Apostila de Algoritmos II

Professor Ms. Eduardo Rosalém Marcelino
eduardormbr@gmail.com

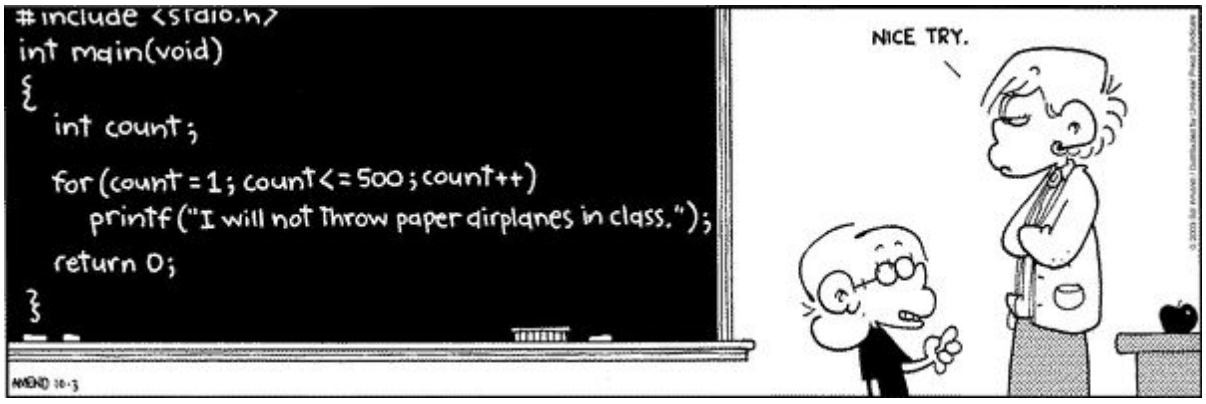
(2009 - 2016)

Bibliografias utilizadas:

ESCLARECIMENTOS

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

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



Índice

Métodos

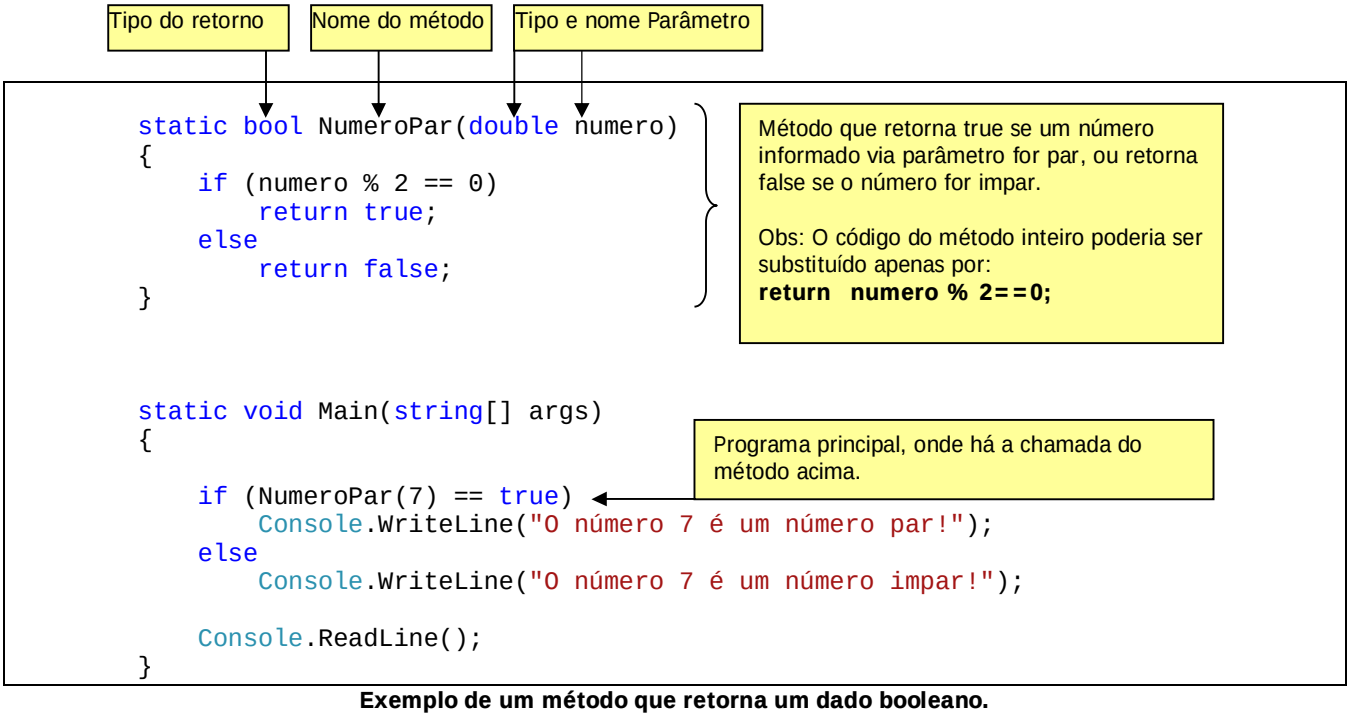
Os métodos em C# são conceitualmente similares a procedimentos e **funções** em outras linguagens de alto nível. Normalmente correspondem a "trechos" de código que podem ser chamados em um objeto específico (de alguma classe). Os métodos podem admitir parâmetros como argumentos, e seu comportamento depende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

