Macros no Excel

Roberto Guena

26 de novembro de 2012

Sumário

1 Introdução

O Excel, assim como os outros componentes do MS Office, pode ser programado através da linguagem interpretada VBA (Visual Basic for Aplications).[1] Os programas escritos em VBA são chamados de macros e funções. Há duas formas de definir uma macro. A primeira, mais fácil e mais limitada, é através da gravação de uma macro — o Excel é capaz de gravar uma sequência de comandos para, posteriormente, chamar a mesma sequência. A segunda maneira de definir uma macro é escrevendo seu código diretamente na janela de edição do VBA.

No prsente texto, iremos discutir os princípios de elaboração de macros e funções no MS Excel. Antes disso, registramos que é possível definir algumas macros sem escrever sequer uma linha de programação. Isso é feito através do gravador de macros.

1.1 Gravando macros

Imagine que você esteja trabalhando com um conjunto de planilhas com nomes de pessoas expressos em três colunas: uma com o último sobrenome, uma com o primeiro nome e outras com os nomes do meio, tal como ilustrado na Figura 1. Caso você queira criar uma coluna contendo o nome inteiro, pode inserir na célula B3 a fórmula

```
=E3 & " " & D3 & " " & C3.
```
[2]

mas você acha um tanto quanto desagradável inserir essa fórmula sempre sempre que precisar concatenar os três elementos de um nome. Claro, se os nomes divididos em três partes só aparecessem em uma tabela, você poderia inserir a fórmula na célula B3 e copiar essa célula para o intervalo B3:B7. Porém se essa divisão dos nomes aparecer em outras tabelas, você terá que ou inserir novamente a fórmula ou ir até a célula B3 e copiá-la para a coluna ao lado da nova tabela. Para evitar esse processo, você poderia gravar uma macro. Vejamos como isso é feito.

Primeiramente, certifique-se de que a faixa de opções Desenvolvedor esteja aparecendo. Para tal, clique em Arquivo, selecione Opções, Personalizar Faixa de Opções e deixe checada a caixa Desenvolvedor em Guias principais, conforme mostra a Figura 2. Clique OK.

Agora, você pode iniciar a gravação de sua macro. Selecione a faixa de opções Desenvolvedor. Clique em Usar Referências Relativas (logo explicamos porquê essa opção). Selecione a célula B3 e clique em Gravar Macro. Você verá uma caixa de diálogo similar à mostrada na Figura 3(a). No campo Nome da macro digite o nome que quer dar a sua macro, por exemplo nome_completo. Tal nome deve começar com uma letra e pode conter, além de letras, números

[1] Recurso similar é oferecido pelas suites LibreOffice e OpenOffice

[2] Alternativamente, você pode usar a função `CONCATENAR`, fazendo a fórmula da célula B3 igual a `=CONCATENAR(E3, " ", D3, " ", C3)`

	Sobrenome	Nome	Nome do meio
3	Andrade	Ana	Clara
4	Junqueira	Ana	Beatriz
5	Oliveira	Pedro	Luis
6	Mazoni	Júlia	Almeida
7	César	João	Pedro Cerqueira

Figura 1: Planilha com nomes em três partes

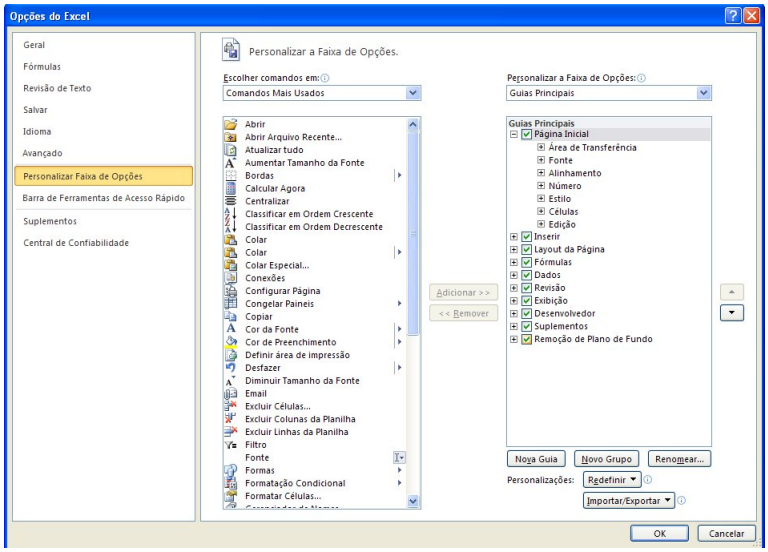
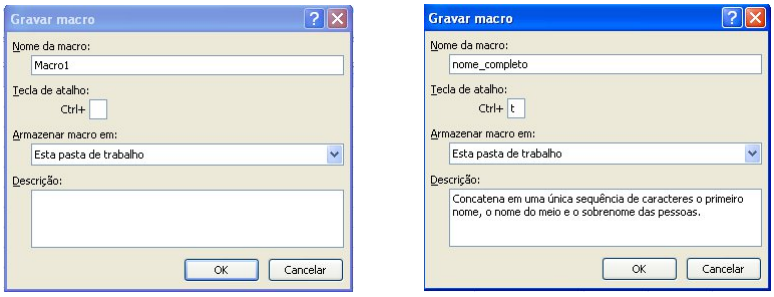


Figura 2: Para gravar macros, a caixa Desenvolvedor deve estar checada.

e carácteres de sublinhado. No campo Tecla de atalho escolha uma letra para que, combinada com a tecla Ctrl, sirva de atalho para acionar sua macro; por exemplo, escolha a letra j. No campo Armazenar essa macro em escolha esta pasta de trabalho caso queira usar sua macro apenas nessa pasta ou Pasta de trabalho pessoal de macros caso queira que sua macro possa ser usada sempre que você iniciar o Excel. A Figura 3(b) mostra um possível preenchimento dessa caixa de diálogo.

Clique OK e insira a fórmula para concatenar os nomes na célula B3: `= D3 & " " & E3 & " " & C3`, pressione Enter (ou Tab) e clique em Para gravação na faixa de opções Desenvolvedor. Pronto, sua macro está pronta para ser usada. Selecione a célula B4 e pressione Ctrl+t. O Excel deverá compor o segundo nome para você. Para ver uma forma alternativa de chamar a mesma



(a) (b)

Figura 3: Caixa de diálogo para início de gravação de uma macro antes de ser preenchida (a) e (b) depois de preenchida.

macro, selecione a célula B5, clique em Macros na faixa de opções Desenvolvedor, clique sobre nome_completo e escolha executar.[3]

1.2 Sobre a opção Usar Referências Relativas

Quando ao opção Usar Referências Relativas da faixa de opções Desenvolvidor não está selecionada, o Excel grava macros considerando as posições absolutas de cada célula. Por exemplo, caso tivéssemos gravado a macro nome_completo sem a opção Usar Referências Relativas selecionada, sempre que executássemos essa macro, o Excel inseriria a fórmula

```
= D3 & " " & E3 & " " & CE
```

na célula B3, *independentemente da célula selecionada ao executar a macro*. Como a opção Usar Referências Relativas estava selecionada quando gravamos a macro, quando ela é executada, o Excel insere na célula selecionada um fórmula que concatena o texto da célula duas colunas à direita da célula selecionada com o texto da célula três colunas à direita da célula selecionada e o texto da célula uma coluna à direita da célula selecionada. Assim, por exemplo, caso a macro nome_completo seja executada com a célula J502 selecionada, o Excel inserirá nesta célula a fórmula

```
= L502 & " " & M502 & " " & K502.
```

[3] Você ainda pode pressionar Alt+F8 para mostrar a mesma caixa de diálogo.

2 Escrevendo macros e funções

O recurso de gravação de macros permite um uso bastante limitado do recurso de macros do Excel. Você pode fazer muito mais caso escreva diretamente a sua própria macro. Isso é feito através da janela do VBA. Para acessar essa janela, você pode clicar em Visual Basic na faixa de opções Desenvolvedor ou teclar Alt+F11. Você deverá se defrontar com uma janela similar à mostrada na Figura 4.

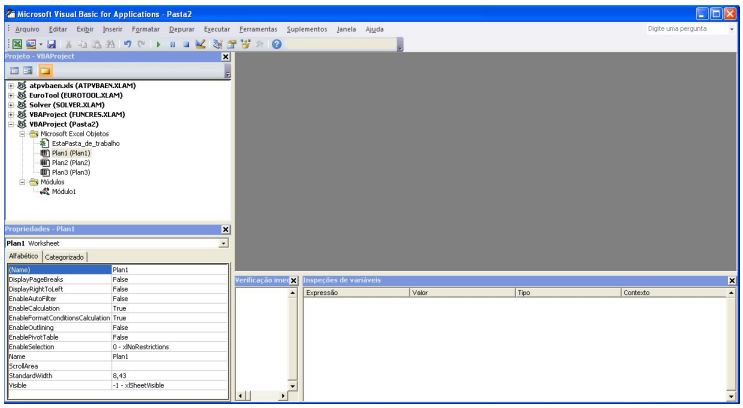


Figura 4: A janela do VBA.

Caso você clique duas vezes em Módulo1, será aberta uma janela, contendo o conteúdo da macro que você já gravou, que será semelhante a

```
Sub nome_completo()
'
' nome_completo Macro
' Concatena em uma única sequência de caracteres o primeiro nome,
', o nome do meio e o último sobrenome das pessoas.
'
' Atalho do teclado: Ctrl+t
'
  ActiveCell.FormulaR1C1 = "=RC[2] & "" "" & RC[3] & "" "" &RC[1]"
  ActiveCell.Select
End Sub
```

Toda macro começa com uma instrução `Sub` seguida do nome da macro e `()` e termina com a instrução `End Sub`. Todo texto após uma aspa simples é ignorado pelo Excel. Você deve usar aspas simples para inserir comentários em seu código. As linhas não iniciadas em aspas simples contém instruções enviadas ao VBA para executar a macro.

2.1 Primeira macro

Vamos agora escrever uma macro muito simples com algumas variações. Abaixo da linha `End Sub` que encerra a macro nome_completo insira o seguinte código:

```
Sub Ola()
'Pergunta o nome do usuário e envia uma saudação personalizada
  msgbox("Olá, " & inputbox("Escreva seu nome."))
End Sub
```

Vamos analisar esse código linha a linha:

- A primeira linha, `Sub Ola()` é uma declaração. Ela informa ao VBA que aqui começa uma macro cujo nome é Ola.

