

BORLAND DELPHI®



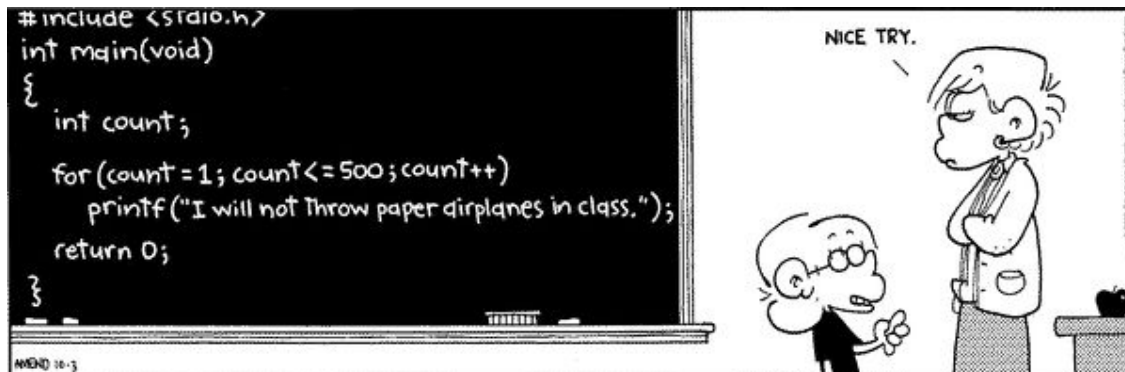
ESCLARECIMENTOS

Parte desta apostila foi criada utilizando-se partes de apostilas encontradas na Internet. Qualquer problema, por favor, contatar o professor para correção.

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

1. Cantu, Marco, "Dominando o Delphi 7 – A Bíblia", São Paulo, Makron Books, 2003
2. Cortes, Pedro Luiz, "Trabalhando com Banco de Dados Utilizando o Delphi", Érica, 2005

O curso parte do pressuposto que a pessoa é iniciante em Delphi, mas que conhece Lógica de Programação.



ÍNDICE

Índice	2
Um Pouco Sobre Delphi E Programação Para Windows	4
Princípios Da Programação Para Windows	5
Component Palette (Paleta De Componentes)	6
Conceito De Programação Orientada A Objeto	11
Orientação A Objeto	11
Evolução Das Linguagens Para Oo	13
Benefícios Da Oop	13
Linguagens Procedurais X Orientada A Objetos :	13
Alguns Exemplos De Linguagens Procedurais Com Extensões Oo:	14
Linguagens Compiladas X Interpretadas :	14
Eventos, Métodos E Propriedades	14
Eventos	14
Propriedades	15
Métodos	16
Object Pascal	16
Palavras Reservadas	17
Variáveis	18
Arrays (Vetores)	19
Records (Registros)	19
Principais Teclas De Atalho Da Ide Do Delphi	20
Como É Formada Uma Aplicação Em Delphi	20
Arquivos Que Compõe Uma Aplicação Delphi.	21
Arquivos Gerados No Desenvolvimento	21
Arquivos Gerados Pela Compilação	22
Código Fonte Do Arquivo Project(.Dpr)	22
Estrutura Da Unit De Um Form (.Pas)	23
O Parâmetro "Sender" E Os Operadores "Is" E "As"	24
O Operador "Is"	24
O Operador "As"	24
O Parâmetro Sender	25
Um Exemplo Completo Com Os Operadores "As" E "Is"	26
Eventos Mais Utilizados No Delphi:	28
Oncreate	28
Onkeypress	28
Onkeydown	28
Onclick	29
Ondestroy	29
Ondblclick	29
Onmousemove	30
Onenter	30
Onexit	30
Associando Um Procedimento A Um Evento Em Tempo De Execução	30
A Classe Twincontrol	31
Principais Propriedades	31
Principais Métodos	31
Acessando Diversos Componentes De Um Twincontrol	31
Exemplo 1	31
Exemplo 2	33
Exemplo 3	34
Alguns Componentes Visuais Do Delphi 2005	35
Principais Eventos Do Formulário	35
Palheta Standard	35
Manipulando Data E Hora No Delphi	37
Aplicações Robustas - Tratamento De Exceções	39
Exceções	40
Blocos Protegidos	40

Principais Exceções	41
Blocos De Finalização	41
Geração De Exceções	42
Exibindo O Erro Gerado Pela Exceção	42
Exceções Silenciosas	43
Manipulando Exceções Globais	43
Entendendo Um Pouco Sobre O Que É Um Gpf	44
Configurando O Delphi Para Não "Parar" Em Exceções	44
Conclusão:	44
Arquivos Ini	45
Criando Uma Biblioteca De Procedures, Funções, Etc.	47
Trabalhando Com Formulários	49
Principais Tecnologias De Acesso A Dados	51
Bde (Borland Database Engine):	51
Dbexpress (Dbx):	51
Ado (Activex Data Objects):	51
"Interbase Express">Interbase Express (Ibx):	52
Configurando Uma Conexão Com Bd Usando Ado No Borland Delphi 2005	53
Criando Um Cadastro Simples (Apenas 1 Tabela)	56
Validando Os Dados Do Cadastro	58
Criando Um Cadastro Com Mais De Uma Tabela (Fk)	59
Sugerindo Um Código Automaticamente	60
Solicitando Um Dado Ao Usuário (Inputbox)	61
Cadastros Sem O Tdbnavigator	62
Uma Tela Criada Sem O Tdbnavigator	63
Construindo Uma Tela De Consulta	65
Passando Parâmetros Para O Objeto Adoquery	66
Datamodule	66
Continuando Com A Construção Da Tela De Consulta	67
O Método Locate Da Classe Tdataset	67
Criação Da Tela De Consulta – Segunda Alternativa	69
Verificando Se Um Registro Existe Em Uma Tabela	71
Exibindo Os Erros Em Uma Tela Separada	72
Trabalhando Com Vetores Dinâmicos	74
Transações	77
Filtrando Dados De Uma Tabela	80
Frames	81
Objeto Tfield	85
Acessando Os Dados De Um Tfield:	87
Fields Editor	88
Validando Através Do Evento Onvalidade Dos Objetos Tfield	89
Campos Calculados	90
Campos Lookup	92
O Evento Ongettext	94
O Evento Onsettext	95
Tabelas Temporárias Com O Tclientdataset	95
Formulários Padrões	97
Criando Um Cadastro Master - Detail	99
Instalando E Criando Relatórios No Fortes Report Para Delphi 2006	110
Criando Threads No Delphi 2006	116
Instalação Do Sql Server 2000 – Versão Para Desktop	127
Apêndice - Funções Para Trabalhar Com Strings (Retiradas Do Help Do Delphi 7)	136
Apêndice - Object Pascal: Estruturas Básicas	139
Apêndice - Relações Entre Classes	147
Apêndice - Descrição Dos Principais Procedimentos E Funções Pré Definidas	148
Apêndice - Tecnologias Aplicáveis Ao Delphi 7	169

UM POUCO SOBRE DELPHI E PROGRAMAÇÃO PARA WINDOWS

No início, programar em Windows era algo extremamente complicado e acessível apenas a programadores dispostos a investir muito tempo e fofatos na leitura de pilhas de livros, intermináveis testes e análise de programas exemplos que mais confundem do que explicam. Mas porque era tão difícil fazer programas para Windows? Para começar, o Windows usa o conceito de GUI (Graphic User Interface – Interface Gráfica com o Usuário), que embora fosse muito familiar para usuários do Unix e do Mac OS, era novidade para usuários do DOS.

O uso de um sistema GUI implicava em aprender vários conceitos que eram estranhos ao usuário de um sistema baseado em texto como o DOS. Para complicar um pouco mais, o Windows é um sistema multi-tarefa, e as aplicações são orientadas a eventos, o que implica em aprender um novo estilo de programação. Finalmente, o programador tinha que ter alguma familiaridade com as centenas de funções oferecidas pela API do Windows. Por tudo isso, programação em Windows era um assunto que costuma provocar arrepios nos programadores.

Felizmente as linguagens visuais chegaram para mudar esta situação. Foi só com elas que o Windows conseguiu cumprir sua promessa de ser um sistema amigável e fácil de usar também para os programadores, que sempre tiveram que pagar a conta da facilidade de uso para o usuário.

Entre as linguagens visuais que surgiram, nenhuma veio tão completa e bem acabada quanto o Delphi. Desde o início ele possuía um compilador capaz de gerar código diretamente executável pelo Windows, proporcionando uma velocidade de execução de 5 a 20 vezes maior que as linguagens interpretadas como o Visual Basic e Visual FoxPro que geravam executáveis Pcode que precisam de arquivos auxiliares de run-time. Além disso, o Delphi também possuía uma engine para acesso a diversos bancos de dados e um gerador de relatórios.

O tempo de desenvolvimento de qualquer sistema foi reduzido a uma fração do tempo que seria necessário usando outras linguagens e o resultado é sempre muito melhor. É por isso que o Delphi fez e faz tanto sucesso no mundo inteiro, sempre ganhando prêmios como melhor ferramenta de desenvolvimento para Windows.

O objetivo principal de qualquer ferramenta de desenvolvimento ou linguagem de programação é a criação de aplicações. Determinadas linguagens ou ferramentas devido aos recursos que possuem são mais indicadas para a criação de aplicações comerciais, outras se destinam mais a aplicações científicas ou ainda para a criação de sistemas operacionais.

O Delphi é uma ferramenta RAD (Rapid Application Development – Desenvolvimento Rápido de Aplicações) criada pela Borland. É uma ferramenta de propósito geral, permitindo o desenvolvimento de aplicações tanto científicas como comerciais com a mesma facilidade e alto desempenho.

Integra-se facilmente com a API (Application Program Interface) do Windows, permitindo a criação de programas que explorem ao máximo os seus recursos, assim como os programas escritos em linguagem C/C++.

Possui um compilador extremamente rápido, que gera executáveis nativos (em código de máquina, não interpretado), obtendo assim melhor performance e total proteção do código fonte.

O Delphi é extensível, sua IDE (*Integrated Development Environment – Ambiente de Desenvolvimento Integrado*) pode ser ampliada e personalizada com a adição de componentes e ferramentas criadas utilizando-se o Object Pascal, a linguagem de programação do Delphi. Neste ambiente constroem-se as janelas das aplicações de maneira visual, ou seja, arrastando e soltando componentes que irão compor a interface com o usuário.

O Object Pascal é uma poderosa linguagem Orientada a Objeto, que além de possuir as características tradicionais das mesmas como classes e objetos, também possui interfaces (semelhantes às encontradas em COM e Java), tratamento de exceção, programação multithreaded e algumas características não encontradas nem mesmo em C++, como RTTI (Runtime Type Information). Assim como o C++, o Object Pascal é uma linguagem híbrida, pois além da orientação a objeto possui também uma parte da antiga linguagem estruturada (Pascal)

Devido ao projeto inicial da arquitetura interna do Delphi e da orientação a objeto, suas características básicas mantêm-se as mesmas desde o seu lançamento em 1995 (ainda para o Windows 3.1, pois o Windows 95 ainda não havia sido lançado), o que demonstra um profundo respeito com o desenvolvedor. Isto permite que uma aplicação seja facilmente portada de uma versão anterior para uma nova, simplesmente recompilando-se o código fonte.

Obs: Embora as características, teorias e exemplos abordados aqui sejam sobre o Delphi 2005 (última versão disponível), tudo pode ser aplicado em versões anteriores do Delphi, excetuando-se o caso da utilização de componentes e ferramentas introduzidos apenas nesta versão.

PRINCÍPIOS DA PROGRAMAÇÃO PARA WINDOWS

Antes de começar a trabalhar com o Delphi, é importante ter algumas noções do que está envolvido na programação Windows e no Delphi em particular. Algumas coisas tornam a tarefa de programação no Windows (e ambientes baseados em eventos e interface gráfica) bem diferente de outros ambientes e das técnicas de programação estruturada normalmente ensinadas nos cursos de lógica de programação:

Independência do Hardware: No Windows, o acesso aos dispositivos de hardware é feito com intermédio de drivers fornecidos pelo fabricante do hardware, o que evita que o programador tenha que se preocupar com detalhes específicos do hardware. Como acontecia com a programação em DOS.

Configuração Padrão: O Windows armazena centralmente as configurações de formato de números, moeda, datas e horas, além da configuração de cores, livrando o programador de se preocupar com esses detalhes específicos.

Multitarefa: Antigamente, no DOS (não estamos falando do Prompt do MS-DOS), um programa geralmente tomava o controle da máquina só para si, e outros programas não rodavam até que o mesmo fosse fechado. Já no Windows vários programas são executados de maneira simultânea e não há como evitar isso.

Controle da Tela: No DOS geralmente um programa ocupa todo o espaço da tela, e o usuário via e interagia apenas com aquele programa. Já no Windows, todas as informações mostradas e todas as entradas recebidas do usuário são feitas por meio de uma *janela*, uma área separada da tela que pode ser sobreposta por outras janelas do mesmo ou de outros programas.

Padrões de Interface: No Windows, todos os elementos de interface aparecem para o usuário e interagem da mesma forma. Além disso, existem *padrões* definidos pela Microsoft que são recomendados para conseguir a consistência entre aplicativos. Falaremos de alguns deles no curso, mas a melhor forma de aprendê-los é analisar os aplicativos Windows mais usados do mercado.

Eventos e a Cooperação com o Sistema: Num programa criado para DOS (como os programas escritos em Clipper) ele é responsável pelo fluxo de processamento, temos que definir claramente não só que instruções, mas também em que ordem devem ser executadas, ou seja a execução segue uma ordem preestabelecida pelo programador, e o programa só chama o sistema operacional quando precisa de alguma coisa dele. Em Windows não é bem assim. Nosso programa não controla o fluxo de processamento, ele responde e trata *eventos* que ocorrem no sistema.

Existem muitos *eventos* que podem ocorrer, sendo que os principais são aqueles gerados pelo usuário através do mouse e do teclado. A coisa acontece mais ou menos assim: O usuário clica o mouse e o Windows verifica que aplicação estava debaixo do mouse no momento em que foi clicado. Em seguida ele manda uma mensagem para a aplicação informando que ocorreu um clique e as coordenadas do cursor do mouse na tela no momento do clique. A aplicação então responde à mensagem executando uma função de acordo com a posição do mouse na tela.

É claro que o Delphi toma conta do serviço mais pesado e facilita muito as coisas para o programador. Detalhes como as coordenadas da tela em que ocorreu o clique, embora estejam disponíveis, dificilmente são necessários nos programas.

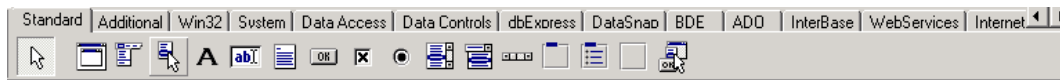
Isso, como veremos, afeta radicalmente o estilo de programação e a forma de pensar no programa. A sequência de execução do programa depende da sequência de eventos.

COMPONENT PALETTE (PALETA DE COMPONENTES)

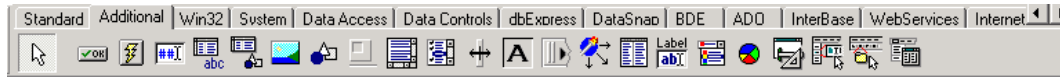
Cada ícone na paleta refere-se a um componente que, quando colocado em um Form, executa determinada tarefa: por exemplo, um TLabel mostra um texto estático, e um TEdit é uma caixa de edição que permite mostrar e alterar dados, o TComboBox é uma caixa que permite selecionar um dentre os itens de uma lista, etc.

A paleta de componentes tem diversas guias, nas quais os componentes são agrupados por funcionalidade. Outras guias podem ser criadas com a instalação de componentes de terceiros.

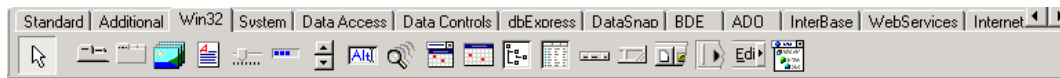
Segue abaixo algumas das principais paletas do Delphi. As imagens foram tiradas do Delphi 6, onde a paleta era na horizontal.



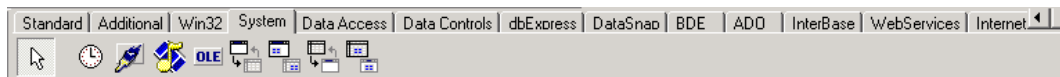
Standard: componentes padrão da interface do Windows, usados para barras de menu, exibição de texto, edição de texto, seleção de opções, iniciar ações de programa, exibir listas de itens etc. Geralmente são os mais usados.



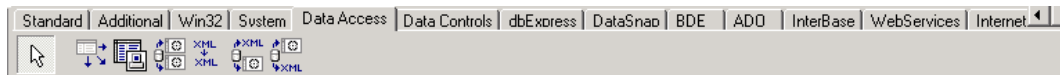
Additional: componentes especializados que complementam os da página Standard. Contém botões com capacidades adicionais, componentes para exibição e edição de tabelas, exibição de imagens, gráficos etc.



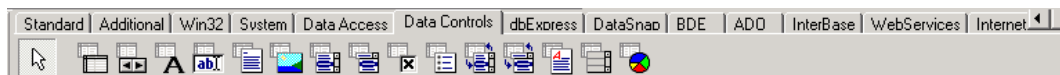
Win32: componentes comuns de interface que são fornecidos pelo Windows 95/NT para os programas. Contém componentes para dividir um formulário em páginas, edição de texto formatado, barras de progresso, exibição de animações, exibição de dados em árvore ou em forma de ícones, barras de status e de ferramentas etc.



System: componentes que utilizam funções avançadas do sistema operacional, como temporização (timers), multimídia e conexões OLE e DDE.



Data Access: componentes de acesso a bancos de dados e conexão com controles de exibição de dados.

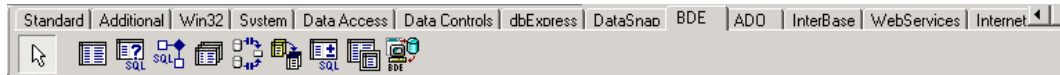


Data Controls: componentes semelhantes aos encontrados nas guias Standard e Additional, porém ligados a banco de dados.

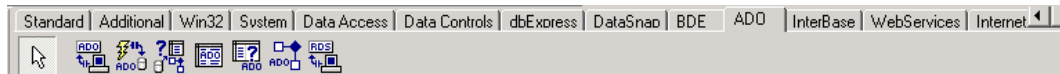


dbExpress: componentes de conexão com bancos de dados SQL introduzida no Delphi 6 e no Kylix (Delphi para Linux). Entre os principais recursos desta nova arquitetura estão dispensar o uso do BDE (Borland Database Engine) para acesso a dados e fazer um acesso muito mais rápido e leve. A configuração e instalação também são mais simples que a do BDE. Atualmente existem drivers para Oracle, DB/2, Interbase e MySQL entre outros. Não existe suporte para bancos de dados Desktop,

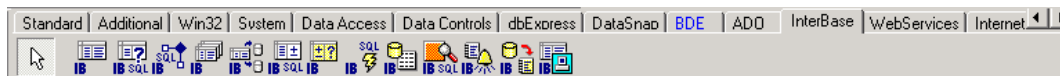
assim, não permite acesso a dBase, Paradox ou Access para estes você deverá continuar a utilizar o BDE.



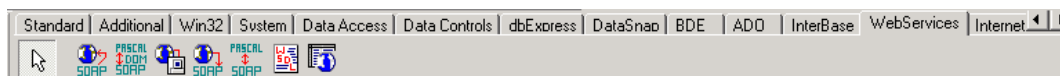
BDE: componentes de acesso a dados utilizando a BDE (até o Delphi 5 faziam parte da guia Data Access). A BDE é a engine que acompanha o Delphi desde sua primeira versão. Muito completa, permite acessar desde bases de dados desktop, como Paradox, dBase ou Access, até bases de dados SGDB, como Interbase, DB/2, Oracle, Informix, SyBase ou MS-SQL Server, todos em modo nativo. Permite ainda o acesso a outros bancos de dados através de ODBC.



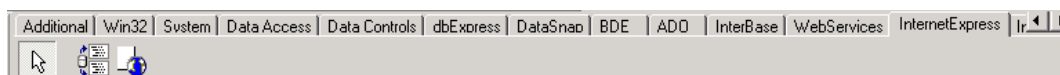
DBGO: componentes de acesso a dados da interface dbGo (introduzida no Delphi 5 como o nome de ADO Express) através da tecnologia ADO (ActiveX Data Objects), da Microsoft. Tanto a ADO como o OLE DB (drivers de acesso) estão incluídos no MDAC (Microsoft Data Access Components) e permitem acesso a uma série de bancos de dados e à ODBC. Sua principal vantagem é estar incorporada as versões mais recentes do Windows (2000/XP e ME) não sendo necessário nenhuma instalação de engine de acesso. Também é escalável, permitindo acesso desde bases de dados desktop até aplicações multicamadas. A desvantagem é que não é portátil para outras plataformas, caso queira portar seu sistema para Linux, terá que trocar todos os componentes de acesso a dados.



Interbase: componentes para acesso nativo ao Interbase, através de sua API, constituindo o método de acesso mais rápido e eficiente para este banco de dados. Por não ser uma interface genérica permite utilizar todos os recursos que o Interbase disponibiliza. A desvantagem, no entanto é que ao utiliza-los perde-se a possibilidade de alterar o banco de dados sem mudar o programa, visto que os mesmos se destinam apenas ao Interbase.



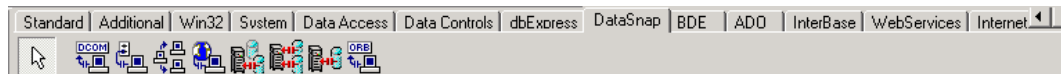
WebServices: componentes que formam a BizSnap, uma plataforma RAD para desenvolvimento de Web services, que simplifica a integração B2B criando conexões e Web services baseados em XML/SOAP



InternetExpress: componentes para manipulação de XML e produção de páginas para internet.



Internet: componentes para manipulação de Sockets, páginas e web browser.



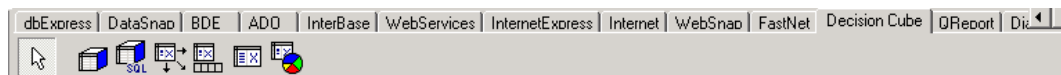
DataSnap: componentes que permitem a criação de middleware de alto desempenho capazes de trabalhar com Web services, possibilitando fácil conexão de qualquer serviço ou aplicação de cliente com os principais bancos de dados, como Oracle, MS-SQL Server, Informix, IBM, DB2, Sybase e InterBase, através de Serviços Web padrão da indústria e XML, DCOM ou CORBA. (No Delphi 5 esta guia era chamada de Midas).



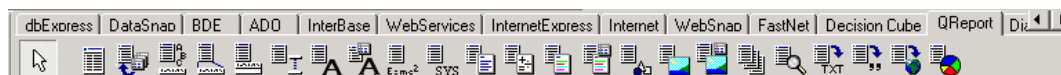
WebSnap: componentes para o desenvolvimento de aplicações Web que suporta os principais Servidores de Aplicações Web, inclusive Apache, Netscape e Microsoft Internet Information Services (IIS);.



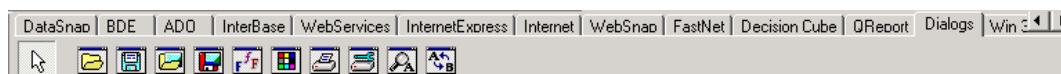
FastNet: componentes para manipulação de protocolos e serviços da internet como http, nntp, ftp, pop3 e smtp entre outros. (São mantidos por compatibilidade com o Delphi 5, nesta versão foram inseridos os componentes Indy com maior funcionalidade).



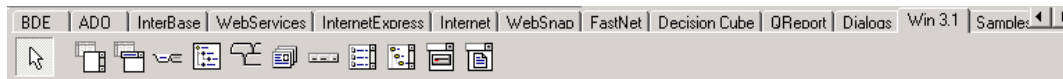
Decision Cube: componentes para tomada de decisão através da análise multidimensional de dados, com capacidades de tabulação cruzada, criação de tabelas e gráficos. Estes componentes permitem a obtenção de resultados como os obtidos por ferramentas OLAP (On-Line Analytical Processing – Processamento Analítico On-Line) utilizados para análise de Data Warehouses.



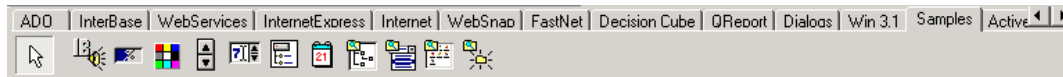
Qreport: QuickReport é um gerador de relatórios que acompanha o Delphi e se integra totalmente ao mesmo, sem a necessidade de run-time ou ferramentas externas como o Cristal Report, etc. Estes componentes permitem a criação de diversos tipos de relatórios de forma visual e também a criação de preview personalizado.



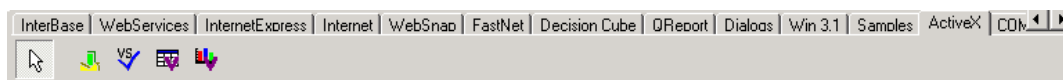
Dialogs: O Windows tem caixas de diálogo comuns, como veremos, que facilitam mostrar uma interface padrão dentro do seu programa para as tarefas comuns, como abrir e salvar arquivos, impressão, configuração de cores e fontes etc. Esta guia tem componentes que permitem utilizar essas caixas de diálogo comuns.



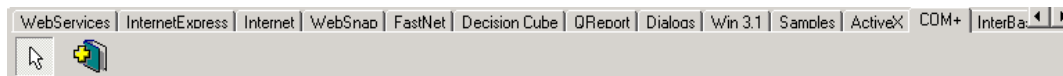
Win 3.1: esta guia contém controles considerados obsoletos, que estão disponíveis apenas para compatibilidade com programas antigos. Não crie programas novos que utilizem esses controles.



Samples: contém exemplos de componentes para que você possa estudá-los e aprender a criar seus próprios componentes. O código fonte desses exemplos está no subdiretório SOURCE\SAMPLES do diretório de instalação do Delphi.



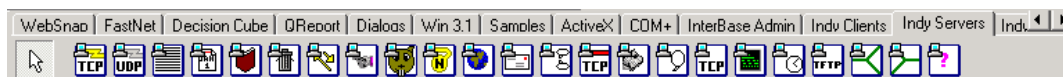
ActiveX: um componente ActiveX é um tipo de componente que pode ser criado em outra linguagem (como C++) e utilizado no Delphi. Esta página contém alguns exemplos de componentes ActiveX prontos para utilizar, que têm funções de gráficos, planilha, etc. O Delphi também pode criar componentes ActiveX, que podem ser utilizado em ambientes como Visual Basic e Visual FoxPro.



COM+: catálogo de objetos COM (Component Object Model), tecnologia desenvolvida pela Microsoft que possibilita a comunicação entre clientes e aplicações servidores. Uma interface COM é a maneira como um objeto expõe sua funcionalidade ao meio externo.



Indy Clients: componentes para criação de aplicativos clientes para protocolos HTTP, TCP, FTP, DNS Resolver, POP3, SMTP, TELNET, entre outros.



Indy Servers: componentes para criação de aplicativos servidores de HTTP, TCP, FTP, TELNET, GOPHER, IRC, entre outros.



Indy Misc: componentes complementares aos das guias Indy Clients e Indy Servers, para criação de aplicativos clientes e servidores com acesso a internet, como clientes de ftp, irc e browsers.



Servers: componentes para automatização do Microsoft Office, permitindo o controle, impressão e criação de documentos destes aplicativos dentro do seu programa.

CONCEITO DE PROGRAMAÇÃO ORIENTADA A OBJETO

Para compreendermos melhor a novo ambiente de desenvolvimento do *Borland* o *Delphi* é necessário que você aprenda e tenha em mente os conceitos de POO (**Programação Orientada a Objetos**) e não confunda os conceitos com POE (**Programação Orientada a Eventos**) muito difundido com o *Access 2.0*® (um ambiente baseado em Objetos). A longo deste capítulo você vai notar as sensíveis diferenças que existem entre esses dois conceitos.

A POO e a POE são facilmente confundidas, mas lembre-se a POO contém a POE mas a POE não contém a POO. Um objeto pode existir mesmo que não exista nenhum evento associado a ele, mas um evento não pode existir se não houver um objeto a ele associado. Outra característica que pode causar confusão são os ambientes Orientados a Objetos e ambientes Baseados em Objetos. Em ambiente Orientado a Objetos consegue-se criar e manipular objetos enquanto que o Baseado em Objetos não é possível a criação de objetos apenas a sua manipulação.

A POO é um conceito desenvolvido para facilitar o uso de códigos de desenvolvimento em interfaces gráficas. Sendo a *Borland*, uma das primeiras a entrar neste novo conceito, possui suas principais linguagens de programação (tais como *Object Pascal* e *C++*), totalmente voltadas para este tipo de programação. A POO atraiu muitos adeptos principalmente pelo pouco uso de código que o projeto (diferente de sistema) carrega no programa fonte, ao contrário das linguagens mais antigas como o *Clipper'87*® muito utilizado no final da década de 80 e início da década de 90. O resultado desta "limpeza" no código resulta que a manutenção do projeto torna-se muito mais simples.

Por ser baseado no Pascal Object, o Delphi permite que se construa aplicações orientadas a objetos. Em linhas gerais, aplicações orientadas a objetos se baseiam no conceito de classe. A classe é um tipo de dado, contendo atributos e serviços. O objeto é uma variável de determinada classe. Por exemplo, um formulário nada mais é do que um objeto da classe Formulário (Tform).

ORIENTAÇÃO A OBJETO

Antes de começarmos a falar realmente de linguagem orientada a objetos e necessário que você possua os conceitos básicos da orientação a objetos, são eles:

Classe - A modelagem de um conceito do mundo real. As propriedades associadas ao conceito são representadas por atributos (variáveis) e operações (i.e. funções). Uma classe descreve os atributos que seus objetos vão ter e as funções que podem executar. (i.e. a classe das Janelas descreve o tamanho, formato, funcionalidades, etc. de uma Janela)

Por exemplo, no domínio universitário os conceitos de professores, aulas, alunos, poderiam ser representados por classes.

Objeto - é qualquer estrutura modular que faz parte de um produto. Uma janela por exemplo, é um objeto de uma casa, de um carro ou de um software com interface gráfica para o usuário.

Cada objeto *pertence* a uma classe. A estrutura do objeto como as operações que ele pode executar são descritas pela classe.

Objetos são entidades independentes uns dos outros. Dois objetos com exatamente os mesmos atributos são dois objetos diferentes. Na informática, se pode pensar na memória de um computador: dois objetos com os mesmos atributos, estão em diferentes partes da memória, na verdade eles tem um atributo implícito (o endereço na memória onde eles ficam) que é diferente.

Instância - Um objeto. Por exemplo, uma casa pode possuir várias instâncias da classe janela.

Propriedades- São as características do objeto, como cor e tamanho, a janela, por exemplo, tem atributos como o modelo, tamanho, abertura simples ou dupla, entre outros.

Encapsulação - é um mecanismo interno do objeto “escondido” do usuário. Uma pessoa pode abrir uma janela girando a tranca sem precisar saber o que há dentro dela.

Herança - um objeto novo nem sempre é criado do zero. Ele pode “herdar” atributos e ações de outros já existentes. Uma janela basculante herda atributos das janelas e das persianas.

Mais formalmente, Herança é a modelagem da noção de especialização/generalização. No mundo real, conceitos são especializações uns dos outros: um professor visitante é um caso especial de professor; um professor, um aluno são seres humanos; um ser humano é um animal, ...Os conceitos mais especiais têm todas as propriedades dos conceitos mais gerais, mais algumas propriedades em si próprio.

Na POO, uma sub-classe possui todos os atributos e todas as operações da super-classe. A sub-classe pode acrescentar alguns atributos e métodos. Ela pode também redefinir, dentro de alguns limites, alguns métodos da super-classe. Ela não pode tirar nenhuma propriedade da super-classe.

Polimorfismo - O polimorfismo é um recurso das linguagens orientadas a objetos, que literalmente indica a capacidade de um objeto assumir várias formas. Em outras palavras, o polimorfismo permite que você referencie propriedades e métodos de classes diferentes por meio de um mesmo objeto. Ele também possibilita que você execute operações com esse objeto de diferentes maneiras, de acordo com o tipo de dado e com a classe atualmente associada a esse objeto. O comando “abre”, por exemplo, faz um objeto entrar em ação, seja ele uma janela, uma porta ou uma tampa de garrafa.

Várias classes podem implementar a mesma operação de maneira diferente. A mesma operação (com o mesmo nome) tem várias (“poli”) formas (“morfismo”) em cada classe. Uma característica poderosa do modelo a objetos é que todas as formas de uma operação podem ter o mesmo nome. Assim se torna mais fácil de reconhecer qual operação um método particular implementa. Isso é possível porque cada objeto sabe qual é sua própria classe, e pode executar os métodos apropriados.

EVOLUÇÃO DAS LINGUAGENS PARA OO

A Orientação a Objetos em computação tem 20 anos apesar de ter surgido fortemente apenas nos últimos 7 anos. Surgiu mesmo na área acadêmica.

- 1967 - Simula (Noruega)
- 1980 - Small Talk (Xerox) com objetivos comerciais e acadêmicos, Adele Goldberg (autora)
- 1980's Objective C(Cox), C++ (Stroustrup), Eiffel (Meyer)

- Anos 70 - Época da não estruturada, dados e códigos emaranhados.
- Anos 80 - Época da estruturada, dados separados de códigos (modulares)
- Anos 90 - Época da OO, dados e códigos organizados em objetos.

A partir dos anos 80, as principais linguagens incluíram conceitos de OO. Ex. Pascal, C, Lisp, Cobol, depois evoluíram com a inclusão de classes. C++ foi um marco para aplicações em todos os níveis. Surge o Visual Object Oriented Cobol (c/ win95) e o Visual Basic. As linguagens mais utilizadas no mundo: 1 - Cobol, 2 – Visual Basic (informação do Richard Soley e do Jon Siegel da OMG, em dezembro 96).

BENEFÍCIOS DA OOP

Listaremos a seguir os principais ganhos, para o desenvolvedor, caso este troque a Metodologia Tradicional pela Orientada ao Objeto.

Exatidão- Devido à característica do desenvolvimento estruturado, onde se elabora um projeto e DEPOIS se faz os programas, podemos ter no final um sistema que não atenda perfeitamente seus objetivos depois de implementado. No desenvolvimento OOP, devido ao fato deste ser feito de maneira quase que interativa com o usuário, este risco é significativamente diminuído. A pouca quantidade de código programável também reduz os problemas inerentes as mudanças das especificações durante o desenvolvimento do projeto.

Potencialidade- Definimos potencialidade a forma como o programa reage aos erros imprevistos como uma falha na impressora, ou a um disco cheio. Tanto maior for a potencialidade, maior a capacidade do programa em causar o menor estrago possível aos dados e evitar uma saída drástica do sistema.

Extensibilidade - Dizemos que quanto maior for a extensibilidade do software, maior será sua capacidade em adequar-se as especificações definidas pelos analistas.

Reutilização - A capacidade de se otimizar a produtividade do programador depende diretamente da maneira como o software disponibiliza a reutilização do código gerado. De fato, a maioria dos programadores profissionais, já reutiliza código anteriormente gerado, porém a perfeita reutilização consiste na utilização COMPLETA de um código gerado para algum sistema SEM qualquer outra adaptação prévia.

LINGUAGENS PROCEDURAIS X ORIENTADA A OBJETOS :

Mesmo as linguagens procedurais oferecem alguma característica funcional OO. Seu conjunto de primitivas porém permite que você digite comandos para que o computador execute. A organização

e manipulação dos dados vêm depois. A linguagem OO é projetada para permitir a definição dos objetos que compõem os programas e as propriedades que estes objetos contêm. O código é secundário. Pode-se programar na forma procedural com estilo OO quando o programa tem um nível tal de sofisticação que você começa a criar seus próprios tipos e estruturas de dados. A linguagem OO pura tem como um parâmetro de referência que todos os dados sejam representados na forma de objetos. A única pura é a Small Talk (tudo via classe, não tem var. global). Eiffel também.

ALGUNS EXEMPLOS DE LINGUAGENS PROCEDURAIS COM EXTENSÕES OO:

C++, PASCAL do ambiente DELPHI, PERL, OBJECTIVE C

O JAVA implementa tipos simples de dados (integer, real, char) do C, não sendo objetos mas tudo o mais lida com objetos. O código e os dados residem em classes e objetos.

LINGUAGENS COMPILADAS X INTERPRETADAS :

Linguagem Compilada - código gerado tem alta performance pois é ajustado para um tipo específico de processador. Código Objeto em binário para aquele processador. Ex. Pascal (entenda-se também no Delphi).

Linguagem Interpretada - só existe em código fonte. Durante a execução o interpretador pega o fonte e vai executando as ações. Facilita o uso em múltiplas plataformas. Ex. Clipper

Um caso de linguagem Compilada/Interpretada é o Java. O Java edita o fonte e salva como código fonte, depois compila o fonte produzindo arquivo binário chamado arquivo de classe que não é executável direto. Este código intermediário chama-se bytecode que são instruções de máquina mas para um processador virtual (o JVM Java Virtual Machine). Por fim o interpretador Java implementa em software executando os arquivos de classe.

A OO tem a filosofia de implementar uma abstração do mundo real facilitando a reutilização de código, denominado de caixas pretas de softwares ou ainda softwares IC's. (Integrated Circuits) que estão sendo utilizado aqui como uma analogia de chip com objeto.

EVENTOS, MÉTODOS E PROPRIEDADES

EVENTOS

Os programas feitos em Delphi são orientados a eventos. Eventos são ações normalmente geradas pelo usuário e que podem ser reconhecidas e tratadas pelo programa. Ex.: Clicar o mouse pressionar uma tecla, mover o mouse etc. Os eventos podem ser também gerados pelo windows.

Existem eventos associados ao formulário e cada componente inserido neste.

Exemplos:

- Ao formulário está ligado *on show*, que ocorre quando mostramos o formulário na tela.
- Ao componente botão está ligado o evento *on click*, que ocorre quando damos um click com o mouse sobre o botão.

Eventos comuns ao formulário e aos componentes.

Alguns eventos ligados tanto ao formulário quanto aos componentes estão listados a seguir.

- **OnClick**: ocorre quando o usuário clica o objeto.
- **OnDblClick**: ocorre quando o usuário dá um duplo clique.
- **OnKeyDown**: ocorre quando o usuário pressiona uma tecla enquanto o objeto tem foco.
- **OnKeyUp**: ocorre quando o usuário solta uma tecla enquanto o objeto tem o foco.
- **OnKeyPress**: ocorre quando usuário dá um clique numa tecla ANSI.
- **OnMouseDown**: ocorre quando o usuário pressiona o botão do mouse.
- **OnMouseUp**: ocorre quando o usuário solta o botão do mouse.
- **OnMouseMove**: ocorre quando o usuário move o ponteiro do mouse.

Rotinas que Respondem a Eventos

Cada evento gera uma procedure, aonde você deve inserir as linhas de código que envolvem este evento. Por exemplo, o evento OnClick, que é gerado ao clicarmos em um botão chamado BTNSair, cria a procedure:

```
Procedure TForm1.BTNSairClick(Sender: TObject);
```

onde TForm1 é o objeto TForm que contém o botão BTNSair, e Sender é um objeto TObject que representa o componente que deu origem ao evento.

Se você quiser inserir uma rotina que trate um determinado evento de um componente, faça o seguinte:

- clique sobre o componente;
- no Object Inspector, selecione a página Events;
- dê um duplo clique sobre o evento para o qual quer inserir o código;
- entre no editor de código e escreva as linhas de código.

Exemplo:

```
Procedure TForm1.BTNSairClick(Sender: TObject);  
begin  
  Form1.Close;  
end;
```

Obs.: Escreva seu código entre o begin e o end, se por acaso você quiser retirar o evento e o componente, retire primeiro os eventos do componente removendo somente o código que você colocou e depois o componente; os resto dos procedimentos o DELPHI tira para você.

PROPRIEDADES

Uma propriedade representa um atributo de um objeto. No Delphi todas as coisas que aparecem no Object Inspector são propriedades, inclusive os eventos, porém sua referência é a um método. Como vimos, eventos podem estar associados a modificações em propriedade de componente e formulário, ou seja, você pode modificar propriedades de formulários e componentes durante a execução do sistema. Para isto você deverá usar a sintaxe:

```
<componente>.<propriedade>;
```

Por exemplo, para modificar a propriedade *text* de uma caixa de edição Edit1 para “Bom Dia” faça:

Edit1.Text := 'Bom Dia';

Se a propriedade do componente tiver subpropriedades, para acessá-lá, utilize a seguinte sintaxe:

<componente>.<propriedade>.<subpropriedade>

Por exemplo, para modificar a subpropriedade Name referente a propriedade fonte, de uma caixa de edição Edit1, para 'Script', faça:

Edit1.Font.name := 'Script';

Obs.: Verifique o tipo da propriedade para antes de mandar o valor, consultando no Object Inspector.

Obs.: Cada componente tem suas próprias propriedades e eventos, e podem existir propriedades iguais de um componente para o outro, portanto lembre-se das propriedades e eventos comuns entre eles.

MÉTODOS

São procedures ou funções embutidas nos componentes e formulários, previamente definidas pelo Delphi.

Alguns métodos são descritos a seguir:

- Show : Mostra um formulário;
- Hide : Esconde um formulário mais não o descarrega;
- Print : Imprime um formulário na impressora;
- SetFocus : Estabelece o foco para um formulário ou componente;
- BringToFront: Envia para frente.

Chamado de métodos como resposta a eventos:

Um evento pode gerar a chamada para um método, ou seja, uma subrotina previamente definida para um componente.

No código, para utilizar um método, use a seguinte sintaxe:

<nome do objeto>.<método>

Por exemplo, clicar em um botão pode dar origem ao evento Show de um outro formulário, mostrando este novo formulário na tela:

Form2.show;

OBJECT PASCAL

Object Pascal é uma linguagem Orientada a Objetos não pura mas híbrida por possuir características de programação não só visual mas também escrita, para os programadores que já conhecem técnicas de estruturas de programação, com o *C*, *Basic*, *Pascal* ou *xBASE* entre outras

linguagens a **Object Pascal** providência uma migração de forma natural oferecendo um produto de maior complexibilidade. **Object Pascal** força a você executar passos lógicos isto torna mais fácil o desenvolvimento no ambiente *Windows*® de aplicações livres ou que utilizam banco de dados do tipo *Cliente/Servidor*, trabalha com o uso de ponteiros para a alocação de memória e todo o poder de um código totalmente compilável. Além disso possibilita a criação e reutilização (vantagem de re-uso tão sonhado com a **Orientação a Objetos**) de objetos e bibliotecas dinâmicas (*Dynamic Link Libraries* - DLL).

Object Pascal contém todo o conceito da orientação a objetos incluindo encapsulamento, herança e polimorfismo. Algumas extensões foram incluídas para facilitar o uso tais como conceitos de propriedades, particulares e públicas, e tipos de informações em modo run-time, manuseamento de exceções, e referências de classes. O resultado de toda esta junção faz com que **Object Pascal** consiga suportar as facilidades de um baixo nível de programação, tais como:

- Controle e acesso das subclasses do *Windows*® (API);
- Passar por cima das mensagens de loop do *Windows*®;
- Mensagens semelhantes as do *Windows*®;
- Código puro da linguagem *Assembler*.

Como deu para perceber a base de toda a programação *Delphi* é a linguagem **Object Pascal**, então neste capítulo trataremos exclusivamente deste tipo de programação.

Símbolos Especiais

A **Object Pascal** aceita os seguintes caracteres ASCII:

- ✓ Letras - do Alfabeto Inglês: **A** até **Z** e **a** até **z**.
- ✓ Dígitos - Decimal: **0** até **9** e Hexadecimal: **0** até **9** e **A** até **F** (ou **a** até **f**)
- ✓ Brancos - Espaço (**ASCII 32**) e todos os caracteres de controle **ASCII** (**ASCII 0** até **ASCII 31**), incluindo final de linha e Enter (**ASCII 13**).
- ✓ Especiais - Caracteres: **+ - * / = < > [] . , () : ; ^ @ { } \$ #**
- ✓ Símbolos - Caracteres: **<= >= := .. (* *) (. .) //**

+ O colchetes esquerdo (**[**) e equivalente ao (**(** e o colchetes direito (**]**) e equivalente a **.)**. A chave esquerda (**{**) e equivalente ao (***** e a chave direita (**}**) e equivalente a *****).

PALAVRAS RESERVADAS

A **Object Pascal** se utiliza das seguintes palavras reservadas, não podendo as mesmas serem utilizadas ou redefinidas:

And	Exports	Library	Set
Array	File	Mod	Shl
As	Finally	Nil	Shr
Asm	For	Not	String
Begin	Function	Object	Then
Case	Goto	Of	To
Class	If	On	Try
Const	Implementation	Or	Type
Constructor	In	Packed	Unit
Destructor	Inherited	Procedure	Until

Div	Initialization	Program	Uses
Do	Inline	Property	Var
Downto	Interface	Raise	While
Else	Is	Record	With
End	Label	Repeat	Xor
Except			

VARIÁVEIS

Nossos dados são armazenados na memória do computador. Para que nós não tenhamos que nos referir a estes dados de forma direta, através de um endereço numérico difícil de memorizar, o compilador nos permite utilizar variáveis com esta finalidade. Escolhendo nomes sugestivos (mnemônicos) para nossas variáveis (tais como *nome*, *funcao*, *idade*, *salario*) facilitamos bastante a compreensão de nosso código.

Para que o Delphi possa usar nossas variáveis, devemos primeiro declará-las, isto é, informar o nome e o tipo desejados. Por exemplo : o comando a seguir declara *idade* como sendo uma variável do tipo inteiro (*integer*) : *idade : integer;*

As variáveis inteiras podem assumir valores entre -32768 e +32767. Elas ocupam 2 bytes na memória. Assim sendo, a declaração acima faz com que o Delphi reserve 2 bytes para a nossa variável *idade*. Note que a declaração do tipo de uma variável, em princípio não lhe atribui valores. Um erro comum em programação é tentarmos *ler* valores de variáveis não inicializadas, ou às quais ainda não se atribuiu valores...

Damos a seguir uma lista dos tipos de variáveis mais comuns do Object Pascal com suas faixas de valores e o espaço ocupado em memória:

- BOOLEAN** - Tipo lógico que pode assumir somente os valores TRUE ou FALSE e ocupa 1 byte de memória.
- BYTE** - Tipo numérico inteiro, pode assumir valores numa faixa de 0 a 255, ocupa 1 byte.
- CHAR** - Tipo alfa-numérico, pode armazenar um caractere ASCII, ocupa 1 byte.
- COMP** - Tipo numérico real, pode assumir valores na faixa de $-9.2.10^{-18}$ a $9.2.10^{+18}$, ocupa 8 bytes, pode ter entre 19 e 20 algarismos significativos.
- EXTENDED** - Tipo numérico real, pode assumir valores na faixa de $-3.4.10^{-4932}$ a $+1.1.10^{+4932}$, ocupa 10 bytes de memória e tem entre 19 e 20 algarismos significativos.
- INTEGER** - Tipo numérico inteiro, pode assumir valores numa faixa de -32768 a +32767, ocupa 2 byte de memória.
- LONGINT** - Tipo numérico inteiro, pode assumir valores numa faixa de -2147483648 a +2147483647, ocupa 4 bytes de memória.
- REAL** - Tipo numérico real, pode assumir valores na faixa de $-2.9.10^{-39}$ a $+1.7.10^{+38}$, ocupa 6 bytes de memória e tem entre 11 e 12 algarismos significativos.
- CURRENCY** - Tipo numérico Real utilizado para armazenar valores monetários.
- SHORTINT** - Tipo numérico inteiro, pode assumir valores numa faixa de -128 a +127, ocupa 1byte de memória.
- SINGLE** - Tipo numérico real, pode assumir valores numa faixa de $-1.5.10^{-45}$ a $+3.4.10^{+38}$, ocupa 4 bytes de memória, e tem de 7 a 8 algarismos significativos.
- WORD** -Tipo numérico inteiro, pode assumir valores numa faixa de 0 a 65535, ocupa 2bytes de memória.
- STRING** - Tipo alfanumérico, possuindo como conteúdo uma cadeia de caracteres. O número de bytes ocupados na memória varia de 2 a 256, dependendo da quantidade máxima de caracteres definidos para a string. O primeiro byte contém a quantidade rela de caracteres da cadeia.