Declarando Métodos

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

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

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



Exemplo de um método que retorna um dado booleano.

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

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


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

Exemplo de um método que não retorna nada.

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


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

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

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

Escopo das variáveis

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

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

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

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

    }
}
```

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

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

```
.....

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

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

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

Passagem de Parâmetros por Valor e por Referência

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

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

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

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

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

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


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

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

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

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


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

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

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 vaiáveis passadas por referência não têm importância. Esses valores serão sobrescritos pela função, que pode até nunca chegar a examiná-los.
Seria conveniente se pudéssemos usar a mesma convenção em C#, mas, como você deve se lembrar, o C# requer que as variáveis sejam inicializadas com algum valor antes se serem referenciadas. Embora pudéssemos inicializar nossas variáveis de entrada com valores insignificantes antes de sua passagem dentro da função, o que dará a elas o valor real, essa prática parece desnecessária, e pior, confusa. Contudo, há um meio de acabar com a insistência dos compiladores c# sobre os valores iniciais dos argumentos de entrada.

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

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


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

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

            MaiorMenor(vetor, out maior, out menor);

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

Exemplo de um método que retorna dados nos parâmetros “maior” e “menor”.

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

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

   Console.ReadLine();
}
```

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

Controle de Exceção - Básico

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

Erros de sintaxe ou erro de compilação:

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

Erros de Runtime:

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

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

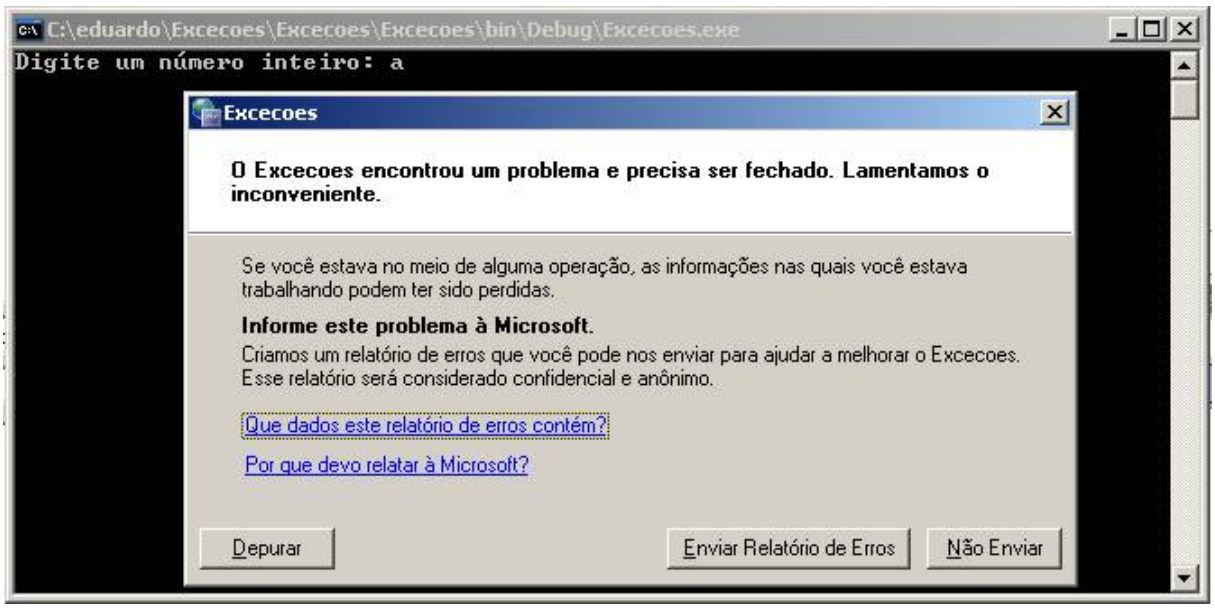
Erros lógicos:

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

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

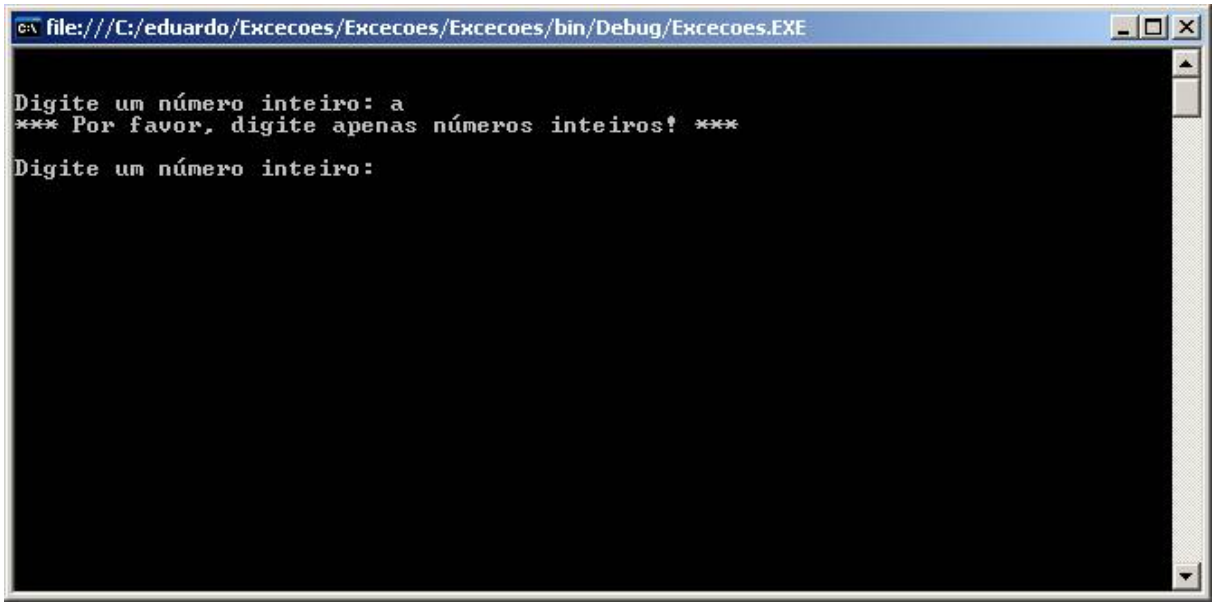
Para que uma aplicação seja segura, seu código necessita reconhecer uma exceção quando ela ocorrer e responder adequadamente a esta. Se não houver tratamento consistente para uma exceção, será exibida uma mensagem padrão descrevendo o erro e todos os processamentos pendentes não serão executados. Uma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

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



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

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



BLOCOS PROTEGIDOS

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

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

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

```
try
{

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

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

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

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

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

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

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

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

Complexidade Computacional

Fontes: http://www.dca.fee.unicamp.br/~ting/Courses/ea869/faq1.html
http://pt.wikipedia.org/wiki/Complexidade_computacional
http://pt.wikipedia.org/wiki/Complexidade_quadr%C3%A1tica

O que é um problema computável?

Um problema é computável se existe um procedimento que o resolve em um número finito de passos, ou seja se existe um algoritmo que leve à sua solução.

Observe que um problema considerado "em princípio" computável pode não ser tratável na prática, devido às limitações dos recursos computacionais para executar o algoritmo implementado.

Por que é importante a análise de complexidade computacional de um algoritmo?

A complexidade computacional de um algoritmo diz respeito aos recursos computacionais - espaço de memória e tempo de máquina - requeridos para solucionar um problema.