- A segunda linha começa com uma aspa simples (`'`) e, não será lida pelo VBA para efeito da execução da macro. Usamos ela para inserir uma descrição do que faz essa macro. De um modo geral todo texto em uma linha após uma aspa simples não é lido pelo VBA e dizemos que esse texto está *comentado*.

- A terceira linha contém as instruções de nossa macro. Vamos analisá-la por partes:

  - `inputbox(`*text*`)` é uma função. Tipicamente, uma função processa um argumento e retorna um resultado ao VBA. O argumento da função `inputbox` é um texto que constitui uma mensagem a ser enviada ao usuário. Essa função abre uma caixa de diálogo contendo esse texto, uma caixa de entrada na qual o usuário pode entrar alguma informaça e, por padrão, um botão de OK para encerrar o diálogo. Ela retorna ao VBA o valor ou texto que o usuário inseriu na caixa de diálogo.

  - O símbulo "&" denota um operador de texto que concatena duas sequências de caracteres, assim como faz na planilha do Excel. Desse modo `("Olá, "& inputbox("Escreva seu nome."))` diz ao VBA para acrescentar antes do texto retornado pela função inputbox o texto "Olá, ". Por exemplo, se alguém inserir na caixa de diálogo iniciada por nossa macro o nome "Almeida", essa expressão será convertiva em "Olá, Almeida".

  - Finalmente, a função `msgbox(`*texto*`)` instrui o VBA para apresentar uma caixa de diálogo contento a mensagem *texto* e um botão OK para encerrar o diálogo.

- A última linha, contendo `End Sub` informa o VBA que a macro termina aqui.

Experimente agora rodar a macro. Para tanto, caso você esteja na janela do VBA posicione o cursor em qualquer ponto do código da macro e clique sobre o ícone  ou, simplesmente, tecle F5. Se você quiser inicial a macro a partir do Excel, clique em Macros na faixa de opções Desenvolvedor ou tecla Alt+F8, selecione Ola e clique no botão Executar.

Essa primeira macro contém duas características essenciais da maioria dos programas de computador: ela possui uma *entrada* que é a informação coletada na primeira caixa de diálogo e retorna uma *saída* que é a segunda caixa de diálogo. Quase todos os programas relevantes coletam informações externas a eles via algum mecanismo de entrada que pode ser uma caixa de diálogo, uma informação contida em um arquivo, etc. *Todos* programas relevantes executam uma saída que pode consistir em uma caixa de mensagem, na alteração de um arquivo, na chamada de outro programa, etc.

Vejamos uma macro similar, mas que difere na forma que opera com entradas e saídas de dados.

```
Sub Olab()
'
'Coleta o nome na célula ativa e insere na célula ao lado
'uma saudação personalizada
'
  Application.ActiveCell.Offset(0,1).formula = "Olá, " _
                                      & Application.ActiveCell
End Sub
```

A palavra `Application` refere-se ao aplicativo do Excel. Trata-se de um caso particular de um *objeto*. Um objeto é algo que contém dados estruturados na forma de valores, textos e propriedades. Por exemplo, o objeto *range* corresponde a um intervalo em uma planila do Excel e contém as fórmulas inseridas em cada célula, e as informações sobre a formatação de cada célula. Quando escrevemos `Application.ActiveCell`, aplicamos ao objeto Application o método ActiveCell. Um método é usado para alterar ou ler uma propriedade do objeto ou para retornar um objeto mais específico contido no objeto original. No caso `Application.ActiveCell` retorna o objeto range correspondente à célula ativa. `Application.ActiveCell.Offset(0,1)` retorna o intervalo correspondente à célula na mesma linha da célula ativa e uma coluna à direita da célula ativa. Finalmente, `Application.ActiveCell.Offset(0,1).formula = "Olá, "& Application.ActiveCell` indica que a fórmula dessa célula deve ser "Olá, " mais o conteúdo da célula ativa. O símbolo de sublinhado precedido de um espaço (_) indica ao VBA que o comando continua na próxima linha.

Vamos testar essa macro. Em uma célula qualquer de sua planilha digite um nome. Com essa célula selecionada, mande executar a macro "Olab". Na célula ao lado deverá aparecer o nome que você digitou precidido de "Olá, ".

2.2 Primeira função

Uma outra forma de programa no VBA são as funções. Uma função é um programa que usualmente demanda um ou mais argumentos e retorna algum tipo de dados. Para construir nossa primeira função digitemos abaixo do fim da macro "Olab", o seguinte código:

```
Function Oi(nome)
'Gera uma saudação personalizada a partir de um nome
Oi = "Olá, " & nome
End Function
```

A primeira linha desse código simplesmente informa o VBA que aqui começa uma função cujo nome é "Oi" e cujo único argumento será chamado no código de nome, isto é, sempre que a palavra `nome` aparecer no código, o VBA deverá substituí-la pelo argumento da função. A segunda linha contém informação adicional sobre a função. A terceira linha diz ao VBA que essa função deve retornar o texto que concatena "Olá " ao argumento da função. A quarta linha informa ao VBA que a função termina aqui.

Insira agora em uma célula qualquer de sua planilha a fórmula `=Oi("João")`. O Excel deverá mostrar nessa célula Olá, João. A função que você acabou de criar pode ser chamada tanto de sua planilha quanto por outras funções e macros do VBA. Por exemplo, a macro "Ola" continuará funcionando perfeitamente se seu código for substituído por

```
Sub Ola()
'Pergunta o nome do usuário e envia uma saudação personalizada
MsgBox (Oi(InputBox("Entre seu nome")))
End Sub
```

2.3 Funções sem argumento

Nem toda função precisa ter um argumento. Por exemplo, se você inserir em uma célula de uma planilha do Excel `= pi()` e teclar Enter, verificará que o Excel exibirá nessa célula uma aproximação decimal da constante matemática π. A função `pi()` do Excel é um exemplo de uma função sem argumentos.

Vamos também criar uma função sem argumentos. Desde os antigos gregos considera-se que um retângulo cuja razão entre o seu lado mais longo e seu lado mais curto seja igual a

$$\frac{1+\sqrt{5}}{2}$$

possui uma estética ideal. A razão assim definida é conhecida como proporção áurea. Vamos criar uma função sem argumento que retorna a representação decimal de tal proporção. Para tal, basta que escrevamos o código

```
Function Aurea()
area = (1 + 5 ^ 0.5) / 2
End Function
```

2.4 Operadores aritméticos do VBA

A função que acabamos de definir executa algumas operações aritméticas. A Tabela 1 abaixo apresenta todos os operadores aritméticos disponíveis no VBA

Operador	operação realizada	Observação
`^`	exponenciação	
`-`	negação	`-` quando usado para representar o negativo de um número, não a operação de subtração.
`*`	multiplicação	
`/`	divisão	
`\`	divisão de inteiros	por exemplo, `5 \ 2` retorna 2, visto que 5 dividido por 2 dá 2 com resto 1. Se um dos números não for inteiro, o VBA arredondará esse número para o inteiro mais próximo. Por exemplo `5 \ 1.6` retornará 2, pois 1.6 será arrendondado para 2.
`Mod`	resto de uma divisão	exemplo: `5 Mod 2` retorna 1. Se um dos argumentos desse operador não for um inteiro, o VBA arredondará esse argumento para o inteiro mais próximo. Por exemplo, `5 Mod 1.6` retorna 1, pois 1.6 é arredondado para 2.
+	adição	
−	subtração	

Tabela 1: Operadores aritméticos do VBA

Os operadores foram listados na ordem em que são executados pelo VBA, com exceção das operações de multiplicação (
texttt*) e divisão (`/`) que são executadas simultaneamente assim como as operações de adição (`+`) e subtração (`-`). Assim, por exemplo, a expressão

```
-3 ^ 2 + 5 * 8 - 13 \ 2 * 3
```

é resolvida pelo VBA na seguinte sequência

```
1 -3 ^ 2 + 5 * 8 - 13 \ 2 * 3
2 -9 + 5 * 8 - 13 \ 2 * 3
3 -9 + 40 - 13 \ 6
```

```
-9 + 40 - 2
29
```

Essa sequência pode ser alterada empregando-se parênteses da forma usual. Por exemplo, a expressão

```
-3 ^ (2 + 5) * 8 - (13 \ 2) * 3
```

é resolvida pelo VBA na seguinte sequência

```
-3 ^ (2 + 5) * 8 - (13 \ 2) * 3
-3 ^ 7 * 8 - 6 * 3
-2187 * 8 - 6 * 3
-17496 - 18
-17514
```

3 Constantes e variáveis

3.1 Constantes

Uma constante é simplesmente um nome que damos a um dado qualquer. Por exemplo, suponha que, em uma macro, você use recorrentemente o número 3,141 593 como aproximação da constante matemática π. Você pode, evidentemente, digitar esse número sempre que tiver necessidade de usá-lo, mas talves julque mais conviniente fazer a seguinte declaração:

```
Const pi = 3.141593
```

Essa declaração diz ao VBA que, sempre que ele se deparar com a palavra `pi` ele deve substituí-la pela valor 3,141 592 653 589 793 238 44. Em outras palavras, para o VBA, `pi` passa ser uma *constante* com valor 3,141 592 653 589 793 238 44.

```
Function area_circulo(raio)
Const pi = 3.14159265358979323844 'Sempre que aparecer a palavra
                                  'pi o VBA entenderá
                                  'entenderá 3.14159265358979323844
area_circulo = pi * raio ^ 2      'Aqui usamos a fórmula da área
                                  'de um círculo
End Function
```

Exercício 1 *Sabendo que a fórmula para o cálculo do volume de uma esfera é*

$$V = \frac{4}{3}\pi r^3$$

na qual V é o volume da esfera e r é seu raio, escreva uma função que calcule o volume de uma esfera a partir de seu raio.

3.2 Variáveis

Após definir uma constante, você não pode fazer o VBA alterar seu valor. Caso você precise definir um nome para um valor que pode ser alterado quando o código VBA é executado, você deve criar uma variável. Se uma constante é um nome que é atribuído a um dado específico, podemos pensar uma variável como o nome dado a um "recipiente" para um dado.[4] Podemos trocar o dado que está no recipiente. Quando o VBA lê o nome de uma variável, ele busca o dado que está naquele momento nessa variável.

