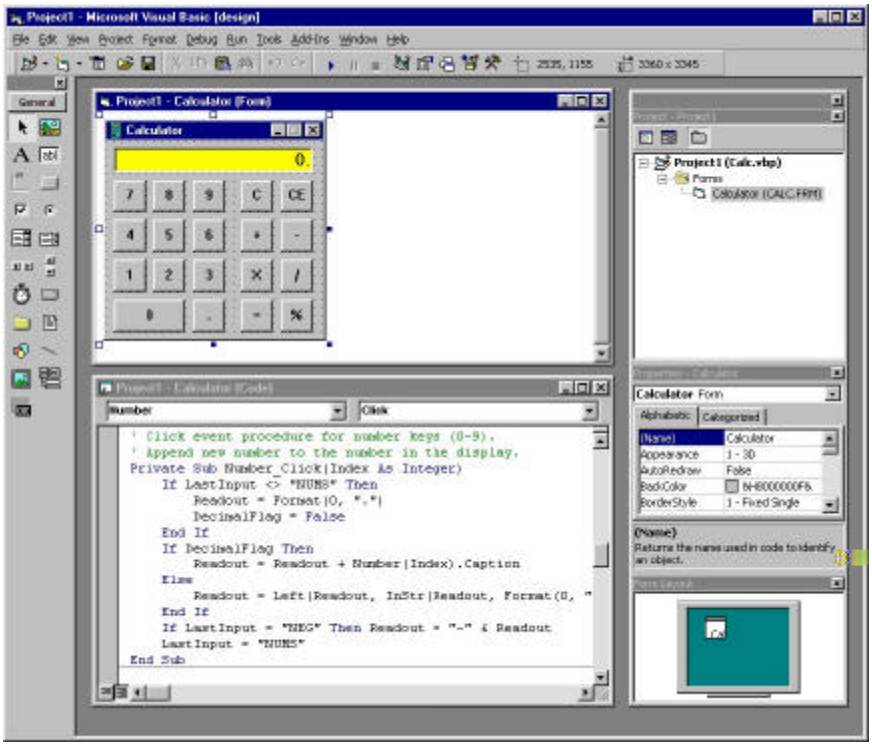
VISUAL BASIC

Sebenta Teórica (versão 1)



Instituto Superior de Engenharia do Porto

Departamento de Engenharia Informática

Janeiro de 2001

ÍNDICE

1 - Conceitos Gerais

Programação orientada ao objecto

A programação tradicional baseia-se numa distinção clara entre o programa propriamente dito e os dados que esse programa processa. Nesta óptica, um programa é entendido como uma sequência de instruções que manipulam os dados que lhe são fornecidos.

Do ponto de vista da programação orientada para objectos, a forma de encarar um programa é substancialmente diferente.

Um programa passa a ser visto como uma simulação de um ou vários aspectos do mundo real, pretendendo-se com ele modelizar um conjunto de objectos que interagem com o propósito de alcançar um dado objectivo.

Assim sendo, o próprio programa é estruturado como um conjunto de objectos que interagem entre si e com o mundo real. Cada objecto do programa é uma entidade com características e capacidades próprias, contendo dentro de si quer os dados a processar quer os módulos de programa que processam esses dados.

Classes e Objectos

Os objectos são normalmente agrupados em **classes**. Na vida real, um *Fiat Punto* ou um *BMW* podem ser classificados como objectos que pertencem à classe "automóvel". São diferentes, mas não deixam de partilhar as características que os definem como automóveis. Também na programação por objectos são utilizados objectos que pertencem a classes. Cada objecto pertencendo a uma mesma classe possui características comuns a todos os objectos pertencentes a essa classe.

Pode-se, assim, definir uma **classe** como a matriz que descreve quais as características e capacidades de um dado conjunto de objectos. Em programação, uma **classe** será um conjunto de especificações que regulam a criação de objectos de um dado tipo.

A linguagem Visual Basic

Esta **l**nguagem incorpora os conceitos de objecto e classe. Nela existe uma vasta gama de classes de objectos predefinidas, com base nas quais o programador pode criar os objectos de que necessita no seu programa. Por outro lado, a linguagem permite também que o programador crie classes com características específicas.

A Caixa de Ferramentas ("**ToolBox**")

O exemplo mais comum de objecto utilizado por aplicações desenvolvidas em linguagens orientadas para objectos como o Visual Basic é o "botão de comando" que, nesta linguagem, deriva da classe **CommandButton**. Sempre que o programador necessita de um botão de comando, limita-se a requerer a criação de um novo objecto dessa classe.

A criação de objectos standard ou predefinidos é um processo tão elementar como a escolha numa janela denominada "caixa de ferramentas" (toolbox) da classe a que pertence o novo objecto a criar, seguida do desenho com o rato dos contornos desse objecto na janela que o vai conter.

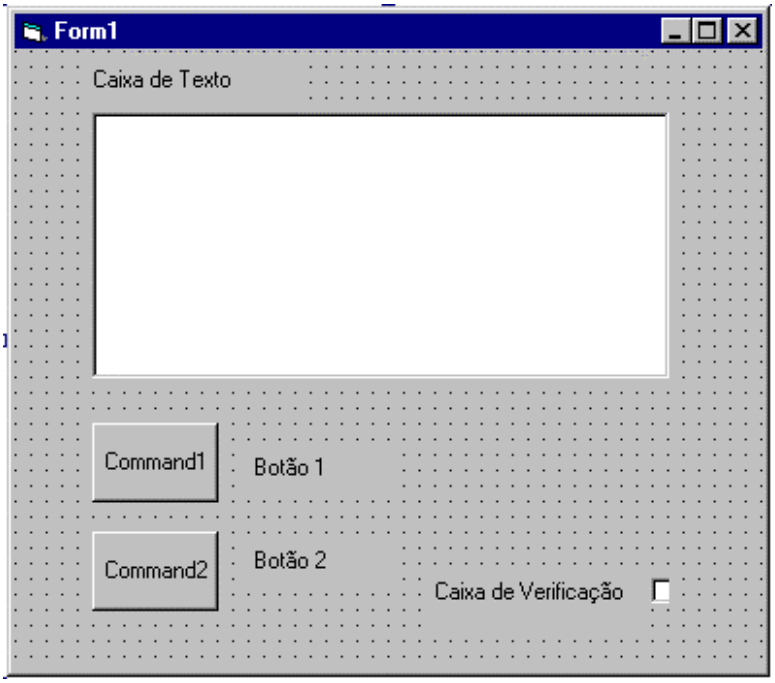
Em Visual Basic, a criação de um programa é um projecto em duas fases:

- a definição da interface com o utilizador ou seja da forma como o programa se vai apresentar visualmente ao utilizador
- a especificação do código que permite ao programa executar as acções requeridas

Nas linguagens tradicionais, a criação da interface era responsabilidade do próprio programa, ou seja, o código que gerava a interface estava contido no programa. Em linguagens como o Visual Basic, a interface pode ser criada recorrendo às classes de objectos preexistentes, como botões, barras de menus ou caixas de texto, sem que para isso o programador tenha tido necessidade de escrever uma linha de código.

Forms e Controlos

Uma aplicação informática, em ambiente Windows, possui sempre uma ou mais janelas, mediante as quais se estabelece a interacção com o utilizador.



Uma **form** com vários **controlos**

Durante a fase de desenvolvimento dessa aplicação em Visual Basic, o nome dado às janelas é "**Forms**".

Em cada janela, ou "Form", é possível instalar objectos como botões ou caixas de texto a que chamamos "**Controlos**" e que permitirão interagir com a interface. É ainda possível alterar as características (dimensionais ou outras) das "**Forms**" bem como configurar os **Controlos** de forma a adaptar a interface ao objectivo pretendido com o programa.

As **forms** pertencem à classe "**Form**". Por seu lado os controlos que tenham sido inseridos nas **forms** podem ser oriundos de diversas classes. Assim, um botão de comando ("commandButton") pertence à classe CommandButton, enquanto uma caixa de texto (textBox) pertence à classe TextBox.

Propriedades

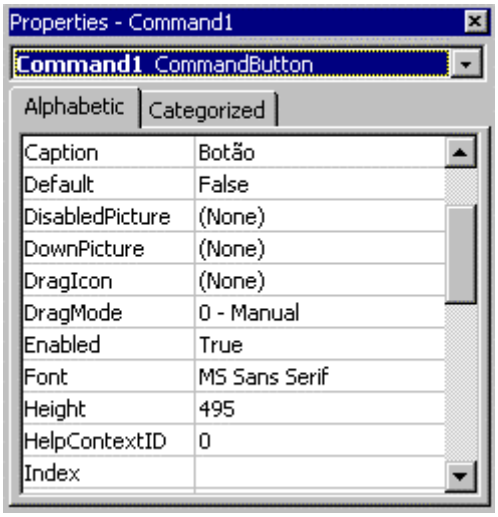
Como na vida real, cada objecto possui características próprias ou **propriedades**, que podem ser quantificadas (por exemplo, as dimensões de um botão de comando ou a cor dum rótulo). Cada **form** ou objecto tem associada uma lista de propriedades às quais é possível atribuir valores que determinam a sua aparência, localização e outros detalhes. Pode-se então dizer que as propriedades de um objecto definem a forma como ele se apresenta.

Diversos objectos podem partilhar a mesma propriedade. Essa propriedade, no entanto, pode afectar esses objectos de forma diferente.

Algumas das propriedades mais importantes e que são comuns à maior parte dos objectos são as seguintes:

Propriedade	**Efeito**
Caption	Define o texto a afixar na barra de título das forms, da legenda ("caption") dos botões de comando, ou nos rótulos ("label")
Name	Define o nome pelo qual o objecto é identificado
Left	Define o afastamento entre uma "form" e o limite esquerdo do ecrã ou entre um controlo e o limite esquerdo da "form"
Top	Define o afastamento entre uma "form" e o topo do ecrã ou entre um controlo e o topo da "form"
Height	Define a altura do objecto
Width	Define a largura do objecto
Font	Especifica qual o tipo de letra a usar nos controlos
Visible	Permite controlar o aparecimento de um dado objecto

Os valores que tomam as propriedades de um dado objecto podem ser consultados ou modificados mediante a *janela de propriedades*. Nessa janela aparece a lista de propriedades do objecto que estiver nesse momento seleccionado.



Janela de Propriedades

Métodos

Um **método** pode ser visto como uma função associada a um dado objecto. Essa função possibilita efectuar uma determinada acção sobre esse objecto. Tais funções estão disponíveis àpartida, bastando, pois, saber como as utilizar para obter o efeito requerido.

Dois métodos comuns àmaior parte dos objectos e de utilização frequente são os seguintes:

Move

Este método tem como resultado a deslocação do objecto ou form, bastando para tal, que lhe seja fornecida a informação de qual a localização final.

A sua sintaxe é: ***Objecto.Move*** *Left****,*** *Top****,*** *Width****,*** *Height*

Objecto representa o objecto ao qual o método Move vai ser aplicado, enquanto *Left, Top, Width* e *Height* são os parâmetros a ser fornecidos ao método, ou seja, as informações acerca de como efectuar a deslocação. De entre todos os parâmetros, somente o parâmetro *Left* é obrigatório.

SetFocus

Permite definir qual o objecto que está actualmente assinalado. Diz-se, em linguagem típica do ambiente Windows, que esse objecto possui o *focus*, querendo com isso dizer que será esse o objecto a ser accionado pela próxima acção do utilizador. Neste ambiente, ao mesmo tempo, não pode haver mais do que uma janela a possuir o *focus* e, dentro dela, só um objecto nas mesmas condições.

A sintaxe deste método é: ***Objecto.SetFocus***

Para que o método seja executado, basta associá-lo com ***Objecto***. Não há necessidade de qualquer informação (parâmetro) adicional.

Um objecto pode ser assinalado (receber o *focus*) através da inclusão do comando acima referido num programa, ou mediante um "clique" do rato sobre o próprio objecto.

Eventos

Por **evento** entende-se um acontecimento a que um objecto é capaz de reagir. A lista de eventos que um objecto pode reconhecer é específica desse objecto. Um evento pode resultar de uma acção do utilizador ou decorrer do funcionamento do próprio programa.

Dois dos eventos mais vulgares são o **Click** (pressão simples no objecto com o rato) e o **DoubleClick** (dupla pressão sobre o objecto).

Quando ocorre um evento que o objecto é capaz de reconhecer, uma resposta pode ser produzida. Essa resposta está exclusivamente dependente de código especificamente escrito para o efeito pelo programa78 Essa96 rck

2 - O primeiro exemplo

Nesta parte começa-se por apresentar o ambiente de desenvolvimento integrado do Visual Basic (*menu bar*, *toolbars*, *toolbox*, *project explorer window*, *properties window*, *form designer* e *code editor window*).

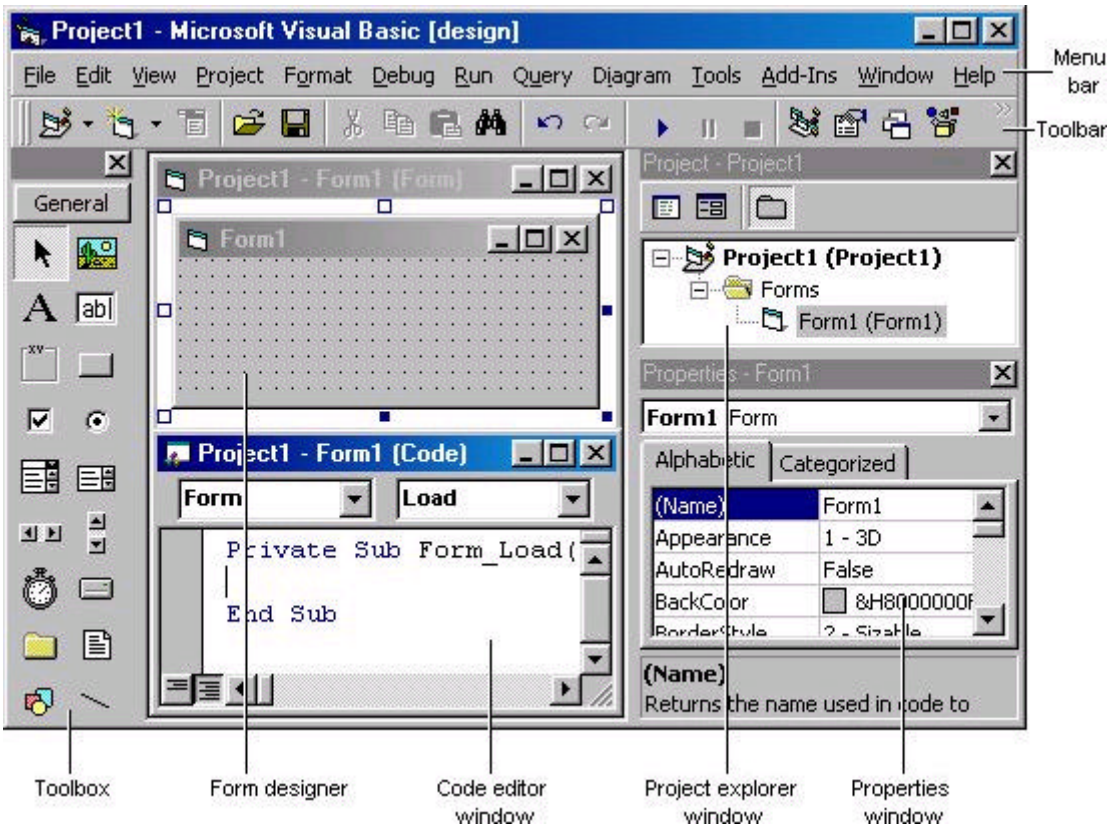
Depois explica-se como se cria um projecto e detalham-se os passos necessários àconstrução de uma aplicação em VB. São eles:

- Desenhar a interface
- Ajustar as propriedades
- Escrever o código

Por fim mostra-se como se corre uma aplicação e se guarda o trabalho.

Tudo isto é complementado com um pequeno exemplo cuja implementação vai sendo explicada e que consiste em desenvolver um pequeno programa que efectue a conversão entre graus Celsius e graus Farenheit.

Ambiente de desenvolvimento



Ambiente de Desenvolvimento Integrado do Visual Basic

O Ambiente de Desenvolvimento do Visual Basic é composto pelos seguintes elementos:

Menu Bar

Apresenta os comandos que se usam para trabalhar com o Visual Basic. Para além dos habituais menus *File*, *Edit*, *View*, *Window* e *Help*, existem outros menus que permitem o acesso a funções específicas da programação tais como *Project*, *Format* ou *Debug*.

Toolbars

Possibilitam um acesso rápido aos comandos mais frequentemente usados. Basta clicar uma vez num botão da toolbar para executar a acção associada a esse botão.

Toolbox

Fornece um conjunto de ferramentas usadas para inserir controlos nos forms.

Project explorer window

Lista os forms e módulos existentes no projecto actual. Um projecto é uma colecção de ficheiros usados para construir uma aplicação.

Properties window

Permite modificar a aparência ou comportamento do form ou controlo seleccionado. As propriedades são características dos objectos, tais como tamanho, texto ou cor.