Geralmente existe mais de um algoritmo para resolver um problema. A análise de complexidade computacional é portanto fundamental no processo de definição de algoritmos mais eficientes para a sua solução. Apesar de parecer contraditório, com o aumento da velocidade dos computadores, torna-se cada vez mais importante desenvolver algoritmos mais eficientes, devido ao aumento constante do "tamanho" dos problemas a serem resolvidos.

O que entendemos por tamanho de um problema?

O tamanho de um problema é o tamanho da entrada do algoritmo que resolve o problema. Vejamos os seguintes exemplos:

- A busca em uma lista de N elementos ou a ordenação de uma lista de N elementos requerem mais operações à medida que N cresce;
- O cálculo do fatorial de N tem o seu número de operações aumentado com o aumento de N;
- A determinação do valor de F_N na sequência de Fibonacci F_0, F_1, F_2, F_3, ... envolve uma quantidade de adições proporcional ao valor de N.

A Teoria da complexidade computacional é a parte da teoria da computação que estuda os recursos necessários durante o cálculo para resolver um problema. O termo foi criado pelo Juris Hartmanis e Richard Stearns[1]. Os recursos comumente estudados são o tempo (número de passos de execução de um algoritmo para resolver um problema) e o espaço (quantidade de memória utilizada para resolver um problema). Pode-se estudar igualmente outros parâmetros, tais como o número de processadores necessários para resolver o problema em paralelo. A teoria da complexidade difere da teoria da computabilidade a qual se ocupa de factibilidade de se solucionar um problema como algoritmos efetivos sem levar em conta os recursos necessários para ele.

Os problemas que têm uma solução com ordem de complexidade linear são os problemas que se resolvem em um tempo que se relaciona linearmente com seu tamanho.

Os problemas com custo fatorial ou combinatório estão agrupados em NP. Estes problemas não têm uma solução prática, significa dizer quer, uma máquina não pode resolver-los em um tempo razoável.

Análise assintótica

A importância da complexidade pode ser observada no exemplo abaixo, que mostra 5 algoritmos A1 a A5 para resolver um mesmo problema, de complexidades diferentes. Supomos que uma operação leva 1 milisegundo para ser efetuada. A tabela seguinte dá o tempo necessário por cada um dos algoritmos.

Tk(n) é a complexidade do algoritmo.

	A_1	A_2	A_3	A_4	A_5
n	$T_1(n) = n$|	$T_2(n) = nlogn$	$T_3(n) = n^2$	$T_4(n) = n^3$	$T_5(n) = 2^n$
16	0,016s	0,064s	0,256s	4s	1m4s
32	0,032s	0,16s	1s	33s	46 dias
512	0,512s	9s	4m22s	1 dia 13h	10^{137} séculos

A **complexidade do tempo** de um problema é o número de passos que se toma para resolver uma instância de um problema, a partir do tamanho da entrada utilizando o algoritmo mais eficiente à disposição. Intuitivamente, caso se tome uma instância com entrada de longitude *n* que pode resolver-se em n^2 passos, se diz que esse problema tem uma complexidade em tempo de n^2. Supostamente, o número exato de passos depende da máquina em que se programa, da linguagem utilizada e de outros fatores. Para não ter que falar do custo exato de um cálculo se utiliza a notação assintótica. Quando um problema tem custo dado em tempo $O(n^2)$ em uma configuração de computador e linguagem, este custo será o mesmo em todos os computadores, de maneira que esta notação generaliza a noção de custo independentemente do equipamento utilizado.

Notação O, Ω (Omega) e Theta

São usadas três perspectivas no estudo do caso da complexidade:

Melhor Caso - Ω( )

Representado por Ω( ) (Ômega), consiste em assumir que vai acontecer o melhor. Pouco utilizado e de baixa aplicabilidade.
Exemplo 1: Em uma lista telefônica queremos encontrar um número, assume-se que a complexidade do caso melhor é Ω (1), pois está pressupondo-se o número desejado está no topo da lista.

Exemplo 2: Extrair qualquer elemento de um vetor. A indexação em um vetor ou array, leva o mesmo tempo seja qual for o índice que se queira buscar. Portanto é uma operação de complexidade constante Ω (1).