Para criar uma variável, inserimos no código do VBA uma declaração tal como

```
Dim nome_da_variável
```

na qual `nome_da_variável` é qualquer palavra formada por letras, números e subscritos que comece com uma letra.

Para atribuir um dado a uma variável usamos o símbulo de "=". Por exemplo, suponha um código VBA tenha as duas linhas abaixo:

```
Dim a
a = 5
```

Quando executado, esse código cria a variável `a` (primeira linha) e atribuia a ela o valor 5 (segunda linha). Caso à direita do símbulo "=" haja uma expressão, o VBA primeiramente avalia essa expressão e, em seguida, atribui obtido à variável. Por exemplo o código

```
Dim a
a = 5 + 2
```

diz para o VBA criar a variável `a`, calcular a expressão "`5+2`" e atribuir à variável `a` o resultado desse cálculo, isto é, o número 7. Isso permite que se atribua um valor novo a uma variável baseado em uma expressão que dependa do valor antigo dessa variável.

Para dar um exemplo simples, experimente escrever a macro abaixo

```
Sub Macro1
Dim a
a = 1
a = a + 1
MsgBox("Agora o valor da variável a é " & a)
End Sub
```

Ao ler a quarta linha desse código, o VBA executa primeiro a intrução "pegue o valor guardado na variável `a` (1) e acrescente o número 1", ou seja, ele avalia a expressão à direita do símbolo "=", e depois atribui à variável `a` esse valor. Assim, ao executar a quarta linha, o VBA deverá exibir uma caixa de mensagem

[4] Tecnicamente, o "recipiente" é um espaço reservado na memória do computados.

com a mensagem Agora o valor da variável a é 2. Conforme veremos em alguns exemplos mais à frente, essa possibilidade de atribuir a uma variável o valor de uma expressão que depende do valor atuar dessa variável é bastante útil em diversos problemas de programação.

3.3 Omissão de variáveis e opção explicit

Experimente excluir a segunda linha da macro acima de tal sorte que seu códio fica como abaixo:

```
Sub Macro1
a = 1
a = a + 1
MsgBox("Agora o valor da variável a é " & a)
End Sub
```

Como você, pode ver, a variável a não é declarada pela instrução Dim. Ainda assim, o código é executado normalmente pelo VBA. De fato, quando o VBA lê uma linha de atribuição de valore a uma variável não declarada, ele cria essa variável automaticamente. O problema com a criação automática de variáveis não declaradas é que seu código pode não funcionar como planejado em virtude de um eventual erro de digitação. Por exemplo, imagine que, ao digitar a Macro1, você tenha inserido o seguinte código:

```
Sub Macro1
Dim a
a = 1
s = a + 1
MsgBox("Agora o valor da variável a é " & a)
End Sub
```

Na quarta linha, você digitou, por engano, a letra "s" no lugar da letra "a". Nesse caso, ao ler a quarta linha, o VBA irá criar a variável s e atribuir a ela o valor da soma do conteúdo da variável a mais 1. Como essa operação não afeta o valor da variável a, a mensagem exibida ao final da macro será Agora o valor da variável a é 1 e não Agora o valor da variável a é 2, conforme você pretendia.

Para evitar esse tipo de erro, muitos programadores preferem forçar que todas as variável sejam explicitamente declaradas antes de serem usadas. Para fazer isso, é preciso inserir, antes da declaração de qualquer macro ou função, o seguinte código:

```
Option Explicit
```

Se você fizer isso, sempre que o VBA se deparar com uma variável que não foi definida explicitamente, emitirá a seguinte mensagem de erro: Erro de compilação: variável não declarada. Nesse caso, você terá certeza que, caso seu código seja executado, nenhum nome de variável foi digitado erroneamente.

3.4 Tipos de dados

Conforme dissemos, uma variável pode ser pensada como um recipiente que acomodar algum tipo de dados. Assim como no mundo real escolhemos o tamanho do recipiente de acordo com os objetos que pretendemos colocar neles, podemos escolher o tamanho da memória reservada a uma variável de acordo com o tamanho da informação que pretendemos armazenar nela. Para fazer isso, devemos escolher o *tipo* de dado que a variável deverá conter. A Tabela 2 descreve os tipos de dados disponíveis no VBA.

Na prática, os tipos de dados mais usados são o Integer e o Longo, usados para armazenar números inteiros, o Double, usado para armazenar aproximações de números reais na forma de ponto flutuante, o Boolean usado para armazenar os valores de verdade "verdadeiro"(True) ou "falso"(False) e o String usado para armazenar sequências textuais de caracteres. Além desses tipos, há um tipo genérico de dados chamado Variant que aceita qualquer tipo de informação, isto é, números inteiros, números reais, textos e booleanos. Quando você declara uma variável sem especificar seu tipo, ou quando você usa uma variável sem declará-la explicitamente, o VBA criará essa variável como uma variável do tipo Variant, isto é, que pode conter qualquer tipo de dado.

Se quisermos especificar o tipo de dados de uma variável explicitamente, devemos declará-la na forma:
`Dim` *nome da variável* `as` *tipo da variável*.

Por exemplo, a declaração

```
Dim MinhaVariavel As Long
```

instrui o VBA a criar a variável "MinhaVariavel" e fazer com que ela só aceite dados com o formato de números inteiros. Já a instrução

```
Dim MeuTexto As String
```

instrui o VBA a criar a variável "MeuTexto" e impor que os dados dessa variável serão sequências textuais de caracteres.

Frequentemente os programadores preferem declarar explicitamente as variáveis que vão usar em seus programas. Entre as razões para tal podem-se citar

1. Diferentemente do VBA, algumas linguagens de programação obrigam que todas variáveis sejam declaradas explicitamente assim como seus tipos.

2. A declaração das variáveis e de seus tipos tornam os códigos mais claros para os humanos.

3. Em programas longos ou de execução complexa, a escolha de tipos de variáveis que ocupem menor espaço de memória pode implicar um ganho de velocidade de execução significativo.

Tabela 2: Tipos de dados do VBA

Tipo de dado	Tamanho	Intervalo
Byte	1 byte	de 0 a 255
Boolean	2 bytes	True ou False
Integer	2 bytes	de -32.768 a 32.767
Long	4 bytes	de -2.147.483.648 a 2.147.483.647
LongLong	8 bytes	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807 (válido somente em plataformas de 64 bits)
LongPtr	4 bytes em sistemas de 32 bits, 8 bytes em sistemas de 64 bits	-2.147.483.648 a 2.147.483.647 em sistemas de 32 bits, -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807 em sistemas de 64 bits
Single	4 bytes	de –3,402823E38 a -1,401298E-45 para valores negativos; de 1,401298E-45 a 3,402823E38 para valores positivos
Double	8 bytes	de –1,79769313486231E308 a-4,94065645841247E-324 para valores negativos; de 4,94065645841247E-324 a 1,79769313486232E308 para valores positivos.
Currency	8 bytes	de -922.337.203.685.477,5808 a 922.337.203.685.477,5807
Decimal	14 bytes	+/-79.228.162.514.264.337.593.543.950.335 sem vírgula decimal;+/-7,9228162514264337593543950335 com 28 casas decimais à direita; o menor número diferente de zero é +/-0,0000000000000000000000000001. Esse tipo de dados só pode ser usado dentro de uma Variant.
Date	8 bytes	De 1 de janeiro de 100 a 31 de dezembro de 9999
String	10 bytes + comprimento da sequência	De 0 a aproximadamente 2 bilhões
String	Comprimento da seqüência	De 1 a aproximadamente 65.400
Variant	16 bytes	Qualquer valor numérico até o intervalo de um Double
Variant	22 bytes + comprimento da sequência	O mesmo intervalo de String de comprimento variável
Definido pelo usuário	Número requerido por elementos	O intervalo de cada elemento é igual ao intervalo do seu tipo de dados.

4. A declaração do tipo da variável ajuda na verificação do seu código, pois se você tentar inserir em uma variável uma informação incompatível com o tipo usado para declarar essa variável, o VBA emitirá uma mensagem de tipos incompatíveis apontando onde está o erro em seu código. Por exemplo, imagine um código contendo as duas linhas abaixo:

   ```
   Dim a As Double
   a = "b"
   ```

   Na primeira dessas linhas, o VBA é instruído a criar a variável `a` como uma variável do tipo Double que armazena números reais em representação decimal. Na segunda linha, o VBA é instruído a armazenar o caractere "b" na variável `a`. Como você disse ao VBA que `a` é do tipo Double, quando esse código for executado, o VBA abrirá uma caixa de mensagem informando que os tipos são incompatíveis.

A última razão acima não é muito efetiva no VBA pois, diferentemente do que ocorre em outras linguagens, este interpretador, sempre que possível, realiza a conversão automática de tipos ao invés de apontar incompatibilidade de tipos. Para ver isso, experimente rodar a macro abaixo

```
Sub teste()
Dim a As Long 'Define a variável a como do tipo Long, que só
              'aceita valores inteiros
a = 3.6       'Atribui valor não inteiro a uma variável tipo
              'Long que só admite valores inteiros
MsgBox a
End Sub
```

Como a variável "a" foi definida com o tipo Long, ela só admite valores inteiros. Quando o código solicita para atribuir o valor 3,6 à variável "a", o VBA, ao invés de emitir uma mensagem dizendo que a variável "a" não admite esse tipo de valor (com dígitos depois da vírgula), força a conversão de 3,6 em um inteiro, arredondando esse número para o inteiro mais próximo. Por isso, a caixa de mensagem mostra, quando executamos a macro o número 4.

Para tentar aumentar a transparência de nossos códigos, A partir de agora, procuraremos sempre definir as variáveis que pretendemos usar assim como seus tipos.

3.5 Declaração de tipos em funções

Também podemos definir tipos em quando declaramos funções. Para exemplificar, tomemos a função "area_circulo" que definimos na seção 3.1. Para deixar claro qual o tipo do argumento dessa função assim como o tipo de dado que ela retorna, escrevemos

```
Function area_circulo(raio As Double) As Double
'
'calcula a área de um círculo em função de seu raio
'a declaração raio As Double deixa claro que o argumento
'da função é um número que pode ser não inteiro.
'a declaração "As Double" ao final da linha indica que a
'função retorna um valor numérico que pode conter casas decimais
'
Const pi = 3.14159265358979323844 'Sempre que aparecer a palavra
                                  'pi o VBA entenderá
                                  'entenderá 3.14159265358979323844
area_circulo = pi * raio ^ 2      'Aqui usamos a fórmula da área
                                  'de um círculo
End Function
```

A expressão `raio As Double` na primeira linha é usada para indicar que o argumento da função será tratado como a variável de nome "raio" e tipo "Double" (número com ponto flutuante). O `As Double` ao final da linha indica que a função "area_circulo" retorna um dado do tipo "Double".

