Visual Basic .NET

Procedimentos, Escopo e Tratamento de Exceções

Professor: Danilo Giacobo

Página pessoal: www.danilogiacobo.eti.br
E-mail: danilogiacobo@gmail.com

Objetivos da aula

- Como criar procedimentos?
- Como criar funções?
- Como comentar seus procedimentos?
- Como passar um número de argumentos variável para um procedimento?
- Como especificar argumentos opcionais em um procedimento?
- Como preservar o valor de varáveis entre chamadas de procedimentos?
- Como criar uma procedimento delegado?
- Como criar propriedades?
- Como entender o escopo de um programa?
- Como usar o tratamento de exceções não estruturada?
- Como usar as declarações Resume Next e Resume Line?
- Como usar a declaração On Error GoTo 0?
- Como recuperar o código e a descrição de um erro?
- Como usar o tratamento de exceções estruturada?
- Como usar os blocos Try, Catch e Finally?

Introdução

Dividir o seu código em procedimentos permite que você "quebre" o código em unidades mais modulares. A linguagem VB .NET trabalha com dois tipos de procedimentos:

- **Sub procedures:** não retornam valores quando terminam a execução.
- **Functions:** retornam valores quando terminam de executar.

O **Escopo** de um código é importante a medida que o programa cresce e você precisa definir quais partes do código serão acessíveis por outras partes, principalmente se tratando da Orientação a Objetos.

O tratamento de erros em Visual Basic .NET pode ser **estruturado** ou **não estruturado**. A forma estruturada é parecida com a de outras linguagens como Java e usa os blocos **Try**, **Catch** e **Finally**. O tipo não estruturado é tradicional do próprio VB e usa as declarações **On Error GoTo**.

Sub Procedures

Sub Procedures são criadas a partir de um conjunto de códigos que executam uma tarefa específica mas que não retornam um valor ao final de sua execução.

Exemplo:

```
1  Module Module1
2      Sub Main()
3          MostrarMensagem()
4      End Sub
5
6      Sub MostrarMensagem()
7          System.Console.WriteLine("Olá Turma de Desenvolvimento em Ambiente Visual!")
8      End Sub
9  End Module
```

Dica: Opcionalmente você pode usar a declaração **Call** para chamar uma sub procedure. Por exemplo: **Call** MostrarMensagem(). Os "()" indicam os argumentos que um procedimento recebe (se houver).

Sub Procedures

Para especificar **argumentos** para um procedimento é necessário especificar o **tipo de dados** e o **nome** dos mesmos.

Exemplo:

```
1  Module Module1
2      Sub Main()
3          MostrarMensagem("Olá Turma de Desenvolvimento em Ambiente Visual!")
4      End Sub
5
6      Sub MostrarMensagem(ByVal strTexto As String)
7          System.Console.WriteLine(strTexto)
8      End Sub
9  End Module
```

A palavra **ByVal** significa que o texto é passado *por valor*, isto é, uma cópia do valor é passada para o procedimento. Essa é a opção padrão. A outra possibilidade é passar o argumento *por referência* (**ByRef**). Essa opção permite que o procedimento altere o valor de uma variável e quando este é terminado você continua com o mesmo valor.

Sub Procedures

O exemplo abaixo mostra um procedimento usando um argumento passado por referência. Note que depois que eu altero o valor do parâmetro dentro do procedimento e este termina, eu perco o valor original.

Exemplo:

```
1  Module Module1
2      Sub Main()
3          Dim strMensagem As String
4          strMensagem = "Olá Turma de Desenvolvimento em Ambiente Visual!"
5          MostrarMensagem(strMensagem)
6          System.Console.WriteLine(strMensagem)
7      End Sub
8
9      Sub MostrarMensagem(ByRef strTexto As String)
10         System.Console.WriteLine(strTexto)
11         strTexto = "Eu alterei o texto do argumento!"
12     End Sub
13 End Module
```

Functions

Funções são criadas a partir de um conjunto de códigos que executam uma tarefa específica e que retornam um valor ao final de sua execução. Você deve informar o tipo de dados do retorno da função. O valor a ser retornado pela mesma é definido pela palavra **Return**.

Exemplo:

```
1  Module Module1
2      Sub Main()
3          Dim intValor As Integer = 2
4          System.Console.WriteLine("{0} + {1} = {2}", _
5                                   intValor, intValor, Somar(intValor, intValor))
6      End Sub
7
8      Function Somar(ByVal int1 As Integer, int2 As Integer) As Long
9          Return int1 + int2
10         'Somar = int1 + int2 (você pode usar assim também)
11     End Function
12 End Module
```

Dica: Em vez da palavra **Return** você pode usar o próprio **nome da função** para retornar um valor.

