilizar as janelas de **watch**: Para exibir a janela de watch, comece a depurar o programa e acesse a opção Watch 1 no menu:



A janela Watch irá aparecer na parte de baixo do visual Studio.

A tecla F9 permite inserir um **Break Point** na linha. Isso significa que o programa que não esteja sendo depurado entrará em modo de depuração assim que a execução chegar na linha com o break point:

```

25
26     Console.WriteLine("Digite um número:");
27     int numero = Convert.ToInt32(Console.ReadLine());
28
29     int contador = 0;
30     do
31     {
32         Console.WriteLine("{0}x{1}={2}", numero, contador,
33             contador++);
34     }
35     while (contador <= 10);
36
37     Console.ReadLine();
38 }
```

## Estrutura de Repetição FOR

O laço **for** segue o mesmo estilo das linguagens C/C++, e a sua sintaxe tem as seguintes características:

- Uma variável de inicialização pode ser declarada dentro do for.
- Uma condição avalia uma expressão para abandonar o for ou executá-lo de novo.
- Uma expressão incrementa o valor da variável de inicialização.

Exemplo:

```
static void Main(string[] args)
{
    for (int i = 0; i < 5; i++)
    {
        Console.WriteLine("Iteração número {0}", i);
    }
    Console.ReadLine();
}
```

Para abandonar o laço antes que a condição for seja falsa, usa-se a palavra reservada **break**.

Exemplo:

```
static void Main(string[] args)
{
    for (int i = 0; i < 5; i++)
    {
        Console.WriteLine("Iteração número {0}", i);
        if (i == 3)
            break;
    }
    Console.ReadLine();
}
```

A palavra reservada **continue** permite que o fluxo de execução da iteração corrente seja abandonado, mas não o laço, e a iteração seguinte dê início no topo do laço, uma vez que a condição do for seja satisfeita.

Exemplo:

```
static void Main(string[] args)
{
    for (double i = 0; i < 5; i++)
    {
        if (i == 3)
            continue;

        // a linha abaixo não será impressa quando i = 3
        Console.WriteLine("Iteração número {0}", i);
    }
    Console.ReadLine();
}
```



## Laços for infinitos

Veja no exemplo a seguir uma forma de usar laços for para implementar iterações infinitas:

```
static void Main(string[] args)
{
    string texto ;
    Console.WriteLine("Digite múltiplas linhas separadas por enter. " +
        "Para sair digite \"sair\" ");

    for (; ; )
    {
        texto = Console.ReadLine();
        if ((texto.ToUpper() == "SAIR"))
            break;
    }
}
```

Observe que o uso de **break** é necessário para poder abandonar o laço; caso contrário, o seu programa entrará num loop infinito.

## Laços for aninhados

Laços aninhados são laços dentro de laços. Nestes casos, não podemos usar a mesma variável de controle para ambos os FOR.

Exemplo:

```
static void Main(string[] args)
{
    for (int i = 0; i <= 10; i++)
    {
        for (int x = 0; x <= 20; x++)
        {
            Console.WriteLine("{0}, {1}", i, x);
        }
    }
    Console.ReadLine();
}
```

Se uma declaração **break** estiver no laço interno, este será abandonado e o controle será passado para o laço externo; mas se estiver no laço externo, os dois laços serão abandonados e o controle passará para a próxima declaração após o laço.

Tanto a expressão de inicialização de variáveis quanto a de incremento podem conter mais de uma expressão e estas deverão estar separadas por vírgula.

Exemplo:

```
static void Main(string[] args)
{
    int i, j;

    for (i = 0, j = 1; j < 5; j++, i++)
    {
        Console.WriteLine("i={0} j={1}", i, j);
    }
    Console.ReadLine();
}
```

## Laços for em decremento

Também podemos implementar laços cuja variável de controle decremente em lugar de incrementar.