4 Estruturas de decisão

4.1 Instrução if

Um dos recursos mais importantes das linguagens de programação é a capacidade de fazer comparações entre valores e definir ações condicionadas ao resultado dessas comparações. No VBA isso é implementado pela instrução `If`. A instrução `if` tem a forma
If *condição* Then *instruções*

Na qual *condição* é uma expressão que pode assumir o valor True ou False e *instruções* são as instruções a serem seguidas caso *condição* seja avaliada como True. Caso prefira dividir essa instrução em mais de uma linha de código, pode usar a forma
If *condição* Then
instruções
End If

Para um exemplo simples, considere a seguinte variação da macro "Ola" que apresentamos na seção 2.1:

```
Sub Ola()
'
'Pergunta o nome do(a) usuário(a) e envia
'uma saudação personalizada
'
Dim nome as string     'Variável para armazenar o nome do usuário
```

```
Dim mensagem as string 'Variável para armazenar a mensagem
Dim hora as double     'Variável para armazenar a hora
'Pede o nome do(a) usuário(a) e o armazena na variável "nome":
nome = inputbox("Escreva seu nome.")
'Calcula a hora corrente e armazena na variável "hora":
hora = hour(time())
mensagem = "Olá, " & nome & "."     'Cria a mensagem básica
'Se for de madrugada, manda o(a) usuário(a) ir dormir:
If hora < 6 Then
    mensagem = mensagem & " É madrugada, vá para a cama!"
End If
'Envia a mensagem em uma caixa de mensagem:
msgbox(mensagem)
End Sub
```

Nesse exemplo, usamos as funções "hour" e "time". Quando o VBA lê time() ele retorna o horário corrente (horas, minutos e segundos), quando ele lê `hour(time())`, ele retorna apenas a hora do horário corrente. Assim, o código acima atribui à variável nome o valor da hora corrente. O nosso interesse aqui está nas linhas

```
If hora < 6 Then
    mensagem = mensagem & " É madrugada, vá para a cama!"
End If
```

Elas dizem para o VBA que, caso seja antes de 6hs da manhã (`hora < 6`), ele acrescente à mensagem a ser enviada o texto "É madrugada, vá para a cama!".

4.2 Operadores de comparação e operadores lógicos

Quando escrevemos `hora < 6` na macro acima, empregamos o *operador de comparação* <, cujo significado é evidente. Segue uma lista do operadores de comparação do VBA que usaremos ao longo desse texto:

`<` é menor que. Exemplos: `2 < 5` retorna `True` e `1.5 < 1.2` retorna `False`.

`<=` é menor ou igual a. Exemplos: `5 <= 5` retorna `True` e `6 <= 4` retorna `False`

`>` é maior que. Exemplos: `2 < 5` retorna `False` e `1.5 > 1.2` retorna `True`.

`>=` é maior ou igual a. Exemplos: `5 >= 5` retorna `True` e `4 >= 5` retorna `False`

= é igual a. Exemplos: `2 = 1` retorna `False` e `3 = 3` retorna `True`.

<> é diferente de. Exemplos: `2 <> 1` retorna `True` e `1 <> 1` retorna `False`.

O VBA apresenta mais dois operadores de comparação além dos citados acima: o operador `Like`, usado para comparar padrões de texto e o operador `Is` usado para comparar objetos. Não usaremos esses operadores aqui.

Os operadores de comparação são executados após os operadores aritméticos, a menos que o uso de parênteses indique o contrário.

Além dos operadores de comparação,também usaremos os chamados operadores lógicos. Os operadores lógicos são executados após os operadores de comparação. Tratam-se de operadores que realizam operações entre booleanos. São eles

And e lógico. Exemplo `1 > 0 And 2 < 3` retorna `True`.

Or ou lógico. Retorna `True` caso ao menos um dos operandos seja verdadeiro. Exemplos: `1 < 2 Or 4 = 4` retorna `True`, `1 = 1 Or 1 = 2` retorna `True`, `1 = 2 Or 1 > 3` retorna `False`.

Not não lógico, transforma `True` em `False` e `False` em `True`. Exemplos: `Not 2 = 2` retorna `False`, `Not 1 > 2` retorna `True`.

Xor ou exclusivo. `a Xor b` retorna `True` quando `a` é `True` e `b` é `False` ou quando `b` é verdadeiro e `a` é `False`. Caso contrário, retorna `False`. Exemplos: `1 = 1 Xor 3 > 2` retorna `False`, `1 = 1 Xor 3 < 2` retorna `True`, `0 = 2 Xor 1 > 9` retorna `False`.

Imp implicação lógica. `a Imp b`, em que `a` e `b` são dois booleanos, retorna `True` caso `a` seja `False` ou caso `a` e `b` sejam ambos `True`. Caso contrário, retorna `False`. Exemplos: `1 > 2 Imp 3 = 3` retorna `True`, `1 > 2 Imp 4 < 3` retorna `True`, `1 = 1 Imp 3 <> 3` retorna `False`.

Eqv equivalência lógica. `a Eqv b` retorna `True` caso `a` e `b` sejam ambos `True` ou caso `a` e `b` sejam ambos `False` e retornam `False` caso contrário. Exemplos: `1 <> 1 Eqv 5 < 4` retorna `True`, `1 = 1 Eqv 5 > 4` retorna `True`, `1 = 1 Eqv 5 < 4` retorna `False`.[5]

Não se preocupe muito com os operadores `Imp` e `Eqv`. Eles não serão usados aqui.

4.3 Combinando a instrução If com as palavras-chave Else e ElseIf

Imagine que o professor Epaminondas tenha calculado as notas finais de seus alunos em uma planilha do Excel tal como descreve a Figura 5. Esse professor

[5]Na presente lista, pressupomos que os operandos das operadores `Xor`, `Imp` e `Eqv` assumam todos valores `True` ou `False`. Esses operadores admitem, todavia que um dos operandos possa ter valor nulo (`Null`). Furtamo-nos aqui dessa discução técnica. Para uma descrição completa do comportamento desses operadores quando um dos operandos é `Null`, consulte a ajuda do VBA.

deseja acrescentar a essa tabela uma coluna com o título "situação" que assuma valor "aprovado(a)" para o aluno ou a aluna que tenha obtido nota não inferior a 5, o valor "recuperação" para o aluno ou a aluna que tenha obtido nota inferior a 5 e não inferior a 3 e o valor "reprovado(a)" para o aluno ou a aluna que tenha obtido nota inferior a 3.

Nome	1ª prova	2ª prova	Média
Ana Amália Fagundes	6,5	5,0	5,5
Josimar Silva	6,0	8,2	7,5
Júlio Julião Vergueiro	0,5	4,0	2,8
Maria Carolina Vasconcellos	7,5	9,0	8,5
Pedro Aguiar	9,0	8,5	8,7
Raquel Gusmão	8,3	7,1	7,5
Rômulo Rêmulo Arantes	4,4	4,0	4,1
média	6,0	6,5	6,4

Figura 5: Notas dos alunos de um professor hipotético.

Epaminondas sabe como fazer essa coluna usando as fórmulas do Excel. Porém, como ele tem muitas turmas, gostaria de definir uma função que gerasse automaticamente a situação do aluno ou da aluna sem que ele precisasse digitar uma fórmula algo tediosa e pouco transparente do tipo `=if(E3>5,"aprovado(a)",if(E3>3,"recuperação", "reprovado(a)"))`. Para tal, ele usa a instrução `if` e gera a função `situacao` descrita no código abaixo

```
Function situacao(nota As Double) As String
If nota >= 5 Then situacao = "aprovado(a)"
If nota < 5 And nota >= 3 Then situacao = "reavaliação"
If nota < 3 Then situacao = "reprovado(a)"
End Function
```

Após escrever esse código em um módulo no editor do VBA, ele digita na célula F3 a fórmula `=situacao(E3)` e verifica que o Excel responde mostrando nessa célula, conforme esperava, aprovado(a). Ele copia a célula F3 para o intervalo F3:F9 e vê que a função que ele criou funciona perfeitamente. Porém, ele não está plenamente satisfeito. O código que ele digitou ao definir a função "situacao" não deixa claro que a segunda instrução if só terá efeito nos casos em que a primeira não tem e que a terceira instrução if só terá efeito caso as duas primeiras não tenham.

Depois de pesquisar um pouco, Epaminondas descobre que há uma forma mais elegante de construir sua função. Na ajuda do VBA, ele descobre que a instrução If pode ser colocada na forma

```
If teste 1 Then
   instruções 1
ElseIf teste 2 Then
   instruções 2
ElseIf teste 3 Then
   instruções 3
        ⋮
ElseIf teste n Then
   instruções n
Else
   instruções n+1
End If
```

na qual *texte 1*, *texte 2* ... *teste n* são expressões que retornam um booleano e *instruções 1*, *instruções 2* ... *instruções n+1* são instruções passadas ao VBA condicionalmente a esses testes. Quando o VBA se defronta com um código com essa estrutura, ele primeiramente verifica se *teste 1* é uma expressão que resulta em `True`. Se for esse o caso, ele executa as *instruções 1* ignora todas as linhas que se seguem até onde aparece o `End If`. Caso, todavia, a expressão *teste 1* resulte em `False`, o VBA verifica a expressão *teste 2*. Caso ela seja verdadeira ele executa as *instruções 2*. Caso ela seja falsa, ele verifica a expressão *teste 3* e assim, sucessivamente. Caso nenhuma das condições *teste 1*, *teste 2* ... *teste n* resulte em `True`, o VBA executa as instruções *instruções n+1*.