Pior Caso - O( )

Representado normalmente por O( ) (BIG O) . Se dissermos que um determinado algoritmo é representado por g(x) e a sua complexidade Caso Pior é n, será representada por g(x) = O(n). Consiste em assumir que o pior dos casos pode acontecer, sendo de grande utilização e também normalmente o mais fácil de ser determinado.

Exemplo 1: Será tomado como exemplo o jogo de azar com 3 copos, deve descobrir-se qual deles possui uma moeda debaixo dele, a complexidade caso pior será O(3) pois no pior dos casos a moeda será encontrada debaixo do terceiro copo, ou seja, será encontrada apenas na terceira tentativa.

Exemplo 2: O processo mais comum para ordenar um conjunto de elementos têm complexidade quadrática. O procedimento consiste em criar uma coleção vazia de elementos. A ela se acrescenta, em ordem, o menor elemento do conjunto original que ainda não tenha sido eleito, o que implica percorrer completamente o conjunto original (O(n), sendo *n* o número de elementos do conjunto). Esta percorrida sobre o conjunto original se realiza até que se tenha todos seus elementos na seqüência de resultado. Pode-se ver que há de se fazer *n* seleções (se ordena todo o conjunto) cada uma com um custo *n* de execução: o procedimento é de ordem quadrática O(n^2). Deve esclarecer se de que há diversos algoritmos de ordenação com melhores resultados.

Caso médio - θ( )

Representado por θ( ) (Theta). Este é o caso que é o mais difícil de ser determinado, pois, necessita de análise estatística e em conseqüência de muitos testes, contudo é muito utilizado, pois é o que representa mais corretamente a complexidade do algoritmo.

Exemplo: Procurar uma palavra em um dicionário. Pode-se iniciar a busca de uma palavra na metade do dicionário. Imediatamente se sabe se foi encontrada a palavra ou, no caso contrário, em qual das duas metades deve se repetir o processo (é um processo recursivo) até se chegar ao resultado. Em cada busca (ou sub-busca), o problema (as páginas em que a palavra pode estar) vão se reduzindo à metade, o que corresponde com a função logarítmica. Este procedimento de busca (conhecido como busca binária) em uma estrutura ordenada têm complexidade logarítmica θ($\log_2 n$).

Exemplos de Complexidade

Complexidade Constante

Representada por O(1). Complexidade algorítmica cujo tempo de execução independe do número de elementos na entrada.

```
            Console.Write("Digite um texto: ");
            texto = Console.ReadLine();

            Console.Write("Digite uma letra: ");
            letra = Console.ReadLine()[0];
```

Apesar de termos 4 instruções, o processo como um todo será considerado como O(1), pois neste caso executar 1 ou 4 instruções é irrelevante.

Complexidade Linear - O(n)

Representada por O(*n*). Complexidade algorítmica em que um pequeno trabalho é realizado sobre cada elemento da entrada. Esta é a melhor situação possível para um algoritmo que tem que processar *n* elementos de entrada ou produzir *n* elementos de saída. Cada vez que *n* dobra de tamanho o tempo de execução dobra.

Programa exemplo: Determinar o maior elemento em um vetor:

```
        static void Main(string[] args)
         {
             int[] numeros = { 7, 5, 21, 33, 79, 8, 6, 29, 44, 10, -5 };

             int maior = numeros[0];

             for (int n = 1; n < numeros.Length; n++)
             {
                 if (numeros[n] > maior)
                     maior = numeros[n];
             }

             Console.WriteLine(maior);
         }
```

A complexidade é dada por **n** elementos (o elemento que determina a taxa de crescimento do algoritmo acima) ou **O(n) (**linear**). Ou seja, depende da quantidade de elementos no vetor!**

Complexidade Logaritmica

Representada por O(log *n*). Complexidade algorítmica no qual algoritmo resolve um problema transformando-o em partes menores. Nestes casos, o tempo de execução pode ser considerado como sendo menor do que uma constante grande. Por exemplo, quando *n* é um milhão, $log_2 n$ é aproximadamente 20.

Complexidade Quadrática - O(n^2)

Representada por O(n^2). Complexidade algorítmica que ocorrem quando os itens de dados são processados aos pares, muitas vezes em uma repetição dentro da outra. Por exemplo, quando n é mil, o número de operações é da ordem de 1 milhão. Algoritmos deste tipo somente são úteis para resolver problemas de tamanhos relativamente pequenos.