Exemplo:

```
static void Main(string[] args)
{
    for (int i = 10; i >= 0; i--)
    {
        Console.WriteLine(i);
    }
    Console.ReadLine();
}
```

## Estrutura de Repetição DO WHILE

O laço **do while** é usado quando não sabemos o número de vezes que devemos executar um bloco de código, mas apenas a condição que deve ser satisfeita para executar o bloco dentro do **do while**. Essa condição é uma expressão booleana que deverá ser verdadeira para garantir a execução do bloco. Este tipo de laço é usado quando queremos que um bloco de código seja executado pelo menos uma vez, dado que a condição para sua repetição é testada após a execução do bloco.

Ex:

Vamos simular uma estrutura de repetição for utilizando o **do while**:

```
static void Main(string[] args)
{
    int contador = 0;

    do
    {
        Console.WriteLine(contador);
        contador++;
    }
    while (contador <= 10);

    Console.ReadLine();
}
```

No exemplo a seguir, o usuário deve digitar -1 para sair da estrutura de repetição:

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

    do
    {
        Console.WriteLine("Digite qualquer número ou -1 para sair");
        numero = Convert.ToInt32(Console.ReadLine());
    }
    while (numero != -1);

    Console.WriteLine("Tchau!");
    Console.ReadLine();
}
```

Exemplo de validação de valores digitados pelo usuário:

```
static void Main(string[] args)
{
    //O código abaixo faz uma validação na leitura de um número,
    //não permitindo que seja digitado um valor menor ou igual a zero

    int numero;
    do
    {
        Console.Write("Digite um número inteiro maior que zero:");
        numero = Convert.ToInt32(Console.ReadLine());
    }
    while (numero <= 0);

    Console.WriteLine("O número digitado foi: {0}", numero);
    Console.ReadLine();
}
```

Conforme mostramos no laço **for**, o uso das declarações **break** e **continue** também é permitido dentro do **do while** atendendo à mesma funcionalidade.

## Estrutura de Repetição WHILE

Assim como no laço **do while**, o laço **while** é usado quando não sabemos o número de vezes que devemos executar um bloco de código, mas apenas a condição que deve ser satisfeita para executar o bloco dentro do **while**. Essa condição é uma expressão booleana que deverá ser verdadeira para garantir a execução do bloco. Diferentemente do **do while**, se a condição não for satisfeita o bloco de código não será executado nenhuma vez, visto que a condição já é testada logo de início.

Exemplo: simulando uma estrutura de repetição **for**:

```
static void Main(string[] args)
{
    int contador = 0;

    while (contador <= 10)
    {
        Console.WriteLine(contador);
        contador++; // não esqueça desta linha,!!!!
    }
    Console.ReadLine();
}
```

Observe que para ambos os laços, **while** e **do while**, caso utilizemos um contador, devemos tomar o cuidado de incrementá-lo porque se assim não o fizéssemos, a nossa aplicação entraria num loop infinito. Também as declarações **break** e **continue** podem ser usadas da mesma forma que no laço **for**.

## Vetores e Matrizes

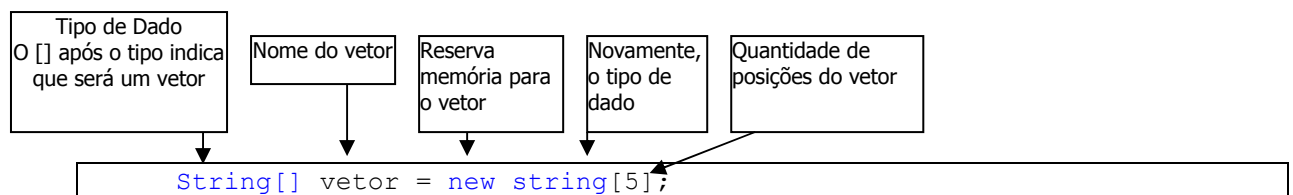
Arrays, ou simplesmente vetores, são elementos dimensionais de um determinado tipo de dados, sendo eles tipo referência ou tipo valor, onde são agrupados num número fixo de elementos. Em C# os vetores são acessados através de um índice (valor numérico) onde o primeiro índice sempre será zero (0).

Um vetor pode ser criado na forma unidimensional ou multidimensional. Um vetor é um tipo referência. Ele indica que a variável, a qual representa o vetor, aponta para os elementos em memória, bem como quando passada através de métodos, seus elementos podem ser alterados. Os vetores de tamanho fixo são baseados na classe `System.Array` do .NET Framework.

Exemplo:

	Vetor				
Dado	'ana'	'maria'	'cláudia'	'beatriz'	'daniela'
Índice	0	1	2	3	4

Para criar o vetor acima, que seja capaz de armazenar valores do tipo string, o comando é:



Para preencher o vetor com os nomes, poderíamos utilizar:

```
vetor[0] = "ana";
vetor[1] = "maria";
vetor[2] = "cláudia";
vetor[3] = "beatriz";
vetor[4] = "daniela";
```

Para inicializarmos o vetor já no momento de sua criação, poderíamos fazer da seguinte forma:

```
string[] vetor = { "ana", "maria", "joana" };
```

Para varrer os dados do vetor, ou seja, para acessar suas informações, a forma mais simples é utilizando estruturas de repetição.

Ex: Vamos imprimir todos os nomes do vetor acima:

**Length:** Retorna a quantidade de posições vetor

```
for (int i = 0; i < vetor.Length; i++)
{
    Console.WriteLine(vetor[i]);
}
```

Se quisermos solicitar ao usuário que informe os 5 nomes, a instrução seria:

```
for (int i = 0; i < vetor.Length; i++)
{
    vetor[i] = Console.ReadLine();
}
```

O conceito de vetores pode ser entendido para vetores multidimensionais. Em vez de pensarmos em algo linear, com um índice, podemos pensar em estruturas com mais de um índice. Um exemplo simples é a caso das matrizes, que podem ser pensadas como vetores de duas dimensões.

Vejamos como declaramos uma matriz (vetor de duas dimensões) :

```
static void Main(string[] args)
{
    // cria uma matriz de duas dimensões: 3 por 3 (3 linhas e 3 colunas)
    int[,] matriz = new int[3,3];

    /* representação da matriz:
        0   1   2
    0[  |  |  ]
    1[  |  |  ]
    2[  |  |  ]
    */

    // inserindo dados na matriz, sem intervenção do usuário.
    matriz[0,0] = 1;
    matriz[0,1] = 2;
    matriz[0,2] = 3;
    matriz[1,0] = 4;
    matriz[1,1] = 5;
    matriz[1,2] = 6;
    matriz[2,0] = 7;
    matriz[2,1] = 8;
    matriz[2,2] = 9;

    // impressão dos dados da matriz:
    // observe que não podemos utilizar o matriz.Length pois agora
    // a matriz possui 2 dimensões e a propriedade Length irá retornar 9 e não 3.
    for (int linha=0; linha<=2; linha++)
    {
        for (int coluna=0; coluna<=2; coluna++)
        {
            Console.Write("{0} ", matriz[linha, coluna]);
        }
        Console.WriteLine();
    }

    /* o resultado em vídeo será:

        0   1   2
    0[ 1 | 2 | 3 ]
    1[ 4 | 5 | 6 ]
    2[ 7 | 8 | 9 ]

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

## Estrutura de decisão (SWITCH)

Ref.: <http://msdn.microsoft.com/pt-br/library/06tc147t.aspx>  
<http://www.arquivodecodigos.net/dicas/c-sharp-a-instrucao-switch-da-linguagem-c-2496.html>.

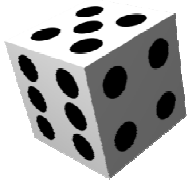
A instrução switch da linguagem C# é útil quando queremos testar condições nas quais o uso de if..else if..else poderia ser considerado excessivo ou não aplicável. Esta instrução recebe uma variável como argumento e testa seu valor por meio de cláusulas case. Veja o exemplo abaixo.