Vejamos como fica a função "situacao" quando escrita usando-se essa estrutura

```
Function situacao(nota As Double) As String
If nota >= 5 Then
  situacao = "aprovado(a)"
ElseIf nota >= 3 Then
  situacao = "reavaliação"
Else
  situacao = "reprovado(a)"
End If
End Function
```

Vejamos como o VBA executará essa função. Ele começa testando se o argumento dessa função é maior ou igual a 5. Nesse caso, a função retorna "aprovado(a)". Isso é o que ocorre quando Epaminondas insere na célula `F3` a fórmula `=situacao(E3)`. Caso o argumento da função "situacao" não seja maior ou igual a 5, então, o VBA verifica se esse argumento é maior ou igual a 3. Se esse for o caso, ele faz com que a função retorne "reavaliação". É isso que ocorre quando Epaminondas insere na célula `F9` a fórmula `=situacao(E9)`, pois o valor da célula `E9` não é maior ou igual a 5 e é maior ou igual a 3. Finalmente, caso o argumento da função "situacao" não seja maior ou igual a 5 ou sequer

maior ou igual a 3, o VBA executa as instruções que vêm após a palavra chave `Else`, isto é faz com que a função "situacao" retorne "reprovado(a)".

Exercício 2 *O* índice de massa corporal *(IMC) é uma medida usada para calcular se a pessoa está no peso ideal. Sua fórmula é*

$$IMC = \frac{massa\ (Kg)}{Altura^2(\mathrm{m}^2)}.$$

Por exemplo, uma pessoa com $1{,}72\,\mathrm{m}$ *de altura e* $70\,\mathrm{Kg}$ *de peso apresenta índice de massa corporal igual a*

$$\frac{70}{1,72^2} \approx 23,66\frac{\mathrm{Kg}}{\mathrm{m}^2}$$

A Organização Mundial de Saúde classifica a população adulta de acordo com o IMC conforme a Tabela 3. Faça uma macro que solicite que a usuária informe

IMC $\left(\frac{\mathrm{Kg}}{\mathrm{m}^2}\right)$	Classificação
menos de 16,00	magreza severa
entre 16,00 e 16,99	magreza moderada
entre 17,00 e 18,49	magreza suave
entre 18,50 e 24,99	peso normal
entre 25,00 e 29,99	pré obesidade
entre 30,00 e 34,99	Obesidade classe I
entre 35,00 e 39,99	Obesidade classe II
acima de 40,00	Obesidade classe III

Tabela 3: A classificação internacional de adultos de acordo com o IMC.

sua altura e peso, calcula seu IMC e gera sua classificação de acordo com a Tabela 3.

4.4 A instrução Select Case

Quando a instrução If... Then... Else... tem por condição uma expressão envolvendo o conteúdo de uma única variável, você pode substitui-la pela instrução Select Case. Essa instrução tem a forma

```
Select Case var
  Case exp_1
    inst_1
  Case exp_2
    inst_2
    ⋮
  Case exp_n
```

```
    inst_n
  Case Else
    inst_n+1
End Select
```

na qual

var é uma variável.

exp_ 1, exp_ 2... exp_ n são expressões relativas à variável *var* que podem assumir uma das seguintes formas:

- Uma lista de valores tal como, no caso de uma variável numérica, `1, 3, 6, 9`. Se o valor da variável *var* for um dos valores listados nessa expressão, então o VBA a considerará verdadeira.
- Um intervalo de valores do tipo `1 To 8`, significando, no exemplo, qualquer valor entre 1 e 8, inclusive esses valores. Evidentemente os números 1 e 8 são apenas ilustrativos. A única regra a seguir é que o número que precede a palavra `To` deve ser maior do que o número que vem depois dessa palavra. Se o valor da variável *var* estiver no intervalo descrito por essa expressão, o VBA a considerará verdadeira.
- Uma expressão do tipo `Is = 1` ou `Is > 1` ou `Is < 1`. O VBA considerará verdadeira a expressão `Is = 1` caso o valor da variável *var* seja igual a 1. Do mesmo modo, o VBA considerará verdadeira as expressões `Is < 1` ou `Is > 1` caso o valor da variável *var* seja, respectivamente, menor do que 1 ou maior do que 1.

inst_ 1, inst_ 2... inst_ n* e *inst_ n+1 São conjuntos de instruções a serem executadas condicionalmente. O conjunto de instruções $inst\_i$ ($i = 1, \ldots, n$) será executado se a expressão exp_i for avaliada como verdadeira.

A função situação pode ser reescrita substituindo a instrução If Then Else pela instrução Select Case ficando como abaixo:

```
Function situacao(nota As Double) As String
Select Case nota
  Case Is >= 5
    situacao = "aprovado(a)"
  Case Is >= 3
    situacao = "reavaliação"
  Case Else
    situacao = "reprovado(a)"
End Select
End Function
```

Exercício 3 *Refaça o Exercício 2 usando a instrução Select Case.*

5 Desvios e loops

5.1 Rótulos e a instrução `GoTo`

Uma das principais razões pelas quais usamos programas de computador é a possibilidade de fazer com que os computadores executem em nosso lugar tarefas repetitivas. Em um programa, as tarefas repetitivas usualmente ocorrem quando o computador lê repetidas vezes o mesmo conjunto de instruções. O VBA oferece diversas instruções que podem ser usadas para fazer com que o computador leia repetidas vezes as mesmas linhas. Dentre essas instruções, a mais simples e mais flexível, embora também a possivelmente menos usada é a instruções GoTo. Para usar a instrução GoTo, precisamos *rotular* uma linha de nosso código. Fazemos isso escrevendo um nome qualquer (o rótulo) seguido de dois pontos. Depois, quando queremos instruir o VBA a ler os comandos que iniciam a partir da linha rotulada, basta inserir a instrução GoTo seguida do nome usado para rotular essa linha.

Ilustremos isso com um exemplo. Vamos fazer um jogo muito simples no qual o computador seleciona um número inteiro entre 0 e 9 e o jogador deve tentar adivinhar que número é esse. Para o sorteio do número usaremos a função `rnd` que retorna um número aleatório maior ou igual a zero e menor do que 1. Para a função `rnd` se comportar adequadamente, precisamos fazer com que a instrução `randomize` a preceda. Para transformar esse número em um inteiro entre 0 e 9, usaremos a função `Int` que converte um número qualquer em um número inteiro simplesmente ignorando sua parte decimal. Em síntese, as linhas usadas para sortear aleatoriamente um número inteiro entre 0 e 9 e armazená-lo na variável `numero` são

```
Randomize
numero = Int(10*Rnd())
```

O código de nosso programa, com diversos comentários, é o seguinte:

```
1  Sub adivinha()
2  'Sorteia um número inteiro de zero a nove e deixa o usuário
3  'tentar acertar esse número
4  '
5  'A variável número irá conter o número sorteado
6  Dim numero As Integer
7  'A variável tentativas será usada para contar o número de
8  'tentativas de acerto
9  Dim tentativas As Integer
10 'A variável chute será usada para armazenar o número porposto
11 'pelo usuário
12 Dim chute As Integer
13 'Definimos uma variável texto para conter a mensagem exibida
14 'ao final do jogo.
```

```
15  Dim mensagem As String
16  'Precisamos da instrução Randomize para iniciar o gerador
17  'números aleatórios
18  Randomize
19  'Agora pedimos para o VBA gerar um númeo aleatório (Rnd()),
20  'multiplicamos esse número por 10, ficamos apenas com
21  'sua parte inteira. Esse valor é armazenado na variável
22  'número e é o número que o usuário precisa advinhar
23  numero = Int(10 * Rnd())
24  'Segue uma mensagem para iniciar o jogo
25  MsgBox ("Eu escolhi um número inteiro entre zero e nove. " & _
26          "Vamos ver em quantas tentativas você adivinha que " & _
27          "número é")
28  'Aqui introduzimos um rótulo, visto que queremos que o VBA
29  'repita as instruções abaixo até que o usuário
30  'advinhe corretamente o número sorteado:
31  desvio:
32  'Acrescentamos 1 ao contador de tentativas:
33  tentativas = tentativas + 1
34  'Usamos um InputBox para obter o chuto do usuário e o
35  'armazenamos na variável chute.
36  chute = InputBox("Tentativa de número " & tentativas)
37  'Se o usuário acertou, siga em frente, senão volte para desvio
38  If chute <> numero Then GoTo desvio
39  'A instrução If abaixo escolhe a mensagem de final de jogo
40  'em função do número de tentativas necessárias para o usuário
41  'advinhar o número sorteado:
42  If tentativas = 1 Then
43      mensagem = "Você deve ter poderes mediúnicos!"
44  ElseIf tentativas <= 3 Then
45      mensagem = "Você acertou em " & tentativas & _
46          " tentativas. Você tem bastante sorte!"
47  ElseIf tentativas <= 6 Then
48      mensagem = "Você acertou em " & tentativas & _
49          " tentativas. Você não é exatamente uma pessoa, " & _
50          " sortuda mas tampouco é uma pessoa azarada."
51  ElseIf tentativas <= 8 Then
52      mensagem = "Hoje não é seu dia de sorte. Você precisou " & _
53          "de " & tentativas & " tentativas para acertar."
54  Else
55      mensagem = "Você está com muito azar. " & _
56      "Já pensou em se benzer?"
57  End If
58  'Finalmente, a mensagem é exibida:
59  MsgBox (mensagem)
60  End Sub
```

Observe especialmente o rótulo "desvio:" na linha 31. Após esse rótulo seguem instruções para acrescentar um ao contador do número de tentativas (linha 33) e para a obtenção de uma nova tentativa numéria (linha 36) e, na linha 38, uma instrução If que instrui o VBA a voltar à linha 31 (com o rótulo "desvio:") caso o usuário não tenha adivinhado o número.

Note que a instrução If com início na linha 42 e fim na linha 57 poderia ser substituída por uma instrução Select Case do tipo

```
Select Case tentativas
Case 1 'Caso tentativas = 1 então
    mensagem = "Você deve ter poderes mediúnicos!"
Case 2, 3 'Caso tentativas = 2 ou 3 então
    mensagem = "Você acertou em " & tentativas & _
        " tentativas. Você tem bastante sorte!"
Case 4 To 6
    mensagem = "Você acertou em " & tentativas & _
        " tentativas. Você não é exatamente uma pessoa, " & _
        " sortuda mas tampouco é uma pessoa azarada."
Case 7, 8
    mensagem = "Hoje não é seu dia de sorte. Você precisou " & _
        "de " & tentativas & " tentativas para acertar."
Case Else
    mensagem = "Você está com muito azar. " & _
    "Já pensou em se benzer?"
End If
```

5.2 Instrução for

Desvios como o ilustrado na seção anterior são uma das estruturas mais frequentes em programação. Entre esses desvios, há algumas formas tão frequentes que acabaram tendo tratamento próprio nas diversas linguagens de programação. Um caso em que isso ocorre se dá quando desejamos que o computador repita um determinado conjunto de instruções um certo número de vezes. Para ilustrar esse caso, tomemos o seguinte exemplo:

Exemplo 1 *Considere a sequência numérica denotada por* $f_0, f_1, \ldots, f_i, \ldots$ *na qual* $f_0 = 0$, $f_1 = 1$ *e, para todo* $i > 2$, $f_i = f_{i-1} + f_{i-2}$. *Os primeiros termos dessa sequência aparecem abaixo:*

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377 \ldots$$

Essa sequência é conhecida como sequência de Fibonnaci, *em homenagem a Leonardo de Pisa, também conhecido como Fibonacci, que, em um livro intitulado* Liber Abaci *publicado em 1202, apresentou essa sequência para o público ocidental.* [6] *Escreva uma função no VBA que retorne o* n*-ésimo número de Fibonnaci em função de* n.