Os nomes de variáveis devem começar com uma letra ou o caractere sublinhado (_) seguido por uma sequência de letras, dígitos ou caractere sublinhado (_) e não podem conter espaço em branco nem quaisquer tipos de acentos. Os nomes de variáveis podem ter qualquer tamanho mas somente os 63 primeiros caracteres serão considerados.

Exemplos : Para definir uma variável *Nome* do tipo string e uma variável *Salario* do tipo *double*, podemos inserir as seguintes linhas de código na cláusula **var** da unidade de código correspondente.

Nome : string;
Salario : double;

Pode-se declarar mais de uma variável do mesmo tipo na mesma linha, separando-as por vírgula.
 nome, funcao, endereco : string;

ARRAYS (VETORES)

Arrays são conjuntos de variáveis com o mesmo nome e diferenciadas entre si por um índice. Eles são úteis para manipularmos grandes quantidades de dados de um mesmo tipo pois evitam a declaração de diversas variáveis.

Considere o caso de um programa de Folha de Pagamento que precise armazenar os seguintes dados referentes a 100 funcionários : nome, funcao, salário, etc... Seríamos obrigados a declarar 100 variáveis nome, 100 variáveis funcao, etc... O array nos permite declarar uma única variável com um índice para apontar para as diferentes ocorrências.

Declara-se um array da seguinte forma :

nome_da_variável : array[i1..i2] of tipo_de_variável; onde i1 e i2 representam os valores mínimo e máximo, respectivamente, do índice.

O Object Pascal permite que i1 e i2 possuam qualquer valor desde que i1 seja menor ou igual a i2. Assim, poderíamos declarar um array de 100 variáveis inteira *idade* de várias formas diferentes :

idade : array [1..100] of integer; ou
idade : array [-100..-1] of integer; ou
idade : array [0..99] of integer, etc...

Pode-se definir arrays multidimensionais (com vários índices) como, por exemplo :
espaco3d:array[1..10,-5..20,0..30] of double; que pode armazenar 10x26x31=8060 variáveis do tipo double.

Um dos casos mais comuns é a matriz com m linhas e n colunas : *matriz : array[1..m,1..n] of qqer_tipo.*

Os elementos dos arrays podem ser quaisquer tipos de variáveis ou objetos.

RECORDS (REGISTROS)

O Object Pascal permite definir tipos compostos de variáveis denominados registros. Define-se da seguinte forma :

```
nome_do_tipo : Record
    variavel1 : primeiro_tipo;
    variavel2 : segundo_tipo;
    .....
    variaveln : n-ésimo-tipo;
end;
```

variavel1,variavel2.. variaveln são chamadas de campos do registro.

Declaramos uma variável deste tipo da mesma forma que procedemos para declarar variáveis de qualquer tipo pré-definido

```
variavel : nome_do_tipo;
```

Usamos a notação de ponto para acessar um campo de uma variável composta :

```
nome_da_variavel.nome_do_campo;
```

Exemplo :

```
funcionario = Record
  nome : string;
  funcao : string;
  salario : double;
end;
```

Assim, no exemplo citado anteriormente, ao invés de declararmos um array nome de 100 elementos, um array funcao de 100 elementos, um array salario de 100 elementos, podemos declarar uma única variável chamada empregado, por exemplo, como sendo um array de 100 elementos do tipo funcionário.

```
empregado : array[1..100] of funcionario;
```

Para obter os dados do décimo funcionário, basta fazer :

```
empregado[10].nome;
empregado[10].funcao;
empregado[10].salario;
```

PRINCIPAIS TECLAS DE ATALHO DA IDE DO DELPHI

F9 → Executa

F8 → Executa passo a passo sem entrar em sub-rotinas

F7 → Executa passo a passo entrando em sub-rotinas

CTRL+F5 → Adiciona uma variável aos *Watch List* para ser monitorada durante o Debug

CTRL+F9 → Compila

CTRL + SHIFT + (um número de 0 a 9) → marca uma posição no texto

CTRL + (um número de 0 a 9) → vai para uma posição previamente marcada/

F11 → Painel de propriedades

F12 → Comuta entre o formulário e a tela de codificação

CTRL+F2 → pára a execução de um programa.

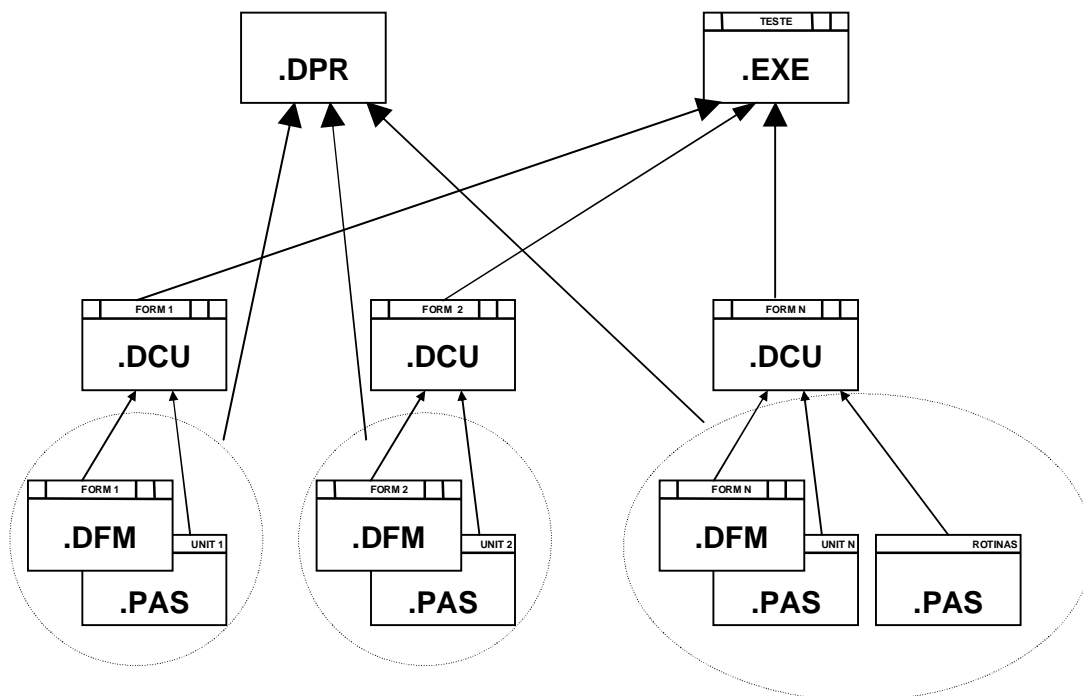
CTRL+F1 → Executa o HELP a respeito do texto onde o cursor estava posicionado.

COMO É FORMADA UMA APLICAÇÃO EM DELPHI

Quando você abre um projeto no Delphi, ele já mostra uma UNIT com várias linhas de código. Este texto tem como objetivo explicar um pouco desta estrutura que o ele usa. Um projeto Delphi tem, inicialmente, duas divisórias: uma UNIT, que é associada a um Form, e outra Project, que engloba todos os FORM e UNITs da aplicação.

Em Delphi temos: o Project, os Forms e as Units. Para todo Form temos pelo menos uma Unit (Código do Form), mas temos Units sem form (códigos de procedures, funções, etc).

ARQUIVOS QUE COMPÕE UMA APLICAÇÃO DELPHI.



ARQUIVOS GERADOS NO DESENVOLVIMENTO

Extensão Arquivo	Definição	Função
.DPR	Arquivo do Projeto	Código fonte em Pascal do arquivo principal do projeto. Lista todos os formulários e units no projeto, e contém código de inicialização da aplicação. Criado quando o projeto é salvo.
.PAS	Código fonte da Unit(Object Pascal)	Um arquivo .PAS é gerado por cada formulário que o projeto contém. Seu projeto pode conter um ou mais arquivos .PAS associados com algum formulário. Contem todas as declarações e procedimentos incluindo eventos de um formulário.
.DFM	Arquivo gráfico do formulário	Arquivo binário que contém as propriedades do desenho de um formulário contido em um projeto. Um .DFM é gerado em companhia de um arquivo .PAS para cada formulário do projeto.
.OPT	Arquivo de opções do projeto	Arquivo texto que contém a situação corrente das opções do projeto. Gerado com o primeiro salvamento e atualizado em subsequentes alterações feitas para as opções do projeto.
.RES	Arquivo de Recursos do Compilador	Arquivo binário que contém o ícone, mensagens da aplicação e outros recursos usados pelo projeto.
.~DP	Arquivo de Backup do Projeto	Gerado quando o projeto é salvo pela segunda vez.
.~PA	Arquivo de Backup da Unit	Se um .PAS é alterado, este arquivo é gerado.
.~DF	Backup do Arquivo gráfico do formulário	Se você abrir um .DFM no editor de código e fizer alguma alteração, este arquivo é gerado quando você salva o arquivo.
.DSK	Situação da Área de Trabalho	Este arquivo armazena informações sobre a situação da área de trabalho específica para o projeto em opções de ambiente(Options Environment).

Obs.: .~DF, .~PA, .~DP são arquivos de backup(Menu Options, Environment, Guia Editor Display, Caixa de Grupo Display and file options, opção Create Backup Files, desativa o seu salvamento).

Devido a grande quantidade de arquivos de uma aplicação, cada projeto deve ser montado em um diretório específico.

ARQUIVOS GERADOS PELA COMPILAÇÃO

Extensão Arquivo	Definição	Função
.EXE	Arquivo compilado executável	Este é um arquivo executável distribuível de sua aplicação. Este arquivo incorpora todos os arquivos .DCU gerados quando sua aplicação é compilada. O Arquivo .DCU não é necessário distribuir em sua aplicação.
.DCU	Código objeto da Unit	A compilação cria um arquivo .DCU para cada .PAS no projeto.

Obs.: Estes arquivos podem ser apagados para economizar espaço em disco.

CÓDIGO FONTE DO ARQUIVO PROJECT(.DPR)

Nesta arquivo está escrito o código de criação da aplicação e seus formulários. O arquivo Project tem apenas uma seção.

Esta seção é formada pelo seguinte código:

PROGRAM - Define o Projeto;

USES - Cláusula que inicia uma lista de outras unidades.

Forms = É a unidade do Delphi que define a forma e os componentes do aplicativo

In = A clausula indica ao compilador onde encontrar o arquivo Unit.

Unit1 = A unidade que você criou

{ \$R *.RES } - Diretiva compiladora que inclui o arquivo de recursos.

Abaixo veja como fica o Project quando você abre um projeto novo:

```
program Project1;
```

```
uses
```

```
  Forms,
```

```
  Unit1 in 'UNIT1.PAS' {Form1};
```

```
{ $R *.RES }
```

```
begin
```

```
  Application.CreateForm(TForm1, Form1);
```

```
Application.Run;
end.
```

ESTRUTURA DA UNIT DE UM FORM (.PAS)

As Units do Delphi possuem uma estrutura que deve ser obedecida. Quando um Form é criado também é criada uma Unit associada ao mesmo.

A estrutura básica de uma Unit pode ser visualizada observando-se o código fonte da mesma no editor de código. Será semelhante ao exibido a seguir:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs;

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

end.
```

Vamos analisar o código acima:

- Na primeira linha, o nome em frente à palavra **unit**, no caso Unit1, indica o nome dado ao arquivo com a programação do formulário. Se o formulário fosse salvo com este nome ele geraria um arquivo externo com o nome de Unit1.pas e outro com o nome de Unit1.dfm. (Quando for salvar seus formulários você deve dar nomes mais significativos).
- Na linha seguinte, a palavra **interface** delimita a seção de interface na qual serão colocadas as definições de funções, procedimentos, tipos e variáveis que poderão ser vistos por outras units da aplicação.
- A cláusula **Uses** dentro da seção interface indica quais units deverão ser ligadas para poder complementar a nossa . Ao criar um Form as Units definidas no código acima são inseridas automaticamente, pois fornecem o suporte para criação do mesmo. Ao inserir componentes num Form, outras Units podem ser adicionadas a esta lista.

- A seguir temos a definição dos tipos do programa, identificada pela palavra **type**. Neste ponto temos a definição de uma classe TForm1 que é derivada da classe base TForm. Ao se acrescentar componentes no Form também será gerado no código definição correspondente aos mesmos. (O conceito de classes e objetos será explicado no Capítulo 2)
- O próximo item é a definição de variáveis e constantes globais, através da palavra reservada **var**. Neste ponto é criada uma variável com visibilidade global (pode ser vista em outras units nas quais a mesma seja incluída na cláusula uses)
- A palavra chave **implementation** delimita a segunda seção da unit, onde serão colocadas as funções e variáveis que serão acessadas apenas por ela mesma (não são visíveis em outras units).
- O símbolo *{ \$R *.dfm }* faz a associação da unit com seu respectivo form e não deve ser modificado. Uma unit de funções não possui esta diretiva.
- Ao Final da Unit, temos uma linha com **end**. Ele é o marcador de final de arquivo. Qualquer coisa colocada após esta linha será ignorada.
- Opcionalmente, uma unit pode ter ainda duas seções: **initialization** e **finalization**, com comandos que são executados quando a aplicação é iniciada ou finalizada.

O PARÂMETRO “SENDER” E OS OPERADORES “IS” E “AS”

O OPERADOR “IS”

O operador **IS** é utilizado para lidar com classes e objetos. Ele verifica se um objeto é de uma determinada classe. Tipicamente ele aparece da seguinte forma:

*If NomeObjeto **is** NomeClasse then....*

Mais adiante veremos um exemplo.

O OPERADOR “AS”

O operador **AS** também é utilizado lidar com classes e objetos. Ele diz ao delphi que determinado objeto deve ser tratado como se fosse de determinada classe (e ele tem que ser, senão ocorrerá um erro). Este processo é chamado de **TypeCast**. Normalmente ele é utilizado após a utilização do operador **IS**. Ex:

```
If NomeObjeto is NomeClasse then
Begin
    (NomeObjeto as NomeClasse).ação/propriedade;
    ou
    NomeClasse( NomeObjeto).ação/propriedade
End;
```

O PARÂMETRO SENDER

Agora veremos o que é o parâmetro **SENDER : OBJECT** que aparece na maioria dos eventos. Veremos também exemplos para os operadores IS e AS.

O parâmetro SENDER é uma referência ao componente que acionou o evento, como se fosse um nome diferente para o componente dentro do procedimento, e através dele podemos acessar as propriedades e métodos do componente.

'Sender' é declarado como '**TObject**', o que quer dizer que ele é um objeto genérico do Delphi, que pode ser um componente qualquer. Se você quiser acessar uma propriedade específica de um componente, por exemplo, Color, precisa dizer ao Delphi que temos certeza que 'Sender' é um componente da classe TEdit. Por exemplo:

```
(Sender as TEdit).Color := clYellow;
```

Ou

```
TEdit(Sender).color := clYellow;
```

Ou

```
with Sender as TEdit do Color := clYellow;
```

A expressão **Sender as TEdit** força o Delphi a tratar 'Sender' como um objeto da classe TEdit, ou seja, um componente Edit. Usando essa expressão, podemos acessar propriedades do componente.

Isso é extremamente interessante em casos onde um mesmo evento é utilizado por muitos componentes. Isso mesmo! Um único evento associado a vários componentes, e estes nem precisam ser do mesmo tipo! Nesses casos, eles devem ter como ancestral a mesma classe.

Exemplo:

Imagine 2 botões, um da classe TBUTTON e outro da classe TBITBTN que devem executar a mesma ação: Ao serem clicados, devem exibir uma mensagem na tela e sua propriedade CAPTION deve ser alterada para "LIGADO". Normalmente programaríamos o evento onClick dos botões da seguinte forma:



```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Button1.Caption := 'Ligado';
  ShowMessage('Botão foi ligado.');
```

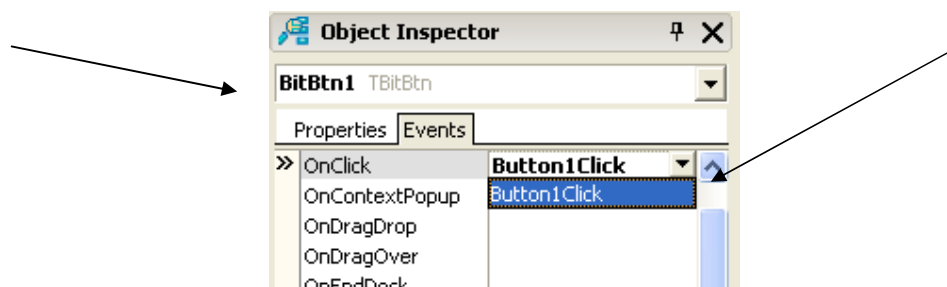
```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  BitBtn1.Caption := 'Ligado';
  ShowMessage('Botão foi ligado.');
```

Perceba que a única coisa que muda nos códigos acima é que no da esquerda o código faz referência ao `Button1` e no da direita a referência é feita ao componente `BitBtn1`. Não que desta forma não funcione, porém imagine que tivéssemos uns 20 ou mais componentes com o mesmo código.

E em que o parâmetro **sender** nos ajudaria nesse caso? Simples! Programamos em um dos botões da seguinte forma:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  (sender as TButton).Caption := 'Ligado';
  ShowMessage('Botão foi ligado.');
```

Agora, associamos o evento `onClick` do objeto `Button1` ao evento `onClick` do objeto `BitBtn1`:



Observe a terceira linha `(sender as TButton).caption := 'Ligado';` Nesta linha o Delphi assume que o objeto passado no parâmetro `SENDER` é da classe `TBUTTON` e portanto acessará todas as propriedades, métodos e eventos como tal.

Mas e quanto ao objeto da classe `TBITBTN`? Ele não é da classe `TBUTTON`, portanto, como pode funcionar? Simples: funciona perfeitamente porque o objeto `TBITBTN` tem como ancestral o objeto `TBUTTON` e portanto pode ser tratado como tal. Este é um bom exemplo do polimorfismo!

Note que neste caso não importa o nome do componente! Isso gera um código mais eficiente, que ficará substancialmente menor e de manutenção mais fácil.

UM EXEMPLO COMPLETO COM OS OPERADORES “AS” E “IS”

Imagine um formulário com vários componentes e de várias classes, como `TLabel`, `TEdit`, `TButton`, `TMaskEdit`, `TMemo`, etc. Neste formulário temos um botão chamado Limpar onde desejamos limpar todos os compentes classe `TEdit` e desligar todos os componentes da classe `TButton`, menos um botão chamado `btnFechaTela`. Vamos ver uma forma eficiente de fazer isso sem ter que programar componente a componente.

Exemplo do formulário

Código do botão Limpar:

```

procedure TForm1.LimpaBtnClick(Sender: TObject);
var i : integer;
begin
    // componentcount : quantidade todos os componentes presentes no formulário.
    //                  // Eles são armazenados como sendo da classe TComponent.
    // componentes[] : vetor que faz referência à todos os componentes do formulário inicia do 0 (zero).

    for i:=0 to componentcount -1 do
    begin
        if components[i] is TEdit then // Utilizamos IS para saber se o componente da posição i é da classe TEdit
            ( components[i] as TEdit) .clear // Trate-o como um TEdit e execute seu método clear, ou .Text := ""
        else if (components[i] is TButton) then // se componentes[i] for da classe um TButton
        begin
            if (components[i] <> btnFechaTela) then // e se for diferente do botao de fechar a tela
                ( components[i] as TButton).Enabled := false; // trate-o como um TButton e desligue-o
            end;
        end;
    end;
end;

```

Não poderíamos ter feito diretamente `Components[i].clear` porque o Vetor `Components` armazena os objetos como sendo da classe `TComponent` (ele faz isso pois `TComponent` é pai [ancestral] de todos os componentes passíveis de serem “colocados” em um formulário) e esta classe não implementa o método `Clear`, tão pouco a propriedade `Text`.

Observe que no código acima os objetos OBS não será limpo, apesar de ter também a propriedade `TEXT`. Isso acontece porque sua classe é `TMEMO` e não `TEDIT`. Para limpá-lo junto com os objetos `TEdit` devemos usar uma classe ancestral que seja a mesma para ambos. Neste caso, verificando o Help do Delphi podemos observar que ambos `TEdit` e `TMemo` são descendentes da classe `TCustomEdit` (que detém a propriedade `Text`). Sendo assim, poderíamos usar `TCustomEdit` e o campo OBS também teria sido “limpo”.

EVENTOS MAIS UTILIZADOS NO DELPHI:

Alguns eventos estão presentes na maioria dos objetos. Vamos detalhá-los aqui.

Obs: Alguns trechos foram retirados do help do Delphi 2005. Para maiores detalhes consulte-o.

ONCREATE

Este evento ocorre durante a criação de um objeto na memória. Você poderia, por exemplo, inicializar variáveis ou mesmo criar outros objetos neste evento.

Ex:

```
procedure TForm1.FormCreate(Sender: TObject) // evento oncreate do formulário;
begin
    LimpaCamposTela; // procedure que limpará os campos da tela
    v_DadosSalvos := False; // inicializando uma variável
end;
```

ONKEYPRESS

Use este evento para capturar as teclas pressionadas sobre o objeto. Este evento captura apenas teclas da tabela ASC II, utilizando para isso uma variável **KEY** do tipo *char*. Teclas como shift ou F1 não são possíveis de serem capturadas neste evento. Não é possível também capturar seqüências de teclas, como Shift+A. Para estes casos utilize o evento `onKeyDown` ou `onKeyUp`. A variável **KEY** representa o caractere ASC II da tecla pressionada. Para anularmos a tecla pressionada atribuímos **#0** ao parâmetro **KEY**.

Ex:

```
// neste exemplo capturamos a tecla ENTER através de seu caractere ASC II
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char); // evento de um TEdit
begin
    if key = #13 then begin
        showmessage('Você pressionou ENTER.');
```

```
        Edit2.setfocus; // manda o foco para o componente edit2
    end;
end;
```

```
// neste exemplo capturamos teclas através de seu valor e não se seu nº na tabela ASC II
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
    if key in ['a' .. 'z', 'A' .. 'Z'] then begin // permite apenas as letras de 'a' à 'Z' maiúsculas e minúsculas
        showmessage('Você pressionou a tecla: [' + key + '] '); // exibe a tecla pressionada
    end;
end;
```

ONKEYDOWN

Este evento faz tudo o que o evento **onKeyPress** faz porém é capaz de processar sequências de teclas, como CTRL+Z por exemplo. Também é capaz de capturar teclas como TAB, F1, CAPS LOCK, NUM LOCK, INSERT, etc... A variável **KEY** é do tipo **word** (um inteiro que não aceita negativo). Para anularmos a tecla pressionada atribuímos **0** ao parâmetro **KEY**.

A variável **Shift** é do tipo **TShiftState** e pode conter um dos seguintes valores:

Value	Meaning
ssShift	The Shift key is held down.
ssAlt	The Alt key is held down.
ssCtrl	The Ctrl key is held down.
ssLeft	The left mouse button is held down.
ssRight	The right mouse button is held down.
ssMiddle	The middle mouse button is held down.
ssDouble	The mouse was double-clicked.

Ex:

```
// neste exemplo exibimos o nº da tecla pressionada. Ideal para se descobrir o nº de uma tecla.
procedure TForm1.Edit2KeyDown(Sender: TObject; var Key: Word; Shift: TShiftState);
begin
  showmessage( 'Você pressionou a tecla nº: ' + intToStr(key) );
end;

// Este exemplo foi programado em um objeto FORM. Ele anula a sequência de teclas
// ALT+F4, sequência esta que fecha um formulário.
//Para que este exemplo funcione, a propriedade KeyPreview do formulário deve estar como TRUE.
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word; Shift: TShiftState);
begin
  if (key = 115) and (ssALT in Shift) then // 115 é o nº da tecla F4. utilizamos o operador IN pois Shift
                                         // pode conter um conjunto de teclas, como CTRL e ALT.
    key := 0 // anulamos a tecla pressionada;
end;
```

ONCLICK

Este evento é disparado sempre que clicamos sobre o objeto.

Ex:

```
procedure TForm1.FormClick(Sender: TObject);
begin
  showmessage('Você clicou no formulario!');
end;
```

ONDESTROY

Este evento é disparado quando um objeto é destruído da memória. Por exemplo, quando executamos o método FREE de um objeto o removemos da memória. Neste instante o evento onDestroy é disparado e poderíamos, por exemplo, destruir outros objetos. Dependendo do tipo de aplicação este evento é pouco utilizado.

ONDBLCLICK

Este evento é disparado sempre que executamos um duplo clique sobre o objeto em questão. Este evento **não** ocorrerá se o evento **OnClick** também foi programado.

Ex:

```
procedure TForm1.FormDbClick(Sender: TObject);
begin
  showwwmessage("Você deu um duplo clique no formulario!");
end;
```

ONMOUSEMOVE

Este é um evento pouco utilizado em aplicações normais (cadastros p.ex.). Ele é disparado sempre que o mouse é movimentado sobre o objeto. É possível ainda saber o status de teclas como CTRL, ALT, SHIFT, etc. além das posições X e Y do mouse.

Ex:

```
procedure TForm1.Edit1MouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);
begin
  showMessage('Tira esse mouse daqui!');
end;
```

ONENTER

Este evento é disparado quando o objeto recebe o foco.

Ex:

```
procedure TForm1.Edit1Enter(Sender: TObject);
begin
  (sender as TEdit).color := clYellow; // altera a cor do objeto passado no parâmetro SENDER
end;
```

ONEXIT

Este evento é disparado quando o objeto perde o foco.

Ex:

```
procedure TForm1.Edit1Exit(Sender: TObject);
begin
  (sender as TEdit).color := clWhite; // altera a cor do objeto passado no parâmetro SENDER
end;
```

ASSOCIANDO UM PROCEDIMENTO À UM EVENTO EM TEMPO DE EXECUÇÃO

A associação entre procedimentos e eventos pode ser feita também durante a execução do programa. Para isso, basta usar o nome do evento como uma propriedade e atribuir um valor que é o nome do procedimento de evento. Por exemplo, para associar o evento OnEnter do objeto Edit1 com o procedimento "EditRecebeuFoco", basta usar o seguinte comando:

```
Edit1.OnEnter := EditRecebeuFoco;
```

A CLASSE TWINCONTROL

Esta é a uma das principais classes do Delphi. Dentre outras características, seus descendentes podem receber foco. Alguns também, como o TPANEL, podem servir de container para outros componentes.

PRINCIPAIS PROPRIEDADES

ControlCount : Integer	Retorna a quantidade de controles filhos
Controls: TControls	Um vetor que armazena todos os controles filhos. Inicia do Zero. Ex: Controls[1]
Enabled : Boolean	Indica se o componente está habilitado ou não. Ex: enabled := false;
Parent : TWinControl	Indica quem é o "Pai" do objeto
ComponentCount : Integer	Retorna a quantidade de controles pertencentes, filhos, netos,etc.
Components : TComponent	Um vetor que armazena todos os controles pertencentes, filhos, netos, etc..
Visible : Boolean	Indica se o componente é visível ou não
TabOrder : Integer	Ordem em que o foco é direcionado através da tecla TAB
TabStop : boolean	Indica se o componente será focado através da tecla TAB

PRINCIPAIS MÉTODOS

CanFocus : Boolean	Indica se o componente pode receber foco
Create	Cria o componente
Free	Destrói o componente
Focused : Boolean	Indica se o componente detém o foco
SelectNext	Muda o foco para o próximo componente de acordo com o TabOrder
SetFocus	Muda o foco para o componente
FindComponent : TComponent	Indica se um dado componente é filho deste.

ACESSANDO DIVERSOS COMPONENTES DE UM TWINCONTROL

É possível realizar operações sobre diversos componentes de uma única vez quando utilizamos as propriedades **ComponentCount** , **components[]**, **controcount** e **controls[]**. Como exemplo,

imagine o seguinte trecho de código no evento **onclick** do botão  abaixo:

EXEMPLO 1

```
procedure TForm1.btnExecutarClick(Sender: TObject);
var i : integer;
begin
    for i := 0 to Form1.controlcount -1 do begin
        if Form1.controls[i] is TEdit then begin
            (Form1.controls[i] as TEdit).Color := clyellow;
        end;
    end;
```

end;

Formulário antes da execução do código.

Após executar o código, o formulário ficará assim:

Formulário após a execução do código.

```
procedure TForm1.btnExecutarClick(Sender: TObject);
var i : integer;
begin
  for i := 0 to Form1.controlcount - 1 do begin
    if Form1.controls[i] is TEdit then begin
      (Form1.controls[i] as TEdit).Color := clyellow;
    end;
  end;
end;
```

Vamos analisar o código: Foi criada uma estrutura de repetição (for i := 0 to Form1.controlcount - 1 do begin) que verifica todos os components que são filhos diretos do formulário (**controlcount** conta esses componentes e **Controls** é o vetor que faz referência à esses componentes) e que são da classe **TEdit** (if Form1.controls[i] is TEdit then begin).


Então, caso o componente em questão seja da classe **TEdit**, ele será pintado de **amarelo** ((Form1.controls[i] as TEdit).Color := clyellow;).

Mas porque o componente **Edit5**, que está dentro do panel cinza não foi pintado de amarelo, já que ele também é da classe **TEdit**?

Isso aconteceu porque a propriedade **componentcount** conta os componentes filhos diretos do formulário (no caso, o **Form1**) e o objeto **Edit5** é filho direto do **Panel1**, e não do formulário! (Faça um teste: coloque um objeto dentro de um panel e tente arrastá-lo para fora do panel. Você não vai

conseguir!). OBS: no código acima, o objeto FORM1 pode ser removido, já que o Delphi entende que propriedades e métodos onde não está identificado o objeto pertencem ao formulário.

EXEMPLO 2

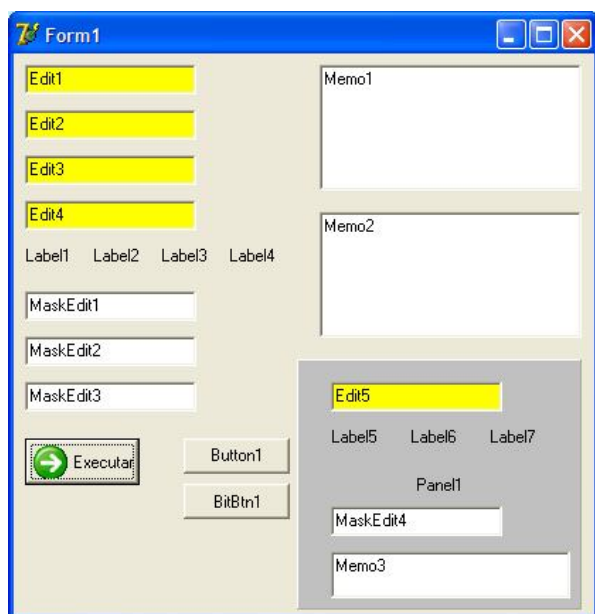
Agora, testaremos o seguinte código (no mesmo formulário) no evento onclick do botão  :

```
procedure TForm1.btnExecutarClick(Sender: TObject);
var i : integer;
begin
    for i := 0 to Form1.ComponentCount -1 do begin
        if Form1.Components[i] is TEdit then begin
            (Form1.Components[i] as TEdit).Color := clyellow;
        end;
    end;
end;
```

O que mudou foi:

De: controlCount para componentCount

De: controls para components

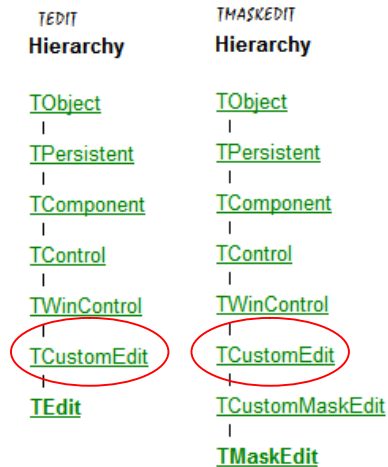


Formulário após execução

Observe que agora o componente **Edit5** foi pintado de amarelo. Isso porque **componentCount** conta TODOS os objetos, inclusive aqueles que não filhos diretos do formulário.

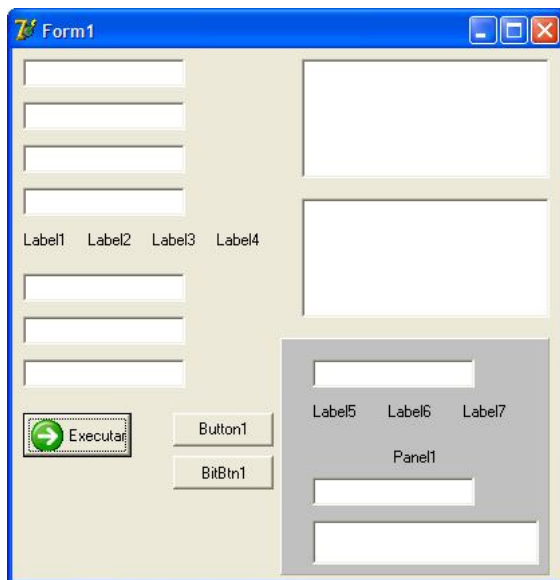
EXEMPLO 3

Consultando o HELP do Delphi, é possível verificar na Hierarquia de componentes que o ancestral comum ao TEdit e o TMaskEdit é a classe **TCustomEdit**.



Portanto, é possível acessar propriedades dessas duas classes através de sua classe ancestral em comum, **desde que essa classe ancestral possua a propriedade/método/evento que desejamos acessar**.

No Código abaixo, todos os objetos colocados neste formulário que pertencem à classe **TCustomEdit** serão “limpos”. Observe que até os objetos da classe **TMemo** sofreram a ação do código. Isso aconteceu porque eles também são descendentes da classe **TCustomEdit**!



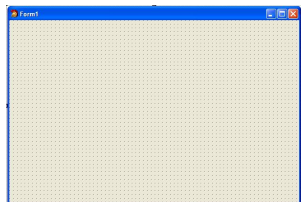
```

procedure TForm1.btnExecutarClick(Sender: TObject);
var i : integer;
begin
  for i := 0 to Form1.ComponentCount - 1 do begin
    if Form1.Components[i] is TCustomEdit then begin
      (Form1.Components[i] as TCustomEdit).clear;
    end;
  end;
end;

```

ALGUNS COMPONENTES VISUAIS DO DELPHI 2005

PRINCIPAIS EVENTOS DO FORMULÁRIO.



Os formulários (objeto Form) são os pontos centrais para o desenvolvimento Delphi. Você se utilizará deles para desenhar sua comunicação com o usuário, colocando e organizando outros objetos. Para criar um novo: **FILE→NEW→FORM - DELPHI FOR WIN32**. Ele tem diversos eventos, porém os principais são:

➤ **onClose**

Este evento ocorre sempre ocorre uma tentativa de se fechar o formulário, por exemplo, ao executar-se o método **close**. Podemos neste evento por exemplo fechar as tabelas utilizadas. O parâmetro **ACTION** pode alterar o funcionamento deste evento. Veja:

Valor	Significado
caNone	Não será permitido fechar o formulário.
caHide	O formulário é colocado como invisível.
caFree	O formulário é fechado e a memória utilizada para ele é liberada não sendo este mais acessível.
caMinimize	O formulário é apenas minimizado.

Ex:

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  showmessage('Não é possível fechar este form.');
```

action := caNone;

```
end;
```

➤ **onShow**


Este evento ocorre sempre que o formulário é exibido. Por exemplo, ao executar-se o método **show**. Podemos neste evento por exemplo limpar os componentes da tela, inicializar variáveis, abrir tabelas, etc.


Ex:


```
procedure TForm1.FormShow(Sender: TObject);
begin
  showmessage('Exibindo o formulário.');
```


end;


PALHETA STANDARD

 **TMainMenu** Este componente é utilizado para criar menus. Sua utilização é bem simples: coloque um na tela e execute um duplo - clique para começar a criar o menu. Todos os itens do menu são objetos do tipo TmenuItem e possuem o evento onClick.

 **TPopupMenu** Este componente permite a exibição de um menu popUp de forma simples. Coloque um na tela e associe-o à propriedade *popupMenu* dos objetos (nem todos a possuem). Para criar o menu dê um duplo – clique sobre o objeto popupMenu e monte o da mesma forma que no objeto TMainMenu.


 **TLabel** Utilizado para exibir uma mensagem no formulário.


 **TEdit** Descendente da classe **TWinControl**. Este componente permite que o usuário entre com um texto. Sua principal propriedade é a TEXT que armazena o que foi digitado.

 **TMemo** Descendente da classe **TWinControl**. Utilizado para digitação de texto onde é necessário mais que uma linha. Sua principal propriedade é a LINES, onde é possível acessar o texto digitado através da propriedade TEXT ou então linha por linha através de LINES[linha]. Para adicionar um texto podemos utilizar o método LINES.ADD() ou mesmo através de TEXT.


 **TButton** Descendente da classe **TWinControl**. Este botão não permite a utilização de imagens.

 **TCheckBox** Descendente da classe **TWinControl**. Este objeto armazena o valor de ligado ou não em sua propriedade boolean denominada checked.

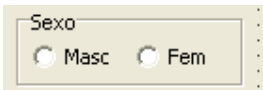
 **TRadioButton** Descendente da classe **TWinControl**. Este objeto é utilizado quando possuímos várias opções, porém apenas uma deve ser selecionada. Utilize-o dentro do componente

 **TGroupBox** para que seja criado um grupo de opções e este não seja agrupado à outros grupos. Sua principal propriedade é a checked que indica se objeto foi selecionado ou não.

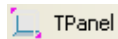
 **TListBox** Descendente da classe **TWinControl**. Este objeto permite a seleção de uma ou mais opções.

 **TComboBox** Descendente da classe **TWinControl**. Este objeto é utilizado quando possuímos várias opções, porém apenas uma deve ser selecionada. Utilize a propriedade ItemIndex para saber se alguma opção foi selecionada. Esta propriedade inicia do Zero e -1 indica que nada foi selecionado.

 **TRadioGroup** Descendente da classe **TWinControl**. Este componente é similar ao TRadioButton, porém seus itens são criados através da propriedade ITENS, onde cada linha indica uma nova

opção. Ex:  Para saber se uma determinada opção foi selecionada devemos utilizar a propriedade ItemIndex, que indicará -1 caso nada tenha sido selecionado. Vale lembrar que os itens iniciam do ZERO.

 **TGroupBox** Descendente da classe **TWinControl**. Utilizado para agrupar componentes.



Descendente da classe **TWinControl**. Utilizado para agrupar componentes.

MANIPULANDO DATA E HORA NO DELPHI

O Tipo de dado utilizado no Delphi para armazenar Data e Hora é o **TDateTime**. Apesar de ter recebido nome próprio ele é do tipo **DOUBLE**, ou seja, igual a um dos tipos predefinidos para a representação de números reais.

A parte **inteira** é utilizada para armazenar os **Dias** que se passaram desde o dia 30/12/1899. A parte **fracionada** armazena as **Horas** do dia. Veja alguns exemplos na tabela abaixo:

Valor	Data e hora
0	30/12/1899, 0:00h
1,25	31/12/1899, 6:00h
-1,75	29/12/1899, 18:00h
35291,625	14/08/1996, 15:00h

Sendo *TDateTime* do Tipo *Double*, algumas operações numéricas passam a ter um significado especial. Por exemplo, para comparar duas datas, basta compará-las como se estivessemos comparando dois números:

```
Var
  DataPagamento, DataVencimento : TdateTime;
.....
If DataPagamento > DataVencimento then
  Showmessate('Pago no prazo.');
```

As seguintes operações são válidas para o tipo *TDateTime*:

Sendo D1, D2, D3 : *TDateTime*; e Dias : *Integer*;

D1 := D1 + 1; // D1 passa a ser o dia seguinte

D1 := D1 - 1; // D1 passa a ser o dia anterior

D3 := D1 - D2; // diferença entre as 2 datas

Dias := Trunc(D1 - D2); // Diferença em dias entre as datas (Trunc pega só a parte inteira)

D1 := D1 + StrToTime('00:30:00'); //Adiciona 30 min. à hora. Pode ocorrer mudança de dia.

D1 := StrToDate('25/12/2005'); // atribui uma nova data, neste caso a hora será perdida.

D2 := StrToTime('14:25:00'); // atribui uma nova hora, neste caso a data será perdida.

D3 := StrToDate('25/12/2005') + StrToTime('14:25:00'); // atribui uma nova data e hora.

D2 := Frac(D3); // pega só a hora da D3. Frac retorna a parte fracionada.

Existem diversas rotinas para manipular Data e Hora que podem ser encontradas no help do Delphi (especialmente no Delphi 7) intituladas “*Date Time Routines*”. Dentre as principais podemos destacar:

Now : retorna a data e hora atual;

Date : retorna a data atual;

Time : retorna a hora atual;

StrToDate, StrToTime, TimeToStr, DateToStr, DateTimeToStr, StrToDateTime : conversão.

DayOf, MonthOf, YearOf : Retornam dia, mês e ano respectivamente de uma data informada.

FormatDateTime : Importantíssima função para formatação de data e hora. Não colocaremos aqui todos os seus parâmetros devido a grande quantidade.

O exemplo abaixo:

```
label1.caption := formatDateTime("São Bernardo do Campo," dd "de" mmmm "de" yyyy', date);
```

Produzirá o resultado:

São Bernardo do Campo, 17 de setembro de 2005

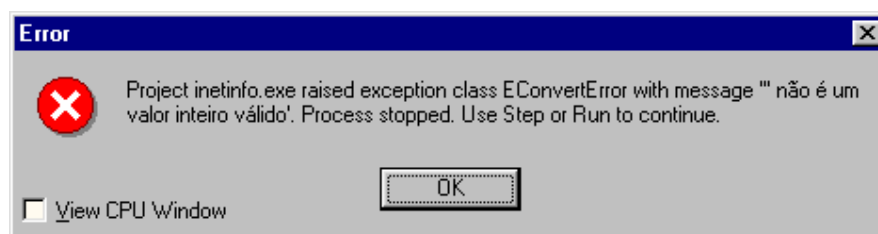
Obs: As “ASPAS” estão em **vermelho**. Elas separam a parte “fixa” das variáveis.

APLICAÇÕES ROBUSTAS - TRATAMENTO DE EXCEÇÕES

Fonte: <http://www.geocities.com/SiliconValley/Bay/1058/except.html>

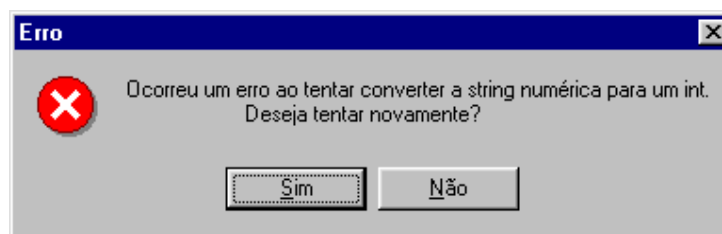
O tratamento de exceção é um mecanismo capaz de dar robustez a uma aplicação, permitindo que os erros sejam manipulados de uma maneira consistente e fazendo com que a aplicação possa se recuperar de erros, se possível, ou finalizar a execução quando necessário, sem perda de dados ou recursos.

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



No exemplo acima, poderia-se clicar no botão OK e continuar a executar a aplicação. Mas muitas vezes o erro ocorrido impede a continuação da operação, ou leva a perda de informações valiosas.

O ideal seria que se pudesse tratar estes erros ocorridos, evitando a perda de dados ou a necessidade de encerrar a aplicação. Além de tratar o erro, a rotina de tratamento de erros poderia enviar ao usuário uma mensagem em português, mais significativa. A forma mais simples para responder a uma exceção é garantir que algum código limpo é executado. Este tipo de resposta não corrige o erro, mas garante que sua aplicação não termine de forma instável. Normalmente, usa-se este tipo de resposta para garantir a liberação de recursos alocados, mesmo que ocorra um erro. O tratamento mais simples seria uma simples mensagem ao usuário com a proposta dele tentar novamente a execução da operação que tenha causado o erro, conforme podemos ver no exemplo abaixo:



Outra forma de se responder a uma exceção é tratando o erro, ou seja, oferecendo uma resposta específica para um erro específico. Ao tratar o erro destrói-se a exceção, permitindo que a aplicação continue a rodar, mas o mais ideal mesmo é que além da mensagem, coloque os procedimentos devidos que sua aplicação deverá fazer caso ocorra uma exceção. Um exemplo disto é uma operação de gravação em uma tabela, caso ocorra um erro, ela deverá ser cancelada.

Existe também a prática da geração de logs que gravam no disco, um arquivo texto contendo todos os dados possíveis do erro ocorrido de forma que possa chegar a causa o mais rápido possível.

EXCEÇÕES

Exceções são classes definidas pelo Delphi para o tratamento de erros. Quando uma exceção é criada, todos os procedimentos pendentes são cancelados e, geralmente é mostrada uma mensagem de erro para o usuário. As mensagens padrão nem sempre são claras, por isso é indicado criar seus próprios blocos protegidos.

BLOCOS PROTEGIDOS

Bloco protegido é uma área em seu código que é encapsulada em uma instrução que ao invés de executar a linha de código ele tentará executá-la. Se conseguir beleza, prossegue com o programa, se não conseguir então ele cria uma resposta a este insucesso em um bloco de código resguardando de erros que podem parar a aplicação ou perder dados. Um bloco protegido começa com a palavra reservada **try** e termina com a palavra reservada **end**. Entre essas palavras determinam-se os comandos protegidos e a forma de reação aos erros.

Quando se define um bloco protegido, especifica-se respostas a exceções que podem ocorrer dentro deste bloco. Se a exceção ocorrer, o fluxo do programa pula para a resposta definida, e após executá-la, abandona o bloco.

Um bloco protegido é um grupo de comandos com uma seção de tratamento de exceções.

```
try
  A := StrToFloat(Edtit.Text);
  B := StrToFloat(Edtit.Text);
  ShowMessage(Format('%f / %f = %f', [A, B, A + B]));
except
  ShowMessage('Números inválidos.');
```

end;

A aplicação irá executar os comandos na parte **except** somente se ocorrer um erro. Se na parte **try** chamar uma rotina que não trata erros, e um erro ocorrer, ao voltar para este bloco a parte **except** será executada. Uma vez que a aplicação localiza um tratamento para a exceção ocorrida, os comandos são executados, e o objeto exceção é destruído. A execução continua até o fim do bloco. Dentro da parte **except** define-se um código a ser executado para manipular tipos específicos de exceção.

Algumas vezes você pode precisar especificar quais exceções quer tratar, como mostrado abaixo.

```
try
  Soma := StrToFloat(EdtSoma.Text);
  NumAlunos := StrToInt(EdtNum.Text);
  ShowMessage(Format('Média igual a %f.', [Soma / NumAlunos]));
except
  on EConvertError do
    ShowMessage('Valor inválido para soma ou número de alunos.');
```

on EZeroDivide **do**

```
  ShowMessage('O número de alunos tem que ser maior que zero.');
```

else

```
  ShowMessage('Erro na operação, verifique os valores digitados.');
```

end;

PRINCIPAIS EXCEÇÕES

O Delphi define muitas exceções, para cada erro existe uma exceção correspondente.

Classe	Descrição
Exception	Exceção genérica, usada apenas como ancestral de todas as outras exceções
EAbort	Exceção silenciosa, pode ser gerada pelo procedimento Abort e não mostra nenhuma mensagem
EAccessViolation	Acesso inválido à memória, geralmente ocorre com objetos não inicializados
EConvertError	Erro de conversão de tipos
EDivByZero	Divisão de inteiro por zero
EInOutError	Erro de Entrada ou Saída reportado pelo sistema operacional
EIntOverflow	Resultado de um cálculo inteiro excedeu o limite
EInvalidCast	TypeCast inválido com o operador as
EInvalidOp	Operação inválida com número de ponto flutuante
EOutOfMemory	Memória insuficiente
EOverflow	Resultado de um cálculo com número real excedeu o limite
ERangeError	Valor excede o limite do tipo inteiro ao qual foi atribuída
EUnderflow	Resultado de um cálculo com número real é menor que a faixa válida
EVariantError	Erro em operação com variant
EZeroDivide	Divisão de real por zero
EDatabaseError	Erro genérico de banco de dados, geralmente não é usado diretamente
EDBEngineError	Erro da BDE, descende de EDatabaseError e traz dados que podem identificar o erro

BLOCOS DE FINALIZAÇÃO

Blocos de finalização são executados sempre, haja ou não uma exceção. Geralmente os blocos de finalização são usados para liberar recursos. Entre estes recursos estão : arquivos, memória, recursos do Windows, objetos.