Escopo

Em VB .NET onde você declara um elemento é o que determina qual é o seu escopo. Ele pode ser:

- **Escopo de Bloco**
- **Escopo de Procedimento**
- **Escopo de Módulo**
- **Escopo de Namespace**

```
1  Module Module1
2      Dim mintValor As Integer    'Escopo de Módulo
3
4      Sub Main()
5          Dim lintValor As Integer = 2    'Escopo de Procedimento
6      End Sub
7
8      Sub Teste()
9          'lintValor = 2  'Erro
10         mintValor = 2   'OK
11     End Sub
12 End Module
```

Escopo

Dentro de cada nível de escopo você ainda tem outras possibilidades:

- **Public**
  - Elementos podem ser acessados de qualquer lugar dentro do mesmo projeto.
  - Elementos podem ser acessados de outros projetos que referenciam ele.
  - Essa declaração só pode ser usada a nível de Módulo, Namespace ou Arquivo.
- **Protected**
  - Elementos podem ser acessados apenas dentro da mesma classe ou de uma classe derivada.
  - Essa declaração só pode ser usada em membros de uma classe.
- **Friend**
  - Elementos podem ser acessados de qualquer lugar dentro do mesmo projeto mas não fora dele.
  - Essa declaração só pode ser usada a nível de Módulo, Namespace ou Arquivo.
- **Protected Friend**
  - Elementos podem ser acessados de classes derivadas ou do mesmo projeto ou de ambos.
  - Essa declaração só pode ser usada em membros de uma classe.
- **Private**
  - Elementos só podem ser acessados dentro do mesmo módulo, classe ou estrutura.
  - Essa declaração só pode ser usada a nível de Módulo, Namespace ou Arquivo.

Escopo - Exemplos

```
1  Module Module1
2      Sub Main()
3          Dim intValor As Integer = 1 'Escopo de Procedimento
4          If intValor = 1 Then
5              Dim strTexto As String = "Sem problemas."    'Escopo de Bloco
6              System.Console.WriteLine(strTexto)
7          End If
8          System.Console.WriteLine(strTexto)  'Não irá funcionar
9      End Sub
10 End Module
```

```
1  Module Module1
2      Sub Main()
3          System.Console.WriteLine(Module2.Function1())
4      End Sub
5  End Module
6
7  Module Module2
8      Public Function Function1() As String 'OK
9          Return "Olá do Visual Basic .NET"
10     End Function
11 End Module
```

Tente trocar a palavra Public por Private

Tratamento de Exceções

Há duas formas de tratamento de erros em tempo de execução:

- **Estruturada**
- **Não Estruturada**

O que é uma exceção?

Exceções são erros em tempo de execução.

Você **PODE** e **DEVE** "capturar" as exceções que ocorrem em seu programa para que elas (as exceções) não levem seu programa a um fim inglório.

Há vários tipos de exceções que podem ser tratadas e a linguagem VB .NET possui um vasto campo de possibilidades para cuidar das mesmas.

Tratamento de Exceções

A tabela abaixo apresenta o nome e a descrição de algumas das exceções mais utilizadas em VB .NET:

Nome	Descrição
ArgumentException	Lançada quando um argumento fornecido para o método não é válido.
ArithmeticException	Classe base para exceções que ocorrem durante operações aritméticas.
ArrayTypeMismatchException	ArrayTypeMismatchException é lançada quando um objeto incompatível está sendo armazenado em um array.
DivideByZeroException	Ocorre quando há uma tentativa de dividir um número por zero.
IndexOutOfRangeException	IndexOutOfRangeException é lançada quando se tenta acessar um array usando um índice que é menor que zero ou maior que o máximo permitido.
InvalidCastException	Lançada quando um uma conversão de tipo explícita de um tipo base para um tipo derivado falha em tempo de execução.
NullReferenceException	Lançada quando um objeto ao ser acessado possui uma referência nula.
OutOfMemoryException	OutOfMemoryException é lançada quando não há memória suficiente para realizar a operação.
OverflowException	OverflowException é lançada quando em uma operação aritmética ocorre overflow.