[6] A sequência já era conhecida de longa data pelos hindus.

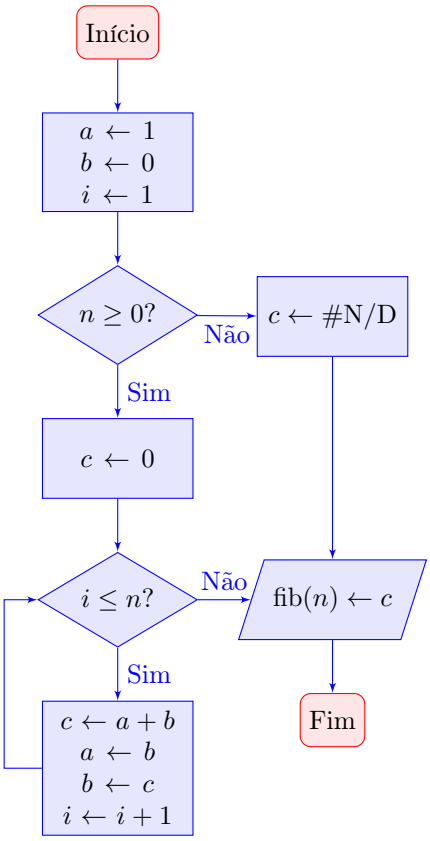


Figura 6: Diagrama de fluxo de um algoritmo para encontrar o n-ésimo número de Fibonnaci.

Chamemos de fib a função procurada e de n o seu argumento. O algoritmo empregado para resolver esse problema está ilustrado na Figura 6. Seja n o índice do número de Fibonnaci que se pretende obter. Criamos quatro variáveis auxiliares:

i um contador.

a variável para armazenar o valor do penúltimo número de Fibonnaci.

b variável para armazenar o valor do último número de Fibonnaci.

c variável para armazenar o número de Fibonnaci procurado.

Nesse algoritmo, os retângulos arredondados são usados para definir o início e o fim do processamento, os retângulos normais são usados para representar uma série de processos a serem executados, os losangos são usados para representar

um momento em que uma decisão deve ser tomada em resposta a uma pergunta e os paralelepípedos representam uma entrada ou uma saída de dados. Ademais, para maior clareza, optamos por usar o símbolo $\leftarrow$ para representar a atribuição de valores à uma variável. Assim $a \leftarrow b$ significa "atribua o valor b à variável a. O algoritmo começa atribuindo o valor 1 às variáveis a e i e o valor 0 à variável b. Em seguida, caso $n < 0$, situação em que o argumento da função é inadequado, ele atribui um valor de erro à variável c. Caso contrário, atribui valor 0 a essa variável. Ela agora armazena o número de Fibonnaci de ordem 0. Em seguida, é feita a pergunta "$i \leq n$?". Se a resposta for negativa, o número de Fibonnaci procurado foi encontrado e seu valor está na variável c ou ele não pode ser encontrado no caso em que $n < 0$. Nos dois casos, a execução estará completa quando a função retornar o valor armazenado na variável c. Se, entretanto, $i \leq n$, será necessário calcular o próximo número de Fibonnaci. Para tal, somam-se as variáveis a e b e armazena-se o resultado na variável c, armazena-se o valor da variável b na variável a e, na sequência, o valor da variável c na variável b. Agora, a variável c contém o valor do próximo número de Fibonnaci. Para indicar que o índice do número de Fibonnaci armazenado em c avançou, somamos 1 à variável i. Com essas operações concluídas, voltamos à pergunta "$i \leq n$" e, novamente, caso a resposta seja negativa, a função deverá retornar o valor da variável c e caso seja positiva, inicia-se um novo ciclo com atribuição do valor $a + b$ à variável c, etc. A parte de um código de programa que instrui o computador a realizar diversas vezes as mesmas tarefas é chamada *loop*.

O código abaixo implementa esse algoritmo no VBA usando a instrução GoTo para fazer a volva ao teste "$i \leq n$?". Note o uso da função CVErr. Essa função é usada pelo VBA para retornar uma mensagem de erro. Como queremos retornar essa mensagem para argumentos negativos indicando que a função "fib" não é definida para esses argumentos, queremos que, nesse caso, o Excell exiba #ND. Para tal, usamos o argumento xlErrNA. Nota também que não especificamos que as variáveis "a", "b" e "c", sejam do tipo Long. Isso foi feito porque as variáveis do tipo Double aceitam valores maiores — os número de Fibonnaci crescem rapidamente. Além disso a variável "c" foi definida como do tipo Variant, pois, caso o argumento da função seja negativo ela deve retornar uma mensagem de erro e não um número. O mesmo ocorre com a função "fib". O nosso interesse, todavia está na forma empregada para instruir o VBA checar recorrentemente se $i \leq n$. Isso é feito aplicando-se o rótulo `desvio:` uma linha antes desse teste e chamando-se a instrução `GoTo desvio` logo após a operação `i = i + 1`.

```
Function fib(n As Long)
'
'Retorna o n-ésimo número da sequência de Fibonnaci
'para qualquer valor de n não negativo.
'Por convenção, o número zero dessa série é zero.
'
'Variáveis empregadas:
```

```
Dim a As Double, b As Double, c As Variant, i As Long
'Valores iniciais:
a = 1
b = 0
i = 1
'Testa se o argumento da função é um número
'não negativo:
If n >= 0 Then
'caso o argumento n seja não negativo,
'armazena o valor do número de Fibonnaci de
'ordem zero na variável c
'
c = 0
desvio:
'se i ainda não for igual a n é preciso
'calcular o próximo número de Fibonnaci
'caso contrário, a variável c contém o
'número procurado:
  If i <= n Then
    c = a + b
    a = b
    b = c
    i = i + 1
'A variável c contém o número de Fibonnaci de ordem i
'precisamos agora verificar novamente se i=n ou se é
'necessário calcular o próximo número de Fibonnaci. Para
'tal, voltamos à linha rotulada desvio.
   GoTo desvio
  End If
Else
'Se n<0, não existe fib(n), a função retorna a
'mensagem #N/D. Usamos a função CVErr que retorna um
'erro. Para o erro retornado ser #N/D, o argumento usado
'deve ser xlErrNA:
c = CVErr(xlErrNA)
End If
'Agora só precisamos dizer que o valor da função fib
'é igual ao da variável c:
fib = c
End Function
```

O *loop* realizado por esse código é de um tipo muito frequente: uma série de instruções, entre elas um acréscimo (usualmente de uma unidade) a um contador (no caso, a variável i), é repetida até que o contador atinja um determinado valor (n no presente código). Para deixar mais clara a presença desse tipo de *loop*, a maioria das linguagens de programação permitem que ele seja

representado por uma instrução específica para gerá-lo. No caso do VBA, tal instrução é a instrução `for`. Ela tem o seguinte formato:

```
For contador = valor inicial To valor final Step incremento
  instruções
Next contador
```

Na qual *contador* é uma variável, *valor inicial* e *valor final* e *incremento* são valores inteiros tais que *valor inicial* $\leq$ *valor final* e *instruções* é um conjunto de instruções. Ao ler esse código, o VBA atribui o valor *valor inicial* à variável *variável*, executa as *instruções*, acrescenta *incremento* ao valor de *variável* e repete os mesmos procedimentos (exceto atribuir o valor inicial à variável *variável*) até que o valor em *variável* não seja mais inferior a *valor final*.

Eis como fica o código da função "fib" com o uso da instrução `for` no lugar do *loop* construído com a instrução `GoTo`:

```
Function fib(n As Long)
'
'Retorna o n-ésimo número da sequência de Fibonnaci
'para qualquer valor de n não negativo.
'Por convenção, o número zero dessa série é zero.
'
'Variáveis empregadas:
Dim a As Double, b As Double, c As Variant, i As Long
'Valores iniciais:
a = 1
b = 0
'Testa se o argumento da função é um número
'não negativo:
If n >= 0 Then
'caso o argumento n seja não negativo,
'armazena o valor do número de Fibonnaci de
'ordem zero na variável c
'
c = 0
'Se n = 0, c já contém o número de Fibonnaci
'procurado. Caso contrário, usamos a instrução
'for para calcular os sucessivos números de Fibonnaci
'até chegar ao número desejado:
  For i = 1 To n
    c = a + b
    a = b
    b = c
  Next i
Else
```

```
'Se n<0, não existe fib(n), a função retorna a
'mensagem #N/D. Usamos a função CVErr que retorna um
'erro. Para o erro retornado ser #N/D, o argumento usado
'deve ser xlErrNA:
c = CVErr(xlErrNA)
End If
'Agora só precisamos dizer que o valor da função fib
'é igual ao da variável c:
fib = c
End Function
```

O uso da instrução For para *loops* como o aqui considerado no lugar do uso de rótulo de linha combinado com a instrução GoTo contribui para deixar o código mais legível e bem estruturado e, por essa razão, é recomendado.