Programa exemplo: Preencher uma matriz NxN com números aleatórios.

```
        static int[,] CriaMatriz(int tamanho)
        {
            int[,] matriz = new int[tamanho, tamanho];

            Random r = new Random();

            //preenche a matriz NxN (onde N = tamanho) com números
randomicos
            for (int i = 0; i < tamanho; i++)
                for (int j = 0; j < tamanho; j++)
                    matriz[i, j] = r.Next(1000);

          return matriz;
        }
```

A complexidade é dada por **n^2 (i*j)** elementos ou **O(n^2)** (quadrática)

Funções limitantes superiores mais conhecidas:

Melhor → Pior

Constante	Logarítmica	Linear	Quadrática	Exponencial
O(1)	O(log n)	O(n)	O(n^2)	O(a^n) com a > 1

Exemplo de crescimento de várias funções

n	log n	n	n^2	2^n
2	1	2	4	4
4	2	4	16	16
8	3	8	64	256
16	4	16	256	65.536
32	5	32	1.024	4.294.967.296

64	6	64	4.096	$1,84 \times 10^{19}$
128	7	128	16.384	$3,40 \times 10^{38}$
256	8	256	65.536	$1,18 \times 10^{77}$
512	9	512	262.144	$1,34 \times 10^{154}$
1024	10	1024	1.048.576	$1,79 \times 10^{308}$

Tempos de execução dos algoritmos:

Os Algoritmos têm tempo de execução proporcional a:

O(1) - muitas instruções são executadas uma só vez ou poucas vezes (se isto for verdade para todo o programa diz-se que o seu tempo de execução é constante)

O(log N) - tempo de execução é logarítmico (cresce linearmente à medida que **N** cresce) (quando **N** duplica **log N** aumenta mas muito pouco;**)**

O(N) - tempo de execução é linear (típico quando algum processamento é feito para cada dado de entrada) (situação ótimas quando é necessário processar **N** dados de entrada) (ou produzir **N** dados na saída)

O (N log N) - típico quando se reduz um problema em subproblemas, se resolve estes separadamente e se combinam as soluções (se **N** é 1 milhão, **N log N** é perto de 20 milhões)

O (N^2) - tempo de execução típico **N** quadrático (quando é preciso processar todos os pares de dados de entrada) (prático apenas em pequenos problemas). EX: para **N**=10, $N^2 = 100$.

O (N^3) - tempo de execução cúbico. EX: para **N** = 100, N^3 = 1 milhão

O (N^3) - tempo de execução cúbico. EX: para **N** = 100, N^3 = 1 milhão

O (2^N) - tempo de execução exponencial (provavelmente de pouca aplicação prática; típico em soluções de força bruta) (para **N** = 20, 2^N = 1 milhão)

Pesquisa Binária

Retirado de : http://pt.wikipedia.org/wiki/Pesquisa_bin%C3%A1ria

A pesquisa ou busca binária (em inglês binary search algorithm ou binary chop) é um algoritmo de busca em vetores que requer acesso aleatório aos elementos do mesmo. **Ela parte do pressuposto de que o vetor está ordenado**, e realiza sucessivas divisões do espaço de busca comparando o elemento buscado (chave) com o elemento no meio do vetor. Se o elemento do meio do vetor for a chave, a busca termina com sucesso. Caso contrário, se o elemento do meio vier antes do elemento buscado, então a busca continua na metade posterior do vetor. E finalmente, se o elemento do meio vier depois da chave, a busca continua na metade anterior do vetor. A complexidade desse algoritmo é da ordem de $O(\log_2 n)$, onde n é o tamanho do vetor de busca.

Um pseudo-código recursivo para esse algoritmo, dados V o vetor com elementos comparáveis, n seu tamanho e e o elemento que se deseja encontrar:

```
BUSCA-BINÁRIA (V[], inicio, fim, e}
     i recebe o índice no meio de inicio e fim
     se V[i] é igual a e
          então devolva o índice i    # encontrei e
          senão se V[i] vem antes de e
                     então faça a BUSCA-BINÁRIA(V, i+1, fim, e)
                     senão faça a BUSCA-BINÁRIA(V, inicio, i-1, e)
```

