Curso de Programação em Delphi

O Curso de Delphi em questão é uma iniciativa do PET – Programa de Educação Tutorial, e do Ramo Estudantil do IEEE – UFJF.

Esta apostila foi desenvolvida com o intuito de orientar os estudantes durante a realização do curso de Delphi. Trata-se de uma compilação de diversos materiais disponíveis na internet, sem direitos de copyright, com as modificações necessárias para se adaptar ao escopo do curso, além da inclusão de diversos tópicos considerados importantes pelos autores.

Sugestões e correções serão bem-vindas.

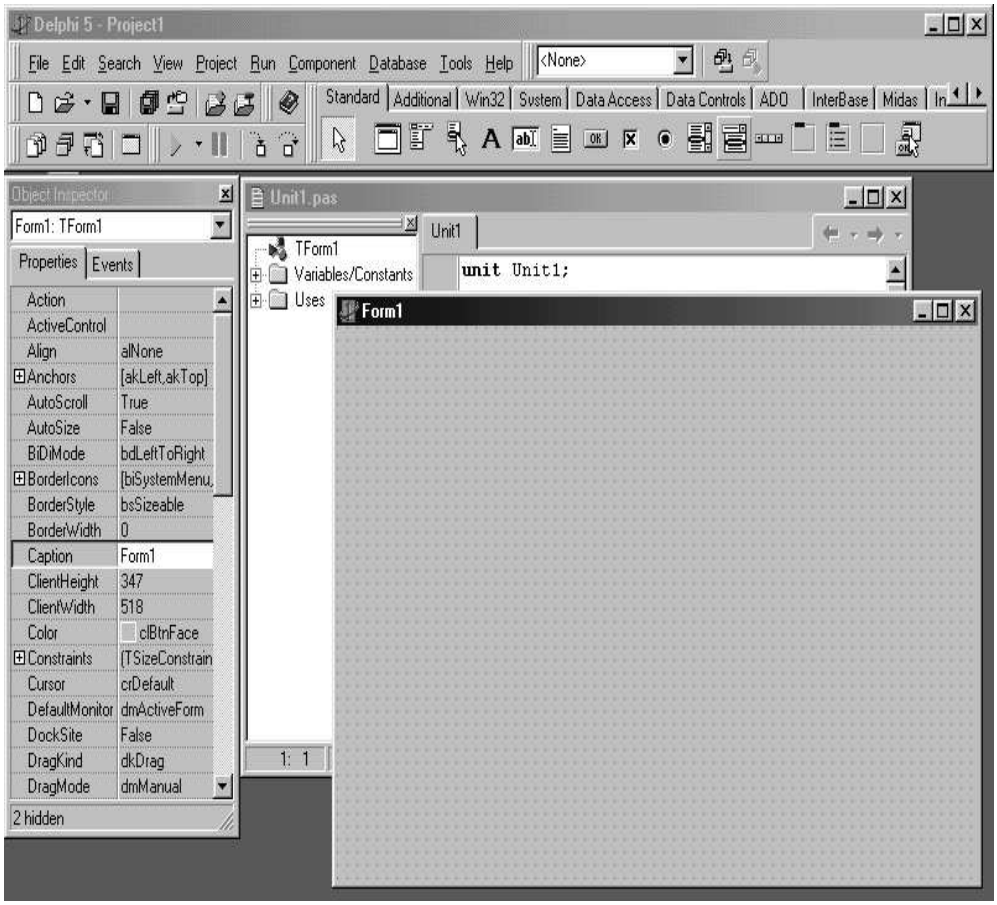
Atenciosamente,

Murilo Pereira Soares
André Pedretti
Thiago Corrêa César

Índice

1. O Ambiente Integrado de Desenvolvimento do Delphi

O ambiente de desenvolvimento do Delphi é composto de várias 'partes' compondo um conjunto integrado de 'janelas' que interagem entre si.



Vamos abordar cada uma separadamente:

➢ O Form Editor

Form é o termo utilizado para representar as janelas do Windows que compõem uma aplicação. Os forms servem como base para o posicionamento dos componentes, que são responsáveis pela interação entre usuário e máquina.

Para *selecionarmos* o form devemos clicar (uma vez) em sua área interna ou na object inspector, e não simplesmente em seu título. As características iniciais do form como tamanho, botões (minimizar, maximizar, fechar, controle) e ícone podem (e serão) ser modificadas através de recursos que veremos adiante.

➢ A Barra de Menus

Como todo programa padrão Windows, há uma janela onde estão situados os *menus* da aplicação. No entanto, no Delphi a barra que contem os menus também agrupa outras

partes.

➢ A Paleta de Componentes

Aplicativos orientados a objetos trabalham com elementos que denominamos *componentes*. No Delphi, os componentes encontram-se em uma paleta com várias *guias*.

Pode-se configurar a ordenação das *guias* clicando com o **botão direito** do mouse sobre qualquer componente e clicar na opção **Properties**.

Há basicamente três maneiras de **inserirmos os componentes** no formulário:

- Clicar uma vez no componente, e clicar dentro do formulário (*não* arrastar *para* o form).
- Clicar duas vezes rapidamente no componente desejado.
- Segurar a tecla *Shift* e clicar no componente desejado; clicar no form várias vezes.

Na terceira opção, o componente será '*travado*' ao mouse. Para '*destravá-lo*' clique no ícone da *seta*, o primeiro ícone da paleta.

➢ A Speed Bar

A speedbar está posicionada ao lado esquerdo da barra principal do Delphi. Possui diversos botões (ícones) que representam comandos muito utilizados durante o desenvolvimento.

Pode-se customizar a speedbar adicionando ou retirando algum botão através do botão direito em qualquer ícone (da speedbar) e escolher o comando **customize**. Na janela aberta, seleciona-se a guia **Commands**. Neste momento pode-se arrastar nos dois sentidos, para adicionar ou retirar botões.

➢ Object Inspector

Uma das 'partes' mais importantes da orientação a objeto é a possibilidade de definir características personalizadas aos componentes. No Delphi, utilizamos a janela object

inspector para realizar esta tarefa. Há uma *caixa de listagem* que permite a escolha de qual componente deverá ser selecionado. Existem também duas *guias*:

Properties – Define as propriedades e valores do Objeto selecionado.

Events – Define quais os eventos serão manipulados pelo desenvolvedor. Algumas *propriedades* trazem opções diferenciadas para alteração. Por exemplo:

Caption – Permite a inserção de uma string de caracteres.

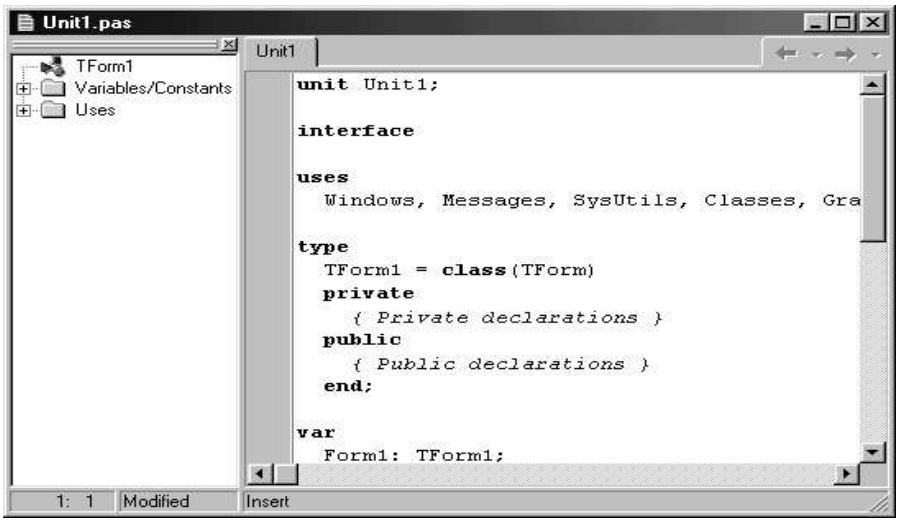
Color – Permite a inserção de um dos valores *pré-definidos* na caixa de listagem.

BorderIcons – Toda propriedade que possui o sinal de + tem a característica de mostrar *subpropriedades*. Deve-se clicar no sinal de + para expandir e no sinal de – para ocultar.

Icon – Exibe um botão de reticências (...) que dará origem a uma caixa de diálogo.

➢ Code Editor

O editor de código é responsável por receber todas as declarações criadas pelo Delphi e *handlers*[1] criados pelo desenvolvedor. É no ambiente Code Editor que implementamos o algoritmo na linguagem Object Pascal.



Na janela do editor *pode* haver uma outra janela denominada ***Code Explorer***. É a parte esquerda da janela, onde podemos ter uma orientação sobre os objetos, procedimentos, funções e classes utilizadas na aplicação. Para desligar o *code explorer* clique no pequeno X ao lado da guia do *code editor*, para visualiza-lo clique com o botão direito dentro do editor e escolha **View Explorer** ou pelo teclado Ctrl+Shift+E. Pode-se personalizar o Editor através do menu **Tools | Editor Options**.

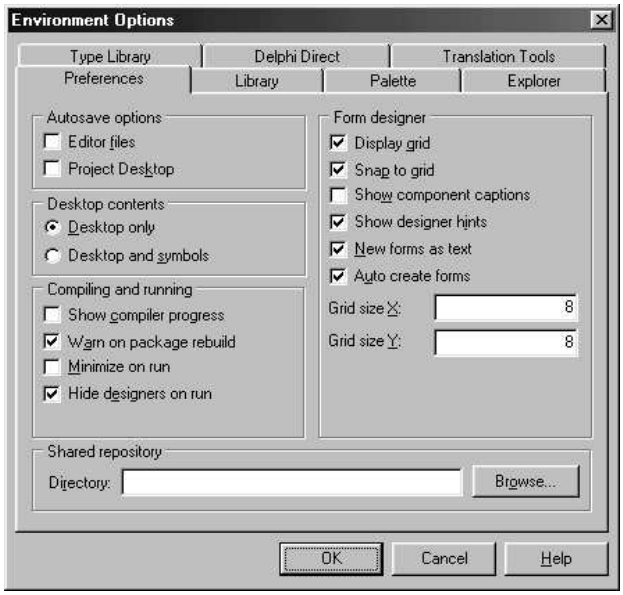
[1] Manipulador de eventos.

➢ DESKTOP TOOLBAR

Este novo recurso permite gravar vários layouts[2] de tela personalizando seu ambiente de trabalho. Estas opções (layouts) podem ser manipuladas pelos ícones ou pelo menu: **View – Desktops.**

➢ CONFIGURAÇÃO DO AMBIENTE

O Delphi permite que você personalize o ambiente através do menu **Tools | Environment Options**.



Algumas opções da janela *Environment Options* que a princípio, podemos julgar importantes:

✗ *AUTOSAVE OPTIONS*

Editor files – Grava os arquivos fonte (.PAS) no momento da compilação, evitando perda de código em caso de travamento da máquina. Porém, não permite compilar um determinado projeto sem salva-lo antes.

Project Desktop - Grava a posição das janelas do projeto atual.

✗ *COMPILING AND RUNNING*

Minimize on run – Para minimizar o Delphi no momento da compilação em efeito de testes. Evita confusões de janelas.

[2] Disposições das janelas no monitor.

✗ *FORM DESIGNER*

New forms as text –Para tornar ***compatível*** os arquivos *de definição de formulário (.DFM)* criados no ***Delphi5*** para o ***Delphi4***, desligue esta opção.

➢ TECLAS IMPORTANTES DO IDE

Tecla	Função
F12	Alterna entre o *code editor* e o *form designer.*
F11	Alterna entre o *code editor*, *form designer* e a *oject inspector.*
F10	Torna o foco para a janela *principal.*
F9	(RUN) Permite *compilar* e *executar* o projeto para testes. Este processo gera *automaticamente* o arquivo .EXE no diretório onde foi gravado o arquivo de projeto (.DPR).
CTRL + F9	Permite *compilar* o projeto *sem* executar. Ideal para conferência de código.
SHIFT + F12	Permite alternar entre os formulários do projeto. Equivalente ao ícone **View Form** na SpeedBar.
CTRL + F2	Permite 'destravar' o Delphi em caso de testes onde ocorram *exceções*, como veremos mais adiante.

2. Fundamentos de Projeto em Delphi

O conceito de projeto em Delphi é baseado em um conjunto de arquivos necessários para gerar uma aplicação. Vamos destacar os principais arquivos:

Extensão	Tipo e descrição	Criação	Necessário para compilar?
.PAS	Arquivo Pascal: o código-fonte de uma unidade Pascal, ou uma unidade relacionada a um formulário ou uma unidade independente.	Desenvolvimento	Sim.
.DPR	Arquivo Delphi Project. (Contém código-fonte em Pascal.)	Desenvolvimento	Sim.
.DFM	Delphi Form File: um arquivo binário (na versão 5 pode ser convertido para texto) com a descrição das propriedades de um formulário e dos componentes que ele contém.	Desenvolvimento	Sim. Todo formulário é armazenado em um arquivo PAS e em um arquivo DFM.
.DCU	Delphi Compiled Unit: o resultado da compilação de um arquivo Pascal.	Compilação	Apenas se o código-fonte não estiver disponível. Os arquivos DCU para as unidades que você escreve são um passo intermediário; portanto, eles tornam a compilação mais rápida.
.BMP, .ICO, .CUR	Arquivos de bitmap, ícone e cursor: arquivos padrão do Windows usados para armazenar imagens de bitmap.	Desenvolvimento: Image Editor	Normalmente não, mas eles podem ser necessários em tempo de execução e para edição adicional.
.CFG	Arquivo de configuração com opções de projeto. Semelhante aos arquivos DOF.	Desenvolvimento	Necessário apenas se opções de *compilação* especiais foram configuradas.
.DOF	Delphi Option File: um arquivo de texto com as configurações atuais para as opções de projeto.	Desenvolvimento	Exigido apenas se opções de *compilação* especiais foram configuradas.
.DSK	Arquivo de Desktop: contém informações sobre a posição das janelas do Delphi, os arquivos abertos no editor e outros ajustes da área de trabalho.	Desenvolvimento	Não. Você deve excluí-lo se copiar o projeto em um novo diretório.
.EXE	Aquivo executável: o aplicativo Windows que você produziu.	Compilação: Ligação (linking)	Não. Esse é o arquivo que você vai distribuir. Ele inclui todas as unidades compiladas, formulários e recursos.

.~PA	*Backup* do arquivo Pascal Pode ser ativado ou desativado através do Menu **Tools – Editor Options** - *guia display* – Item: Create backup file.	Desenvolvimento	Não. Esse arquivo é gerado automaticamente pelo Delphi, quando você salva uma nova versão do código-fonte.
.TODO	Arquivo da lista **to-do**, contendo os itens relacionados ao projeto inteiro.	Desenvolvimento	Não. Esse arquivo contém notas para os programadores.

O Delphi possui um mecanismo de *gerência de arquivos de projeto* informando os *principais* arquivos e seu *path*. Clique em **View – Project Manager**

A figura acima é um exemplo de um projeto inicial, ainda não salvo. O diretório padrão para criação dos arquivos é *projects*, obviamente devemos definir na gravação pasta e nomes de arquivos mais específicos.

➢ Arquivos .PAS e .DPR

Para visualizarmos o código fonte da unidade (.PAS) em Delphi basta selecionarmos o *code editor* (F12). Para visualizarmos o código fonte no arquivo de projeto (.DPR) basta selecionarmos o menu **Project – View Source**. O arquivo de projeto é exibido em uma *guia* no *code editor*. Para fechar a guia basta clicar com o botão direito e escolher *close page*.

✗ *Salvar Projeto*

Como vimos anteriormente, o conceito de projeto em Delphi se faz através de um conjunto de arquivos. No Menu **File** do Delphi temos quatro opções para a gravação do projeto:

Comando	Objetivo
Save	Salvar apenas a unidade selecionada
Save As...	Salvar a unidade selecionada como... pode-se renomear ou trocar de pasta (duplicando) o arquivo.
Save Project As...	Salvar o projeto como... pode-se renomear ou trocar de pasta (duplicando) o arquivo.
Save All	Grava todos os arquivos do projeto, e atualiza-os caso já sejam salvos.

Ao clicar em Salve All abre-se uma caixa de diálogo padrão do Windows onde deve ser preenchido o nome do arquivo e escolhida uma pasta para armazenar o projeto. Observe o título da janela, pois após a gravação do 'arquivo da unidade', será exibida a mesma caixa

(com título diferente) para a gravação do 'arquivo de projeto'.

➢ Opções de Projeto

O Delphi permite a configuração de vários itens do sistema através do menu **Project – Options**.

Forms

- Main form - Nesta guia permite a escolha do formulário *principal* da aplicação.
- Available form - Os formulários *available* (disponíveis) em caso de criação em tempo de execução.

Application

- Title - Define um nome para a sua aplicação diferente do nome do arquivo de projeto (.DPR).
- Help file – Define o nome do arquivo de Help (.HLP) associado à aplicação.
- Icon – Define o ícone utilizado no arquivo executável. (.EXE)

Compiler

- Estas opções permitem especificar uma *compilação personalizada***,** ou seja, cada projeto pode ser compilado com uma característica.

Linker

- Estas opções incluem informações para a depuração.

Directories/Conditionals

- Nesta guia pode-se configurar o diretório de saída para os arquivos gerados pela aplicação.