Form designer

É a janela de trabalho que serve para desenhar a interface da aplicação com o utilizador. Inserem-se controlos, gráficos e imagens no form para se obter o resultado e aparência desejados. Cada form da aplicação tem o seu próprio "*form designer*".

Code editor window

Editor onde se escrevem as instruções que irão responder às acções do utilizador: botão premido, movimentos do rato, entrada de dados, etc. Existe um editor separado para cada form da aplicação.

Criar o projecto

No desenvolvimento de qualquer aplicação em VB começa-se por criar um projecto que reunirá todas as partes necessárias ao funcionamento da aplicação (forms, módulos, ...).

Para criar um projecto deve-se escolher a opção ***New Project*** do menu *File* e de seguida seleccionar ***Standard EXE*** na janela *New Project*.

Nota: Quando se corre o Visual Basic pela primeira vez, a janela *New Project* surge por defeito.

EXEMPLO

Crie então um projecto da forma indicada.

Desenhar a interface

O primeiro passo no desenvolvimento de uma aplicação consiste em criar o form que será a base da interface dessa aplicação. De seguida desenham-se os controlos necessários no form criado.

Os controlos são caixas, botões ou texto desenhados no form para receber ou mostrar informação.

Como desenhar um controlo no form?

1. Seleccionar o controlo pretendido na *toolbox*
2. Mover o apontador do rato para o form - o apontador do rato passa a ser uma cruz
3. Colocar a cruz onde se deseja o canto superior esquerdo do controlo
4. Premir o botão esquerdo do rato e sem largar efectuar um movimento de arrastamento até o controlo ter o tamanho desejado
5. Libertar o botão do rato - o controlo aparece no form

Para mover um controlo basta clicar com o botão esquerdo do rato sobre o controlo a mover e, sem largar, efectuar um movimento de arrastamento até chegar àposição pretendida.

Para alterar as dimensões selecciona-se o controlo e arrastam-se os *handles* (8 pequenos quadrados) em seu redor para dar o tamanho desejado.

EXEMPLO (cont.)

Neste pequeno exemplo vamos usar 3 tipos de controlos:

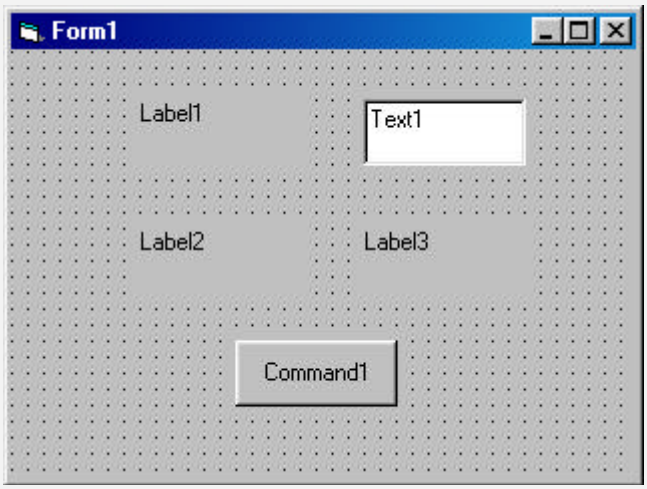
Controlo	Designação	Descrição
	Label (etiqueta)	Contém texto e é normalmente usado para descrever algo
	Text box (caixa de texto)	Permite a visualização e edição de dados
	Command button (botão de comando)	Executa uma acção (previamente associada) quando premido

Insira os seguintes controlos no form:

- 3 labels (etiquetas)
- 1 text box (caixa de texto)
- 1 command button (botão de comando)

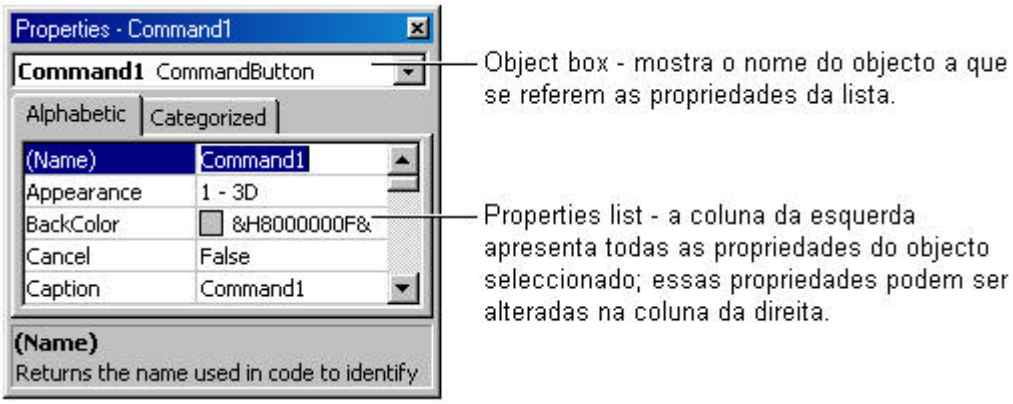
Após ter inserido estes controlos deverá obter algo semelhante ao form a seguir apresentado.

O próximo passo será o de ajustar as propriedades dos controlos e do próprio form.



Ajustar as propriedades

Todos os objectos que constituem a parte visível de um programa em Visual Basic têm propriedades (o próprio form é um objecto e tem propriedades).



Janela das propriedades

As propriedades de um controlo configuram-se na janela das propriedades - ***Properties Window***. Esta janela mostra sempre as propriedades do controlo (eventualmente um form)

seleccionado. Por esta razão, para configurar as propriedades de um objecto é necessário seleccioná-lo previamente.

Em alguns casos, o valor da propriedade pode ser escolhido de uma lista de opções predefinidas.

EXEMPLO (cont.)

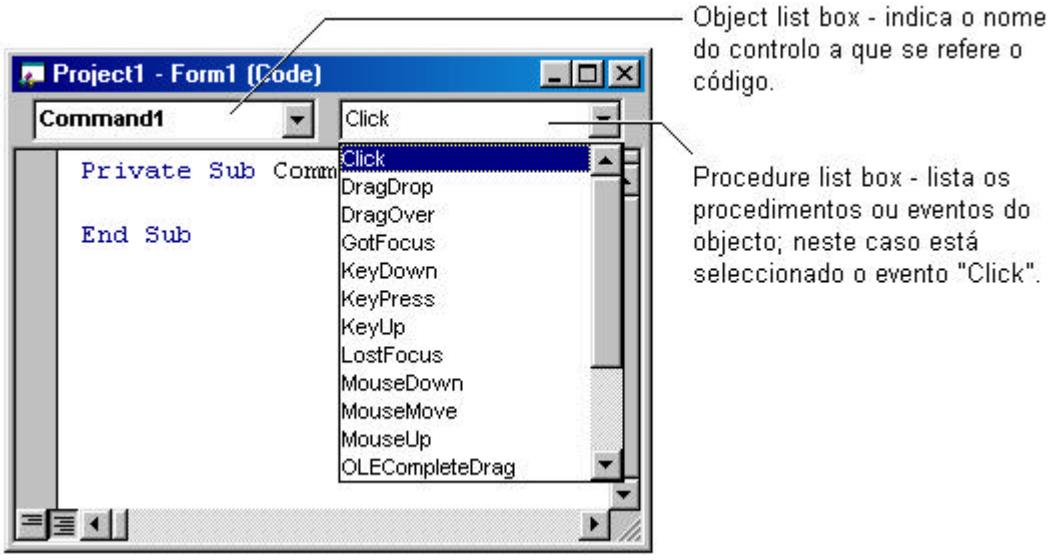
Altere então as seguintes propriedades:

Objecto	Propriedade	Valor
Form	Caption	Conversão Celsius – Farenheit
Label1	Caption	Graus Celsius:
Text1	Name	txtCelsius
	Text	(vazio)
Label2	Caption	Graus Farenheit:
Label3	Name	lblFarenheit
	Caption	(vazio)
Command1	Name	cmdConverter
	Caption	Converter

Escrever o código

O editor de código **- *Code Editor* -** do Visual Basic é onde se escrevem as instruções de resposta às acções do utilizador. Através deste editor pode-se rapidamente ver ou editar o código da aplicação.

Para abrir o editor de código basta fazer duplo-clique no controlo ou form para o qual se pretende escrever o código.



Editor de código

O código numa aplicação VB está dividido em pequenos blocos a que se dá o nome de procedimentos. Um procedimento de evento **- event procedure -** contém código que é executado quando o evento ocorre como por exemplo quando se prime um botão.

Cada objecto possui um conjunto próprio de eventos aos quais reage. O nome de um procedimento de evento é sempre composto pelos nomes do objecto e do evento separados por um caracter de *underscore* (_). Por exemplo, se quisermos que o botão **Command1** chame um evento quando premido, usamos o procedimento **Command1_Click**.

EXEMPLO (cont.)

Faça duplo-clique no botão ***Converter***, escolha o evento ***Click*** e no espaço entre as declarações "Private Sub cmdConverter_Click()" e "End Sub" escreva a seguinte instrução:

lblFarenheit.Caption = 1.8 * txtCelsius.Text +32

Correr e testar a aplicação

EXEMPLO (cont.)

Corra a aplicação seleccionando a opção ***Start*** que se encontra no menu *Run*, ou simplesmente premindo a tecla F5.

Para testar a aplicação introduza valores na caixa de texto relativa aos graus *Celsius*, prima o botão *Converter* e confirme o resultado que aparece na caixa de texto dos graus *Farenheit*.

Gravar o trabalho

Para guardar o trabalho usa-se a opção ***Save Project*** do menu *File*. O Visual Basic pede então para indicar os nomes dos forms e do projecto separadamente, bem como o local onde o trabalho ficará guardado.

O VB usa as extensões ".frm" para forms e ".vbp" para projectos.

Para efectuar alterações a um programa gravado bastará abrir o ficheiro do projecto, usando para o efeito a opção ***Open Project*** do menu *File*.

EXEMPLO (cont.)

Como o nosso pequeno programa tem um único form, o Visual Basic usará apenas dois ficheiros para o guardar. Por defeito o VB atribuirá os nomes "Form1.frm" e "Project1.vbp" ao form e projecto, respectivamente. Estes nomes podem no entanto ser alterados no momento da gravação.

Proceda então àgravação do trabalho. Poderá escolher um nome mais apropriado para o projecto, como por exemplo "ConverteGraus".

3 - Variáveis e Constantes

Tipos de dados

A informação processada por um programa pode ser de diferente natureza e existir em diferentes formatos. Genericamente um programa pode utilizar informação numérica e informação chamada alfanumérica, ou seja texto.

A linguagem Visual Basic suporta diversos tipos de dados, entre os quais:

Tipo	Descrição
Integer (inteiro)	usado para representar inteiros entre -32768 e 32767
String (cadeia de caracteres)	usado para representar informação alfanumérica como letras, algarismos e símbolos especiais.
Boolean (booleano)	usado para representar valores lógicos.

A partir dos tipos de dados preexistentes é ainda possível criar novos tipos de dados especificamente adaptados às necessidades do programador.

Conceito de variável

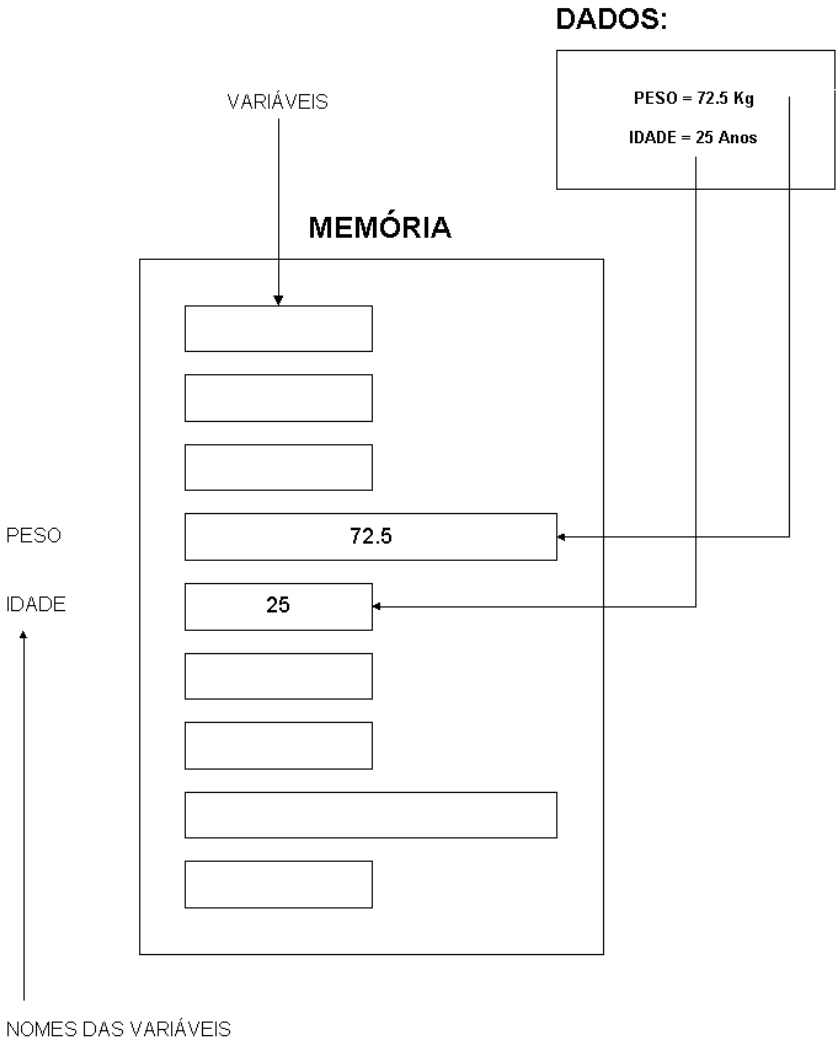
Uma variável é uma localização de memória em que a informação pode ser guardada de modo a ser usada por um programa. Cada variável é caracterizada pelo seu nome e pelo seu tipo, ou seja, o tipo de dados que pode armazenar.

O conteúdo de uma variável pode mudar durante a execução do programa.

Existem algumas regras governando a escolha do nome duma variável:

- Deve obrigatoriamente começar por uma letra
- Não pode conter espaços nem caracteres como vírgulas ou pontos
- Não pode exceder 255 caracteres

O tipo da variável especifica qual o tipo de dados que pode conter. Uma variável de um determinado tipo não está preparada para armazenar dados de um tipo diferente. A razão para este facto é que o espaço necessário para armazenar diferentes tipos de dados não é o mesmo. Enquanto um inteiro simples pode ser guardado em 2 bytes de memória [1], para guardar um número real pode-se necessitar de 8 bytes (ou mesmo mais, dependendo da precisão requerida).



Noção de variável

[1] Para armazenar números que podem variar entre -32768 e 32767, ou seja 65536 valores diferentes, há necessidade de dispor de 16 unidades básicas de informação (bits), ou seja dois bytes (1 byte = 8 bits). De facto, se cada bit apenas pode representar um valor binário (0 ou 1), 16 bits poderão representar até 2^{16}=65536 valores diferentes.

Declaração de variável

A declaração de variáveis é o acto pelo qual são criadas. Criar uma variável envolve dar-lhe um nome e reservar em memória o espaço necessário para que ela possa guardar o tipo de dados para o qual está a ser criada.

Nenhuma variável deve ser utilizada antes de ser criada. A declaração deve, pois, preceder a utilização.

Em Visual Basic, existem duas formas de declaração de variáveis: **explícita** e **implícita**.

A declaração **explícita** exige a utilização da instrução específica "**Dim ... As**" (Dimensionar...Como).

Por exemplo,

Dim Preço **As** Integer

...cria (declara) uma variável com o nome *Preço* e do tipo Integer, ou seja, preparada para receber dados do tipo inteiro simples.

Pode-se usar a mesma instrução **Dim** para declarar mais do que uma variável de uma só vez desde que se especifique para cada variável, individualmente, qual o seu tipo.

Por exemplo,

Dim Preço **As** Integer**,** Desconto **As** Single**,** IVA **As** Single

...cria a variável *Preço* como sendo do tipo Integer e as variáveis *Desconto* e *IVA* como sendo do tipo Single.

A declaração **implícita** resume-se a utilizar pela primeira vez uma variável sem qualquer declaração explícita prévia, dando-lhe um nome e atribuindo-lhe um valor. O Visual Basic cria automaticamente a variável do tipo correcto.

Esta segunda forma de declarar variáveis tem, a despeito da sua simplicidade, um problema grave: é possível, por distracção, criar uma variável nova indesejada, quando o que se pretendia era apenas escrever o nome de uma variável já existente. Atente-se no exemplo:

Suponha que havia criado uma variável **Distancia** mediante a instrução

Dim **Distancia** As Integer

...e que adiante no programa, por engano, escrevia **Distncia** ao referir-se àvariável em causa. O Visual Basic não emitirá nenhum alerta, já que aceitou tranquilamente **Distncia** como uma nova variável.