```

FrmSobre := TFrmSobre.Create(Application);
try
  FrmSobre.Image.LoadFromFile('Delphi.bmp');
  FrmSobre.ShowModal;
finally
  FrmSobre.Release;
end;

```

Você pode usar blocos de proteção e finalização aninhados

```

FrmOptions := TFrmOptions.Create(Application);
try
  FrmOptions.ShowModal;
  try
    Tbl.Edit;
    TblValor.AsString := EdtValor.Text;
  except
    on EDBEngineError do
      ShowMessage('Alteração não permitida.');
```

```

    FrmOptions.Release;
end;

```

A aplicação sempre executará os comandos inseridos na parte **finally** do bloco, mesmo que uma exceção ocorra. Quando um erro ocorre no bloco protegido, o programa pula para a parte **finally**, chamada de código limpo, que é executado. Mesmo que não ocorra um erro, estes comandos são executados.

No código a seguir, foi alocada memória e gerado um erro, ao tentar-se a divisão por 0 (zero). Apesar do erro, o programa libera a memória alocada:

```

Procedure TForm1.Button1click (Sender : Tcomponent );
Var
    Ponteiro : Pointer;
    Inteiro, Dividendo : Integer;
Begin
    Dividendo:= 0;
    GetMem(Ponteiro, 1024);
    Try
        Inteiro:= 10 div dividendo;
    Finally
        FreeMem(Ponteiro, 1024);
    End;
End;

```

GERAÇÃO DE EXCEÇÕES

Você pode provocar uma exceção usando a cláusula **raise**.

```

raise EDatabaseError.Create('Erro ao alterar registro.');
```

Também é possível criar seus próprios tipos de exceções.

```

type
    EInvalidUser = class (Exception);

```

```

raise EInvalidUser.Create('Você não tem acesso a essa operação.');
```

Se você quiser que uma exceção continue ativa, mesmo depois de tratada, use a cláusula **raise** dentro do bloco de tratamento da exceção. Geralmente isso é feito com exceções aninhadas.

```

try
    Tbl.Edit;
    TblContador.Value := TblContador.Value + 1;
    Tbl.Post;
except
    ShowMessage("Erro ao alterar contador.");
    raise;
end;

```

EXIBINDO O ERRO GERADO PELA EXCEÇÃO

Para exibir a mensagem de erro que foi gerada pela exceção devemos criar uma programação especial no bloco `except`. Ex:

```
try
  A := StrToFloat(Edit1.Text);
  B := StrToFloat(Edit2.Text);
  ShowMessage(Format('%f / %f = %f', [A, B, A + B]));
except
  On E:Exception do
    begin
      ShowMessage('Números inválidos. Erro gerado: ' + E.message);
    end;
end;
```

EXCEÇÕES SILENCIOSAS

Pode-se definir exceções que não mostrem um quadro de mensagem para o usuário quando aparecem. São chamadas exceções Silenciosas. O caminho mais curto para criar esta exceção é através da procedure *Abort*. Esta procedure automaticamente gera uma exceção do tipo *Eabort*, que abortará a operação sem mostrar uma mensagem.

O exemplo abaixo aborta a operação de inclusão de itens em um Listbox quando tentamos inserir o terceiro elemento:

```
For i := 1 to 10 do
  Begin
    Listbox.Items.Add(IntToStr(i));
    If i = 3 then
      Abort;
  End;
```

MANIPULANDO EXCEÇÕES GLOBAIS

O Delphi oferece um evento chamado *OnException*, ligado à a classe *TApplication*, que permite manipular qualquer exceção que ocorra em seu programa, mesmo que não sabendo em que parte do programa ela ocorreu. Inicialmente, deve-se criar manualmente, como um método de *TForm1*, a chamada para o método que tratará os erros :

```
TForm1 = Class (TForm)
Procedure Trata_Erros (Sender : TObject ; E : Exception );
End;
```

Depois de declarar a procedure, deve-se atribuí-la ao evento *OnException* de *TApplication*. A atribuição é feita no evento *OnCreate* do formulário principal :

```
Procedure TForm1.FormCreate ( Sender : TObject );
Begin
  Application.OnException := Trata_Erros;
End;
```

O método *Trata_Erros* será agora chamado quando qualquer exceção ocorrer. Pode-se determinar mensagens para erros específicos, ou mensagens gerais :

Procedure TForm1.Trata_Erros (Sender : TObject; E : Exception);

Begin

If E is EdatabaseError **then**

Showmessage('Erro no banco de dados');

Else

Showmessage('Há erros no programa');

End;

ENTENDENDO UM POUCO SOBRE O QUE É UM GPF

A bem da verdade as exceções não tratadas ou mal tratadas em uma aplicação são as uma das causas do GPF (Falha geral de proteção) que ocorrem no Windows e que geralmente o derrubam. Como isto acontece? Se seu programa conseguir sobreviver à uma exceção sem que ele tenha caído (O que não é muito difícil em se tratando de exceções silenciosas), ele pode perder o seu endereçamento da memória. Daí começar a acessar uma área protegida, ou uma área que está sendo usada por outro programa, é questão de tempo. Um problema destes, normalmente acaba com uma mensagem de erro do Windows que fecharia tanto este aplicativo como os outros envolvidos no problema e em seguida, uma simples reinicialização da máquina volta ao normal. Mas podem ocorrer problemas mais sérios como uma colisão com a FAT, o que resultaria na perda de todo ou parte do conteúdo do HD. Obviamente que um caso destes é muito raro de acontecer, mas que existe a probabilidade existe!

O GPF (General Protection Faults) é um evento de hardware, causado pela própria CPU. No Windows ocorrem basicamente por culpa dos aplicativos, que não possuem um tratamento consistente de suas exceções internas. Como consequência deste problema pode ocorrer os seguintes eventos:

- Tentativa, pela aplicação, de acessar memória fora da área determinada pelo sistema operacional;
- Tentativa de acessar um ponteiro inválido, podem fazer com que a aplicação acesse uma área protegida;
- Tentativa de acessar uma área de memória que está sendo usada por outro aplicativo ou biblioteca (dll);
- Tentativa de execução de dados, ao invés de código (programa);
- Tentativa de efetuar uma operação binária inválida (Não confunda com operação aritmética);
- Tentativa de acesso indevido ao hardware;

CONFIGURANDO O DELPHI PARA NÃO “PARAR” EM EXCEÇÕES

O Delphi vem configurado para parar em todas as exceções, mesmo aquelas onde há utilização de blocos protegidos. Para desligar esta opção:

No Delphi 7: Menu TOOLS | DEBUGGER OPTIONS | LANGUAGE EXCEPTIONS | desmarque a opção “Stop on Delphi Exceptions”

No Delphi 2005: Menu TOOLS | OPTIONS | DEBUGGER OPTIONS |

CONCLUSÃO:

Depois de ler este artigo, você pode concluir que não deve, sob nenhuma circunstância, negligenciar o tratamento de exceções dentro de sua aplicação uma vez que sempre que elas ocorrem, acabam afetando não só a esta referida aplicação mas como outras ainda. Um exemplo disto é o caso de uma aplicação que dá um *lock* em uma tabela no banco de dados e em seguida tentará abri-la. Caso

ocorra uma exceção que aborte o programa, a aplicação fecha sem que nada ocorra com o Windows. Mas a Tabela está lá no banco bloqueada e, conforme for o SGDB, nem um Restart ou na máquina ou no banco conseguirão desbloqueá-la, causando transtornos e perda de tempo tentando solucionar um problema que poderia ter sido evitado perfeitamente. Outro exemplo disto, é o caso de exceções silenciosas que foram criadas e consistidas incorretamente. Ai elas se tornarão uma mina enterrada dentro de seu executável. Uma exceção destas pode danificar um banco de dados inteiro ou uma partição do Disco Rígido, conforme a operação que se tentou fazer mas que não foi bem sucedida e foi mal consistida. Pior do que isto, sem que o usuário perceba o que está acontecendo.

Se o autor deste software tivesse olhado com mais atenção as possíveis exceções que esta aplicação poderia causar, talvez a incidência destes imprevistos seriam bem menores ou quase que nulos.

ARQUIVOS INI

Os arquivos .INI são uma boa maneira de se salvar as configurações de uma aplicação para que quando tal aplicação for executada novamente estas configurações sejam restauradas. Esses arquivos vêm sendo bastante utilizados desde as versões mais antigas do Windows, embora nas últimas versões a Microsoft tenha estimulado o uso do Registro do Windows.

Os arquivos .INI são arquivos de texto comuns que podem ser visualizados e editados no Bloco de Notas do Windows ou em qualquer outro editor de texto de sua preferência. Contudo essa extensão de arquivo tem algumas exigências quanto ao seu layout e uma restrita quantidade de tipos de dados disponíveis. Seu layout é dividido em blocos ou seções onde seus nomes são colocados entre colchetes [] e cada seção pode conter vários atributos que podem ser apenas dos tipos string, integer ou boolean.

Para trabalhar com arquivos .INI o Delphi dispõe para os desenvolvedores a classe ***TIniFile***. Ao criarmos um objeto para a classe *TIniFile* e o ligar a um arquivo, este estará disponível para leitura e escrita de dados.

A classe dos arquivos .INI, a *TIniFile* possui três métodos de leitura e três para a escrita que são: *ReadString*, *ReadInteger*, *ReadBool* e *WriteString*, *WriteInteger*, *WriteBool*, respectivamente. Existem ainda outros métodos que definem valores padrão de leitura para os arquivos .INI e que permitem que uma seção inteira seja lida ou apagada.

Criar um arquivo .INI é super simples, basta chamar o construtor *TIniFile.Create*, passar um nome, em formato string, para o arquivo, como por exemplo, LoginIni.ini. Depois é só adicionar à cláusula uses a unidade *IniFiles*. Por padrão os arquivos .INI são criados na pasta raiz do Windows (C:\Windows), no entanto é mais seguro e recomendado cria-lo dentro da mesma pasta onde está seu executável, para isso informe junto com o nome também o caminho completo, por exemplo, C:\Programa\Teste.ini.

Ex: Vamos criar um programa que será capaz de armazenar o nome e o sexo de uma pessoa em um arquivo INI localizado no c:\ chamado Teste.INI. Também será possível carregar esses dados.

Observe que toda a operação é realizada em um bloco protegido. Isso acontece porque temos que criar o objeto não visual *TIniFile* e durante o processo precisamos nos assegurar que se um erro ocorrer este objeto será liberado da memória através do método *FREE*.

Utilizaremos dois tipos de dados : String para o nome e Integer para o Sexo (propriedade ItemIndex).

A tela do aplicativo é a seguinte:

Para utilizar um arquivo INI é necessário incluir na USES da unit o arquivo INIFILES.

Programação dos botões:

Botão Grava:

```
procedure TForm1.gravaBtnClick(Sender: TObject);
var arquivo : TIniFile;
begin
  try
    arquivo := TIniFile.Create('c:\teste.ini'); // cria um objeto para manipular o arquivo ini
    arquivo.WriteString('CONFIGURACAO', 'Nome', edNome.text); // Cria a seção "Configuração" se não existir
    arquivo.WriteInteger('CONFIGURACAO', 'Sexo', opSexo.ItemIndex);
    arquivo.Free; // apaga o objeto da memória
    ShowMessage('Dados gravados com sucesso!');
  except
    arquivo.Free; // apaga o objeto da memória
    ShowMessage('Ocorreu um erro ao gravar os dados!');
  end;
end;
```

Botão Recupera

```
procedure TForm1.RecupBtnClick(Sender: TObject);
var arquivo : TIniFile;
begin
  try
    arquivo := TIniFile.Create('c:\teste.ini'); // cria um objeto para manipular o arquivo ini
    edNome.text := arquivo.ReadString('CONFIGURACAO', 'Nome', '');
    opSexo.ItemIndex := arquivo.ReadInteger('CONFIGURACAO', 'Sexo', -1);
    arquivo.Free; // apaga o objeto da memória
  except
    arquivo.Free; // apaga o objeto da memória
    ShowMessage('Ocorreu um erro ao ler os dados!');
  end;
end;
```

Ex: do arquivo Teste.ini para o Nome João da Silva sexo Masculino:

```
[CONFIGURACAO]
Nome=João da Silva
Sexo=0
```

CRIANDO UMA BIBLIOTECA DE PROCEDURES, FUNÇÕES, ETC.

Uma biblioteca de funções, procedures, constantes, etc. nada mais é que uma **UNIT** (unidade .pas), ou seja um arquivo.pas sem um formulário atrelado. Nele podemos inserir procedures, funções, constantes, etc. que serão utilizadas em todo o programa. Por exemplo: função para validar CPF, constante contendo a cor dos formulários, procedure para validar datas, etc.

Para criar uma biblioteca é muito simples: vá ao menu FILE | NEW | OTHER | DELPHI FILES | UNIT.

A estrutura de uma unit vazia é a seguinte:

```
unit Unit1; { este é o nome da biblioteca. Não altere-o por aqui!!!!. Use a opção Save-as}

interface
    { aqui colocamos a cláusula USES e na sequência constantes, variáveis e o protótipo das
      funções/procedures }

implementation
    {aqui nós efetivamente programamos os protótipos informados na seção Interface }
end.
```

Exemplo Completo de uma biblioteca:

```
unit uUnit;

interface

uses SysUtils, Dialogs;

const x=50; {esta é uma constante}

var y : integer; {esta é uma variável global}

function soma(a,b : longint) : longint;
function converte( valor : string) : longint;
Function ValidaData( data : string; exibemsg : boolean =true ) : boolean;
function InteiroValido( valor : string; exibemsg : boolean ) : boolean;

implementation

// soma 2 numeros

function soma(a,b : longint) : longint;
Begin
    result := a + b;
end;

// tenta converter um string para um longint e se nao conseguir devolve zero.
function converte( valor : string) : longint;
begin
    try
        result := strToInt( trim(valor) );
    except
        result := 0;
    end;
end;
```

```

end;
end;

// valida uma data. pode exibir uma msg de erro dependendo do parâmetro exibemsg.
Function ValidaData( data : string; exibemsg : boolean =true ) : boolean;
Begin
  try
    StrToDate(data);
    result := true;
  except
    result := false;
    if exibemsg = true then
      showmessage('Data inválida!!!');
    end;
  end;
end;

// verifica se um numero inteiro eh valido
function InteiroValido( valor : string; exibemsg : boolean ) : boolean;
begin
  try
    strToInt(valor);
    result := true;
  except
    result := false;
    if exibemsg = true then
      showmessage('Número Inteiro Inválido!!!');
    end;
  end;
end;

end.

```

Para utilizar esta biblioteca basta colocar o nome dela na seção USES das outras units.

Ex:

```

unit uMain;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs, uUnit, StdCtrls, Buttons, Mask,
  ExtCtrls;

```



Depois utilizamos normalmente como se fosse uma função local:

```

procedure TForm1.Button1Click(Sender: TObject);
begin
  edit3.text := intToStr( soma( converte(edit1.text) , converte(edit2.text) ) );
end;

```


TRABALHANDO COM FORMULÁRIOS

Ao iniciar um novo projeto, o Delphi já cria automaticamente um novo formulário. Porém, um formulário pode ser muito pouco para nossas pretensões. O Delphi permite a criação e manipulação de vários formulários.

Para criar um novo formulário vá ao menu *File | New | Form – Delphi for Win32*.

Para exibir um formulário temos basicamente 2 opções: Os métodos **show** e **showmodal**. A principal diferença entre eles é que ao exibir um form com o evento **show** ainda nos é permitido acessar a tela anterior (a que evocou o novo form). Já o método **showmodal** faz com que a tela exibida seja a única possível de ser acessada no sistema. Isso é muito comum nas telas de opções do Delphi.

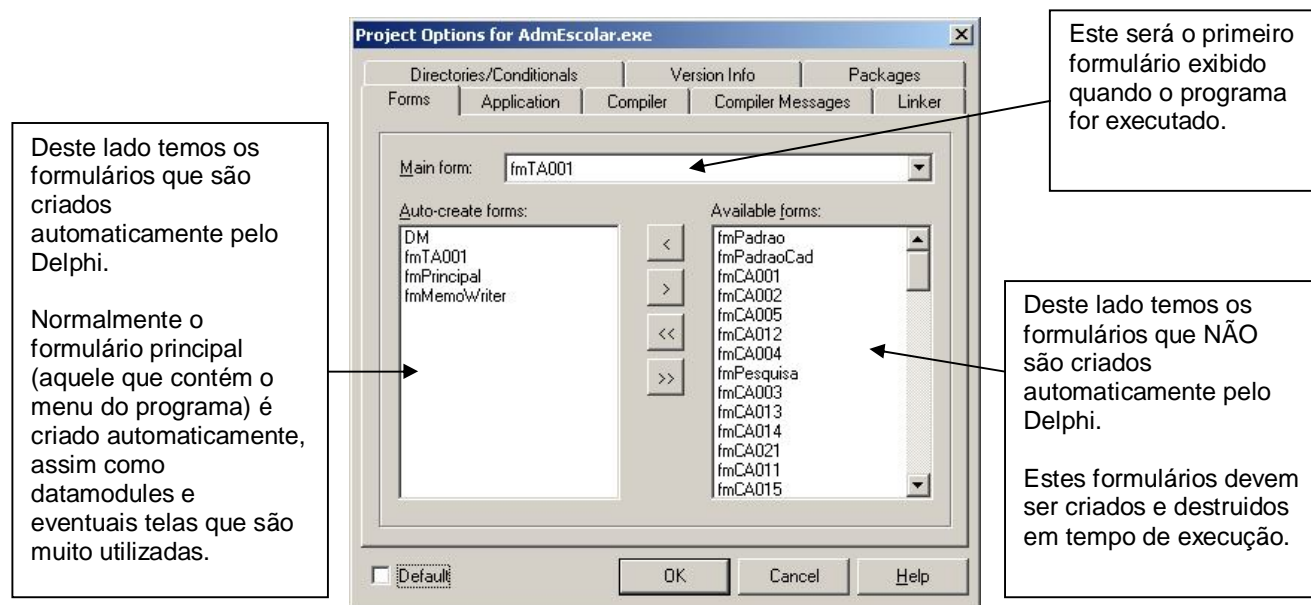
Outro detalhe interessante é que o método **showmodal** quando executado interrompe a execução das linhas de programa após a sua chamada. Sendo assim, esses “códigos pendentes” só serão executados quando o formulário exibido for fechado.

No exemplo abaixo, a mensagem só aparecerá após o Form2 ser fechado.

```
Form2.showmodal;
Showmessage('O Form2 foi fechado.');
```

Se tivéssemos utilizado o método **Show** no lugar do **ShowModal** a mensagem seria exibida assim que form fosse exibido.

O Delphi cria automaticamente todos os formulários do programa assim que o sistema é carregado. Isso pode ser muito ruim no caso de um sistema com muitos formulários pois consome muita memória. Seria interessante que os formulários fossem criados à medida que forem sendo utilizados, e destruídos quando não forem mais necessários. Para manipular quais formulários serão ou não criados automaticamente no Delphi, vá ao menu **PROJECT | OPTIONS** na aba **Forms**.



Para criar um formulário em tempo de execução faça o seguinte:

```
Formulário := TFormulario.create(Self) ;           ou  
Application.CreateForm(TFormulario,Formulario);
```

Onde:

Formulário é o nome da instância que representará o novo formulário, geralmente usa-se o mesmo nome dado ao formulário, ex: Form1.

TFormulario é o nome da classe do formulário. Ex : TForm1.

Create é o nome do método que cria um formulário na memória. O parâmetro indica o nome do PAI do formulário. Quando o PAI de um objeto é destruído seus filhos morrem também! (Triste né) A palavra reservada *SELF* diz respeito ao formulário em que se este código foi inserido. Poderíamos também ter a palavra *Application*, que representa a aplicação. Assim, o formulário só será destruído quando a aplicação terminar. Para nós isso não vai importar muito, pois o formulário será destruído assim que não for mais utilizado.

Para destruir o formulário exibido com o método *showmodal* podemos utilizar o código:

```
Formulário := TFormulario.create(Self) ; // cria o formulário  
Formulário.showmodal;                 // exhibe o form  
Formulário.Free;                     // Destrói o formulário, liberando memória
```

Pergunta: Poderíamos utilizar o *FREE* também após utilizar o método *show* no lugar do *showmodal*?

Resposta: Não! Pois o método *FREE* seria executado imediatamente após a chamada do método *show*, sem esperar que o formulário exibido seja fechado! Isso acontece pois como já mencionamos o método *show* não espera o novo formulário fechar para continuar executando os códigos pendentes após sua chamada.

```
Formulário := TFormulario.create(Self) ; // cria o formulário  
Formulário.show;                 // exhibe o form  
Formulário.Free;                 // Esta linha será executada antes mesmo do Formulário  
                                   // ser fechado, causando um erro.
```

E como resolvemos este impasse? Isso é realmente simples de se resolver, aliás, esta forma de solução é interessante até mesmo quando utilizamos o método *Showmodal*. Nós faremos com que o Formulário se “auto-destrua” após ser fechado. Assim não precisamos nos preocupar em que momento destruí-lo. Assim que o mesmo for fechado, será automaticamente destruído e liberado da memória.

Para tanto, adicione ao evento **onClose** do formulário (aquele que será exibido) a seguinte linha :

```
Action := caFree;
```

Essa linha diz que o formulário deve ser eliminado da memória assim que for fechado. Sendo assim, não precisaremos mais acionar o método *FREE* do objeto *Form*.

PRINCIPAIS TECNOLOGIAS DE ACESSO A DADOS

Retirado do apêndice “*Tecnologias Aplicáveis ao Delphi 7*” onde foi dado o devido crédito. O significado das siglas pode ser encontrado neste apêndice.

O Delphi possui diversas tecnologias de acesso a bancos de dados. São elas:

BDE (BORLAND DATABASE ENGINE):

- Acesso a bancos de dados locais: Paradox, dBase e servidores SQL, bem como tudo o que possa ser acessado por meio de drivers ODBC.
- A Borland a considera obsoleta, tende a sair do ambiente Delphi.
- A paleta BDE do Delphi 7 ainda mantém os componentes Table, Query, StoredProc e outros do BDE.

DBEXPRESS (DBX):

- Tecnologia de Acesso a Banco de Dados, sem depender do BDE.
- É chamada camada ‘magra’ de acesso a banco de dados, substituindo o BDE.
- Baseia-se no componente ClientDataSet (paleta Data Access), o qual tem a capacidade de salvar tabelas em arquivos locais – algo que a Borland tenta vender com o nome de MyBase.
- Disponível para as plataformas Windows e Linux.
- Basicamente não exigem nenhuma configuração nas máquinas dos usuários finais.
- Limitada em termos de recursos: pode acessar apenas servidores SQL (nada de arquivos locais, só se salvá-los localmente); não tem recursos de cache e oferece apenas acesso unidirecional (só para consultas, não inclui, altera, exclui, etc.)
- Indicada para produção de relatórios em HTML (só consultas).
- Para uso numa interface de edição de dados, deve ser usado juntamente com um componente Provider (cache e queries), que pode importar dados. É útil para arquiteturas cliente/servidor ou de múltiplas camadas.
- Acessa muitos mecanismos de bancos de dados diferentes: Interbase, Oracle, MySql (Linux), Informix, Db2, Ms SQL Server, mas enfrenta alguns problemas de diferenças de dialeto SQL.

ADO (ACTIVEX DATA OBJECTS):

- Interface de alto nível da Microsoft para acesso a banco de dados.
- Implementado na tecnologia de acesso a dados OLE DB da Microsoft.
- Acessa banco de dados relacionais, não relacionais, sistemas de arquivos, e-mail e objetos corporativos personalizados.
- Independe de servidor de banco de dados, com suporte a servidores locais e SQL.
- Mecanismo pesado e configuração simplificada. O tamanho da instalação do MDAC (Microsoft Data Access Components) atualiza grandes partes do sistema operacional.

- Compatibilidade limitada entre versões: obriga o usuário a atualizar os computadores para a mesma versão que utilizou para desenvolver o programa.
- Apresenta vantagens para quem usa Access ou SQL Server, oferecendo melhor qualidade de drivers que os dos provedores de OLE DB normais.
- Não serve para desenvolvimento independente de plataforma: não disponível no Linux ou em outros sistemas operacionais.
- O pacote ADO foi denominado pela Borland de dbGo, e seus componentes principais são: ADOConnection (para conexão ao banco de dados), ADOCommand (para execução de comandos SQL) e ADODataSet (para execução de requisições que retornam um conjunto de resultados). Além desses, existem 3 componentes de compatibilidade: ADOTable, ADOQuery e ADOSToredProc, que podem ser usados para portar aplicativos baseados no BDE. E ainda há o componente RDSConnection, que permite acessar dados em aplicativos multicamadas (multitier) remotos.
- A Microsoft está substituindo o ADO pela sua versão .NET, a qual se baseia nos mesmos conceitos básicos. Assim, o uso do ADO pode fornecer uma boa preparação para quem caminha na direção de aplicativos .NET nativos (embora a Borland também planeje portar o dbExpress para aquela plataforma).

"INTERBASE EXPRESS">INTERBASE EXPRESS (IBX):

- Não é um mecanismo de banco de dados independente de servidor, mas um conjunto de componentes para acessar um servidor de banco de dados específico.
- Esse conjunto de componentes aplica-se a diferentes ambientes (Delphi, Oracle) e plataformas (Linux).
- Oferecem melhor desempenho e controle, à custa de flexibilidade e portabilidade.
- Transações: Cada operação de edição/postagem é considerada uma transação implícita, mas deve-se alterar esse comportamento manipulando as transações implicitamente. Deve-se evitar as transações que esperam entrada do usuário para serem concluídas, pois o usuário poderia ficar fora indefinidamente e a transação permaneceria ativa por um longo tempo, consumindo performance do sistema. Deve-se isolar cada transação corretamente, por meio do nível de isolamento SnapShot para relatórios e ReadCommitted para formulários interativos. Uma transação começa automaticamente quando você edita qualquer conjunto de dados ligados a ela. O comando CommitRetaining não abre uma nova transação, mas permite que a transação atual permaneça aberta.
- Componentes Ibx: IbSQL: executa instruções SQL que não retornam um conjunto de dados (instruções DDL ou instruções update e delete); IbDataBaseInfo: consultar estrutura e status do banco de dados; IbSqlMonitor: depurador do sistema (o SQL monitor do Delphi é específico para o BDE); IbEvents: recebe eventos enviados pelo servidor.

CONFIGURANDO UMA CONEXÃO COM BD USANDO ADO NO BORLAND DELPHI 2005

Selecione o componente de conexão na palheta DBGO. Selecione também um objeto QUERY

O Componente DATASOURCE fica na palheta DATA ACCESS

Object Inspector

Property	Value
Align	alNone
⊕ Anchors	[akLeft,akTop]
BiDiMode	bdLeftToRight
BorderStyle	bsSingle
Color	<input type="checkbox"/> clWindow
Columns	(TDBGridColumns)
⊕ Constraints	(TSizeConstraints)
Ctl3D	True
Cursor	crDefault
DataSource	
DefaultDrawing	True
DragCursor	crDrag
DragKind	dkDrag

All shown

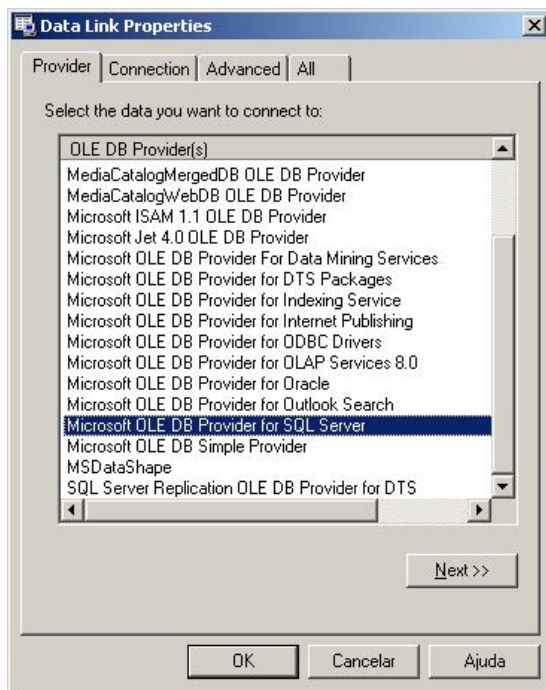
A propriedade DATASOURCE é comum a todos os componentes da palheta DATA CONTROLS.

Configurando o objeto CONNECTION para conectar no SQL SERVER 2000:

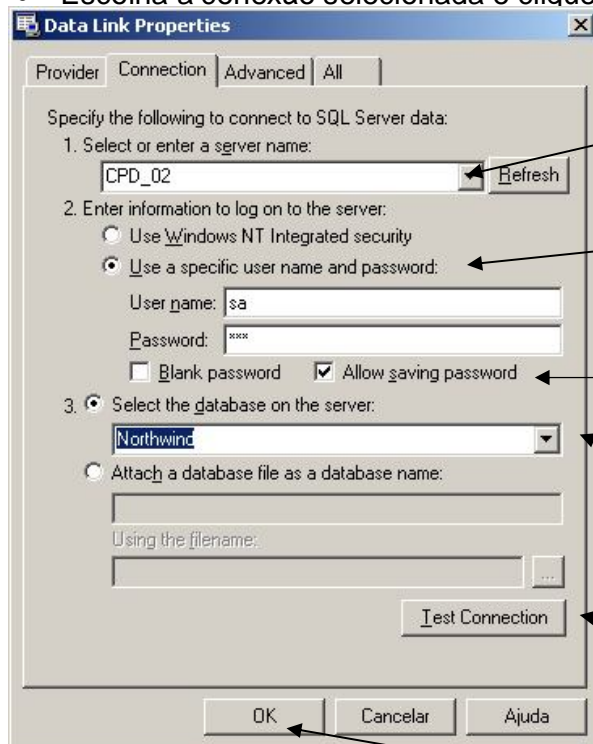
- Duplo clique no objeto



- Clique no Botão BUILD



- Escolha a conexão selecionada e clique em NEXT



Escolha o servidor (computador) que gerencia o Banco.

Selecione uma forma de autenticação. Usaremos no laboratório a segunda opção. Informe no usuário: SA e na senha: 123

Se esta opção (Allow saving password) estiver disponível, ligue-a.

Selecione um banco de dados

Teste a conexão para ver se tudo está OK

Clique em OK para finalizar

- ALTERE a propriedade **LoginPrompt** para FALSE.

Configurando o componente QUERY:

Preencha as seguintes propriedades:

- Connection (com o objeto de conexão)
- SQL (com o comando SQL)
- **Não preencha a propriedade DATASOURCE!!!!**

Configurando o componente DATASOURCE:

Preencha as seguintes propriedades:

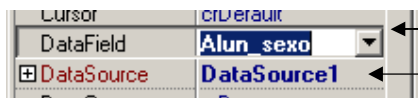
- Dataset (com o nome do objeto QUERY que será controlado)
- Autoedit (**true** para permitir edição automática e **false** para o contrário)

CRIANDO UM CADASTRO SIMPLES (APENAS 1 TABELA)

- Crie um novo formulário;
- Coloque no formulário os seguintes componentes: TAdoConnection, TAdoQuery e TDataSource;
- Faça todas as conexões como explicado acima.
- Vamos criar a seguinte tabela no banco NorthWind:

```
Create table Alunos(
  Alun_cod int primary key,
  Alun_nome varchar(50),
  Alun_nascimento datetime,
  Alun_sexo varchar(1),
  Alun_cor int );
```

- Digite o seguinte comando SQL na QUERY: select * from alunos
- Durante o desenvolvimento do cadastro, vamos deixar a Query “aberta”, para isso mude a propriedade ACTIVE para TRUE.
- Monte a tela de formas que ela fique com a seguinte aparência: (use os componentes da palheta DATA CONTROLS)



Os componentes da palheta DATA CONTROLS tem as seguintes propriedades que devem ser preenchidas.

DBNAVIGATOR

DBEDIT

DBRADIOGROUP

Objeto SEXO

Propriedade ITEMS:

*Masculino**Feminino*

Propriedade Values:

*M**F*

Objeto COR

Propriedade ITEMS:

*BRANCA**PRETA**PARDA**AMARELA**INDÍGENA*

Propriedade Values:

*1**2**3**4**5*

- Agora vamos programar 2 eventos do FORMULÁRIO:
- Evento **onFormShow** do formulário:

```
procedure TForm1.FormShow(Sender: TObject);
begin
    if not ADOConnection1.Connected then // abre a conexão com o banco de dados
        ADOConnection1.Open;
    ADOQuery1.close; // já deveria estar fechada, mas se não estiver, nós a fechamos aqui;
    ADOQuery1.open;
end;
```

- Evento **onFormClose** do formulário:

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    ADOQuery1.close; // fechamos a conexão
    ADOConnection1.close; // uma boa prática de programação: fechar as query's abertas.
end;
```

VALIDANDO OS DADOS DO CADASTRO

Existem várias formas de se realizar as validações em um cadastro. Pode ser no botão de OK, no evento beforpost da Query, etc.. Vamos ver como ficaria no evento onBeforePost do objeto adoQuery:

```
procedure TForm1.ADOQuery1BeforePost(DataSet: TDataSet);
begin
    // para cancelar a gravação neste evento, precisamos gerar uma exceção com raise ou abort.

    try
        StrToInt( edCodigo.text )
    except
        // se usar o raise não dá para usar o setfocus.. mas ...
        raise exception.create('Código Inválido!');
    end;

    if length(edNome.text) = 0 then begin
        showmessage('Preencha o nome corretamente. ');
        abort; // ... você pode usar o conjunto mensagem + abort!
    end;

    // você pode validar também diretamente pelos campos da tabela, ao invés dos
    // componentes de tela. ex:

    try
        StrToDate( ADOQuery1.fieldbyname('alun_nascimento').asString );
    except
        raise exception.create('Data Inválida!');
    end;

    if ADOQuery1.fieldbyname('alun_sexo').IsNull then // isnull significa que o campo nao foi preenchido!
        raise exception.create('Informe o sexo');

    // outra forma de ver se foi preenchido seria:

    if ADOQuery1.fieldbyname('alun_cor').asString = '' then // e existem ainda outras formas.
        raise exception.create('Informe a cor');
    end;
```

CRIANDO UM CADASTRO COM MAIS DE UMA TABELA (FK)

No cadastro anterior, as cores foram colocadas fixas no código. Se houver a necessidade de adicionarmos mais cores teremos que mudar no código. Seria interessante que as cores fossem selecionadas de uma tabela, e é isso que vamos fazer agora.

Primeiro vamos criar a tabela de cores. Não esqueça de criá-la no banco NORTHWIND!!!

```
create table cor
(cor_cod int primary key,
 cor_desc varchar(20) );
```

Cadastrando as cores:

```
insert into cor values (1, 'BRANCA');
insert into cor values (2, 'PRETA');
insert into cor values (3, 'PARDA');
insert into cor values (4, 'AMARELA');
insert into cor values (5, 'INDÍGENA');
select * from cor;
```

Coloquem na tela os seguintes componentes:

ADOQUERY (mude o nome para quCor) e DATASOURCE (mude o nome para dsCor)

Faça as devidas conexões como já descrito anteriormente e no objeto quCor digite : *select * from cor*

Insira no lugar do componente TRadioGroup que controlava as cores o seguinte componente que está na palheta DataControls: DBLookupComboBox (este componente é capaz de exibir todos os registros de uma tabela) e preencha as seguintes propriedades dele:

DataSource : o mesmo que controla os outros componentes (DataSource1)

DataField: o campo em que será gravado o dado escolhido pelo usuário (alun_cor)

ListSource : DsCor (o datasource da tabela cujos dados serão listados)

ListField : cor_desc (o campo que sera listado)

KeyField: cor_cod (o campo chave desta tabela e que está relacionado com o campo datafield)

A tabela quCor deve ser aberta no evento onShow do Formulário e fechada no evento onClose também do formulário.

SUGERINDO UM CÓDIGO AUTOMATICAMENTE

Em cadastros onde existe um campo código seqüencial (mas não é autoincrement), seria interessante que o sistema sugerisse um valor, como por exemplo o próximo valor. Isto pode ser resolvido facilmente selecionando-se o maior valor para o campo código na tabela e adicionando-se 1.

Um bom local para inserir este código seria no evento Afterinsert da tabela onde está sendo realizado o cadastro. No nosso cadastro de alunos, teríamos que fazer o seguinte:

- Insira um objeto ADOQUERY (quMAX) na tela, faça as conexões e na propriedade SQL digite:

“Select isNull(max(alun_cod) +1, 1) "valor" from alunos“ (o commando isnull é específico para o sql-server)

O comando acima retornará o próximo valor para alun_cod, ou 1 caso a tabela esteja vazia.

- Digite o seguinte no evento afterinsert da tabela que controla o cadastro de alunos:

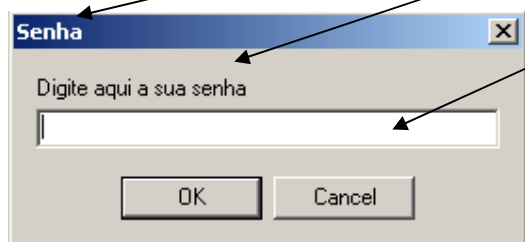
```
procedure TForm1.ADOQuery1AfterInsert(DataSet: TDataSet);
begin
    quMAX.close;
    quMAX.open;
    Dataset['alun_cod'] := quMAX['valor'];
end;
```

Observe que DATASET faz referência ao objeto ADOQUERY1 que é a query que representa o nossa tabela de alunos neste cadastro. Veja que ela é passada via parâmetro para o evento AfterInsert. Poderíamos substituir a terceira linha por: ADOQUERY1 ['alun_cod'] := quMAX['valor'];

Você poderia também não permitir que o usuário modifique o campo alun_cod, para isso basta modificar a propriedade readonly para true, mudar a cor do campo para cinza e a propriedade tabstop para false.

SOLICITANDO UM DADO AO USUÁRIO (INPUTBOX)

Em algumas situações seu programa deve solicitar um dado ao usuário, como por exemplo, uma informação que não faz parte de seu cadastro. Para estas situações podemos usar o InputBox, cuja sintaxe é: `function InputBox(const ACaption, APrompt, ADefault: string): string;`



Onde: ACaption é a informação que aparece no caption
APrompt é a informação que aparece no label
ADefault é o valor padrão.

O valor digitado é retornado na função. Se o usuário clicar em CANCEL ele retorna o valor padrão informado em ADefault.

Ex. que gera uma tela como a exibida acima:

```
procedure TForm1.Button1Click(Sender: TObject);  
var x : string;  
begin  
  x := InputBox('Senha', 'Digite aqui a sua senha', '');  
end;
```

CADASTROS SEM O TDBNAVIGATOR

O objeto TDBNavigator é um bom quebra-galho para executar as principais operações sobre um objeto Tdataset (query, table, etc...). Porém em algumas situações seria interessante que nós usássemos nossa própria barra de controle. Para isso precisamos conhecer os principais métodos e propriedades de um objeto TDataset:

Prior	registro anterior
Next	próximo registro
First	primeiro registro
Last	ultimo registro
Cancel	cancela a edição/inserção
Post	grava o registro
Edit	edita o registro
Insert	Insere um novo registro na posição atual
Append	Insere um novo registro no final da tabela
Iseempty	Retorna verdadeiro se a tabela está vazia e false caso contrário
Open	Abre a tabela
Close	Fecha a tabela
Locate	Realiza uma pesquisa na tabela. Veremos em detalhes mais adiante.
RecordCount	Retorna a quantidade de registros.
BOF	Retorna verdadeiro se a tabela estiver apontando para o primeiro reg.
EOF	Retorna verdadeiro se a tabela estiver apontando para o último reg.

Sendo assim, podemos usar botões para representar estes métodos em nosso cadastro.

Uma propriedade interessante da classe TDATASET é a State, que retorna o estado atual de uma tabela. Os principais estados são:

dsInsert	indica que a tabela está em nodo de inserção
dsedit	indica que a tabela está em nodo de edição
dsbrowse	indica que a tabela está em nodo de navegação

UMA TELA CRIADA SEM O TDBNAVIGATOR

Programação dos botões :

```
procedure TForm1.btnNovoClick(Sender: TObject);
begin
  ADOQuery1.append;
end;
```

```
procedure TForm1.btnAlterarClick(Sender: TObject);
begin
  ADOQuery1.edit;
end;
```

```
procedure TForm1.btnApagaClick(Sender: TObject);
begin
  if not ADOQuery1.isempty then
    if application.messagebox('Deseja realmente apagar?', 'Atenção', mb_yesno + mb_iconquestion +
      mb_defbutton2) = id_yes then
      ADOQuery1.delete;
end;
```

```
procedure TForm1.btnOKClick(Sender: TObject);
begin
  ADOQuery1.post;
end;
```

```
procedure TForm1.btnCancelClick(Sender: TObject);
```

```
begin
  ADOQuery1.cancel;
end;
```

```
procedure TForm1.btnPrimeiroClick(Sender: TObject);
begin
  ADOQuery1.first;
end;
```

```
procedure TForm1.btnAntClick(Sender: TObject);
begin
  ADOQuery1.prior;
end;
```

```
procedure TForm1.btnProxClick(Sender: TObject);
begin
  ADOQuery1.next;
end;
```

```
procedure TForm1.btnUltimoClick(Sender: TObject);
begin
  ADOQuery1.last;
end;
```

Para habilitar/desabilitar os botões de controle, podemos utilizar o evento StateChange do componente TDataSource que controla o cadastro. Este evento é disparado sempre que o estado da tabela muda. Exemplo: ao iniciar uma edição, cancelar a gravação, começar a inclusão, etc...

```
procedure TForm1.DataSource1StateChange(Sender: TObject);
begin
  btnAnt.enabled := ADOQuery1.State = dsBrowse;
  btnProx.enabled := ADOQuery1.State = dsBrowse;
  btnPrimeiro.enabled := ADOQuery1.State = dsBrowse;
  btnUltimo.enabled := ADOQuery1.State = dsBrowse;
  btnNovo.Enabled := ADOQuery1.State = dsBrowse;
  btnAltera.Enabled := ADOQuery1.State = dsBrowse;
  btnApaga.enabled := ADOQuery1.State = dsBrowse;

  btnOK.enabled := ADOQuery1.State in [dsInsert, dsEdit];
  btnCancel.enabled := ADOQuery1.State in [dsInsert, dsEdit];
end;
```


CONSTRUINDO UMA TELA DE CONSULTA

Form1

Novo Alterar Apagar OK Cancel << < > >> Consultar

Código: 1 Nome: maria Data de nascimento: 01/01/2001

Sexo: ☐ Masculino ☒ Feminino

Cor: AMARELA

Alun_cod	Alun_nome	Alun_nascimento
1	maria	01/01/2001
2	Pedro	01/01/2005
3	maria da silva	01/08/1986

Na tela de cadastro devemos colocar um botão de consulta e no evento onclick a chamada à tela de consulta:

```
fmconsulta.showmodal;
```

Consulta

Cor: *** TODAS ***

Consulta pelo nome: maria da silva

Alun_cod	Alun_nome	Alun_nascimento	Alunsexo	Alun_cor
1	maria	01/01/2001	F	
2	Pedro	01/01/2005	M	
3	maria da silva	01/08/1986	F	

ComboBox

Edit

DBGrid

Para construirmos uma tela de consulta, precisamos aprender como funcionam os parâmetros em uma Query:

PASSANDO PARÂMETROS PARA O OBJETO ADOQUERY

Um Objeto TADOQUERY permite que sejam passados parâmetros (valores) antes que ela seja aberta, restringindo assim os registros que serão devolvidos pelo SGBD.

No exemplo acima, o usuário pode filtrar os alunos pela cor, sendo assim, tem que ser possível a passagem de parâmetros para a query de formas que ela só retorne os registros que satisfaçam a cor escolhida.

Para criar um parâmetro, devemos digitar no comando SQL do Objeto ADOQuery o caractere ":" seguido do nome do parâmetro. Para o exemplo acima, o SQL ficaria:

```
select * from alunos where Alun_cor between :p_cor_INI AND :p_cor_FIM
```

Podemos preencher os parâmetros em tempo de desenvolvimento através da propriedade Parameters, ou via código, como segue abaixo:

```
ADOQuery1.close;  
ADOQuery1.Parameters.ParamByName('p_cor_ini').Value := 1;  
ADOQuery1.Parameters.ParamByName('p_cor_fim').Value := 1;  
ADOQuery1.open;
```

A Query acima retornará todos os registros que contém a cor igual à 1. É necessário fechar e abrir a query novamente sempre que mudamos os parâmetros.

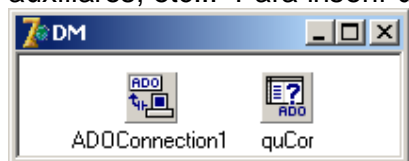
DATAMODULE

Observe que usamos uma nova query na tela de consulta, e isto implica em conecta - lá a um objeto TADOConnection, que no nosso exemplo está localizado no formulário de cadastro de Alunos (FORM1). Sendo assim, teremos que fazer uma referência no formulário de consulta ao formulário de Alunos, o criaria uma dependência.

Em sistemas onde temos mais de uma tela que acesse dados, é uma ótima prática utilizar o DATAMODULE, que nada mais é que um "repositório", uma espécie de "FORM" onde nós podemos centralizar, além de outros objetos, o objeto de Conexão com o Banco de dados, visto que só haverá 1 para toda a aplicação. Desta forma, nossos formulários dependerão apenas do Datamodule.

Quando utilizar um datamodule, vá em PROJECT-OPTIONS e coloque o datamodule como o primeiro formulário a ser criado! Você também deve colocar o nome da unit dele no USES dos outros formulários.

Podemos colocar no Datamodule também Query's que são utilizadas em todo o sistema, Query's auxiliares, etc... Para inserir um, vá em: file-> new-> Other -> Delphi Files -> Datamodule.



CONTINUANDO COM A CONSTRUÇÃO DA TELA DE CONSULTA

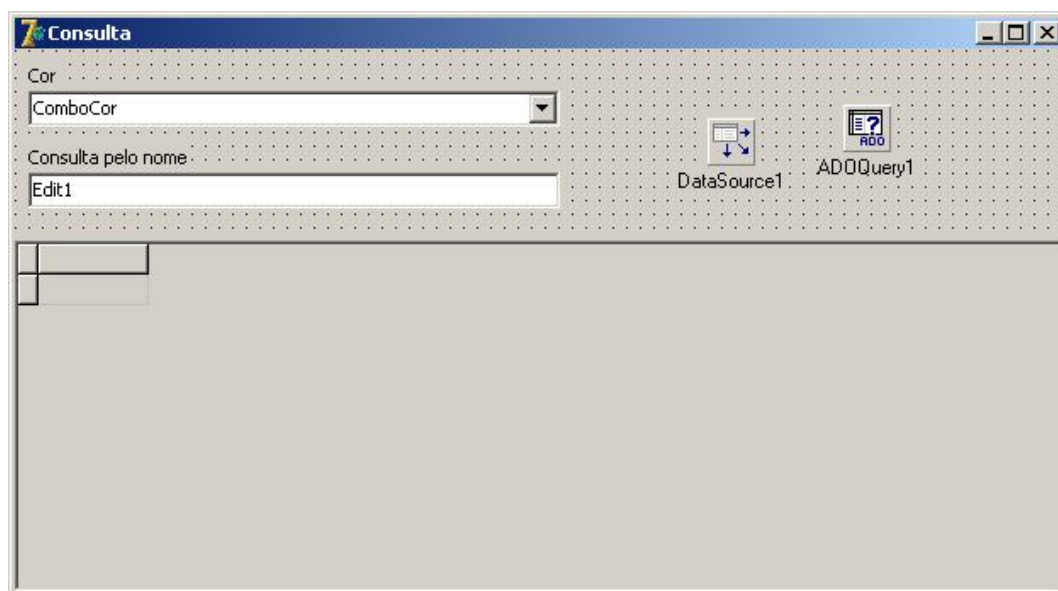
O MÉTODO LOCATE DA CLASSE TDATASET

Antes de prosseguirmos com a tela de consulta, precisamos entender o método **LOCATE**. Ele permite uma pesquisa em uma tabela, posicionando o cursor no registro pesquisado e retornando TRUE se encontrou ou FALSE se não encontrou.