Tratamento de Exceções - Não Estruturada

- Velho mecanismo herdado da versão 6 do VB
- Usa a declaração **On Error GoTo**
- Transfere o controle do programa em caso de exceção
- Envolve a criação de "labels"
- Precisa da palavra **Exit Sub**

Visão Geral de Funcionamento:

```
1  Module Module1
2      Sub Main()
3          On Error GoTo Tratamento
4
5          ' Código do programa
6
7          Exit Sub
8
9  Tratamento:
10
11         ' Código para tratar o erro
12
13     End Sub
14 End Module
```

Tratamento de Exceções - Não Estruturada

O programa do exemplo abaixo causa uma exceção. Quando a exceção ocorre o controle do programa é passado para a linha 8. Ele trata o erro e mostra uma mensagem; e usando as palavras **Resume Next** retorna o controle para a linha 6.

```
1  Module Module1
2      Sub Main()
3          Dim int1 = 0, int2 = 1, int3 As Integer
4          On Error GoTo Tratamento
5          int3 = CInt(int2 / int1)
6          System.Console.WriteLine("A resposta é {0}", int3)
7          Exit Sub
8  Tratamento:
9          System.Console.WriteLine("Erro: divisão por zero.")
10         Resume Next 'Com essa instrução seu programa mostra o erro
11         'e continua executando (não é interrompido)
12     End Sub
13 End Module
```

Tratamento de Exceções - Estruturada

- Mecanismo parecido com o de outras linguagens
- Usa a sintaxe **Try...Catch...Finally**
- Precisa da classe **Exception**
- Para cada exceção é necessário um bloco **Catch**

Visão Geral de Funcionamento:

```
1  Module Module1
2      Sub Main()
3          Try
4              'Código a ser tratato
5          Catch ex As Exception
6              'Tratamento da exceção
7          Finally
8              'Bloco Opcional
9              'Faz alguma coisa independente se deu erro ou não
10         End Try
11     End Sub
12 End Module
```

Tratamento de Exceções - Estruturada

O exemplo abaixo causa uma exceção. Quando a exceção ocorre o controle do programa é passado para a linha 8. Ele trata o erro e mostra uma mensagem.

```
1  Module Module1
2      Sub Main()
3          Dim int1 = 0, int2 = 1, int3 As Integer
4          Try
5              int3 = int2 / int1
6              System.Console.WriteLine("A resposta é {0}", int3)
7          Catch ex As Exception
8              System.Console.WriteLine(ex.ToString)
9          End Try
10     End Sub
11 End Module
```

Dica: Você pode usar o campo **ex.Message** que contém a seguinte mensagem:

Arithmetic operation resulted in an overflow.

Comentando seus procedimentos

De uma forma geral, você deve adicionar um novo comentário quando você declara uma nova e importante variável. Uma boa prática também é aplicar esse método para procedimentos. A tabela abaixo mostra os possíveis elementos que um comentário de procedimento deve ter.

Cabeçalho da seção	Descrição do comentário
Propósito	O que o procedimento faz.
Suposições	Lista de cada variável externa, controle ou arquivo aberto.
Efeitos	Lista de cada variável externa, controle ou arquivo afetado pela função.
Entradas	Cada argumento que pode não parecer óbvio (um por linha).
Saídas	Explanação dos valores retornados pelas funções.

Comentando seus procedimentos

Exemplo:

```
1  Module Module1
2      '*************************************************************
3      ' dblQuadrado()
4      ' Propósito: Obter o quadrado de um número
5      ' Entradas: intNumero, o valor a ser elevado ao quadrado
6      ' Saídas: O valor do número ao quadrado
7      '*************************************************************
8      Function dblQuadrado(intNumero As Integer) As Double
9          dblQuadrado = intNumero * intNumero 'Use *, não ^2, para velocidade
10     End Function
11 End Module
```

Dica: Clicando com o botão direito do mouse em cima de um procedimento e selecionando a opção **Insert Comment** você pode incluir os comentários no padrão do Visual Studio que depois podem ser transformados em um tipo de documentação do sistema.

Passando um número variável de argumentos

- Usualmente, você não pode chamar um procedimento com mais argumentos que a declaração da mesma permite.

- Quando você quiser um número indefinido de argumentos você pode declarar um array de parâmetros, que permite a um procedimento aceitar um array de valores como um argumento.

- Na definição do procedimento você não precisa especificar o número de elementos do array de parâmetros. O tamanho dele é determinado quando o procedimento é chamado.

- Os argumentos deste tipo são sempre passados usando **ByVal**.

- Todos os argumentos do array devem ser do mesmo tipo de dados.

- Para este fim é usado o elemento **ParamArray**.