A forma mais prudente de lidar com declarações de variáveis é, pois, utilizar apenas declarações explícitas, e instruir o Visual Basic para não aceitar declarações implícitas, mediante a selecção da opção "Require Variable Declaration" no sub-menu **Options** do menu **Tools.**

Outros tipos de dados

A linguagem Visual Basic inclui, entre outros, os seguintes tipos de dados adicionais:

Tipo	Descrição
Long	inteiro longo, ou seja, compreendido entre -2.147.483.648 e 2.147.483.647
Single	real representado com precisão simples, com valores negativos compreendidos entre cerca de -3,4E38 e -1,4E-45 e valores positivos entre cerca de 1,4E-45 e 3,4E38
Double	real representado com precisão dupla
Date	uma variável deste tipo pode armazenar datas e horas

Operação de atribuição

A operação de Atribuição permite guardar um dado numa variável, ou seja, atribuir-lhe um valor.

A sintaxe utilizada por esta operação é a seguinte:

Variável = Valor

O resultado da operação será, portanto, o de guardar **Valor** em **Variável**. **Valor** pode ser um valor constante ou o conteúdo de outra variável. Neste caso, a atribuição consistirá na cópia do conteúdo de uma variável para outra do mesmo tipo. Conforme veremos mais à frente, pode ainda ser atribuído a **Variável** o resultado de uma expressão ou o valor devolvido por uma função.

Trata-se de uma operação destrutiva. Se a variável contiver já um valor, uma operação subsequente de atribuição sobre essa variável, substituirá o valor nela contido pelo novo valor.

Há ainda que ter em atenção o facto de que não é normalmente aconselhável atribuir um valor de um dado tipo a uma variável de tipo diferente. Os resultados podem ser a perda de informação ou o mau funcionamento do programa.

Constantes

Uma constante consiste num nome que é dado a um valor numérico ou a uma cadeia de caracteres, e que pode ser usado dentro do programa na vez desses valores. Funciona como uma espécie de sinónimo.

A utilização de constantes em substituição dos valores que representa justifica-se pelo seguinte facto: se um dado valor constante for utilizado muitas vezes ao longo dum programa, caso ocorra a necessidade de o modificar, haverá necessidade de corrigir manualmente todas as ocorrências desse valor, correndo, além disso, o risco de se enganar. Se, ao invés, for definida uma constante com esse valor, bastará modificar essa definição inicial para que tal mudança automaticamente se repercuta em todas as ocorrências dessa constante no decurso do programa.

A sintaxe da definição de constantes é a seguinte:

Const nome **As** tipo **=** expressão

Por *expressão* entende-se um valor numérico, uma cadeia de caracteres, ou uma expressão cujo resultado seja um destes tipos de valores.

Aqui ficam alguns exemplos de declarações de constantes:

Const **Pi** As Double **= 3.14159265358979**

Const **Raio** As Single **= 12.5**

Const **Perímetro** As Double **= 2 * Pi * Raio**

Operadores

A linguagem Visual Basic prevê os seguintes operadores, utilizáveis em expressões:

Operadores Aritméticos	
+	adição
-	subtracção
*****	multiplicação
****	divisão inteira
/	divisão real
MOD	resto da divisão inteira
^	exponenciação

Operadores Relacionais	
=	igualdade

<>	desigualdade
<	menor que
>	maior que
<=	menor ou igual a
>=	maior ou igual a

Operadores Lógicos	
AND	conjunção (e)
OR	disjunção (ou)
NOT	negação (não)

Aqui ficam as tabelas de verdade dos operadores lógicos apresentados.

A	**B**	**A AND B**
0	0	0
0	1	0
1	0	0
1	1	1

A	**B**	**A OR B**
0	0	0
0	1	1
1	0	1
1	1	1

A	**NOT A**
0	1
1	0

Existe ainda um operador usado para concatenação de strings. Esse operador é o **&**.

Comentários

Comentários ao código do programa podem ser inseridos mediante a utilização da "plica". Assim, todo o texto escrito a seguir a uma plica, será considerado como um comentário e não como pertencendo a uma instrução.

Exemplo:

```
' isto é um comentário
```

É de toda a conveniência a utilização de comentários para documentar o programa, o seu objectivo, as funcionalidades dos seus módulos, as técnicas particulares utilizadas, o significado de algumas variáveis mais importantes, e outros detalhes cujo significado não seja óbvio. Tal informação será preciosa quando futuramente se pretender corrigir ou modificar o programa.

Convenções a seguir na escolha de nomes de objectos

Se bem que não seja obrigatório, o seu uso é, no entanto, conveniente, para permitir identificar com facilidade o tipo de objecto a que um dado nome se refere.

Classe de Objecto	Nome começa por
Form	Frm
CommandButton	Cmd
Frame	Fra
Horizontal ScrollBar	Hsb
Image	Img
Label	Lbl
List Box	Lst
Menu	Mnu
Option Button	Opt
Picture Box	Pic
Text Box	Txt
Vertical SrcrollBar	Vsb

4 - Estruturas de controlo condicional

Muitos problemas requerem que diferentes cursos de acção sejam escolhidos em função do estado de uma determinada condição.

As **estruturas de controlo** permitem condicionar o fluxo do programa, ou seja, a sequência pela qual as instruções são executadas.

Estruturas de selecção simples

Uma estrutura de controlo fundamental é a estrutura **condicional**, ou de **selecção**. Usando esta estrutura, as instruções podem ser executadas condicionalmente. Se uma dada condição for verdadeira, será executada uma dada sequência de instruções. Se for falsa, uma sequência diferente será escolhida.

A estrutura **If ... Then** efectua o teste de uma *condição*. Se tal condição for verdadeira, desencadeará a execução das instruções representadas por *Acção1*. Em caso contrário, será executada a *Acção2*.

Sintaxe:

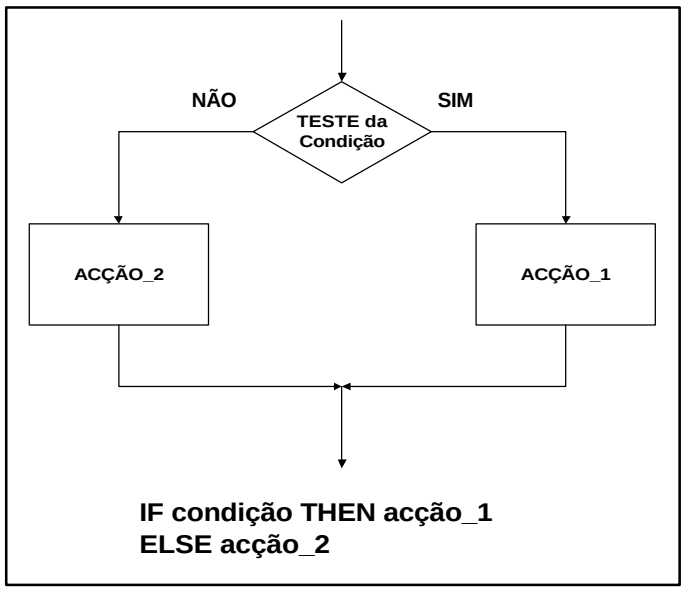
If condição **Then**

Acção1

Else

Acção2

End If



A condição pode consistir numa comparação, ou em qualquer expressão de que resulte um valor numérico: um valor não nulo será interpretado como Verdadeiro, enquanto um valor nulo será considerado como Falso.

A condição é, portanto, uma expressão booleana (lógica). Uma expressão booleana representa um valor booleano, TRUE (verdadeiro) ou FALSE (falso) e pode ser constituída por uma variável, uma função ou uma combinação destas entidades através de operações.

Exemplos de condições:

X >= 100

Saldo > limite_credito

(Nota > 0) And (Nota < 20)

(X >= 100) And ((Y=20) Or (Y=40))

sqr(X) <= 1000

Not encontrado

É possível combinar vários operadores lógicos na mesma condição

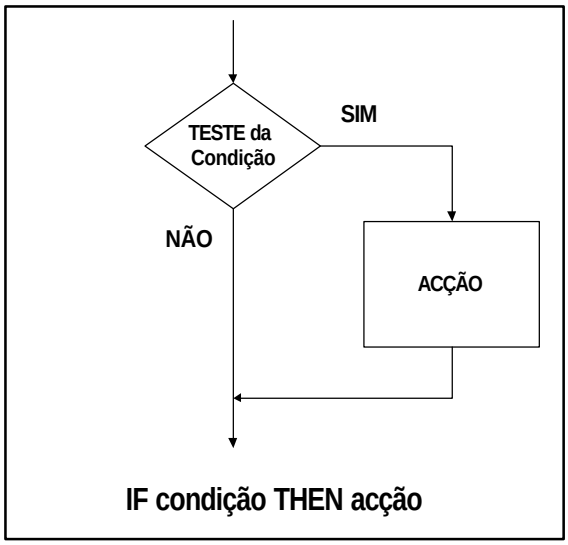
Há casos em que é desejável executar uma sequência de instruções caso uma dada condição se verificar, e nada no caso contrário. Nessa situação deve ser utilizada a seguinte variante da estrutura de controlo condicional:

Sintaxe:

If condição **Then**

Acção

End If



IF condição THEN acção

Nota: Caso a instrução **If...Then** seja escrita numa só linha, é dispensado o uso do delimitador "**End If**".

Exemplos de utilização:

1) **If** nota > 10 **Then**

Positivas = Positivas + 1

End If

2) **If** NotaTeste >= 10 **Then**

lblResultado.Caption = "Passado"

Else

lblResultado.Caption = "Reprovado"

End If

Estruturas de selecção embutidas

Estrutura If ... Then ... ElseIf ...

É possível imbricar estruturas condicionais dentro de outras estruturas condicionais, permitindo, assim, a construção de estruturas de controlo mais complexas. Para inserir uma estrutura condicional dentro de outra, é utilizada a palavra reservada **ElseIf**.

A sintaxe é a seguinte:

If condição1 **Then**

Acção1

ElseIf condição2 **Then**

Acção2

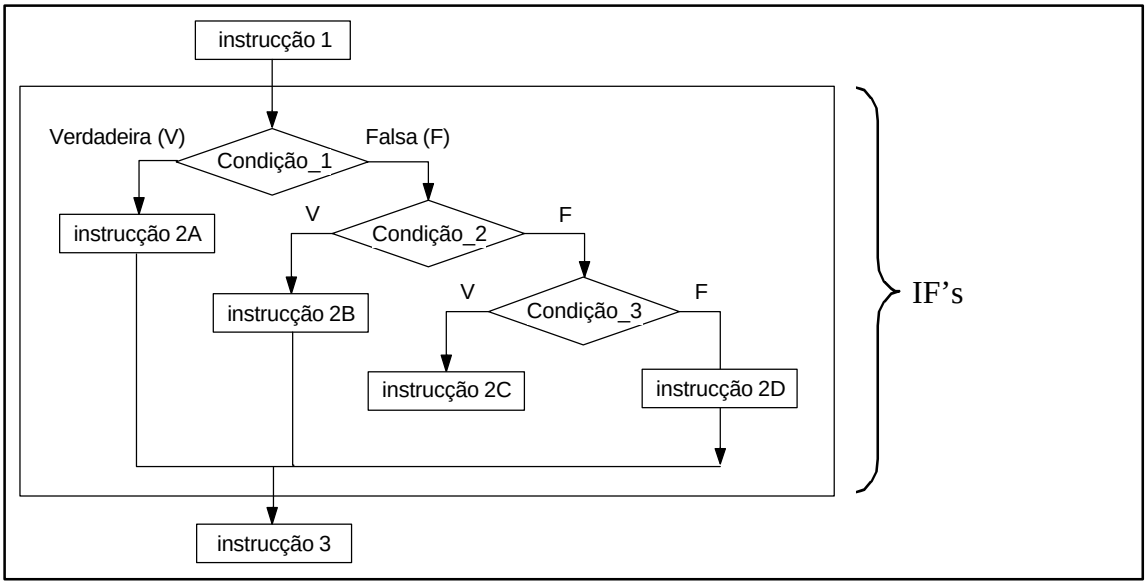
ElseIf condição3 **Then**

...

Else

AcçãoN

End If

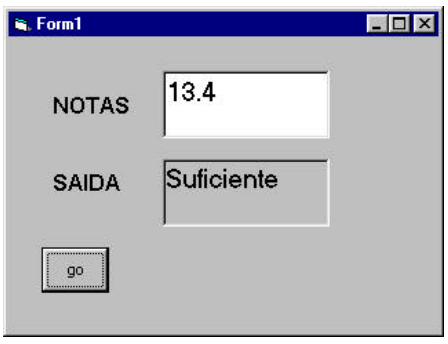


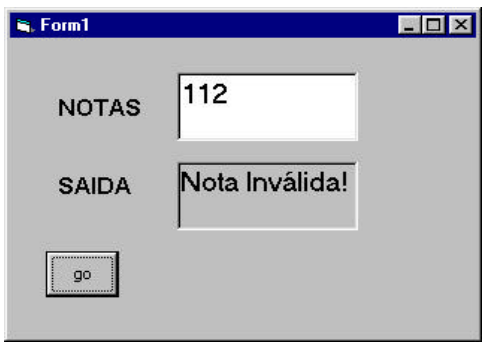
Esta estrutura condicional permite a selecção de uma entre várias alternativas mutuamente exclusivas. As instruções que se seguem à palavra reservada **Else** (aqui representadas por "AcçãoN") serão executadas apenas se nenhuma das condições se tiver verificado.

É possível imbricar um qualquer número de blocos **ElseIf** dentro de uma dada estrutura condicional.

Exemplo:

```
Private Sub cmdGo_Click()

    Dim nota As Single

    nota = txtNotas.Text

    If (nota < 0) Or (nota > 20) Then

        lblSaida.Caption = "Nota Inválida!"

    ElseIf nota < 6 Then

        lblSaida.Caption = "Mau"

    ElseIf nota < 10 Then

        lblSaida.Caption = "Medíocre"

    ElseIf nota < 14 Then

        lblSaida.Caption = "Suficiente"

    ElseIf nota < 17 Then

        lblSaida.Caption = "Bom"

    Else

        lblSaida.Caption = "Muito Bom"

    End If

End Sub
```





Estrutura Select ... Case

A estrutura de selecção **Select ... Case** é muito semelhante à estrutura **If ... Then ... ElseIf**, no entanto, a sua utilização apenas pode ser feita nas situações em que a condição a testar depende do valor de uma única variável.

Sintaxe da estrutura Select ... Case:

```
Select Case variável
    Case valor1
        Acção1
    Case valor2
        Acção2
    Case valor3
        …
    Case Else
        AcçãoN
End Select
```

Pode ser incluído um número qualquer de cláusulas **Case** numa estrutura Select ... Case e pode-se indicar mais do que um valor numa cláusula **Case**, desde que separados por vírgulas.

É também possível utilizar operadores de comparação (operadores relacionais) para indicar gamas de valores nas cláusulas **Case**. Para utilizar esses operadores temos que recorrer às palavras reservadas **Is** ou **To** de forma a identificar o tipo de comparação.

Exemplos de cláusulas Case:

Case 3.5	...se a variável tiver o valor 3.5
Case 1, 2	...se a variável tiver o valor 1 ou 2
Case Is <= 500	...se a variável tiver um valor menor ou igual a 500
Case 10 **To** 20	...se a variável tiver um valor entre 10 e 20, inclusive

Tal como na estrutura If … Then … ElseIf, a selecção da acção a tomar é feita de entre várias alternativas mutuamente exclusivas. A acção relativa à cláusula **Case Else** será executada apenas se o valor da variável a testar não "encaixar" em nenhuma cláusula anterior.

Exemplo:

A substituição da estrutura If ... Then ... ElseIf, usada no exemplo atrás apresentado, pela estrutura Select ... Case poderia ser feita da seguinte forma:

```
Private Sub cmdGo_Click()

    Dim nota As Single

    nota = txtNotas.Text

    Select Case nota

        Case Is < 0

            lblSaida.Caption = "Nota Inválida!"

        Case Is < 6

            lblSaida.Caption = "Mau"

        Case Is < 10

            lblSaida.Caption = "Medíocre"

        Case Is < 14

            lblSaida.Caption = "Suficiente"

        Case Is < 17

            lblSaida.Caption = "Bom"

        Case 17 To 20

            lblSaida.Caption = "Muito Bom"

        Case Else

            lblSaida.Caption = "Nota Inválida!"

    End Select

End Sub
```

5 - Estruturas de controlo repetitivo

Enquanto as estruturas de controlo condicional permitem alterar o fluxo do programa mas sem deixar de executar de forma linear uma dada sequência de instruções, as estruturas de controlo repetitiva (também conhecidas por *ciclos*) permitem repetir um dado conjunto de instruções o número de vezes que for necessário.

Existem diversas variantes de *ciclos*, diferindo umas das outras pela forma como é controlada a execução das instruções contidas no corpo do ciclo.

Genericamente, pode-se dizer que uma estrutura de controlo repetitiva (ou *ciclo*) assegura a execução repetida de um dado conjunto de instruções dependendo do resultado do teste de uma determinada condição de funcionamento.