Ex: Vamos pesquisar o aluno de número 5:

```
If adoquery1.Locate('alun_cod', 5, [])= true then
  Showmessage('Encontrado')
Else
  Showmessage('NÃO Encontrado !!!')
```

MAS ATENÇÃO: Ao pesquisar um registro a tabela muda seu estado para dsBrowse, portanto, se a tabela estava antes da pesquisa em modo de edição ou inserção, ela tentará executar o POST para passar para o estado dsBrowse e então executar o método locate.



Mude as seguintes propriedades da grade:
 Color= clmenu
 Readonly=True
 Options.dgRowSelect=true
 Options.dbAlwaysShowSelection = true

Conecte todos os objetos, inclusive a propriedade connection do Adoquery ao objeto de conexão que está no DATAMODULE e digite na propriedade SQL da Query:

```
select * from alunos
where Alun_cor between :p_cor_INI AND :p_cor_FIM
```

Eventos que devem ser preenchidos:

```
procedure TfmConsulta.FormShow(Sender: TObject);
begin
  // preenchendo as cores do combobox
  adoquery1.close;
  combocor.Clear;
```

```

dm.quCor.Close;
dm.quCor.open;
while not dm.quCor.eof do // enquanto nao for o fim do arquivo...
begin
    ComboCor.Items.Add(dm.quCor.fieldbyname('cor_cod').asString + ' - ' +
        dm.quCor.fieldbyname('cor_desc').asString );
    dm.quCor.next;
end;
ComboCor.Items.Add('*** TODAS ***');
dm.quCor.close;

Edit1.Clear;
ComboCor.SetFocus;
end;

```

A programação abaixo faz com que a query seja aberta de acordo com a cor que o usuário escolher:

```

procedure TfmConsulta.ComboCorSelect(Sender: TObject);
var p_ini, p_fim : integer;
begin
    // descobrindo o valor da cor para usar como parâmetro na query
    if comboCor.text = '*** TODAS ***' then
    begin
        p_ini := 0;
        p_fim := 9999999;
    end
    else
    begin
        // copiar até 1 caractere a menos que a posição do traço (-), tirar os espaços e converter. p/ inteiro
        p_ini := StrToInt(Trim(copy(comboCor.text, 1, pos('-', comboCor.text) - 1 )) );
        p_fim := StrToInt(Trim(copy(comboCor.text, 1, pos('-', comboCor.text) - 1 )) );
    end;

    // preenchendo os parâmetros da query
    ADOQuery1.close;
    ADOQuery1.Parameters.ParamByName('p_cor_ini').Value := p_ini;
    ADOQuery1.Parameters.ParamByName('p_cor_fim').Value := p_fim;
    ADOQuery1.open;
end;

```

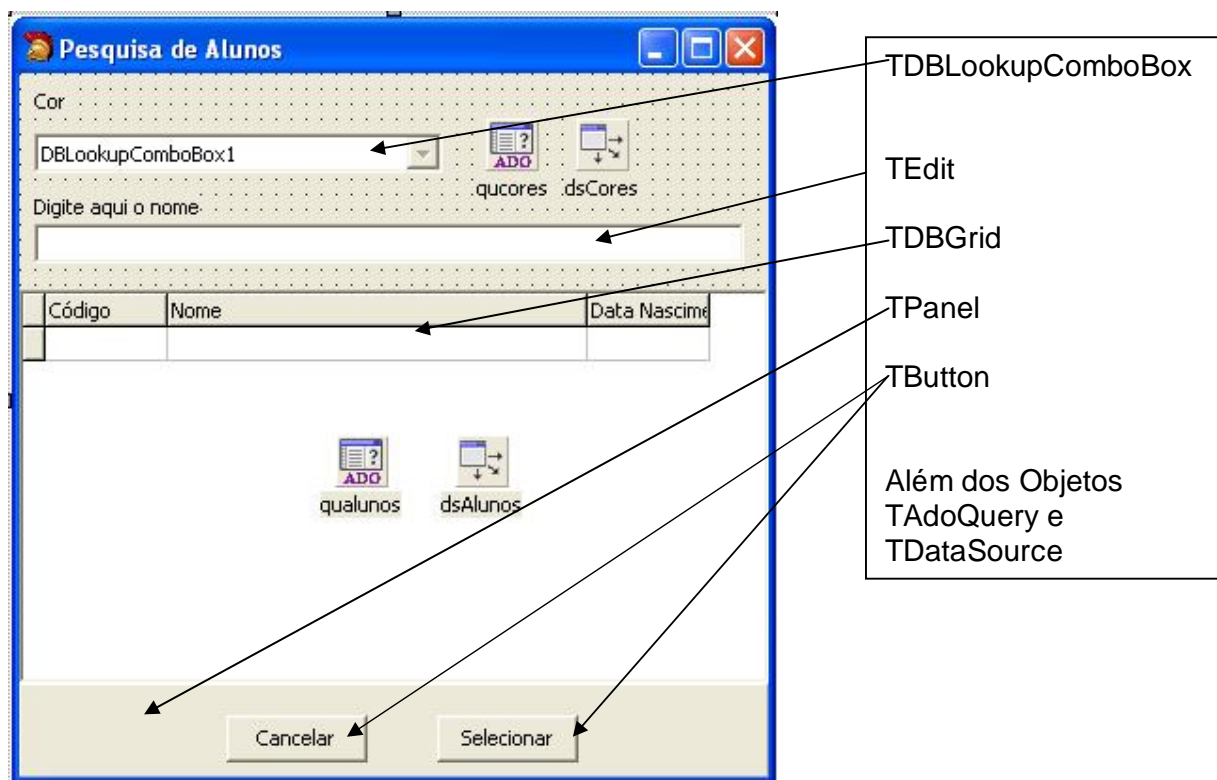
O código abaixo permite que o usuário realize uma pesquisa incremental na tabela. As opções loPartialKey, loCaseInsensitive são utilizadas apenas em campos *TEXTO*.

```

procedure TfmConsulta.Edit1Change(Sender: TObject);
begin
    if (ADOQuery1.Active) and (trim(Edit1.text) <> '') then begin
        adoquery1.Locate('alun_nome', trim(Edit1.text), [loPartialKey, loCaseInsensitive ]);
    end;
end;

```

CRIAÇÃO DA TELA DE CONSULTA – SEGUNDA ALTERNATIVA



Programação:

No PUBLIC, criar uma variável para retornar o código do aluno selecionado:

```
public
  retorno : integer;
```

Abrir a tabela qucores no formshow e definir o valor default para o retorno:

```
procedure TfmPesquisa.FormShow(Sender: TObject);
begin
  retorno := -1; // valor padrão para o caso de nada ser selecionado na grade...
  qucores.open;
end;
```

No evento onclick do componente TDBLookupcomboBox (que é disparado sempre que se seleciona um item nele), digite o comando que permitirá abrir a query de pesquisa de alunos apenas com os registros que sejam da cor escolhida:

```
procedure TfmPesquisa.DBLookupComboBox1Click(Sender: TObject);
begin
  qualunos.close;

  qualunos.Parameters.ParamByName('p_alun_cor').Value := DBLookupComboBox1.KeyValue;

  qualunos.Open;
end;
```

Permite realizar a pesquisa por nome:

```
procedure TfmPesquisa.Edit1Change(Sender: TObject);
begin
  if qualunos.active = true then begin
    qualunos.locate('alun_nome', Trim(Edit1.text), [loPartialKey, loCaseInsensitive]);
  end;
end;
```

Quando o usuário der um duplo-clique na DbGrid, o sistema retorna o código do aluno selecionado:

```
procedure TfmPesquisa.DBGrid1DbClick(Sender: TObject);
begin
  if qualunos.IsEmpty = false then begin
    retorno := qualunos.fieldbyname('alun_cod').asInteger;
    close;
  end;
end;
```

O mesmo ocorre quando o usuário clica no botão “SELECIONAR”, mas para evitar redundância no código, nós simplesmente executamos o mesmo evento DbClick da dgBrid:

```
procedure TfmPesquisa.btnSelecionarClick(Sender: TObject);
begin
  // executamos o mesmo evento ja programado na grid para selecionar
  DBGrid1.DbClick(nil);
end;
```

Se o usuário não selecionar nada, o retorno será -1 (definido no onshow) e se ele clicar no botão CANCELAR, simplesmente fechamos o form:

```
procedure TfmPesquisa.btnCancelarClick(Sender: TObject);
begin
  close;
end;
```

Alun_cod	Alun_nome	Alun_nascimento	Alun_
1	Madalena da Silva Sauro	01/01/2001	F
2	pedro		M
3	sdsdfsdfsdfdaf	01/01/2001	F
4	marcio	01/01/2001	M

Na tela que efetuará a chamada a pesquisa, pode-se colocar um botão e em sua programação, verificamos se o usuário escolheu algum aluno (retorno <> -1) e localizamos o aluno:

```
fmpesquisa.showmodal;
if fmpesquisa.retorno = -1 then exit;
```

```
ADOQuery1.Locate('alun_cod',
fmpesquisa.retorno, []);
```

VERIFICANDO SE UM REGISTRO EXISTE EM UMA TABELA

Quando vamos incluir um novo registro, precisamos averiguar se não violaremos a chave primária, tentando, por exemplo, gravar um registro com código repetido. O método locate não serve para verificar se um registro já existe em uma tabela quando utilizamos para a pesquisa a mesma tabela onde estamos cadastrando o nosso registro, pois, como já visto, o registro muda seu estado para dsBrowse. Faremos esta verificação de outra forma.

Para isso podemos fazer uma pesquisa ANTES de gravar o registro, procurando pelo código que o usuário está tentando cadastrar. A chave primária completa deve ser utilizada na busca. Se a pesquisa retornar algo é porque este código já existe na base de dados e não pode ser aceito. Caso contrário a gravação poderá prosseguir normalmente.

Um bom local para este tipo de programação é no evento onBeforePost do DataSet que se vai gravar o registro. Utilizaremos uma query para realizar esta pesquisa, pode ser uma query auxiliar pois preencheremos sua instrução SQL em tempo de execução.

Ex:

```
procedure TForm1.ADOQuery1BeforePost(DataSet: TDataSet);
begin
    // para cancelar a gravação neste evento, precisamos gerar uma exceção com raise ou abort.
    try
        StrToInt( edCodigo.text );
    except
        raise exception.create('Código Inválido!');
    end;

    // é importante que o campo edCodigo esteja preenchido caso contrário a query abaixo não funcionará!
    // Validando se o código informado já existe na tabela alunos. A query precisa retornar apenas 1 campo....
    quAux.close;
    quAux.Sql.text := 'select alun_cod from alunos where alun_cod = ' + edCodigo.text;
    quAux.open;

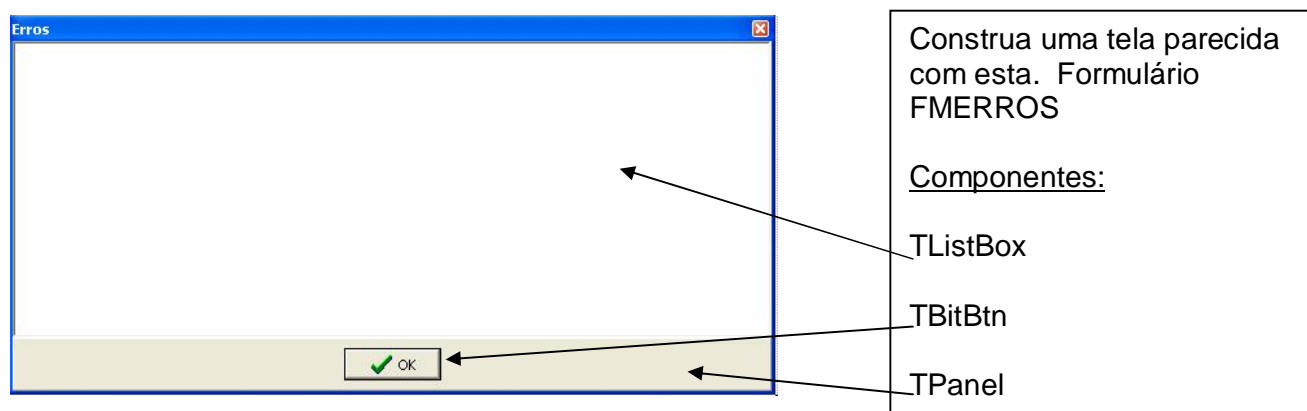
    // se a query não estiver vazia é porque já existe um aluno com este código...
    if quAux.isempty = false then
    begin
        quAux.close;
        raise exception.create('Já existe um aluno cadastrado com este código. Tente outro!');
    end;
    quAux.close;

    ..... continuação das outras validações.....
end;
```

EXIBINDO OS ERROS EM UMA TELA SEPARADA

Em algumas situações torna-se interessante exibir de uma única vez os erros de um cadastro ao usuário.

Para fazer isso vamos criar um formulário para exibir os erros em um TListBox (poderia ser um TMemo p.ex.) e também criaremos 3 métodos neste formulário: Um para Inserir erros, um para verificar se tem erros e outro para limpar os erros. Assim o programador não precisará se preocupar em como usar a tela, ele apenas precisa saber que existem 3 métodos e que a tal tela precisa ou não ser criada antes de ser utilizada (que também poderia ser feito automaticamente). Para não complicar muito, e, como esta normalmente é uma tela muito utilizada, vamos deixá-la no auto-create.



Parte da programação da tela FMRERROS:

```
public
{ Public declarations }
Procedure LimpaErros;
Procedure AdicionaErro(p_erro : string);
Function VerificaSeTemErro : Boolean;
end;

var
  fmErros: TfmErros;

implementation

{$R *.dfm}

procedure TfmErros.AdicionaErro(p_erro: string);
begin
  ListBox1.Items.Add(p_erro);
end;

procedure TfmErros.LimpaErros;
begin
  ListBox1.items.Clear;
end;

function TfmErros.VerificaSeTemErro: Boolean;
begin
  result := ListBox1.Items.Count > 0;
end;
```


Para utilizar esta tela basta executar primeiro o método limpaErros, ir adicionando erros com o método AdicionaErro e ao final da validação, verificar se tem erros com o método VerificaSeTemErro e, caso ele retorne True exibir o a tela de erros com o método showmodal.

Observe que agora os erros não devem mais ser exibidos a medida que são detectados! Ex: Vamos fazer uma alteração na validação do cadastro de alunos:

```
procedure TForm1.ADOQuery1BeforePost(DataSet: TDataSet);
begin
    // para cancelar a gravação neste evento, precisamos gerar uma exceção com raise ou abort.

    // Primeiro, vamos limpar os erros da tela de erros caso ela já tenha sido usada....
    fmErros.LimpaErros;

    try
        StrToInt( edCodigo.text );
    except
        fmErros.AdicionaErro('Código Inválido!'); // observe que não paramos mais o programa a cada erro....
    end;

    if length(edNome.text) = 0 then begin
        fmErros.AdicionaErro ('Preencha o nome corretamente.');
```

```
    end;

    try
        StrToDate( ADOQuery1.fieldbyname('alun_nascimento').asString );
    except
        fmErros.AdicionaErro ('Data Inválida!');
```

```
    end;

    if ADOQuery1.fieldbyname('alun_sexo').IsNull then
        fmErros.AdicionaErro('Informe o sexo');
```

```
    if ADOQuery1.fieldbyname('alun_cor').asString = '' then
        fmErros.AdicionaErro ('Informe a cor');
```

```
    // ao final da validação verificamos se existe erro, e se existir, paramos o programa com uma exceção e
    // exibimos os erros para que o usuário possa corrigí-los.
    If fmErros.VerificaSeTemErro = True then begin
        fmErros.showModal; // exibimos a tela de erros...
        abort; // agora sim, cancelamos a gravação gerando uma exceção!
    end;
end;
```

TRABALHANDO COM VETORES DINÂMICOS

Em algumas situações precisamos trabalhar com os dados de uma tabela em um vetor. Por exemplo, se quisermos ordenar, escolher um elemento aleatoriamente, fazer cálculos, etc.

Para isso precisamos criar um vetor que tenha a mesma capacidade da tabela. Mas como fazer isso em tempo de execução se precisamos definir o tamanho do vetor em tempo de programação? Para situações como essa podemos usar a procedure **SetLength** que modifica o tamanho de um vetor. Porém, ao criarmos este vetor não podemos especificar o tamanho inicial. Podemos usar também a função **LENGTH** para retornar a quantidade de elementos de um vetor. Observe que a posição inicial de um vetor dimensionado pelo **SetLength** é 0 (zero) . EX:

Delphi syntax:

procedure SetLength(var S; NewLength: Integer); // onde S é o vetor e NewLength é o tamanho.

Var

vetor : array of integer;

Begin

SetLength(vetor, 10); // cria um vetor de inteiros de tamanho 10.

SetLength(vetor, 1200); // muda o tamanho para 1200 sem perder o que já havia nas 10 primeiras posições.

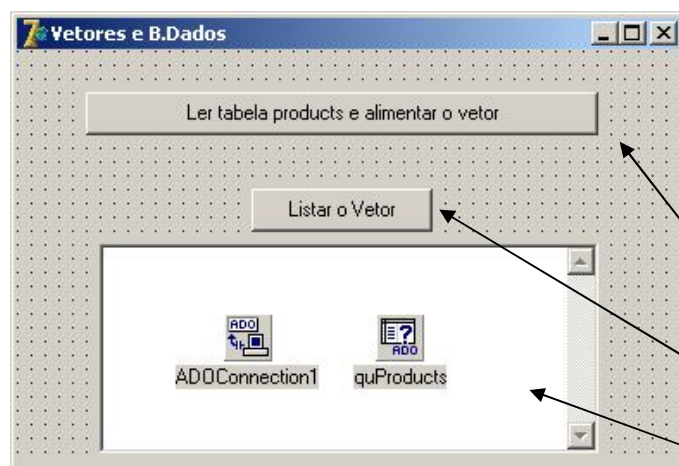
SetLength(vetor, 200); // muda o tamanho para 200 ... e é claro, perde o que havia nas outras posições.

showmessage(' O vetor tem : ' + intToStr(length(vetor)) + ' posições. '); // exibe o tamanho do vetor.

End;

Um exemplo completo:

Este exemplo lê a tabela products do banco northwind e guarda os campos productID, productName e unitprice em um vetor dinâmico do tipo TDados, que é um registro definido no código.



Faça a conexão com o banco de dados NorthWind e digite na query QuProducts:

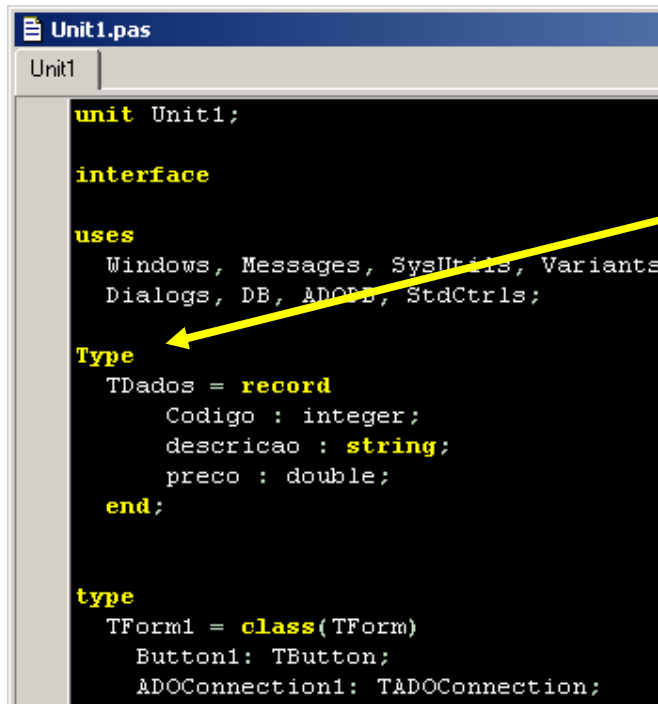
*Select * from products*

Mude a propriedade enabled do botão [listar o vetor] para false.

Button1

Button2

Memo1



Para poder armazenar mais de uma informação por elemento do vetor, vamos criar um registro.

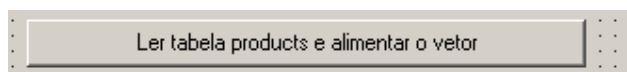
A vetor foi definido na seção PRIVATE do código, criamos assim uma variável global, que só tem visibilidade nesta unit.

```

private
  vetor : array of Tdados;

```

Programando os botões:



```

procedure TForm1.Button1Click(Sender: TObject);
var i : integer;
begin
  Button2.enabled := true;
  if not ADOConnection1.Connected then
    ADOConnection1.Open;

  quProducts.close;
  quProducts.open;
  quProducts.last; // é bom fazer isso para que Delphi carregue todos os registros na memória
  quProducts.first;

  SetLength(vetor, quProducts.RecordCount ); // redimensiona o vetor

  i := 0;
  while not quProducts.eof do
  begin
    vetor[i].Codigo := quProducts['productID'];
    vetor[i].descricao := quProducts['productname'];
    vetor[i].preco := quProducts['unitprice'];
  end;
end;

```

```

    quProducts.next;
    inc(i); // incrementa a variável I
end;
quProducts.close;

showmessage('Concluído! O vetor tem : ' + intToStr(length(vetor)) + ' posições. ');
end;

```

Listar o Vetor

```

procedure TForm1.Button2Click(Sender: TObject);
var i : integer;
begin
    Memo1.Lines.Clear;
    Memo1.Lines.add('Primeira posição começa em : ' + intToStr(low(vetor)) );

    for i:=low(vetor) to High(vetor) do // low = primeira posição do vetor e high = última
    begin
        memo1.Lines.Add('Código: ' + intToStr(vetor[i].codigo));
        memo1.Lines.Add('Descrição: ' + vetor[i].descricao);
        memo1.Lines.Add('Preço: ' + FormatFloat('#0.00', vetor[i].preco) ); // *FormatFloat formata um Float!!!
        memo1.Lines.Add('*****');
    end;

    Memo1.Lines.add('Última posição : ' + intToStr( high(vetor)) );
end;

```

*O comando **FormatFloat** formata um valor do tipo Real, Double, etc.. em um formato String.

Delphi syntax:

function FormatFloat(const Format: string; Value: Extended): string;

Exemplos:

The following table shows some sample formats and the results produced when the formats are applied to different values:

Format string-	1234	-1234	0.5	0
	1234	-1234	0.5	0
0	1234	-1234	1	0
0.00	1234.00	-1234.00	0.50	0.00
###	1234	-1234	.5	
###0.00	1,234.00	-1,234.00	0.50	0.00
###0.00;(###0.00)	1,234.00	(1,234.00)	0.50	0.00
###0.00;;Zero	1,234.00	-1,234.00	0.50	Zero
0.000E+00	1.234E+03	-1.234E+03	5.000E-01	0.000E+00
#####E-0	1.234E3	-1.234E3	5E-1	0E0

TRANSAÇÕES

De uma forma direta, uma transação é qualquer operação de escrita em uma tabela em um banco de dados. Executar uma instrução UPDATE é uma transação por si só, assim como da mesma forma o é executar um DELETE ou um INSERT. Esse trio de comandos efetivamente altera o conteúdo dos dados dentro de uma tabela. Entretanto, comandos de alteração de estruturas de dados, como o ALTER TABLE, por exemplo, não são considerados transacionais.

Controlar transações significa ser capaz de definir regras de negócio para o funcionamento dessa escrita nas nossas tabelas de dados. Na verdade, o que realmente queremos fazer é controlar múltiplas transações, torná-las consistentes e dependentes entre si. Um exemplo clássico: transferência de dinheiro entre contas. Uma conta corrente e uma conta de poupança. Se pretendemos, por exemplo, transferir 1.000 reais de uma conta para outra, isso irá requerer diversas transações no banco de dados.

Vejamos:

O saldo da conta corrente é subtraído em 1.000 (UPDATE)
 É criado um novo registro na sua tabela de extrato bancário (INSERT)
 É somado um valor de 1.000 ao saldo da sua conta de poupança (UPDATE)
 É adicionado um novo registro no extrato de sua conta de poupança (INSERT)

Como podemos observar, a operação completa envolve 4 transações, cada uma em uma tabela diferente. A pergunta é: o que aconteceria com o dinheiro se o sistema conseguisse completar apenas as transações 1 e 2 e deixasse de executar a 3 e 4 por falta de energia, por exemplo?

Acho que pouca gente toleraria perder 1.000 reais por falha de um banco de dados, a situação hoje em dia está muito difícil para se abrir mão de qualquer dinheiro. Portanto, num caso de falha, o ideal é que TODO o processo seja cancelado. É o que chamamos de “tudo ou nada”.

INÍCIO E FIM DE UMA TRANSAÇÃO

Toda transação tem um ponto de partida, normalmente uma chamada ao método StartTransaction. O término da transação, por sua vez, é definido por uma chamada ao método Commit (confirma) ou Rollback (cancela). Portanto, em termos de código, seria algo assim:

```
var t : TTransactionDesc;
begin
  // dbExpress
  try
    t.IsolationLevel := xilREADCOMMITTED;
    SQLConnection.StartTransaction( t );
    // Aqui seriam feitos os updates, deletes e inserts
    SQLConnection.Commit( t );
  except
    SQLConnection.Rollback( t );
  end;

  // ADO
  try
    ADOConnection.BeginTrans;
    // Aqui seriam feitos os updates, deletes e inserts
    ADOConnection.CommitTrans;
  except
    ADOConnection.RollbackTrans;
  end;

  // BDE
  try
    Database.StartTransaction;
    // Aqui seriam feitos os updates, deletes e inserts
    Database.Commit;
  except
```

```

Database.Rollback;
end;

// IBX
try
  IBTransaction.StartTransaction;
  // Aqui seriam feitos os updates, deletes e inserts
  IBTransaction.Commit;
except
  IBTransaction.Rollback;
end;

end;

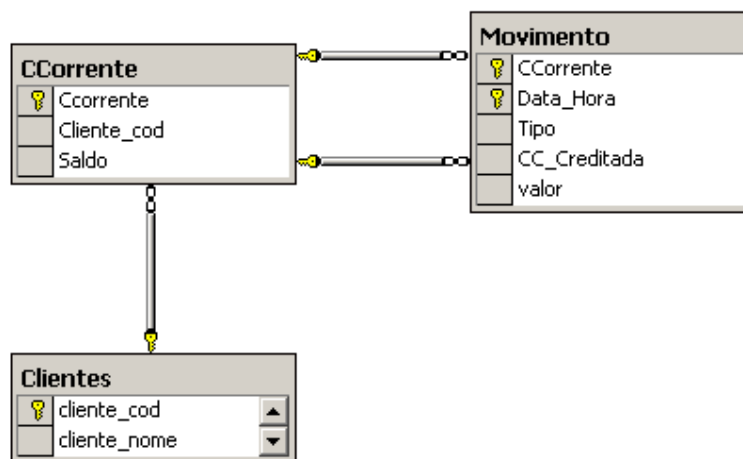
```

A materia completa pode ser acessada em :

<http://www.activedelphi.com.br/modules.php?op=modload&name=News&file=article&sid=46>

Observação: por: Eugênio Reis

Vamos utilizar como exemplo as seguintes tabelas que devem ser criadas no banco de dados NorthWind:



Script para criação das tabelas no banco NorthWind:

```

if exists (select * from dbo.sysobjects where id = object_id(N'[dbo].[FK_Movimento_CCorrente]') and OBJECTPROPERTY(id, N'IsForeignKey') = 1)
ALTER TABLE [dbo].[Movimento] DROP CONSTRAINT FK_Movimento_CCorrente
GO

```

```

if exists (select * from dbo.sysobjects where id = object_id(N'[dbo].[FK_Movimento_CCorrente1]') and OBJECTPROPERTY(id, N'IsForeignKey') = 1)
ALTER TABLE [dbo].[Movimento] DROP CONSTRAINT FK_Movimento_CCorrente1
GO

```

```

if exists (select * from dbo.sysobjects where id = object_id(N'[dbo].[FK_CCorrente_Clientes]') and OBJECTPROPERTY(id, N'IsForeignKey') = 1)
ALTER TABLE [dbo].[CCorrente] DROP CONSTRAINT FK_CCorrente_Clientes
GO

```

```

if exists (select * from dbo.sysobjects where id = object_id(N'[dbo].[CCorrente]') and OBJECTPROPERTY(id, N'IsUserTable') = 1)
drop table [dbo].[CCorrente]
GO

```

```

if exists (select * from dbo.sysobjects where id = object_id(N'[dbo].[Clientes]') and OBJECTPROPERTY(id, N'IsUserTable') = 1)
drop table [dbo].[Clientes]
GO

```

```

if exists (select * from dbo.sysobjects where id = object_id(N'[dbo].[Movimento]') and OBJECTPROPERTY(id, N'IsUserTable') = 1)
drop table [dbo].[Movimento]
GO

```

```

CREATE TABLE [dbo].[CCorrente] (
    [Ccorrente] [nvarchar] (10) COLLATE
    SQL_Latin1_General_CP1_CI_AS NOT NULL ,
    [Cliente_cod] [int] NOT NULL ,
    [Saldo] [money] NULL
)

```

```
) ON [PRIMARY]
GO
```

```
CREATE TABLE [dbo].[Clientes] (
```

```
SQL_Latin1_General_CP1_CI_AS NULL ,
```

```
) ON [PRIMARY]
GO
```

```
CREATE TABLE [dbo].[Movimento] (
```

```
SQL_Latin1_General_CP1_CI_AS NOT NULL ,
```

```
SQL_Latin1_General_CP1_CI_AS NULL ,
```

```
SQL_Latin1_General_CP1_CI_AS NULL ,
```

```
) ON [PRIMARY]
GO
```

```
ALTER TABLE [dbo].[CCorrente] WITH NOCHECK ADD
CLUSTERED
```

```
GO
```

```
ALTER TABLE [dbo].[Clientes] WITH NOCHECK ADD
CLUSTERED
```

```
GO
```

```
ALTER TABLE [dbo].[Movimento] WITH NOCHECK ADD
CLUSTERED
```

```
GO
```

```
ALTER TABLE [dbo].[CCorrente] ADD
```

```
GO
```

```
ALTER TABLE [dbo].[Movimento] ADD
```

```
KEY
```

```
GO
```

```
[cliente_cod] [int] NOT NULL ,
[cliente_nome] [nvarchar] (50) COLLATE
```

```
[Agencia] [int] NULL
```

```
[CCorrente] [nvarchar] (10) COLLATE
```

```
[Data_Hora] [datetime] NOT NULL ,
[Tipo] [char] (1) COLLATE
```

```
[CC_Creditada] [nvarchar] (10) COLLATE
```

```
[valor] [money] NULL
```

```
CONSTRAINT [PK_CCorrente] PRIMARY KEY
```

```
(
    [CCorrente]
) ON [PRIMARY]
```

```
CONSTRAINT [PK_Clientes] PRIMARY KEY
```

```
(
    [cliente_cod]
) ON [PRIMARY]
```

```
CONSTRAINT [PK_Movimento] PRIMARY KEY
```

```
(
    [CCorrente],
    [Data_Hora]
) ON [PRIMARY]
```

```
CONSTRAINT [FK_CCorrente_Clientes] FOREIGN KEY
```

```
(
    [Cliente_cod]
) REFERENCES [dbo].[Clientes] (
    [cliente_cod]
) ON DELETE CASCADE ON UPDATE CASCADE
```

```
CONSTRAINT [FK_Movimento_CCorrente] FOREIGN KEY
```

```
(
    [CCorrente]
) REFERENCES [dbo].[CCorrente] (
    [CCorrente]
) ON DELETE CASCADE ON UPDATE CASCADE ,
CONSTRAINT [FK_Movimento_CCorrente1] FOREIGN
```

```
(
    [CC_Creditada]
) REFERENCES [dbo].[CCorrente] (
    [CCorrente]
)
```

A aplicação completa pode ser encontrada nos arquivos das aulas.

FILTRANDO DADOS DE UMA TABELA

O Delphi possui uma maneira simples de filtrar os dados de objetos da classe *TDATASET*. Quando nos referimos à classe *TDATASET* estamos também nos referindo a todos os seus descendentes, como o *TADOQUERY*, *TTABLE*, etc.

Sempre que possível devemos optar por filtrar os dados na própria instrução SQL, como na cláusula *WHERE*, porém, em algumas situações isso não é possível e então a propriedade *FILTER* torna-se interessante. Quando a tabela está processando um filtro sua propriedade *STATE* passa para *dsState*.

Ao executarmos a propriedade *RecordCount* sobre uma tabela “Filtrada” este indicará a quantidade de registros que passaram pelo filtro.

Para que o filtro seja executado a propriedade ***FILTERED*** deve ser alterada para ***TRUE***. Para desligar o filtro basta alterá-la para ***False***.

Para filtrar uma tabela devemos construir o filtro. Esta etapa pode ser realizada de 2 formas:

- Através da propriedade *FILTER* da Tabela:

A propriedade *Filter* : String do objeto *TDataSet* permite que seja informado um filtro no formato String. Este filtro pode ser informado em tempo de execução. Exemplos:

- `Tabela.Filter := 'codigo = 3';`
- `Tabela.Filter := 'codigo = 3 and salario < 5000';`
- `Tabela.Filter := '((codigo = 3) and (salario < 5000)) or (idade > 100) '; // expressões`
- `Tabela.Filter := 'nome = ' + quotedStr('MARIA DA SILVA') ; // nome igual ao informado`
- `Tabela.Filter := 'nome like ' + QuotedStr('%silva%'); // todos que tem silva no nome`
- `Tabela.Filter := 'desconto > acrescimo'; // comparando 2 campos da tabela`

- Através do Evento *onFilterRecord* da Tabela:

Este evento permite que o filtro seja elaborado de forma mais apurada. Podemos utilizar estruturas de decisão, repetição, etc. na construção do filtro. Este evento é acionado para cada registro da tabela em questão a fim de verificar se ele “passará” pelo filtro. A variável *ACCEPT* indica se o registro deve ou não ser aceito.

Ex:

```
procedure TForm1.quFuncionariosFilterRecord(DataSet: TDataSet; var Accept: Boolean);
begin
    // só são aceito registros cujo salario é maior que os descontos
    if DataSet.fieldbyname('valor_salario').asFloat >=
        DataSet.fieldbyname('valor_Atrasos').asFloat +
        DataSet.fieldbyname('valor_convenio').asFloat then
        accept := true
    else
        accept := false;
end;
```

No exemplo acima poderíamos ter utilizado a tabela *quFuncionarios* ao invés do parâmetro *DATASET*.

FRAMES

Um Frame é, digamos assim, uma mistura das facilidades de manipulação de um Painel com a robustez e reutilização de código dos Formulários. O propósito deste componente é criar uma classe incorporada de vários outros componentes. Uma das coisas mais interessantes do Frame que é ele pode também ser adicionado a sua palheta de componentes, assim como os Component Templates, só que com muito mais facilidade e, sendo assim instanciado futuramente em outras aplicações de forma independente da original.

A utilização de Frames é muito simples, basta clicar em 'File | New | Other | Delphi Files | Frame'. Um novo Frame é criado na forma de um formulário. Podemos então incluir os objetos que desejamos utilizar, escrever alguns manipuladores de evento para os componentes. Depois basta simplesmente incluir em um Formulário um objeto Frame da palheta de componentes Standard. Selecione o Frame criado na janela exibida e clique em Ok. Será adicionado um objeto parecido com o TPanel no Formulário contendo os objetos incluídos no Frame que você criou.

As alterações feitas em objetos TFrame são automaticamente refletidas nas instâncias que a referenciam. O código é mantido na unidade do Frame.

Na prática, o uso dos Frames é útil quando se deseja utilizar o mesmo grupo de componentes em vários formulários dentro de um aplicativo.

Os Frames podem ser encarados como objetos. Podemos criar variáveis públicas que podem atuar como propriedades, além de métodos e eventos tornando assim os Frames mais genéricos e poderosos. Ex:

Vamos criar um frame que encapsule as ações mais comuns realizadas em um cadastro, como Inclusão, Alteração, Exclusão, Botões de Navegação, etc. Isto é muito útil pois em aplicações profissionais todas as telas de cadastro costumam (pelo menos deveriam) seguir um padrão, com as mesmas funcionalidades.

Crie uma nova aplicação. O Objetivo será criar 2 telas de cadastro, podemos utilizar o banco NorthWind e criar cadastros para as tabelas *products* e *Regions*. Crie uma tela principal (um menu) e adicione 2 botões, um para cada cadastro.



Agora vamos criar o Frame: File | New | Other | Delphi Files | Frame

Deixe o frame com a seguinte aparência:



Adicione à cláusula *USES* de nosso frame as seguintes units: *ADODB*, *DB*. Elas serão necessárias para a manipulação de objetos ADO e *TDataSet*.

Este frame será utilizado em TODAS as telas de cadastro (no nosso caso, em 2) e ele não fará qualquer referência à tela onde foi instanciado. Porém ele necessita saber qual é a tabela que deverá manipular. Por exemplo, ao clicar-se no botão INSERT deve ser criado um novo registro na tabela para inclusão.

Uma forma simples seria criar uma variável na seção PUBLIC do frame. Poderíamos chamá-la de TABELA e nela apontaríamos a tabela que deve ser manipulada. Quando utilizarmos este frame na tela de cadastro de Produtos apontaremos esta variável para quPRODUCTS e quando a tela for cadastro de Regiões a tabela apontada será quREGIONS. Vamos ver como fazer isso.

Na seção PUBLIC do frame crie a seguinte variável:

```
public
{ Public declarations }
TABELA : TADOQuery;
```

A programação dos botões do frame segue abaixo Vela pelo nome da procedure a qual botão se aplica o código:

```
procedure TFrameBotoes.btNovoClick(Sender: TObject);
begin
    TABELA.insert;
end;

procedure TFrameBotoes.btEditaClick(Sender: TObject);
begin
    TABELA.edit;
end;

procedure TFrameBotoes.btDeleteClick(Sender: TObject);
begin
    TABELA.delete;
end;

procedure TFrameBotoes.btOKClick(Sender: TObject);
begin
    TABELA.post;
end;

procedure TFrameBotoes.btCancelClick(Sender: TObject);
begin
    TABELA.cancel;
end;

procedure TFrameBotoes.BtFirstClick(Sender: TObject);
begin
    TABELA.first;
end;

procedure TFrameBotoes.btPriorClick(Sender: TObject);
begin
    TABELA.prior;
End7;

procedure TFrameBotoes.BtNextClick(Sender: TObject);
begin
    TABELA.next;
```

```

end;

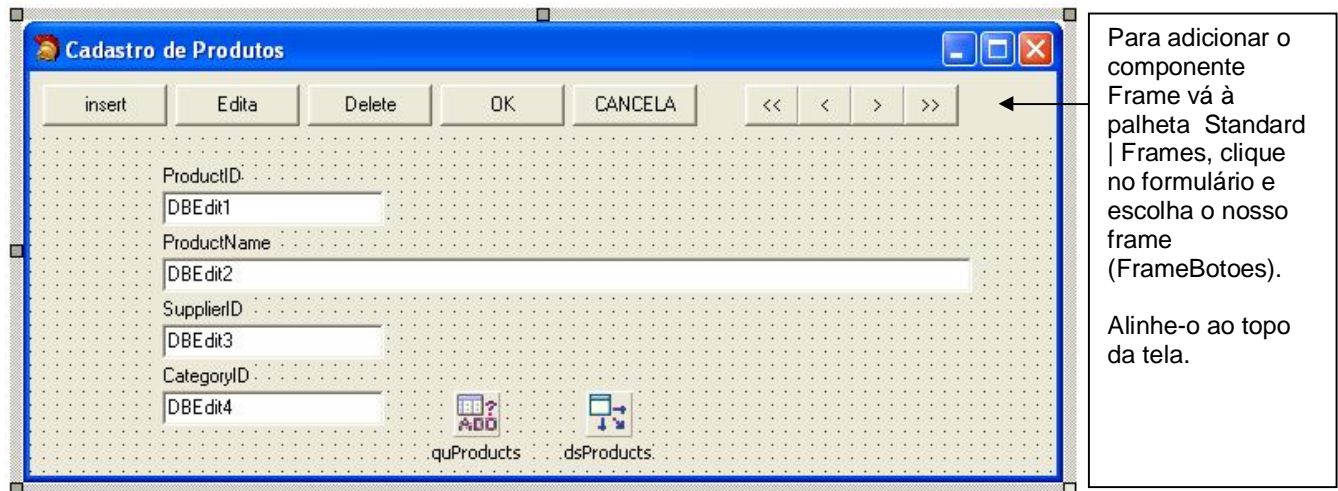
procedure TFrameBotoes.btLastClick(Sender: TObject);
begin
  TABELA.last;
end;

```

Na propriedade NAME do frame coloque “FRAMEBOTOES” e SALVE SEU TRABALHO. Agora que nosso frame já está QUASE pronto podemos começar a construir nossa tela de Cadastro.

Crie um datamodule e adicione um objeto TADOCONNECTION. Configure-o para acessar o banco NORTHWIND.

Crie um novo formulário e monte uma tela para cadastro de produtos como segue abaixo:



Ainda no formulário de cadastro programe os eventos onShow e onClose para que a tabela quProducts seja Aberta e Fechada respectivamente.

Adicione também ao evento onShow a seguinte linha : **FrmBotoes.tabela := quProducts;** Este comando atualiza a variável pública TABELA do frame para que ele saiba qual a tabela que ele deverá manipular.

Atualize nosso menu para que ele exiba a tela de cadastro de produtos ao ser pressionado o respectivo botão.

Neste instante já é possível testar nossa aplicação. Porém, veja que apesar de todos os botões funcionarem corretamente eles não estão sendo habilitados/desabilitados. Como já vimos anteriormente, uma forma simples de se fazer isso seria programando o evento *OnStateChange* do objeto TDATASOURCE. Mas há um inconveniente: teríamos que realizar esta programação em todos os formulários de cadastro! E o pior: ela seria quase igual para todos eles, apenas o nome da tabela seria alterado.

Uma forma simples e eficiente para resolver este problema seria criarmos um método para controlar os botões no nosso FRAME.

Vamos ver como:

Primeiro crie uma *procedure* publica em nosso FRAME. Vá á clausula PUBLIC e adicione abaixo de nossa variável TABELA a seguinte declaração:

```
public
{ Public declarations }
TABELA : TADOQuery;
Procedure ControlaBotoes;
end;
```

Agora implemente a *procedure* como segue:

```
procedure TFrameBotoes.ControlaBotoes;
begin
  btNovo.enabled := TABELA.state = dsBrowse;
  btEdita.enabled := TABELA.state = dsBrowse;
  btDelete.enabled := TABELA.state = dsBrowse;
  BtFirst.enabled := TABELA.state = dsBrowse;
  btPrior.enabled := TABELA.state = dsBrowse;
  BtNext.enabled := TABELA.state = dsBrowse;
  btLast.enabled := TABELA.state = dsBrowse;

  btOK.enabled := TABELA.state <> dsBrowse;
  btCancel.enabled := TABELA.state <> dsBrowse;
end;
```

Chegou a hora de utilizar esta tralha toda! No nosso programa de cadastro de produtos, coloque o seguinte código no evento OnStateChange do objeto TDATASOURCE:

```
procedure TForm1.dsProductsStateChange(Sender: TObject);
begin
  FrmBotoes.ControlaBotoes;
end;
```

onde FRMBOTOES é o nome da instância do FRAME no formulário.

Pronto! Execute a aplicação e veja como ficou!

Agora faça o mesmo no cadastro de regiões.

Se você quiser incluir novos botões ao Frame, como por exemplo um botão para Fechar a tela é só incluir no FRAME principal e programá-lo. Todas as telas que instanciaram o Frame serão atualizadas automaticamente.

Os arquivos completos podem ser encontrados nos arquivos das aulas.

OBJETO TFIELD

Os campos de um Dataset são sempre representados por componentes invisíveis, derivados da classe TFIELD. O acesso ao valor corrente de um campo pode ser feito através de métodos e propriedades do TField associado ao campo. Os objetos que representam o campo não são da classe TFIELD, mas de classes derivadas dele. Eles são chamados de TFields, mas a classe específica a que pertencem depende do tipo do campo associado, definido na criação da tabela.

TADTField	TDateField	TReferenceField	TAggregateField
TDateTimeField	TSmallIntField	TArrayField	TFloatField
TSQLTimeStampField	TAutoIncField	TFMTBCDField	TStringField
TBCDField	TGraphicField	TTimeField	TBinaryField
TGuidField	TVarBytesField	TBlobField	TDispatchField
TVariantField	TBooleanField	TIntegerField	TWideStringField
TBytesField	TInterfaceField	TWordField	TCurrencyField
TLargeintField	TDataSetField	TMemoField	

Exemplos de descendentes da classe TFIELD

Existem 2 formas de criar esses componentes:

- Criação dinâmica, feita automaticamente pelo Delphi;
- Criação de TFields persistentes, feita durante o projeto, através do **FieldsEditor**.

A Classe TFIELD possui diversas propriedades. Veja as mais utilizadas retiradas do Help do Delphi 2005:

Properties of TField

Alignment	Determines how the field's data is displayed within a data-aware control.
AsBCD	Represents the field's value as a TBcd value.
AsBoolean	Represents the field's value as a boolean value.
AsCurrency	Represents the field's value as a Currency value.
AsDateTime	Represents the field's value as a TDateTime value.
AsFloat	Represents the field's value as a double value.
AsInteger	Represents the field's value as a 32-bit integer.
AsSQLTimeStamp	Represents the field's value as a TSQLTimeStamp.
AsString	Represents the field's value as a string (Delphi) or an AnsiString (C++).
AsVariant	Represents the Value of the field as a Variant.

Calculated	Determines whether the value of the field is calculated by the OnCalcFields event handler of its dataset.
CanModify	Specifies whether a field can be modified.
DataSet	Identifies the dataset to which a field component belongs.
DataType	Identifies the data type of the field component.
DisplayLabel	Contains the text to display in the corresponding column heading of a data grid.
DisplayName	Represents the name of the field for display purposes.
DisplayText	Represents the field's value as it is displayed in a data-aware control.
DisplayWidth	Specifies the number of characters that should be used to display a field's value by a cooperating data-aware control.
EditMask	Contains a mask that restricts the data that can be entered into a data field.
FieldKind	Indicates whether a field represents a column in a dataset, a calculated field, or a lookup field.
FieldName	Indicates the name of the physical column in the underlying table or query result to which a field component is bound.
FieldNo	Indicates the ordinal position of the field's column in the underlying table or query result.
IsNull	Indicates whether the field has a value assigned to it.
KeyFields	Identifies the field or fields in the dataset that must be matched in a lookup dataset when doing a lookup.
Lookup	Determines whether the field is specified as a lookup field.
LookupCache	Determines whether the values of a lookup field are cached or looked up dynamically every time the current record in the dataset changes.
LookupDataSet	Identifies the dataset used to look up field values.
LookupKeyFields	Identifies the field or fields in the lookup dataset to match when doing a lookup.
LookupList	Indicates a cache of values from the LookupDataSet indexed by a set of values from the KeyFields property.
LookupResultField	Identifies the field from the lookup dataset whose value becomes the Value property of the field component.
NewValue	Represents the current value of the field component including pending cached updates.
OldValue	Represents the original value of the field (as a Variant).

ReadOnly	Determines whether the field can be modified.
Required	Specifies whether a nonblank value for a field is required.
Size	Indicates the size used in the definition of the physical database field for data types that support different sizes.
Text	Contains the string to display in a data-aware control when the field is in edit mode.
Value	Represents the data in a field component.
Visible	Determines whether the field appears in a data grid.

ACESSANDO OS DADOS DE UM TFIELD:

Podemos acessar os dados de um TField da seguinte forma:

```
Tabela.Fieldbyname('campo').asString;
```

Onde asString pode substituído por um outro tipo de retorno válido, como por exemplo asInteger caso o campo realmente possa ser transformado em um valor inteiro. Se não puder o retorno neste caso será zero.

Outra forma de acessar os campo de uma tabela é utilizar

```
Tabela['campo'];           ou  
Tabela.FieldValues['campo'];
```

Porém o retorno será sempre do tipo Variant.