Note que a instrução switch possui uma cláusula default para os casos nos quais o valor da variável não estiver presente em nenhuma das cláusulas case. Observe também o uso da instrução break para evitar a avaliação das cláusulas seguintes àquela na qual o valor desejado foi encontrado.

```
Console.WriteLine("Digite o dia da semana:");
int diaSemana = Convert.ToInt32(Console.ReadLine());

switch (diaSemana)
{
    case 1:
        Console.WriteLine("Domingo");
        break;
    case 2:
        Console.WriteLine("Segunda");
        break;
    case 3:
        Console.WriteLine("Terça");
        break;
    case 4:
        Console.WriteLine("Quarta");
        break;
    case 5:
        Console.WriteLine("Quinta");
        break;
    case 6:
        Console.WriteLine("Sexta");
        break;
    case 7:
        Console.WriteLine("Sábado");
        break;
    default: //a cláusula default é opcional!
        Console.WriteLine("Dia da semana inválido.");
        break;
}
```

## Números randômicos



Ref.: <http://kleberandrade.wordpress.com/2010/01/19/c-numeros-aleatorios-random/>  
<http://msdn.microsoft.com/pt-br/library/system.random.aspx>

Às vezes precisamos gerar números aleatórios, como por exemplo, para simular um dado, ou mesmo para gerar os números da mega-sena. Para isso, a linguagem C# possui a classe `Random`.

Para gerar um número aleatório, primeiro você precisa criar um objeto da classe `Random`:

```
Random gerador = new Random();
```

Após criar o objeto, podemos usá-lo para gerar os números aleatórios.

EX: Para gerar um número aleatório entre 1(inclusive) e 4:

```
int numero = gerador.Next(1, 5);
```

O comando acima poderá gerar os seguintes valores: 1,2,3 ou 4.

O algoritmo abaixo irá gerar 5 números aleatórios entre 1 e 499 e exibi-los em vídeo:

```
Random gerador = new Random();  
  
for (int n = 1; n <= 5; n++)  
    Console.WriteLine(gerador.Next(1, 500));
```

## Estrutura de repetição foreach/in

Este tipo de laço é usado para varrer arrays ou coleções. As suas vantagens em relação ao laço for são as seguintes:

- Não precisamos nos preocupar com a avaliação de uma condição booleana para garantir a sua execução;
- Nem com a inicialização de variáveis com o seu incremento/decremento;
- Nem com a forma de extração do conteúdo do array ou coleção, já que ambos possuem formas diferentes de extração dos seus valores;
- Quando todos os elementos do array/coleção tiverem sido varridos, o laço foreach/in será abandonado. Usando um laço for, uma vez que uma das condições acima falhar, a tentativa de extrair os elementos do array/coleção será malsucedida.

Vejamos um exemplo:

```
string[] vetor = new string[5];

// preenchendo os valores do vetor:
vetor[0] = "ana";
vetor[1] = "maria";
vetor[2] = "cláudia";
vetor[3] = "beatriz";
vetor[4] = "daniela";

// exibindo os valores do vetor:
foreach (string valor in vetor)
{
    Console.WriteLine(valor);
}
```

A cada iteração do foreach, um elemento do "vetor" é atribuído à variável "valor".

Como você pode perceber, o array de strings foi varrido sem termos de nos preocupar com os limites inferior e superior de cada dimensão, nem muito menos com o incremento dos mesmos ou com a sua correta inicialização.