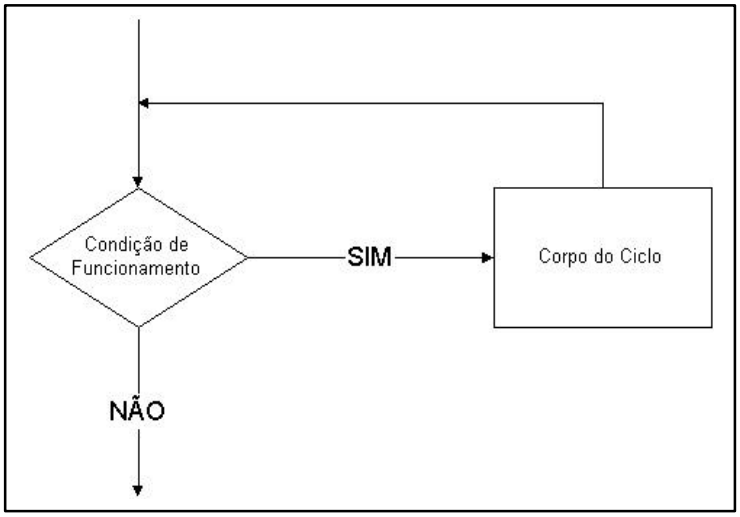


Diagrama de fluxo possível de um ciclo

Ciclo For ... To ... Next

Este ciclo permite repetir um dado conjunto de instruções um número predeterminado de vezes. Nem sempre é possível saber à partida quantas vezes as instruções contidas no corpo do ciclo devem ser repetidas. Nesse caso, deverão ser utilizadas outras estruturas de controlo repetitivo que não o ciclo "For ... To ... Next".

A sintaxe desta estrutura de controlo repetitiva é:

For variável **=** início **To** fim **Step** passo

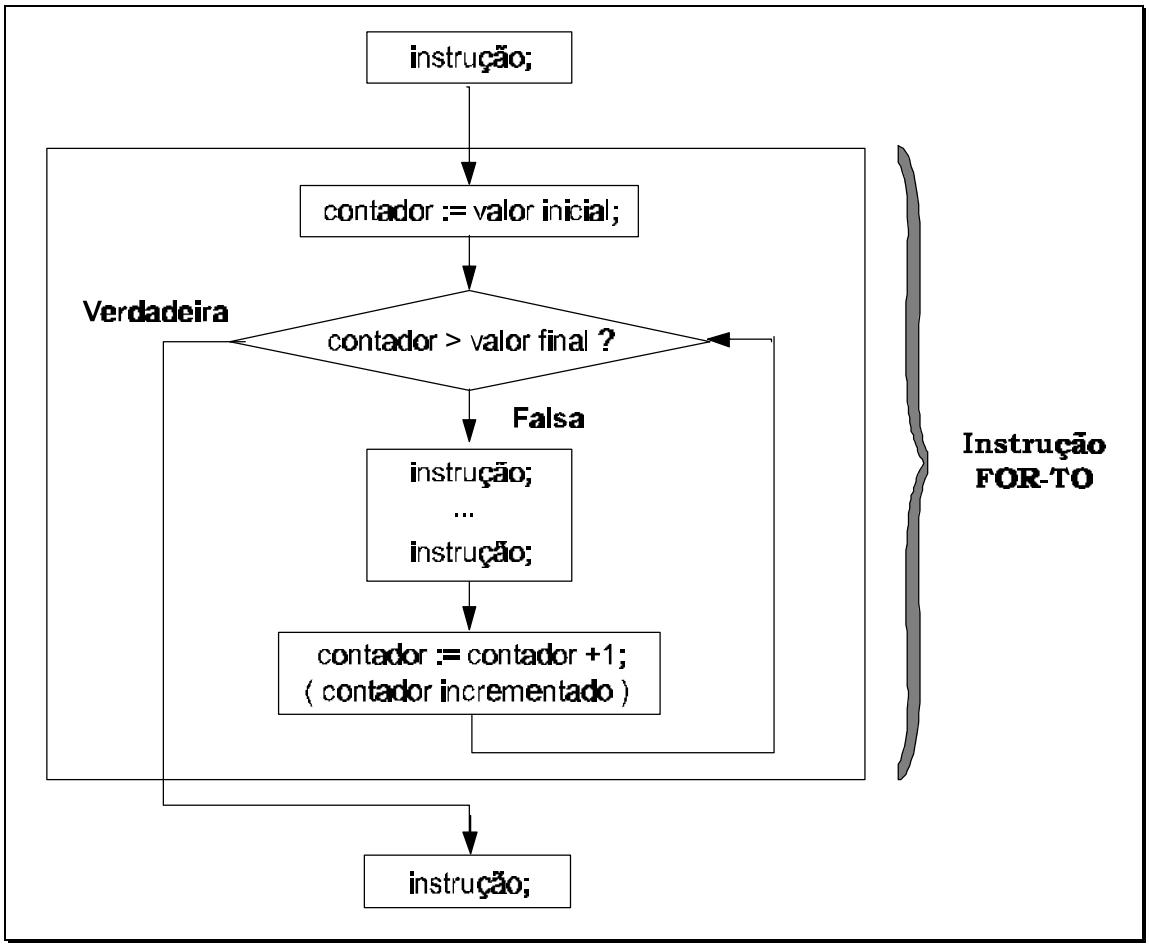
Instrução1

Instrução2

…

InstruçãoN

Next variável



Funcionamento do ciclo For … To … Next

Esta estrutura baseia-se na existência dum contador que incrementa automaticamente o conteúdo de *variável*, chamada variável de controlo do ciclo, cada vez que o ciclo funciona, isto é, cada vez que as instruções contidas no corpo do ciclo são executadas. O valor inicial dessa variável é *início*.

A inicialização da *variável* contadora, o seu incremento/decremento e a verificação da condição de funcionamento do ciclo (*variável* <= *fim*) é da responsabilidade da própria estrutura de controlo. O programador deve, apenas, especificar qual o valor de *início* e de *fim* (ou, por outras palavras, o número de vezes que o ciclo vai funcionar) e quais as instruções que o ciclo vai repetir (o *corpo* do ciclo).

O conteúdo da variável de controlo do ciclo pode ser utilizado por instruções contidas no corpo do ciclo, mas não deve, sob pretexto algum, ser modificado por estas instruções, sob pena de se perder o controlo do funcionamento do ciclo.

A estrutura de controlo verifica no início de cada iteração (repetição) do ciclo se a condição de funcionamento do ciclo é ainda verdadeira. Caso seja falsa, o ciclo terminará, e o programa passará a executar as instruções que se seguem.

A inclusão de **Step** é opcional: caso se pretenda que o conteúdo da variável contadora seja incrementada uma unidade de cada vez, é desnecessário especificar *passo*, caso contrário, *passo* permitirá incrementar o valor de variável de um valor diferente da unidade (positivo ou negativo).

Caso o valor de *passo* seja positivo a variável contadora será incrementada. Se pretendermos, no entanto, efectuar um decremento, deverá ser utilizado um valor negativo para *passo*. Obviamente, nesse caso, a condição de funcionamento do ciclo passará a ser *variável* >= *fim*.

Exemplos:

1)
```
' o ciclo abaixo calcula o factorial de Numero

Factorial = 1

For num = 1 To Numero

        Factorial = Factorial * num

Next num
```

2)
```
' versão utilizando decremento da variável de controlo (num)

Factorial = 1

For num = Numero To 1 Step -1

        Factorial = Factorial * num

Next num
```

3)
```
' determinação do maior divisor de um dado número inteiro (sem ser o próprio número)

MaxDivisor = 1

For CandidatoADivisor = 2 To Numero \ 2

        If Numero Mod CandidatoADivisor = 0 Then MaxDivisor = CandidatoADivisor

Next CandidatoADivisor
```

```
If MaxDivisor = 1 Then
        MaxDivisor = Numero
        lblSaida.Caption = Str(Numero) & " é número primo"
Else
        lblSaida.Caption = MaxDivisor
End If
```

Ciclo Do While

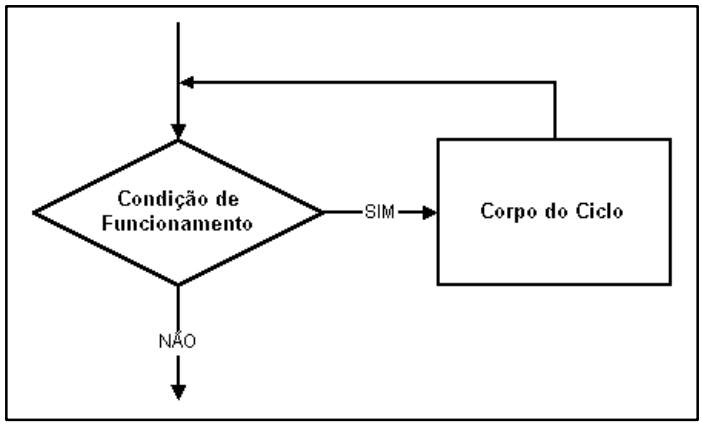
A estrutura **For ... To ... Next** permite realizar um ciclo controlado por um variável contadora que é incrementada ou decrementada até atingir um valor final pré-fixado. É possível obter este efeito usando estruturas de controlo alternativas, como seja a estrutura **Do While**.

Sintaxe:

Do While condição

[Corpo do Ciclo]

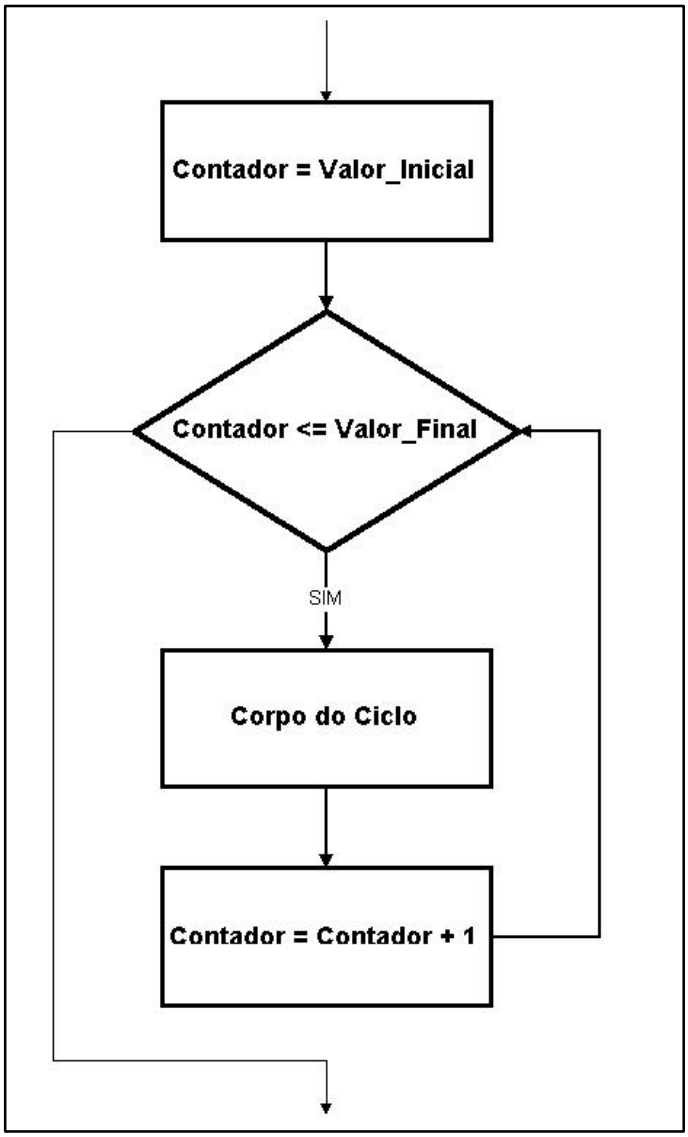
Loop



Ciclo controlado por contador

Um ciclo controlado por contador baseado na estrutura **Do While** pode assumir a seguinte forma genérica:

```
contador = valor_inicial
Do While contador <= valor_final
        ' Corpo do Ciclo
        contador = contador + 1
Loop
```



Ciclo controlado por contador

Exemplo:

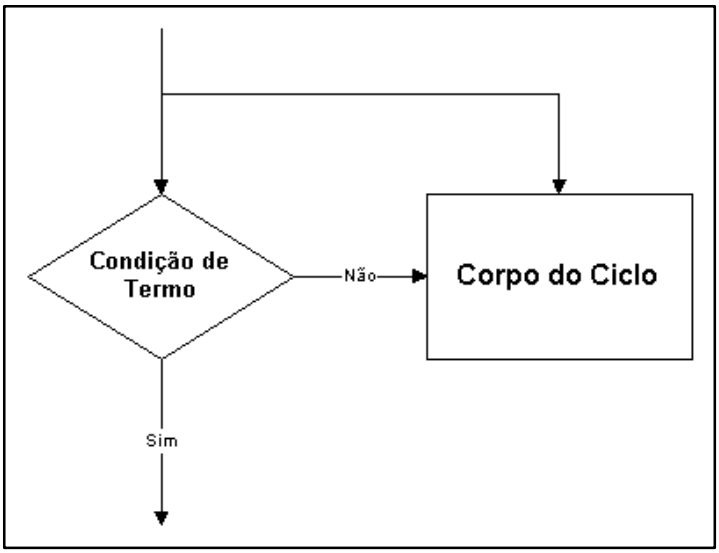
```
Factorial = 1
contador = 1
Do While contador <= Numero
    Factorial = Factorial * contador
    contador = contador + 1
Loop
```

Questões a ter em conta na construção de um ciclo **Do While** controlado por contador:

- Especificar a condição de funcionamento do ciclo
- Inicializar a variável contadora
- Incluir no corpo do ciclo uma instrução que incremente ou decremente a variável contadora

Estas acções encontram-se asseguradas de forma automática no caso da estrutura **For ... To ... Next**. Em todas as outras formas de ciclos, como seja o ciclo **Do While**, é da responsabilidade do utilizador assegurar-se de que tais acções são correctamente executadas.

Ciclo Do Until



Sintaxe:

Do Until condição

[Corpo do Ciclo]

Loop

Neste caso a condição especificada é a condição de termo do ciclo. O ciclo será executado até que ("until") a condição de termo seja verdadeira. É possível, assim, transformar um ciclo **Do While** em **Do Until** negando a condição de funcionamento.

Exemplo:

```
Factorial = 1

contador = 1

Do Until contador > Numero

        Factorial = Factorial * contador

        contador = contador + 1

Loop
```

Ciclos controlados por sentinela

Nem sempre é possível conhecer à partida o número de vezes que o ciclo vai ser executado, ou seja, quantas vezes as instruções contidas no corpo do ciclo vão ser repetidas. Nesses casos, não é viável a utilização de ciclos controlados por contador, devendo-se usar uma técnica diferente: ciclos controlados por sentinela.

Por *sentinela* deve entender-se um valor limite que assinala o fim de uma dada sequência de valores, mas que não esteja incluído nesse conjunto de valores.

Por exemplo, se o utilizador estiver a introduzir os dados referentes a um conjunto de alunos identificados pelos seus números de matrícula, a introdução de um número com menos de 6 dígitos (no caso do ISEP) como por exemplo o valor 1, permitirá indicar ao programa que a presente sequência de introdução de dados deve terminar.

A selecção do valor *sentinela* é da responsabilidade do programador, devendo ser escolhido fora do intervalo possível de valores a introduzir, podendo ainda, ter-se em atenção a possível ocorrência de valores fora desse intervalo que possam resultar de algum eventual erro de digitação. O valor *sentinela* escolhido não deve pois ser passível de facilmente ocorrer por mero acidente.

Exemplos:

1)
```
Dim Numero As Long

Do While Numero <> 1

    Numero = InputBox ("Digite o numero de identificação",
                       "ISEP - Entrada de dados de alunos")

    If Numero <> 1 Then lstbox1.AddItem Numero

Loop
```

Nota: Neste exemplo, a variável *numero* não foi inicializada pelo programador. Quando esta tarefa não é realizada antes da utilização da variável, o próprio Visual Basic se encarrega de a inicializar a zero.

2)
```
num = InputBox("Introduza um número positivo não nulo.", "Cálculo de factoriais")

Do Until num < 0

    ' calculo e apresentação do factorial de num

    num = InputBox("Introduza um numero positivo não nulo.",
                   "Cálculo de factoriais")

Loop
```

Ciclos imbricados

Foi dito anteriormente que o **corpo** de um ciclo era constituído pelo conjunto de instruções que o ciclo irá executar repetidamente. Foi também dito que nesse conjunto de instruções se poderiam incluir qualquer tipo de instruções, mesmo constituindo outras estruturas de controlo repetitivo. Prosseguindo com os exemplos apresentados na secção anterior, abaixo se pode ver um exemplo de uma utilização de um ciclo dentro de outro. Destes ciclos se diz que se encontram **imbricados** um dentro do outro.