Existem outras formas, como através da propriedade **FIELDS** do objeto **TDATASET** que é um vetor que inicia do zero contendo todos os TField's criados. Ex:

```
Tabela.fields[0].asString;
```

FIELDS EDITOR

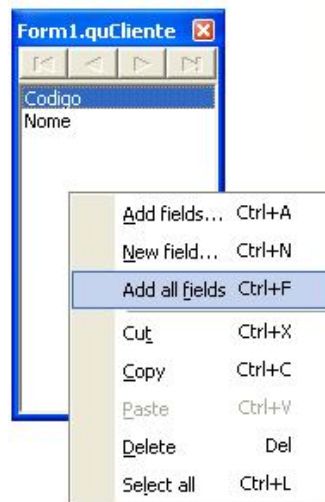
O Fields Editor é uma ferramenta fornecida pelo Delphi para criação de `TField`'s em tempo de programação. A vantagem de se criar `TFields` persistentes é que eles servem como uma referência para os campos da tabela em questão. Podemos assim especificar em tempo de desenvolvimento características especiais para sua edição e exibição nos componentes `dataware`. Podemos também realizar validações, criar campos virtuais, calculados, lookups.

Porém temos que tomar um cuidado especial quando adicionamos os campos em tempo de projeto. Neste processo o Delphi cria esses campos de acordo como eles foram criados fisicamente no banco de dados. Isso quer dizer que, caso esses campos sejam alterados no banco de dados, como por exemplo o tamanho ou o tipo de dados, o Delphi vai gerar uma exceção ao executar a tela pois o campo não poderá mais ser criado visto que seu tipo mudou. Isso também pode acontecer se o sistema trabalha com diferentes SGBD's, como Oracle e Sql-Server. Alguns campos podem não ser compatíveis entre os SGBD's e o Delphi criará um tipo para cada banco caso os campos estejam sendo criados automaticamente, porém se os mesmos forem criados em tempo de projeto podem ocorrer erros.



Fields Editor

Para exibir o Fields Editor basta dar um duplo clique sobre o objeto `Tdataset`.



Para adicionar os campos automaticamente devemos clicar com o botão direito do mouse sobre o Fields Editor e escolher a opção *Add all Fields*

O Delphi irá criar um objeto `TFIELD` para cada campo retornado pela instrução SQL.

A grande maioria das propriedades podem ser editadas em tempo de desenvolvimento através do *object inspector*. Por exemplo, podemos inserir máscaras de edição como em campos Data, formatação para valores fracionados, etc.

Algumas propriedades, como *Displayformat* e *EditFormat* são específicas de campos que representam valores numéricos.

Observe que o Delphi inseriu automaticamente no código da Unit os códigos necessários para a criação dos componentes `TFIELD`.

VALIDANDO ATRAVÉS DO EVENTO ONVALIDADE DOS OBJETOS TFIELD

Os componentes TField permitem que se faça uma validação através do evento *OnValidate*. A forma como o Delphi sabe se tudo está OK é através de uma exceção. Para notificar o Delphi que o valor não deve ser aceito, basta disparar uma exceção

O interessante de se realizar validações neste evento é que o usuário não consegue passar para o próximo campo enquanto não corrigir o problema no campo atual. Isso é extremamente interessante quando se está editando em um componente do tipo *DBGrid*.

Este evento é disparado sempre que o valor do campo for alterado. Via código ou via interferência do usuário.

Ex:

```
procedure TForm1.quClienteNomeValidate(Sender: TField);
begin
  if length(sender.asString) < 3 then
  begin
    showmessage('O nome deve ter pelo menos de 3 letras');
    abort;
  end;
end;
```

No código acima podemos observar algumas coisas importantes:

- O parâmetro *SENDER : TFIELD* representa o campo ao qual este evento foi associado, neste caso ao campo *Nome* facilmente identificado pelo próprio nome da procedure.
- Poderíamos ter utilizado diretamente o nome do campo, como veremos mais adiante
- A exceção é gerada no comando **ABORT**; que gera uma exceção silenciosa.

Outra forma que produziria o mesmo resultado seria:

```
procedure TForm1.quClienteNomeValidate(Sender: TField);
begin
  if length(quClienteNome.AsString) < 3 then
  begin
    raise exception.create('O nome deve ter pelo menos de 3 letras');
  end;
end;
```

Neste caso utilizamos diretamente o nome do componente *TFIELD* *quClienteNome* e geramos a exceção através do comando *raise*.

CAMPOS CALCULADOS

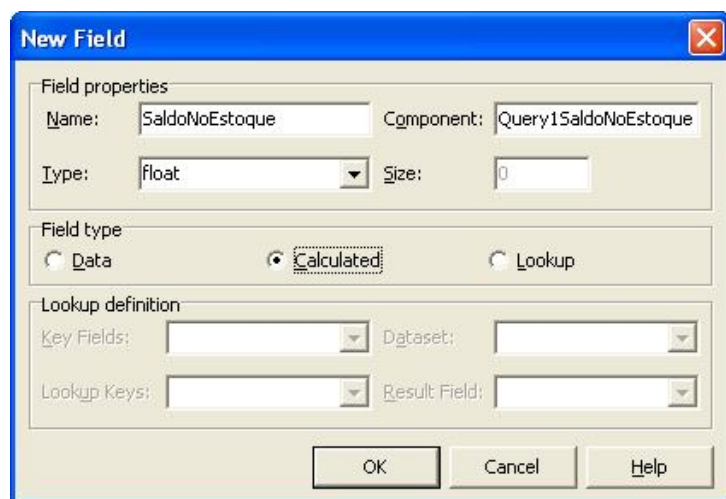
Campos calculados são campos pertencentes ao *Dataset* mas que não pertencem à tabela original. Eles são usados para exibir valores calculados dinamicamente, geralmente com base no valor de outros campos do *Dataset*.

O valor destes campos é calculado através do evento *onCalcFields* do objeto *TDataset*. Mas cuidado: Este evento é executado com muita frequência portanto seu código deve ser pequeno e eficiente. Na execução deste evento o Delphi muda o estado do *Dataset* para *CalcFields* e então podemos atribuir valores apenas aos campos calculados. Vamos ver um exemplo:

Para este exemplo utilizaremos a tabela *Products* do banco *NorthWind* do *SqlServer*. Nesta tabela temos um cadastro de produtos. Utilizaremos 2 campos em particular: **UnitsInStock** e **UnitPrice**. O primeiro indica a quantidade de produtos no estoque e o segundo o preço unitário. Imagine que seja necessário exibir em uma grade o Saldo no estoque, um valor calculado resultante da Fórmula: $\text{SaldoNoEstoque} = \text{UnitsInStock} * \text{UnitPrice}$. Este campo saldo não existe na tabela física, portanto este é um caso típico onde podemos utilizar com eficácia os campos calculados. É fato que neste exemplo poderíamos resolver o problema colocando na própria instrução SQL o novo campo como: **'Select UnitsInStock e UnitPrice as "SaldoNoEstoque", ...'**, porém imagine que esses campos fossem editáveis na grade, o que nos obrigaria a recalculá-los assim que o usuário alterasse algo.

Crie uma tela com um *dbGrid*, *TADOConnection*, *TADOQuery* e um *TDataSource*, ligue-os e na instrução SQL do *TADOQuery* digite: **'select productName, ProductId, unitPrice, unitsInStock, categoryId from products'** sem os apóstrofes é claro!

Agora clique com o botão direito sobre o objeto *TADOQuery* e adicione todos os campos "ADD ALL FIELDS".



Assim que adicionar todos os campos clique novamente com o botão direito e escolha a opção "NEW FIELD". A tela à esquerda deverá surgir:

Nesta tela podemos preencher as propriedades do campo calculado, como o nome, tipo, tamanho, etc. Preencha-a como a da imagem.

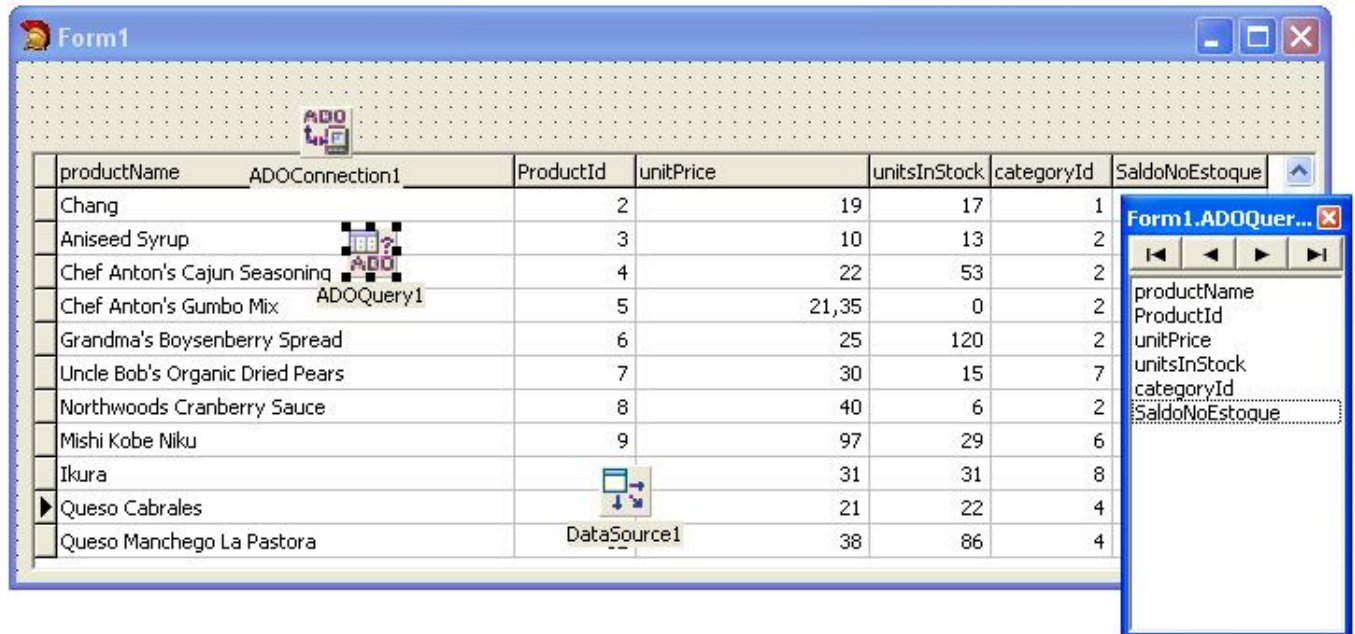
Observe que a opção **CALCULED** deve ser selecionada.

Clique em **OK** e o campo aparecerá no *Fields Editor*.

Agora precisamos programar o evento *onCalcFields* do objeto *TADOQuery*. Para isso selecione o objeto e dê um duplo clique neste evento. Em sua programação digite:

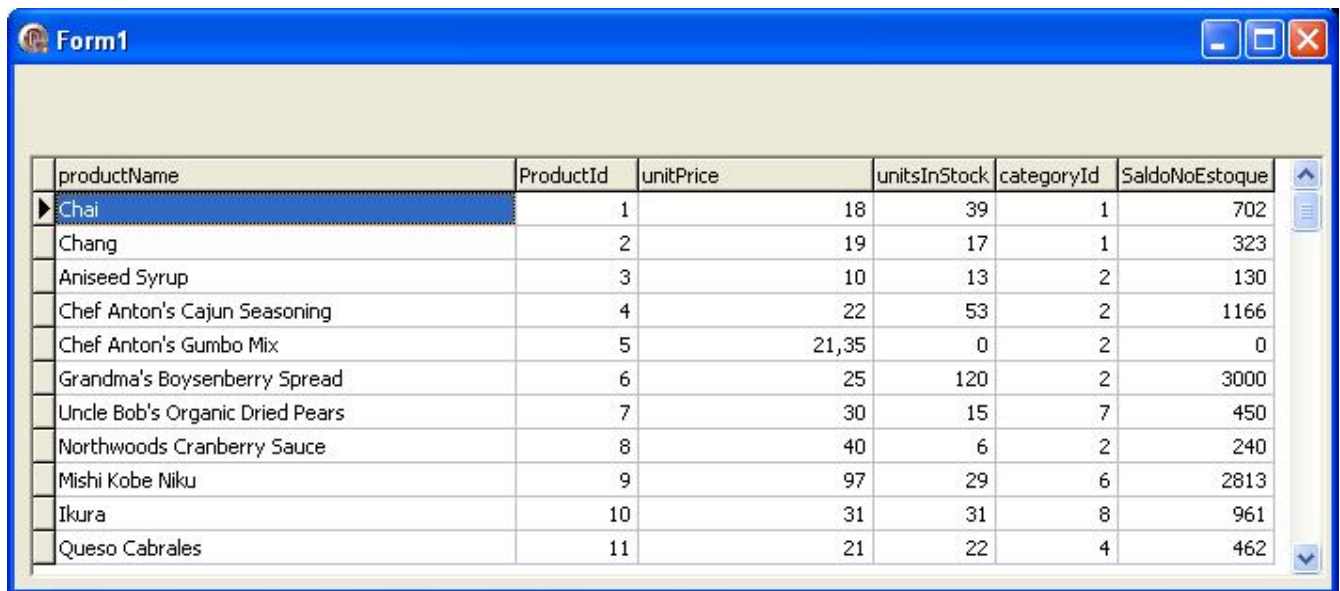
```
Dataset['SaldoNoEstoque'] := DataSet.FieldName('unitPrice').asFloat *
    DataSet.FieldName('unitsInStock').asFloat ;
```

Durante o desenvolvimento sua tela deverá se parecer com a da figura abaixo:



Após executar o programa, o Delphi irá executar o evento onCalcFields para cada registro. Não esqueça de abrir a conexão e a Query no evento onShow e fechar no onClose!!!

Sua tela deverá se parecer com a tela abaixo:



Os campos calculados podem ser utilizados também para dar significado à campos codificados, como por exemplo o campo Sexo, que só armazena 'M' ou 'F' mas deve exibir 'Masculino' ou 'Feminino'.

CAMPOS LOOKUP

Os campos *LookUp*, assim como os campo Calculados, não pertencem à Tabela Original, eles são criados com o intuito de exibir valores obtidos em outro *DataSet*, através de alguma chave de pesquisa obtida no *DataSet* Original.

Para que possamos utilizar esse tipo de campo, a tabela de pesquisa precisa estar aberta e com todos os registros disponíveis, o que pode comprometer a performance do aplicativo.

Esse tipo de campo pode ser substituído facilmente por um *JOIN* na tabela original, porém o uso de joins pode inabilitar a edição de *DataSets*, já o uso de campos lookup não! De qualquer forma, hoje em dia, com a disseminação dos SBGD's o uso de campos lookup diminuiu bastante.

Para criar um campo lookup utilizaremos como base o exemplo utilizado no tópico “CAMPOS CALCULADOS”. Observe que na tabela *Products* temos um campo denominado *CategoryId* que é uma chave estrangeira da tabela *Categories* onde está contido o campo *CategoryName* que é o campo que nos interessa exibir no *DbGrid*, junto com os dados da tabela *Products*.

Como já dissemos anteriormente, poderíamos resolver este problema facilmente inserindo uma instrução SQL do tipo :

```
“select productName, categoryName, ProductId, unitPrice, unitsInStock, Categories.categoryId  
from products, categories  
where products.categoryID = categories.categoryID”
```

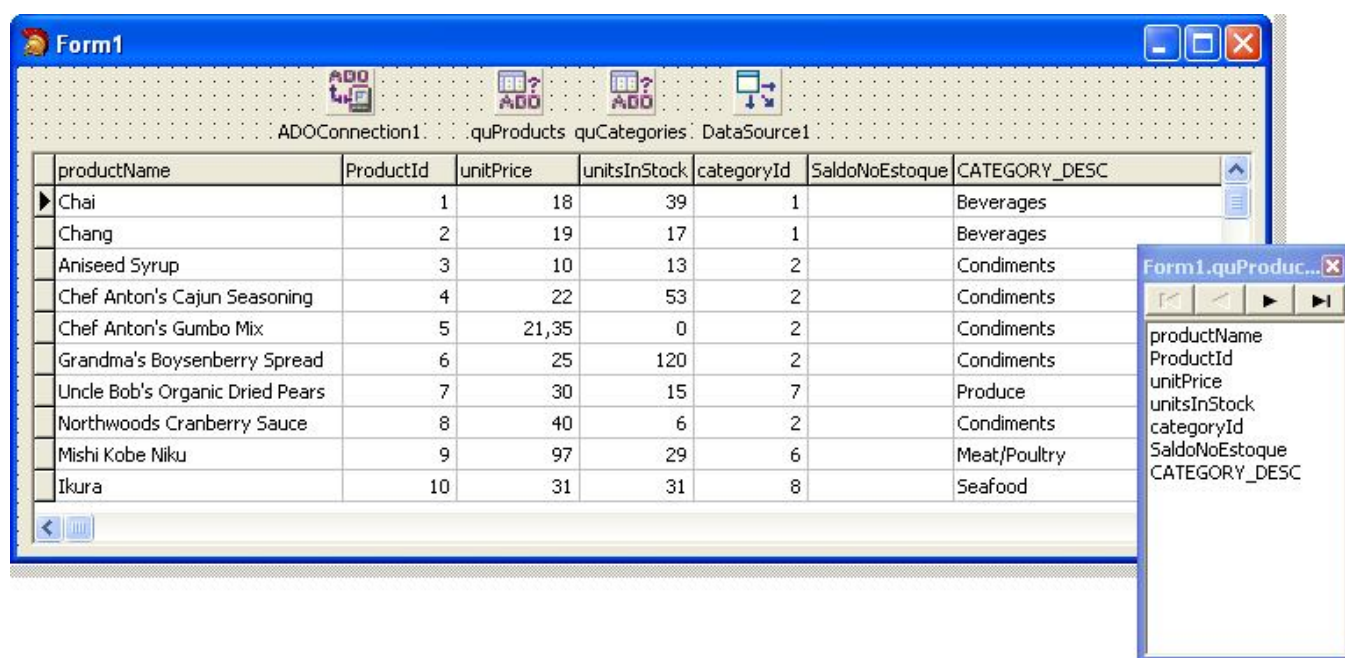
Mas vamos resolver utilizando os campos *LookUp*. Para tanto, siga os seguintes passos:

1. Vamos renomear a query ADOQUERY1 para quProducts;
2. Crie uma nova Query, renomeie-a para quCategories e insira a seguinte instrução Sql: ‘*Select * from categories*’. Inclua a sua abertura no evento *onShow* e fechamento no *onClose*;
3. Clique com o botão direito sobre o objeto quProdutcs escolha a opção “NEW FIELD”. Novamente a tela de cadastro surgirá e você deverá preenchê-la na seguinte ordem de formas que o resultado final se assemelhe com a tela baixo;

4. No campo **name** Preencha com o nome do Campo, pode ser qualquer nome que identifique o campo que será exibido, desde que este nome já não tenha sido usado;

5. Em **Type** devemos escolher o tipo do campo que será exibido, neste caso exibiremos a descrição da categoria, que é um String;
6. No campo **Size** devemos especificar o tamanho do campo, ele não é utilizado em campos numéricos.;
7. No **Field Type** selecione LookUp;
8. No campo **Key Field** selecione o campo chave estrangeira na tabela quProducts que está relacionado à tabela quCategories, no caso o *CategoryID*;
9. No campo **DataSet** preencha com o nome da Tabela onde será feita a pesquisa, no caso, quCategories;
10. No campo **LookUp Keys** devemos agora selecionar qual é o campo chave da tabela informada em **DataSet** que está relacionado com o campo informado em **Key Field**;
11. Indicar no campo **Result Field** qual o campo da tabela informada em **DataSet** deverá ser exibido.

Veja como fica a tela em tempo de projeto e com as queries abertas:



A título de curiosidade, este problema também poderia ter sido resolvido com campos Calculados.

O EVENTO ONGETTEXT

O evento `onGetText` pertence a classe `TFIELD` e é utilizado para mascarar o conteúdo de um campo, exibindo um valor diferente do originalmente armazenado. Ele pode ser muito útil para valores codificados. Exemplo: um campo `SEXO` que só armazena 'M' ou 'F' mas deve exibir 'Masculino' ou 'Feminino'. Isso também pode ser feito com campos calculados, porém neste caso teríamos que criar um novo campo, já com a utilização evento `OnGetText` não precisamos criar um novo campo pois o novo valor será exibido no mesmo campo onde a informação está contida!

Ex:

Para este exemplo utilizaremos o mesmo exemplo utilizado no tópico `CAMPOS LOOKUP`. Porém, para a realização deste exemplo não são necessários campos *Lookup* ou *Calculados*. Nosso objetivo aqui será fazer com que o campo *Discontinued* apresente em vídeo os valores SIM para (1) e Não para (0). Porém veja que o campo *discontinued* é do tipo Bit e o Delphi reconhece este tipo como sendo `BOOLEAN`, portanto deveremos tratar como `True` ou `False` ao invés de 1 e 0. Roteiro:

1. Primeiro, insira mais um campo à instrução SQL da query `quProducts` : *discontinued*. Este campo armazena os valores 0 ou 1 para indicar se o produto em questão foi discontinued (true) ou não (false);
2. Agora, crie o campo em tempo de projeto clicando com o botão direito no *Fields Editor* e selecionando a opção *ADD ALL FIELDS*;
3. Ainda no *Fields Editor*, selecione o novo campo adicionado (*Discontinued*) e programe o evento *onGetText* da seguinte forma:

```
procedure TForm1.quProductsdiscontinuedGetText(Sender: TField; var Text: string; DisplayText: Boolean);
begin
  if sender.AsBoolean = true then // pode ser sender ou então quProductsdiscontinued
    Text := 'Sim'
  else
    Text := 'Não';
end;
```

4. O parâmetro `TEXT` indica o conteúdo que deve ser exibido para o campo passado no parâmetro `SENDER`.
5. Execute o programa e veja como ficou.

discontinued
False
False
False
False
True
False
False
False
True
False

Antes de processar o evento `onGetText`

discontinued
Não
Não
Não
Não
Sim
Não
Não
Não
Sim
Não

Após processar o evento `onGetText`

O EVENTO ONSETTEXT

O evento *onSetText* pertence a classe *TFIELD* e é utilizado para gravar um valor diferente daquele exibido nos componentes *Data Controls*. Sempre que utilizarmos o evento *onGetText* e a tabela em questão for passível de alterações, como em um cadastro, onde temos edição e inserção, devemos programar também o evento *onSetText*.

No exemplo dado no tópico *onGetText*, alteramos o valor de exibição do campo *Discontinued* de *true* para *Sim* e de *false* para *Não*. Se o usuário fosse realizar qualquer alteração neste campo ele informaria também *Sim* ou *Não*, mas nunca *True* ou *False*. Porém, ao efetivarmos as alterações na tabela física precisamos gravar os valores corretos. Neste caso, não podemos gravar *Sim* tão pouco *Não*!

Sendo assim, programamos no evento *onSetText* o que deve ser gravado para cada opção foi utilizado o *GetText*.

Para completarmos o exemplo dado no item “O EVENTO ONGETTEXT” vamos programar o evento *onSetText* do campo *Discontinued* através do *Fields Editor*. Deixe-o assim:

```
procedure TForm1.quProductsdiscontinuedSetText(Sender: TField; const Text: string);
begin
  if Text = 'Sim' then
    sender.value := true
  else
    sender.value := false;
end;
```

Onde *SENDER* representa o campo que contém o dado, no caso *quProductsdiscontinued*. O parâmetro *TEXT* representa o valor que é exibido para o campo através do *onGetText*.

TABELAS TEMPORÁRIAS COM O TCLIENTDATASET

O objeto *TClientDataSet* permite a criação de tabelas temporárias (em memória) de forma muito simples. Sendo ela descendente da classe *TCustomClientDataSet* (que descende de *TDataSet*), podemos realizar todas as operações básicas de tabelas. As tabelas temporárias podem ser utilizadas como fazíamos nos vetores de registro, porém sua utilização é mais fácil visto que ela funciona exatamente como uma tabela: permite inclusões, alterações, etc. Permite até que os dados sejam salvos no formato binário ou XML. Precisamos inclusive abri-la antes de utilizá-la.

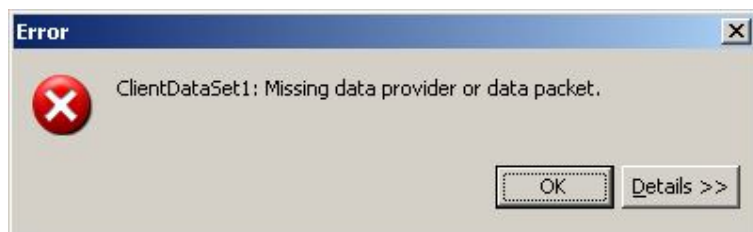
Inclua no *USES* de seu projeto (menu *VIEW | UNITS* e escolha o arq. de projeto) as seguintes bibliotecas: ***MidasLib* e *crtl***. Caso contrário você terá de disponibilizar o arquivo *MIDAS.DLL* junto com a sua aplicação.

Para criar uma tabela temporária, vá a paleta **DATA ACCESS** e escolha o componente ***TClientDataSet***. Este componente também é utilizado para acessar dados com a tecnologia do *DBExpress*, portanto precisamos fazer umas “mutretas” para que ele também funcione “desconectado” de um banco de dados.

Para tanto precisamos criar o objeto via código e este processo só pode ser realizado uma vez. Um bom local para fazer isso é no evento *onCreate* do formulário pois este só é disparado uma vez. Lá inserimos a linha:

```
TabTemporaria.createDataset;
```

Onde *TabTemporaria* é a tabela temporária da classe *TClientDataSet*. Se tentarmos abrir a tabela em tempo de programação receberemos o erro:



Isso acontece porque como já dissemos anteriormente este componente pode trabalhar conectado a um banco de dados. Para evitar esse erro, clique com o botão direito do mouse sobre o objeto *TClientDataSet* e escolha a opção *Create DataSet*.

Feito isso podemos utilizar a tabela normalmente. Podemos criar campos, “clonar” os campos de uma outra tabela através da opção *Assign local data*, tudo isso via *Fields Editor*. Podemos criar um cadastro inteiro, com *dbGrid*’s, *dbEdit*’s, etc. e ao final salvar os dados em um arquivo XML para posterior recuperação.

Algumas características interessantes do objeto *TClientDataSet*:

- Método *EmptyDataSet*: Apaga todos os registros da tabela temporária. Ex:

```
ClientDataSet1.EmptyDataSet;
```

- Método *SaveToFile*: Permite exportar dados no formato XML e binário. Ex:

```
ClientDataSet1. SaveToFile('Dados Exportados.xml', dfXml);
```

- Método *LoadFromFile*: Permite importar dados no formato XML e binário. Ex:

```
ClientDataSet1.LoadFromFile('Dados Exportados.xml');
```

- Propriedade *IndexFieldNames*: Permite mudar o índice da tabela temporária, ordenando-a em tempo de execução. Caso haja mais de um campo, estes devem ser separados por ponto-e-vírgula. Ex:

```
ClientDataSet1.IndexFieldNames := 'data_nascimento;nome'; // indexa por 2 campos
```


FORMULÁRIOS PADRÕES

Em grandes sistemas, existem formulários que tem características muito parecidas, como formulários de cadastro, pesquisa, relatórios, etc. Em um formulário padrão, utilizaremos o poder da HERANÇA para criar um formulário PAI, com as principais características de um determinado tipo de formulário para, depois, criar formulários filhos que herdem essas características, tendo assim o programador que apenas ajustar as características das telas filho.

Isso se traduz em produtividade pois o programador não precisa se preocupar com os detalhes básicos da implementação, pois estes já foram definidos no form Pai.

Para exemplificar, vamos criar um formulário padrão que poderá ser utilizado como referência para qualquer tela. Ele terá as seguintes características:

1. Se auto-destruirá ao ser fechado;
2. Possibilitará trocar de campos com a tecla ENTER.
3. Fechar qualquer objeto DATASET (ou descendentes) utilizado.

É claro que poderíamos incluir mais funcionalidades, porém para simplificar ficaremos apenas com essas.

O código deste formulário seria este (nomeio para fmPadrao e retire-o do AutoCreate):

Form TfmPadrao

```
procedure TfmPadrao.FormClose(Sender: TObject; var Action: TCloseAction);
var i : integer;
begin
  for i:=0 to componentcount -1 do
    if components[i] is TDataSet then
      (components[i] as TDataSet).close;
  action := caFree;
end;

procedure TfmPadrao.FormKeyPress(Sender: TObject; var Key: Char);
begin
  if key = chr(13) then
    begin
      if Not(activeControl is TMemo) and
        Not(activeControl is TDbGrid) then begin
        selectnext(activeControl, true, true);
        key := chr(0);
      end;
    end;
end;
```

NÃO UTILIZE ESTE FORMULÁRIO!!!! NÓS UTILIZAREMOS APENAS OS DESCENDENTES DELE.

Para criar um formulário baseado no TfmPadrao, vá ao menu FILE | NEW | OTHER | INHERITABLE ITENS , escolha o ícone com o nome FmPadrao e clique em OK.

Agora o Delphi criará um formulário filho de TfmPadrao. Ele poderá ser alterado, adicionando-se novos elementos, porém nenhum elemento pertencente ao seu pai poderá ser eliminado.

Este formulário filho terá todas as características presentes em seu pai, como por exemplo a capacidade de trocar de campo com a tela ENTER, assim como se faz com o TAB.

Caso esta característica não seja desejada no formulário filho, basta programar o evento que também foi programado no PAI e comentar o comando **inherited**. Este comando indica ao compilador que ele deve executar a funcionalidade que foi programada no PAI.

Ex: Imagine que no formulário filho não é desejado a troca de campos com ENTER. Para fazer isso basta editar o evento onKeyPress do form Filho e comentar a linha inherited.

Se quiser adicionar novas funcionalidades, como por exemplo tocar um som ao mudar de campo no formulário filoho com ENTER, basta adicionar o comando após o comando inherited no evento onKeyPress no FILHO.

ALTERAÇÕES FEITAS NO PAI SERÃO REFLETIDAS NOS FILHOS AUTOMATICAMENTE, A MENOS QUE O COMANDO INHERITED TENHA SIDO OMITIDO NO FILHO.

CRIANDO UM CADASTRO MASTER - DETAIL

Existem diversas formas de se construir um cadastro Mestre-Detalhe ou Máster–Detail. Uma das formas mais simples é aquela que utiliza um cadastro com DBase para a tabela Master e um objeto DBGrid para os dados da tabela Detail. O faremos aqui utilizando esta estrutura. Vamos criar uma tela para venda de produtos. Teremos um cadastro de Nota e itens da nota. Abaixo listaremos o código das principais telas do sistema. Utilizaremos o banco NorthWind e as tabelas Products e Customers. Criaremos mais 2 tabelas, Nota e NotItem (veja código abaixo).

As telas de pesquisa de produtos e pesquisa de clientes não tem nada a não ser a instrução SQL que retorna todos os dados de suas respectivas tabelas, além da configuração do DBGrid. O menu tem apenas uma chamada à tela de Vendas.

Basicamente, os passos para a criação deste cadastro são:

1. Utilizando um novo formulário baseado no formulário padrão de cadastro, crie um cadastro para os dados da Nota.
2. Na mesma tela, coloque uma nova query (qultens) e um DbGrid.
3. A abertura da query qultens está condicionada ao evento AfterScroll da tabela Nota.
4. Altere os botões GRAVAR, EXCLUIR e CANCELAR da nota pois agora eles deverão gravar e excluir também os dados da tabela qultens.

{ -- Script para as tabelas Nota e NotItem

```
if exists (select * from dbo.sysobjects where id = object_id(N'[dbo].[Nota]') and OBJECTPROPERTY(id, N'IsUserTable') = 1)
drop table [dbo].[Nota]
GO
```

```
if exists (select * from dbo.sysobjects where id = object_id(N'[dbo].[NotItem]') and OBJECTPROPERTY(id, N'IsUserTable') = 1)
drop table [dbo].[NotItem]
GO
```

```
CREATE TABLE [dbo].[Nota] (
```

```
NULL
) ON [PRIMARY]
GO
```

```
[NotaID] [int] NOT NULL ,
[nota_data] [datetime] NULL ,
[customerID] [nchar] (5) COLLATE Latin1_General_CI_AS
```

```
CREATE TABLE [dbo].[NotItem] (
```

```
) ON [PRIMARY]
GO
```

```
[NotaID] [int] NOT NULL ,
[ProductID] [int] NOT NULL ,
[Qtde] [int] NULL ,
[Valor_unit] [numeric](18, 2) NULL
```

```
ALTER TABLE [dbo].[Nota] WITH NOCHECK ADD
```

```
GO
```

```
CONSTRAINT [PK_Nota] PRIMARY KEY CLUSTERED
(
    [NotaID]
) ON [PRIMARY]
```

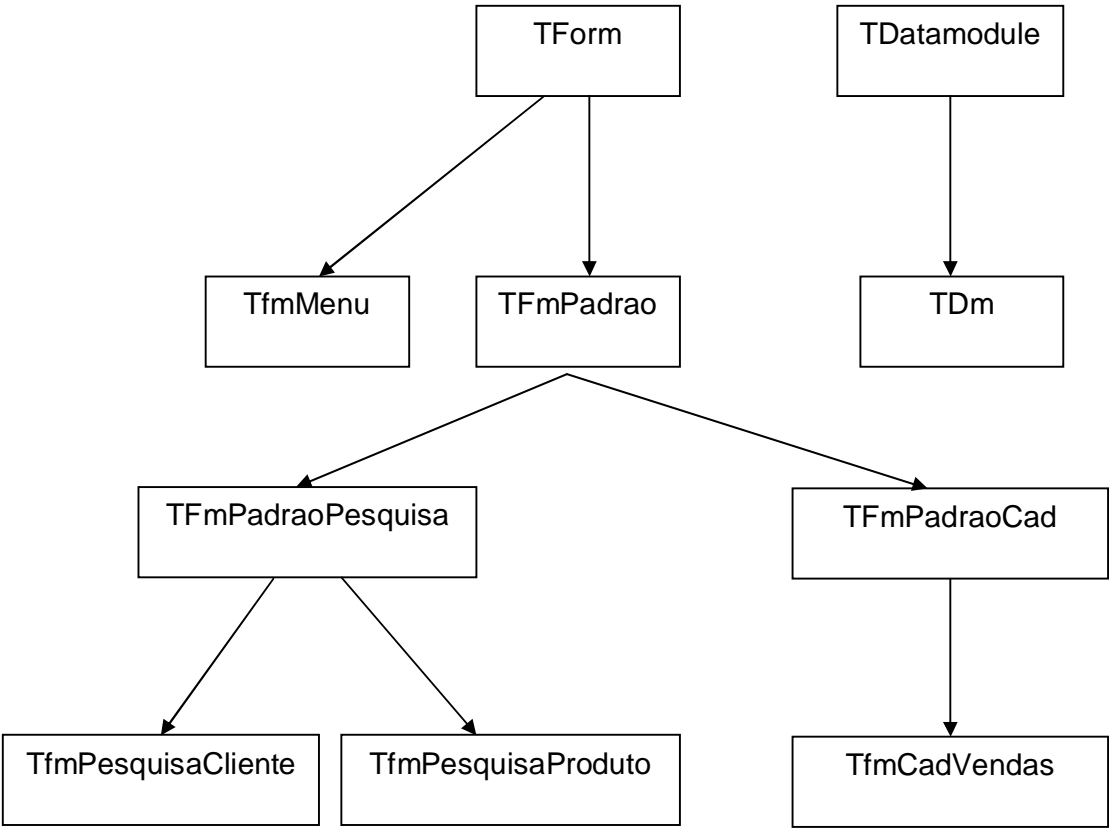
```
ALTER TABLE [dbo].[NotItem] WITH NOCHECK ADD
```

```
CLUSTERED
```

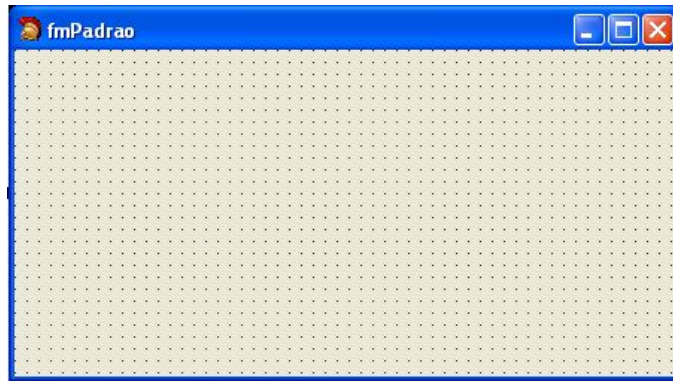
```
GO
```

```
CONSTRAINT [PK_NotItem] PRIMARY KEY
(
    [NotaID],
    [ProductID]
) ON [PRIMARY]
```

Hierarquia dos Formulários utilizados no programa.



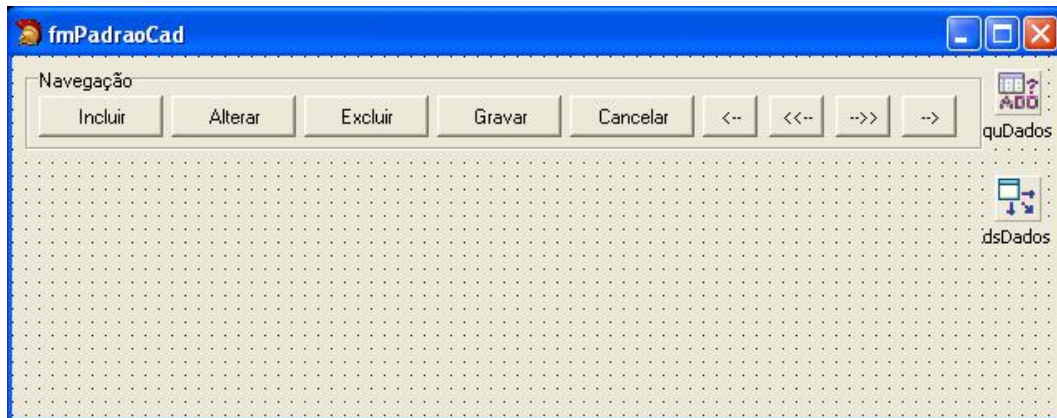
Código do formulário Padrão – TFmPadrao



```
procedure TfmPadrao.FormClose(Sender: TObject; var Action: TCloseAction);
var i : integer;
begin
  for i:=0 to componentcount -1 do
    if components[i] is TDataSet then
      (components[i] as TDataSet).close;
    action := caFree;
  end;
end;
```

```
procedure TfmPadrao.FormKeyPress(Sender: TObject; var Key: Char);
begin
  if key = chr(13) then
    begin
      if Not(activeControl is TMemo) and
        Not(activeControl is TDbGrid) then begin
        selectnext(activeControl, true, true);
        key := chr(0);
      end;
    end;
end;
```

Código do Formulário Padrão para Cadastros– TFmPadraoCad



```

procedure TfmPadraoCad.BtnNovoClick(Sender: TObject);
begin
    quDados.insert;
end;

procedure TfmPadraoCad.btnAlterarClick(Sender: TObject);
begin
    quDados.edit;
end;

procedure TfmPadraoCad.btnExcluirClick(Sender: TObject);
begin
    if application.messagebox('Confirma?', 'Atenção', mb_yesno + mb_iconquestion +
        mb_defButton2) = id_no then exit;
    quDados.Delete;
    quDados.UpdateBatch;
end;

procedure TfmPadraoCad.btnGravarClick(Sender: TObject);
begin
    quDados.post;
    quDados.UpdateBatch;
end;

procedure TfmPadraoCad.btnCancelarClick(Sender: TObject);
begin
    quDados.cancel;
    quDados.CancelUpdates;
end;

procedure TfmPadraoCad.btnFirstClick(Sender: TObject);
begin
    quDados.First;
end;

procedure TfmPadraoCad.btnPriorClick(Sender: TObject);
begin
    quDados.prior;
end;

procedure TfmPadraoCad.btnNextClick(Sender: TObject);
begin
    quDados.next;
end;

procedure TfmPadraoCad.btnLastClick(Sender: TObject);
begin
    quDados.last;
end;

procedure TfmPadraoCad.dsDadosStateChange(Sender: TObject);
begin
    BtnNovo.enabled := quDados.State = dsBrowse;
    btnAlterar.enabled := quDados.State = dsBrowse;
    btnExcluir.enabled := quDados.State = dsBrowse;
    btnFirst.enabled := quDados.State = dsBrowse;
    btnPrior.enabled := quDados.State = dsBrowse;
    btnNext.enabled := quDados.State = dsBrowse;
    btnLast.enabled := quDados.State = dsBrowse;

    btnGravar.enabled := not(quDados.State = dsBrowse);

```

```

btnCancela.enabled := not(quDados.State = dsBrowse);
end;

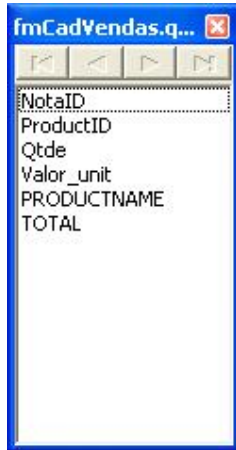
```

```

procedure TfmPadraoCad.FormShow(Sender: TObject);
begin
    inherited;
    quDados.open;
end;

```

Código fonte da tela de Vendas - TfmCadVendas:



No Fields Editor da Tabela *quTens*, foram adicionados 2 campos calculados: *PRODUCTNAME* e *TOTAL*.

Altere a propriedade das queries (*quDados* e *quTens*)

LockType = ItBatchOptimistic

Assim é possível utilizar os dados em batch update mode (memória)

```

procedure TfmCadVendas.FormShow(Sender: TObject);
begin
    inherited;
    // inserindo mascaras via código para não precisar usar o fields editor na tabela quDados
    (quDados.fieldbyname('nota_data') as TDateTimeField).DisplayFormat := 'dd/mm/yyyy';
    (quDados.fieldbyname('nota_data') as TDateTimeField).EditMask := '99/99/9999;1;';

```

end;

```
// ao fecharmos o formulário, verificamos se o usuário está inserindo ou alterando
// uma nota, e, neste caso, perguntamos se ele deseja realmente sair e cancelamos
// todos os updates pendentes.
procedure TfmCadVendas.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  if quDados.state <> dsBrowse then begin
    if application.MessageBox('Os dados não foram salvos. Deseja sair mesmo assim?',
      'Atenção', mb_yseno + mb_iconquestion) = id_no then begin
      action := caNone; // não deixa fechar
      exit;
    end
  else begin
    quDados.cancelupdates;
    qultens.CancelUpdates;
  end;
end;
inherited;
end;
```

```
// realiza a pesquisa de clientes.
procedure TfmCadVendas.btnPesqClienteClick(Sender: TObject);
begin
  fmPesqCliente := TfmPesqCliente.create(self);
  fmPesqCliente.showmodal('contactName', 'customerId');
  if fmPesqCliente.Retorno <> '' then quDados['customerId'] := fmPesqCliente.Retorno;
end;
```

```
// O evento AfterScroll do objeto TADOQuery é disparado sempre que se altera o registro
// apontado. Usaremos ele para que os items sejam atualizados, exibindo-se assim
// apenas aqueles que pertencem à Nota Exibida.
procedure TfmCadVendas.quDadosAfterScroll(DataSet: TDataSet);
begin
  inherited;
  qultens.close;
  qultens.Parameters.ParamByName('p_nota').Value := quDados.fieldbyname('notald').asInteger;
  qultens.open;
end;
```

```
// habilitação de botões, grades, etc. de acordo com o estado da tabela Nota
procedure TfmCadVendas.dsDadosStateChange(Sender: TObject);
begin
  inherited;
  btnPesqCliente.enabled := not(quDados.State = dsBrowse);
  DBGrid1.ReadOnly := quDados.State = dsBrowse;
end;
```

```
// o evento DataChange de um DataSource ocorre a cada mudança nos dados do registro
// atual. Usaremos ele para atualizar o campo Nome do Cliente que está nos
// dados da nota. Assim que o usuário informar o código do cliente, o sistema
// irá procurar através da função RetornaNomeCliente se ele existe
procedure TfmCadVendas.dsDadosDataChange(Sender: TObject; Field: TField);
begin
  inherited;
  edNomeCliente.text := dm.RetornaNomecliente(DBEdit3.text);
end;
```



```

// Este evento ocorre sempre que o botão de pesquisa do DBGrid é pressionado.
// Ele foi configurado na coluna do Código do Produto, através da propriedade
// ButtonStyle = cbsEllipsis.
procedure TfmCadVendas.DBGrid1EditButtonClick(Sender: TObject);
begin
  if quDados.state = dsBrowse then exit; // só pode pesquisar se estiver cadastrando

  fmPesqProdutos := TfmPesqProdutos.create(self);
  fmPesqProdutos.showmodal('ProductName', 'ProductID');

  if fmPesqProdutos.Retorno <> '' then begin
    // pode ocorrer de a tabela não estar em modo de edição ou em inserção, então o faremos aqui
    if qultens.state = dsBrowse then qultens.edit;
    qultens['productId'] := fmPesqProdutos.Retorno;
  end;
end;

// na hora de cancelar, temos que cancelar todos os dados da nota e dos itens que
// porventura foram digitados.
procedure TfmCadVendas.btnCancelaClick(Sender: TObject);
begin
  //inherited;
  quDados.Cancel;
  quDados.CancelBatch;
  qultens.Cancel;
  qultens.CancelUpdates;
end;

procedure TfmCadVendas.btnGravaClick(Sender: TObject);
var Ponteiro : TBookmark;
    erro : boolean;
begin
  // validações da nota ...
  if quDados.fieldbyname('nota_data').isNull then
    raise exception.create('Informe a data.');
```

```

  if length(edNomeCliente.text) = 0 then
    raise exception.create('Informe o código do cliente.');
```

```

  // verifica se foi digitado algum item
  if qultens.IsEmpty then
    raise exception.create('Não foi informado nenhum item.');
```

```

  // valida item a item verificando se foi informado o código do produto
  // a quantidade e o valor unitário, além de verificar produtos repetidos
  qultens.first;
  while not qultens.eof do
  begin
    if qultens.FieldName('productName').AsString = '' then
      raise exception.create('Não foi informado o produto.');
```

```

    if qultens.FieldName('qtde').AsInteger = 0 then
      raise exception.create('Não foi informado a quantidade.');
```

```

    if qultens.FieldName('valor_unit').asFloat = 0 then

```

```

    raise exception.create('Não foi informado o valor unitário.');
```



```

// rotina para verificar se existem produtos duplicados
// vamos filtrar por produto e contar se há mais de 1
// como há poucos registros na grade itens, o uso do filtro não é problemático.
Ponteiro := quitens.GetBookmark; // marca o registro atual
quitens.Filter := 'productId =' + quitens.fieldbyname('productId').asString;
quitens.Filtered := true;
erro := quitens.RecordCount > 1;
quitens.Filtered := false;
quitens.GotoBookmark( ponteiro ); // volta ao registro marcado
quitens.FreeBookmark( ponteiro ); // libera os recursos alocados pelo ponteiro

if erro then
    raise exception.create('Não pode have produtos repetidos!');

quitens.next
end;
```



```

if application.MessageBox('Confirma a gravação?', 'Atenção',
    mb_yesno + mb_iconquestion) = id_no then exit;

try
    // se não houver relacionamentos no banco, deve-se apagar os itens por aqui...
    dm.ADOConnection1.BeginTrans;

    // pegando o próximo código de nota. Isso só deve ocorrer em uma inclusão.
    if quDados.state = dsInsert then begin
        dm.quAux.close;
        dm.quAux.sql.text := 'select isNull( max(notalD) +1, 1) "valor" from Nota';
        dm.quAux.open;
        quDados['NotalD'] := dm.quAux['valor'];
        dm.quAux.close;
    end;

    quDados.post;
    quDados.UpdateBatch; // grava a nota

    // preencher o código da nota na tabela itens, pois durante as inclusões ele nao foi informado
    quitens.first;
    while not qultens.eof do
        begin
            quitens.edit;
            quitens['notalD'] := quDados['notalD'];
            quitens.post;
            qultens.next;
        end;
    quitens.UpdateBatch; // grava todos os itens

    dm.ADOConnection1.CommitTrans;
except
    on e:exception do
        begin
            dm.ADOConnection1.RollbackTrans;
            showmessage('Erro: ' + e.message);
        end;
    end;
end;
```

```

// ao apagar uma nota, apaga-se também todos seus itens pois não há relacionamento
// entre essas tabelas no banco. Tudo é feito em um bloco protegido e com
// controle de transação.
procedure TfmCadVendas.btnExcluiClick(Sender: TObject);
begin
  if application.MessageBox('Deseja realmente apagar?', 'Atenção',
    mb_yesno + mb_iconquestion) = id_no then exit;

  try
    // se não houver relacionamentos no banco, deve-se apagar os itens por aqui...
    dm.ADOConnection1.BeginTrans;

    // apagando os itens
    quitens.first;
    while not quitens.eof do quitens.delete;
    quitens.UpdateBatch; // aplica as atualizações

    quDados.delete;
    quDados.UpdateBatch; // aplica as atualizações

    dm.ADOConnection1.CommitTrans;
  except
    on e:exception do
      begin
        dm.ADOConnection1.RollbackTrans;
        showmessage('Erro: ' + e.message);

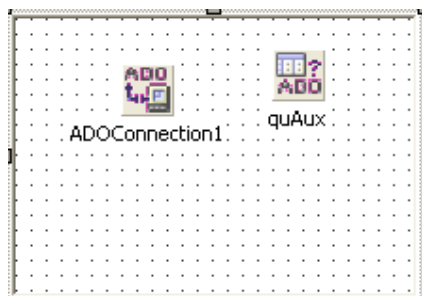
        // para atualizar os itens...
        quDados.close;
        quDados.open;
      end;
    end;
  end;

  // campos calculados da tabela itens: Descrição do produto e Total (valor unit. * qtde)
  procedure TfmCadVendas.quitensCalcFields(DataSet: TDataSet);
  begin
    DataSet['PRODUCTNAME'] := dm.RetornaDescProduto( DataSet.fieldbyname('ProductId').asInteger );

    // calcula apenas se os 2 campos foram informados.
    if (not DataSet.fieldbyname('Valor_unit').isnull) and
      (not DataSet.fieldbyname('qtde').isnull) then
      DataSet['TOTAL'] := DataSet.fieldbyname('Valor_unit').AsFloat *
        DataSet.fieldbyname('qtde').AsInteger;
  end;
end;

```

Código Fonte do Datamodule – TDM



```

function TDM.RetornaDescProduto(p_codigo: integer): String;
begin
    quaux.close;
    quaux.SQL.text := 'select ProductName from Products where ProductID = ' + intToStr(p_codigo);
    quaux.open;
    Result := quaux.fieldbyname('ProductName').asString;
    quaux.close;
end;

function TDM.RetornaNomecliente(p_codigo: string): string;
begin
    quaux.close;
    quaux.SQL.text := 'select contactName from customers where customerID = ' + quotedStr(p_codigo);
    quaux.open;
    Result := quaux.fieldbyname('contactName').asString;
    quaux.close;
end;

```



TfmPadraoPesquisa

Código Fonte da tela Padrão de Pesquisa

```

public
{ Public declarations }
Retorno : String;
Procedure ShowModal(campoPesquisa,CampoRetorno : String);

procedure TfmPadraoPesquisa.BtnOkClick(Sender: TObject);
begin
    if ADOQuery1.IsEmpty then exit;
    Retorno := ADOQuery1.fieldbyname(w_CampoRetorno).asString;
    close;
end;

procedure TfmPadraoPesquisa.btnCancelClick(Sender: TObject);
begin
    close;
end;

```

end;

```
procedure TfmPadraoPesquisa.ShowModal(campoPesquisa, CampoRetorno: String);  
begin
```

```
    retorno := "";  
    w_CampoPesquisa := campoPesquisa;  
    w_CampoRetorno := CampoRetorno;  
    adoQuery1.open;  
    inherited ShowModal;  
end;
```

```
procedure TfmPadraoPesquisa.Edit1Change(Sender: TObject);  
begin
```

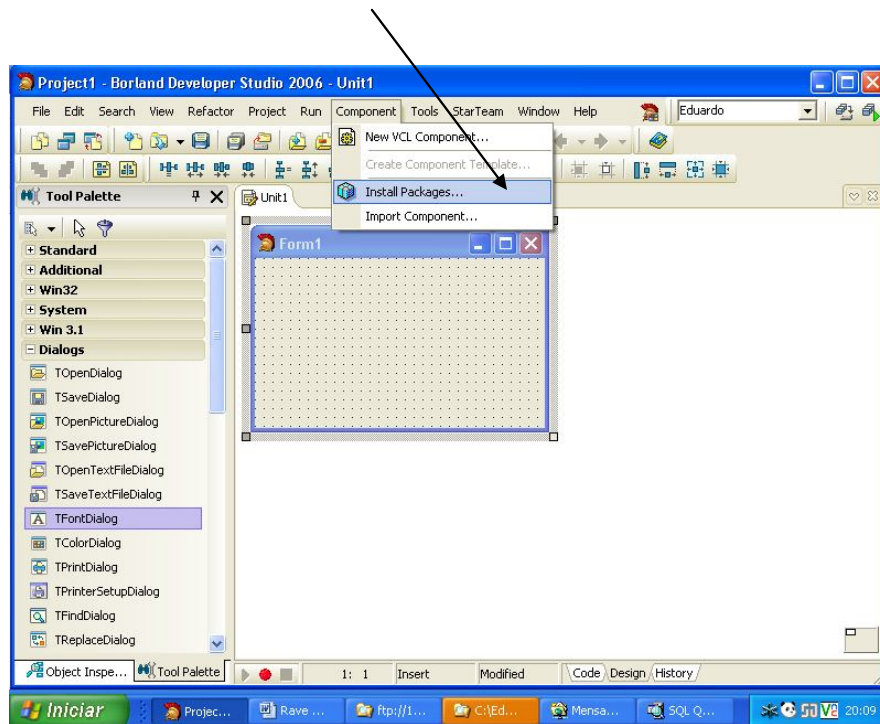
```
    ADOQuery1.Locate(w_CampoPesquisa, edit1.text, [loPartialKey, loCaseInsensitive]);  
end;
```

INSTALANDO E CRIANDO RELATÓRIOS NO FORTES REPORT PARA DELPHI 2006

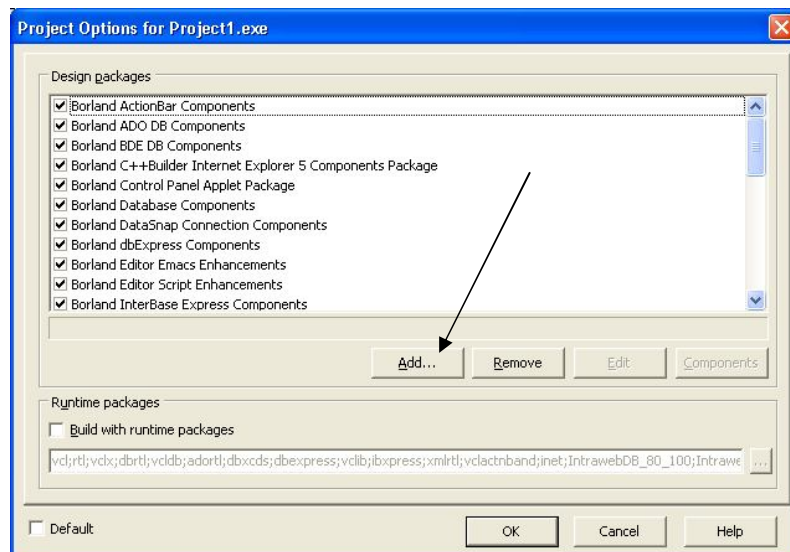
Antes de tudo: Verifique se o Fortes Report já está instalado no seu Delphi! Para tanto, veja se já há a palheta [Fortes Report]. Se já tiver instalado, não precisa instalar novamente!