Passando um número variável de argumentos

Exemplo:

```
1  Module Module1
2      Sub Main()
3          MostrarMensagem("Primeira mensagem", "Oi")
4          MostrarMensagem("Segunda mensagem", "Olá", "Danilo")
5          Dim ArrayTexto() As String = {"Bem-vindo", "ao", "Visual", "Basic"}
6          MostrarMensagem("Terceira mensagem", ArrayTexto)
7      End Sub
8
9      Sub MostrarMensagem(ByVal strTitulo As String, ByVal ParamArray TextoMensagem() As String)
10         Dim intIndice As Integer
11         System.Console.WriteLine(strTitulo)
12         For intIndice = 0 To UBound(TextoMensagem)
13             System.Console.WriteLine(TextoMensagem(intIndice))
14         Next intIndice
15     End Sub
16 End Module
```

Especificando Argumentos Opcionais

- Para tornar um argumento de um determinado procedimento "opcional", você deve usar a palavra **Optional** (ela deve ser colocada antes do nome do parâmetro).

- Se você definir um argumento como opcional, todos os parâmetros seguintes devem ser também opcionais e além disso devem possuir um valor padrão (default). Você define um valor padrão usando o operador = mais o valor padrão do argumento.

- Você pode usar a palavra **Nothing** como valor padrão dos argumentos de um procedimento e usando a função **IsNothing** você consegue verificar quais parâmetros opcionais tiveram seus valores informados.

Especificando Argumentos Opcionais

Exemplo:

```
1  Module Module1
2      Sub Main()
3          MostrarMensagem()
4          MostrarMensagem("Professor: Danilo Giacobo")
5      End Sub
6
7      Sub MostrarMensagem(Optional ByVal strTitulo As String = Nothing, Optional ByVal strMensagem As String = "Olá")
8          If Not IsNothing(strTitulo) Then
9              System.Console.WriteLine(strTitulo)
10         End If
11
12         System.Console.WriteLine(strMensagem)
13     End Sub
14 End Module
```

Preservando valores de variáveis em procedimentos

Veja o seguinte código:

```
1  Module Module1
2      Sub Main()
3          Dim intIndice As Integer, intValor = 0
4          For intIndice = 0 To 4
5              intValor = Contar()
6          Next intIndice
7          System.Console.WriteLine(intValor)
8      End Sub
9
10     Function Contar() As Integer
11         Dim intContarValor As Integer
12         intContarValor += 1
13         Return intContarValor
14     End Function
15 End Module
```

Quando este programa é executado qual é o valor mostrado na tela? **1** ou **5**?

Preservando valores de variáveis em procedimentos

Veja agora o código corrigido:

```
1  Module Module1
2      Sub Main()
3          Dim intIndice As Integer, intValor = 0
4          For intIndice = 0 To 4
5              intValor = Contar()
6          Next intIndice
7          System.Console.WriteLine(intValor)
8      End Sub
9
10     Function Contar() As Integer
11         Static intContarValor As Integer
12         intContarValor += 1
13         Return intContarValor
14     End Function
15 End Module
```

Dica: Você poderia declarar a variável **intContarValor** fora da função **Contar()** tornando-a com escopo modular. Essa mudança não é recomendada porque pode causar conflitos entre variáveis de mesmo nome e escopo diferentes.

Criando procedimentos delegados

Veja o seguinte código:

```
1  Module Module1
2      Delegate Sub SubDelegate1(ByVal strTexto As String)
3
4      Sub Main()
5          Dim Mensageiro As SubDelegate1
6          Mensageiro = AddressOf MostrarMensagem
7          Mensageiro.Invoke("Olá Prof. Danilo Giacobo")
8      End Sub
9
10     Sub MostrarMensagem(ByVal strTexto As String)
11         System.Console.WriteLine(strTexto)
12     End Sub
13 End Module
```

Criando propriedades

Veja o seguinte código:

```
1  Module Module1
2      Sub Main()
3          Module2.Prop1 = "Propriedade 1"
4          System.Console.WriteLine("Prop1 = " & Module2.Prop1)
5      End Sub
6  End Module
7
8  Module Module2
9      Private strValor As String
10
11     Public Property Prop1() As String
12         Get
13             Return strValor
14         End Get
15         Set(value As String)
16             strValor = value
17         End Set
18     End Property
19 End Module
```

Um objeto em VB .NET tem métodos, campos e ***propriedades***.

Criando propriedades

Você pode configurar propriedades de um objeto como um array.

Exemplo:

```
1  Module Module1
2      Sub Main()
3          Module2.Prop1(5) = 1
4          MsgBox(Module2.Prop1(5))
5      End Sub
6  End Module
7
8  Module Module2
9      Private intDados(200) As Integer
10
11     Public Property Prop1(ByVal intIndice As Integer) As Integer
12         Get
13             Return intDados(intIndice)
14         End Get
15         Set(value As Integer)
16             intDados(intIndice) = value
17         End Set
18     End Property
19 End Module
```

Usando o Tratamento de Exceção Não Estruturada

- Sem o tratamento adequado dos erros que podem vir a acontecer no seu código, qualquer exceção que ocorra em seu programa é fatal e o programa irá parar.