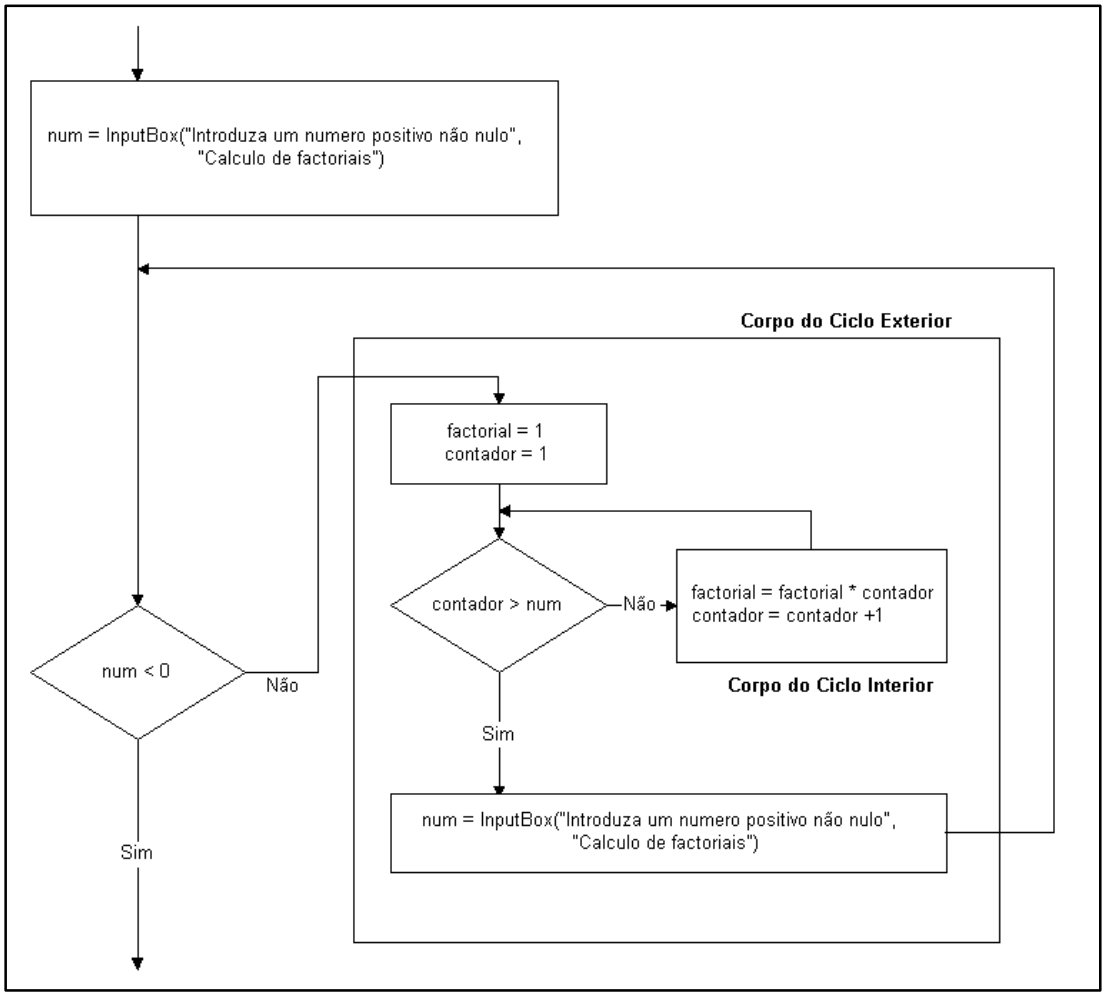


Ilustração gráfica dos dois ciclos imbricados do exemplo seguinte

Exemplos:

1)

```
'Leitura do 1º número cujo factorial vai ser calculado

num = InputBox("Introduza um numero positivo não nulo", "Calculo de factoriais")

Do Until num < 0

    'calculo do factorial de num

    factorial =  1

    contador = 1

    Do Until contador > num

        factorial = factorial * contador

        contador = contador +1

    Loop

    'Leitura do número seguinte

    num = InputBox("Introduza um número positivo não nulo",
                   "Cálculo de factoriais")

Loop
```

Este exemplo utiliza apenas ciclos controlados por sentinela. Quando se usa um ciclo controlado por contador imbricado num outro ciclo controlado por contador, há que ter o cuidado de criar e utilizar diferentes variáveis de controlo para cada ciclo.

2)

```
...

For i = 1 To Njornadas

    For j = 1 To Nequipas

        'Corpo do ciclo interior

    Next j

Next i

...
```

Neste exemplo, para cada valor da variável de controlo **i** (do ciclo exterior), o ciclo interior é executado *Nequipas* vezes. As instruções contidas no corpo do ciclo interior serão assim efectuadas um total de *Njornadas x Nequipas* vezes.

Ciclos com teste no fim

Nas estruturas **Do While** e **Do Until**, o teste da condição de funcionamento (ou de termo) é efectuada no início, isto é, antes da execução das instruções contidas no corpo do ciclo.

Estão disponíveis, no entanto, variantes das estruturas referidas, em que esse teste é efectuado após a execução do corpo do ciclo. O resultado óbvio desta particularidade é o facto de as instruções constantes do corpo do ciclo serem obrigatoriamente executadas pelo menos uma vez. Efectivamente, mesmo que a condição de funcionamento seja à partida falsa (ou a condição de termo seja à partida verdadeira) o corpo do ciclo será executado, pois a verificação dessas condições será efectuada num momento posterior.

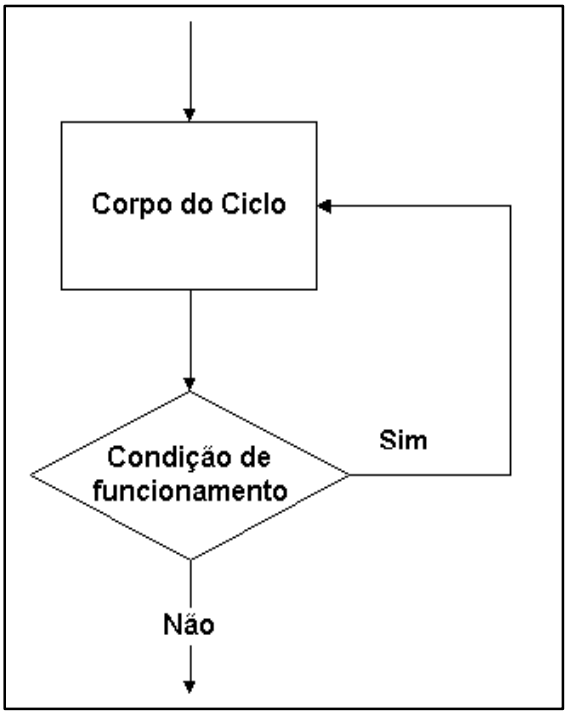
Essas variantes são as estruturas de controlo repetitivo **Do ... Loop While** e **Do … Loop Until**.

Sintaxe Do … Loop While:

Do

[Corpo do Ciclo]

Loop While condição



Exemplo:

```
Do
        ref = Val(InputBox("Nova referência:", "Programa de Stock"))
        If ref > 0 Then
                lstLista.AddItem ref
        End If
Loop While ref > 0
```

O ciclo do exemplo anterior é controlado por sentinela (qualquer valor negativo). Mesmo que um valor negativo seja introduzido em primeiro lugar, o corpo do ciclo será executado, pelo que

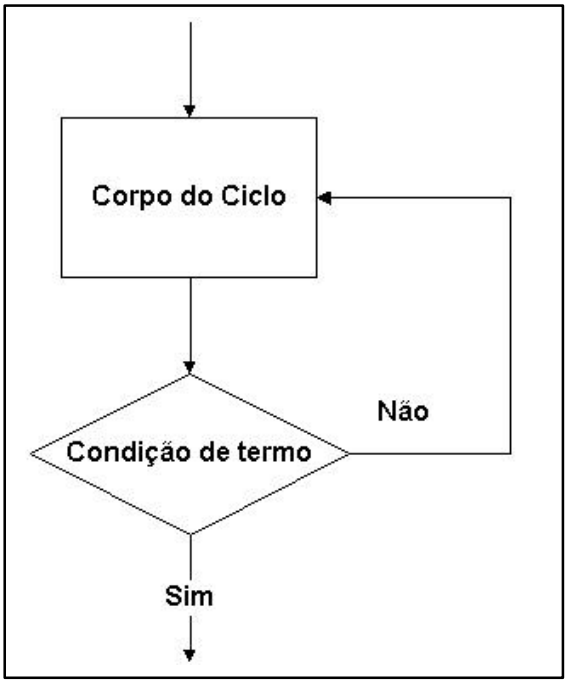
será necessário prever um teste adicional do valor *ref* lido para evitar que um valor sentinela seja adicionado àlista.

Sintaxe Do … Loop Until:

Do

[Corpo do Ciclo]

Loop Until condição



Exemplo:

```
Num = Val(InputBox("Digite um nº positivo ou 0 p/ acabar", "Entrada de dados"))
Do
    Soma = soma + num
    Num = Val(InputBox("Digite um nº positivo ou 0 p/ acabar", "Entrada de dados"))
Loop Until num < 0
```

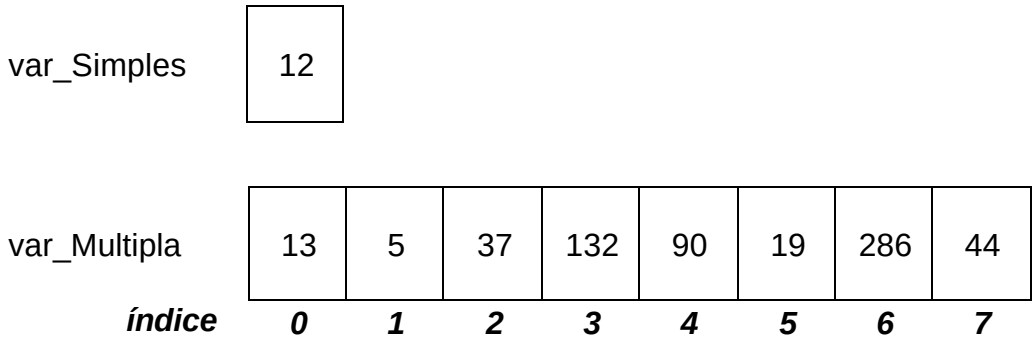
Caso o primeiro número a ser lido seja um número negativo, a condição de termo será de imediato verdadeira, mas como o seu teste só será feito após a execução do corpo do ciclo, a instrução *soma = soma + num* será executada uma vez. Nesse caso, seria talvez mais aconselhável utilizar uma estrutura com teste no início, ou adicionar um teste suplementar ao corpo do ciclo, como no primeiro exemplo.

6 - Vectores e Matrizes

Vectores

Até agora, temos trabalhado essencialmente com variáveis que podemos classificar como individuais, isto é, cada variável podendo conter ao mesmo tempo apenas um só valor. Como essas variáveis não podem conter simultaneamente mais que um dado, a atribuição de um novo valor a essa variável implica a destruição do valor anteriormente nela contido.

Mediante a utilização de um novo tipo de variáveis, as variáveis do tipo **Array** (**Vector**), passa a ser possível armazenar na mesma variável múltiplos valores desde que sejam do mesmo tipo. Estamos, portanto, a utilizar agora variáveis que se podem classificar como variáveis múltiplas.



Uma variável simples var_Simples e um **vector** var_Múltipla com 8 elementos

Um vector é uma lista ordenada de variáveis simples do mesmo tipo. Pode também ser visto como um conjunto de variáveis simples agrupadas. Todos as variáveis membros desse vector partilham o mesmo nome (o **nome** do vector). São identificadas individualmente mediante o valor dum **índice**, que determina qual a sua posição dentro do vector.

Os valores do **índice** devem obrigatoriamente ser do tipo Integer. O primeiro valor do índice é zero.

Um elemento de um vector é identificado utilizando o nome do vector seguido do valor do índice dentro de parêntesis:

nome_vector (índice)

Exemplos:

var_Multipla(3) ‘ 4º elemento do vector ‘var_Multipla’

var_Multipla(7) ‘ 8º e último elemento do vector ‘var_Multipla’

notas(14) ‘ 15º elemento do vector ‘notas’

nomes(0) ‘ 1º elemento do vector ‘nomes’

Declaração de vectores

Como qualquer outra variável, uma variável do tipo Array deve também ser declarada (criada) antes de poder ser usada. Para tal, deve ser usada a instrução **Dim**, que reserva espaço em memória suficiente para armazenar o número previsto de elementos do vector[2]. Uma das formas de utilizar a instrução Dim para declarar vectores é a seguinte:

Dim ***nome_vector(num_elementos)*** As ***Tipo***

Exemplos:

Dim var_Multipla(8) As Integer

Dim notas(30) As Single

Dim nomes(100) As String

Nota: *num_elementos* não se refere ao valor máximo que a variável índice pode assumir (7, no caso do vector var_Multipla) mas sim ao número de elementos do vector (8, neste caso).

Usando esta forma na declaração de vectores, o processo de indicar o limite inferior faz-se de forma implícita: por defeito assume-se como limite inferior do índice o valor zero (ou 1 se tal for especificado mediante a instrução *Option Base 1*).

Uma forma alternativa de utilizar a instrução Dim para declarar vectores implica a utilização da palavra reservada **To**, permitindo especificar o menor e o maior valor que o índice pode assumir:

Dim ***nome_vector(menorÍndice To maiorÍndice)*** As ***Tipo***

Exemplos:

Dim numeros(100 **To** 200) As Integer

Dim valores(-100 **To** 100) As Single

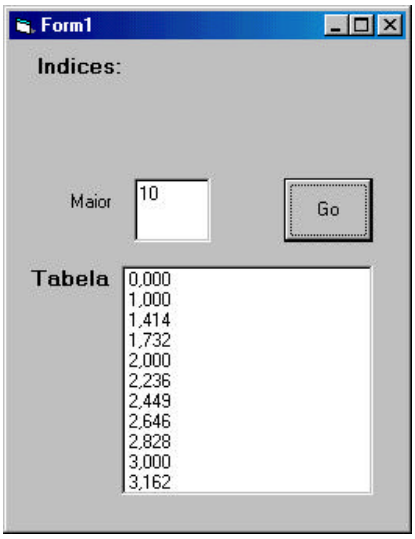
Processamento de vectores

Sendo um vector uma variável múltipla composta por elementos do mesmo tipo agrupados na mesma estrutura, a forma mais adequada de executar uma mesma acção sobre parte ou a totalidade dos seus elementos é utilizando uma estrutura de controlo repetitivo ou ciclo.

[2] Adicionalmente, a instrução Dim atribui valores iniciais a todos os elementos do vector (zeros no caso de vectores numéricos e strings nulas no caso de vectores alfa-numéricos)

Exemplos:

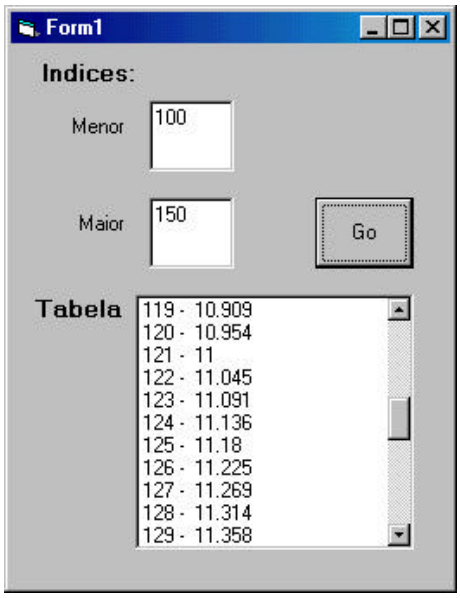
1) Private Sub cmdGo_Click()




```
    Dim vectorSqr(100) As Double
    Dim Maior as Integer
    Dim i As Integer            ' variável índice
    Dim res As Single
    lstTabela.Clear
    Maior = Val(txtMaior.Text)
    For i = 0 To Maior
        res = Sqr(i)
        vectorSqr(i) = res
        lstTabela.AddItem Format(res, "0.000")
    Next i
End Sub
```

Este programa permite calcular e apresentar sob a forma de uma tabela as raízes quadradas de todos os números inteiros compreendidos entre 0 e um dado limite superior a especificar pelo utilizador na TextBox **txtMaior** (a largura do intervalo não deve exceder 100*)*. Os valores calculados são armazenados num vector para eventual futura utilização.

2) Private Sub cmdGo_Click()




```
    Dim vector(100) As Double
    Dim Menor As Integer
    Dim Maior As Integer
    Dim i As Integer
    Dim indice As Integer
    Dim res As Single
    Dim linha As String
    lstTabela.Clear
    Menor = Val(txtMenor.Text)
    Maior = Val(txtMaior.Text)
    indice = 0
```

```
    For i = Menor To Maior
        res = Sqr(i)
        vector(indice) = res
        linha = Format(Str(i), "##0") & " - " & Format(vector(indice), "000.000")
        lstTabela.AddItem linha
        indice = indice + 1
    Next i
End Sub
```

Nesta variante é possível especificar também o limite inferior do intervalo, para além de se demonstrar algumas técnicas de formatação da saída de dados.

De notar que na primeira versão do programa (*exemplo 1*) se usou a mesma variável **i** para controlar o ciclo **For** e para armazenar os valores nos índices do **vector**. Tal aconteceu porque foi possível estabelecer naquele caso uma correspondência directa entre os valores da variável de controlo do ciclo **i** e os valores do índice que controla as posições dos elementos do vector.

Nesta segunda versão tal não era possível, visto que a variável de controlo do ciclo iria conter valores (desde o limite inferior ao limite superior do intervalo) que poderiam não corresponder às posições do vector em que os resultados iriam ser armazenados.