{Algorítmo em Pascal}

```
function BuscaBinaria (Vetor: array of string; Chave: string; Dim: integer): integer;
     var inicio, fim: integer; {Auxiliares que representam o inicio e o fim do vetor analisado}
         meio: integer; {Meio do vetor}
     begin
          fim := Dim; {O valor do último índice do vetor}
          inicio := 1; {O valor do primeiro índice do vetor}
            repeat
               meio := (inicio+fim) div 2;
                 if (Chave = vetor[meio]) then
                      BuscaBinaria := meio;
                 if (Chave < vetor[meio]) then
                      fim:=(meio-1);
                 if (Chave > vetor[meio]) then
                      inicio:=(meio+1);
          until (Chave = Vetor[meio]) or (inicio > fim);
          if (Chave = Vetor[meio]) then
                BuscaBinaria := meio
          else
                BuscaBinaria := -1; {Retorna o valor encontrado, ou -1 se a chave nao foi encontrada.}
     end;
```

Exemplo: Dado vetor abaixo, **Procurar o número 80:**

1	3	7	9	**15**	17	21	65	80	99

Meio = 15
80 é maior que 15, então, procurar no intervalo à direita:

17	21	**65**	80	99

Meio = 65
80 é maior que 65, então procurar no intervalo à direita:

80	99

Meio = 80
É o valor procurado. Parar a pesquisa!

80

Para procurar 80, foram necessárias 3 iterações.

Recursividade ou Recursão

Material retirado de:

http://pt.wikipedia.org/wiki/Recursividade (muito bom)
http://pt.wikipedia.org/wiki/Recursividade_%28ci%C3%AAncia_da_computa%C3%A7%C3%A3o%29 (ótimo)
http://www.di.ufpe.br/~if096/recursao/ (bom)
Referência [1]

O que é Recursividade: Uma Rotina ou Função é recursiva quando ela chama a si mesma, seja de forma direta ou indireta. Uma função recursiva DEVE ter um ponto de parada, ou seja, em algum momento ela deve parar de se chamar. Praticamente todas as linguagens oferecem suporte a recursividade. Trata-se de uma técnica de programação.

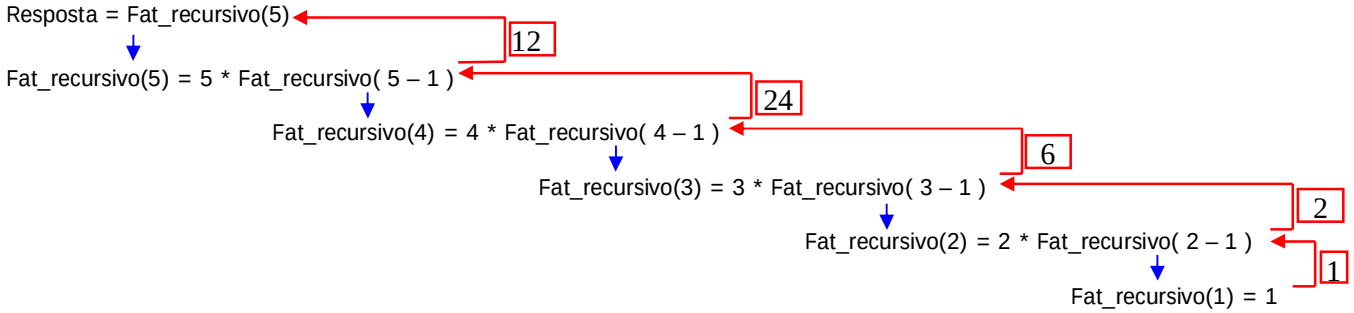
Por exemplo, segue uma definição recursiva da ancestralidade de uma pessoa:

- Os pais de uma pessoa são seus antepassados (caso base);
- Os pais de qualquer antepassado são também antepassados da pessoa em consideração (passo recursivo).

Cálculo do Fatorial **sem Recursão**, usamos apenas uma estrutura de repetição (iterador)	Cálculo do Fatorial **com Recursão** direta

`long fat_iterativo(int numero)`
`{`
`  long r=1;`
 	
`  for (int i=2; i<= numero; i++)`
`  {`
`  r = r * i;`
`  }`
`  return r;`
`}`
	`long fat_recursivo(int numero)`
	`{`
	`  if (numero == 0)`
	`  return 1;`
	`  else if (numero >= 2)`
	`  return numero*fat_recursivo(numero-1);`
	`  else`
	`  return numero;`
	`}`