Copie os arquivos do Fortes Report que estão no FTP para uma pasta que você tenha permissão de escrita.

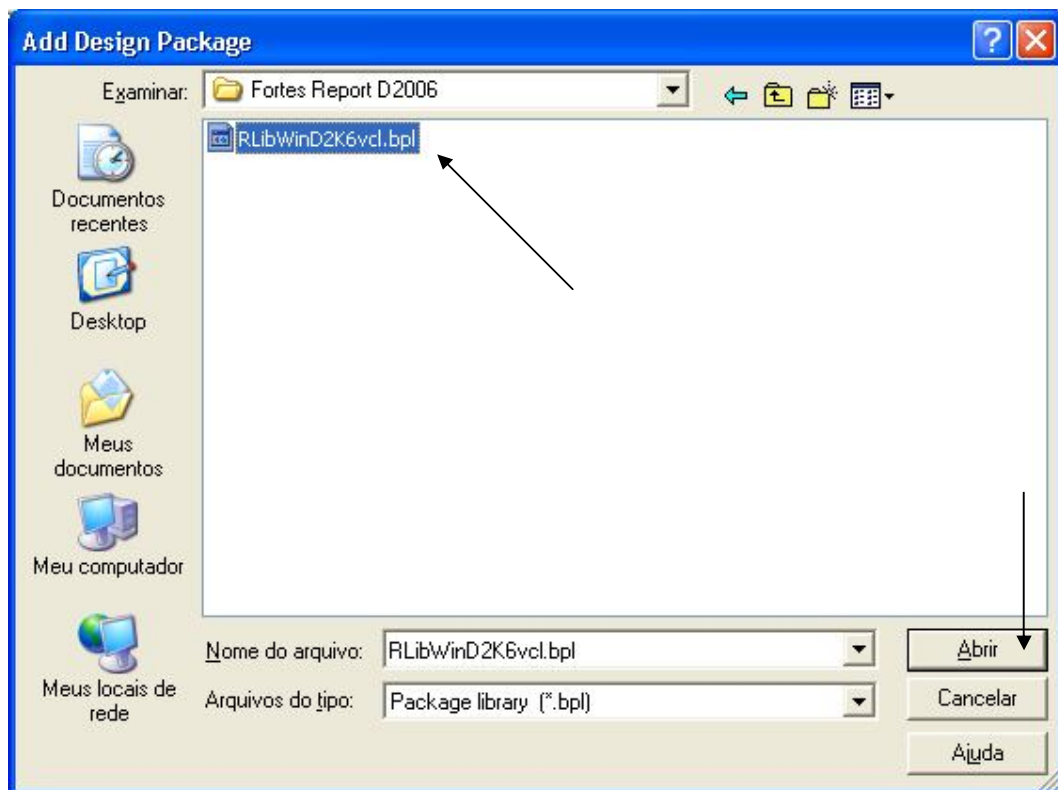
Depois no Delphi 2006, acesse o menu indicado abaixo:



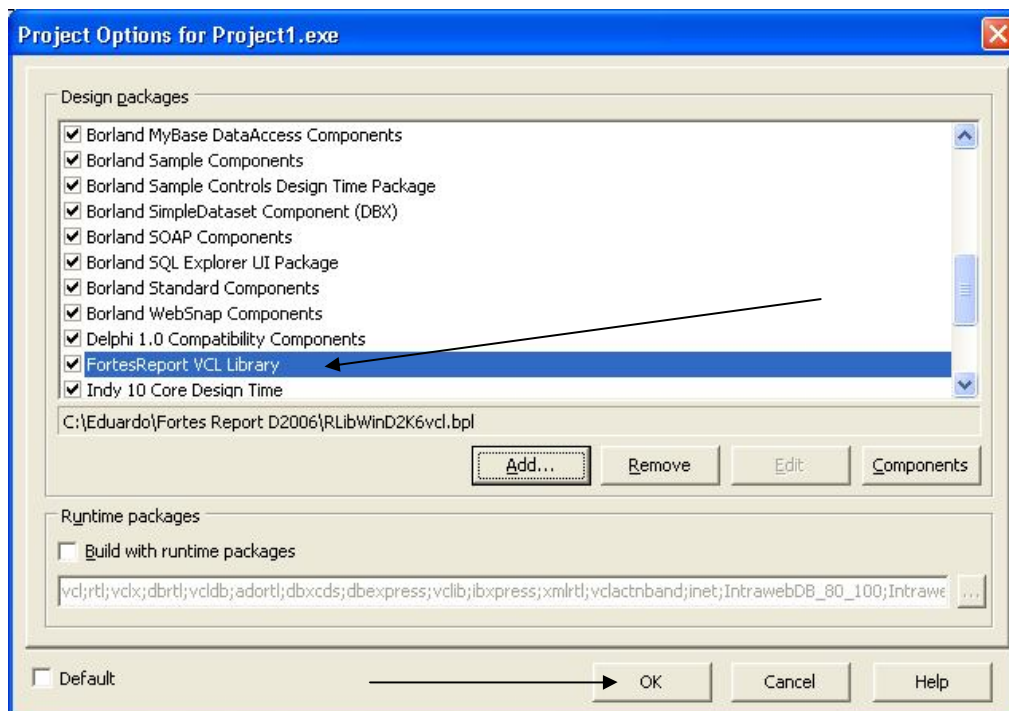
Clique no botão abaixo:



Aponte para a pasta onde você copiou os arquivos do Fortes Report e selecione o arquivo abaixo:



Depois, basta clicar no OK que ele já irá aparecer na palheta de componentes.

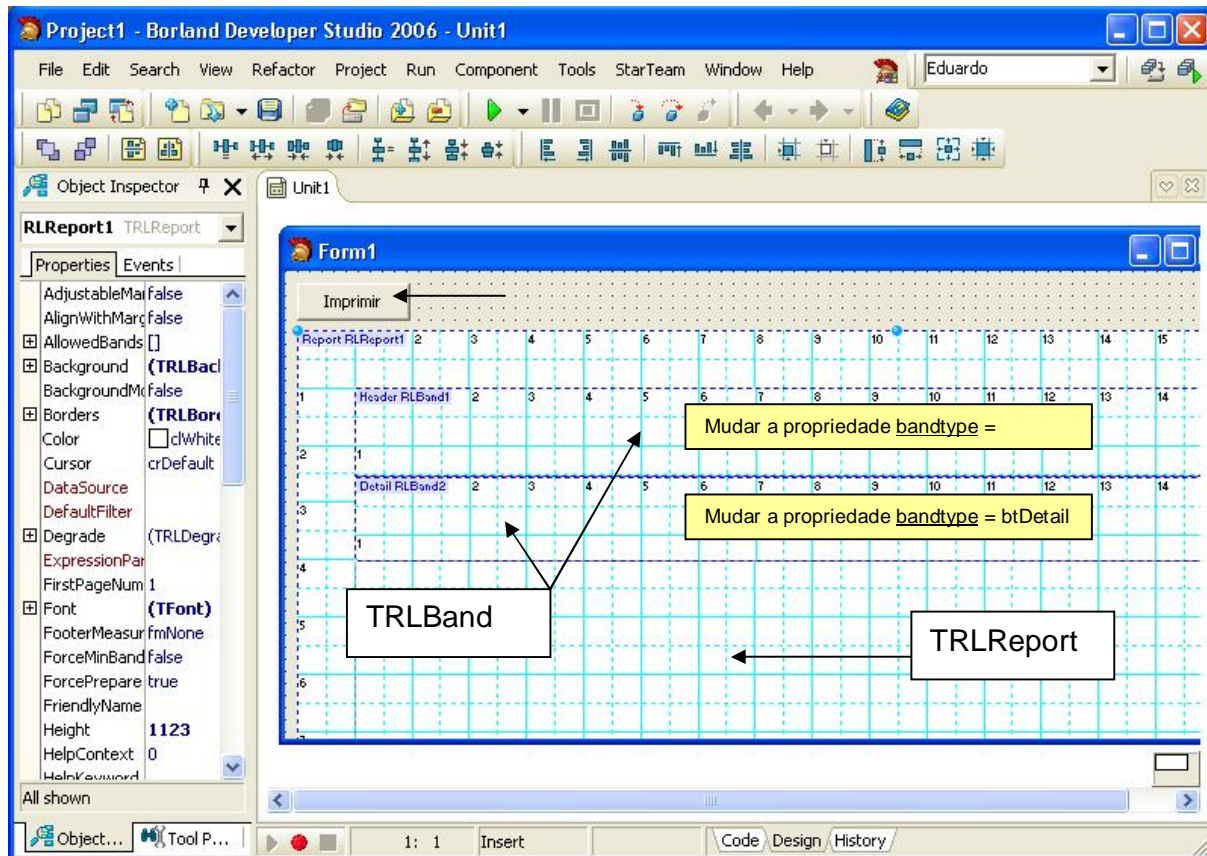


O Fortes Report irá aparecer como último na palheta de componentes do Delphi.

Criando um relatório sem quebras.

Crie um projeto novo. Insira no formulário principal um componente TRLReport, de formas que fique parecido com a tela abaixo:

Verificar sempre se os objetos foram colocados dentro do componente container principal.

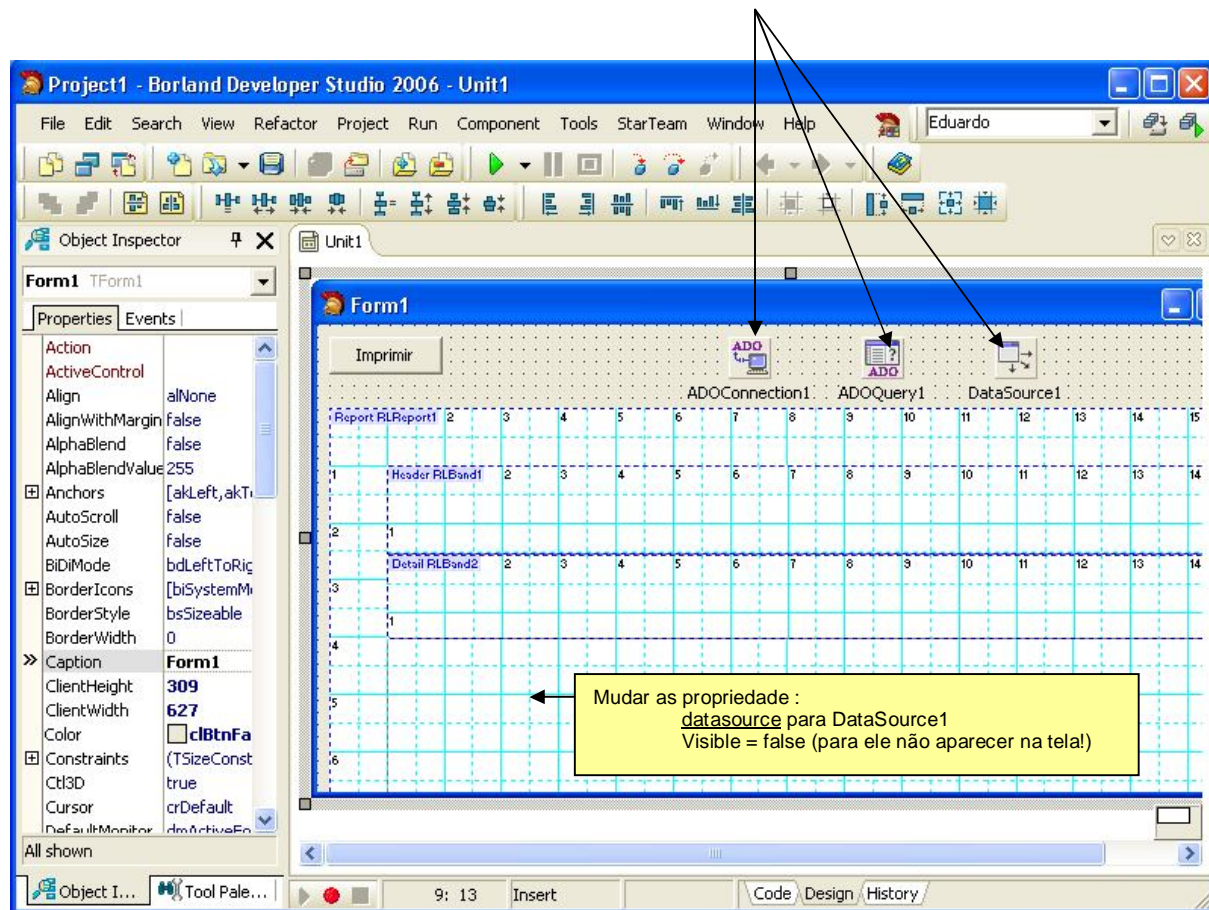


Todo o relatório é desenvolvido sobre o componente **TRLReport**, porém os componentes para exibição dos dados devem ser colocados sobre bandas. Na tela acima temos 2 bandas. Os principais tipos de banda são:

- **btDetail** – Banda que se repetirá para cada registro. Se a tabela tiver 1.000 registros, esta banda será impressa 1.000 vezes.
- **btFooter** – Esta banda será impressa na parte de baixo de todas as páginas do relatório.
- **btHeader** - Esta banda será impressa na parte de cima de todas as páginas do relatório.
- **btTitle** – Esta banda será impressa apenas na primeira página do relatório.
- **btSummary** - Esta banda será impressa apenas na última página do relatório.

Para fazer este relatório iremos acessar a tabela **products** do banco **northwind**.

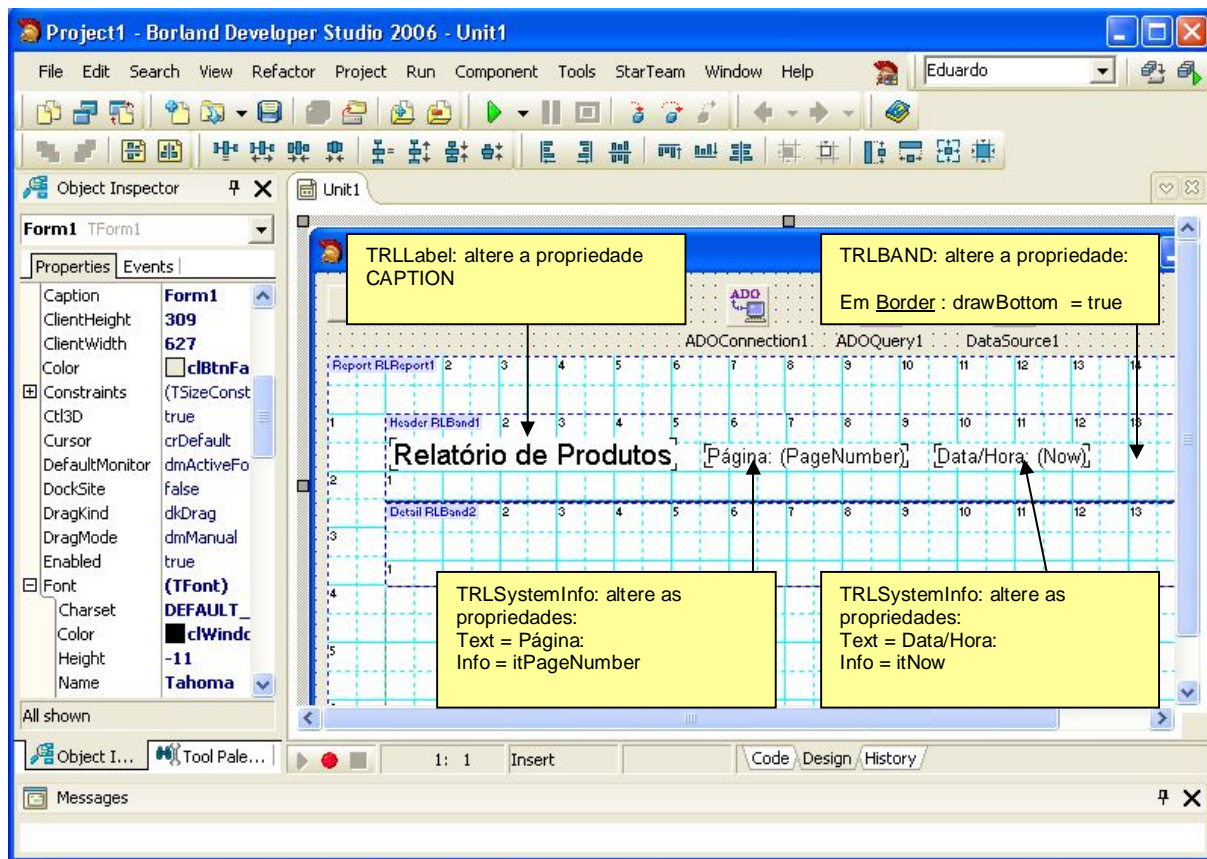
Crie uma conexão com o banco northwind, coloque um objeto TAdoQuery com a instrução sql: *select * from products* . Coloque também um objeto TDataSource e conecte-o à Query. Durante o desenvolvimento do relatório, deixe a query aberta (active=true).



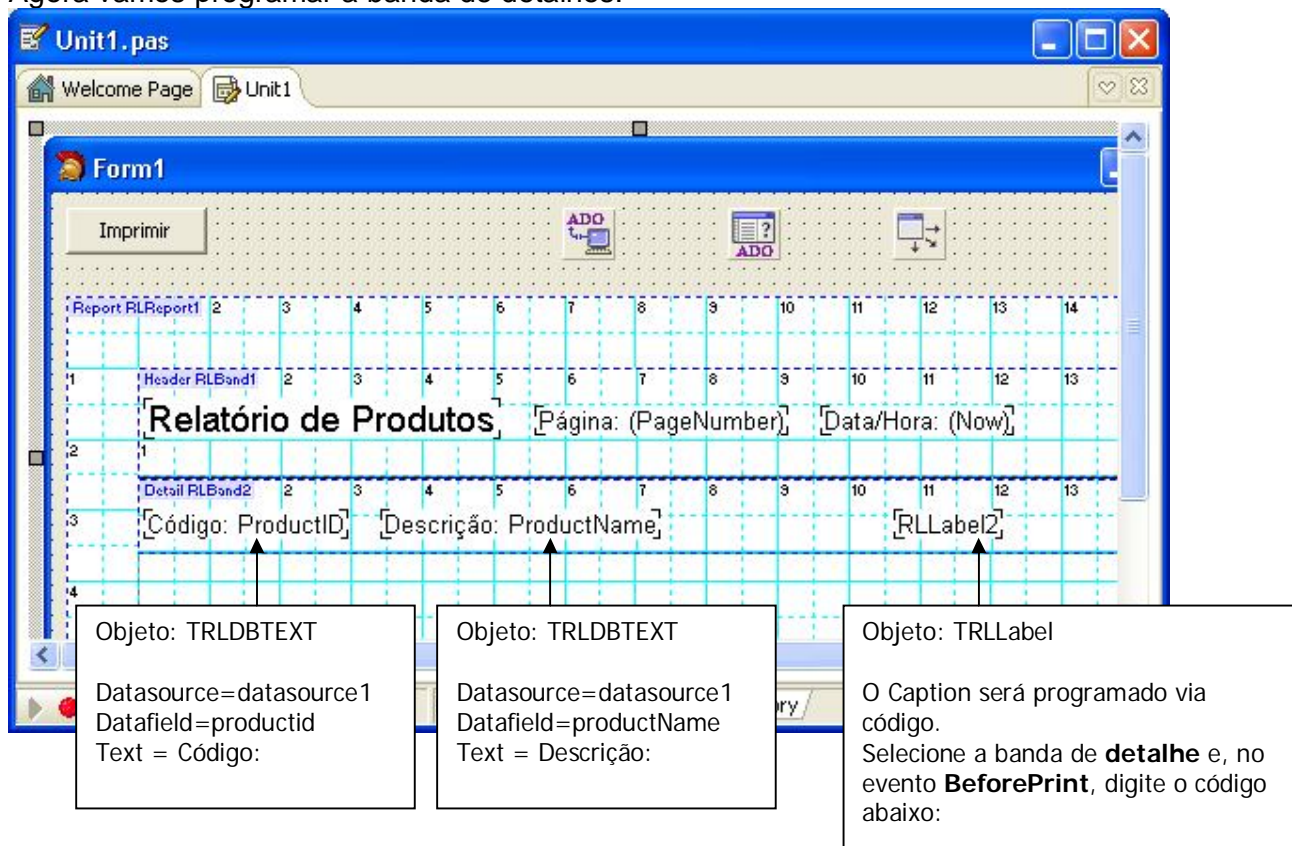
Primeiro, vamos montar o título do relatório:

NÃO UTILIZE LABELS OU OUTROS OBJETOS QUE NÃO SEJAM DA PALHETA FORTES REPORT!!

CERTIFIQUE-SE DE COLOCAR OS OBJETOS DENTRO DAS BANDAS! Para ver se está certo, tente arrastálos para fora da banda; se você conseguir, está errado!



Agora vamos programar a banda de detalhes:



O evento abaixo (BeforePrint) é disparado sempre que a banda é impressa. Como esta é uma banda de detalhe, ela será impressa para cada registro da tabela de produtos. Ela vai calcular o saldo do produto e vai jogar no caption do label.

```
procedure TForm1.RLBand2BeforePrint(Sender: TObject; var PrintIt: Boolean);  
begin  
  RLLabel2.caption :=  
    'Saldo: ' + formatFloat( '##0.00',  
      ADOQuery1.fieldbyname('unitPrice').asFloat *  
      ADOQuery1.fieldbyname('unitsInStock').asFloat);  
end;
```

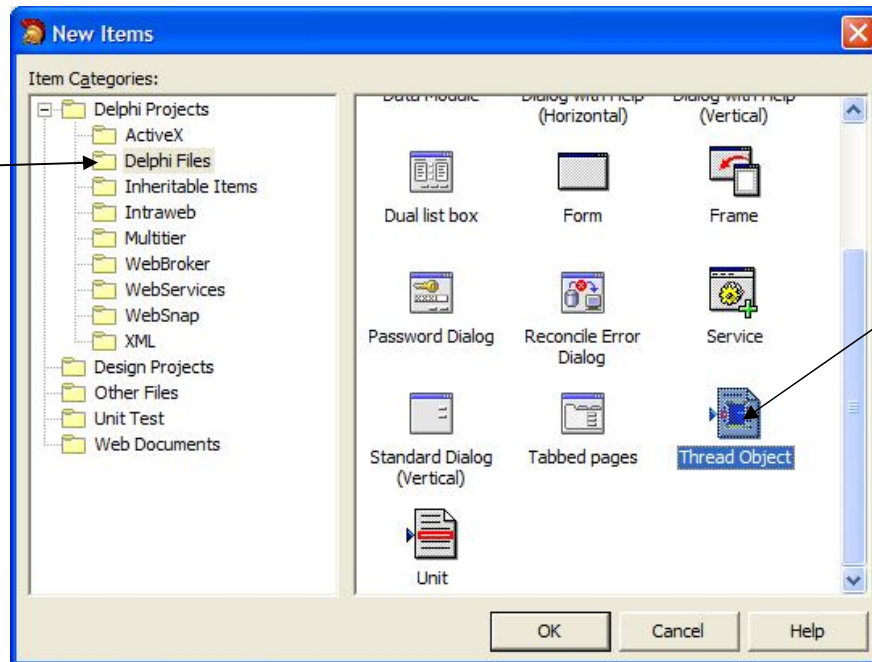
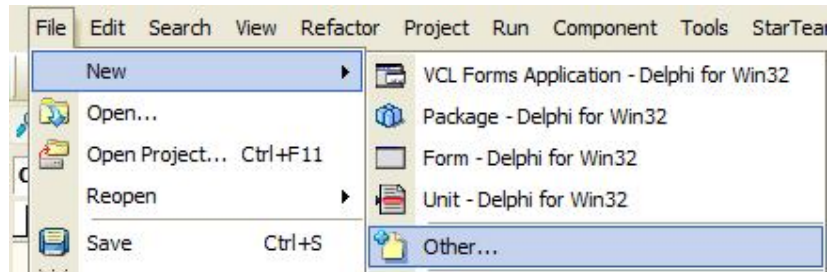
Coloque **visible=false** no objeto RLReport1 para que quando o form for exibido ele não apareça.

No botão, coloque o seguinte código para imprimir o relatório:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  RLReport1.Preview;  
end;
```

Se quiser imprimir diretamente para a impressora, use o método **print** no lugar do **preview**.

CRIANDO THREADS NO DELPHI 2006



Dê um nome para a classe da sua ThRead

A sua nova Unit para a ThRead inicia assim:

Primeira coisa: **Salve-a** como uMoveObjects.

```

unit uMoveObjects;
interface
uses
  Classes;

type
  TMoveObjects = class(TThread)
  private
    { Private declarations }
  protected
    procedure Execute; override;
  end;

implementation

  { TMoveObjects }

procedure TMoveObjects.Execute;
begin
  { Place thread code here }
end;

end.

```

O Objetivo desta ThRead será mover (verticalmente) os objetos da classe TWinControl que nós associarmos a ela.

Programando a ThRead:

Crie a uma variável global, a privada, o objeto e defina a unit controls e forms no uses.

```
unit uMoveObjects;

interface

uses
  Classes, controls, Forms;

type
  TMoveObjects = class(TThread)
  public
    Objeto : Twincontrol;

  private
    { Private declarations }
    direcao : String;
```

Programação do método execute:

```
procedure TMoveObjects.Execute;
begin
  while not terminated do
    Synchronize( MoveObjeto );
end;
```

Crie um nova **procedure MoveObjeto** na sessão private, pois ela só terá visibilidade dentro da ThRead:

```
private
  { Private declarations }
  procedure Moveobjeto;
```

Agora vamos programar a procedure (dica: pressione CTRL+SHIFT+C para criar o código da procedure automaticamente):

```

procedure TMoveObjects.Moveobjeto;
var Formulario : TForm;
begin
    // só funciona com objetos postos diretamente no form
    formulario := (objeto.parent as TForm);

    if direcao = 'Baixo' then
    begin
        objeto.top := objeto.top + 1;
        if objeto.top = formulario.ClientHeight - objeto.Height then
            direcao := 'Cima';
        end
    else begin
        objeto.top := objeto.top - 1;
        if objeto.Top = 0 then
            direcao := 'Baixo';
        end;
    end;
    // atualiza a tela
    Application.ProcessMessages;
end;

```

Código completo da ThRead:

```

unit uMoveObjects;

interface

uses
  Classes, controls, Forms;

type
  TMoveObjects = class(TThread)
  public
    Objeto : Twincontrol;

  private
    { Private declarations }
    direcao : String;

  procedure Moveobjeto;

  protected
    procedure Execute; override;
  end;

implementation

{ TMoveObjects }

procedure TMoveObjects.Execute;
begin
  direcao := 'Baixo';
  while not terminated do
    Synchronize( MoveObjeto );
end;

procedure TMoveObjects.Moveobjeto;
var Formulario : TForm;
begin
  // só funciona com objetos postos diretamente no form
  formulario := (objeto.parent as TForm);

  if direcao = 'Baixo' then
  begin
    objeto.top := objeto.top + 1;
    if objeto.top = formulario.ClientHeight - objeto.Height then
      direcao := 'Cima';
    end
  else begin

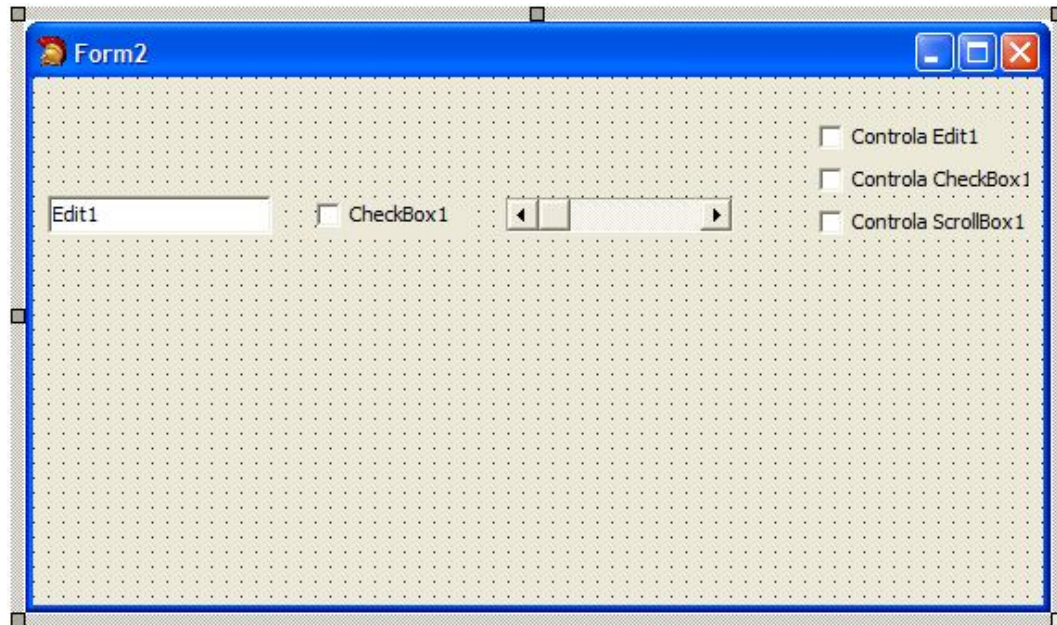
```



```
    objeto.top := objeto.top - 1;  
    if objeto.Top = 0 then  
        direcao := 'Baixo';  
    end;  
    // atualiza a tela  
    Application.ProcessMessages;  
end;  
  
end.
```

Programa principal:

No programa principal, coloque no form os seguintes objetos:




No private, vamos criar 3 objetos do tipo TMoveObjects, que serão as 3 Threads que irão mover os 3 objetos do centro do form. Não esqueça de adicionar no USES a unit Umoveobjects.

```
uses
  Windows, Messages, SysUtils, Variants, Classes,
  Dialogs, StdCtrls, uMoveObjects ;

type
  TForm2 = class(TForm)
    Edit1: TEdit;
    CheckBox1: TCheckBox;
    ScrollBar1: TScrollBar;
    CheckBox2: TCheckBox;
    CheckBox3: TCheckBox;
    CheckBox4: TCheckBox;
    procedure CheckBox2Click(Sender: TObject);
    procedure CheckBox3Click(Sender: TObject);
    procedure CheckBox4Click(Sender: TObject);
  private
    { Private declarations }
    Thread_edit1 : TMoveObjects;
    Thread_checkbox1 : TMoveObjects;
    Thread_ScrollBar1 : TMoveObjects;
```


Programação do evento onclick do componente

 Controla Edit1

```
procedure TForm2.CheckBox2Click(Sender: TObject);
begin
  if (sender as TCheckBox).Checked then begin
    ThRead_edit1 := TMoveObjects.Create(true); // cria pausada
    ThRead_edit1.Objeto := Edit1; // associa o objeto
    ThRead_edit1.FreeOnTerminate := true; // destroi automaticamente
    ThRead_edit1.Priority := tpTimeCritical; // define a prioridade
    ThRead_edit1.Resume;
  end
  else
    ThRead_edit1.Terminate;
end;
```

Programação do evento onclick do componente

 Controla CheckBox1

```
procedure TForm2.CheckBox3Click(Sender: TObject);
begin
  if (sender as TCheckBox).Checked then begin
    ThRead_checkbox1 := TMoveObjects.Create(true); // cria pausada
    ThRead_checkbox1.Objeto := CheckBox1; // associa o objeto
    ThRead_checkbox1.FreeOnTerminate := true; // destroi automaticamente
    ThRead_checkbox1.Priority := tpTimeCritical; // define a prioridade
    ThRead_checkbox1.Resume;
  end
  else
    ThRead_checkbox1.Terminate;
end;
```

Programação do evento onclick do componente

 Controla ScrollBox1

```
procedure TForm2.CheckBox4Click(Sender: TObject);
begin
  if (sender as TCheckBox).Checked then begin
    ThRead_ScrollBar1 := TMoveObjects.Create(true); // cria pausada
    ThRead_ScrollBar1.Objeto := ScrollBar1; // associa o objeto
    ThRead_ScrollBar1.FreeOnTerminate := true; // destroi automaticamente
    ThRead_ScrollBar1.Priority := tpTimeCritical; // define a prioridade
    ThRead_ScrollBar1.Resume;
  end
  else
    ThRead_ScrollBar1.Terminate;
end;
```

Código completo do módulo principal:

```

unit Unit2;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, uMoveObjects ;

type
  TForm2 = class(TForm)
    Edit1: TEdit;
    CheckBox1: TCheckBox;
    ScrollBar1: TScrollBar;
    CheckBox2: TCheckBox;
    CheckBox3: TCheckBox;
    CheckBox4: TCheckBox;
    procedure CheckBox2Click(Sender: TObject);
    procedure CheckBox3Click(Sender: TObject);
    procedure CheckBox4Click(Sender: TObject);
  private
    { Private declarations }
    ThRead_edit1 : TMoveObjects;
    ThRead_checkbox1 : TMoveObjects;
    ThRead_ScrollBar1 : TMoveObjects;
  public
    { Public declarations }
  end;

var
  Form2: TForm2;