## Recebendo parâmetros na linha de comando

Para receber parâmetros na linha de comando, ou seja, na chamada de um programa quando digitamos o nome do executável no prompt da linha de comando do DOS (como "ScanDisk /All /AutoFix", por exemplo), o método `Main()` precisa ser declarado da seguinte forma:

```
// não retorna nenhum valor ao sistema
```

```
static void Main(string[] args)
```

ou

```
// retorna um valor do tipo int ao sistema
```

```
static int Main(string[] args)
```

O parâmetro `args` é um array de strings que recebe os parâmetros passados quando a aplicação é chamada na linha de comando. A seguir mostramos uma das formas de varrer os parâmetros recebidos:

```
for (int i=0; i< args.Length; i++)  
{  
    Console.WriteLine("Par {0}: {1}", i, cmd);  
}
```

Para saber o número de argumentos que foram passados, usamos a propriedade **Length** do array `args` da seguinte forma:

```
numArgs = args.Length;
```

Quando na linha de comando são recebidos parâmetros numéricos, estes devem ser convertidos de string para o tipo numérico respectivo usando a classe **Convert**. Exemplo:

```
int Varint = Convert.ToInt32(varString);
```

## 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_aluno;
```

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 **endereço** 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 **registro**. 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 no vetor.
        struct Registro
        {
            public int    codigo;
            public string nome;
        }

        static void Main(string[] args)
        {
            Registro Dado = new Registro();

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

            Console.WriteLine("{0} - {1}", Dado.codigo, Dado.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 **registro**. 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 Registro
        {
            public int    codigo;
            public string nome;
        }

        static void Main(string[] args)
        {

            Registro[] Vetor = new Registro[5];

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

                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: 1 Nome: Ana
1	Código: 2 Nome: Daniela
2	Código: 3 Nome: Cláudia
3	Código: 4 Nome: Bruna
4	Código: 5 Nome: Paula

Observe que cada célula possui 2 campos.

## Métodos

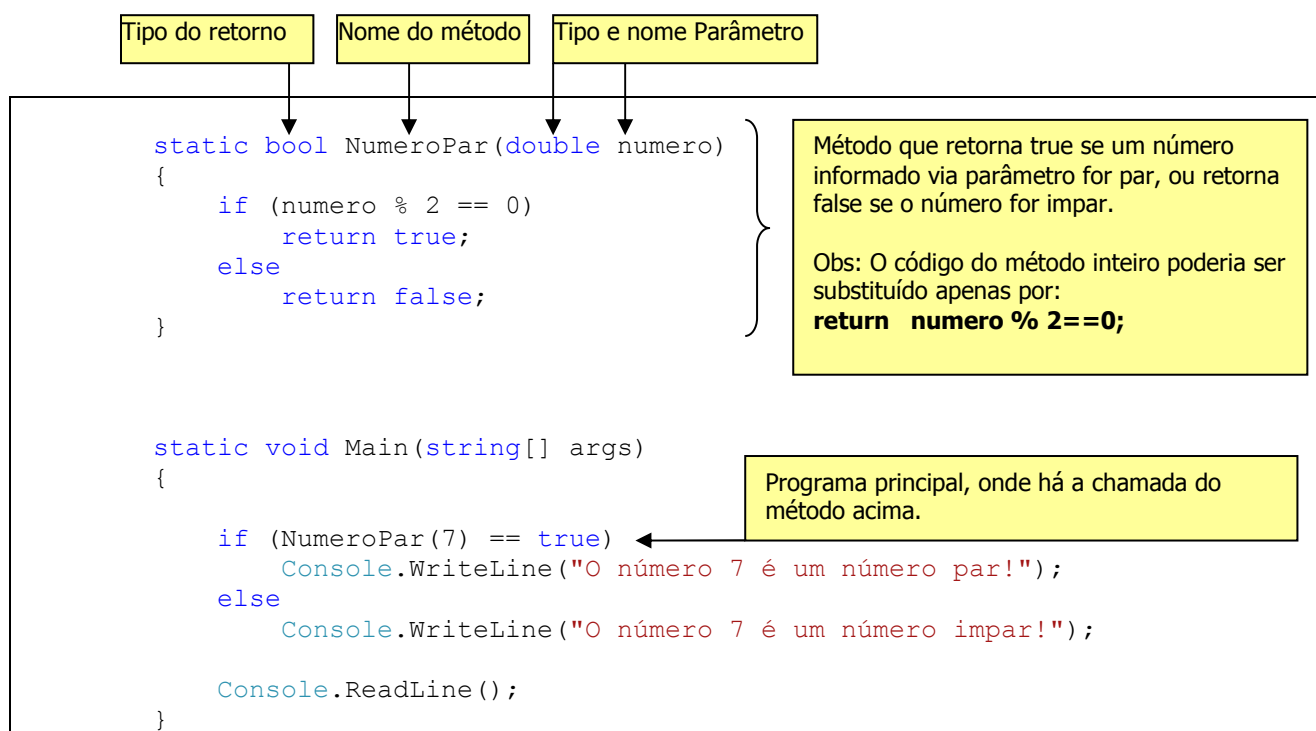
Os métodos em C# são conceitualmente similares a procedimentos e **funções** em outras linguagens de alto nível. Normalmente correspondem a "trechos" de código que podem ser chamados em um objeto específico (de alguma classe). Os métodos podem admitir parâmetros como argumentos, e seu comportamento depende 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 do 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 atual 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("xxxxxxxxxxxxxxxxxxxxxx");
    Console.WriteLine("yyyyyyyyyyyyyyyyyyyy");
    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.**

## Passando Parâmetros para Métodos 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.

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 um 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();
}
```