Teste de Mesa do Fatorial de 5: azul = ida(chamada recursiva) , vermelho = volta (retorno da função recursiva)



Recursão versus Iteração

No exemplo do fatorial, a implementação iterativa tende a ser ligeiramente mais rápida na prática do que a implementação recursiva, uma vez que uma implementação recursiva precisa registrar o estado atual do processamento de maneira que ela possa continuar de onde parou após a conclusão de cada nova excecução subordinada do procedimento recursivo. Esta ação consome tempo e memória.
Existem outros tipos de problemas cujas soluções são inerentemente recursivas, já que elas precisam manter registros de estados anteriores. Um exemplo é o percurso de uma árvore;

Toda função que puder ser produzida por um computador pode ser escrita como função recursiva sem o uso de iteração; reciprocamente, qualquer função recursiva pode ser descrita através de iterações sucessivas. Todos altoritmo recursivo pode ser implementado iterativamente com a ajuda de uma pilha, mas o uso de uma pilha, de certa forma, anula as vantagens das soluções iterativas.

Tipos de Recursividade:

Direta: Quando chama a si mesma, quando dada situação requer uma chamada da própria Rotina em execução para si mesma. Ex: O exemplo de fatorial recursivo dado acima.

Indireta: Funções podem ser recursivas (invocar a si próprias) indiretamente, fazendo isto através de outras funções: assim, "P" pode chamar "Q" que chama "R" e assim por diante, até que "P" seja novamente invocada.

Ex:

```
double Calculo( double a,b )
{
   return Divide(a,b) + a + b;
}
```

```
double Divide( double a, b )
{
   if (b == 0)
       b  = Calculo(a, b
   return a/ b;
}
```

Aqui ocorre a recursividade indireta!

Note que a função `fatorial` usada como exemplo na seção anterior* não *é recursiva em cauda, pois depois que ela recebe o resultado da chamada recursiva, ela deve multiplicar o resultado por `VALOR` antes de retornar para o ponto em que ocorre a chamada.

Qual a desvantagem da Recursão?
Cada chamada recursiva implica em maior tempo e espaço, pois, toda vez que uma Rotina é chamada, todas as variáveis locais são recriadas.

Qual a vantagem da Recursão?

Se bem utilizada, pode tornar o algoritmo: elegante, claro, conciso e simples. Mas, é preciso antes decidir sobre o uso da Recursão ou da Iteração.

Exercícios

Exercício 1:

Faça um programa para calcular a potencia de um número. O método recursivo deve receber como parâmetro a base e o expoente, e devolver o valor da potência.

EX: double CalculaPotencia (int p_base, int p_expoente)
CalculaPotencia (2,3) = 8

Exercício 2:

Faça o teste de mesa do seu método para os valores informados acima.

Exercício 3:

Faça um programa para imprimir a tabuada, usando recursividade.

Exercício 4:

Progressão aritmética é um tipo de sequência numérica que a partir do segundo elemento cada termo (elemento) é a soma do seu antecessor por uma constante.

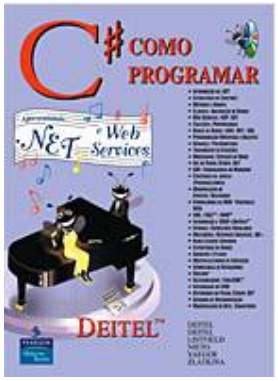
(5,7,9,11,13,15,17) essa sequência é uma Progressão aritmética, pois os seus elementos são formados pela soma do seu antecessor com a constante 2.
a1 = 5
a2 = 5 + 2 = 7
a3 = 7 + 2 = 9
a4 = 9 + 2 = 11
a5 = 11 + 2 = 13
a6 = 13 + 2 = 15
a7 = 15 + 2 = 17

Faça um programa que solicite: O elemento inicial da PA, a constante (razão) e o total de sequencias que se deseja gerar e então faça 2 algoritmos para resolver este problema: 1 recursivo e um sem recursividade.

Programação Orientada a Objetos

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

Vide material na biblioteca virtual nos seguintes livros:

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

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

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

[

UML

Desenvolvimento Windows Forms

Componentes

Eventos

Formulários

Principais paradigmas de projeto de Algoritmos - algoritmos de tentativa e erro, divisão e conquista, balanceamento, programação dinâmica, algoritmos gulosos e algoritmos aproximados.