implementation

{$R *.dfm}

procedure TForm2.CheckBox2Click(Sender: TObject);
begin
  if (sender as TCheckBox).Checked then begin
    ThRead_edit1 := TMoveObjects.Create(true); // cria pausada
    ThRead_edit1.Objeto := Edit1; // associa o objeto
    ThRead_edit1.FreeOnTerminate := true; // destroi automaticamente
    ThRead_edit1.Priority := tpTimeCritical; // define a prioridade
    ThRead_edit1.Resume;
  end
  else
    ThRead_edit1.Terminate;
end;

```

end;

procedure TForm2.CheckBox3Click(Sender: TObject);

begin

if (sender as TCheckBox).Checked then begin

 ThRead_checkbox1 := TMoveObjects.Create(true); // cria pausada

 ThRead_checkbox1.Objeto := CheckBox1; // associa o objeto

 ThRead_checkbox1.FreeOnTerminate := true; // destroi automaticamente

 ThRead_checkbox1.Priority := tpTimeCritical; // define a prioridade

 ThRead_checkbox1.Resume;

end

else

 ThRead_checkbox1.Terminate;

end;

procedure TForm2.CheckBox4Click(Sender: TObject);

begin

if (sender as TCheckBox).Checked then begin

 ThRead_ScrollBar1 := TMoveObjects.Create(true); // cria pausada

 ThRead_ScrollBar1.Objeto := ScrollBar1; // associa o objeto

 ThRead_ScrollBar1.FreeOnTerminate := true; // destroi automaticamente

 ThRead_ScrollBar1.Priority := tpTimeCritical; // define a prioridade

 ThRead_ScrollBar1.Resume;

end

else

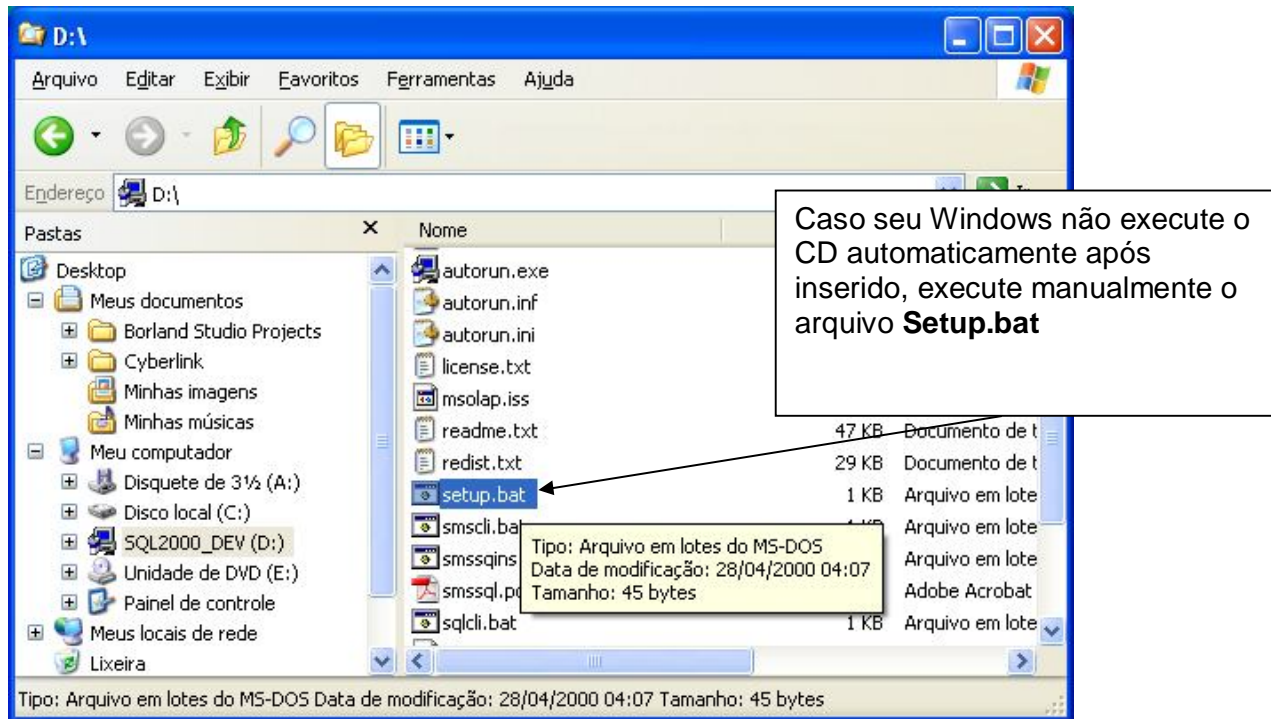
 ThRead_ScrollBar1.Terminate;

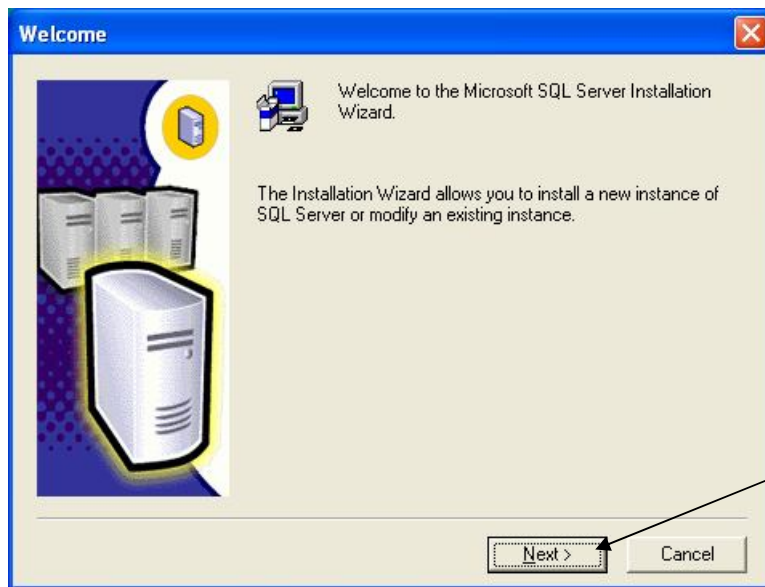
end;

end.

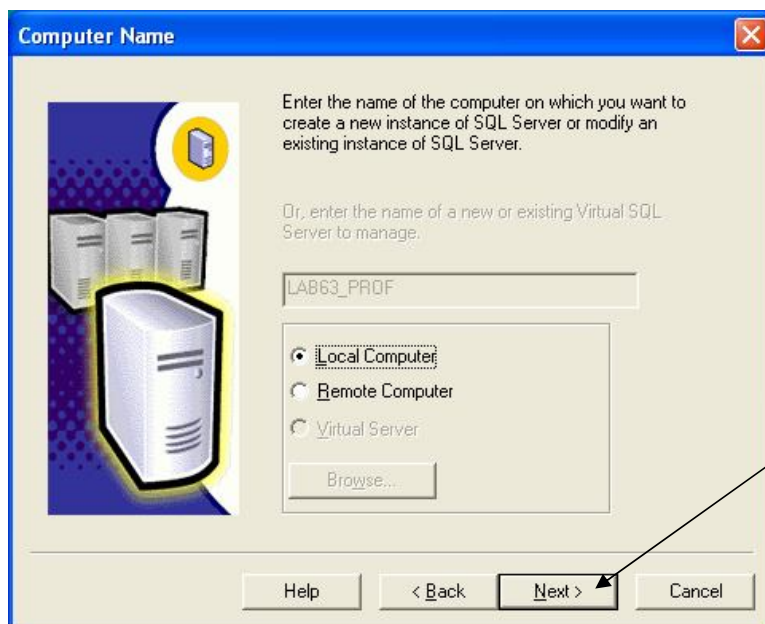
INSTALAÇÃO DO SQL SERVER 2000 – VERSÃO PARA DESKTOP

Verifique as opções das telas antes de prosseguir para a próxima tela.

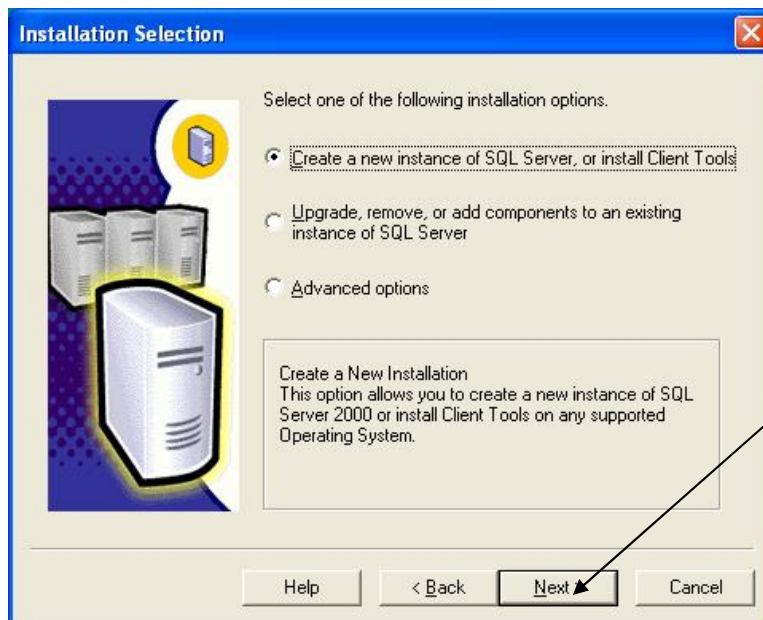




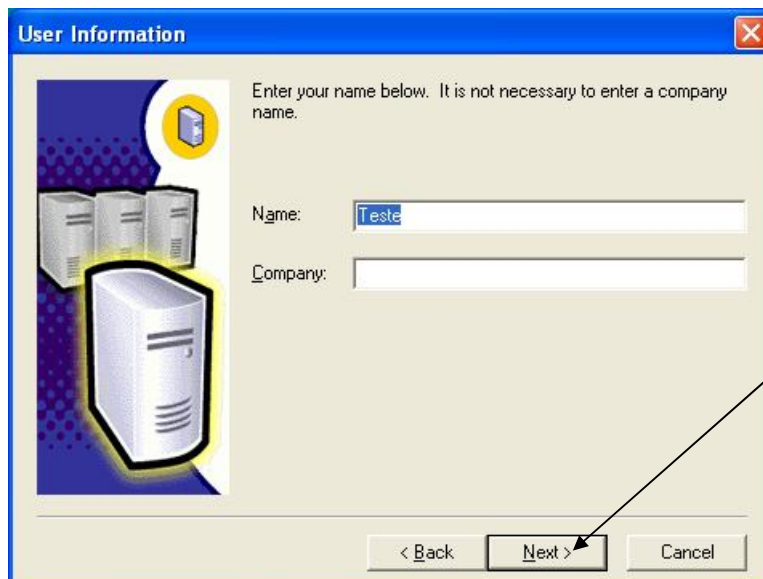
Clique em Next



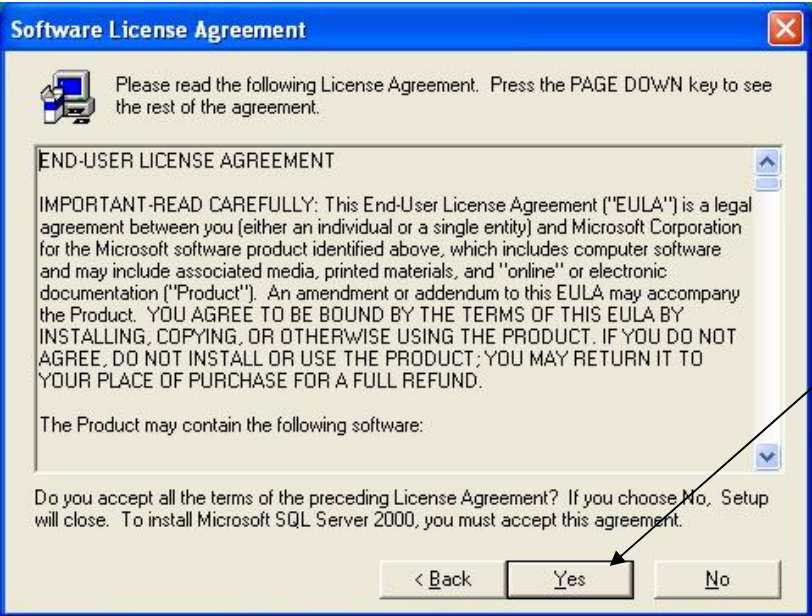
Clique em Next



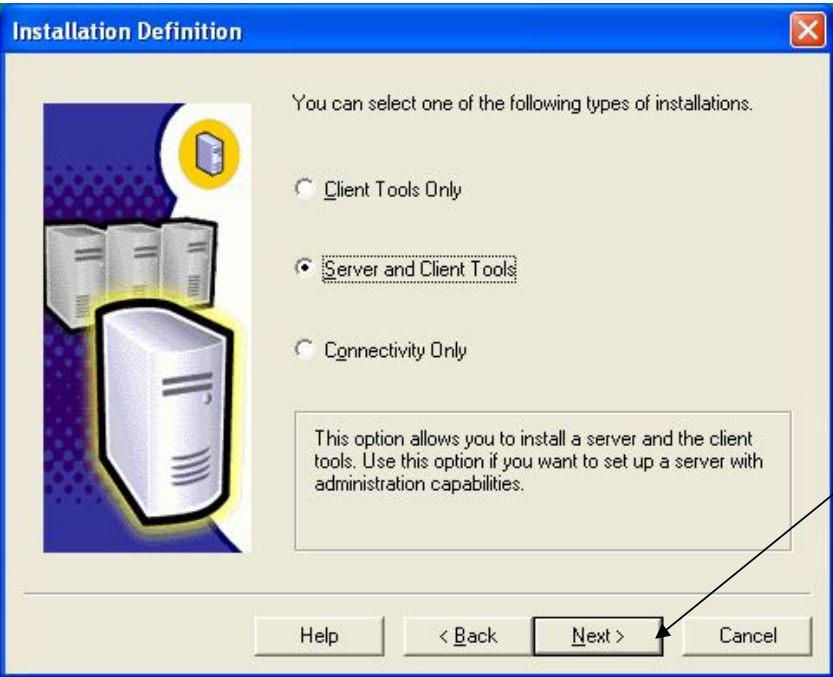
Clique em Next



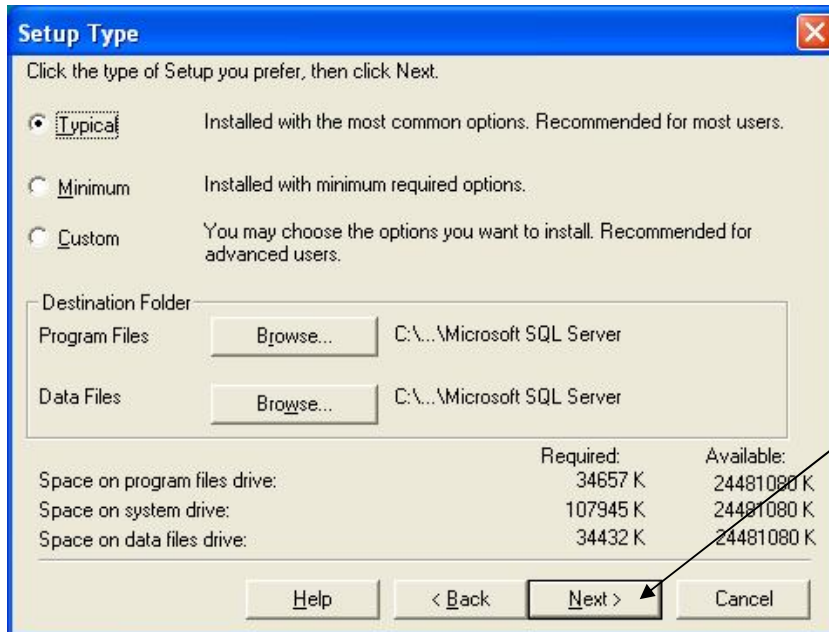
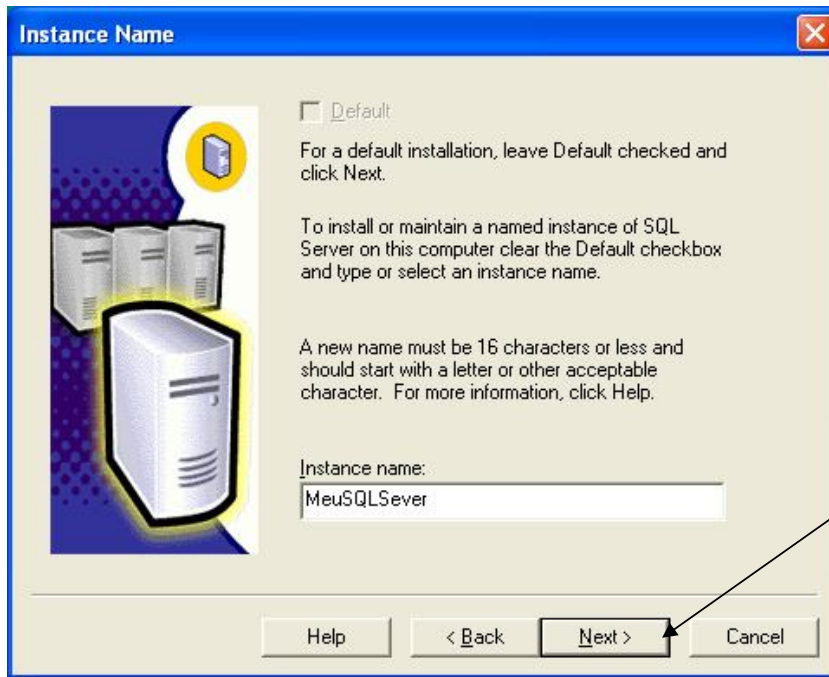
Clique em Next



Clique aqui



Clique aqui



Services Accounts

☒ Use the same account for each service. Auto start SQL Server Service.
☐ Customize the settings for each service.

Services

- ☐ SQL Server
- ☐ SQL Server Agent

Service Settings

☒ Use the Local System account
☐ Use a Domain User account

Username:
Password:
Domain:

☐ Auto Start Service

Help < Back **Next >** Cancel

Verifique se os parâmetros estão desta forma.

Não é necessário informar username, password e domain.

Clique aqui.

Authentication Mode

Choose the authentication mode.

☐ Windows Authentication Mode
☒ Mixed Mode (Windows Authentication and SQL Server Authentication)

Add password for the sa login:

Enter password:
Confirm password:

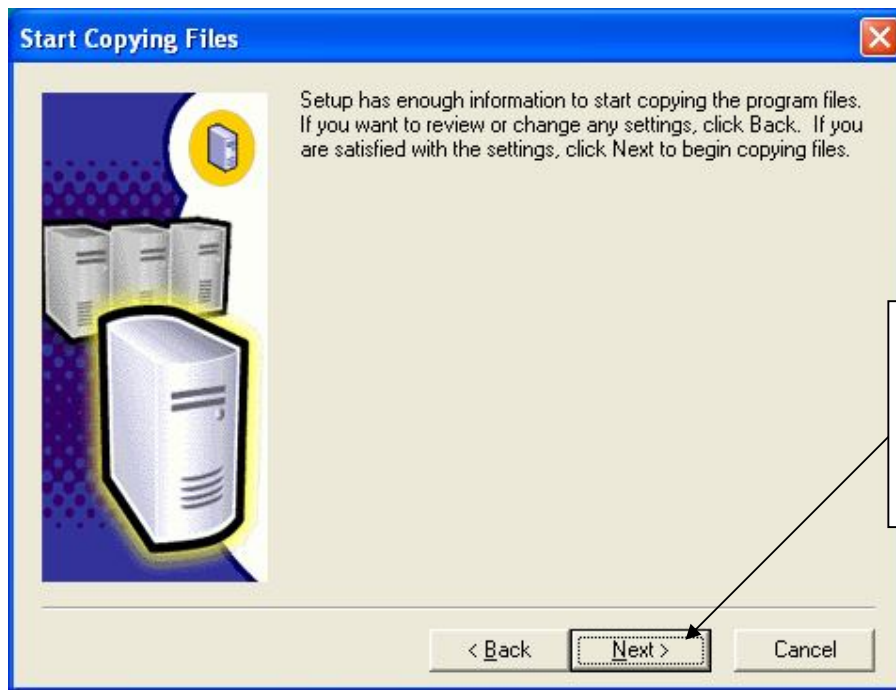
☐ Blank Password (not recommended)

Help < Back **Next >** Cancel


No password, recomendo usar 123 nos 2 campos.

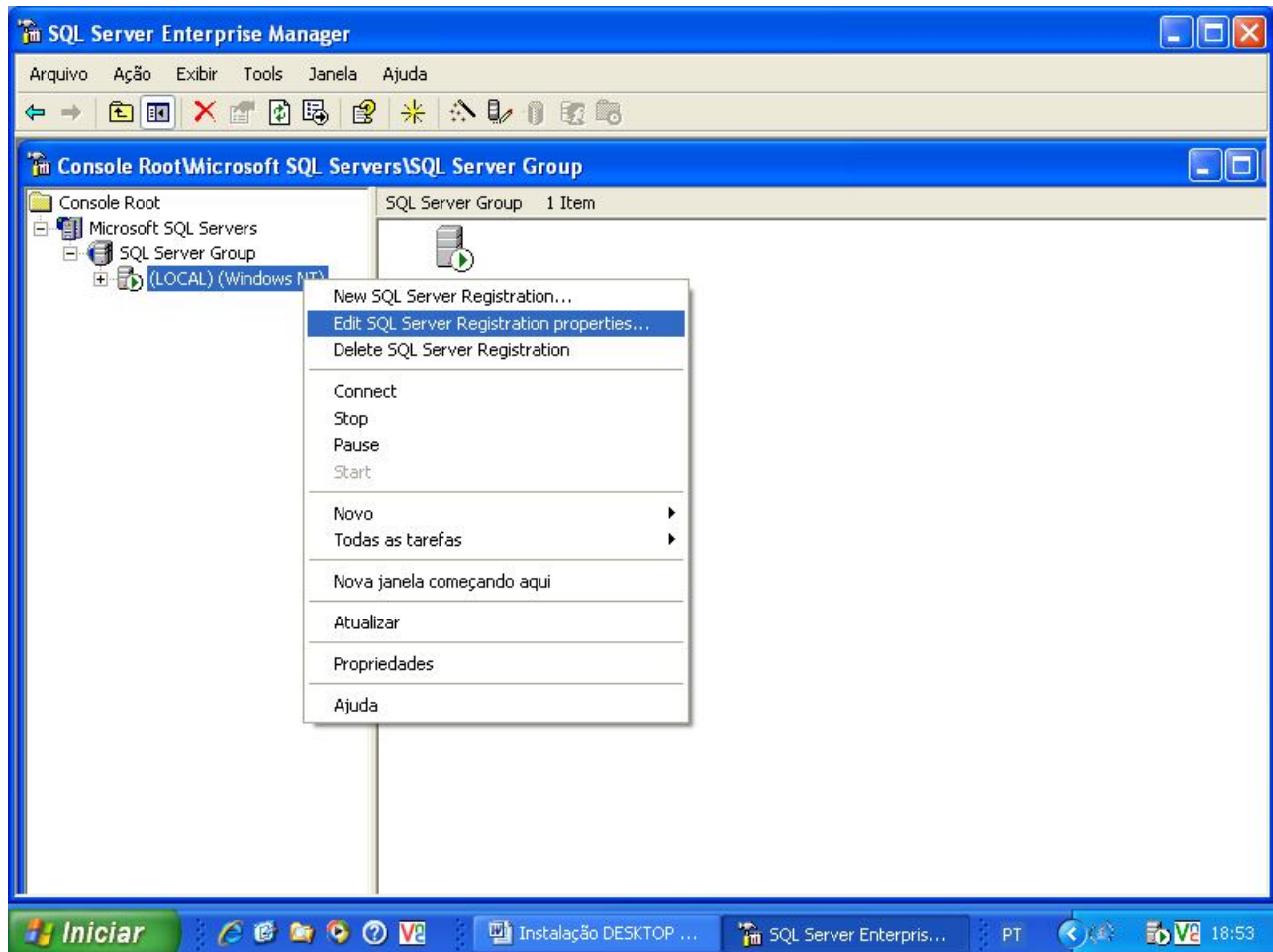
Verifique se os parâmetros estão desta forma.

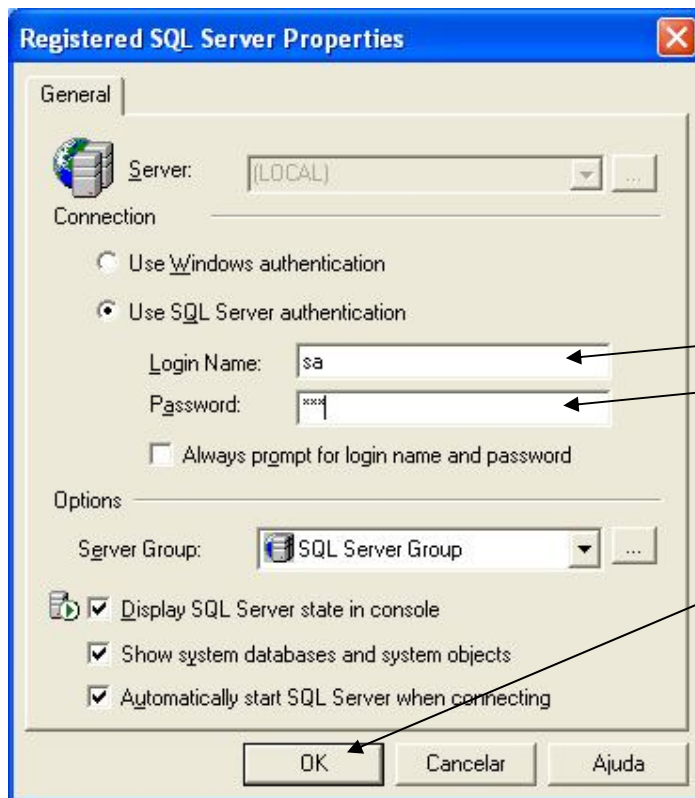
Clique aqui.



Clique aqui e aguarde a instalação.

Vá ao menu de instalação do SQL SERVER e Execute o  Enterprise Manager . Como na imagem abaixo, selecione a mesma opção.





Selecione as mesmas opções que a tela ao lado.

No login name coloque: **sa**
E no password coloque: **123**

Depois clique em OK

PRONTO!

APÊNDICE - FUNÇÕES PARA TRABALHAR COM STRINGS (RETIRADAS DO HELP DO DELPHI 7)

AdjustLineBreaks function : Adjusts line break characters to reflect Windows or Linux style.

AnsiCompareStr function : Compares strings based on the current locale with case sensitivity.

AnsiCompareText function : Compares strings based on the current locale without case sensitivity.

AnsiContainsStr function: Indicates whether one string is a (case-sensitive) substring of another.

AnsiContainsText function : Indicates whether one string is a (case-insensitive) substring of another.

AnsiDequotedStr function : Converts a quoted string into an unquoted string

AnsiEndsStr function : Indicates whether one string is a (case-sensitive) suffix of another.

AnsiEndsText function : Indicates whether one string is a (case-insensitive) suffix of another.

AnsiExtractQuotedStr function : Converts a quoted string into an unquoted string.

AnsiIndexStr function : Provides the index of a specified string in an array of strings.

AnsiIndexText function : Provides the index of a specified string in an array of strings.

AnsiLeftStr function : Returns the substring of a specified length that appears at the start of a string.

AnsiLowerCase function : Returns a string that is a copy of the given string converted to lower case.

AnsiMatchStr function : Indicates whether an array of strings contains an exact match to a specified string.

AnsiMatchText function : Indicates whether an array of strings contains a case-insensitive match to a specified string.

AnsiMidStr function : Returns the substring of a specified length that appears at a specified position in a string.

AnsiPos function : Locates the position of a substring.

AnsiQuotedStr function : Returns the quoted version of a string.

AnsiReplaceStr function : Replaces all occurrences of a substring with another string.

AnsiReplaceText function : Replaces all case-insensitive matches of a substring with another string.

AnsiResemblesProc variable : Controls the algorithm used by AnsiResemblesText to determine when two strings are similar.

AnsiResemblesText function : Indicates whether two strings are similar.

AnsiReverseString function : Returns reversed string.

AnsiRightStr function : Returns the substring of a specified length that appears at the end of a string.

AnsiSameStr function : Compares strings based on the current locale with case sensitivity.

AnsiSameText function : Compares strings based on the current locale without case sensitivity.

AnsiStartsStr function : Indicates whether one string is a (case-sensitive) prefix of another.

AnsiStartsText function : Indicates whether one string is a (case-insensitive) prefix of another.

AnsiUpperCase function : Converts a string to upper case.

CompareStr function : Compares two strings case sensitively.

CompareText function : Compares two strings by ordinal value without case sensitivity.

Concat function : Concatenates two or more strings into one.

Copy function : Returns a substring of a string or a segment of a dynamic array.

DecodeSoundExInt function : Converts an integer representation of a SoundEx encoding into the corresponding phonetic string.

DecodeSoundExWord function : Converts a Word representation of a SoundEx encoding into the corresponding phonetic string.

Delete procedure : Removes a substring from a string.

DupeString function : Returns the concatenation of a string with itself a specified number of repeats.

Insert procedure : Inserts a substring into a string beginning at a specified point.

IsDelimiter function : Indicates whether a specified character in a string matches one of a set of delimiters.

LastDelimiter function : Returns the byte index of the last character that matches any character in a specified set of delimiters.

LeftBStr function : Returns the substring of a specified number of bytes that appears at the start of a string.

LeftStr function : Returns the substring of a specified length that appears at the start of a string.

Length function : Returns the number of characters in a string or elements in an array.

LowerCase function : Converts an ASCII string to lowercase.

MidBStr function : Returns the substring of a specified number of bytes that appears at a specified position in a string.

MidStr function : Returns the substring of a specified length that appears at a specified position in a string.

NullStr constant : Declares a pointer to an empty string.

Pos function : Returns the index value of the first character in a specified substring that occurs in a given string.

PosEx function : Returns the index value of a substring.

QuotedStr function : Returns the quoted version of a string.

ReverseString function : Returns the reverse of a specified string.

RightBStr function : Returns the substring of a specified number of bytes that appears at the end of a string.

RightStr function : Returns the substring of a specified length that appears at the end of a string.

SameText function : Compares two strings by ordinal value without case sensitivity.

SetLength procedure : Sets the length of a string or dynamic-array variable.

SetString procedure : Sets the contents and length of the given string.

SoundEx function : Converts a string into its SoundEx representation.

SoundExCompare function : Compares the SoundEx representations of two strings.

SoundExInt function : Converts a string into an integer that represents its phonetic value.

SoundExProc function : Indicates whether two strings are similar.

SoundExSimilar function : Indicates whether two strings are similar.

SoundExWord function : Converts a string into a Word that represents its phonetic value.

Str procedure : Formats a string and returns it to a variable.

StringOfChar function : Returns a string with a specified number of repeating characters.

StringReplace function : Returns a string with occurrences of one substring replaced by another substring.

StuffString function : Inserts a substring into a specified position of a string, replacing the current characters.

Trim function : Trims leading and trailing spaces and control characters from a string.

TrimLeft function : Trims leading spaces and control characters from a string.

TrimRight function : Trims trailing spaces and control characters from a string.

UpperCase function : Returns a copy of a string in uppercase.

Val procedure : Converts a string to a numeric representation.

WideLowerCase function : Returns Unicode string converted to lower case.

WideSameStr function : Compares Unicode strings based on the current locale with case sensitivity.

WideSameText function : Compares Unicode strings based on the current locale without case sensitivity.

WideUpperCase function : Returns Unicode string converted to upper case.

WrapText function : Splits a string into multiple lines as its length approaches a specified size.

APÊNDICE - OBJECT PASCAL: ESTRUTURAS BÁSICAS

Este apêndice foi retirado da apostila de Paulo Roberto Alves Pereira, 2002

Palavras reservadas

As palavras reservadas independente de qualquer biblioteca, fazem parte do núcleo da linguagem, e por isso são utilizadas por qualquer compilador Delphi. Normalmente elas estão em evidência no programa (negrito).

Palavra	Descrição
Begin	Início de bloco de programa.
End	Fim de Bloco de Programa.
Type	Definição de tipos.
Var	Definição de variáveis.
Uses	Definição de bibliotecas.
Class	Definição de classes.
Implementation	Seção da unidade, onde estão todas as definições da mesma, variáveis, tipos, constantes, bibliotecas, etc., que são privadas à unidade. Além disso está todo o código dos métodos da unidade.
Interface	Seção da unidade, onde estão todas as definições da mesma, variáveis, tipos, constantes, bibliotecas, etc., que serão tornadas públicas pela unidade.
Do	Comando interno de execução de algum outro comando.
while	Comando de repetição.
for	Comando de repetição.
if	Comando de seleção.
else	Comando de exceção a uma seleção.
Case	Comando de seleção.
Then	Comando de auxiliar de seleção.
Public	Propriedades e Métodos que podem ser acessados por qualquer unidade do programa.
Private	Propriedades e Métodos que só podem ser acessados pela própria unidade.
Protected	Propriedades e Métodos que podem ser acessados apenas pela própria unidade e pelos métodos das unidades derivadas desta.
Unit	Chave para a declaração do nome da unidade.
Array	Palavra para a definição de vetores.
Of	Comando auxiliar para definições de tipos.
Repeat	Comando de Repetição.
Until	Comando auxiliar na repetição.
With	Comando utilizado para referir-se a um registro de um objeto de forma direta.

Tipos de Dados

Tipos Inteiros

Tipo	Domínio	Tamanho
------	---------	---------

Shortint	-128 à 127	1
Smallint	-32768 à 32767	2
Longint	-2147483648 à 2147483647	4
Byte	0 à 255	1
Word	0 à 65535	2
Integer	-32768 à 32767	2
Cardinal	0 à 65535	2
Integer	-2147483648 à 2147483647	4 (32-bits)
Cardinal	0 à 2147483647	4 (32-bits)

Tipos Booleanos

Tipo	Descrição
Boolean	1 byte, assume apenas valores TRUE ou FALSE
ByteBool	É um byte com características booleanas, 0(FALSE) e diferente de 0 (TRUE)
WordBool	É um Word com características booleanas, 0(FALSE) e diferente de 0 (TRUE)
LongBool	É um Longint com características booleanas, 0(FALSE) e diferente de 0 (TRUE)

Tipos Caracteres

Tipo	Descrição
Char	1 byte, caracter
String	n bytes, caracteres

Tipos Reais

Tipo	Domínio	Alg. Sig.	Tam.
Real	2.9×10^{-39} à 1.7×10^{38}	11-12	6
Single	1.5×10^{-45} à 3.4×10^{38}	7-8	4
Double	5.0×10^{-324} à 1.7×10^{308}	15-16	8
Extended	3.4×10^{-4932} à 1.1×10^{4932}	19-20	10
Comp	-263+1 à 263 -1	19-20	8
Currency	-922337203685477.5808 à 922337203685477.5807	19-20	8

Definição de Arrays

Um array é um vetor de dados de um determinado tipo. Ele representa uma lista com uma determinada característica, e é definido no Delphi da mesma forma como era definido no Pascal.

Var

Nome_da_variável: **array**[1..n] **of** Tipo_da_variável; // ou

Nome_da_variável: **array**[1..n,1..m,1..x,..] **of** Tipo_da_variável; // Para matrizes

Ex.:

X: **array**[1..10] **of** Integer; // Vetor de 10 elementos inteiros

S: **array**[1..50] **of** Double; // Vetor de 50 elementos reais

M: **array**[1..10,1..20] **of** Boolean; // Matriz booleana de 10x20

Comentários

Comentário de Uma linha: // comentário

Comentários de várias linhas { comentário }

Formas de Atribuição

O Delphi utiliza a mesma forma de atribuição do Pascal (:=). A única diferença é que a atribuição foi estendida aos objetos e aos novos tipos do Delphi.

Ex.:

X := 10 + Y;

Form1 := Form2; Operadores:

O Delphi Possui uma estrutura de operadores muito parecida com a do pascal, apenas com a inclusão de alguns novos operadores.

Operadores

Operadores Aritméticos

Operador	Operação	Tipos Usados	Tipos Resultantes
+	Adição	Inteiros Reais	Inteiro Real
-	Subtração	Inteiros Reais	Inteiro Real
*	Multiplicação	Inteiros Reais	Inteiro Real
/	Divisão	Inteiros Reais	Real Real
Div	Divisão inteira	Inteiros	Inteiro
Mod	Resto da divisão inteira	Inteiros	Inteiro

Operadores Unários

Operador	Operação	Tipos Usados	Tipos Resultantes
+	Identidade do sinal	Inteiros Reais	Inteiro Real
-	Negação de sinal	Inteiros Reais	Inteiro Real

Operadores Lógicos (Bit a Bit)

Operador	Operação	Tipos Usados	Tipos Resultantes
Not	Negação	Inteiros	Booleano
And	E	Inteiros	Booleano
Or	OU	Inteiros	Booleano
Xor	OU Coincidente (Exclusivo)	Inteiros	Booleano
Shl	Shift para a	Inteiros	Booleano

	esquerda		
Shr	Shift para a direita	Inteiros	Booleano

Operadores Booleanos

Operador	Operação	Tipos Usados	Tipos Resultantes
Not	Negação	Booleanos	Booleano
And	E	Booleanos	Booleano
Or	OU	Booleanos	Booleano
Xor	OU coincidente	Booleanos	Booleano

Operadores Relacionais

Operador	Operação	Tipos Usados	Tipos Resultantes
=	Igual	Tipos simples compatíveis, de classe, referencia de classe, ponteiros, conjunto, string ou string empacotado.	Booleano

Operador	Operação	Tipos Usados	Tipos Resultantes
<>	Diferente de	Tipos simples compatíveis, de classe, referencia de classe, ponteiros, conjunto, string ou string empacotado.	Booleano
<	Menor que	Tipos simples compatíveis, strings ou string empacotado ou Pchar	Booleano
>	Maior que	Tipos simples compatíveis, strings ou string empacotado ou Pchar	Booleano
<=	Menor ou igual	Tipos simples compatíveis, strings ou string empacotado ou Pchar	Booleano
>=	Maior ou igual	Tipos simples compatíveis, strings ou string empacotado ou Pchar	Booleano
<=	Subconjunto de	Tipos de conjuntos compatíveis	Booleano
>=	Superconjunto de	Tipos de conjuntos compatíveis	Booleano
In	Membro de	Operando da esquerda, qualquer tipo ordinal; Operando da direita, conjunto cuja base seja compatível com o operando da esquerda	Booleano

Is	Compatível a	Tipos de conjuntos, mais especificamente classes	Booleano
----	-----------------	---	----------

Operadores Especiais

Operador	Operação	Tipos Usados	Tipos Resultantes
@	Atribuição	Qualquer	Ponteiro
As	Relação	Classes	Classe

Precedência dos Operadores

Precedência	Operadores
Primeiro	@, not, -(unário)
Segundo	*,/, div, mod, and, shl, shr, as
Terceiro	+, -, or, xor
Quarto	=, <, >, <=, >=, in, is

Comandos de seleção

If Then

O if é a estrutura de seleção mais simples que existe, e está presente em todas as linguagens de programação. Nele, uma condição é testada, se for verdadeira ira executar um conjunto de comandos, se for falsa, poderá ou não executar um outro conjunto de comandos. A sua estrutura básica é:

```

If condição Then
  Begin
    Comandos executados se a condição for verdadeira;
  End
Else
  Comandos serem executados se a condição for falsa;

```

Ex.:

<pre> If x=0 Then Write('x é zero'); </pre>	<pre> If x=0 Then Write('x é zero'); Else Write('x não é zero'); </pre>
---	---

A utilização dos marcadores de início e fim (Begin e End), e considerada opcional, se for executado apenas um único comando. O If ainda permite o encadeamento de If's.

Ex.:

```

If x=0 Then
  Write('X é zero');
Else
  If x>0 Then
    Write('X é positivo');
  Else

```

Write('X é negativo');

Case

A instrução case consiste em uma expressão usada para selecionar um valor em uma lista de possíveis valores, ou de faixa de valores. Estes valores são constantes e devem ser únicos e de tipo ordinal. Finalmente pode haver uma instrução else que será executada se nenhum dos rótulos corresponder ao valor do seletor. O seu formato básico é:

```

Case Seletor of
  Const1: Begin
    Comandos referentes a constante 1.
  End;
  Const2: Begin
    Comandos referentes a constante 2.
  End;
  Faixa1..Faixa2: Begin
    Comandos referentes a Faixa de valores.
  End;
Else
  Comandos referentes ao else
End;

```

Ex.:

<pre> Case Numero of 1: texto := 'um'; 2: texto := 'dois'; 3: texto := 'três'; end; </pre>	<pre> Case MeuChar of '+': Texto := 'Sinal de mais'; '-': Texto := 'Sinal de Menos'; '0'..'9': Texto := 'Números'; else Begin Texto := 'Caracter desconhecido'; Meuchar := '?'; End; </pre>
--	---

A utilização dos marcadores de início e fim (Begin e End), e considerada opcional, se o caso for executar apenas um único comando. O case ainda pode ser encadeado, ou seja um case dentro do outro.

Comandos de repetição

For

O loop for no Pascal baseia-se estritamente em num contador, o qual pode ser aumentado ou diminuído cada vez que o loop for executado. O contador é inicializado, e o loop irá se repetir enquanto o contador não chegar ao fim da contagem. O seu formato básico é:

```

For contador := inicio_da_contagem to fim_da_contagem do
  Begin
    Comandos;
  End;

```

<p>Ex. aumentando:</p> <pre> K := 0; For i:=1 to 10 do </pre>	<p>Ex. diminuindo:</p> <pre> K := 0; For i:=10 downto 1 do </pre>
---	---

K := K + i;	K := K + i;
-------------	-------------

A utilização dos marcadores de início e fim do loop (Begin e End), e considerada opcional, se o loop for executar apenas um único comando. O for ainda pode ser encadeado, ou seja um for dentro do outro:

Ex.:

```

K := 0;
W := 0;
For i:=1 to 10 do
  For j:=1 to 10 do
    K := K + i * j;
```

While

O while é um comando de repetição que não possui controle de um contador e que testa a condição antes de executar o próximo loop. Este loop irá se repetir enquanto a condição for verdadeira. A sua forma básica é:

```

While condição
  Begin
    Comandos;
  End;
```

Ex.:

```

I:=10;
J:=0;
While I>J do
  Begin
    I := I - 1;
    J := J + 1;
  End;
```

No caso do while, as regras do for para os marcadores de início e fim do loop e a do encadeamento, também são válidas.

Repeat Until

O Repeat é um comando de repetição que não possui controle de um contador e que testa a condição depois de executar o loop. Este loop irá se repetir até que a condição seja verdadeira. A sua forma básica é:

```

Repeat
  Comandos;
Until condição
```

Ex.:

```

I:=10;
J:=0;
Repeat
  I := I - 1;
  J := J + 1;
Until J>I;
```

No caso do Repeat, as regras do for para os marcadores de início e fim do loop e a do encadeamento, também são válidas.

Procedimentos e funções

Um conceito importante que integra linguagens como o Pascal, e consequentemente o Object Pascal do Delphi, é o da sub-rotina. No Pascal as subrotinas podem assumir duas formas diferentes: procedimentos ou funções. A única diferença real entre as duas, é que as funções têm um valor de retorno, enquanto os procedimentos não. Abaixo está a sintaxe do Pascal para procedimentos e funções:

<pre> Procedure ola; Begin ShowMessage('Olá !'); End;</pre>	<pre> Function Dois: Integer; Begin Result := 2; End;</pre>
--	---

Os procedimentos e funções podem ainda ter parâmetros de entrada de qualquer tipo definido.

<pre> Procedure ImprimeMensagem (Mens: string); Begin ShowMessage(Mens); End;</pre>	<pre> Function Duplo (valor: Integer): Integer; Begin Result := valor * 2; End;</pre>
---	---

Estes parâmetros de entrada, podem ainda ser de passagem por valor, como o visto nos exemplos acima, ou de passagem por referência, como o visto abaixo:

<pre> Procedure ImpMens (Var Mens: string); Begin ShowMessage(Mens); End;</pre>	<pre> Function Duplo (Var valor: Integer): Integer; Begin Duplo := valor * 2; Valor := 0; End;</pre>
---	--

Os parâmetros também podem ter valores default, que podem ser omitidos na chamada da função/procedure. No exemplo abaixo, o parâmetro Mens não é obrigatório e se este for omitido a mensagem contendo 'TESTE' será exibida. Após declarar um parâmetro default, todos os parâmetros seguintes também devem ter um valor default.

<pre> Procedure ImpMens (Mens: string='TESTE'); Begin ShowMessage(Mens); End;</pre>	<pre> Ex: ImpMens('Timão EÔ'); // 'Timão EO' ImpMens; // 'TESTE'</pre>
---	--

APÊNDICE - RELAÇÕES ENTRE CLASSES

Uma classe pode acessar métodos de outra classe, desde que estes sejam do tipo público, colocando-se o nome completo, ou seja, o nome da classe ponto o nome do método (MinhaClasse.Meumetodo).

Herança

A herança é um dos recursos mais poderosos de uma linguagem orientada a objetos. Ela permite que as classes secundárias assumam as propriedades de suas classes principais. Elas herdam propriedades, métodos e eventos de sua classe principal, a classe secundária pode acrescentar novos componentes naqueles que herdam. Isso permite que você pegue uma classe que tenha praticamente todas as partes fundamentais de que precisa, e insira novos objetos que a personalizam exatamente de acordo com suas necessidades.

Quando uma classe secundária herda propriedades de uma primária, a classe secundária terá acesso a todas as propriedades, métodos e eventos, que forem públicos ou protegidos.

Inherit

Esta é uma palavra reservada do Object Pascal que é utilizada antes de um método. Ela pode ser utilizada quando uma classe secundária, possui um método com o mesmo nome de um da classe primária. Neste caso se nos quisermos acessar o método da classe primária teremos que utilizar o Inherit antes deste método.

Override

Esta é uma palavra reservada do Object pascal que é utilizada após um método quando em uma classe secundária, queremos redefinir uma função definida em uma classe primária. Baseado nas definições de override e das ligações dinâmicas e que se define o poliformismo.

Poliformismo

O poliformismo é um recurso das linguagens orientadas a objetos, que literalmente indica a capacidade de um objeto assumir várias formas. Em outras palavras, o poliformismo permite que você referencie propriedades e métodos de classes diferentes por meio de um mesmo objeto. Ele também possibilita que você execute operações com esse objeto de diferentes maneiras, de acordo com o tipo de dado e com a classe atualmente associada a esse objeto.

Por exemplo, posso declarar um objeto genérico da classe caixa (digamos Minhacaixa) e então associar a ele objetos da classe Tcaixadeferramentas ou Tcaixadedinheiro. Agora suponhamos que cada classe tenha um procedimento Open, cada um deles com uma implementação diferente. Esse método deve usar a ligação dinâmica, isso significa declará-la na classe-pai como virtual ou dynamic e redefini-la como override na classe-filha.

Quando você aplica o método Open a MinhaCaixa, o que acontece? É chamado o procedimento Open do tipo atual do objeto. Se Minhacaixa for atualmente um objeto Tcaixadedinheiro, a caixa será aberta mediante uma senha secreta, se for Tcaixadeferramentas a caixa será simplesmente aberta.

APÊNDICE - DESCRIÇÃO DOS PRINCIPAIS PROCEDIMENTOS E FUNÇÕES PRÉ DEFINIDAS

procedure Beep;

Toca um beep

procedure ChDir(S: string);

Troca o diretório corrente para o diretório especificado em S.

Ex.:

```
begin
  {$I-}
  { Muda para o diretório especificado em Edit1 }
  ChDir(Edit1.Text);
  if IOResult <> 0 then
    MessageDlg('Diretório não encontrado', mtWarning, [mbOk], 0);
end;
```

function Chr(X: Byte): Char;

Retorna o caracter com o código ASCII X

Ex.:

```
begin
  Canvas.TextOut(10, 10, Chr(65)); { A letra 'A' }
end;
```

function Concat(s1 [, s2,..., sn]: string): string;

Concatena as strings

Ex.:

```
var
  S: string;
begin
  S := Concat('ABC', 'DEF'); { 'ABCDE' }
end;
```

function Copy(S: string; Index, Count: Integer): string;

Retorna uma substring de S, começando a partir de Index e tendo Count caracteres

Ex.:

```
var S: string;
begin
  S := 'ABCDEF';
  S := Copy(S, 2, 3); { 'BCD' }
end;
```

function CreateDir(const Dir: string): Boolean;

Cria um novo diretório e retorna o sucesso da operação

function Date: TDateTime;

Retorna a Data atual

Ex.:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Label1.Caption := 'Hoje é ' + DateToStr(Date);
end;
```

function DateToStr(Date: TDateTime): string;

Converte Data para String

Ex.:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Label1.Caption := DateToStr(Date);
end;
```

function DayOfWeek(Date: TDateTime): Integer;

Retorna o dia da semana especificado entre 1 e 7, onde domingo é um e Sábado é 7

Ex.:

```
procedure TForm1.Button1Click(Sender: TObject);
var
    ADate: TDateTime;
begin
    ADate := StrToDate(Edit1.Text);
    Label1.Caption := IntToStr(DayOfWeek(ADate)) + 'º dia da semana';
end;
```

procedure DecodeDate(Date: TDateTime; var Year, Month, Day: Word);

Quebra os valores especificados no parâmetro Date em Year, Month e Day.

Ex.:

```
procedure TForm1.Button1Click(Sender: TObject);
var
    Present: TDateTime;
    Year, Month, Day, Hour, Min, Sec, MSec: Word;
begin
    Present := Now;
    DecodeDate(Present, Year, Month, Day);
    Label1.Caption := 'Hoje é ' + IntToStr(Day) + ' do Mês '
        + IntToStr(Month) + ' do Ano ' + IntToStr(Year);
    DecodeTime(Present, Hour, Min, Sec, MSec);
    Label2.Caption := 'A hora é ' + IntToStr(Hour) + ' horas e '
        + IntToStr(Min) + ' minutos.';
end;
```

```
procedure DecodeTime(Time: TDateTime; var Hour, Min, Sec, MSec: Word);
```

Quebra os valores especificados em Time nos parâmetros Hour, Min, Sec e MSec.

Ex.:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Present: TDateTime;
  Year, Month, Day, Hour, Min, Sec, MSec: Word;
begin
  Present := Now;
  DecodeDate(Present, Year, Month, Day);
  Label1.Caption := 'Hoje é ' + IntToStr(Day) + ' do Mês '
    + IntToStr(Month) + ' do Ano ' + IntToStr(Year);
  DecodeTime(Present, Hour, Min, Sec, MSec);
  Label2.Caption := 'A hora é ' + IntToStr(Hour) + ' horas e '
    + IntToStr(Min) + ' minutos.';
end;
```

```
procedure Delete(var S: string; Index, Count: Integer);
```

Remove a substring de Count caracteres da string S partir da posição Index

Ex.:

```
var
  s: string;
begin
  s := 'Delphi 3 - Client/Server';
  Delete(s, 8, 4);
  Canvas.TextOut(10, 10, s); { ' Delphi 3 - Client/Server ' }
end;
```

```
function DeleteFile(const FileName: string): Boolean;
```

Apaga o arquivo FileName do disco. Se o arquivo não puder ser apagado a função retorna False.

Ex.:

```
DeleteFile('TEMP0001.TMP');
```

```
function DirectoryExists(Name: string): Boolean;
```

Verifica se Name diretório existe

Ex.:

```
function DiskFree(Drive: Byte): Integer;
```

Retorna o número de bytes livre no driver especificado em Drive.

Onde : 0 = Corrente, 1 = A, 2 = B,...

DiskFree retorna -1 se o driver for inválido

Ex.:

```
var
  S: string;
begin
  S := IntToStr(DiskFree(0) div 1024) + ' Kbytes livres.';
  Canvas.TextOut(10, 10, S);
end;
```

```
function DiskSize(Drive: Byte): Integer;
```

Retorna o tamanho em bytes do driver especificado.

Onde : 0 = Corrente, 1 = A, 2 = B,...

DiskFree retorna -1 se o driver for inválido

Ex.:

```
var
  S: string;
begin
  S := 'Capacidade de ' + IntToStr(DiskSize(0) div 1024) + ' Kbytes.';
  Canvas.TextOut(10, 10, S);
end;
```

```
function EncodeDate(Year, Month, Day: Word): TDateTime;
```

Retorna uma Data formada por Year, Month e Day

Ex.:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyDate: TDateTime;
begin
  MyDate := EncodeDate(83, 12, 31);
  Label1.Caption := DateToStr(MyDate);
end;
```

```
function EncodeTime(Hour, Min, Sec, MSec: Word): TDateTime;
```

Retorna a Hora formada por Hour, Min, Sec e MSec

Ex.:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyTime: TDateTime;
begin
  MyTime := EncodeTime(0, 45, 45, 7);
  Label1.Caption := TimeToStr(MyTime);
end;
```

```
function ExtractFileDir(const FileName: string): string;
```

Retorna o diretório adequado para ser passado para as funções CreateDir, GetCurrentDir, RemoveDir e SetCurrentDir.

O resultado da função é uma string vazia se FileName não contiver um drive e um caminho.

```
Function ExtractFileDrive(const FileName: string): string;
```

Retorna uma string contendo o drive do path de um arquivo.

```
Function ExtractFileExt(const FileName: string): string;
```

Retorna a extensão do arquivo FileName

```
function ExtractFileName(const FileName: string): string;
```

Retorna o nome do arquivo

```
Form1.Caption := 'Editando ' + ExtractFileName(FileName);
```

```
function ExtractFilePath(const FileName: string): string;
```

Retorna o Path de um arquivo

Ex.:

```
ChDir(ExtractFilePath(FileName));
```

```
function FileAge(const FileName: string): Integer;
```

Retorna a data e a hora de um arquivo num valor que pode ser convertido para TDateTime através da função FileDateToDateTime. Retorna -1 se o arquivo não existir

```
function FileExists(const FileName: string): Boolean;
```

Retorna verdade se o arquivo existir

Ex.:

```
if FileExists(FileName) then
  if MsgBox('Tem certeza que deseja excluir' + ExtractFileName(FileName)
    + '?'), [] = IDYes then DeleteFile(FileName);
```

```
function FileSize(var F): Integer;
```

Retorna o tamanho de um arquivo, para usar FileSize o arquivo deve estar aberto. Se o arquivo estiver vazio FileSize(F) retorna 0. F é uma variável do tipo arquivo. FileSize não pode ser usada com arquivo texto

Ex.:

```
var
  f: file of Byte;
  size : Longint;
  S: string;
  y: integer;
begin
  if OpenDialog1.Execute then begin
    AssignFile(f, OpenDialog1.FileName);
    Reset(f);
    size := FileSize(f);
    S := 'Tamanho do arquivo em bytes: ' + IntToStr(size);
    y := 10;
    Canvas.TextOut(5, y, S);
    y := y + Canvas.TextHeight(S) + 5;
    S := 'Posicionando no meio do arquivo...';
    Canvas.TextOut(5, y, S);
    y := y + Canvas.TextHeight(S) + 5;

    Seek(f, size div 2);
    S := 'Posição agora é ' + IntToStr(FilePos(f));
    Canvas.TextOut(5, y, S);
    CloseFile(f);
  end;
end;
```

```
procedure FillChar(var X; Count: Integer; value: Byte);
```

Preenche um vetor com determinado caractere. Value pode ser um byte ou char

Ex.:

```
var
  S: array[0..79] of char;
begin
  { preenche tudo com espaços}
  FillChar(S, SizeOf(S), ' ');
end;
```

function FloatToStr(Value: Extended): string;

Converte um valor em ponto flutuante (real) para uma string

procedure ForceDirectories(Dir: string);

Cria multiplos diretórios de uma só vez

Ex.:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Dir: string;
begin
  Dir := 'C:\SISTEMAS\VENDAS\DADOS';
  ForceDirectories(Dir);
  if DirectoryExists(Dir) then
    Labell.Caption := Dir + ' foi criado'
end;
```

```
function FormatDateTime(const Format: string; DateTime: TDateTime): string;
```

Formata o valor DateTime usando o formato de Format.
Os especificadores de formato abaixo são válidos

Especificador Exibe

c Exibe a data usando o formato definido da variável global ShortDateFormat, seguido pela hora se o formato definido pela variável global LongTimeFormat. A hora não é exibida se a parte fracional de DateTime for zero.

d Exibe o dia como um número, sem zero à esquerda (1-31).

dd Exibe o dia como um número, com zero à esquerda (01-31).

ddd Exibe o nome do dia abreviado (Sun-Sat) usando as strings definidas pela variável global ShortDayNames

dddd Exibe o nome do dia (Sunday-Saturday) usando as strings definidas pela variável global LongDayNames.

dddddd Exibe a data utilizando o formato definido na variável global ShortDateFormat.

dddddd Exibe a data utilizando o formato definido na variável global LongDateFormat.

m Exibe o mês como um número sem zeros a esquerda (1-12). Se o especificador m for seguido de um especificador h ou hh, será exibido os minutos do mês.

mm Exibe o mês como um número com zeros a esquerda (01-12). Se o especificador m for seguido de um especificador h ou hh, será exibido os minutos do mês.

mmm Exibe o mês de forma abreviada (Jan-Dec) usando as strings definidas na variável global ShortMonthNames.

mmm Exibe o mês por extenso (January-December) usando as strings definidas na variável global LongMonthNames.

yy Exibe o ano com dois dígitos (00-99).

yyyy Exibe o ano com quatro dígitos (0000-9999).

h Exibe a hora sem zero a esquerda (0-23).

hh Exibe a hora com zero a esquerda (00-23).

n Exibe o minuto sem zero a esquerda (0-59).

nn Exibe o minuto com zero a esquerda (00-59).

s Exibe o segundo sem zero a esquerda (0-59).

ss Exibe o segundo com zero a esquerda (00-59).

t Exibe a hora usando o formato definido na variável global ShortTimeFormat.

tt Exibe a hora usando o formato definido na variável global LongTimeFormat.

am/pm Exibe a hora no formato de 12 horas seguido de 'am' ou 'pm'.

a/p Exibe a hora no formato de 12 horas seguido de 'a' ou 'p'.

ampm Exibe a hora no formato de 12 horas de acordo com o conteúdo das variáveis globais TimeAMString (se antes do meio dia) e TimePMString (se depois do meio dia).

/ Exibe o caracter separador da data definido na variável global DateSeparator.

: Exibe o caracter separador de hora definido na variável global TimeSeparator.

Obs: Caracteres dentro de aspas simples ou duplas ('xx'/"xx") são exibidos como estão e não afetam a formatação.

Os especificadores de formato podem ser utilizados tanto em maiúsculo quanto em minúsculo - ambos produzem o mesmo resultado.

Se a string definida pelo parametro Format estiver vazia, os valores de data e hora serão formatados como as if a 'c' format specifier had been given.