É necessário utilizar a palavra reservada **ref** na assinatura do método e também quando ele for chamado.

**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 variáveis passadas por referência não têm importância. Esses valores serão sobrescritos pela função, que pode até nunca chegar a examiná-los.

Seria conveniente se pudessemos usar a mesma convenção em C#, mas, como você deve se lembrar, o C# requer que as variáveis sejam inicializadas com algum valor antes de serem referenciadas. Embora pudessemos 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("Conseguir converter para inteiro. Convertido: {0}", numeroInt);
    else
        Console.WriteLine("Não conseguir converter {0} para inteiro.", numeroStr);

    Console.ReadLine();
}
```

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



## 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 de 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.

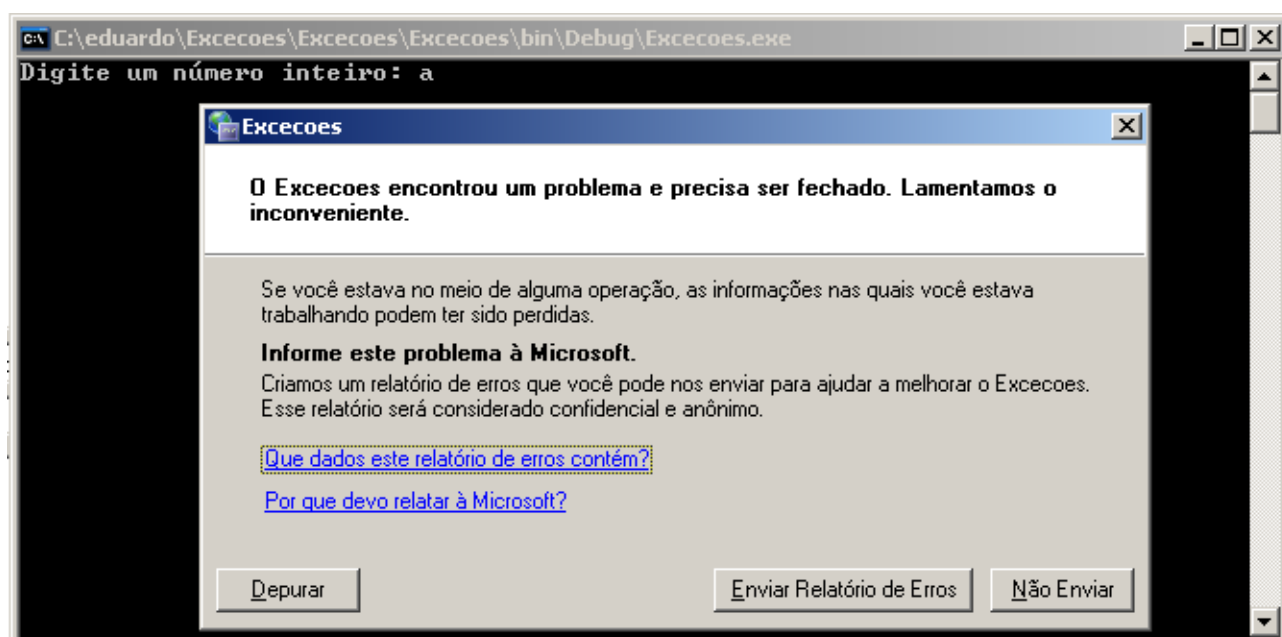