Em qualquer das variantes apresentadas, o processamento dos elementos do vector consistiu em operações de escrita (de atribuição) que modificaram o seu conteúdo. É igualmente possível efectuar operações de leitura sobre todos ou parte dos elementos dum vector. Nesse caso, como é óbvio, a variável do tipo *Array* deverá encontra-se do lado direito da operação de atribuição:

```
var = vector(índice)
```

Pesquisa em vectores

Vários métodos podem ser seguidos para procurar informação armazenada em vectores. O método mais simples é o da **pesquisa linear** (ou sequencial).

Tal método consiste em posicionarmo-nos no primeiro elemento do vector e comparar o conteúdo desse elemento com o valor a procurar. Caso sejam iguais a busca termina e no caso de não serem iguais, efectua-se a mesma comparação em relação ao conteúdo do elemento seguinte e assim sucessivamente até que a busca obtenha sucesso, ou até que o último elemento do vector seja alcançado.

vec	49	13	5	37	132	90	19	286	44
índice	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***

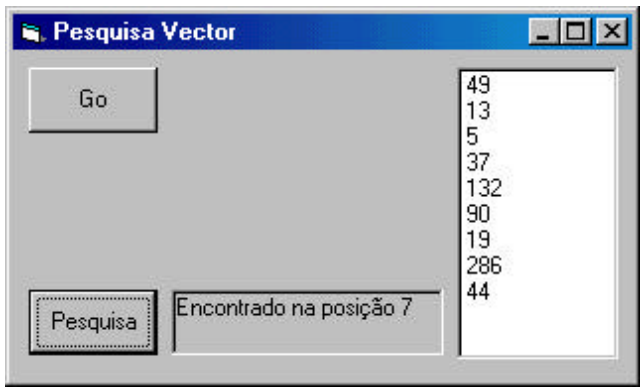
num	19

Exemplo:

Para verificar se o valor contido na variável **num** se encontra armazenado no vector **vec**, podemos usar o programa que a seguir se apresenta.

Este programa encontra-se dividido em dois procedimentos, um associado ao botão de comando "Go" e o outro ao botão "Pesquisa". O primeiro serve para encher o vector **vec** com valores que possam posteriormente ser pesquisados. O segundo destina-se a pesquisar dentro de **vec** a existência de um dado número inteiro a indicar pelo utilizador.

Repare-se nas duas declarações iniciais colocadas fora de qualquer procedimento. Diz-se que estas declarações são **globais** já que, pelo facto de se encontrarem na secção "General" do programa em vez de inseridas num dado procedimento, são reconhecidas por todos os procedimentos do programa, e portanto, acessíveis de dentro de qualquer deles.



```
Const N As Integer = 9
Dim vec(N) As Integer

Private Sub cmdGo_Click()
    Dim i As Integer
    Dim texto As String
    lstVector.Clear
    For i = 0 To N - 1
        texto = "Vec(" + str(i) + "):"
        vec(i) = Val(InputBox(texto, "Entrada de Dados"))
        lstVector.AddItem vec(i)
    Next i
End Sub

Private Sub cmdPesquisa_Click()
    Dim i As Integer
    Dim num As Integer
    num = Val(InputBox("Valor a pesquisar:", "Pesquisa em Vector"))
    i = 0
    Do Until (num = vec(i)) Or (i > N - 1)
        i = i + 1
    Loop
    If i < N Then
        lblBusca.Caption = "Encontrado na posição" + str(i + 1)
    Else
        lblBusca.Caption = "Não encontrado"
    End If
End Sub
```

Outro método bastante usado para procurar informação armazenada em vectores é o método da **pesquisa binária**. Trata-se de uma alternativa mais rápida à procura sequencial mas exige que os elementos do vector estejam ordenados (por ordem ascendente ou descendente).

Este método consiste em pesquisar sucessivamente os elementos a meio de subintervalos do vector, subintervalos esses que vão sendo reduzidos a metade. Começa-se então por verificar o elemento a meio do vector. Se não for o valor procurado, então este só pode estar na metade inferior ou superior do vector, uma vez que é suposto o vector estar ordenado. Assim, a segunda pesquisa incide apenas sobre metade do vector. Este processo repete-se até se encontrar o elemento desejado (sucesso) ou determinar que não se encontra no vector (insucesso). Repare-se que o número de elementos a pesquisar vai sendo reduzido sucessivamente a metade. Daí a designação de pesquisa binária e a razão pela qual é mais rápida que a sequencial.

Exemplo 1:

Para verificar se o valor contido na variável **num** se encontra armazenado no vector **vec** (considerando que este vector se encontra ordenado de forma ascendente) podemos usar o procedimento seguinte:

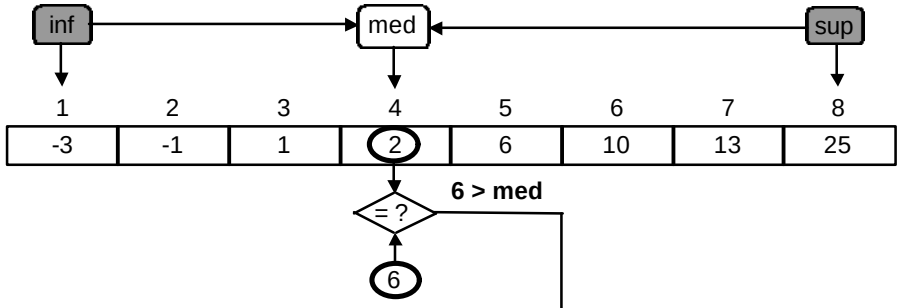
```
Private Sub cmdPesquisa_Click()

    Dim num As Integer, inf As Integer, sup As Integer, med As Integer

    Dim encontrado As Boolean

    num = Val(InputBox("Valor a pesquisar:", "Pesquisa em Vector"))

    encontrado = False

    inf = 0

    sup = N - 1

    Do While inf <= sup And Not encontrado

        med = (inf + sup) \ 2

        If num > vec(med) Then

            inf = med + 1

        ElseIf num < vec(med) Then

            sup = med – 1

        Else

            encontrado = True

        End If

    Loop
```

```
        If encontrado = True Then
            lblBusca.Caption = "Encontrado na posição" + str(i + 1)
        Else
            lblBusca.Caption = "Não encontrado"
        End If
    End Sub
```

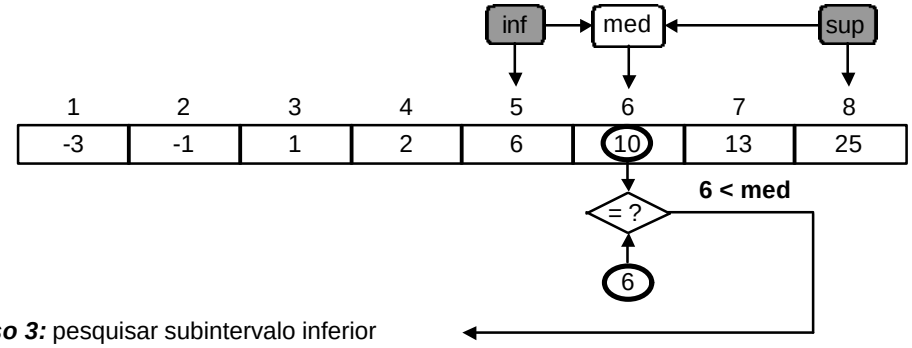
Exemplo 2:

Sucesso na pesquisa do valor 6

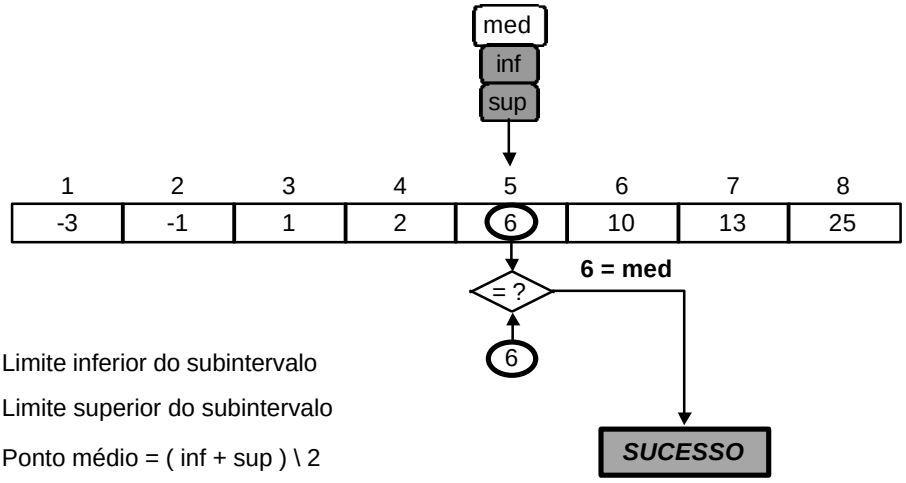
Passo 1:



Passo 2: pesquisar subintervalo superior



Passo 3: pesquisar subintervalo inferior



inf — Limite inferior do subintervalo

sup — Limite superior do subintervalo

med — Ponto médio = (inf + sup) \ 2

Ordenação de vectores

Existem vários métodos para ordenar vectores e essa ordenação pode sempre ser feita de forma ascendente (do menor valor para o maior) ou descendente (do maior para o menor valor). Um dos métodos mais simples é o da **ordenação por selecção**.

Este método consiste em posicionarmo-nos no primeiro elemento do vector e comparar o seu conteúdo com todos os outros elementos. Sempre que se encontrar um elemento cujo valor seja menor que o da primeira posição então os valores são trocados entre si. Quando a comparação do primeiro elemento do vector com todos os outros termina temos a garantia de que o valor do primeiro elemento é já o menor de todos.

Nesta altura avança-se para o segundo elemento e compara-se o seu conteúdo com todos os elementos que lhe seguem no vector (terceiro, quarto, ...) trocando os valores sempre que se encontre um elemento de menor valor.

Este processo repete-se até se chegar ao fim do vector, altura em que se conclui a ordenação do vector.

Vector desordenado:

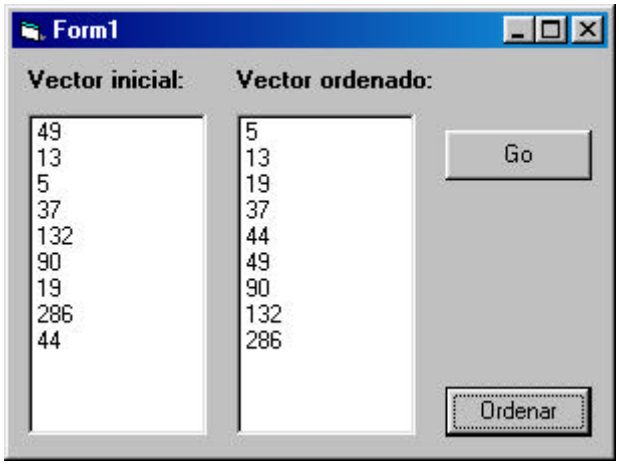
vec	49	13	5	37	132	90	19	286	44
índice	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***

Vector ordenado de forma ascendente:

vec	5	13	19	37	44	49	90	132	286
índice	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***

Exemplo1:

O programa que a seguir se apresenta faz a ordenação do vector de forma ascendente, tal como se mostrou na figura anterior. O programa encontra-se dividido em dois procedimentos, o primeiro associado ao botão de comando “Go” que serve para encher o vector e o segundo ao botão “Ordenar” que efectua a ordenação ascendente dos valores introduzidos e que ficaram guardados no vector **vec**.

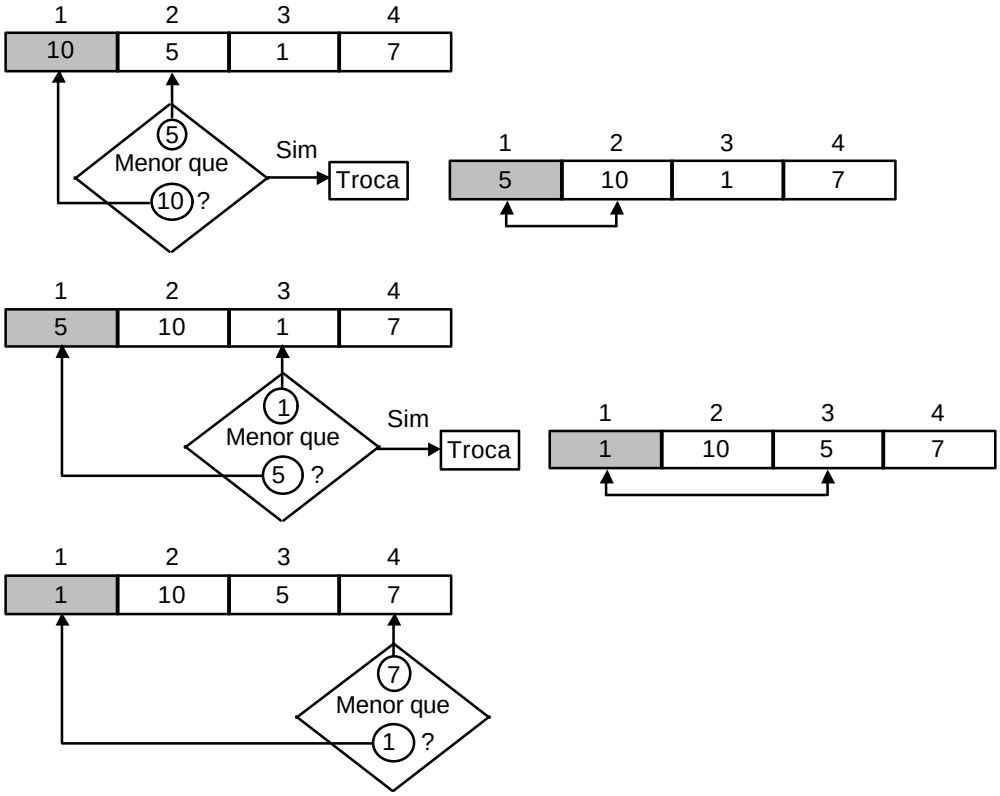



```
Const N As Integer = 9

Dim vec(N) As Integer


Private Sub cmdGo_Click()

    Dim i As Integer

    Dim texto As String

    lstVector.Clear

    lstVectorOrdenado.Clear

    For i = 0 To N - 1

        texto = "Vec(" + str(i) + "):"

        vec(i) = Val(InputBox(texto, "Entrada de Dados"))

        lstVector.AddItem vec(i)

    Next i

End Sub


Private Sub cmdOrdenar_Click()

    Dim i As Integer

    Dim j As Integer

    Dim temp As Integer

    For i = 0 To N - 2

        For j = i + 1 To N - 1

            If vec(j) < vec(i) Then

                temp = vec(i)

                vec(i) = vec(j)

                vec(j) = temp

            End If

        Next j

    Next i

    For i = 0 To N - 1

        lstVectorOrdenado.AddItem vec(i)

    Next

End Sub
```

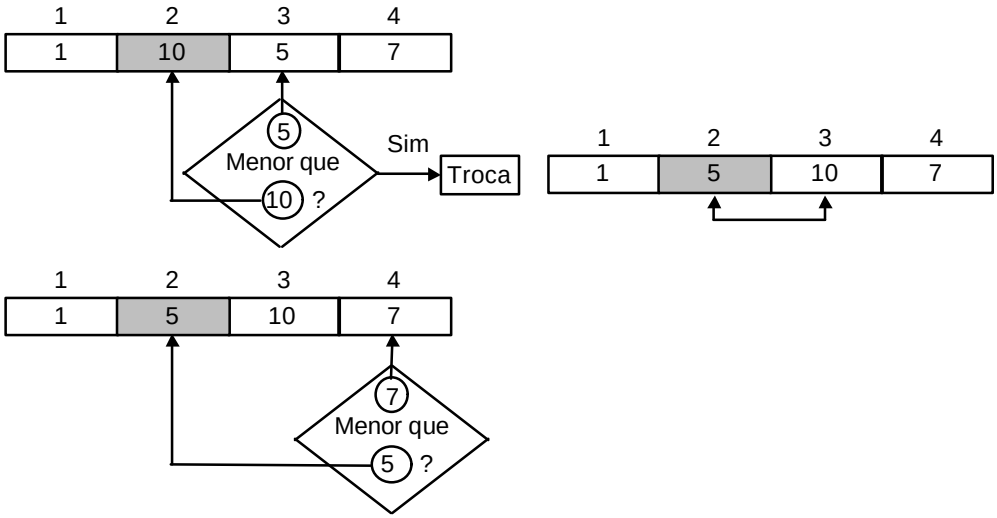
Exemplo 2:

Ordenação do vector de forma crescente

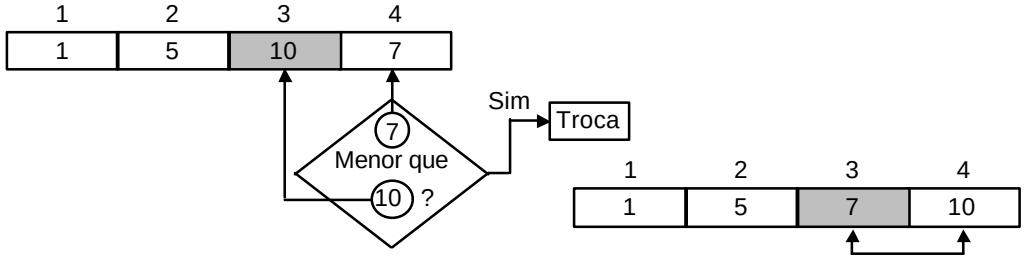
Passo 1: ordenar 1º elemento



Passo 2: ordenar 2º elemento

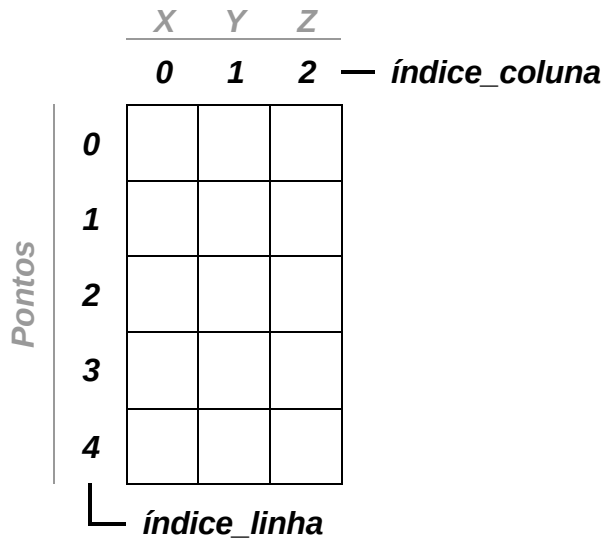


Passo 3: ordenar 3º elemento



Matrizes – Arrays bidimensionais

Por vezes torna-se necessário guardar informação que de alguma forma se encontra relacionada entre si. Por exemplo, quando estamos a trabalhar com sistemas de coordenadas a três dimensões, cada ponto é referenciado através das suas coordenadas **X**, **Y** e **Z**. Se pretendêssemos guardar 10 pontos desse sistema de coordenadas poderíamos usar um array bidimensional, tal como se mostra a seguir.