5.3 Instruções Do Until e Do While

Um outro tipo de *loop* muito usado consiste em instruir o computador para repetir uma série de operações enquanto uma determinada condição não for atendida. Considere, por exemplo, a macro "adivinha" apresentada na seção 5.1. O trecho do código abaixo instrui o VBA a solicitar ao usuário que tente novamente acertar o número sorteado e conte a ocorrência dessa nova alternativa enquanto o número escolhido pelo usuário não coincidir com o número sorteado:

```
desvio:
'Acrescentamos 1 ao contador de tentativas:
tentativas = tentativas + 1
'Usamos um InputBox para obter o chute do usuário e o
'armazenamos na variável chute.
chute = InputBox("Tentativa de número " & tentativas)
'Se o usuário acertou, siga em frente, senão volte para desvio
If chute <> numero Then GoTo desvio
```

Novamente, como trata-se de uma estrutura de *loop* muito frequente, muitas linguagens oferecem uma notação especial para ela. No caso do VBA, a instrução que cumpre esse papel é a instrução Do. Esse instrução apresenta quatro variações. A primeira delas tem a forma

```
Do Until condição
  instruções
Loop
```

Na qual *condição* é a condição que deve ser observada para que as instruções seja repetidas e *instruções* é o conjunto de instruções a serem seguidas enquanto a *condição* é válida. Nesse caso, as *instruções* são realizadas repetidas vezes até que (*until*) a condição seja observada. A palavra-chave `Loop` é um indicador

do final do *loop*. Ao se deparar com essa palavra-chave, o VBA retorna à linha da instruções `Do Until` e verifica se a *condição* é válida, caso positivo, ele sais do *loop*, caso negativo, volta a executar as *instruções*.

Uma característica dessa primeira versão da instrução `Do` é que a condição para o *loop* é verifica em seu início. Caso, ao início do *loop condição* seja uma expressão avaliada como verdadeira, o VBA não executará sequer uma vez as instruções que se seguem. Para exemplificar isso, consideremos a seguinte alternativa para o código do *loop* da macro "adivinha":

```
Do Until chute = numero
'Acrescentamos 1 ao contador de tentativas:
tentativas = tentativas + 1
'Usamos um InputBox para obter o chute do usuário e o
'armazenamos na variável chute.
chute = InputBox("Tentativa de número " & tentativas)
'Se o usuário acertou, siga em frente, senão volte para desvio
Loop
```

Caso seja escrito dessa maneira, quando chegar a esse *loop*, o VBA testará primeiramente se a variável "chute" tem valor igual ao da variável "numero". Se isso for verdadeiro, ele abandona o *loop* se for falso e segue e aumenta de 1 o valor da variável tentativas, pede ao usuário que tente novamente adivinhar o número selecionado e, após isso, volta ao início do *loop*, testando se o novo valor inserido pelo usuário é igual ao número sorteado e decidindo novamente se prossegue no *loop* ou não. O uso dessa forma da instrução Do oferece um problema para nossa macro: caso, quando o VBA atingir o *loop* o valor da variável "chute" ainda não tenha sido definido, o VBA atribuirá a ela valor zero quando for solicitado a compará-la com a variável "número". Nesse caso, caso o número sorteado e atribuído à variável "número" tenha sido igual a zero, ele abandonará o *loop* e o usuário não terá tido chance de tentar adivinhar o número sorteado. Podemos evitar que isso aconteça de um modo pouco elegante, atribuindo à variável chute, antes do início do *loop*, um valor que sabemos que não será sorteado, por exemplo, -1. Nesse caso acrescentamos uma linha antes da instrução Do, para ficarmos com

```
chute = -1
Do Until chute = numero
'Acrescentamos 1 ao contador de tentativas:
tentativas = tentativas + 1
'Usamos um InputBox para obter o chute do usuário e o
'armazenamos na variável chute.
chute = InputBox("Tentativa de número " & tentativas)
'Se o usuário acertou, siga em frente, senão volte para desvio
Loop
```

Como a variável "número" só assume valores inteiros de 0 a 9, podemos ter certeza que, na primeira vez em que o VBA lê a instrução Do, essas duas

variáveis não serão iguais e que, portanto, o *loop* será executado pelo menos uma vez.

Uma solução bem mais elegante consiste em usar a segunda forma da instrução Do na qual a comparação é feita ao final do *loop*. Para tal, escrevemos `Until chute = numero`, isto é, a condição que queremos testar ao final do *loop*, não após a palavra `Do`, mas após a `Loop`, conforme se vê abaixo:

```
Do
'Acrescentamos 1 ao contador de tentativas:
tentativas = tentativas + 1
'Usamos um InputBox para obter o chute do usuário e o
'armazenamos na variável chute.
chute = InputBox("Tentativa de número " & tentativas)
'Se o usuário acertou, siga em frente, senão volte para desvio
Loop Until chute = numero
```

Quando fazemos isso, a comparação entre as variável "chute" e "número" é feita ao final do *loop*, o que garante que o usuário tente adivinhar, ao menos uma vez, o número sorteado.

As duas variações restantes da instrução Do diferem das duas apresentadas pelo uso da palavra chave *While* (enquanto) no lugar da palavra-chave *Until* (até que). Nesse caso, o *loop* é repetido enquanto a condição estabelecida é válida. Assim, podemos reescrever o *loop* da macro "adivinha", com o teste ao início do *loop* da forma abaixo:

```
chute = -1
While chute <> numero
'Acrescentamos 1 ao contador de tentativas:
tentativas = tentativas + 1
'Usamos um InputBox para obter o chute do usuário e o
'armazenamos na variável chute.
chute = InputBox("Tentativa de número " & tentativas)
'Se o usuário acertou, siga em frente, senão volte para desvio
Loop
```

Ou, como seria mais elegante nesse caso, colocando o teste ao final do *loop*, conforme se segue:

```
Do
'Acrescentamos 1 ao contador de tentativas:
tentativas = tentativas + 1
'Usamos um InputBox para obter o chute do usuário e o
'armazenamos na variável chute.
chute = InputBox("Tentativa de número " & tentativas)
'Se o usuário acertou, siga em frente, senão volte para desvio
Loop While chute <> numero
```

Exemplo 2 *Para ilustrar o uso da instrução Do While, apresentamos o código da função ÉPrimo que testa se um número inteiro é primo. Esse código é algo ineficiente e baseia-se no fato de que um número é primo se, e somente se, ele for um número positivo maior do que 1 que possui apenas dois divisores distintos: ele próprio e o número 1. Seja n o número que queremos testar se é ou não primo. A estrutura do código está representada na Figura 7. A variável a é usada como potencial divisor do número n. O código começa atribuindo a essa variável o valor 2 (o menos possível divisor de um número não primo). Na sequência, ele pergunta se n é divisível por a ou se $n \leq 1$. Nos dois casos, n não é primo. Caso n não seja divisível por a ele acrescenta a a uma unidade e testa essa divisibilidade novamente. Esse processo continua até que a atinja o menor valor que divide n. Caso esse valor seja diferente de n, n não é primo e a função "ÉPrimo" retorna o booleano False. Caso o menos divisor do número n, armazenado na variável a, seja igual a n, então, n é primo e a função "ÉPrimo" deve retornar o booleano False.*

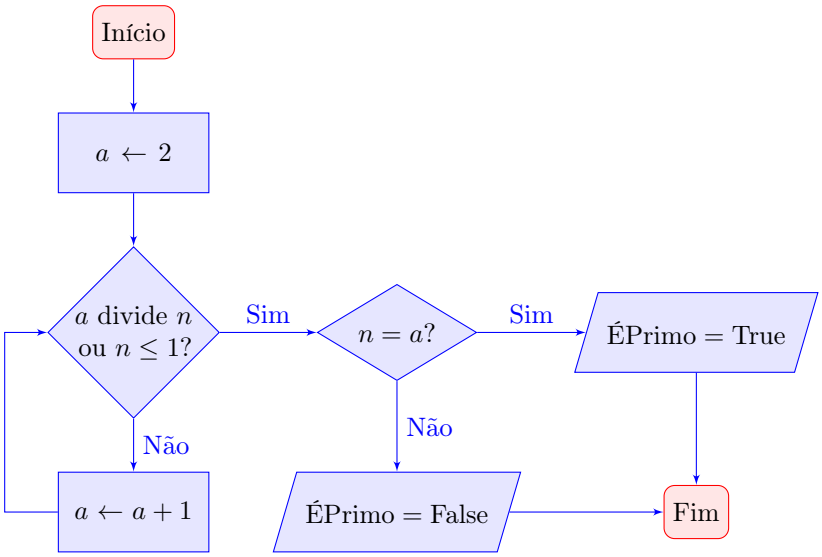


Figura 7: Algoritmo de uma função para definir se um número é primo

O código da função VBA é o que segue:

```
Function ÉPrimo(n As Long) As Boolean
Dim a As Long
Do Until a Mod n = 0 Or n <= 1
  a = a + 1
Loop
```

```
ÉPrimo = (n = a)
End Function
```

Note que, ao atingir a penúltima linha, o VBA, primeiramente avalia a expressão `n = a`. Esta pode retornar False, caso $n \neq a$, ou, caso $n = a$, True. O resultado será o valor retornado pela função ÉPrimo.

Esse código é bastante ineficiente, no sentido que ele demanda mais computação do que seria necessário. Por exemplo, para testar se um número é primo, ele verifica se ele é divisível por 2 e outros múltiplos de 2. Isso é desnecessário, pois se um número não é divisível por 2, tampouco será divisível por seus múltiplos. Na seção 8 apresentamos um código mais eficiente para a mesma função.

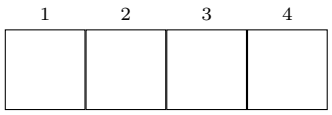
6 Matrizes

Matrizes

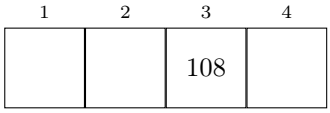
Matrizes, tradução não totalmente precisa para o português do termo inglês "arrays", são no VBA uma forma de armazenar diversos dados usando índices. Se uma variável é um recipiente que pode conter um dado, uma matriz é um conjunto desses recipientes. Criamos uma matriz de modo similar ao que criamos uma variável. Por exemplo, a instrução

```
Dim minha_matriz(1 To 4) As Long
```

diz para o VBA criar uma matriz uma matriz, chamada "minha_matriz", contendo 4 espaços para armazenar dados do tipo Long, enumerados de 1 a 4, que pode ser ilustrada conforme a figura abaixo, na qual cada quadrado representa um espaço reservado para conter um número do tipo Long e os números acima desses quadrados representam a indexação desses números:



Os quadrados estão vazios porquê nenhuma valor foi ainda armazenado nos elementos da matriz. Para chamar um elemento dessa matriz, por exemplo, o segundo elemento, digitamos `minha_matriz(2)`. Esse elemento pode ser tratado como uma variável comum. Assim, caso queiramos que o terceiro elemento de nossa matriz contenha o número 108, digitamos, no código VBA, `minha_matriz(3)=108`. Após o VBA ler esse código, nossa matriz pode ser representada conforme a figura abaixo, na qual o número 108 foi acrescentado ao terceiro quadrado:



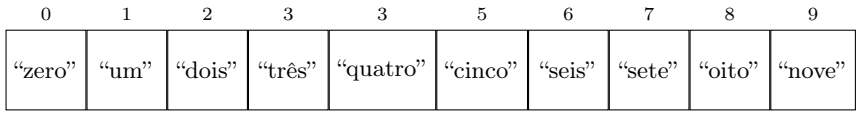
Para ilustrar um possível uso de matrizes no VBA vamos criar uma função, denominada "ext", que escreve na forma extensiva qualquer número de 0 a 9. Isto é, queremos que, por exemplo, ao inserir em uma célula do Excel a fórmula `=ext(7)`, essa célula exiba `sete`. Sabemos como escrever essa função usando a instrução Case ou a instrução If. Deixamos a construção dessa função como exercício.