Version Info

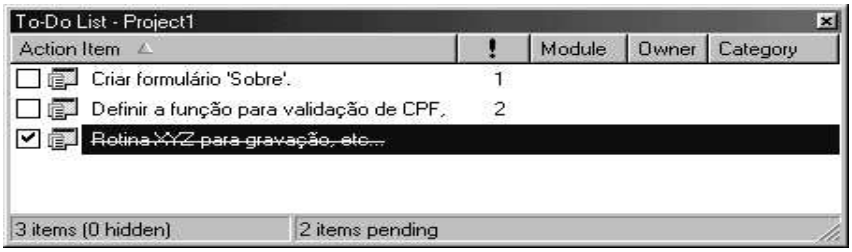
- Estas informações podem ser visualizadas no Windows através do menu rápido do mouse no arquivo executável.

Packages

- Os packages permitem um controle de distribuição através de DLL's básicas *externas* ao executável entre outros recursos.

➢ A Lista To-Do

O Delphi adicionou um recurso de controle/gerenciamento de projeto na versão 5, denominado ***To-Do List***. Você pode incluir incluir ou modificar itens a serem feitos através de diálogos que podem ter dois estados, *prontos* ou à *fazer*.
Para utilizar a **To-Do List** clique no menu **View | To-Do List**.



Clique com o botão direito dentro da janela e escolha **Add**. O arquivo gerado gerado pela lista é gravado no diretório do projeto com a extensão *.todo*

➢ Tipos de Componentes

Fazemos uma distinção de duas categorias básicas quando tratamos dos componentes, são: Componentes Visíveis e Componentes Não-Visíveis.

- Visíveis

Quando um componente pode ser visto pelo usuário em tempo de execução. Exemplo Button e Edit.

- Invisíveis

Alguns componentes aparecem no form durante o tempo de projeto na aparência de um ícone, mas não podem ser vistos pelo usuário em tempo de execução. Exemplo: Timer e MainMenu.

➢ Convenção de Nomeação

A propriedade mais importante de um componente é a propriedade *Name*. É ela que define o nome *interno* com relação ao código escrito em Object Pascal. Para organizar e facilitar o processo de desenvolvimento/manutenção do sistema, grande parte dos desenvolvedores adota uma nomenclatura para tornar o código mais legível possível.

O Delphi adota como nomenclatura padrão o nome da classe da qual o componente é *instanciado* e um número crescente de acordo com o número de ocorrência deste componente no form. Exemplo: Button1, Button2, etc... são componentes *instanciados* da classe TButton .

Não é obrigatória a utilização da convenção de nomes utilizados nesta apostila, mas *é muito importante* fazer uso de uma convenção mais clara possível. Note que, quando trabalharmos com variáveis, também devemos seguir uma convenção, afim de tornar o código mais claro e simples.

Exemplo:

Nome gerado pelo Delphi	Objetivo	Convenção
Button1	Sair da aplicação	BtnSair
Edit1	Receber nome do usuário	EdtNome
Label1	Indicar componente Edit	LblNome

➢ Manipulação de Componentes

Podemos adicionar os componentes ao formulário de três maneiras, (citadas anteriormente) e utilizar ferramentas e técnicas de alinhamento para aumentar nossa produtividade.

Para selecionar um componente, basta clicá-lo uma vez ou na object inspector selecioná-lo na caixa de listagem.
Pode-se então arrastá-lo com o Mouse ou utilizar as teclas CTRL+SETAS para mover o componente. As teclas SHIFT+SETAS alteram as dimensões do componente. Para selecionar mais de um componente ao mesmo tempo, utiliza-se a tecla SHIFT, pode-se mover ou alterar o conjunto.

O recurso de *arrastar e selecionar* (Paint, por exemplo) é válido quando a base é o Form. Quando inserirmos componentes em cima de outro objeto (Panel, por exemplo) é necessário segurar a tecla CTRL no processo de arrastar.

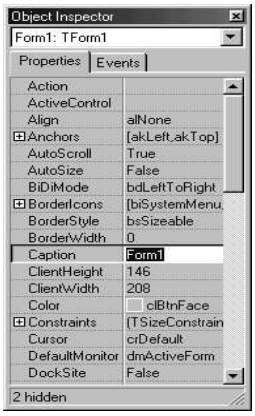
Para definir vários componentes baseados em uma propriedade de outro, altere o componente '*modelo*', selecione-o primeiro e com SHIFT selecione os outros. Na object inspector selecione a propriedade a ser definida para todos Width (largura, por exemplo) e aperte a tecla ESC.

O Delphi dispõe de uma ferramenta para auxílio ao alinhamento dos componentes. Clique no menu **View – Alignment Palette**

Uma outra opção é clicar sobre o componente selecionado e no Speed Menu (Menu rápido) selecionar **Align**.

➢ Utilizando o Object Inspector

O object inspector é uma janela importantíssima na programação orientada a objetos, é através dela que podemos alterar as propriedades e definir os eventos de acordo com o objetivo da aplicação.



Na parte superior da janela há uma caixa de listagem que permite a seleção de componentes já inseridos no formulário. Duas guias (*Properties e Events*) separam as listas de propriedades e eventos.

As propriedades são definidas através de tipos. Podemos citar no exemplo com o objeto form:

Tipos Simples

São tipos String ou valores numéricos definidos ao digitar um valor na frente da propriedade. Exemplo: Name, Caption, Height e Width entre outros.

Tipos Enumerados

São tipos definidos por uma quantidade limitada de opções que devem ser previamente selecionadas, não simplesmente definidas pelo usuário.

Exemplo: Cursor, BorderStyle e WindowState entre outros.

Tipo Set

Algumas propriedades podem conter múltiplos valores. Um exemplo é a propriedade BorderIcons com o sinal + indicando subpropriedades.

Tipos com Editor de Propriedades

As propriedades que são acompanhadas de um ícone de reticências (...) indicam que uma janela de diálogo irá auxiliar na escolha de seu(s) valor(es).

Exemplos: Icon e Font.

3. Introdução à Programação no Delphi

➢ Manipulando Eventos

Conforme podemos perceber, o Windows é um ambiente totalmente orientado a eventos. Todo programa gráfico responde a eventos criados pelo usuário, através do mouse e do teclado. Portanto, os programas que iremos desenvolver também serão orientados a eventos.

A guia *Events* permite o desenvolvedor definir um ***handler***[3] em Object Pascal para um determinado evento que pode ser disparado pelo usuário ou pelo sistema.

Um evento é uma ***ação*** disparada dentro de uma aplicação orientada a Objeto. Podemos citar as ocorrências dos principais eventos que são disponibilizados na maioria dos componentes em Delphi:

Evento	Ocorrência
OnClick	Quando o usuário clicar uma vez com o botão esquerdo do mouse sobre o componente.
OnDblClick	Quando o usuário dá um duplo clique no componente com o botão esquerdo do mouse.
OnEnter	Quando o componente recebe o foco.
OnExit	Quando o componente perde o foco.
OnKeyPress	Quando pressiona uma única tecla de caractere, do teclado.

Exemplo: *Construção de um manipulador de evento para o objeto button.*

- Insira ***um*** componente button no Form, não é necessário mudar nenhuma propriedade.
- Selecione a object inspector a guia *events* e localize o evento *OnClick*.
- Dê um duplo clique no espaço em branco do evento.

Os componentes possuem um evento '**padrão**' para a construção do código, por isso é possível clicar *no componente* duas vezes para abrir um evento.

No *Code Editor* é criada uma declaração do evento na cláusula *Interface* e a implementação do procedimento na cláusula *Implementation*.