- A declaração **On Error GoTo** permite o tratamento da exceção e especifica onde ela será tratada. A sua sintaxe é a seguinte:

On Error { **GoTo** [*line* | *0* | *1*] | **Resume Next** }

```
1  Module Module1
2      Sub Main()
3          Dim intNumero1 As Integer = 1000000000
4          Dim intNumero2 As Integer = 1000000000
5
6          Dim intMult As Integer
7
8          On Error GoTo Excecao
9          intMult = intNumero1 * intNumero2
10         System.Console.WriteLine("{0} * {1} = {2}.", intNumero1, intNumero2, intMult)
11         Exit Sub
12
13 Excecao:
14         System.Console.WriteLine("Erro: O tipo de dados Integer não suporta esta operação.")
15     End Sub
16 End Module
```

Usando o Tratamento de Exceção Não Estruturada

Você pode também tratar exceções específicas de formas diferentes dependendo de qual ocorreu por meio da propriedade **Number** do objeto **Err**.

Exemplo:

```
1  Module Module1
2      Sub Main()
3          Dim int1 = 0, int2 = 1, int3 As Integer
4          On Error GoTo Excecao
5          int3 = CInt(int2 / int1)
6          Exit Sub
7  Excecao:
8          If (Err.Number = 6) Then
9              System.Console.WriteLine("Erro de Número 6: Overflow!")
10         End If
11     End Sub
12 End Module
```

Usando o Tratamento de Exceção Não Estruturada

Outra maneira de verificar qual exceção foi capturada é usar as palavras **TypeOf** e **Is** em uma declaração **If** para testar a classe da exceção.

Exemplo:

```
1  Module Module1
2      Sub Main()
3          Dim int1 = 0, int2 = 1, int3 As Integer
4          On Error GoTo Excecao
5          int3 = CInt(int2 / int1)
6          Exit Sub
7  Excecao:
8          If (TypeOf Err.GetException() Is OverflowException) Then
9              System.Console.WriteLine("Erro de Overflow!")
10         End If
11     End Sub
12 End Module
```

Usando o Tratamento de Exceção Não Estruturada

A grande estrela do tratamento de exceção não estruturada é a declaração **Resume** que permite que o programa termine a sua execução mesmo que uma exceção tenha ocorrido.

Exemplo (com Resume Next):

```
1  Module Module1
2      Sub Main()
3          Dim int1 = 0, int2 = 1, int3 As Integer
4          On Error GoTo Excecao
5          int3 = int2 / int1
6          System.Console.WriteLine("Programa finalizado...")
7          Exit Sub
8  Excecao:
9          If (TypeOf Err.GetException() Is OverflowException) Then
10             System.Console.WriteLine("Overflow error!")
11             Resume Next
12         End If
13     End Sub
14 End Module
```

Usando o Tratamento de Exceção Não Estruturada

A declaração **Resume** *Line* define uma linha do código para onde o código continuará executando depois que uma exceção ocorrer.

Exemplo (com Resume Line):

```
1  Module Module1
2      Sub Main()
3          Dim int1 = 0, int2 = 1, int3 As Integer
4          On Error GoTo Excecao
5          int3 = int2 / int1
6  Nextline:
7          System.Console.WriteLine("Programa finalizado...")
8          Exit Sub
9  Excecao:
10         If (TypeOf Err.GetException() Is OverflowException) Then
11             System.Console.WriteLine("Overflow error!")
12             Resume Nextline
13         End If
14     End Sub
15 End Module
```

Usando o Tratamento de Exceção Não Estruturada

Para informações detalhadas sobre exceções, você pode usar as propriedades **Number** e **Description** do objeto **Err**.

Exemplo:

```
1  Module Module1
2      Sub Main()
3          Dim int1 = 0, int2 = 1, int3 As Integer
4          On Error GoTo Excecao
5          int3 = int2 / int1
6          System.Console.WriteLine("Programa finalizado...")
7  Excecao:
8          System.Console.WriteLine("Erro de número {0} ocorreu: {1}. Origem: {2}.", _
9                                   Err.Number, Err.Description, Err.Source)
10     End Sub
11 End Module
```

Referências Bibliográficas

- HOLZNER, Steven. **Visual basic.NET: black book**. Arizona: Coriolis Group Books, 2002. xxxviii, 1144 p ISBN 1-57610-835-X.