Exercício 4 *Construa uma função que retorne a representação textual extensiva de qualquer número inteiro de 0 a 9.*

Vamos resolver o mesmo exercício empregando matrizes. Para tal, basta escrevermos um código como o que se segue:

```
1  Function extenso(numero As Integer) As String
2  Dim ext(0 To 9) As String
3  ext(0) = "zero"
4  ext(1) = "um"
5  ext(2) = "dois"
6  ext(3) = "três"
7  ext(4) = "quatro"
8  ext(5) = "cinco"
9  ext(6) = "seis"
10 ext(7) = "sete"
11 ext(8) = "oito"
12 ext(9) = "nove"
13 extenso = ext(numero)
14 End Function
```

A primeira linha desse código declara a função "extenso" como uma função com um único argumento do tipo Inteiro que retorna uma sequência de caracteres. A linha 2 declara a matriz "ext" que contém 10 posições indexadas de 0 a 9. As linhas 3 a doze atribuem a cada uma das posições de nossa matriz a representação do número inteiro correspondente, de tal sorte que, após a linha 12, a matriz "ext" possa ser representada pela figura



A linha 13 instrui o VBA a fazer com que a função "extenso" retorne o texto contido no elemento indexado pelo argumento da função (numero) da

matriz “ext”. Assim, caso você insira em uma célula do Excel a fórmula `=extenso(3)`, a função “extenso” retornará o texto contido na célula indexada pelo número 3 da matriz “ext”, ou seja, “três”. Caso você chame essa função com um argumento fora do intervalo de 0 a 9, ela retornará uma mensagem de erro “#VALOR”.

A função Array

Ao inserir o código da função “extenso” acima precisamos de uma linha para cada atribuição de valor aos elementos da matriz “ext”. Isso pode ser bastante tedioso e contribuir para tornar nosso código menos legível (para humanos). Uma forma de resolver esse problema consiste em usar a função “Array” do VBA. Essa função retorna uma matriz com os elementos iguais a seus argumentos separados por vírgula. Essa matriz pode então ser atribuída a uma variável do tipo Var que passará a ser, ela mesma, uma matriz. Os índices dos elementos da matriz gerada pela função “Array” começam em zero. Usando a função “Array”, o código da função “extenso” é reduzido para

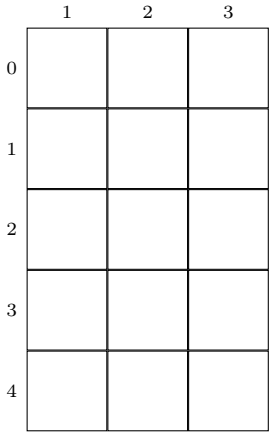
```
Function extenso(numero As Integer) As String
Dim ext As Var
ext =("zero", "um", "dois", "três", "quatro", "cinco", "seis", _
      "sete", "oito", "nove")
extenso = ext(numero)
End Function
```

6.1 Matrizes multidimensionais

Dizemos que uma matriz possui mais de uma dimensão quando seus elementos são indexados por mais de um índice. Por exemplo, podemos pensar uma matriz cujos elementos estão organizados na forma de uma tabela, de tal forma que o primeiro índice se reporta à linha do elemento e, o segundo índice, a sua coluna. Uma tabela é uma matriz com duas dimensões. Como ilustração de como criar uma matriz bi-dimensional suponha que desejemos criar uma matriz, chamada “minha_matriz”, com 5 linhas enumeradas de 0 a 4 e três colunas também enumeradas de 1 a 3, contendo números inteiros. O seguinte código faria isso:

```
Dim minha_matriz(0 To 4, 1 To 3) As Integer
```

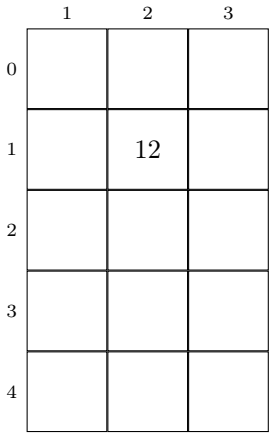
Esse código faz o VBA criar uma matriz que podemos representar da seguinte maneira:



Se quisermos atribuir o número 12 ao elemento localizado na segunda linha (indexada pelo número 1) e na segunda coluna dessa matriz, devemos entrar o código

```
minha_tabela(1,2) = 12
```

Após ler esse código a matriz "minha_matriz" fica similar a



Exemplo 3 *Um quadrado mágico é uma matriz quadrada de ordem n contendo todos os inteiros de* 1 *a* n^2 *organizados de tal forma que a soma dos números em cada uma das linhas, a soma dos números em cada uma das colunas e a soma dos números em cada uma das diagonais seja sempre a mesma. Para exemplificar, mostramos um quadrado mágico de ordem 3:*

8	*1*	*6*
3	*5*	*7*
4	*9*	*2*

<++>

7 Objetos

De um modo bastante vago, podemos dizer que, na linguagem de programação, um *objeto* é um conjunto de informações estruturadas que representam uma única entidade computacional. O VBA já traz diversos objetos previamente definidos e você pode definir novos tipos de objetos, mas não veremos como fazer isso aqui. Objetos são instâncias de uma *classe*. Uma classe é a definição do conjunto de propriedades e de métodos que os objetos membros devem possuir. As propriedades de um objeto são as informações que ele carrega. Os métodos são ações que ele é capaz de perfazer. Método é algum tipo de ação que um objeto pode perfazer.

Um exemplo de objeto é o objeto "Application". Este objeto refere-se ao aplicativo do Excel propriamente dito. Esse objeto possui muitas propriedade, mas, para exemplificar, citemos três: a propriedade "UseSystemSeparators" pode assumer os valores True ou Falsa e indica se o Excel deve os separadores de casas decimais e de milhares definidos no sistema (se for igual a True) ou não (se for igual a False). Outra propriedade é "DecimalSeparator" e ela define o caractere a ser usado como separador decimal, usualmente uma vírgula — convenção comum na Europa continental, na Rússia e na América do Sul – ou um ponto – convenção nos Estados Unidos, na Inglaterra e outros países do Reino Unido, na Índia, na China e no Japão. Finalmente a propriedade "ThousendsSeparator" representa o caractere usado para separar milhares, milhões, bilhões, etc. Novamente, nos países da Europa Continental, e da América do Sul, a convenção é usar um ponto para fazer essa separação e na Inglaterra e outros países de língula inglesa assim como na China e no Japão, a convenção é usar uma vírgula. Para chamar uma propriedade de um objeto, basta escrever uma expressão que identifique esse objeto seguida de um ponto e o nome da propriedade. Por exemplo, a expressão

```
Application.ThousandsSeparator
```

retorna o separador de milhares usado pelo Excel no momento. Experimente escrever a seguinte macro

```
  Sub separadore As String
Dim separador As String
```

```
separador = Application.DecimalSeparator
msgbox("O separador decimal é """ & separador & """")
separador = Application.ThousandsSeparator
msgbox("O separador de milhares é """ & separador & """")
End Sub
```

Essa macro usa a propriedade DecimalSeparator para ler o separador decimal em uso, arquiva esse separador na variável "separador", uma mensagem informando qual o tipo de separador decimal empregado e, na sequência faz o mesmo com o separador de milhares. Ao escrever essa macro, você deve ter percebido que, assim que escrevemos `Application.`, o editor do VBA abre uma lista suspensa tal como a mostrada na Figura 8 na qual ele lista todas as propriedades e todos os métodos que se aplicam ao objeto.

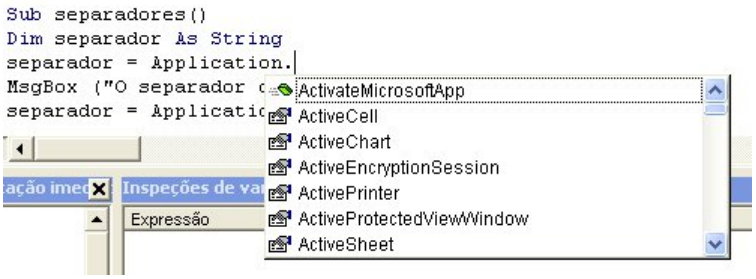


Figura 8: O editor do VBA mostra uma lista suspensa com as propriedades e métodos de um objeto quando colocamos um ponto à direita de uma expressão que o retorna.

Muitas propriedades podem ser, não apenas lidas, mas também alteradas por uma macro. Por exemplo suponha que você esteja usando sua planilha do Excel em um sistema configurado de acordo com os padrão para a língua português do Brasil, mas queira que o Excel use como separador de decimal um ponto e, como separador de milhares, um espaço. Você pode fazer isso alterando os valores das propriedades DecimalSeparator e ThousandsSeparator do objeto Application. Mas, para que suas alterações tenha efeito, você deve também dizer ao Excel que ele não deve usar os padrões definidos em seu sistema. Para tal você precisa também alterar o valor da propriedade UseSystemSeparators de True para False. Isso pode ser feito fazendo rodar a seguinte macro:

```
Sub MudaSep ()
Application.UseSystemSeparators = False
Application.ThousandsSeparator = " "
Application.DecimalSeparators = "."
End Sub
```

Conforme dissemos, os objetos possuem propriedades e métodos. Um método é uma ação que pode ser realizada no objeto. Um exemplo de método do objeto Application, é o método Quit. Ele simplesmente diz ao Excel para realizar a ação de encerrar. Assim, por exemplo, ao executar a macro "adeus" abaixo fazemos com que aplicativo do Excel seja encerrado:

```
Sub adeus()
Application.Quit
End Sub
```

8 Optimização