Como veremos com mais detalhes nos próximos capítulos, todo código em object pascal é delimitado pelas palavras reservadas "**begin**" e "**end**".Defina apenas as duas linhas de código dentro dos delimitadores.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.Caption := 'Curso de Delphi';
  Showmessage('Exemplo de caixa de diálogo');
end;
```

[3] Manipulador de evento

✗ Executando a aplicação

Para executar o programa e visualizar os dois comandos codificados no evento *OnClick* basta teclar **F9** ou o ícone **Run**.

✗ Comentários

Os comentários no código fonte são importantes e podem ser feitos através dos seguintes símbolos:

//Comentário de linha
{ Comentário de bloco }
(*Comentário de bloco *)

Note que o comentário é desprezado pelo compilador, servindo apenas como orientação para o programador.

➢ Um pouco mais sobre a Programação Orientada a Eventos

A programação baseada em eventos (POE[4]), em resumo, tem a característica de obedecer as ações do usuário. Se você já programou em alguma linguagem para o sistema operacional MS-DOS sabe do que estamos falando.

É mais fácil programar com POE. Um programa estruturado difere em muito deste raciocínio porque seu escopo é rígido e baseado em rotinas, ou seja, pode haver (e na maioria há) momentos em que o usuário deve seguir determinados passos orientados pelo programa; enquanto que na POE existe a execução do evento associado à ação do usuário ou do sistema. *Há dois tipos básicos de eventos:*

- ***Usuário*** – São eventos disparados pelo usuário, por exemplo: OnClick, OnKeyPress, OnDblClick.

- ***Sistema*** – São eventos que podem ocorrer baseados no sistema operacional, por exemplo: O evento OnTimer neh... executa um procedimento a cada intervalo em milisegundos. O evento OnCreate ocorre quando uma *instância* do objeto está sendo criada.

É importante notar que o usuário pode disparar mais de um evento em uma única ação, na verdade isso irá ocorrer com freqüência, de maneira que devemos ter consciência que ***os eventos obedecem uma ordem***.

Supondo a existência de três manipuladores de eventos para um objeto da classe Tbutton: OnMouseDown, OnEnter e OnClick. A ordem destes eventos será:

- OnEnter, quando o botão receber o foco.
- OnMouseDown, quando o usuário pressionor o mouse.
- OnClick, quando o botão do mouse voltar à sua posição original, após ser pressionado.

[4] Programação Orientada a Eventos

➢ A Visual Component Library (VCL) e a Paleta Standard

A VCL, foi desenvolvida pelo Borland, e é uma biblioteca de componentes visuais do Delphi. Ela compreende componentes invisíveis e visíveis, usados nas mais diversas aplicações. O desenvolvimento da VCL pode ser comparada à descoberta da roda, tamanha a mudança que ocasionou, tornando tarefas antes complicadas agora simples, e muitas vezes automática. Vamos considerar alguns objetos da VCL e suas principais *propriedades e métodos*.

✗ *Objeto – Form (Formulário)*

Paleta – Standard: É o principal componente *container* pois permite posicionar os demais componentes.

Propriedades

ActiveControl	Permite definir qual o *primeiro* componente a receber ***foco*** assim que o formulário é criado.
Align	Altera o alinhamento e preenchimento do objeto.
AutoScroll	Permite habilitar as barras de rolagem.
AutoSize	Determina se o controle será automaticamente redimensionado.
BorderIcons	Determina os ícones a serem exibidos na barra de título do formulário.
BorderStyle	Define o estilo da borda do formulário. bsDialog – Borda não redimensionável, comum em caixa de diálogo bsSingle – Borda simples e redimensionável. bsNone – Borda invisível, não redimensionável, sem botões de controle. bsSizeable – Borda padrão redimensionável.
BorderWidth	Define a espessura da borda.
Caption	Indica o rótulo exibido para o componente.
ClientHeight / ClientWidth	Define a altura e largura da área cliente.
Color	Define a cor de fundo de um componente.
Cursor	Indica a imagem exibida pelo ponteiro do mouse quando este ficar sobre o objeto.
DefaultMonitor	Associa o form a um monitor específico em uma aplicação que utiliza vários monitores.
Enabled	Define se o componente está habilitado ou não.
Font	Permite controlar os atributos do texto exibido em um componente.
FormStyle	Determina o estilo do formulário. fsNormal – Definição padrão do formulário. fsMDIChild – O formulário será uma janela-filha de uma aplicação MDI. fsMDIForm – O formulário será o formulário-pai de uma aplicação MDI. fsStayOnTop – O formulário permanece *sobre* todos os outros formulários do projeto, exceto aqueles que também têm a propriedade FormStyle igual a fsStayOnTop.
Height	Define a altura do objeto.
HelpContext	Define o tópico do arquivo help que será exibido ao pressionar a tecla F1.
HelpFile	Define um arquivo de help específico.
Hint	Permite exibir um texto de auxílio no momento em que o ponteiro do mouse permanece sobre o controle.
HorzScrollBar	Define o comportamento de uma barra de rolagem horizontal.
Icon	Define o ícone que será usado pelo formulário.
KeyPreview	Define se o formulário deve ou não responder a um pressionamento de tecla, através do evento OnKeyPress, por exemplo.
Left	Define a coordenada da extremidade esquerda de um componente.
Menu	Permite escolher entre mais de um componente MainMenu.

Name	Define o nome *interno* que identifica o componente dentro da aplicação.
PopupMenu	Define o componente PopupMenu a ser utilizado pelo objeto.
Position	Permite definir o tamanho e posição de um formulário no momento em que ele aparece na sua aplicação.
ShowHint	Define se a string de auxílio deve ou não ser exibida quando o usuário mantém o ponteiro do mouse sobre um controle.
Tag	A propriedade Tag é uma variável do tipo Longint que o Delphi coloca à disposição do usuário, que pode atribuir o significado mais conveniente.
Top	Define a coordenada da extremidade superior de um componente.
VertScrollBar	Define o comportamento de uma barra de rolagem vertical.
Visible	Define se o componente aparece ou não na tela.
Width	Define a largura do objeto.
WindowMenu	Permite definir qual o menu responsável por manipular as janelas-filhas de uma aplicação MDI.
WindowState	Define o estado de exibição de um formulário.

Métodos

Show	Exibe o formulário de manipulação não-modal.
ShowModal	Exibe o formulário de manipulação modal.
Close	Permite fechar o formulário.

✗ *Objeto – Button1 (Botão)*

Paleta – Standard: É um dos objetos mais utilizados para confirmar e disparar rotinas associadas.

Propriedades

Action	Referencia uma ação definida em um objeto TActionList.
Anchors	Permite manter a posição relativa do objeto ao objeto 'parente' quando este é redimensionado.
Cancel	Associa o evento OnClick do objeto ao pressionamento da tecla Esc.
Default	Associa ao evento OnClick do objeto ao pressionamento da tecla Enter.
Parent...	As propriedades *Parent* permitem que o componente receba a mesma formatação do objeto proprietário.
TabOrder	Define a ordem na passagem de foco no momento de pressionamento da tecla TAB.
TabStop	Define se o foco *pára* no componente.

Métodos

SetFocus	Envia o foco do windows para o componente.

✗ *Objeto – Edit (Caixa de edição)*

Paleta – Standard: Um dos principais componentes para a entrada de dados do usuário do sistema.

Propriedades

AutoSelect	Define se o texto exibido pelo controle será selecionado quando este receber o foco da aplicação.
AutoSize	Para componentes TEdit a propriedade determina se a altura do controle será redimensionada quando o tamanho da fonte for alterado.
BorderStyle	Determina o tipo da borda do componente.
CharCase	Determina o se tipo da fonte será maiúscula, minúscula ou normal.

HideSelection	Define se o texto perde a seleção ao perder o foco.
Maxlength	Define um limite para a inserção de caracteres.
PasswordChar	Define qual caractere será usado para ocultar o texto inserido no componente. Usado em Edits para inseção de senhas.
Text	Permite manipular os caracteres inseridos no componente pelo usuário.

Métodos

Clear	Limpa o conteúdo da propriedade text.
SetFocus	Envia o foco do windows para o componente.

✗ *OBJETO – LABEL (RÓTULO DE ORIENTAÇÃO)*

Paleta – Standard: Orientar o usuário à escolha de componentes bem como sua utilização.

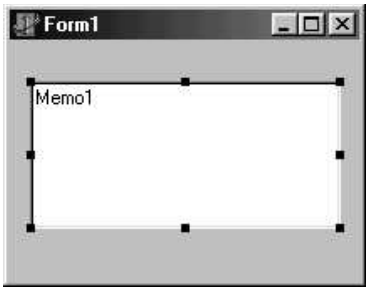
Propriedades

Alignment	Define o alinhamento da string na área do componente.
AutoSize	Para componentes TDBText e TLabel, esta propriedade define se o controle será automaticamente redimensionado para acomodar o texto.
FocusControl	Define qual o componente receberá foco quando o usuário selecionar a combinação de teclas aceleradoras (atalho) se existir.
Layout	Define o alinhamento vertical do texto na área do componente.
ShowAccelChar	Define se o caracter '&' será um literal ou tecla de aceleradora (atalho).
Transparent	Define se o fundo do componente será 'transparente' ou não.
WordWrap	Define se o texto poderá utilizar o 'retorno automático' em caso de ultrapassar a largura definida e se a propriedade *AutoSize* estiver falsa.

➢ **OUTROS COMPONENTES DA PALETA STANDARD**

✗ *OBJETO – MEMO (MEMORANDO)*

Permite o usuário entrar com dados do tipo TStrings, compara-se à funcionalidade do software bloco de notas do Windows.



Propriedades

Lines	Propriedade do tipo TStrings que contém as linhas de texto do componente.
MaxLength	Define o limite máximo de caracteres no componente em sua propriedade Lines.
ReadOnly	Define se o componente é do tipo somente leitura.
ScrollBars	Define se o componente pode trabalhar com barras de rolagem.
WantReturns	Define se a tecla ENTER será utiliza para 'quebra de linha'.
WantTabs	Define a tecla Tab como tabulação ou mudança de foco. Caso falso pode-se utilizar CTRL+TAB para produzir o efeito desejado.

Métodos

LoadFromFile	Permite 'carregar' um arquivo para a propriedade Lines.
SaveToFile	Permite salvar o conteúdo da propriedade Lines em um arquivo especificado.

✗ *Objeto – CheckBox (Caixa de verificação)*

Permite verificar opções boolenas pré-definidas ou re-definidas pelo usuário, ou seja, verifica se determinada opção foi escolhida como True ou False.

Propriedades

AllowGrayed	Define caso verdadeiro, três estados possíveis para o checkbox: checked (ligado), unchecked (desligado) e grayed (parcial). Caso falso, dois estados: checked (ligado) e unchecked (desligado).
Checked	Define se o componente está ligado ou não, caso tenha apenas dois estados.
State	Permite definir três estados se AllowGrayed for verdadeiro.

✗ *Objeto – RadioButton (Botão de 'radio')*

Permite escolher entre um grupo, *pelo menos* uma opção. É muito usado quando desejamos que o usuário escolha uma entre muitas opções disponíveis.

Propriedades

Checked	Define se o componente está ligado ou desligado.

✗ *Objeto – ListBox (Caixa de listagem)*

Permite o usuário entrar ou manipular uma lista de dados.

Propriedades

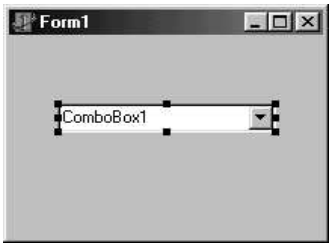
Items	Define uma lista de Strings que aparece no componente.
MultiSelect	Permite selecionar vários itens (Strings) no componente.
Sorted	Define se a lista de Strings será ordenada ou não.

Métodos

Clear	Permite limpar o conteúdo da propriedade Items.
LoadFromFile	Permite 'carregar' um arquvo para a propriedade Items.
SaveToFile	Permite salvar o conteúdo da propriedade Items para um arquivo.

✗ *Objeto – ComboBox1 (Caixa de listagem em formato de cortina)*

Permite o usuário entrar ou manipular uma lista de dados.



Propriedades

Items	Define uma lista de Strings que aparece no componente.
Sorted	Define se os dados serão ordenados.
Text	Define o texto atual da Combobox.

Métodos

Clear	Permite limpar o conteúdo da propriedade Items.
LoadFromFile	Permite 'carregar' um arquvo para a propriedade Items.
SaveToFile	Permite salvar o conteúdo da propriedade Items para um arquivo.

✗ *Objeto – GroupBox (Caixa de agrupamento)*

Permite agrupar componentes e estabelecer um título na propriedade Caption. Muito usado para "isolar" visualmente um grupo de componentes, que por sua funcionalidade no programa se diferem dos demais.

Propriedades

Align	Pemite definir um alinhamento no objeto proprietário.
Caption	Define o texto informativo na parte superior do componente.

✗ *Objeto RadioGroup (Grupo de botões 'radio')*

Permite estabelecer um grupo de botões de radio e manipula-los pela propriedade ItemIndex.

Propriedades

Items	Define os itens disponíveis ao usuário.
ItemIndex	Define qual dos itens está selecionado.
Columns	Define o número de colunas para organização dos componentes.

✗ *Objeto – Panel (Painel)*

Permite agrupar outros objetos e estabelecer um efeito visual nas aplicações.

Propriedades

Align	Define o alinhamento do componente em relação ao seu proprietário.
Bevel...	Define a característica das bordas (interna e externa) bem como sua espessura.
BorderStyle	Define o tipo da borda.

✗ *Objetos – MainMenu e PopupMenu (Menu principal e Menu rápido)*

Define os Menus utilizados pelo usuário pelo botão esquerdo (MainMenu) ou pelo botão direito (PopupMenu) do Mouse.

Propriedades

Items	Define um novo item de Menu.
Images	Define um objeto do tipo 'ImageList'.

O objeto MainMenu permite a construção de *sub-menus* através de seu *construtor* clicando no item com o botão direito e escolhendo a opção *Create submenu*. Pode-se também excluir ou incluir items aleatoriamente através do botão direito no item desejado.

Para criar um separador de menus, utilize o operador de subtração ('-') e confirme com a tecla Enter.

✗ *BitBtn (Botão com figuras opcionais)*

Paleta – Additional: Permite inserir figuras para uma melhor orientação do usuário, estética do programa, além de funções pré-definidas.

Propriedades

Glyph	Define um Bitmap para o componente. (Arquivo com extensão .BMP)
Kind	Define o tipo de Bitmap exibido pelo usuário. **BkCustom:** *Bitmap definido pelo usuário.* **BkOk:** *Botão OK padrão, com uma marca de verificação na cor verde e propriedade Default igual a True.* **BkCancel:** *Botão Cancel padrão, com um "x" na cor vermelha e propriedade Cancel igual a True.* **BkYes:** *Botão Yes padrão, com uma marca de verificação na cor verde e propriedade Default igual a True.* **BkNo:** *Botão No padrão, com uma marca vermelha representando um círculo cortado e propriedade Cancel igual a True.* **BkHelp:** *Botão de auxílio padrão, com uma interrogação na cor cyan. Quando o usuário clica sobre o botão, uma tela de auxílio deve ser exibida (baseada no código do desenvolvedor).* **BkClose:** *Botão Close padrão, com o desenho de uma porta. Quando o usuário clica sobre o botão, o formulário a que ele pertence se fecha.* **BkAbort:** *Botão Abort padrão, com um "x" na cor vermelha e propriedade Cancel igual a True.* **BkRetry:** *Botão Retry padrão, com uma seta circular verde.* **BkIgnore:** *Botão ignore padrão, com o desenho de um homem verde se afastando.* **BkAll:** *Botão All padrão, com uma marca de verificação dupla na cor verde e propriedade default igual a True.*
ModalResult	Permite encerrar a execução de um formulário *Modal* quando o seu valor for diferente de *mrNone*.

✗ *Objeto MaskEdit – (Caixa de edição com máscara)*

Paleta – Additional: Permite estabelecer uma máscara para a entrada de dados no componente. Pode ser considerado literalmente um componente 'Edit com máscara'.

Propriedades

CharCase	Define o tipo dos caracteres.
EditMask	Permite definir uma máscara para entrada de dados.
PasswordChar	Define um caracter para ocultar a entrada de dados.

✗ *Objeto – Image (Imagem)*

Paleta – Additional: Permite inserir uma figura para uso geral na aplicação.

Propriedades

AutoSize	Permite alterar o tamanho do *componente* baseado no tamanho da figura.
Picture	Define a figura a ser exibida.
Stretch	Permite alterar o tamanho da *figura* baseado no tamanho do componente.

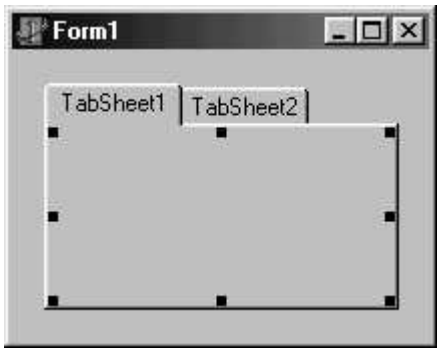
Métodos

LoadFromFile	Permite 'carregar' um arquivo de figura na propriedade Picture.

Para trabalhar com imagens jpg, é necessário acrescentar na cláusula uses da interface a biblioteca **jpeg**.

✗ *OBJETO - PAGECONTROL*

Paleta – Win32: Permite definir guias para agrupar os demais componentes. Cada guia representa um componente TabSheet do tipo TTabSheet, uma espécie de 'sub-objeto' do PageControl.



Propriedades

ActivePage	Permite determinar qual a guia foi selecionada pelo usuário.

Para criar novas páginas, clique com o botão direito no componente *PageControl* e escolha New Page.

✗ *OBJETO – OPENDIALOG (CAIXA DE DIÁLOGO PARA ABERTURA DE ARQUIVOS)*

Paleta – Dialogs: Permite utilizar uma caixa de diálogo pronta com recursos padronizados pelo sistema operacional.

Propriedades

DefaultExt	Especifica a extensão a ser adicionada ao nome de um arquivo quando o usuário digita o nome de um arquivo sem a sua extensão.
FileName	Define o arquivo selecionado no componente.
Filter	Permite definir as máscaras de filtro de arquivo a serem exibidas.
FilterIndex	Define o filtro default a ser exibido na lista drop-down que define os tipos de arquivos selecionáveis.
InitialDir	Define o diretório default quando a caixa de diálogo é aberta.

Options	Neste componente, options define uma série de valores booleanos.
Title	Define o título da caixa de diálogo.

Os componentes da paleta dialogs são executados através do método ***execute.*** Este método é uma função que retorna um valor booleano, assim para exibir uma caixa de diálogo, podemos escrever:

if OpenDialog1.Execute **then**

Se o usuário escolher algum arquivo e confirmar a caixa, *execute* retorna verdadeiro, caso contrário, falso.

Exercício: Crie um programa que mostre arquivos BMP e JPG em uma caixa de imagens. A imagem a ser exibida deve ser escolhida pelo próprio usuário, em tempo de execução.

✗ *Objeto – ImageList (Lista de imagens)*

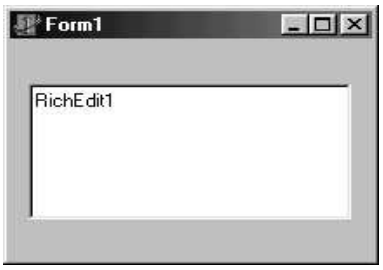
Paleta – Win32: Permite definir um conjunto de ícones para serem re-utilizados por diversos componentes de recebem este objeto como provedor de uma lista de imagens.

Para incluir imagens no componente ImageList, clique 2 vezes rapidamente no componente e clique no botão Add.

O Delphi possui um conjunto de ícones e imagens em uma pasta padrão[5]:
C:\Arquivos de programas\Arquivos comuns\Borland Shared\Images

✗ *Objeto – RichEdit (Texto com formatação)*

Paleta – Win32: Permite formatar o texto (Negrito, Itálico, Sublinhado, Fontes, etc...) para a leitura de outros editores compatíveis com o padrão RTF.



Propriedades

Lines	Define o texto exibido no componente.
WantReturns	Define a tecla Enter como quebra de linha.
WantTabs	Define a tecla Tab como tabulação ou mudança de foco. Caso falso pode-se utilizar CTRL+TAB para produzir o efeito desejado.
WordWrap	Define a quebra de linha automática de texto.

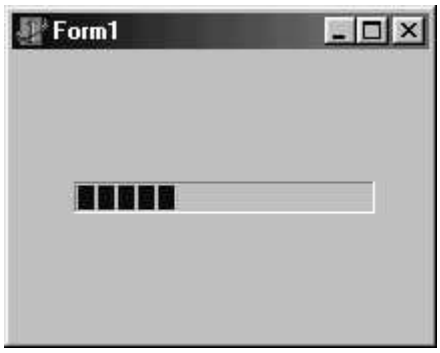
Métodos

Clear	Permite limpar o conteúdo do componente.
LoadFromFile	Permite 'carregar' um arquivo para a propriedade Lines.
SaveToFile	Permite salvar o conteúdo da propriedade Lines em um arquivo.

[5] Caso o Delphi esteja instalado na pasta padrão.

✗ *Objeto – ProgressBar (Barra de progresso)*

Paleta – Win32: Permitir ao usuário ter um acompanhamento de uma rotina demorada.



Propriedades

Max	Permite definir o valor máximo para a faixa de valores no componente.
Min	Permite definir o valor mínimo para a faixa de valores no componente.
Orientation	Define se o componente deverá ser vertical ou horizontal.
Position	Define a posição corrente do controle no componente.
Step	Define o incremento usado na variação do valor da propriedade position.

✗ *Objeto – Animate (Animações)*

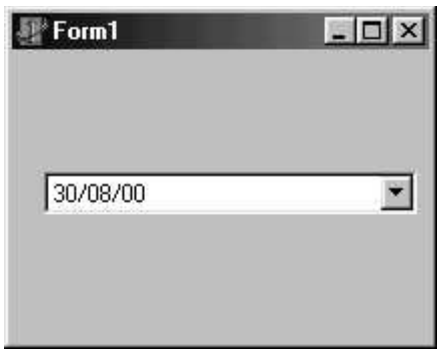
Paleta – Win32: Permite exibir um 'filme' .AVI para ilustrar tarefas (rotinas) em andamento, etc.

Propriedades

CommonAVI	Define o AVI a ser exibido.
Active	Liga e desliga a exibição do AVI.
Repetitions	Define um número inteiro correspondente ao número de repetições. Zero define repetições indefinidas.

✗ *Objeto – DateTimePicker (Data e hora através de uma Combobox)*

Paleta – Win32: Permite ao usuário escolher uma data através de um componente que possui um importante impacto visual e facilidade operacional.



Propriedades

CalColors	Define as cores do calendário.
Date	Define a data selecionada no componente.
DateFormat	Define o formato da apresentação da data.
DateMode	Define o estilo da caixa de listagem.
Kind	Define se o componente deve trabalhar com data ou hora.
MaxDate	Define uma data máxima para uma faixa de valores.
MinDate	Define uma data mínima para uma faixa de valores.

✗ *Objeto – MonthCalendar (Calendário mensal)*

Paleta – Win32: Permite ao usuário escolher uma data através de um componente que possui um importante impacto visual e facilidade operacional.

Propriedades

Date	Define a data selecionada no componente.
FirstDayOfWeek	Define qual o primeiro dia da semana.
WeekNumbers	Permite numerar as semanas.

✗ *Objeto – StatusBar (Barra de status)*

Paleta – Win32: Um dos principais componentes de informações sobre operações gerais no sistema.

Propriedades

AutoHint	Permite exibir o hint do componente automaticamente na barra de status. Se não houver painéis, a barra deve ter a propriedade SimplePanel ligada.
SimplePanel	Define que a barra de status será *sem* divisões.
SimpleText	Define o texto a ser exibido pela barra de status.
Panels	Permite a criação e edição de *painéis* na barra de status. A propriedade SimplePanel deve estar desligada. Pode-se também dar um duplo clique na barra de status.

✗ *Objeto – ToolBar (Barra de ícones)*

Paleta – Win32: Permite criar barras de ícones de maneira rápida e simples.

Propriedades

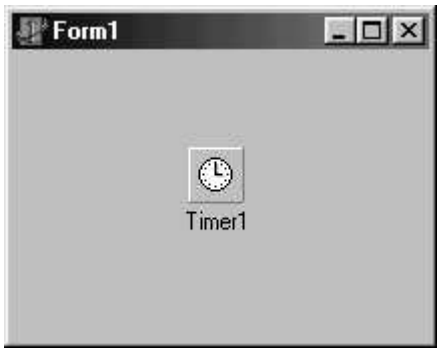
Flat	Define um efeito visual com relevo através do mouse nos botões.

Images	Permite definir um objeto do tipo *ImageList*.
HotImages	Permite definir um objeto do tipo *ImageList* a ser usado no momento em que o mouse passa (ou clica) sobre o componente.
ShowCaptions	Permite exibir a propriedade *caption* dos botões.

Para adicionar botões ou separadores na ToolBar, clique com o botão direito sobre o componente e escolha New Button ou New Separator.

✗ *OBJETO – TIMER (TEMPORIZADOR)*

Paleta – System: este componente permite a execução de rotinas em loop, em um intervalo pré-definido.

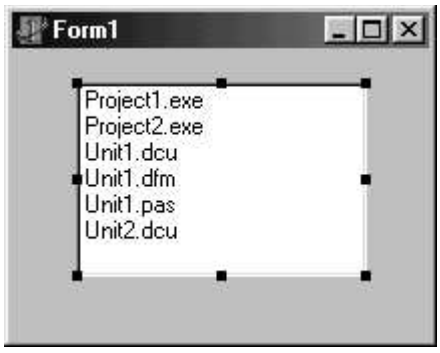


Propriedades

Enabled	Permite 'ligar' o timer, ou seja, ele entra em um loop executando o evento OnTimer até que seja atribuído falso ou terminada a aplicação.
Interval	Define em milisegundos o intevalo de repetição do evento OnTimer.

✗ *OBJETO – FILELISTBOX (CAIXA DE LISTAGEM DE ARQUIVOS)*

Paleta – Win 3.1: permite listar arquivos de determinado diretório.



Propriedades

FileEdit	Define uma caixa de edição (TEdit) que exibirá o arquivo atualmente selecionado.
FileName	Define o nome do arquivo selecionado. Válido em tempo de execução.
Mask	Define máscaras de filtro (separadas por ponto e vírgula) para a exibição dos arquivos.

✗ *Objeto – DirectoryListBox (Caixa de listagem de diretórios)*

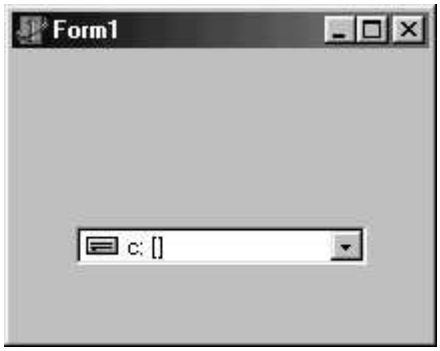
Paleta: Win 3.1: permite listar os diretórios do drive desejado.

Propriedades

DirLabel	Permite exibir o diretório corrente com a propriedade Caption de um componente do tipo TLabel.
FileList	Permite a conexão com um componente TFileListBox.

✗ *Objeto - DriveComboBox (Caixa de listagem de drives)*

Paleta: Win 3.1: permite listar os drives disponíveis no computador.

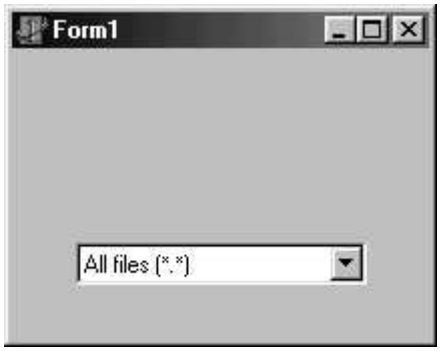


Propriedades

DirList	Permite a conexão com um componente TDirectoryListBox.

✗ *Objeto – FilterComboBox (Caixa de listagem de filtros)*

Paleta: Win 3.1: permite estabelecer filtros para visualização de arquivos.



Propriedades

FileList	Permite a conexão com um componente TFileListBox.
Filter	Permite definir as máscaras de filtro de arquivo a serem exibidas.

✗ *Caixas de Diálogo*

Podemos utilizar alguns tipos de caixas de diálogo pré-definidas pelo Delphi facilitando em muito o desenvolvimento de aplicativos:

✗ *ShowMessage*

A caixa de diálogo ShowMessage é declarada internamente pelo Delphi desta forma:

```
procedure ShowMessage(const Msg: string);
```

Onde o parâmetro Msg é um dado String. Exemplo:

```
ShowMessage('Um texto ou propriedade string será exibida.');
```

✗ *MessageDlg*

A caixa de diálogo MessageDlg é declarada internamente pelo Delphi desta forma:

```
function MessageDlg(const Msg: string; DlgType: TMsgDlgType;
      Buttons: TMsgDlgButtons; HelpCtx: Longint): Word;
```

Onde:

const Msg: string	É uma constante string ou propriedade deste tipo.
DlgType: TmsgDlgType	**MtWarning:** *Contém um ícone exclamação amarelo.* **MtError:** *Contém um ícone vermelho de 'parada'.* **MtInformation:** *Contém um ícone 'i' azul.* **MtConfirmation:** *Contém uma interrogação verde.* **MtCustom:** *Não contém BitMap.*

Buttons: TMsgDlgButtons	mbYes mbNo mbOK mbCancel mbAbort mbRetry mbIgnore mbAll mbNoToAll mbYesToAll mbHelp
HelpCtx: Longint	Define um número para o help de contexto. Por padrão, zero '0'.

O *retorno* da função é o tipo do botão, como *mr*, ao invés de *mb.* Desta maneira pode-se fazer testes lógicos como no exemplo:

```
if MessageDlg('Deseja sair?', mtConfirmation, [mbYes, mbNo], 0)=mrYes
then...
```

✗ *Application.MessageBox*

Uma outra caixa de diálogo é o método MessageBox do objeto Application. Esta função está definida da seguinte maneira:

```
function MessageBox(const Text, Caption: PChar; Flags: Longint):
Integer;
```

Onde:

const Text	É uma constante string ou propriedade deste tipo.
Caption: PChar	Define uma string para o título da janela.
Flags	Define os botões, ícones e a possibilidade de focar um determinado botão. Os valores para ***botões*** são: MB_ABORTRETRYIGNORE, MB_OK, MB_OKCANCEL, MB_RETRYCANCEL, MB_YESNO, MB_YESNOCANCEL Os valores para os ***ícones*** são: MB_ICONEXCLAMATION, MB_ICONWARNING, MB_ICONINFORMATION, MB_ICONASTERISK, MB_ICONQUESTION, MB_ICONSTOP, MB_ICONERROR, MB_ICONHAND Os valores para a definição do ***botão default*** pode ser: MB_DEFBUTTON1, MB_DEFBUTTON2, MB_DEFBUTTON3, MB_DEFBUTTON4

O *retorno* da função é o tipo do botão, mas como id: (IDABORT, IDCANCEL, IDIGNORE ,

IDNO, IDOK, IDRETRY ,IDYES, por exemplo). Desta maneira pode-se fazer testes lógicos, como no exemplo:

```
if Application.MessageBox('Texto','Título',MB_YESNOCANCEL +
MB_ICONINFORMATION + MB_DEFBUTTON2) = IdYes then ...
```

✗ *CAIXAS DE ENTRADA*

Podemos obter dados do usuário através de caixas das caixas de entrada.

✗ *INPUTBOX*

A função InputBox retorna um tipo String, que é dado digitado pelo usuário na sua utilização. Sua definição interna é a seguinte:

```
function InputBox(const ACaption, APrompt, ADefault: string): string;
```

Onde:

const ACaption	Define o título da janela
APrompt	Define um rótulo (texto) para orientação dentro da caixa.
ADefault	Define um valor default para a caixa.

Exemplo:

```
InputBox('Título da janela','Descrição','Valor Padrão')
```

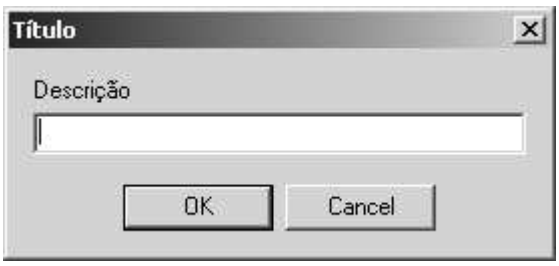
✗ *INPUTQUERY*

A função InputQuery retorna um tipo Booleano, o dado digitado pelo usuário será colocado em uma variável do tipo string *previamente* declarada.
Sua definição interna é a seguinte:

```
function InputQuery(const ACaption, APrompt: string; var Value: string):
Boolean;
```

Exemplo:

```
if InputQuery('Título', 'Descrição', aux) and (aux <> '') then...
```



Neste exemplo acima, a janela só retornará verdade **se** houver algum valor digitado **e** o usuário clicar no botão OK, caso contrário o retorno será falso.

➢ COMPONENTES ÚTEIS PARA A ENGENHARIA

✗ *O COMPONENTE EXPRESS*

Este componente pode ser encontrado na internet, sendo "open source", ou seja, o usuário poderá modificá-lo, depurá-lo, pois tem acesso total ao seu código fonte.

Quando digitamos um texto em um *Edit*, o Delphi o interpreta como uma *string*, ou seja, como um texto. E se desejarmos que este texto seja validado como uma expressão, com diversas variáveis que podem assumir valores distintos e com variações distintas ao longo do tempo? Neste caso, podemos usar o componente *Express*, escrito por Renate Schaaf, que faz com que seja possível interpretar uma string como uma expressão, em tempo de execução.

Desta forma é possível disponibilizar um Editbox para o usuário, que em tempo de execução poderá digitar uma expressão que será entendida pelo Delphi não como uma string, mas como uma expressão com variáveis e constantes. Neste tópico daremos uma breve abordagem sobre a utilização deste componente, assim como um simples exemplo demonstrando seu funcionamento.

O Trecho abaixo é apenas a tradução do arquivo Readme sobre o componente Express.

Com este componente, funções com até 9 variáveis podem ser validadas, sendo que três destas são chamadas *variables* e as outras seis são os chamados *parameters.* As propriedades públicas (disponíveis ao programador) deste componente são:

VariableList: uma string que deve receber os nome das três variáveis que o componente deve interpretar, sem espeço entre elas. Exemplo: xyz.

ParameterList: uma string que recebe os nomes dos seis parâmetros que serão usados, sem espaço entre eles. È importante salientar que a diferença entre os parâmetros e as variáveis é o fato que as variáveis devem variar mais rapidamente que os parâmetros, mas nada impede o programador de usar os parâmetros da mesma forma que usa as variáveis, sem prejudicar consideravelmente o desempenho do programa.

Expression: A expressão que será validada. É uma string, que normalmente é atribuída em tempo de execução, e seu valor é escolhido pelo usuário final.

SyntaxText: Um texto que pode ser digitado para ajudar o usuário final na utilização do componente, explicando as principais funções suportadas, etc. Já existe um texto padrão para o componente.

MessageOnError: Booleana. Se seu valor for *true*, uma mensagem será exibida caso haja algum erro na sintaxe da espresão digitada.

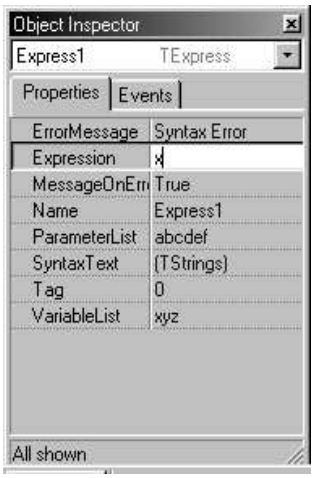
ErrorMessage: String que representa a mensagem que será exibida ao usuário em caso de erro.

A função principal, responsável por validar a string que estiver armazenada em *Expression*, é a função **'TheFunction'**.

Os seguintes arquivos estão presentes no .zip:

Express.pas {main component, registers to 'Samples'}
Parsglb.pas {unit used by express}
Build.pas {unit used by express}
Pars.pas {unit used by express}
SynDlg.pas {unit used by express}
SynDlg.dfm {form file for above}
Expdem.pas {demo unit}
Expdem.dfm {form file for above}
Expdemo.dpr {demo project}
Expdemo.res {resource file for above}
read.me {help}

Abaixo podemos visualizar a janela "Object Inspector" referente ao componente:



Um exemplo de utilização deste componente pode ser encontrado no apêndice desta apostila.

✗ O Componente TChart (TeeChart)

☑ Introdução

Uma tarefa entediante e que aumenta em muito as possibilidades de se cometer erros ao se programar é o fato de ter que se criar um código próprio para plotar gráficos, mesmo que simples. Quanto então se pretende fazer gráficos variáveis no tempo ou então com sucessivas replotagens cresce de forma incalculável a chance de pequenos erros causarem muita dor de cabeça.

Este componente, que recebe o nome de TeeChart® é de manuseio extremamente simples e de desempenho comprovado em situações extremas. A versão do componente que é distribuída atualmente com o Delphi 7®, sem custo adicional, é o TeeChart 4®. Esse componente que descomplica a apresentação de gráficos é de produção da Steema Software™. A sua versão completa tem quase todas as mesmas funções de plotagem que o MatLab® com simplicidades de manuseio ainda maiores.

Mais informações podem ser obtidas em http://www.steema.com ou em qualquer site de busca da Internet, visto que este componente, por sua grande funcionalidade, é bastante difundido.

☑ Princípios Básicos

Neste capítulo abordaremos de forma rápida as principais propriedades e procedimentos envolvendo o TeeChart, de forma que o seu primeiro uso possa ser feito de forma rápida e eficaz.

O que seria um gráfico? Nada mais que uma série de pontos disposta de acordo com os eixos que a delimita. Pode também ser vista simplesmente como a disposição esquemática que um conjunto numérico de informações.

- *Usando o Help On-Line*

Para obter ajuda a respeito de um componente TeeChart basta selecioná-lo e pressionar F1 Da mesma forma, para obter ajuda a respeito de funções, procedimentos ou métodos do TeeChart, selecione o texto no código ou então selecione a propriedade ou evento no Object Inspector e pressione F1.

☑ Usando o TeeChart Wizard

Estando disponível a partir das versões 32bits do Delphi este guia pode ser encontrado em File -> New -> Bussines -> New Itens – TeeChart Wizard. Daí por diante é só seguir as diretrizes apresentadas na tela.

☑ Que componente TeeChart devo usar?

O TQRChart é um componente desenvolvido especialmente pensando-se em aplicações que utilizem o QuSoft's QuickReport. É este um descendente do TDBChart.

Já o TChart e o TDBChart são os pontos de partida para TODOS os gráficos TeeChart. A diferença é que: se você deseja criar uma série a partir de uma rotina que cálculo qualquer ou então entrar manualmente com os valores, utilize o TChart, no entanto se o seu gráfico utilizará como fonte de dados uma tabela ou uma inferência SQL utilize o TDBChart.

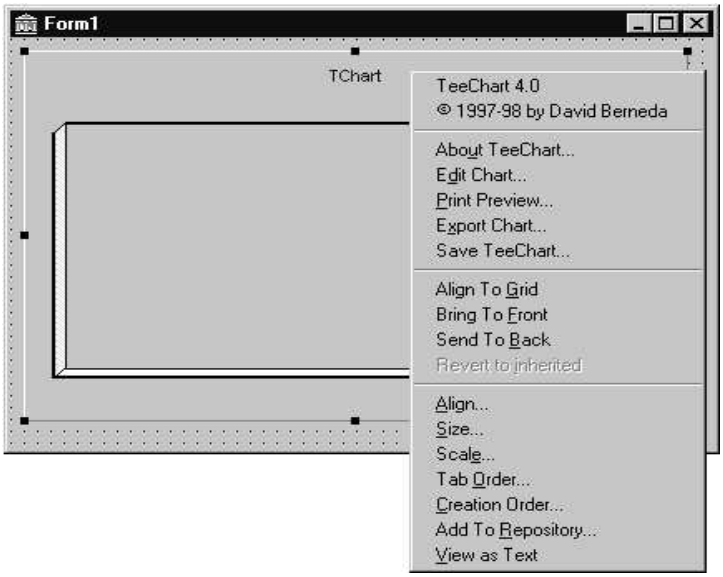
☑ *Criando um novo gráfico usando os componentes TChart ou TDBChart*

Em um formulário de sua escolha coloque um TChart(encontre-o na sua paleta de componentes). Usando o botão direito você pode verificar todas as possíveis opções de edição, ou então, algumas delas estarão disponíveis no Object Inspector.

☑ *O Editor de Gráficos*

O editor de gráficos disponível para o componente TChart é apresentado na primeira das duas abas quando se da um duplo clique sobre o gráfico em questão: Chart e Series. Na primeira, Chart, estão acessíveis propriedades gerais da apresentação visual do componente no formulário, mencionado neste material como "Editor de Gráficos".

Ao se adicionar um Gráfico a um formulário e criar uma série para este mesmo, automaticamente, valores aleatórios para a série são disponibilizados durante o tempo de projeto, para que se tenha idéia da apresentação da série em tempo de execução.



☑ *Configurando Séries usando Tchart*

As séries são editadas na segunda aba da janela que surge após o duplo clique sobre o gráfico, Series: aí estão acessíveis o gerenciamento e propriedades das séries de dados a serem apresentados nos gráficos. Depois de escolhidas as características, da série, editáveis graficamente utilizando-se o Editor de Gráficos, são necessárias algumas linhas de código para que sua série se transforme em algo útil.

☑ *Configurando Séries usando TDBChart*

Para criar uma série que utiliza um banco de dados como fonte de dados é necessário ter o Borland Database Engine (ou equivalente) instalado corretamente. Depois disso é só seguir os passos abaixo apresentados:

- um componente do tipo TTable deve estar disposto no formulário e deve apontar a

um banco de dados e uma tabela(propriedades DatabaseName e TableName, respectivamente);

- um componente do tipo TDatasource deve estar disposto no formulário e deve apontar a um conjunto de dados, que no nosso caso será a tabela disponibilizada pelo componente acima adicinonado(propriedade DataSet recebendo o nome da tabela indicado em TableName do componente TTable);

- um componente do tipo TDBGrid deve estar disposto no formulário e deve apontar a um conjunto de dados, que no nosso caso será a o TDatasource inserido acima. (obs.: para que os dados existentes no componente table sejam exibidos a propriedade Active do componente TTable inserido deve conter o valor "True");

- depois disso, basta que você crie um TDBChart e adicione a tabela do componente TTable como fonte de dados da série atribuída ao gráfico.

Diversos tipos de fontes de dados podem ser atribuídos a um gráfico:

- no data: somente serão atribuídos/adicinados valores para a série em tempo de execução;

- random values: enquanto em tempo de projeto, são atribuídos valores randômicos, que não tem nenhuma influência quando em tempo de execução, para que se possa ver a apresentação da série;

- a function: um conjunto de funções pode operar entre as séries atribuídas para aquele gráfico(soma, subtração, maior valor, menor valor, produto, divisão, etc...);

- dataset: pode-se tomar valores tanto de tabelas, quanto de fontes de dados e inferências (queries).

Estas fontes de dados também estão disponíveis no TChart, exceto o dataset.

☑ *Alterando informações num Gráfico*

Como em qualquer conjunto organizado de dados, a edição pode ser feita através do uso de indexadores, ou seja:

- para apagar o n-ésimo dado de uma série, utiliza-se o procedimento "Delete": Serie.Delete(n);

- para se alterar o valor do n-ésimo dado de uma série, utiliza-se o procedimento "Value": Serie.XValues[n] := 8;

☑ *Referência do Componente*

• O Componente TChart

O componente TChart é um bloco básico de construção de gráficos que não utilizem bancos de dados como fontes de dados,

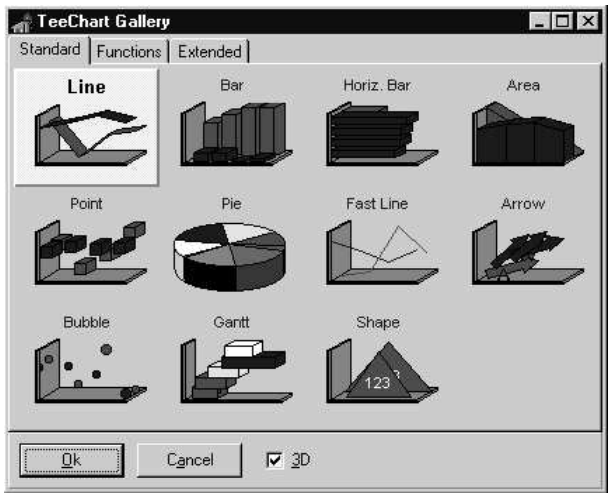
• O Componente TDBChart

O componente TDBChart efetua exatamente a mesma funcionalidade do TChart a diferença é que ele usa um banco de dados como a fonte de dados de uma série.

• As séries disponíveis no Tchart:

- **TChartSeries:** O TChartSeries é um componente que antecede todos os tipos de Séries. Quando fazendo referencia as propriedades específicas de uma série qualquer que esteja trabalhando, observe também as propriedades do TChartSeries para ter uma lista

completa de todas as propriedades comuns a todos os tipos de séries.



- **Linhas:** Há dois tipos de séries para se plotar linhas, Line e FastLine. A diferença é simples, FastLine é uma série mais simples embora otimizada para aplicações onde a velocidade de plotagem não deve interferir nos demais processos da aplicação. A principal diferença de velocidade na manipulação destas duas séries se da na adição de novos pontos para a série.

* *Line - TLineSeries:* Esta série pode ser plotada em 2 ou 3-D e tem a opção de plotar séries em degraus aos invés de uma interpolação linear.

* *FastLine - TFastLineSeries:* Está série só plota em 2-D, preço da velocidade, ficando ainda melhor utilizada quando se seleciona uma quantidade adequada de memória, ajustando a propriedade TeeDefaultCapacity(onde se determina o número esperado de pontos, 10000 é o valor padrão).

- **Outros Tipos de Séries:**

* *Barras - TBarSeries:* Este é o tipo mais comum de gráfico, há uma vantagem na implementação do TChart que possibilita o uso de gráficos de barra mistos. Tipos diferentes de barras pode ser ajustados na propriedade BarStyle. Outras duas propriedades importantes quando se usa gráficos de barras concatenados é a ordem em que eles aparecem e a forma com que eles são concatenados: Stacked = None, Side, Stacked ou Stacked 100%.

* *Barras Horizontais - THorizBarSeries:* É análoga à série anterior com a diferença de ter os valores horizontais como variável dependente. É uma boa opção quando se deseja usar textos longos como variáveis independentes. Tem todas as propriedades e métodos que TBarSeries tem.

* *Área - TAreaSeries:* É similar a série TLineSeries com a diferença que a área sobre a curva é preenchida.

* *Pontos - TPointSeries:* É similar a série TLineSeries com a diferença que a curva em si não é plotada, somente os pontos, o que faz com que o tipo de ponto (Serie.########) seja uma propriedade adicional.

* *Pizza - TPieSeries:* É um tipo único de gráfico por não precisar de eixos.

* *Vetores - TArrowSeries:* É uma boa série pra se plotar tensores(direção, sentido, módulo e ponto de aplicação).

* *Círculos - TBubbleSeries:* É uma série onde se configuram três parâmetros

para cada ponto: XValue, YValue e RadiusValue, gerando assim um círculo para cada ponto referenciado.

* *Gantt - TGanttSeries:* É uma série parecida com a de barras horizontais, no entanto há a uma possibilidade adicional: pode-se plotar segmentos de reta sequenciais. Uma outra opção disponível é a de tracar uma linha ligando dois segmentos de reta plotados: a propriedade NextTask[].

* *Formas - TShapeSeries:* É uma série destinada a adição de informações à área de plotagem do gráfico. É possível adicionar texto associado a uma forma qualquer, que também pode ser relacionada a uma outra série. As formas são posicionadas de acordo com o deslocamento horizontal e vertical do canto superior esquerdo destas mesmas, o que pode ser associado aos valores de uma série em tempo de execução, de forma dinâmica.

- **Combinando Séries:** Teoricamente, o limite de séries a se adicionar a um gráfico é o limite de memória disponível no ambiente que gerencia a execução da aplicação. Qualquer combinação de séries poderia ser usada num mesmo gráfico, quando isso não for possível o Editor de Gráficos sombreia as séries que não podem ser adicionadas(devido a séries imcompatíveis já adicionadas). Esse problema se da ao fato das diferenças entre marcações de eixos entre as séries.

- **Tipos adicionais de séries:**

* *Bezier - TBezierSeries:* Plota uma curva de Bezier, uma curva que passa por pelo menos 1 de cada conjunto 3 pontos da curva, é uma série que faz uma interpolação para plotar o gráfico.

* *Candle - TCandleSeries:* Plota curva com valores máximos, mínimos e valores marcados, boa para plotar informações de medidas com erros ou intervalos de pertinência.

* *Contour - TContourSeries:* É uma série que plota curvas de nível.

* *ErrorBar - TErrorBarSeries:* É uma série de barras com uma marcação de erro para cada valor.

* *Point3D - TPoint3DSeries:* É uma série que plota pontos conectados, sendo que os pontos estão dispostos no espaço tridimensional.

* *Polar - TPolarSeries:* É uma série que plota pontos em coordenadas polares (fase e módulo) em um sistema de eixos apropriado.

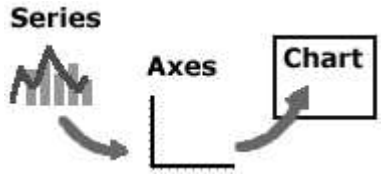
* *Radar - TRadarSeries:* É uma série com todas as propriedades do TpolarSeries.

* *Surface - TSurfaceSeries:* É uma série utilizada para plotar superfícies.

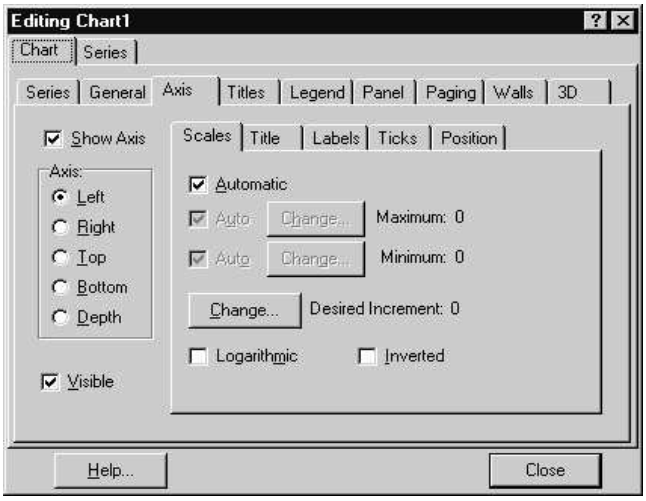
* *Volume - TVolumeSeries:* Mais uma série de uso para gráficos financeiros.

☑ *Trabalhando com Gráficos e Séries*

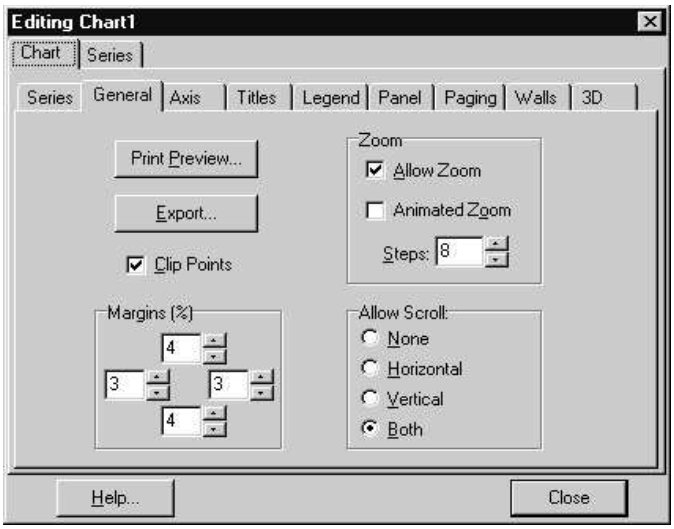
Nesta seção iremos nos atentar aos detalhes da manipulação de dois componentes(TChart e TChartSeries) que irão facilitar ou mostrar potencialidades para as suas aplicações. Uma coisa essencial é notar que se pode trabalhar indistintamente com Gráficos e Séries como se fossem objetos absolutamente distintos, embora estejam atrelados para se plotar um gráfico.



Já utilizando esse visão disjuntista pode observar que algumas propriedades do componente TChart na verdade são alteradas em alguns de seus sub-componentes, tal como as características de eixo: BottomAxis, TopAxis, LeftAxis e RightAxis(para descrições completas procurar por referências sobre TChartAxis na ajuda do Delphi®). Outro exemplo de sub-componente são as legendas.



As características gerais de aparência do seu gráfico são controladas pelo TChart e pode-se também obter informção de gráficos já plotados a partir do TCanvas usado pelo TChart.



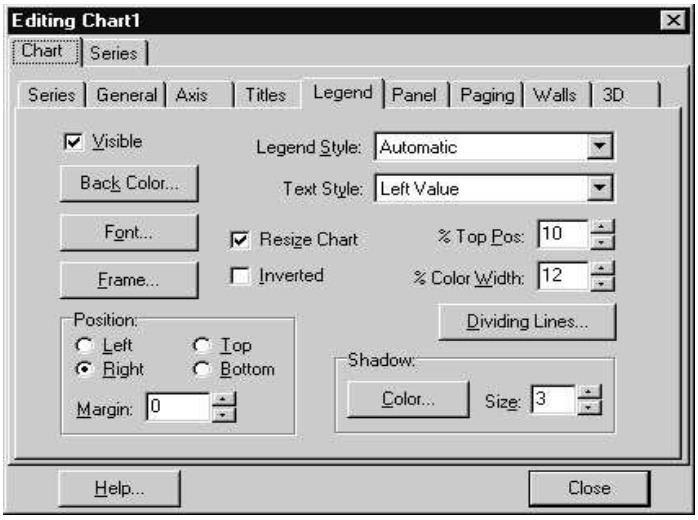
Todo gráfico deve ter pelo menos uma série, mesmo que seja um banco de dados, associado a ele para que algo seja realmente apresentado em tela, isto pode ser facilmente feito em tempo de projeto usando o editor de gráficos ou então em tempo de execução:

```
SerieExemplo.ParentChart := GraficoQualquer;
```

Alguns das propriedades comumente utilizadas são:

- **Incremento das Etiquetas** - GraficoQualquer.[BottomAxis, TopAxis, LeftAxis, RightAxis].Increment: Determina exatamente com que frequência serão exibidas as etiquetas das coordenadas em questão;

– **Separação Percentual** – Grafico.Eixo.LabelsSeparation(Const 0..100 : Integer): Determina a separação percentual existente entre as etiquetas, desta forma se controla o espaçamento entre elas, evitando sobreposição entre as mesmas.



– **Ângulo** – Grafico.Eixo.LabelsAngle(Const 0..360 : Double): Define a angulação com que as etiquetas serão dispostas ao lado dos eixos.

– **BitMaps e Metafiles** – Grafico.SaveToBitmapFile(Const FileName : String)/ SaveToMetafile(Const FileName : String)/ SaveToMetafileEnh(Const FileName : String): O TChart possibilita que você salve os gráficos plotados em dois diferentes formatos de arquivos: BMP e "WMF e EMF". O formato BitMap é usado internamente pelo TChart pois é de plotagem mais rápida, no entanto o formato Metafile tem melhor funcionalidade quando se precisa redimencionar o gráfico depois de salvo.

– **Zoom** - Grafico.AllowZoom[0, 1]: Esta propriedade deve ser modificada quando não se queira que o usuário modifique a vizualização do gráfico apresentado. Também é possível executar usar o zoom em tempo de execução, no entanto é necessário se criar um retângulo referenciando os pixels da figura que o delimitarão: Rect.Left := 10; Rect.Top := 7; Rect.Right := 20; Rect.Bottom := 41; Grafico.ZoomRect(Rect);. Uma maneira mais fácil de se fazer isso é extrair essas informações da própria série de interesse: Rect.Left := Serie.CalcXPosValue(22.5); Rect.Top := Serie.CalcYPosValue(5000); Rect.Right := LineSeries1.CalcXPosValue (57.6); Rect.Bottom := Serie.CalcYPosValue(15000); Chart1.ZoomRect(Rect);.

– **Imagem de Fundo**: Pode-se adicionar uma imagem de fundo para o gráfico.

☑ *Eventos de Clique*

A manipulação de eventos do TChart é muito interessante pois é comum que a aplicação dependa da interessão com o usuário para que de forma dinâmica um resultado qualquer (por exemplo um gráfico mais apropriado às necessidades do usário) seja alcansada. Segue a enumaração e descrição destes eventos:

- **Evento OnClickSeries do Gráfico:** O evento é associado a um procedimento onde os parâmetros passados por referência são:

* Sender: TCustomChart – O gráfico ao qual pertence a série que gerou o evento;

* Series: TChartSeries – A série que gerou o evento;

* ValueIndex: Longint – O indexador do ponto que foi clicado na série que gerou o evento;

* Button: TMouseButton – Qual dos botões do mouse gerou o evento;

* Shift: TShiftState – Qual era o estado da tecla Shift quando o evento foi gerado;

* X, Y: Integer – Os valores de X e Y convertidos em valores da série do ponto no gráfico em que a série foi clicada;

- **Eventos OnClick e OnDblClick da Série:** Os eventos são associados a um procedimento onde os parâmetros passados por referência são todos aqueles citados no evento OnClickSeries, exceto Series, visto que a série que responde a este evento já é determinada;

☑ *DESENHANDO DIRETAMENTE NO GRÁFICO*

A opção de se utilizar métodos para se desenhar diretamente no gráfico é extremamente útil quando se deseja adicionar funcionalidades ainda não implementadas ao componente em uso. Há métodos que facilitam esta tarefa e eles se dividem em dois diferentes objetivos:

– **Calculando Coordenadas:** Quando se está utilizando uma figura para representar um gráfico, está associada a esta tarefa uma função de transformação entre o domínio de valores de dados e o domínio de valores que possibilitam desenhar os pontos na tela, mantendo as proporções desejadas, isso nada mais é que uma relação linear entre o número de pixels disponível para o gráfico e o tamanho do intervalo de dados a ser representado. Assim sendo temos dois conjutos de métodos para realizar estas transformações: "Métodos de Valores de Eixo para Coordenadas de Tela" e "Métodos de Valores de Série para Coordenadas de Tela".

– **Métodos de Valores de Eixo para Coordenadas de Tela:**

* *CalcPosValue – Grafico.Eixo.CalcPosValue(LongInt):* retorna o indexador do pixel na direção do eixo equivalente ao valor LongInt;

* *CalcPosPoint – Grafico.Eixo.CalcPosPoint(Double):* retorna o valor do eixo equivalente ao Double-ésimo pixel. Obs.: A propriedade Grafico.ChartBounds contém o valore das coordenadas inicial e final do retângulo de contorno do gráfico.

* *CalcSizeValue – Grafico.Eixo.CalcSizeValue(LongInt):* retorna o número de pixels que um intervalo de tamanho LongInt ocupa em tela;

* *CalcYPosValue/CalcXPosValue – Grafico.Eixo.CalcXPosValue / CalcYPosValue (Double):* retorna o indexador do pixel referente ao valor Double.

– **Métodos de Valores de Série para Coordenadas de Tela:**

* *CalcPosValue – Serie.CalcPosValue(LongInt):* Exatamente o mesmo procedimento utilizado para o gráfico.

* *CalcXPos/CalcYPos – Serie.CalcXPos/CalcYPos(LongInt):* Exatamente o mesmo procedimento utilizado para o gráfico.

* *XScreenToValue/YScreenToValue – Serie.XScreenToValue/YScreenToValue (Double):* retorna o valor do eixo equivalente ao Double-ésimo pixel, métrica da tela.

☑ *TRABALHANDO COM OS EIXOS*

- **Ajustando a Escala:** As escalas são a forma de passagem da informação ao usuário, assim sendo são de suma importância num gráfico. Sempre que um novo ponto é adicionado a uma série que esteja sendo apresentada, os valores mínimo e máximo da escala de um eixo serão novamente recalculados(pode-se alterar isso desativando este reajuste automático: Grafico.Eixo.Automatic/AutomaticMaximum/AutomaticMinimum := True/False;). Os valores mínimo e máximo visualizados podem ser alterados a qualquer momento editando-se as propriedades Grafico.Minimum/Maximum.

- **Eixos de Data ou Hora:** Os ajustes para eixos utilizando este tipo de dados só distam por necessitar ter os valores convertidos: EncodeDate(aaaa, mm , dd).

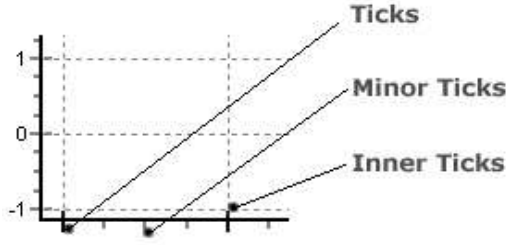
- **Eixos Logarítmicos:** Ao tentar setar a propriedade Eixo.Logarithmic de um gráfico, automaticamente é conferido se Eixo.Minimun e Eixo.Maximum são maiores ou iguais a zero.

- **Eixo Invertido:** Grafico.Eixo.Inverted é uma propriedade que não se aconselha usar pois é um boa fontes de erros por falta de atenção.

- **Estilo do Eixo:** Os eixos podem receber marcações especiais que facilitam a visualização dos valores dispostos. A maioria absoluta destas propriedades é acessível pelo Editor de Gráficos e é listada no Object Inspector.

- **Incrementos nos eixos:** A propriedade Grafico.Eixo.Increment determina a separação entre as marcações que serão dispostas nos eixos. Deve estar atento ao usar esta propriedade pois valores muito baixos para esta mesmas levam à superposição das marcações. Outra propriedade útil é Grafico.Eixo.RoundFirstLabel que determina se a primeira marcação será o menor valor ou então o maior inteiro menor que o menor valor. Quando usando valores do tipo data Grafico.Eixo.ExactDataTime determina se as marcações são exclusivamente múltiplos inteiros de datas ou não.

- **Linhas de Grid e Marcações:** As linhas de Grid facilitam a comparação dos valores dos pontos dispostos no gráfico. Com a propriedade Grafico.Eixo.TickOnLabelsOnly determina-se a obrigatoriedade ou não de só dispor linhas de Grid a partir de marcações de eixos. Na propriedade Grafico.Eixo.LabelStyle pode-se determinar que informação é apresentada ao lado de cada ponto disposto no gráfico.



☑ *MANIPULANDO SÉRIES:*

– **Criando séries em Tempo de Execução:** Para criar uma série em tempo de execução é necessário que se crie uma variável do tipo da série desenhada e que em tempo de execução execute-se um método para criar a série(Var VariavelSerie: TipodeSerie; VariavelSerie := TipodeSerie.Create(Self);), além disso é necessário que se associe um gráfico para a série em questão(VariavelSerie.ParentChart := Grafico;).

De forma análoga pode-se economizar código adicionando a série diretamente ao grafico(Grafico.AddSeries(TipodeSerie.Create(Self));). Há também uma forma de se criar séries quando não se sabe o tipo de série que será utilizando, fazendo referência à classe TChartSeriesClass, podendo assim herdar, em tempo de execução, as características de qualquer um dos tipos de série disponíveis(Var VariavelClasse: TChartSeriesClass; VariavelClasse := TipodeSerie; Grafico.AddSeries (VariavelClasse.Create(Self));).

– **Propriedades das Séries:** As séries tem características de vetores, assim o acesso a dados nelas contidas pode ser feito por indexação. Para se passar uma série inteira a uma variável, da mesma forma, basta utilizar indexação para referenciar a série desejada(NovaSerie := Grafico.SeriesList[n]; NovaSerie := Grafico.Series[n]; NovaSerie := Grafico[n];).

Quando se deseja saber o tamanho de uma série há funções que já realizam esta tarefa: Grafico.SeriesCount que conta o número de séries contido no gráfico e Serie.XValues.Count/Serie.YValues.Count que conta o número de elementos da série (lembrar que no Delphi os indexadores começam em 0). Existem 3 maneiras de se esconder uma determinada série: Serie.Active := False;/Serie.ParentChart := nil;/Serie.Free;

– **Adicionando Pontos:** Os métodos utilizados para adicionar pontos a uma série são:

* *Add/AddX/AddY(Double, String, TColor):* as variáveis de entrada são respectivamente: o valor a ser adicionado, o label associado e a cor de potagem;

* *AddArray(Array of Double):* um vetor de dados é entrado assumindo os label's associados como nulos e a cor de plotagem como a padrão da série;

* AddXY(Double, Double, String, TColor): onde, respectivamente, é feita a passagem de X e Y simultaneamente.

Com a propriedade Serie.Valores.Order[loNone/loAscending/loDescending] controla-se a ordem na qual novos valores são adicionados. Ao se reordenar uma série em função dos valores de Y ele somente troca a ordem dos pares (x,y), ou sejá é necessário que se crie a sequência dos X novamente: Serie.XValue.FillSequence; para refazer a sequência de inteiros para X ou então criar um procedimento que reconstrua os valores de X.

➢ Acesso à Porta Paralela

O acesso à porta paralela pode ser feito de diversas formas no Delphi, sendo que somente as mais simples (e práticas) delas serão abordadas neste curso. Os programadores acostumados com Assembly poderão se aventurar neste tema, bastando procurar alguma referência sobre o assunto nas literaturas disponíveis pela internet.

Neste curso abordaremos basicamente duas formas de acessar a porta paralela. A primeira delas é fazer uso de uma dll, e a segunda é usar o componente IOPort.

Inicialmente daremos uma breve introdução sobre a porta paralela de forma que o programador poderá usá-la com segurança.

A pinagem das portas paralelas é padrão, e segue o seguinte esquema:

13 1

25 14

O endereço da porta paralela, que deverá ser fornecido ao programa, deverá ser checado em cada máquina, e é usualmente $378.

As funções dos pinos são definidas a seguir:

- Output: Pinos 2 ao 9 (bit 0-7)
- Terra: Pinos 18 ao 25
- Input:

Bit	Pin	Name
3	15	Error
4	13	Select In
5	12	Paper Empty
6	10	Acknowledge
7	11	Busy

  - *Obs:* O pino 11 está internamente invertido!!! Assim, se for aplicado "1" a todos os pinos, ao ler a entrada obteremos 01111xxx. Portanto, será necessário negar o valor lido do pino 11. É importante também notar que os pinos 10 e 11 são trocados na leitura.

✗ *INPOUT32.DLL*

Para usar a Dll Inpout32.dll basta colocá-la no mesmo diretório do executável, e incluí-la no código fonte, usando a declaração:

```
external 'inpout32.dll';
```

Esta declaração informa ao Delphi que o arquivo externo será utilizado no programa. O exemplo abaixo exemplifica sua utilização:

Primeiro, inclua um label e um scrollbar em um formulário. Disponha-os da forma como

achar melhor. Em seguida, digite o seguinte código na seção "implementation" do programa.

```
function Out32(wAddr:word;bOut:byte):byte; stdcall;
external 'inpout32.dll';
```

O evento OnChange do ScrollBar deve ser o seguinte:

```
procedure TForm1.ScrollBar1Change(Sender: TObject);
var bWriteMe, bErr:byte;
begin
bWriteMe:=ScrollBar1.position;
Label1.caption:=inttostr(bWriteMe);
bErr:=(Out32($378,bWriteMe));
end;
```

Antes de rodar o programa, certifique-se que copiou a DLL para a pasta em que está o executável. Ao executá-lo, mova o ScrollBar e observe a alteração dos valores da porta paralela. Para observar a mudança dos valores da porta paralela, pode usar um voltímetro, ou um circuito com LEDs e série com resistores.

A função Out32 deve ser usada para conectar-se à DLL. A variável bWriteME e bErr pode receber o nome de sua preferência.

Caso apareça o erro "*Debugger kernal error. Error code:1*", verifique se a DLL se encontra no diretório correto, ou se você está tentando usar uma função que não está presente na DLL.

Em programas menores, indicar o endereço da porta paralela como feito anteriormente pode ser suficiente. No entanto, caso se deseje usar vários endereços, talvez seja melhor usar a declaração a seguir:

```
const PrinterPortAddr=$378;
```

Depois de declarar esta constante, sempre que desejar se referir à porta da impressora, informa a constante PrinterPortAddr.

Agora adicione um segundo Label ao seu formulário principal e um botão. No evento OnClick do botão, adicione o código:

```
label2.caption:='Valor da Porta '+ inttostr((Inp32($379)and $F8));
```

A expressão "and $F8" apenas esconde os 3 bits que não são entradas.

Em seguida, adicione a linha abaixo após a função Out32:

```
function Inp32(wAddr:word):byte; stdcall; external 'inpout32.dll';
```

Pronto, o programa pode ser executado. Note que o endereço $379 funcionará na maioria das máquinas, mas se sua porta paralela não está no endereço $378, use como endereço de input o endereço de sua porta + 1.

Recapitulando, deverá ser lido o endereço PortAddr + 1 para descobrir os valores dos pinos input. Para os demais pinos deverá se referenciar ao endereço PortAddr+2 para escrevê-los, pois são pinos de output. São eles:

Bit	*Pin*	*Name*
0	1	Strobe
1	14	Auto Linefeed
2	16	Initialise
3	17	Select Out

✗ *Componente IOPort*

O uso do componente IOPort é ainda mais simples que o uso de DLLs, uma vez que basta mudarmos suas propriedades para alterar o valor da porta desejado. A seguir segue o texto presente na ajuda deste componente, escrito por Eduardo Divino Dias Vilela:

O componente IOport foi implementado com a finalidade de facilitar o acesso aos ports do PC, primordialmente o Parallel Port - A Porta paralela. Este componente é muito simples de se usar, o que não menospreza a sua utilidade. Ele possui apenas 4 propriedades publicadas, as quais são:

- Name - Tipo:String
  O nome que designa o componente.

- PortAddress - Tipo: Word (0 a 65536 / $00 a $FFFF)
  Endereço do port que se deseja acessar.

- PortData - Tipo: Byte (0 a 255 / $00 a $FF)
  Dado de leitura/escrita no endereço especificado pela propriedade PortAddress.

- Tag - Tipo LongInt
  Propriedade de uso genérico de tipo inteiro longo.

E possui os seguintes métodos, dos quais todos, exceto dois, são procedures, ou seja, retornam valores apenas por parâmetros passados direta ou indiretamente. Os métodos tipo function retornam valor em si mesmo.

- **Procedures**
  (Implícitos: significa que os parâmetros sobre os quais o método será aplicado deverão ser definidos antes da chamada do método)

  - Write; Escreve o conteúdo da propriedade 'PortData' na porta indicada na propriedade 'PortAddress'.

  Exemplo

  ```
  Reset;
  ```

  Leva todos os 8 bits da porta indicada pela propriedade 'PortAddress' para 0 (zero).

  Exemplo

  ```
  Read;
  ```

  Lê o valor presente na porta indicada na propriedade 'PortAddress', e coloca este valor na propriedade 'PortData'.

Exemplo
(Explícitos: significa que os parâmetros sobre os quais o método será aplicado deverão ser definidos durante a chamada do método)

```
WritePort(PortValue, DataValue: word);
```

Escreve o conteúdo indicado pelo parâmetro 'DataValue' na porta indicada pelo parâmetro 'PortValue'.

Exemplo

```
ResetPort(PortValue: word);
```

Leva todos os 8 bits da porta indicada pelo parâmetro 'PortValue' para 0 (zero).

Exemplo
(Mistos: Combina parâmetros implícitos com explícitos)

```
SetBit(Bit: integer);
```

Leva para nível alto o estado do bit indicado pelo parâmetro 'Bit', bit este do endereço dado pela propriedade 'PortAddress'.

Exemplo

```
ClrBit(Bit: integer);
```

Leva para nível baixo o estado do bit indicado pelo parâmetro 'Bit', bit este do endereço dado pela propriedade 'PortAddress'.

Exemplo

```
InvBit(Bit: integer);
```

Inverte o estado do bit indicado pelo parâmetro 'Bit', bit este do endereço dado pela propriedade 'PortAddress'.

Exemplo
(retorna um valor que deve ser associado a uma variável do usuário)

```
Functions
   ReadPort(PortValue: word): word;
```

Retorna o valor da word lida no endereço indicado pelo parâmetro 'PortValue'.

Exemplo

```
GetBit(Bit: integer): Integer;
```

Retorna o estado do bit indicado pelo parâmetro 'Bit', bit este do endereço indicado pela propriedade 'PortAddress'.

- **Eventos**

Este componente não possui manipulação a eventos.

4. A Linguagem Object Pascal

Por mais recursos gráficos que as linguagens orientadas a objetos possuam, em determinado momento não há como fugir do *código*. A programação em Delphi é definida através da Linguagem Object Pascal, uma extensão do Pascal proposto por Niklaus Wirth.

Consideramos uma aplicação em Delphi baseada em um conjunto de arquivos, (citados anteriormente .DPR .PAS e .DFM) básicos. Vamos examinar alguns arquivos de fundamental importância:

➢ O Módulo .DPR

Todo programa em Object Pascal possui um arquivo .DPR, considerado como *arquivo de projeto*, o seu formato é composto inicialmente da seguinte definição:

```
program Project1;
uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

A palavra *program* define o nome do programa, este nome será alterado quando for gravado o arquivo .DPR do projeto. Na cláusula *uses*, são listadas as *units* usadas pelo módulo principal. As *units* (que serão vistas adiante) são responsáveis pela capacidade de dividir o programa em uma visão *modularizada*. Em cada um, declaramos uma série de objetos (funções, variáveis, procedimento, etc...) que podem ser usados por outras *units* e pelo módulo principal. Em seguida vem um conjunto de comandos (denominado *comando composto*) através de dois delimitadores ***begin*** *e* ***end***.

➢ As Units

Um programa em Object Pascal é constituído de um módulo principal (.DPR) e de uma ou mais unidades de compilação (.PAS). O compilador gera um arquivo com o código objeto correspondente, e considera o mesmo nome do arquivo .PAS com a extensão .DCU.

As *units* são entidades independentes, ou seja, no momento da criação não há vínculo lógico (nem físico) entre uma *unit* e um programa principal que a utiliza. Com esta característica, podemos utilizar as *units* em qualquer projeto. A principal característica do conceito de *unit* é que possibilita estruturar o programa em módulos funcionais, com cada *unit* provendo um conjunto de funções e procedimentos. Cada formulário corresponde a uma *unit,* mas podemos criar *units* independentes, não associadas a um *form*.

Se considerarmos o código uma *unit* com um componente *Button* e um manipulador de evento, teremos o seguinte código:

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs,  StdCtrls, ExtCtrls, ComCtrls;
type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.Caption := 'Curso de Delphi';
  Showmessage('Exemplo de caixa de diálogo');
end;
end.
```

Uma *unit* possui cinco partes:

- *CABEÇALHO*

Contém a palavra reservada *unit* seguida de um identificador que é o nome da *unit*. Este nome é o mesmo nome do arquivo com extensão .PAS

```
unit Unit1;
```

- *INTERFACE*

Contém tudo o que a *unit* exporta: constantes, tipos, variáveis, procedimentos, funções,

etc... Na declaração dos procedimentos e funções que a *unit* exporta, deve constar apenas o cabeçalho (nome e parâmetros). A declaração completa fica na parte da *implementação*.

✗ *IMPLEMENTAÇÃO*

Contém a definição completa das *funções* e *procedimentos* que constam na *interface*. Se na implementação são usados identificadores definidos em outra *unit*, o nome desta outra *unit* deve ser incluído na lista de *units* da cláusula *uses* da implementação.

```
implementation
{$R *.DFM}
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.Caption := 'Curso de Delphi – SENAC MG';
  Showmessage('Exemplo de caixa de diálogo');
end;
```

✗ *INICIALIZAÇÃO*

É uma parte opcional. Quando usada, não pode conter nenhuma declaração. Apenas comandos são permitidos nesta parte. Ela começa com a palavra *initialization*, e os comandos de inicialização são executados "antes" do programa começar.

```
initialization
  <comandos>
```

✗ *FINALIZAÇÃO*

É também uma parte opcional, com uma observação: ela só pode existir se na *unit* houver também uma parte de inicialização e só pode conter comandos, que serão executados dentro do processo de finalização do programa, após a execução do programas principal.

```
finalization
  <comandos>
```

Toda *unit* termina com a palavra ***end*** seguida de um ponto final ('**.**').

➢ **DECLARAÇÃO E MANIPULAÇÃO DE VARIÁVEIS**

As variáveis podem ser classificadas em:

Globais: Quando são feitas diretamente na seção *interface* de uma *unit* (ou seja, fora dos procedimentos e funções). Pode-se ter varíaveis *públicas* e *privadas*.

Locais: Quando é feita a declaração *dentro* de um procedimento ou função.

```
var
```

```
N: Single;
S: String;
I: Integer;
```

Tipos para manipulação de variáveis:

Tipos de variáveis Inteiras

Tipo	Faixa de Valores	Formato
Integer	-2147483648.. 2147483647	32 bits
Cardinal	0..4294967295	32 bits, sem sinal
Shortint	-128..127	8 bits
Smallint	-32768..32767	16
Longint	-2147483648.. 2147483647	32
Int64	-2^63..2^63-1	64
Byte	0..255	8 bits, sem sinal
Word	0..65535	16 bits, sem sinal
Longword	0..4294967295	32 bits, sem sinal

Tipos de números Reais

Tipo	Faixa de Valores
Real	2.9*10E-39..1.7*10E38
Single	1.5*10E-45..3.4*10E38
Double	5.0*10E-324..1.7*10E308
Extended	3.4*10E-4932..1.1*10E4932
Comp	-2*10E63+1..2*10E63-1
Currency	-9.22*10E14..9.22*10E14

Tipos de variáveis booleanas

Tipo	Faixa de Valores
Boolean	False ou True
ByteBool	*
WordBool	*
LongBool	*

Tipos de variáveis de caracteres

Tipo	Valores
Char	Permite armazenar um caractere ASCII.
ShortString	Permite armazenar uma cadeia de até 255 caracteres.
String	Permite armazenar uma cadeia 'ilimitada' de caracteres.

Tipo genérico (Variant): Objetos variant são essencialmente variáveis *sem tipo* podendo assumir diferentes tipos, automaticamente. Esta vantagem aparente tem a característica de ser ineficiente se utilizada indiscriminadamente. Recomenda-se que não sejam usadas, pois torna o código difícil de ser entendido, até mesmo pelo seu autor.

✗ *Atribuição*

Ao declarar uma variável, o compilador cuida de alocar na memória uma área que seja suficiente para armazenar qualquer dos valores definidos através do seu tipo. Os valores que podem ser atribuídos à variável são definidos através de um *comando de atribuição* que pode ser considerado da seguinte forma:

```
Variável := expressão;
```

✗ *Funções de Conversão e manipulação de variáveis*

Os objetos do Delphi para entrada e/ou exibição de dados utilizam propriedades do tipo String, as propriedades Text e Caption são bons exemplos disto. O problema ocorre quando tentamos realizar cálculos matemáticos com os dados que devem ser manipulados por estas propriedades.

Desta maneira precisamos de funções para converter dados String em tipos Inteiros ou Reais ou Datas, para então manipulá-los. As principais funções de conversão do Delphi são:

Função	Objetivo
StrToInt(const S: String)	Converte um dado String em tipo Inteiro.
IntToStr(value: Integer)	Converte um dado Inteiro em tipo String.
StrToFloat(const S: String)	Converte um dado String em tipo Ponto Flutuante.
FloatToStr(Value: Extended)	Converte um dado Ponto Flutuante em tipo String.
DateToStr(Date: TdateTime)	Converte um dado TDateTime em String.
DateTimeToStr(DateTime: TDateTime)	Converte um dado TDateTime em String.
StrToDate (const S: String)	Converte um dado String em tipo TDateTime.
StrToDateTime(const S: String)	Converte um dado String em tipo TDateTime
FormatFloat(const Format: string; Value: Extended): string	Permite formatar um tipo ponto flutuante retornando uma string. Edit2.Text := FormatFloat('###,###.00',soma); Sendo soma uma variável real.

O tipo TDateTime é internamente manipulado como tipo Ponto Flutuante.

✗ *Expressões Booleanas*

São expressões que retornam valor *booleano* (falso ou verdadeiro).

Operador	Operação
not	Negação
and	E lógico
or	OU lógico
xor	OU EXCLUSIVO lógico

O operador *not* é unário, por exemplo: **if *not*** (X > Z) **then ...**
Devemos usar parênteses ao compararmos expressões lógicas, por exemplo:

if (X > Z) **or** (W > Y) **then ...**

➢ **Estruturas de Decisão, Repetição e Seleção**

✗ *O comando IF*

O comando condicional ***if*** pode ser composto de uma ou mais condições de processamento, por exemplo:

- **if** (A > B) **then**
  B := B + 1; *// ou INC(B);*

- **if** (A > B) **then**
  B := B + 1

```
else
        A := A - 1; // ou DEC(A);
```

-
```
if (A > B) then
begin
        B := B + 1;
        X := B + A;
end
else begin
        A := A - 1;
        Y := Y + B;
end;
```

No último exemplo para representar um bloco de comandos em caso verdadeiro ou falso, utiliza-se dos delimitadores ***begin*** e ***end***.

O comando ***if-then-else*** é considerado como uma estrutura única, portanto, não há ponto e vírgula (‘**;**’)antes da palavra reservada **else**, mas somente no fim da estrutura condicional.

✗ *O comando Case*

O comando *case .. of* oferece uma alternativa para comandos ***if-then-else*** com um ‘grande’ número de testes. Por exemplo:

```
case Key of
        'A'..'z':          Label1.Caption := 'Letras';
        '0'..'9':          Label1.Caption := 'Números';
        '+', '-', '*', '/': Label1.Caption := 'Operador'
else
        Label1.Caption := 'Caracter especial';
end; //fim do case
```

✗ *O Comando Repeat*

O comando *repeat .. until* é uma opção para estruturas de repetição. A grande diferença com o comando *while* é o fato do comando *repeat* ser executado pelo menos uma vez, como veremos adiante.