Exemplo de uma **matriz 5 x 3** (5 linhas e 3 colunas)

Os elementos de uma matriz são identificados utilizando o nome da matriz seguido dos índices, linha e coluna, entre parêntesis:

nome_matriz (índice_linha, índice_coluna)

Exemplos:

```
Coord(2, 0)     ' coordenada X do 3º ponto

notas(1, 3)     ' elemento que se encontra na 2ª linha e 4ª coluna da matriz 'notas'

nomes(0, 0)     ' 1º elemento da matriz 'nomes' – encontra-se na 1ª linha e 1ª coluna
```

Declaração de matrizes

Tal como na declaração de vectores, a declaração de matrizes pode ser feita indicando apenas o número de elementos de cada dimensão (número de linhas e colunas), ou usando a palavra reservada To, permitindo especificar o menor e o maior valor que os índices de cada dimensão poderão assumir.

Exemplos:

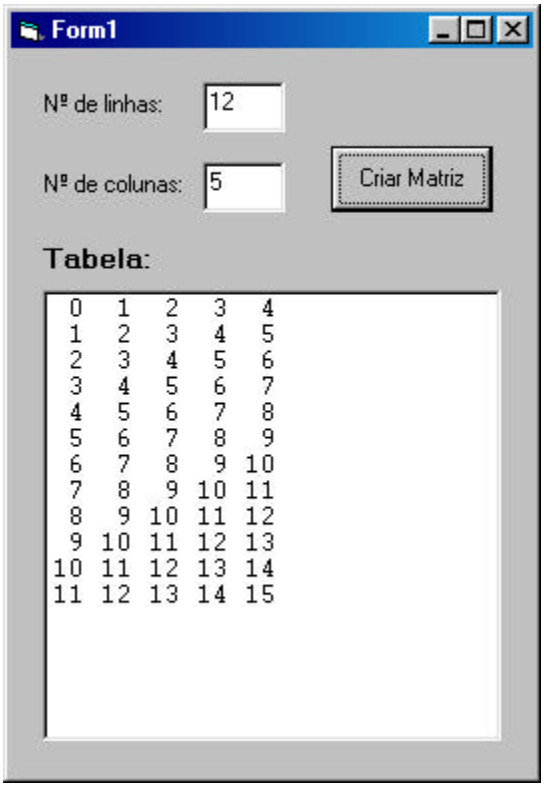
```
Dim Coord(5, 3) As Single

Dim Coord(1 To 5, 1 To 3) As Single
```

Processamento de matrizes

Nas matrizes, tal como nos vectores, a forma mais adequada de executar uma mesma acção sobre parte ou a totalidade dos seus elementos é utilizando duas estruturas de controlo repetitivo: uma para processar as linhas e outra para as colunas da matriz.

Exemplo:

```
Private Sub cmdCriarMatriz_Click()
    Dim Matriz(100, 100) As Integer
    Dim Linhas As Integer
    Dim Colunas As Integer
    Dim i As Integer
    Dim j As Integer
    Dim Texto As String
    lstTabela.Clear
    Linhas = Val(txtLinhas.Text)
    Colunas = Val(txtColunas.Text)
    For i = 0 To Linhas - 1
        Texto = ""
        For j = 0 To Colunas - 1
            Matriz(i, j) = i + j
            Texto = Texto & Format(Matriz(i, j), "@@  ")
        Next j
        lstTabela.AddItem Texto
    Next i
End Sub
```



Este programa apresenta uma matriz em que cada elemento é achado somando os índices que o identificam. As dimensões da matriz a ser criada são especificadas pelo utilizador e os valores calculados são armazenados numa matriz para eventual futura utilização.

De notar que neste programa se usaram duas variáveis: **i** e **j**, para controlar os ciclos de repetição e armazenar os valores achados na respectiva matriz. A variável **i** percorre todas as linhas e para cada linha é usada a variável **j** para percorrer as colunas.

Vectores de controlos

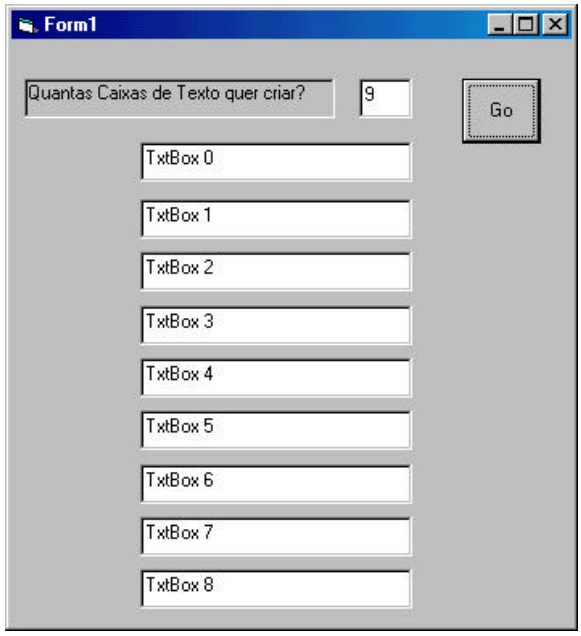
É possível criar **Arrays de Controlos**, isto é, conjuntos de objectos de controlo do mesmo tipo, agrupados sob o mesmo nome e identificáveis através do valor de um índice (contido na propriedade **Index** de cada elemento do vector de controlos). Usando estes vectores, torna-se possível dar a um programa a capacidade de criar de forma dinâmica um dado conjunto de objectos do mesmo tipo, sem ter que especificar previamente o seu número, ou de modificar em bloco propriedades desses controlos durante a execução do programa.

Os elementos desses Arrays de Controlo podem ser introduzidos manualmente, por exemplo, duplicando um controlo preexistente, mediante a sequência de cópia e inserção (*Ctrl-C + Ctrl-V*), ou usando a instrução **Load** durante a execução do programa.

No exemplo seguinte, o objecto *txtBox* original foi criado manualmente e foi dado o valor de zero à sua propriedade *Index*. O programa encarregou-se de criar os restantes elementos do vector de controlos, no número especificado pelo utilizador.

Exemplo:

```
Private Sub cmdGo_Click()

    Dim n As Integer

    Dim i As Integer

    n = Val(txtInput.Text)

    txtBox(0).Visible = True

    txtBox(0).Text = "TxtBox 0"

    For i = 1 To n - 1

        Load txtBox(i)

        txtBox(i).Top = 1000 + i * 500

        txtBox(i).Visible = True

        txtBox(i).Text = "TxtBox" & Str(i)

    Next i

End Sub
```



7 - Sub-rotinas

Procedimentos

No Visual Basic existem procedimentos que são executados em resposta às acções do utilizador, como por exemplo quando o utilizador prime um botão (*Click*). Esses procedimentos são conhecidos por ***"event procedures"*** - procedimentos de evento.

No entanto, existem outros procedimentos que não estão directamente associados a qualquer evento e que para serem executados têm que ser explicitamente invocados. Estes são procedimentos mais gerais - ***"general procedures"***.

A principal razão para se usar este tipo de procedimentos tem a ver com o facto de, por vezes, diferentes procedimentos de evento necessitarem de executar o mesmo conjunto de acções (instruções). Nestas situações o ideal é colocar esse conjunto de acções num procedimento para evitar a duplicação de código e tornar a aplicação mais fácil de manter.

A sintaxe para declarar um procedimento é a seguinte:

```
Sub NomeProcedimento (ListaArgumentos)

    ' Instruções

End Sub
```

A lista de argumentos é o conjunto de variáveis às quais são passados valores no momento da chamada do procedimento, valores esses que serão usados pelo procedimento durante as suas operações. Para cada argumento deve ser indicado o seu tipo e no caso de haver mais do que um argumento devem ser separados por vírgulas.

De cada vez que o procedimento é chamado, as instruções entre **Sub** e **End Sub** são executadas.

Passagem de argumentos

Nos procedimentos, os argumentos podem ser passados de duas maneiras: **por referência** ou **por valor**.

Quando uma variável é passada **por referência** quaisquer alterações feitas ao conteúdo dessa variável dentro do procedimento far-se-ão reflectir na variável usada na chamada ao procedimento.

Quando não se pretende que uma determinada variável seja devolvida alterada esta terá de ser passada ao procedimento por **valor**. Para isso é necessário usar a palavra **ByVal** na lista de argumentos quando o procedimento for declarado.

Exemplos:

1) Private Sub Command1_Click()

```
    Dim Custo As Integer
    Dim Desconto As Single
    Custo = 1000
    Desconto = 0.2
    CalcularDesconto Custo, Desconto
    Print Custo
End Sub
Sub CalcularDescontos(C As Integer, D As Single)
    C = C * (1 - D)
    Print C
End Sub
```

Produz o resultado: **800**
800

2) Private Sub Command1_Click()

```
    Dim Custo As Integer
    Dim Desconto As Single
    Custo = 1000
    Desconto = 0.2
    CalcularDesconto Custo, Desconto
    Print Custo
End Sub
Sub CalcularDescontos(ByVal C As Integer, D As Single)
    C = C * (1 - D)
    Print C
End Sub
```

Produz o resultado: **800**
1000

Funções

O Visual Basic tal como qualquer outra linguagem de programação inclui um vasto conjunto de funções predefinidas. Para além destas funções o VB permite definir outras funções usando para o efeito a sintaxe seguinte:

```
Function NomeFunção (ListaArgumentos) As Tipo
    ' Instruções
End Function
```

As funções distinguem-se dos procedimentos porque devolvem um valor. Sendo assim, e tal como nas variáveis, as funções também têm um tipo de dados que será o tipo do valor a retornar.

O valor, normalmente resultante de algumas operações e/ou cálculos, é devolvido por atribuição a uma "variável" com o mesmo nome da função.

Exemplo:

```
Function Par(Valor As Integer) As Boolean
    If Valor Mod 2 = 0 Then
        Par = True
    Else
        Par = False
    End If
End Function
```

8 - Funções predefinidas

Funções matemáticas

Abs() Retorna o **valor absoluto** de um número.

```
Num1 = Abs(13.5)

Num2 = Abs(-13.5)

‘ Num1 e Num2 contêm o mesmo valor, 13.5
```

Exp() Retorna um valor do tipo Double representando a **potência de base *e*** elevada ao expoente passado como parâmetro.

```
Num = Exp(1)

‘ Num contém o valor 2.71828 (e)
```

Log() Retorna um valor do tipo Double representando o **logaritmo natural** do número passado como parâmetro.

Sqr() Retorna um valor do tipo Double representando a **raiz quadrada** do valor passado como parâmetro.

```
Num = Sqr(25)

‘ Num contém o valor 5
```

Int() Retorna a parte inteira de um dado número real; Se o número for negativo, Int retorna o primeiro negativo inteiro menor ou igual ao número passado por parâmetro.

```
Num = Int(12.565)

‘ Num contém o valor 12
```

Fix() Retorna a parte inteira de um dado número real; Se o número for negativo, Int retorna o primeiro negativo inteiro maior ou igual ao número passado por parâmetro.

```
Num1 = Int(-9.5)

Num2 = Fix(-9.5)

‘ Num1 contém o valor –10 enquanto Num2 contém -9.
```

Cos() Retorna um valor do tipo Double representando o **coseno** do número passado como parâmetro.

Sin() Retorna um valor do tipo Double representando o **coseno** do número passado como parâmetro.

Tan() Retorna um valor do tipo Double representando o **coseno** do número passado como parâmetro.

Funções de manipulação de strings

Val() Retorna como valor numérico um número contido dentro duma string.

```
Num1 = Val(“123”)

Num2 = Val(“12  3”)

Num3 = Val(“12 e 3”)

‘ Num1 e Num2 contêm o mesmo valor, 123

‘ Num3 contém o valor 12
```

Str() Retorna uma string representando um número.

```
Num1 = Str(123)

Num2 = Str(-123)

‘ Num1 contém a string “ 123” e Num2 contém a string “-123”
```

Len()

Retorna um valor do tipo Long representando o número de caracteres contido numa string.

```
Nome = "ISEP"

Comp = Len(Nome)

' Comp contém o valor 4
```

InStr()

Retorna um valor do tipo Long representando a posição da primeira ocorrência de uma string dentro de outra.

Sintaxe:

InStr (InícioPesquisa, String1, String2, Opção **)**

Em que:

InícioPesquisa é a posição na *String1* em que se inicia a pesquisa.

String1 é a string onde a pesquisa se efectuará.

String2 é a string a ser procurada em *String1*.

Opção é um valor numérico que especifica se a busca é sensível à distinção entre maiúsculas e minúsculas: **0** - faz distinção entre maiúsculas e minúsculas (valor por defeito); **1** - não faz distinção.

Nota: *InícioPesquisa* e *Opção* são parâmetros opcionais, no entanto, *InícioPesquisa* é obrigatório se *Opção* tiver sido especificado.

```
Texto = "Visual Basic 6.0"

StringAProcurar = "basic"

Pos1 = InStr(1, Texto, StringAProcurar, 1)

Pos2 = InStr(Texto, StringAProcurar)

' Pos1 contém o valor 8 (foi encontrada na posição 8)

' Pos2 contém o valor 0 (não foi encontrada)
```

UCase() Converte os caracteres de uma string para **maiúsculas**.

```
Texto = UCase("Visual Basic 6.0")
' Texto contém a string "VISUAL BASIC 6.0"
```

LCase() Converte os caracteres de uma string para **minúsculas**.

```
Texto = LCase("Visual Basic 6.0")
' Texto contém a string "visual basic 6.0"
```

Left() Retorna uma string com um número especificado de caracteres mais à esquerda da string passada como parâmetro.

```
Texto1 = "Visual Basic 6.0"
Texto2 = Left(Texto1, 6)
' Texto2 contém a string "Visual"
```

Right() Retorna uma string com um número especificado de caracteres mais à direita da string passada como parâmetro.

```
Texto1 = "Visual Basic 6.0"
Texto2 = Right(Texto1, 9)
' Texto2 contém a string "Basic 6.0"
```

Mid() Retorna uma string com um número especificado de caracteres da string passada como parâmetro.

Sintaxe:

Mid (String, Início, Comprimento **)**

Em que:

String representa a string a ser tratada.

Início é a posição do primeiro caracter a ser retornado.

Comprimento é o número de caracteres a retornar.