### Erros lógicos:

- 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.

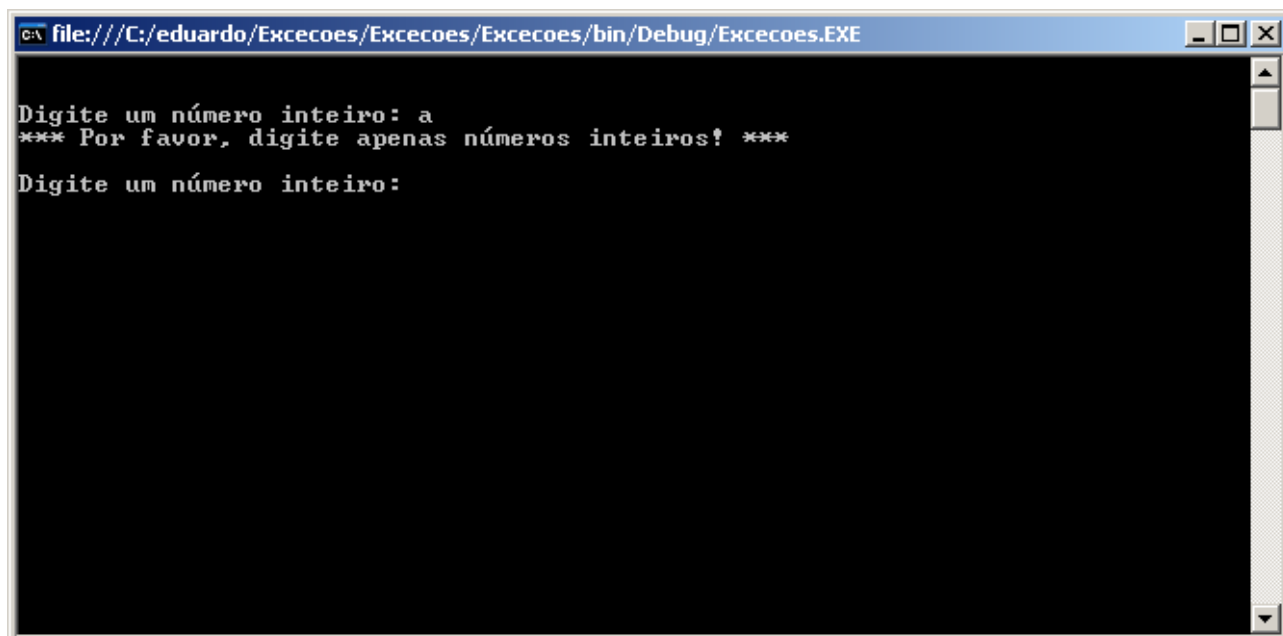
Para que uma aplicação seja segura, seu código necessita reconhecer uma exceção quando ela ocorrer e responder adequadamente a esta. Se não houver tratamento consistente para uma exceção, será exibida uma mensagem padrão descrevendo o erro e todos os processamentos pendentes não serão executados. Uma 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.WriteLine("\n\nDigite um número inteiro: ");
    numero = Convert.ToInt32(Console.ReadLine());
}
catch
{
    Console.WriteLine("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.WriteLine("\n\nDigite um número inteiro: ");
            numero = Convert.ToInt32(Console.ReadLine());
            correto = true;
        }
        catch
        {
            correto = false;
            Console.WriteLine("*** 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**.