```
repeat
        X := X + 1;
        INC(Z,3); //equivale a Z := Z + 3;
        // aqui poderia vir mais código ...
until X >= 200;
```

✗ *O Comando While*

O comando *while.. do* também permite a construção de estruturas de repetição, com diferença de efetuar o teste lógico antes de executar as instruções.

```
while X <= 200 do
```

```
begin
  X := X + 1;
  INC(Z,3);
  DEC(AUX,2);
end;
```

✗ O Comando For

O comando *for.. do* estabelece uma estrutura de repetição considerando um controle inicial e final. Pode ser construído de maneira crescente ou decrescente, usando-se os comando "to" e "downto", respectivamente.

```
for i:=0 to 500 do
  Label1.Caption := IntToStr(i);

for i:=500 downto 100 do
  begin
    Label1.Caption := IntToStr(i);
    Edit1.Caption := IntToStr(i);
  end;
```

Note que, no primeiro caso não é necessário usar "begin" e "end", pois eles deverão apenas ser usados quando deseja-se que mais de uma linha de comandos seja executada dentro do loop.

✗ O Comando Break

O comando *break* é usado para alterar o fluxo normal de comandos de repetição, o controle é desviado para o comando seguinte ao comando repetitivo.

```
frase := Edit1.Text;
for i:=1 to length(frase) do
  begin
    if frase[I] = 'A' then
      break;
    aux := aux + frase[I];
  end;
Label1.caption := aux; //Label1 recebe o conteudo de frase até a letra
'A'
```

✗ O Comando With

O comando *with.. do* é usado para abreviar a referência a campos de registro, ou a métodos, e *propriedades* de um objeto.

```
begin
  Form1.Caption := 'Pet';
  Form1.Color := ClBlue;
  Form1.Top := 95;
end;
```

```
//Equivalente à:
with Form1 do
  begin
    Caption := 'Pet';
    Color := ClBlue;
    Top := 95;
  end;
```

➢ **PROCEDURES E FUNÇÕES**

Procedimentos e funções são blocos de código (rotinas) em Object Pascal que podem ou não receber parâmetros (valores) para processamento. Uma vez definida a rotina pode-se chamá-la de diversas partes do programa através de seu nome.

A grande diferença entre as formas de definição destas rotinas (se procedimentos ou funções) está no fato de que:

- *Procedimento* – **NÃO** retorna valor.
- *Função* – Retorna valor.

✗ *DECLARAÇÃO E ATIVAÇÃO DE PROCEDURES*

Um procedimento pode ser declarado dentro da cláusula **private** ou **public.** Vejamos o exemplo abaixo:

```
procedure Soma(X, Y: String);
```

Com o cursor posicionado na mesma linha, pressione: **CTRL+SHIFT+C** e perceba que o próprio Delphi realiza a construção do procedimento dentro da cláusula **implementation**. Esse recurso é chamado **Class Completion.** Nossa tarefa é apenas definir o código a ser realizado pelo procedimento.

```
procedure TForm1.Soma(X, Y: String);
begin
  Label1.Caption := FloatToStr(StrToFloat(X)+StrToFloat(Y));
end;
```

Supondo a existência de dois componentes *Edit*, um componente *Button* e um componente *Label*, este código pode ser ativado da seguinte forma:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Soma(Edit1.Text, Edit2.Text);
end;
```

✗ *DECLARAÇÃO E ATIVAÇÃO DE FUNÇÕES*

A construção de funções tem o mesmo raciocínio, com a diferença que a função retorna um valor. Uma função pode ser declarada da seguinte maneira, dentro da cláusula **private** ou **public.**

```
function Subtrai(X, Y: String): String;
```

Observe que agora, depois dos parâmetros há um tipo de definição de retorno da função (no caso, uma String).

Pode-se utilizar a mesma dica de construção do procedimento, na linha da declaração tecle **CTRL+SHIFT+C.** Nossa tarefa é apenas definir o código a ser realizado pela função.

```
function TForm1.Subtrai(X, Y: String): String;
begin
      result := FloatToStr(StrToFloat(X)-StrToFloat(Y));
end;
```

A palavra reservada result é o recurso usado pela Object Pascal para estabelecer o retorno da rotina. Não se deve declarar esta variável, ela é declarada no momento da utilização da função.

Supondo a existência de dois componentes Edit, 'um' componente Button e um componente Label, esta função pode ser ativada da seguinte forma:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := Subtrai(Edit1.Text, Edit2.Text);
end;
```

Neste caso, o Label recebe o **result** de "subtrai", ou seja, a subtração dos dados passados nos parâmetros.

➢ Chamadas de Formulários

Uma característica importante da apresentação dos formulários em uma aplicação, é o fato de ser apresentado como **MODAL** ou **NÃO-MODAL**. Há dois métodos para executar a visualização, mas antes vamos entender como isso funciona.

- MODAL – O foco fica *preso* no formulário e não é liberado para outro form até que ele seja fechado. O usuário pode ativar outra aplicação do Windows, mas não poderá trabalhar em outra janela daquele programa cuja janela foi aberta como modal (até que seja fechada).
- NÃO MODAL – O foco pode ser transferido para outra janela sem que esta precise ser fechada.

Os métodos que o Delphi utiliza para apresentar os forms são: **Show** para apresentar forms NÃO-MODAIS, ou **ShowModal** para apresentar forms MODAIS.

5. Tratamento de Exceções no Delphi

Quando criamos e executamos nossos programas, estamos sujeitos à situações de ***erros*** em tempo de *execução*, a isto denominamos ***exceção***. As exceções devem ser tratadas de maneira a **não** permitir:

- Travar a aplicação
- Emitir mensagens 'técnicas' ao usuário leigo
- Deixar o Sistema Operacional instável

Quando uma exceção ocorre, o fluxo de controle é automaticamente transferido para blocos de código denominados *handlers*[6] de exceções, definidos através de comandos específicos do Object Pascal.

No Object Pascal, uma *exceção é uma classe*. A definição de exceções como classes permite agrupar exceções correlatas. Esse agrupamento é feito através da própria hierarquia de classes, de modo que podemos ter várias classes dependentes de uma única.

O que ativa o mecanismo de tratamento de erros através de exceções é o uso da unit *SysUtils*. Ela permite detectar os erros e convertê-los em exceções.

➢ *O Comando TRY-EXCEPT*

Podemos tratar as exceções através do comando *try-except*. Sua sintaxe:

```
try
      <comandos a serem executados>
except
      <bloco de exceção>
end; //finaliza o bloco
```

Os *comandos a serem executados* são tratados seqüencialmente na ordem em que foram criados, caso não haja alguma exceção o *bloco de exceção* é ignorado. O programa prossegue normalmente obedecendo aos eventos provocados pelo usuário.

Caso ocorra alguma exceção, o fluxo de controle é desviado para o *bloco de exceção.* É importante lembrar que podemos inserir qualquer comando, inclusive fazer chamadas a procedimentos e funções que por sua vez, podem chamar outros procedimentos e funções.

O *bloco de exceção* pode ser definido através de uma construção genérica, exemplo:

```
try
  Abre(Arq);
  while not Fim(Arq) do
    processa(Arq);
except
    Showmessage ('Houve um erro inesperado.');
```

[6] *Handler* = Manipulador de evento.

```
end; //bloco try
```

No exemplo acima tratamos os erros com uma mensagem genérica dentro de um um bloco try-except.

➢ A CONSTRUÇÃO ON-DO

```
try
  // aqui digitaríamos nosso código
except
  on EInOutError do //erro de entrada e saída
     begin
       Showmessage('Problemas...');
       // código para encerrar o que estávamos fazendo, etc...;
     end;
  on EdivByZero do //erro de divisão de nº inteiro por zero
     Showmessage('Erro ao dividir por zero');
  on EconvertError do //erro de conversão de tipos
     Showmessage('Erro de conversão de tipos de dados');
end; //bloco try
```

Podemos ainda definir utilizando a cláusula *on-do* com um *handler* genérico usando *else*, da seguinte forma:

```
try
  Processa;
except
  on Exceção1 do Trata1;
  on Exceção2 do Trata2;
  else TrataOutras;
end;
```

Os principais tipos de exceção da RTL (*RunTime Library*) do DELPHI, a serem tratadas nos blocos **on ... do** são:

Nome	Descrição
EaccessViolation	Ocorre quando se tenta acessar uma região de memória inválida (ex: tentar atribuir valor a um ponteiro cujo conteúdo é nil).
EconvertError	ocorre quando se tenta converter um string em um valor numérico (ex: utilizar a função StrToInt em uma letra).
EdivByZero	ocorre na divisão de um número **inteiro** por zero.
EinOutError	ocorre numa operação incorreta de I/O (ex: abrir um arquivo que não existe).
EintOverFlow	ocorre quando o resultado de um cálculo excedeu a capacidade do registrador alocado para ele (para variáveis inteiras).
EinvalidCast	ocorre quando se tenta realizar uma operação inválida com o operador as (ex: tentar usar um Sender com uma classe que não corresponde a seu tipo).
EinvalidOp	ocorre quando se detecta uma operação incorreta de ponto flutuante.

EinvalidPointer	ocorre quando se executa uma operação invalida com um ponteiro (ex: tentar liberar um ponteiro duas vezes).
EoutOfMemory	ocorre quando se tenta alocar memória mas já não existe mais espaço suficiente.
EoverFlow	ocorre quando o resultado de um cálculo excedeu a capacidade do registrador alocado para ele (para variáveis de ponto flutuante).
ErangeError	ocorre quando uma expressão excede os limites para a qual foi definida (ex: tentar atribuir 11 ao índice de um vetor que pode ir no máximo até 10).
EstackOverflow	ocorre quando o sistema não tem mais como alocar espaço de memória na Stack.
Eunderflow	ocorre quando o resultado de um cálculo é pequeno demais para ser representado como ponto flutuante.
EzeroDivide	ocorre quando se tenta dividir um valor **de ponto flutuante** por zero.

➢ O COMANDO TRY-FINALLY

Há um outro comando cuja sintaxe começa com ***try***. Este controle de finalização nos permite lidar de forma estruturada com as situações em que alocamos algum tipo de recurso e, *haja o que houver*, precisamos depois liberá-lo.

```
  <aloca o recurso>
try
  <usa o recurso>
finally
  <libera o recurso com ou sem exceção>
end;
```

O comando funciona da seguinte forma: os comandos especificados após o *Try* são executados seqüencialmente. Se não ocorrer nenhuma exceção, os comandos especificados após *finally* **são** executados, e o programa prossegue com a execução normal, com o comando seguinte ao *try-finally*. Porém, se houver alguma exceção – qualquer uma – durante a execução da lista de comandos do *try*, este é interrompido e o trecho após o *finally* é executado e, no final, a exceção é reativada.

➢ TRATAMENTO DE EXCEÇÕES DE FORMA GLOBAL

Há no Delphi um objeto chamado *Application* criado sem a decisão do desenvolvedor em todo o projeto. Este objeto representa a applicação como um todo e possui um evento muito importante: *OnException*. Este evento permite manipular as exceções em um nível global, podemos afirmar que os tratamentos de erro através do comando *try* são tratamentos *locais*.

Como o objeto tem como finalidade generalizar e centralizar tratamentos, deve haver um único objeto na aplicação.

A utilização do evento OnException pode ser criado da seguinte forma; utilização um **if** na variável **Erro** (que recebe o erro atual) tomando uma decisão na condição verdadeira:

```
procedure TForm1.ApplicationEvents1Exception(Sender: TObject;
  Erro: Exception);
begin
   if Erro is EConvertError then
     ShowMessage('Erro de conversão de dados.');
   // outras condições...
end;
```

Neste exemplo acima, em *qualquer lugar do programa* (e não apenas em uma determinada rotina) que venha a levantar um erro do tipo EConvertError, uma mensagem genérica será exibida. Este objeto deve estar inserido ou no formulário principal ou no formulário especial denominado Data Module como veremos adiante.

➢ Tratamento de exceções silenciosas

Podemos utilizar o comando *Abort* para gerar exceções silenciosas, ou seja, sem nenhuma mensagem.