Nota: *Comprimento* é um parâmetro opcional. Se não for especificado são retornados todos os caracteres desde a posição *Início* até ao fim da string.

```
Texto1 = "Visual Basic 6.0"
Texto2 = Mid(Texto1, 8, 5)
' Texto2 contém a string "Basic"
```

LTrim() Remove todos os espaços existentes no início da string.

```
Texto = LTrim("          Visual Basic 6.0          ")
' Texto contém a string "Visual Basic 6.0          "
```

RTrim() Remove todos os espaços existentes no fim da string.

```
Texto = RTrim("          Visual Basic 6.0          ")
' Texto contém a string "          Visual Basic 6.0"
```

Trim() Remove todos os espaços existentes quer no início quer no fim da string.

```
Texto = Trim("          Visual Basic 6.0          ")
' Texto contém a string "Visual Basic 6.0"
```

Chr() Retorna uma string com o caracter associado ao código passado como parâmetro.

```
Texto = Chr(65)
' Texto contém a string "A"
```

Asc() Retorna um Integer representando o código do primeiro caracter da string passada como parâmetro.

```
Num1 = Asc("A")

Num2 = Asc("Amarelo")

' Num1 e Num2 contêm o valor, 65
```

Outras funções standard

IsNumeric() Retorna um valor booleano traduzindo o facto de uma dada expressão ter ou não um resultado numérico.

```
Verif1 = IsNumeric("4475")

Verif2 = IsNumeric("4479 Codex")

' Verif1 contém o valor True

' Verif2 contém o valor False
```

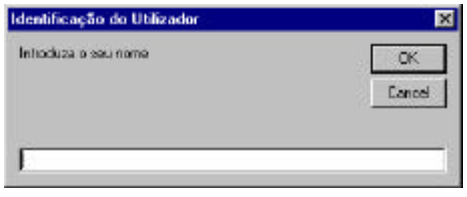
Date Retorna um valor representando a data do sistema.

Time Retorna um valor representando a hora do sistema.

Timer Retorna um valor do tipo Single representando o número de segundos que já passaram desde a meia-noite.

Função InputBox

Esta função permite apresentar uma mensagem ao utilizador, recolhendo ao mesmo tempo uma string contendo a sua resposta. Assim, ao contrário da função **MsgBox** (ver a seguir), esta função produz um resultado do tipo String e não do tipo Integer.



Exemplo de uma **Input Box**

Esta função cria um objecto composto, uma **Input Box**, que consiste numa caixa de texto e num *label* dentro de uma pequena janela.

A sintaxe da função InputBox é a seguinte:

Variável = **InputBox (** Mensagem, Título, String_de_Defeito, xpos, ypos **)**

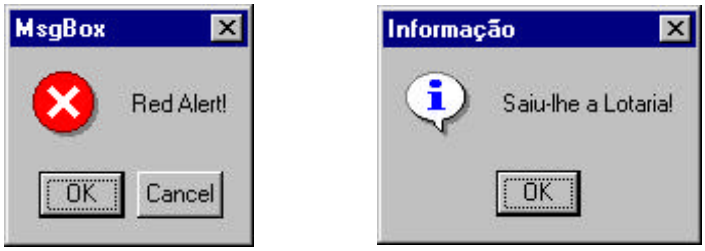
...em que:

Mensagem	É o texto da mensagem a mostrar na janela.
Título	É o texto a apresentar na barra de títulos da janela.
String_de_Defeito	Este parâmetro é opcional e representa a string que aparece por defeito na caixa de texto da Input Box.
xpos e *ypos*	São parâmetros opcionais que indicam a localização horizontal e vertical, respectivamente, da Input Box no ecrã. Se não forem indicados a janela aparece no centro do ecrã.

Função MsgBox

A função **MsgBox** permite criar caixas de diálogo temporário vulgarmente designadas por **Caixas de Mensagem**, contendo um pedaço de texto, um ou mais botões de comando e, opcionalmente, um ícone ilustrativo da finalidade da caixa de mensagem.

Como o próprio nome indica, a utilidade destas caixas de mensagem consiste em permitir que o programa transmita alguma informação ao utilizador de forma sucinta.



Exemplos de **Message Boxes**

A sintaxe da função MsgBox é a seguinte:

Variável = **MsgBox (** Mensagem, Características, Título **)**

...em que:

Mensagem	É o texto da mensagem a mostrar na janela.
Características	Valor numérico que especifica quais os botões, qual o ícone e qual o botão por defeito (botão com *focus*) devem aparecer na janela.
Título	É o texto a apresentar na barra de títulos da janela.

O valor para o parâmetro *Características* obtém-se pela soma de cada um dos seguintes componentes:

Botões	**Ícone**	**Botão com *focus***
0 OK	**0** Nenhum	**0** 1º Botão
1 OK, Cancel	**16** Mensagem Crítica 	**256** 2º Botão
2 Abort, Retry, Ignore	**32** Pergunta 	**512** 3º Botão
3 Yes, No, Cancel	**48** Aviso 	
4 Yes, No	**64** Informação 	
5 Retry, Cancel		

Exemplo:

Para obter as caixas de mensagem apresentadas atrás, os valores a utilizar para o parâmetro *Características* seriam os seguintes:

- Caixa de mensagem da esquerda: 1 + 16 + 0 = 17
- Caixa de mensagem da direita: 0 + 64 + 0 = 64

A função MsgBox devolve um valor que será guardado em *Variável*, indicando qual o botão premido, segundo a tabela:

Valor	**Botão Premido**
1	OK
2	Cancel
3	Abort
4	Retry
5	Ignore
6	Yes
7	No

Nota: O valor **2** será devolvido caso o botão *Cancel* seja premido ou caso a tecla *Escape* seja carregada

Formatação da saída de dados - Função Format

A função **Format** permite formatar o resultado de uma expressão de acordo com as instruções contidas numa "string de formatação", de modo a poder ser convenientemente visualizadas.

A sintaxe desta função é a seguinte:

Format (Expressão, String_De_Formatação **)**

Formatação de valores numéricos

A *string de formatação* pode ser constituída por símbolos ou nomes de formatos padrão. Os formatos padrão produzem saídas que são baseadas nas definições regionais do sistema operativo do utilizador.

Os símbolos mais utilizados são:

0 **Localizador de dígitos** – Permite mostrar um dígito ou zero. Caso o valor a afixar tenha um dígito na posição em que (0) aparece na *string de formatação*, esse dígito será mostrado; caso contrário, será mostrado um zero.

Se o número contiver menos dígitos (para um ou outro lado do separador decimal) do que o número de zeros na *string de formatação*, os zeros a que não correspondam dígitos serão mostrados. Se o número tiver mais casas decimais que zeros na *string de formatação* à direita do separador decimal, ocorrerá então um arredondamento. Quando tal se passa à esquerda do separador decimal, não ocorre qualquer modificação.

**Localizador de dígitos** – Permite mostrar um dígito ou não mostrar nada. Se o valor numérico a afixar tiver um dígito na posição em que (#) aparece na *string de formatação*, esse dígito será mostrado; caso contrário, nada será apresentado nessa posição.

Funciona como o caracter (0) exceptuando o facto de que zeros em excesso à esquerda e à direita não são mostrados caso o número tenha tantos ou menos dígitos que a *string de formatação*.

. **Separador decimal** (há a possibilidade de configurar o sistema para usar o caracter vírgula - "**,**")

, **Separador dos milhares** (há a possibilidade de configurar o sistema para usar o caracter ponto - "**.**")

% **Localizador de percentagem** – O resultado da expressão será multiplicado por 100, sendo o caracter (%) inserido na posição especificada na *string de formatação*.

Os formatos padrão são especificados através dos seus nomes sempre dentro de aspas. Os formatos padrão que podem ser utilizados são:

Nome do Formato	Descrição
General Number	Número sem separador dos milhares.
Fixed	Número com pelo menos um dígito àesquerda e dois dígitos à direita do ponto decimal.
Standard	Número com separador dos milhares, pelo menos um dígito à esquerda e dois dígitos àdireita do ponto decimal.
Percent	Multiplica o valor por 100 e acrescenta o sinal de percentagem.
Scientific	Usa notação científica padrão.
Currency	É utilizado para formato monetário. A formatação depende das definições regionais do sistema operativo.

Formatação de cadeias de caracteres (strings)

A *string de formatação* poderá incluir os seguintes caracteres:

@ **Localizador de caracteres** – Permite mostrar um caracter ou um espaço. Caso a string a tratar tenha um caracter na posição em que (@) aparece na *string de formatação*, esse caracter será mostrado; caso contrário, será mostrado um espaço.

& **Localizador de caracteres** – Permite mostrar um caracter ou não mostrar nada. Se a string a tratar tiver um caracter na posição em que (@) aparece na *string de formatação*, esse caracter será mostrado; caso contrário, nada será mostrado nessa posição.

< Passa todos os caracteres para minúsculas.

> Passa todos os caracteres para maiúsculas.

! Força o tratamento da string inicial da esquerda para a direita. Por defeito as strings são tratadas da direita para a esquerda.

Exemplos:

O que é pedido	O que é mostrado
AMinhaString = Format(3256.3, “###0.00”)	3256.30
AMinhaString = Format(3256.3, “fixed”)	3256.30
AMinhaString = Format(3256.3, “#,##0.00”)	3,256.30
AMinhaString = Format(3256.3, “standard”)	3,256.30
AMinhaString = Format(0.125, ”0.00”)	0.13
AMinhaString = Format(0.125, ”#.0000”)	.1250
AMinhaString = Format(14.312587, ”000.0”)	014.3
AMinhaString = Format(“Olá”, “>”)	OLÁ

Geração de números aleatórios - Funções Rnd e Randomize

Em Visual Basic é possível gerar números aleatórios mediante a utilização da função **Rnd** e da instrução **Randomize**.

A função **Rnd** utiliza uma sequência pseudo-aleatória para gerar números entre 0 e 1. Essa sequência deve ser previamente inicializada mediante a instrução **Randomize**. Esta instrução deve ser executada apenas uma vez, enquanto que a função Rnd pode ser chamada quantas vezes forem necessárias no decorrer dum programa.

Como a função **Rnd** produz números aleatórios entre 0 e 1, caso se pretenda obter valores dentro de um intervalo mais alargado, haverá que introduzir um factor de escala:

```
Rnd * (LarguraDoIntervalo) + LimiteInferior
```

A instrução **Randomize** deverá receber um parâmetro numérico que determinará o ponto de inicialização da sequência pseudo-aleatória. Para reforçar o caracter aleatório deste processo é conveniente utilizar valores imprevisíveis como, por exemplo, os resultantes da chamada da função **Timer** (que devolve o número de segundos passados desde a meia-noite):

```
Randomize Timer
```

Os números produzidos pela função **Rnd** são valores reais. Caso se pretenda obter valores inteiros, haverá que usar uma das seguintes técnicas:

- Atribuir o resultado da expressão contendo **Rnd** a uma variável do tipo Integer, provocando a truncagem do valor real produzido;
- Usar a função **Int** para converter o valor real obtido no seu equivalente inteiro, eliminando a parte decimal.

Exemplo:

```
Private Sub cmdDado_Click()
    Dim al As Single
    Randomize Timer
    al = Rnd * 6 + 1
    Label6.Caption = Int(al)
End Sub
```

9 - Controlos

Option Buttons – Botões de Opção

Estes objectos permitem ao utilizador efectuar escolhas entre diversas alternativas.

São compostos pelo botão propriamente dito (de forma circular) e um pequeno texto que o acompanha, especificado pela propriedade *Caption* do objecto.

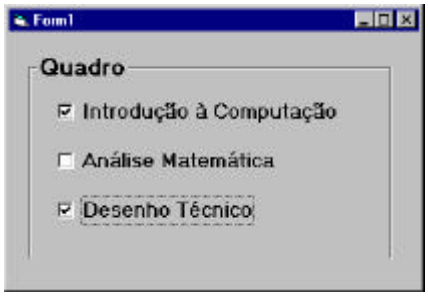
Objectos da classe **OptionButton** reunidos numa **frame**

Os botões de opção são geralmente agrupados em conjuntos de dois ou mais, estando relacionados entre si. São normalmente inseridos em "frames" (quadros), objectos que servem para agrupar controlos.

Para além da propriedade *Caption*, outra propriedade importante dos botões de opção é a propriedade *Value*, que pode assumir o valor "True" ou "False", conforme o botão se encontrar ou não seleccionado. Ao mesmo tempo, só é possível existir um botão seleccionado dentro do mesmo grupo de botões de opção.

Check Boxes – Caixas de Verificação

Estes objectos comportam-se de forma semelhante à dos botões de opção mas, neste caso, é possível encontrar vários controlos deste tipo activados simultaneamente, já que tais objectos funcionam de forma independente (isto é, não se encontram relacionados entre si).



Objectos da classe **CheckBox** reunidos numa **frame**

Possuem também uma propriedade *Value* que, neste caso, pode apresentar os seguintes valores:

0 - não activada

1 - activada

2 - não disponível

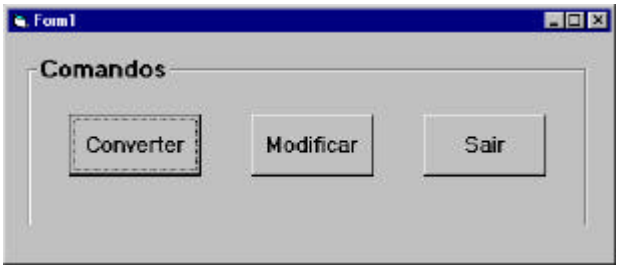
O texto a inserir junto de cada caixa de verificação deve ser especificado mediante a propriedade *Caption*.

Frames – Quadros

Tais objectos destinam-se a agrupar outros objectos (controlos). São usados muitas vezes para organizar um dado conjunto de botões de opção (*Option Buttons*), tornando-os independentes de outros botões de opção eventualmente existentes na mesma *form*. Sem as *Frames* isto não era possível porque se os *Option Buttons* fossem todos colocados sobre o *Form* então só teríamos um grupo de opções de exclusão mútua.

É importante criar o quadro antes de aí inserir os controlos. Se o controlo for criado antes do quadro, não é possível apenas deslocá-lo para dentro do quadro uma vez este criado.

Quando se insere um objecto dentro do quadro, esse quadro passa a constituir o "contentor" do objecto. Quer isto dizer que a sua localização passa a ser definida não em relação à *form* mas em relação ao quadro que o contem.



Uma **Frame** agrupando três botões de comando

Outra utilidade dos quadros é servir de "moldura" a um dado conjunto de controlos, de modo a melhorar a aparência e a organização da *form* em que estão inseridos, agrupando os diversos controlos de acordo com as suas funcionalidades.

List Boxes – Caixas de Listagem

Este controlo mostra uma lista de opções passíveis de escolha, indicando com uma barra escura a selecção actual. Por defeito, as opções são apresentadas numa única coluna vertical, podendo-se no entanto definir várias colunas. Se o número de opções exceder a quantidade de elementos que pode ser visto ao mesmo tempo na *List Box* surgem barras de deslocamento verticais e/ou horizontais.

Uma **List Box** com várias opções

A introdução dos elementos na lista pode ser feita enquanto se desenham os controlos na *form*, usando para o efeito a propriedade **List**, ou durante a execução do programa.

Os elementos da lista estão guardados numa estrutura de dados acessível através da propriedade **List(n)** que simula um array de strings. Assim, o primeiro elemento encontra-se em **List(0)**, o segundo elemento em **List(1)** e assim sucessivamente.

Inserir um elemento na lista

A inserção de elementos na lista durante a execução do programa deve ser feita usando o método **AddItem**, cuja sintaxe é:

listbox**.AddItem** elemento, posição

O parâmetro *elemento* representa a nova opção (novo elemento) a acrescentar àlista.

O parâmetro *posição* é opcional. Se for especificado, o elemento é inserido no local indicado, senão é inserido no fim da lista. A primeira posição da lista é a posição 0.

Remover um elemento da lista

O método usado na eliminação de um item da lista é o **RemoveItem** e a sintaxe é a seguinte:

listbox**.RemoveItem** posição

Aqui, *posição* representa o índice do elemento a eliminar.

A posição do elemento seleccionado é dada pela propriedade **ListIndex** que apenas se encontra disponível durante a execução do programa. O valor desta propriedade é **0** se estiver seleccionado o primeiro elemento da lista, **1** se o elemento seleccionado for o seguinte, e assim por diante. **ListIndex** tem o valor **-1** se nenhum elemento está seleccionado.

Para remover todos os elementos da lista pode-se fazer: listbox**.Clear**

Para além das já referidas, as propriedades mais usadas são as seguintes:

Propriedade	Descrição
Text	Permite obter o elemento actualmente seleccionado.
ListCount	Permite conhecer a qualquer momento o número de elementos contidos na lista.
Sorted	Permite especificar se os elementos da lista devem aparecer por ordem alfabética.
Columns	Permite indicar se a lista de elementos deve aparecer numa única coluna (*Columns = 0*) ou em duas (*Columns = 2*), três (*Columns = 3*), ... colunas.
MultiSelect	Permite controlar a forma de selecção de elementos dentro da lista. Pode ter um dos seguintes valores: **0 -** só é possível seleccionar um elemento **1 -** é possível seleccionar vários elementos simultaneamente, pressionando cada elemento **2 -** é possível seleccionar vários elementos simultaneamente, usando as tecla *Shift* e *Control*
Selected	Se a propriedade *MultiSelect* tiver o valor 1 ou o valor 2, a selecção múltipla está autorizada. Nesse caso, as propriedades *Text* e *ListIndex* referem-se apenas ao último elemento seleccionado. Assim, para conhecer quais os elementos seleccionados haverá que utilizar a propriedade *Selected* da forma: **Selected (*índice*)** avaliando para cada elemento (*índice*) o valor da propriedade *Selected*. Esta propriedade apresenta o valor *True* caso o elemento esteja seleccionado e *False* caso contrário.
SelCount	Permite conhecer a qualquer momento o número de elementos seleccionados. Esta propriedade é particularmente útil nas situações em que a *listbox* permite múltiplas selecções.