```
try
  Form1.Caption :=
               FloatToStr(StrToFloat(Edit1.Text) / StrToFloat
(Edit2.Text));
except
  on EZeroDivide do
    begin
      Beep;
      ShowMessage('Divisão por zero');
    end;
  on EInvalidOp do ShowMessage('Operação inválida');
  else
    Abort;
end;
```

6. Banco de Dados no Delphi

Na maioria dos casos, as aplicações que vamos construir com o Delphi, ou outra ferramenta visual, deve manipular um banco de dados.

Um banco de dados pode ser visto de diferentes perspectivas; pode ser um arquivo ou pode ser um diretório com vários arquivos. Vamos entender essas diferenças um pouco mais adiante.

➢ Modelagem Básica

Se você já tem alguma experiência com a *teoria* de banco de dados, tenha paciência, vamos estudar *rapidamente* neste tópico alguns conceitos básicos que independem da plataforma no desenvolvimento de banco de dados.

- O *Modelo Conceitual* procura abstrair a realidade independente da plataforma de harware ou software, é uma visão global na construção da aplicação.
- O *Modelo Lógico* define as regras básicas e quais os dados devem ser armazenados no banco e depende das características do software, é também dependente do modelo conceitual.
- O *Modelo Físico* implementa as definições do modelo lógico na 'prática', ou seja, desenvolver o projeto definido anteriormente, depende dos recursos de software e hardware.

✗ *Modelo Conceitual e Lógico*

A princípio vamos entender como podemos definir os dados que serão armazenados em um computador, através do conceito de **entidade**.

Ao pensarmos em cadastrar dados de clientes, alunos, fornecedores, etc... temos exemplos de entidades. A entidade possui propriedades que serão identificados como os dados propriamente ditos, estas propriedades são chamadas de **atributos**[7], ou seja:

Entidade	Alunos
Atributos	AluCodigo
	AluNome
	AluDataNasc

Entidade	Funcionario
Atributos	FunCodigo
	FunCPF
	FunTelefone

✗ *Modelo Físico*

No modelo físico a **entidade** se chama ***tabela*** e os **atributos** se chamam ***campos***. A linha de dados que deriva do conjunto de campos se chama **registro**. Exemplo:

Tabela: Funcionario		
FunCodigo	FunNome	FunDataNasc
1	Edileumar Jamaica	01/01/1975
2	Jorgina Alambradus	25/04/1979

[7] Os atributos devem ter um tipo de dados definido (Inteiro, AlfaNumérico, Data, etc..)

É necessário definirmos nas entidades (tabelas) um *campo* que seja **idenficador** de cada *registro* ou seja, um dado que **não** deve repetir, identificando aquele registro como único. Exemplo: FunCodigo. Na prática este campo poderia ser um CPF ou RG, por exemplo.

Esse campo recebe o nome de **chave primária** porque ele é capaz de identificar aquele determinado registro como único, além disso, a tabela começa a ser ordenada pelos dados deste campo.

➢ RELACIONAMENTOS

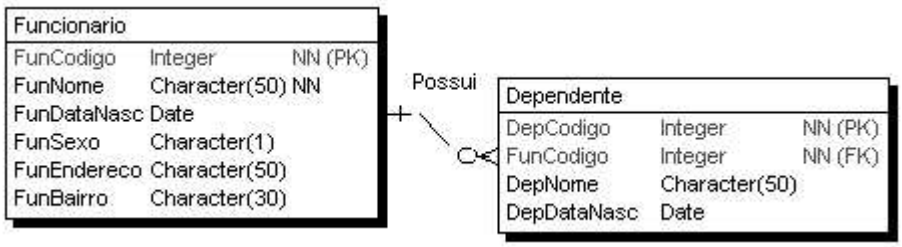
O conceito: "Banco de Dados Relacional" significa que as tabelas que compõem o banco "relacionam-se entre si". Um exemplo: A tabela Funcionarios tem um relacionamento com a tabela Dependentes.

Relacionamento é 'uma regra de associação entre duas ou mais entidades'.

O relacionamento pode ser representado através de *cardinalidades*, onde são definidas restrições que as entidades devem obedecer. Existem três tipos básicos de relacionamentos: **1:1** , **1:N** ou **N:N**.

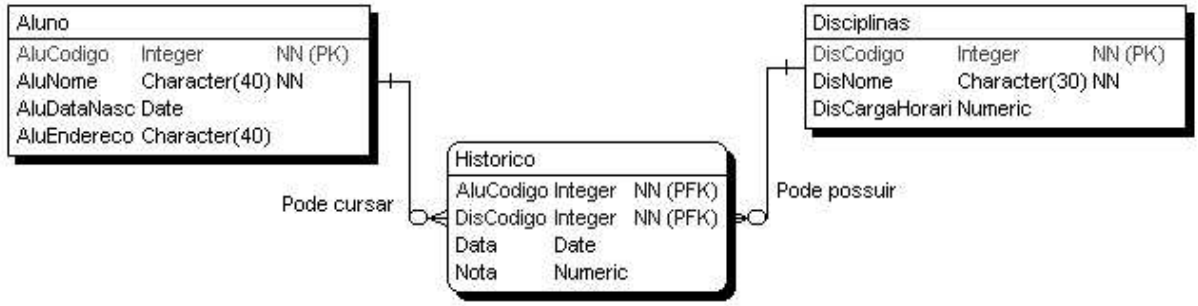
Existem regras a seguir, por exemplo: no relacionamento **1:N** ou **N:1** (este é o relacionamento que é mais utilizado), a chave primária (Primary Key – PK) da entidade **1** deve aparecer como **chave estrangeira** (Foreign Key – FK) na entidade **N**.

Exemplo de uma notação em um software CASE:



No caso de um relacionamento **N:N** deverá ser criada uma 3º tabela onde haverá no *mínimo* as chaves primárias das entidades relacionadas. Estes atributos devem fazer parte da chave primária desta nova tabela e funcionar como chave estrangeira da tabela onde se originou o relacionamento.

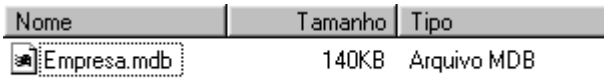
Exemplo de uma notação em um software CASE:



➢ **VISÃO FÍSICA DO BANCO DE DADOS**

O banco de dados pode ser visto (**fisicamente**) como um arquivo ou como um diretório (pasta), isso dependerá da estrutura que o fabricante do banco de dados vai abordar, exemplo:

No produto Microsoft *Access* o banco de dados é representado fisicamente como um **único arquivo** (.MDB) contendo as tabelas e demais recursos. Para o Delphi o banco de dados é representado pelo arquivo .MDB



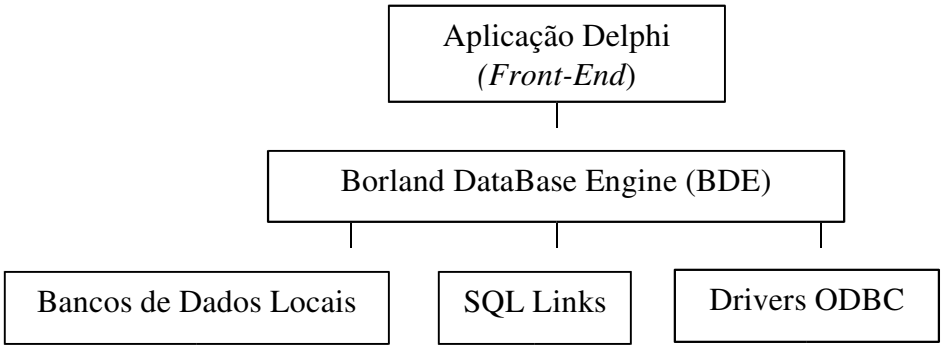
Para tabelas do banco de dados *Paradox* cada tabela dará origem a um arquivo .DB e arquivos separados para índices (.PX) entre outros recursos. No exemplo abaixo, a tabela (.DB) contém campos BLOB (figuras ou memorandos), estes dados serão armazenados em um arquivo .MB Neste caso, para o Delphi, o banco de dados é o **diretório** (pasta) onde se encontram os arquivos que compõem o banco de dados.

➢ **CONEXÃO AO BANCO DE DADOS**

O Delphi utiliza uma estrutura de camadas para fazer com que o *front-end* (formulário) manipulado pelo usuário venha interagir com a *base de dados*. O caminho deve ser percorrido por uma série de componentes configurados entre si, porém, há uma camada intermediária que não fica *dentro* do ambiente Delphi, nem diretamente preso ao *banco*, é o BDE.

✗ *BDE*

O BDE é um conjunto de DLLs que deve acompanhar as aplicações que fazem uso de seus recursos de acesso ao banco. É nesta camada que são definidas características específicas de cada banco de dados, bem como sua localização, ou seja, o *front-end* não acessa diretamente a *base de dados*, o BDE é responsável para estabelecer este funcionamento.



Existem outras maneiras de acessarmos banco de dados SEM O BDE, a paleta InterBase disponível a partir do Delphi 5 é um bom exemplo.

✗ *COMPONENTES DE CONTROLE DE ACESSO*

O sistema para conexão com o banco de dados utiliza além do BDE um conjunto de componentes denominados: Session, DataBase, *DataSet*, *DataSource* e *Data-Aware*. O componente Session não será o foco de nosso estudo.

Uma visão 'geral' sobre estes componentes pode ser vista da seguinte maneira:

- ***Session***: Aplicações simples, trabalham com apenas um banco de dados. Porém o Delphi permite mais de uma conexão simultânea à bancos de dados distintos, e também mais de uma conexão com o mesmo banco de dados. O controle *global* das conexões é feito através do componente da classe TSession, criado *automaticamente* pelo Delphi na execução do programa. Esse componente representa a sessão *default* da aplicação.

- ***DataBase***: O componente DataBase é responsável pela conexão da aplicação a um banco de dados com a finalidade maior de implementar segurança (*transações*) *e* definir características de comunicação entre uma aplicação Delphi-Client/Server. Embora em aplicações locais, sua utilização *explicita* é recomendada.

- ***DataSet***: Existem três componentes que descendem de uma classe chamada TDataSet e implementam importantes métodos de manipulação de banco de dados além de suas características específicas. De fato, não utilizamos a classe TDataSet diretamente, mas através dos componentes TTable, TQuery(SQL) e TStoreProc. Estes componentes estão na paleta DataAccess

- ***DataSource***: O componente DataSource é responsável por conectar os componentes Data-Aware à uma determinada tabela representada pelo DataSet. Este componente está na paleta DataAccess.

- ***Data-Aware***: Os componentes Data-Aware são responsáveis pela visualização e manipulação direta dos dados. Todo componente Data-Aware tem uma propriedade para conectar-se ao DataSource correspondente à tabela destino. Estes componentes estão na paleta DataControls.

➢ EXEMPLO

Vamos exemplificar a utilização de componentes básicos de *acesso* e *controle* através de um exemplo baseado em uma tabela (arquivo de dados) já pronta, criada na instalação do Delphi. O objetivo é entendermos o funcionamento destes componentes.

1. Crie uma nova aplicação e salve-a na pasta especificada pelo instrutor.

O Formulário: UFrmPeixes
Projeto: Peixes

2. Insira dois componentes: Um DataSource e um Table. Configure suas propriedades de acordo com a orientação abaixo:

```
object TbPeixes: TTable
  DatabaseName = 'DBDEMOS'
  TableName = 'biolife.db'
  Active = True
  Name = TbPeixes
end
object DsPeixes: TDataSource
  AutoEdit = False
  DataSet = TbPeixes
  Name = DsPeixes
end
```

3. Insira um componente DBGrid e configure-o:

```
object DBGrid1: TDBGrid
  Align = alBottom
  DataSource = DsPeixes
end
```

4. Insira um componente DBImage e configure-o:

```
object DBImage1: TDBImage
  DataField = 'Graphic'
  DataSource = DsPeixes
  Stretch = True
end
```

5. Insira um componente DBMemo e configure-o:

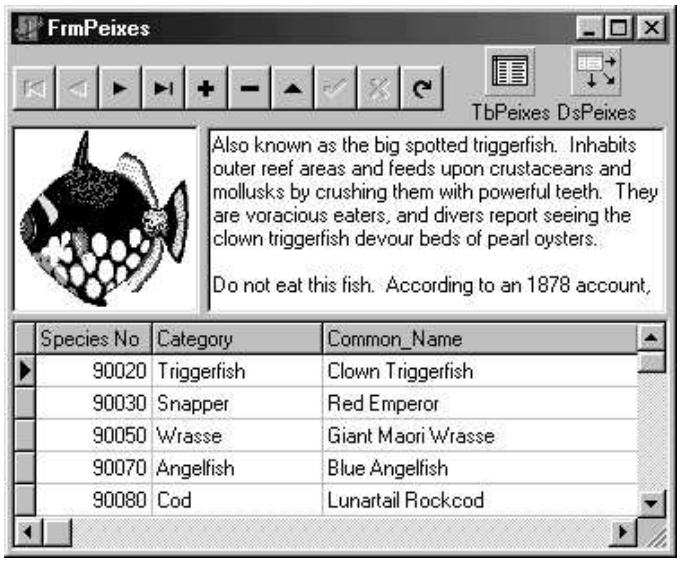
```
object DBMemo1: TDBMemo
  DataSource = DsPeixes
  DataField = 'Notes'
end
```

6. Insira um componente DBNavigator e configure-o:

```
object DBNavigator1: TDBNavigator
  DataSource = DsPeixes
end
```

Uma sugestão do visual pode ser a seguinte:

7. Salve novamente e execute a aplicação.



Podemos perceber que a aplicação funciona manipulando a tabela ´biolife.DB´ sem a necessidade de nenhuma linha de código explícita.

É claro que este é apenas um exemplo para entendermos o mecanismo de componentes de acesso e controle. Desse modo, veremos a seguir que linhas de código serão necessárias para implementar funcionalidades específicas (pesquisas), ou mesmo básicas como inclusão, edição e remoção.

Por último, vamos personalizar o componente *DBGrid* para *não* exibir os campos *Grafic* e *Notes* (campos BLOB).

Pode-se usar a propriedade **Columns** ou um duplo clique no componente *DBGrid*, O Editor de colunas é exibido, clique no ícone **Add All Fields;** os campos disponíveis serão exibidos, selecione o campo desejado (Grafic) e clique em **Delete Selected.**

Maiores informações sobre bancos de dados, assim como o exemplo desenvolvido no curso poderão ser obtidos diretamente com os organizadores do mesmo.

7. Noções Básicas de Depuração no Delphi

Porque pouquíssima importância que é dada à depuração nos cursos de informática, apesar desta ser uma importante tarefa para os programadores em geral? A resposta é muito simples: a maioria dos programadores comuns (não estudantes de engenharia) trabalha com tarefas onde pouca lógica é exigida, e rotinas maceteadas são comuns. No entanto, para nós, estudantes de engenharia, o conhecimento destras ferramentas permite descobrir erros muito difíceis de serem encontrados: os erros de lógica.

Muitas vezes, quando o programa que estamos desenvolvendo se torna grande, repleto de cálculos, não obtemos o resultado esperado, apesar de aparentemente não haver erro algum com o programa. Nestes casos, devemos dispor de recursos, como observar o valor de determinadas variáveis ao longo do tempo, acompanhar a execução do programa, linha a linha, etc.

O Delphi oferece poderosas ferramentas de depuração, que serão rapidamente explicadas no decorrer deste capítulo.

- **Menu View --> Debug Windows:**
  - Breakpoints: esta opção é usada em diversos momentos, e é usada para interromper o fluxo de programa no ponto em que é colocado. No entanto, esta ferramenta fornece muitas opções, não sendo restrita à interrupção do fluxo do programa. Clicando com o botão direito do mouse na janela que aparecerá, podemos escolher "Add". No formulário seguinte, temos as seguintes opções:

  - Filename: especifica o arquivo fonte onde o breakpoint será inserido.

  - Line number: linha em que será inserido o breakpoint.

  - Condition: especifica uma expressão condicional que é validada toda vez que o o breakpoint é encontrado. Se a expressão retornar um valor verdadeiro, o programa pára.

  - Pass count: nesta opção podemos especificar o número n de vezes que desejamos que o programa passe pela linha especificada, antes de parar. Esta opção é útil para sabermos em qual iteração de um loop ocorre o erro. Desta forma poderíamos atribuir a "pass count" o número de iterações do loop em questão, e quando o erro ocorrer, basta verificar o número de passagens ocorridas pelo breakpoint.

  - Group: outra opção do comando breakpoint é a possibilidade de criar grupos de breakpoints, que podem ser habilitados ou desabilitados simultaneamente. Esta opção é útil para programas maiores e mais complexos.

  - Botão Advanced: marcando esta opção, o usuário poderá ainda escolher ações avançadas a serem executadas, dentre as quais destacamos;

    - Break: ação tradicional e default. Simplesmente pára a execução do programa no ponto especificado.

    - Ignore subsequent exceptions: se marcado, todas as exceções geradas por erros serão ignoradas, e o programa não parará em nenhuma delas. Esta opção é muito útilo quando usada em paralelo com a opção "Handle

subsequent exceptions", explicada a seguir.

- Handle subsequent exceptions: com esta opção marcada, a execução só se interromperá se as exceções forem do tipo especificadas no formulário Tools|Debuger options. Maiores detalhes desta opção podem ser obtidos na ajuda do Delphi.

Ao invés de descrever todos os menus e itens relacionados ao assunto, faremos aqui um guia de referência rápida, que poderá ser consultado facilmente. Todas as opções listadas a seguir estão disponíveis na opção "Debug" do menu que aparece ao se clicar com o botão direito do mouse no texto de uma *unit*, e no menu Run, na barra de menus principal do programa.

- Toggle Breakpoint: Adiciona um breakpoint padrão na linha clicada.
- Run to Cursor: executa o programa até que encontre a linha clicada.
- Evaluate/Modify: útil para calcular valores que uma expressão assume de acordo com os valores das variáveis envolvidas.
- Add Watch: Adiciona um Watch para a variável selecionada. Esta opção é importante para se acompanhar o comportamento das variáveis do programa, evitando overflows, etc...
- Run: executa o programa.
- Step over: executa linha a linha, não exibindo operações internas a funções que estejam sendo executadas.
- Trace Into: similar ao Step Over, com a diferença que todas as linhas que estão sendo executadas são exibidas.
- Run until Return: roda o programa até que a execução retorne ao ponto atual.
- Show Execution Point: mostra a linha que está sendo executada.

Ainda restam muitas opções a serem exploradas, mas o objetivo deste capítulo, de forma alguma, é ser uma referência completa sobre o assunto. O usuário interessado encontrará uma extensa descrição de cada tópico na ajuda do Delphi.

8. ANEXOS

Este capítulo consiste de dicas e assuntos interessantes, abordados de forma direta e fácil. A intenção deste capítulo é que esteja em constante construção, visando sempre melhorar o teor desta apostila.

➢ INPUT / OUTPUT NO DELPHI

Essas funções são aplicáveis a qualquer variável do tipo File. Para se poder realizar qualquer operação sobre uma variável desse tipo, antes da primeira operação, deve-se associar um arquivo externo à variável através da função *AssignFile*. A função *Reset* prepara um arquivo no modo somente-leitura. Já os funçãos *Rewrite* e *Append* preparam um arquivo no modo somente-escrita.

Os arquivos são tratados como conjuntos seqüenciais de dados, ou seja, seu acesso é feito de forma indexada, como qualquer outro vetor. É importante lembrar que os indexadores são iniciados em zero. Outro fato importante é que sempre que se executa uns das funções *Write* ou *Read* o indexador de controle do arquivo é automaticamente incrementado para a posição seguinte no arquivo. Assim então as funções *FilePosition* e *FileSize* são úteis para se nortear ao percorrer um arquivo, enquanto o função *Seek* retorna o indexador para a posição na qual o determinado parâmetro de busca é encontrado.

A atualização da informação contida no arquivo que esteja sendo editado só é feita após a execução da função *CloseFile*.

Sempre que a manipulação de arquivos é feita, executa-se automaticamente uma verificação de erros, que pode ser habilitada ou desabilitada pelas diretivas {+I} e {-I}, respectivamente. Quando esta verificação esta desabilitava ainda há a possibilidade de se verificar a marcação de erros a partir da função *IOResult*.

✗ *ARQUIVOS DE TEXTO:*

Procedure or function	**Description**
Append	Opens an existing text file for appending.
AssignFile	Assigns the name of an external file to a file variable.
BlockRead	Reads one or more records from an untyped file.
BlockWrite	Writes one or more records into an untyped file.
ChDir	Changes the current directory.
CloseFile	Closes an open file.
Eof	Returns the end-of-file status of a file.
Eoln	Returns the end-of-line status of a text file.
Erase	Erases an external file.
FilePos	Returns the current file position of a typed or untyped file.
FileSize	Returns the current size of a file; not used for text files.
Flush	Flushes the buffer of an output text file.
GetDir	Returns the current directory of a specified drive.
IOResult	Returns an integer value that is the status of the last I/O function performed.
MkDir	Creates a subdirectory.
Read	Reads one or more values from a file into one or more variables.
Readln	Does what Read does and then skips to beginning of next line in the text file.

Rename	Renames an external file.
Reset	Opens an existing file.
Rewrite	Creates and opens a new file.
RmDir	Removes an empty subdirectory.
Seek	Moves the current position of a typed or untyped file to a specified component. Not used with text files.
SeekEof	Returns the end-of-file status of a text file.
SeekEoln	Returns the end-of-line status of a text file.
SetTextBuf	Assigns an I/O buffer to a text file.
Truncate	Truncates a typed or untyped file at the current file position.
Write	Writes one or more values to a file.
Writeln	Does the same as Write, and then writes an end-of-line marker to the text file