```
S := FormatDateTime('"The meeting is on" dddd, mmmm d, yyyy, ' +  
    '"at" hh:mm AM/PM', StrToDateTime('2/15/95 10:30am'));
```

```
function FormatFloat(const Format: string; Value: Extended): string;
```

Transforma um Float numa string usando a formatação contida em Format

Os especificadores de formato abaixo são válidos

Especificador	Representa
---------------	------------

- | | |
|-----------|---|
| 0 | Posição do dígito. Se o valor sendo formatado tem um dígito na posição que especificador '0' aparece na string de formatação, este dígito será copiado para a string de saída. Senão, um '0' é armazenado nesta posição na string de saída. |
| # | Se o valor sendo formatado tem um dígito na posição que especificador '#' aparece na string formatada, este dígito será copiado para a string de saída. Senão, nada será armazenado na string de saída. |
| . | Ponto decimal. O primeiro caracter '.' na string de formatação determina a localização do ponto decimal no valor sendo formatado; quaisquer caracteres '.' adicionais são ignorados. O caracter que será utilizado como separador decimal é determinado pelo conteúdo da variável global DecimalSeparator. O valor padrão de DecimalSeparator é especificado no Painel de Controle do Windows no ícone Propriedades de Configurações Regionais, guia Número, item Símbolo decimal. |
| , | Separador de milhar. Se a string de formatação contém um ou mais caracteres ',' (vírgula), a saída terá caracteres separadores de milhar inseridos entre os grupos de três dígitos à esquerda do ponto decimal. A posição e o número de caracteres ',' na string de formatação não afetam a saída, exceto para indicar que separadores de milhar são necessários. O caracter que será utilizado como separador de milhar é determinado pelo conteúdo da variável global ThousandSeparator. O valor padrão de ThousandSeparator é especificado no Painel de Controle do Windows no ícone Propriedades de Configurações Regionais, guia Número, item Símbolo decimal. |
| E+ | Notação científica. Se qualquer uma das strings 'E+', 'E-', 'e+', ou 'e-' estiverem contidas na string de formatação, o número é formatado usando notação científica. Um grupo de quatro caracteres '0' (zero) pode seguir imediatamente após os especificadores 'E+', 'E-', 'e+', ou 'e-' para determinar o número mínimo de dígitos no expoente. Os especificadores 'E+' e 'e+' fazem com que o sinal do expoente (positivo ou negativo) seja exibido. Os especificadores 'E-' e 'e-' fazem com que o sinal do expoente seja exibido apenas quando o mesmo for negativo. |
| 'xx'/'xx' | Caracteres entre aspas simples ou duplas, não afetam a formatação. |
| ; | Separa seções para números positivos, negativos e o zero na string de formatação. |

Obs.: A localização do '0' extremo esquerdo antes do ponto decimal e do '0' extremo direito após o ponto decimal na string de formatação determinam a faixa de dígitos sempre exibida na string de saída. O número sendo formatado sempre é arredondado para o número de dígitos definidos após o ponto decimal (com '0' ou '#'). Se a string de formatação não contém o ponto decimal, o valor sendo formatado será arredondado para o número mais próximo.

Se o número sendo formatado possui mais dígitos à esquerda do separador decimal que os definidos pelos especificadores de dígito na string de formatação, dígitos extras serão exibidos antes do primeiro especificador de dígito.

Para definir formatação diferente para valores positivos, negativos, e zero, a string de formatação pode conter entre uma e três seções separadas por ';' (ponto-e-vírgula).

Uma seção: A string de formatação é aplicada para todos os valores.

Duas seções: A primeira seção é aplicada para valores positivos e o zero, e a segunda seção se aplica a valores negativos.

Três seções: A primeira seção é aplicada para valores positivos, a segunda seção se aplica a valores negativos, e a terceira seção para o zero.

Se a seção para valores negativos ou a seção para o zero está vazia, ou seja, se nada estiver definido entre os separadores de seção ';' (ponto-e-vírgula), então a seção para valores positivos será utilizada.

Se a seção de valores positivos estiver vazia, ou se a string de formatação estiver vazia, o valor é formatado usando a formatação geral de números de ponto flutuante com 15 dígitos significativos, correspondendo a uma chamada a `FloatToStrF` com o formato `ffGeneral`. Formatação geral de ponto flutuante também é usada quando o valor tem mais de 18 dígitos à esquerda do ponto decimal e a string de formatação não especifica formatação com notação científica.

Exemplos de strings de formatação e valores resultantes:

0	1234	-1234	1	0	
0.00	1234.00	-1234.00	0.50	0.00	
###	1234	-1234	.5		
###0.00	1,234.00	-1,234.00	0.50	0.00	
###0.00;(#,###0.00)	1,234.00	(1,234.00)	0.50	0.00	
###0.00;;Zero	1,234.00	-1,234.00	0.50	Zero	
0.000E+00	1.234E+03	-1.234E+03	5.000E-01	0.000E+00	
#####E-0	1.234E3	-1.234E3	5E-1	0E0	

function `Frac(X: Real): Real;`

Retorna a parte fracional do parâmetro X

Ex.:

```
var
  R: Real;
begin
  R := Frac(234.567);    { 0.567 }
  R := Frac(-234.567);  { -0.567 }
end;
```

function `GetCurrentDir: string;`

Retorna uma string contendo o diretório corrente

procedure `GetDir(D: Byte; var S: string);`

Retorna o diretório corrente do drive especificado.
O onde D pode ser :

Valor	Drive
0	Corrente
1	A
2	B
3	C

Ex.:

```
var
  s: string;
begin
  GetDir(0,s); { 0 = drive corrente }
  MessageDlg('Drive e diretório atual: ' + s, mtInformation, [mbOk], 0);
end;
```

```
procedure Inc(var X [ ; N: Longint ] );
```

Incrementa de uma ou N unidades o parâmetro X

Ex.:

```
var
  IntVar: Integer;
  LongintVar: Longint;
begin
  Inc(IntVar);      { IntVar := IntVar + 1 }
  Inc(LongintVar, 3; { LongintVar := LongintVar + 3}
end;
```

```
function IncMonth(const Date: TDateTime; NumberOfMonths: Integer): TDateTime;
```

Retorna Date acrescido ou decrescido de NumberOfMonths meses.

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

Exibe uma Caixa de Entrada onde o usuário pode digitar uma string.

ACaption representa o título do Input Box e APrompt é o título do edit e ADefault representa o valor inicial do Edit.

Ex.:

```
uses Dialogs;
procedure TForm1.Button1Click(Sender: TObject);
var
  InputString: string;
begin
  InputString:= InputBox('Informe', 'Nome do arquivo a ser criado', 'MeuTexto.TXT');
end;
```

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

Semelhante ao InputBox, sendo que retorna True se o usuário fechou a O InputBox com OK e False se fechou com Cancel ou ESC. E Value armazena a string digitada.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NewString: string;
  ClickedOK: Boolean;
begin
  NewString := 'MeuTexto.TXT';
  Labell.Caption := NewString;
  ClickedOK := InputQuery('Informe', 'Nome do arquivo a ser criado', NewString);
  if ClickedOK then { NewString contém nova string digitada.}
    Labell.Caption := 'A nova string é ' + NewString + ' ';
end;
```

```
procedure Insert(Source: string; var S: string; Index: Integer);
```

Insere uma string em outra a partir da posição Index

Ex.:

```
var
  S: string;
begin
  S := 'Delphi 3 /Server';
  Insert('Client', S, 9);      { 'Delphi 3 Client/Server' }
end;
```

```
function IntToHex(Value: Integer; Digits: Integer): string;
```

Converte o inteiro Value num Hexadecimal (Base 16). Digits indica o número mínimo de dígitos Hexa a serem retornados

Ex.:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Edit2.Text := IntToHex(StrToInt(Edit1.Text), 6);
end;
```

```
function IntToStr(Value: Integer): string;
```

Transforma um Inteiro numa String

Ex.:

```
procedure TForm1.Button1Click(Sender: TObject);
var
    Value: Integer;

begin
    Value := 1583;
    Edit1.Text := IntToStr(Value);
end;
```

```
function IsValidIdent(const Ident: string): Boolean;
```

Indica se um identificador é válido para o Pascal

```
function Length(S: string): Integer;
```

Retorna o número de caracteres usados na string S.

Ex.:

```
var
    S: string;

begin
    S := 'Curso de Programação em Delphi 3 - Object Pascal';
    Canvas.TextOut(10, 10, 'Tamanho da String = ' + IntToStr(Length(S)));
end;
```

```
function MaxIntValue(const Data: array of Integer): Integer;
```

Retorna o maior inteiro de um vetor

```
function MaxValue(const Data: array of Double): Double;
```

Retorna o maior valor de um vetor

```
function Mean(const Data: array of Double): Extended;
```

Retorna a média aritmética de um vetor

```
function MessageDlg(const Msg: string; AType: TMsgDlgType;
  AButtons: TMsgDlgButtons; HelpCtx: Longint): Word;
```

Exibe uma Caixa de Mensagem e obtém uma resposta do usuário.
Onde

Msg : Mensagem a ser exibida.

AType : Tipo da caixa de mensagem

Valor	Significado
mtWarning	Uma caixa de mensagem contendo um ponto de exclamação amarelo.
mtError	Uma caixa de mensagem contendo um símbolo de pare vermelho.
mtInformation	Uma caixa de mensagem contendo um "i" azul.
mtConfirmation	Uma caixa de mensagem contendo um ponto de interrogação verde.
mtCustom	Uma caixa de mensagem sem bitmap.

Obs.: O caption da caixa de mensagem é o nome do arquivo executável da aplicação.

AButtons: Quais botões aparecerão na caixa de mensagem.

Onde:

Valor	Significado
mbYes	Um botão com o texto 'Yes'
mbNo	Um botão com o texto 'No'
mbOK	Um botão com o texto 'OK'
mbCancel	Um botão com o texto 'Cancel'
mbHelp	Um botão com o texto 'Help'
mbAbort	Um botão com o texto 'Abort'
mbRetry	Um botão com o texto 'Retry'
mbIgnore	Um botão com o texto 'Ignore'
mbAll	Um botão com o texto 'All'
mbYesNoCancel	Coloca os botões Yes, No, e Cancel na caixa de mensagem
mbOkCancel	Coloca os botões OK e Cancel na caixa de mensagem
mbAbortRetryIgnore	Coloca os botões Abort, Retry, e Ignore na caixa de mensagem

MessageDlg retorna o valor do botão selecionado. Os valores de retorno possíveis são:

mrNone	mrAbort	mrYes
mrOk	mrRetry	mrNo
mrCancel	mrIgnore	mrAll

Ex.:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if MessageDlg('Deseja sair da aplicação agora?',
    mtConfirmation, [mbYes, mbNo], 0) = mrYes then
  begin
    MessageDlg('Tenha um bom dia...ject Pascal application.', mtInformation,
      [mbOk], 0);
    Close;
  end;
end;
```

```
function MessageDlgPos(const Msg: string; AType: TMsgDlgType;
  AButtons: TMsgDlgButtons; HelpCtx: Longint; X, Y: Integer): Word;
```

Semelhante a MessageDlg exceto por permitir indicar a posição na qual a janela será exibida

Ex.:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  MessageDlgPos('Sair sem gravar alterações ?', mtConfirmation, mbYesNoCancel, 0, 200, 200);
end;
```

```
function MinIntValue(const Data: array of Integer): Integer;
```

Retorna o menor inteiro do vetor

```
function MinValue(const Data: array of Double): Double;
```

Retorna o menor valor de um vetor

```
procedure Mkdir(S: string);
```

Cria um novo diretório

Ex.:

```
uses Dialogs;
begin
  {$I-}
  { Pega o nome do diretório de um controle Tedit }
  Mkdir(Edit1.Text);
  if IOResult <> 0 then
    MessageDlg('Não posso criar o diretório.', mtWarning, [mbOk], 0)
  else
    MessageDlg('Diretório criado', mtInformation, [mbOk], 0);
end;
```

```
function Now: TDateTime;
```

Retorna a data e a hora corrente

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := 'Data e hora atual é ' + Str(Now);
end;
```

```
function Ord(X): Longint;
```

Retorna a ordem de um valor ordinal

```
uses Dialogs;
type
  Colors = (RED, BLUE, GREEN);
var
  S: string;
begin
  S := 'BLUE (Azul) tem um valor ordinal de ' + IntToStr(Ord(BLUE)) + #13#10;
  S := 'O código decimal ASCII para "p" é ' + IntToStr(Ord('p'));
  MessageDlg(S, mtInformation, [mbOk], 0);
end;
```

function Pi: Extended;

Retorna o valor de PI
3.1415926535897932385.

function Pos(Substr: string; S: string): Integer;

Procura por uma sub-string numa string e retorna a posição da primeira ocorrência ou zero se não encontrou

Ex.:

```
var S: string;
begin
  S := '  123.5';
  { Converte espaços para zeros }
  while Pos(' ', S) > 0 do
    S[Pos(' ', S)] := '0';
end;
```

function Power(Base, Exponent: Extended): Extended;

Retorna a Potência.

function Pred(X);

Retorna o predecessor de um ordinal

```
uses Dialogs;
type
  Colors = (RED,BLUE,GREEN);
var
  S: string;
begin
  S := 'O predecessor de 5 é ' + IntToStr(Pred(5)) + #13#10;
  S := S + 'O sucessor de 10 é ' + IntToStr(Succ(10)) + #13#10;
  if Succ(RED) = BLUE then
    S := S + 'Nos tipos de Cores, RED (Vermelho) é o predecessor de BLUE (Azul).';
  MessageDlg(S, mtInformation, [mbOk], 0);
end;
```

function Random [(Range: Integer)];

Retorna um valor Randômico
 $0 \leq X < \text{Range}$.

```
var
  I: Integer;
begin
  Randomize;
  for I := 1 to 50 do begin
    { Escreve na janela em posições randomicas }
    Canvas.TextOut(Random(Width), Random(Height), 'Olá!');
  end;
end;
```

procedure Randomize;

Inicializa o modo Randomico. Deve ser utilizada antes de Random

```
function RemoveDir(const Dir: string): Boolean;
```

Remove um diretório retornando True caso tenha conseguido e False caso contrário

```
procedure Rename(var F; Newname);
```

Altera o nome de um arquivo.

F é uma variável de arquivo (file). Newname é uma expressão do tipo string ou do tipo Pchar.

Ex.:

```
uses Dialogs;
var
  f : file;
begin
  OpenFileDialog1.Title := 'Escolha um arquivo... ';
  if OpenFileDialog1.Execute then begin
    SaveDialog1.Title := 'Renomear para...';
    if SaveDialog1.Execute then begin
      AssignFile(f, OpenFileDialog1.FileName);
      Canvas.TextOut(5, 10, 'Renomeando ' + OpenFileDialog1.FileName + ' para ' +
        SaveDialog1.FileName);
      Rename(f, SaveDialog1.FileName);
    end;
  end;
end;
```

```
function RenameFile(const OldName, NewName: string): Boolean;
```

Renomeia arquivos e retorna o sucesso ou insucesso

Ex.:

```
{ O código a seguir renomeia um arquivo }
if not RenameFile('OLDNAME.TXT', 'NEWNAME.TXT') then
  ErrorMsg('Erro ao renomear arquivo!');
```

```
procedure RmDir(S: string);
```

Remove um diretório

Ex.:

```
uses Dialogs;
begin
  {$I-}
  { Pega o nome do diretório de um controle TEdit }
  RmDir(Edit1.Text);
  if IOResult <> 0 then
    MessageDlg('Não posso remover o diretório', mtWarning, [mbOk], 0)
  else
    MessageDlg('Diretório removido', mtInformation, [mbOk], 0);
end;
```

```
function Round(X: Extended): Longint;
```

Arredonda um número real

```
function SelectDirectory(var Directory: string; Options: TSelectDirOpts; HelpCtx: Longint): Boolean;
```

Exibe um Caixa de Dialogo para seleção de Diretório. O Diretório passado para a função aparece como diretório corrente e o diretório escolhido é retornado no mesmo Diretório (Directory). O valor do diretório corrente não é alterado

Os valores abaixo podem ser adicionados ao parametro Options:

Valor	Significado
-------	-------------

sdAllowCreate	Uma caixa de edição aparece para permitir ao usuário digitar o nome do diretório que não existe. Esta opção não cria o diretório, mas a aplicação pode acessar o parametro Directory para criar o diretório desejado.
---------------	---

sdPerformCreate	Usado somente quando Options contém sdAllowCreate. Se o usuário entrar com um diretório que não existe, SelectDirectory cria-o.
-----------------	---

sdPrompt	Usado quando Options contém sdAllowCreate. Exibe uma caixa de mensagem que informa o usuário quando o diretório digitado não existe e pergunta se deseja criá-lo. Se o usuário selecionar OK, o diretório é criado se Options contém sdPerformCreate. Se Options não contém sdPerformCreate, o diretório não é criado: a aplicação precisa criá-o quando SelectDirectory retornar.
----------	--

A função retorna True se o usuário selecionar o diretório e clicar em OK, e False se o usuário selecionar Cancel ou fechar a caixa de diálogo sem selecionar um diretório.

Ex.:

```
uses FileCtrl;

procedure TForm1.Button1Click(Sender: TObject);
var
  Dir: string;
begin
  Dir := 'C:\MYDIR';
  if SelectDirectory(Dir, [sdAllowCreate, sdPerformCreate, sdPrompt]) then
    Label1.Caption := Dir;
end;
```

```
function SetCurrentDir(const Dir: string): Boolean;
```

Torna Dir o diretório corrente e retorna o sucesso da operação

```
procedure SetLength(var S: string; NewLength: Integer);
```

Coloca um novo tamanho para uma string. O efeito é similar ao código abaixo:

S[0] := NewLength.

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

Exibe uma mensagem ao usuário

Ex.:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage('Olá !');
end;
```

```
function Sqr(X: Extended): Extended;
```

Retorna o quadrado de X

```
function Sqrt(X: Extended): Extended;
```

Retorna a raiz quadrada de X

```
function StrComp(Str1, Str2 : PChar): Integer;
```

Compara duas strings em case sensitivity (diferencia maiúsculas e minúsculas)

Valor de retorno	Condição
<0	if Str1 < Str2
=0	if Str1 = Str2
>0	if Str1 > Str2

```
function StringOfChar(Ch: Char; Count: Integer): string;
```

Retorna uma string contendo Count caracteres Ch

Ex.:

```
S := StringOfChar('A', 10);
{sets S to the string 'AAAAAAAAAA'}
```

```
function StrToDate(const S: string): TDateTime;
```

Converte uma string em data

Ex.:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ADate: TDateTime;
begin
  ADate := StrToDate(Edit1.Text);
  Label1.Caption := DateToStr(ADate);
end;
```

```
function StrToDateTime(const S: string): TDateTime;
```

Converte uma string para o formato DateTime

Ex.:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ADateAndTime: TDateTime;
begin
  ADateAndTime := StrToDateTime(Edit1.Text);
  Label1.Caption := DateTimeToStr(ADateAndTime);
end;
```

```
function StrToFloat(const S: string): Extended;
```

Converte uma string num Float

```
function StrToInt(const S: string): Integer;
```

Converte uma string num inteiro

Ex.:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  S: string;
  I: Integer;
begin
  S := '22467';
  I := StrToInt(S);
  Inc(I);
  Edit1.Text := IntToStr(I);
end;
```

```
function StrToTime(const S: string): TDateTime;
```

Converte uma string em hora

Ex.:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ATime: TDateTime;
begin
  ATime := StrToTime(Edit1.Text);
  Label1.Caption := TimeToStr(ATime);
end;
```

```
function Succ(X);
```

Retorna o sucessor de um ordinal

Ex.:

```
uses Dialogs;
type
  Colors = (RED,BLUE,GREEN);
var
  S: string;
begin
  S := 'O predecessor de 5 é ' + IntToStr(Pred(5)) + #13#10;
  S := S + 'O sucessor de 10 é ' + IntToStr(Succ(10)) + #13#10;
  MessageDlg(S, mtInformation, [mbOk], 0);
end;
```

```
function Sum(const Data: array of Double): Extended register;
```

Calcula a soma de dos elementos de um vetor

```
function SumInt(const Data: array of Integer): Integer register;
```

Calcula a soma dos elementos de um vetor de inteiros

function Time: TDateTime;

Retorna a hora corrente

Ex.:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Label1.Caption := 'A hora é ' + TimeToStr(Time);
end;
```

function TimeToStr(Time: TDateTime): string;

Converte hora para string

Ex.:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Label1.Caption := TimeToStr(Time);
end;
```

function Trim(const S: string): string;

Retira os espaços em brancos a esquerda e a direita da string S

function TrimLeft(const S: string): string;

Retira os espaços em brancos a esquerda da string S

function TrimRight(const S: string): string;

Retira os espaços em brancos a direita da string S

function Trunc(X: Extended): Longint;

Retorna a parte inteira de um número

function UpCase(Ch: Char): Char;

Converte o caracter Ch em maiúscula

Ex.:

```
uses Dialogs;
var
    s : string;
    i : Integer;
begin
    { Pega a string de um controle TEdit }
    s := Edit1.Text;
    for i := 1 to Length(s) do
        s[i] := UpCase(s[i]);
    MessageDlg('Tudo em maiúsculo: ' + s, mtInformation, [mbOk], 0);
end;
```

function UpperCase(const S: string): string;

Converte a string S em maiúsculas

Ex.:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to ListBox1.Items.Count -1 do
    ListBox1.Items[I] := UpperCase(ListBox1.Items[I]);
end;
```

APÊNDICE - TECNOLOGIAS APLICÁVEIS AO DELPHI 7

Quinta, Setembro 25, 2003 , por [Jorge Cavalheiro Barbosa](#)

Quando executamos o Delphi® 7, deparamo-nos com uma coleção de páginas organizadas numa paleta contendo centenas de componentes visuais e não visuais. Para quem é neófito no uso dessa ferramenta, dependendo da pessoa, isso pode ser desencorajador ou, contrariamente, desafiante. Até para mim, que já uso o Delphi há alguns anos, todo esse arsenal de recursos ainda apresentava uma certa nublidade.

As questões que me surgiam eram: qual das tecnologias presentes no Delphi é adequada para que propósitos? Quais as relações entre as centenas de siglas existentes neste planeta interconectado e as maneiras de trabalhar, no Delphi, com os recursos que elas representam? Quais dessas tecnologias vão permanecer e quais estão em processo de mutação ou substituição, notadamente para integração às novas plataformas (.NET, por exemplo) e novas concepções do mundo digital (software livre, sistemas independentes de plataforma, etc.)? O compêndio a seguir visa reunir alguns aspectos marcantes dessas tecnologias, e foi elaborado a partir do livro Dominando o Delphi 7, de Marco Cantù. Essa obra, bastante extensa, considerada “a Bíblia” do Delphi, assim mesmo não alcança toda a profundidade e extensão dos recursos oferecidos por esse ambiente de desenvolvimento, tal a sua abrangência. Quando produzi este material, tinha em mente apenas a minha própria orientação, mas, ao finalizá-lo, percebi que se foi útil para esclarecer alguns dos meus questionamentos, também poderia (e deveria) ser disponibilizado para a comunidade de desenvolvedores Delphi e quem mais tivesse interesse no tema. Afinal, conhecimento não disseminado não produz energia. Ao final do artigo, temos um diagrama que mostra visualmente essas relações.

ODBC (Open DataBase Connectivity):

- Conectividade de banco de dados aberta, da Microsoft.
- A Microsoft começou a substituir o ODBC pelo **OLE DB** em meados dos anos 90, através do [COM](#).
- É grande, complexo, impõe maiores demandas sobre programador e exige um nível mais alto de conhecimento, em troca de uma produtividade menor.
- É destinado a ser usado por programadores **em nível de sistema operacional**.
- O [ADO](#) é a camada sobre o **OLE DB** e é referido como uma interface **em nível de aplicativo**, sendo mais simples que esse.

IDAPI (Integrated DataBase Application Programming Interface): interface de programação de aplicativo de banco de dados integrada, da Borland, mais conhecida como [BDE](#).

SOAP (Simple Object Access Protocol – protocolo de acesso a objetos simples):

- Integrante da estrutura de WebServices (vide [WDSL](#)).
- Criado com base no HTTP-padrão para que um servidor Web pudesse manipular os pedidos SOAP e os pacotes de dados relacionados pudessem **passar por firewalls**.
- Define uma notação baseada em [XML](#) para solicitar a execução de um método por um objeto no servidor, passando parâmetros para ele, e uma notação para definir o formato de uma resposta.
- É uma pedra fundamental da arquitetura [.NET](#) DA Microsoft e também das plataformas atuais da Sun e da Oracle.
- O SOAP substituirá as chamadas ao [COM](#), pelo menos entre computadores diferentes.

WDSL (WebServices Description Language – linguagem de descrição de serviços para a Web):

- Integrante da estrutura de WebServices.
- Substituirá a IDL e as bibliotecas de tipo usadas pelo [COM/COM+](#).

- Os documentos WDSL são outro tipo de documento [XML](#) que fornece a definição de metadados de uma requisição **SOAP**.
- Um **WebService** normalmente está contido em um servidor Web que usa uma das tecnologias de extensão de servidor Web disponíveis ([CGI](#), [ISAPI](#), [módulos Apache e outras](#)).

Um módulo **WebServices** contém pelo menos três componentes básicos:

- a) **HTTPSoapDispatcher**: recebe a requisição da Web, como qualquer outro distribuidor HTTP.
- b) **HTTPSoapPascalInvoker**: realiza a operação inversa do componente **HTTPRIO**, pois ele pode transformar requisições [SOAP](#) em chamadas a interfaces **Pascal** (em vez de transformar chamadas a métodos de interfaces em requisições [SOAP](#)).
- c) **WSDLHTMLPublish**: pode ser usado para extrair a definição WSDL do serviço das interfaces que ele suporta e executar o papel oposto do **WebServices Import Wizard**.

Indy (Internet Direct):

- A página **Internet** da paleta de componentes do Delphi contém os componentes **TcpClient** e **TcpServer**, desenvolvidos em substituição a **ClientSocket** e **ServerSocket**, disponíveis em versões anteriores do Delphi. A Borland sugere o uso dos componentes Indy correspondentes.
- A paleta Indy tem mais de cem componentes. Esses componentes usam exclusivamente conexões com bloqueio (blocking).
- A base do trabalho com soquetes é o protocolo **TCP/IP**. Cada conexão de soquete é composto por um endereço IP e a porta TCP. As portas mais comuns são: 80-HTTP; 21-FTP; 25-SMTP; 110-POP3; 23-TelNet. O arquivo **services** lista as portas padrão.

Existem três tipos de conexão:

- a) de **cliente**: são iniciadas pelo cliente e conectam um soquete cliente local a um soquete servidor remoto.
 - b) de **escuta** (listening connections): são soquetes servidores passivos esperando por clientes. Quando um cliente faz uma requisição, o servidor gera um soquete dedicado a essa conexão específica e, em seguida, volta ao estado de escuta. Os servidores de escuta indicam a porta que representa o serviço que fornecem.
 - c) as conexões de **servidor** são conexões ativadas por servidores, pois elas aceitam uma requisição de um cliente.
- Os componentes TCP do Indy são: **IdTCPClient** e **IdTCPServer**. Coloca-se um deles em um programa e o outro em um programa diferente, configurando-os para a mesma porta e deixa-se que o programa cliente consulte o programa servidor.
 - Os sockets também podem ser usados para operações em bancos de dados remotos, usando [DataSnap](#) e [SOAP](#). São adequados para uso em correio eletrônico, protocolos **HTTP**.

Através dos componentes produtores de **HTML** do Delphi, presentes na paleta **Internet**, é possível gerar tabelas **HTML** a partir de tabelas de bancos de dados, usando a tecnologia [WebBroker](#):

- a) **PageProducer**: manipula arquivos **HTML** nos quais foram incorporadas tags especiais.
- b) **DataSetPageProducer**: extensão do **PageProducer**, substituindo automaticamente as tags correspondentes aos nomes de campo de uma fonte de dados conectada.
- c) **DataSetTableProducer**: útil para apresentar o conteúdo de uma tabela, consulta o outro conjunto de dados, com pré-visualização em tempo de projeto..
- d) **QueryTableProducer** e **SQLQueryTableProducer**: são especificamente personalizados para a construção de consultas paramétricas (para o [BDE](#) ou [dbExpress](#)).

DataSnap:

Arquitetura de multicamada (multitier) física:

- a) **servidor de dados**: instalação e configuração de acesso ao banco de dados;
 - b) **servidor de aplicativos**: comunicam-se com o servidor de dados;
 - c) **clientes magros**: não se comunicam diretamente com o servidor de banco de dados, mas com o servidor de aplicativos).
- A vantagem dessa arquitetura é que a configuração do acesso ao banco de dados precisa ser instalada apenas no servidor de aplicativos.
 - Essa arquitetura era chamada [Midas](#).
 - Com o Delphi 7, não é preciso pagar taxa de distribuição para os aplicativos **DataSnap**.
 - Desenvolvimento cliente/servidor: o banco de dados e o mecanismo de acesso ficam no servidor, reduzindo a necessidade de processamento nos clientes.

- Exige instalação de bibliotecas específicas na camada intermediária que fornece a seus computadores clientes os dados extraídos do servidor de banco de dados.
- Não exige um servidor SQL para armazenamento dos dados. Pode servir dados de uma variedade de fontes, incluindo SQL, **CORBA** e outros servidores DataSnap.

Suporta os seguintes protocolos e conexões:

- a) **DCOM (COM Distribuído)**, e **COM sem estados (MTS ou COM+)**: oferecem recursos como segurança, gerenciamento de componentes e transações de bancos de dados. Devido a problemas de passagem através de firewalls, até a Microsoft está abandonando o **DCOM** em favor de soluções baseadas em **SOAP**.
- b) **Soquetes TCP/IP**: distribuição de clientes através da Web onde não se pode contar com **DCOM**. Para isso, o computador da camada intermediária deve executar o **ScktSrvr.exe**, fornecido pela Borland, como aplicativo ou serviço, que recebe os pedidos dos clientes e os encaminha para o módulo de dados remoto, usando COM. Os soquetes não fornecem proteção contra falhas no lado cliente, pois o servidor não é informado e pode não liberar recursos quando um cliente desligar inesperadamente. Apresenta problemas com firewalls.
- c) **HTTP e SOAP**: **HTTP** simplifica conexões através de firewalls ou servidores proxy (que não gostam de soquetes **TCP/IP** personalizados). Nesse caso, usa-se um aplicativo servidor Web específico, **httpsrvr.dll**, que aceita requisições de clientes e cria-se os módulos de dados remotos apropriados, usando **COM**. Essas conexões Web podem usar segurança **SSL**. As conexões Web baseadas no transporte HTTP podem usar o suporte de pooling de objetos do DataSnap.

O transporte HTTP do DataSnap pode usar **XML** como formato de pacote de dados, permitindo que qualquer plataforma ou ferramenta que possa ler **XML** participe. O uso de **XML** sob **HTTP** também é a base do **SOAP**.

- O componente DCOMConnection pode ser usado no lado cliente para se conectar a um servidor **DCOM** e MTS, localizado no computador atual ou em outro computador indicado pela propriedade ComputerName. A conexão se dá com um objeto registrado que tem determinado ServerGUID ou ServerName.
- O componente SocketConnection pode ser usado para se conectar com o servidor por meio de **TCP/IP**. Deve-se indicar o endereço IP ou o nome do host, e o IntercepGUID do objeto servidor.
- O componente WebConnection é usado para manipular uma conexão **HTTP**, que pode passar por um firewall facilmente. Deve ser indicada a URL na qual sua cópia do arquivo **httpsrvr.dll** está localizada e o nome ou GUID do objeto remoto no servidor.
- O componente ConnectionBroker pode ser usado como um alias de um componente de conexão real, o que é útil quando se tem um único aplicativo com múltiplos conjuntos de dados-cliente. Para mudar a conexão física de cada um dos conjuntos de dados, basta mudar a propriedade Connection.
- O componente SharedConnection permite que um aplicativo conecte-se a múltiplos módulos de dados do servidor, com uma única conexão compartilhada.
- O componente LocalConnection pode ser usado para fazer com que um provedor de conjuntos de dados local atue como a fonte do pacote de dados, permitindo escrever um aplicativo local com o mesmo código de um aplicativo multicamada complexo.
- Os componentes XMLTransform, XMLTransformProvider e XMLTransformClient se relacionam com a transformação do pacote de dados DataSnap em formatos XML personalizados.
- O recurso de pooling permite que se compartilhe alguns objetos da camada intermediária entre um número muito maior de clientes, porque quando um cliente não está chamando um método do objeto camada intermediária, esse mesmo módulo remoto pode ser usado para um outro cliente.

UDDI (Universal Description, Discovery and Integration – descrição, descoberta e integração universais):

- Esforço para criar um catálogo de serviços Web oferecido por empresas de todo o planeta, objetivando criar uma estrutura aberta, global e independente de plataforma para que as entidades de negócios se encontrem, definam como interagem com a rede Internet e compartilhem de um registro global de negócios, agilizando a adoção do comércio eletrônico na forma de **B2B** (business to business).

Basicamente é um registro global de negócios, incluindo:

- a) **White Pages**: informações de contato, endereços e outras informações.
 - b) **Yellow Pages**: registra a empresa em uma ou mais taxonomias, incluindo a categoria industrial, produtos vendidos, informações geográficas e outras.
- Green Pages: relaciona os serviços Web oferecidos pelas empresas. Cada serviço é relacionado por tipo (chamado Tmodel).

- A API UDDI se baseia no **SOAP**.

O Delphi 7 inclui um navegador UDDI que você pode usar para localizar um serviço Web ao importar um arquivo **WSDL**.

IntraWeb: Desenvolvimento de aplicativos para Web (não **sites** Web).

- Propõe-se a programadores que querem apenas criar aplicativos Web no Delphi da maneira como eles criavam aplicativos VCL ou CLX.
- Os aplicativos criados podem ser usados em um programa [WebBroker](#) ou [WebSnap](#).
- Arquitetura relativamente aberta (o código fonte pode ser adquirido).
- Disponível para Delphi, Kylix, C++ Builder, Java e .NET (esta última em andamento).
- Aplicativos IntraWeb podem ser totalmente independentes de plataforma, dependendo de como são desenvolvidos.
- Ao se trabalhar com [WebBroker](#) ou [WebSnap](#), faz-se isso muito no nível de **geração de HTML**. Quando se trabalha com IntraWeb, faz-se isso em termos de componentes, de suas propriedades e de seus manipuladores de evento, como faz no desenvolvimento visual do Delphi.
- Podem ser usadas diferentes arquiteturas para criar e implantar aplicativos: o **modo application** leva em conta todos os recursos da IntraWeb, e pode ser implantado como biblioteca [ISAPI](#), módulos [Apache](#) ou modo standalone da IntraWeb. O modo page é uma versão simplificada que você pode ligar aos aplicativos Delphi [WebBroker](#) ou [WebSnap](#) existentes, e pode ser implantado como [ISAPI](#), [Apache](#), [CGI](#), etc.
- Incorpora gerenciamento de sessões pré-definido e simplifica o trabalho com elas.
- Integra-se com WebBroker e WebSnap.

ModelMaker: Ambiente de Projeto: UML (Unified Modeling Language): ferramenta CASE e de diagramação em UML de ciclo completo extensível, personalizável e específica do Delphi.

Rave Reports: Geração de Relatórios.

CLX (Component Library for Cross-Platform): Criação de Sistemas que funcionam em ambientes Windows e Linux (independente de plataforma)

.NET

- O Runtime Environment da estrutura .NET pode ser baixado gratuitamente no site Web MSDN da Microsoft. O .NET Framework SDK (Software Development Kit), que é mais completo e inclui documentação e ferramentas para desenvolvedores também pode ser baixado dali. Antes de rodar o Delphi for .NET Preview, uma dessas estruturas deve estar instalada.
- Sob o ponto de vista do programador, o recurso básico da plataforma .NET é o seu ambiente gerenciado, que permite a execução de código intermediário compilado por várias linguagens de programação diferentes (desde que sejam compatíveis com uma definição comum para os tipos de dados básicos). Esse ambiente incorpora muitos recursos, que variam do gerenciamento de memória sofisticado à segurança integrada.
- A biblioteca de classes cobre diversas áreas de desenvolvimento (formulários baseados no Windows, desenvolvimento para a Web, desenvolvimento de serviços Web, processamento XML, acesso a bancos de dados e muito mais).

CLI (Common Language Infrastructure – infra-estrutura de linguagens comum): compreende:

- CTS** (Common Type System – sistema de tipos comuns): cria a fundação para integrar linguagens de programação diferentes. Ela especifica como os tipos são declarados e usados, e os compiladores de linguagem devem ser compatíveis com essa especificação, para aproveitar a integração independente de linguagem da plataforma .NET.
- Extensible Metadata** (metadados extensíveis): Cada unidade (assembly) deve trazer com ela os dados que a identificam completamente (nome, versão, cultura e chave pública opcionais), os dados que descrevem os tipos definidos, os dados que relacionam os outros arquivos referenciados e todas as permissões especiais de segurança exigidas. Pode conter ainda outros elementos.
- CIL** (Common Intermediate Language – linguagem intermediária comum): é a linguagem de programação de um ambiente de execução virtual com um microprocessador abstrato. Os compiladores que têm como destino a plataforma .NET não geram instruções de CPU nativas, mas instruções na linguagem intermediária do processador abstrato. É semelhante aos bytecodes da Java.
- P/Invoke** (Platform Invocation Services – serviços de chamadas de plataforma): Os programas executados no ambiente em tempo de execução da .NET são reproduzidos em sua própria sandbox particular. Existe uma camada grande e complicada de software entre os programas e o sistema operacional. O ambiente de execução da .NET não substitui completamente a API Win32, de modo que deve haver uma maneira de fazer a ponte entre os dois mundos.

e) **FCL** (Framework Class Library – bibliotecas de classe da estrutura): é uma hierarquia de classes semelhante à VCL. Todas as classes, numa aplicação Delphi .NET devem ser herdadas da .NET Fx.

f) **PE** (Extended Portable Executable – executável portátil estendido): formato usado pelos arquivos-padrão executáveis do Win32 – para dar suporte aos requisitos de metadados da CLI. Assim, o sistema operacional pode carregar um aplicativo .NET da mesma forma que carrega um aplicativo Win32 nativo.

- **CLR** (Common Language Runtime – ambiente de tempo de execução de linguagens comum): a **CLI** é uma especificação e o CLR é a implementação daquela. É uma biblioteca runtime que unifica a grande variedade de serviços oferecidos pelo sistema Operacional Windows e os apresenta ao programador dentro de uma estrutura orientada a objetos. É responsável por todo o universo .NET: carregamento da imagem executável, verificação de sua identidade e segurança quanto a tipos de dados, compilação do código CIL em instruções nativas de CPU e gerenciamento do aplicativo durante sua vida útil. O código **CIL** que deve ser executado no CLR é chamado de código gerenciado, enquanto o código restante (como o código executável Intel produzido pelo Delphi 7) é não gerenciado.

CLS (Common Language Specification – especificação de linguagens comum): é um subconjunto da CTS, definindo regras que governam o modo como os tipos criados nas diferentes linguagens de programação podem interoperar.

"OLE/COM/COM+ /DCOM">OLE/COM/COM+ /DCOM (Distributed Component Object Model):

- Tecnologia que define uma maneira-padrão para um módulo (aplicativo ou biblioteca) cliente e um servidor se comunicarem, por meio de uma interface específica. Ex.: geração de dados para Excel, Word, AutoCad, um webbrowser, etc.

A página **Servers** da paleta de componentes do Delphi tem os principais servidores de automação DCOM. Os controles **ActiveX** podem também ser desenvolvidos no Delphi.

"InterBase Express">InterBase Express (IBX):

- Não é um mecanismo de banco de dados independente de servidor, mas um conjunto de componentes para acessar um servidor de banco de dados específico.
- Esse conjunto de componentes aplica-se a diferentes ambientes (Delphi, Oracle) e plataformas (Linux).
- Oferecem melhor desempenho e controle, à custa de flexibilidade e portabilidade.
- Transações: Cada operação de edição/postagem é considerada uma transação implícita, mas deve-se alterar esse comportamento manipulando as transações implicitamente. Deve-se evitar as transações que esperam entrada do usuário para serem concluídas, pois o usuário poderia ficar fora indefinidamente e a transação permaneceria ativa por um longo tempo, consumindo performance do sistema. Deve-se isolar cada transação corretamente, por meio do nível de isolamento Snapshot para relatórios e ReadCommitted para formulários interativos. Uma transação começa automaticamente quando você edita qualquer conjunto de dados ligados a ela. O comando CommitRetaining não abre uma nova transação, mas permite que a transação atual permaneça aberta.
- Componentes Ibx: **IbSQL**: executa instruções SQL que não retornam um conjunto de dados (instruções DDL ou instruções update e delete); **IbDataBaseInfo**: consultar estrutura e status do banco de dados; **IbSqlMonitor**: depurador do sistema (o SQL monitor do Delphi é específico para o BDE); **IbEvents**: recebe eventos enviados pelo servidor.

Midas (Middle-tier Distributed Applications Services):

- Serviços de aplicativos distribuídos de camada intermediária.
- Vide [DataSnap](#).

WebBroker:

Permite a geração páginas HTML com relatórios dinâmicos, a partir de bancos de dados, incluindo contadores de acesso, pesquisas interativas, etc.

WebSnap:

- Complementa a base fornecida pelo [WebBroker](#). A diferença básica é que, em vez de ter um único módulo de dados com múltiplas ações eventualmente conectadas aos componentes produtores, o WebSnap tem múltiplos módulos de dados, cada um correspondente a uma ação e tendo um componente produtor com um arquivo **HTML** anexado.
- Um aplicativo WebSnap é tecnicamente um programa [CGI](#), um [módulo ISAPI](#) ou [Apache](#).
- As vantagens em relação ao [WebBroker](#): permite que cada um de vários módulos Web corresponda a uma página, a integração de script no lado servidor, a **XLS** (Extensible Stylesheet Language – linguagem de folhas de estilo extensível) e a tecnologia [Internet Express](#). Além disso, existem muitos outros componentes prontos para manipular tarefas comuns, como login de usuários, gerenciamento de sessões, **permissões**, etc.
- Oferece um bom suporte à manipulação de conjuntos de dados: o **DataSetAdapter** conecta-se a um conjunto de dados, através de um **ClientDataSet** e exibe seus valores em um formulário ou tabela, por meio do editor visual do componente **AdapterPageProducer**.
- Além de visualização, permite **edição** dos dados de formulários.
- Permite relacionamentos mestre/detalhe.

ActiveX:

- Evolução da tecnologia VBX (16 Bits, Delphi 1): componentes da Microsoft para Visual Basic, quando passou a se chamar [OLE](#), ou **OCX** (que é a extensão usada).
- Os controles típicos Windows usam uma interface baseada em mensagens, enquanto os objetos de Automação e os controles ActiveX usam propriedades, métodos e eventos (como os objetos do Delphi).
- Mesmos componentes ActiveX podem ser usados em vários ambientes de desenvolvimento, como Delphi, Borland C++ Builder, etc.

dbExpress (DBX):

- Tecnologia de Acesso a Banco de Dados, sem depender do [BDE](#).
- É chamada **camada 'magra'** de acesso a banco de dados, substituindo o [BDE](#).
- Baseia-se no componente ClientDataSet (paleta **Data Access**), o qual tem a capacidade de salvar tabelas em arquivos locais – algo que a Borland tenta vender com o nome de [MyBase](#).
- Disponível para as plataformas Windows e Linux.
- Basicamente não exigem nenhuma configuração nas máquinas dos usuários finais.
- Limitada em termos de recursos: pode **acessar apenas servidores SQL (nada de arquivos locais)**, só se salvá-los localmente); **não tem recursos de cache** e oferece apenas **acesso unidirecional (só para consultas, não inclui, altera, exclui, etc.)**
- Indicada para produção de relatórios em **HTML** (só consultas).
- Para uso numa interface de edição de dados, deve ser usado juntamente com um componente **Provider** (cache e queries), que pode importar dados. É útil para arquiteturas cliente/servidor ou de múltiplas camadas.
- Acessa muitos mecanismos de bancos de dados diferentes: Interbase, Oracle, MySQL (Linux), Informix, Db2, Ms SQL Server, mas enfrenta alguns problemas de diferenças de dialeto SQL.

ADO (ActiveX Data Objects):

- Interface de alto nível da Microsoft para acesso a banco de dados.
- Implementado na tecnologia de acesso a dados [OLE DB](#) da Microsoft.
- Acessa banco de dados relacionais, não relacionais, sistemas de arquivos, e-mail e objetos corporativos personalizados.
- Independe de servidor de banco de dados, com suporte a servidores locais e SQL.
- Mecanismo pesado e configuração simplificada. O tamanho da instalação do [MDAC](#) (Microsoft Data Access Components) atualiza grandes partes do sistema operacional.
- Compatibilidade limitada entre versões: obriga o usuário a atualizar os computadores para a mesma versão que utilizou para desenvolver o programa.
- Apresenta vantagens para quem usa Access ou SQL Server, oferecendo melhor qualidade de drivers que os dos provedores de [OLE DB](#) normais.
- Não serve para desenvolvimento independente de plataforma: não disponível no Linux ou em outros sistemas operacionais.

- O pacote ADO foi denominado pela Borland de **dbGo**, e seus componentes principais são: **ADOConnection** (para conexão ao banco de dados), **ADOCommand** (para execução de comandos SQL) e **ADODataset** (para execução de requisições que retornam um conjunto de resultados). Além desses, existem 3 componentes de compatibilidade: **ADOTable**, **ADOQuery** e **ADOSToredProc**, que podem ser usados para portar aplicativos baseados no **BDE**. E ainda há o componente **RDSConnection**, que permite acessar dados em aplicativos multicamadas (multitier) remotos.
- A Microsoft está substituindo o ADO pela sua versão **.NET**, a qual se baseia nos mesmos conceitos básicos. Assim, o uso do ADO pode fornecer uma boa preparação para quem caminha na direção de aplicativos **.NET** nativos (embora a Borland também planeje portar o **dbExpress** para aquela plataforma).

ADO .NET:

- Evolução significativa da ADO, para a arquitetura **.NET**, voltada a desenvolvimento para a Web.
- Supre as falhas do **COM**, que é a base da **ADO** como por exemplo:
 - a) o **COM** é inaceitável como mecanismo de transporte;
 - b) o **COM** só roda em ambiente Windows;
 - c) o **COM** não penetra firewalls corporativos.
- A solução para os problemas da **ADO** veio com o uso de **XML**.

ASP .NET:

Adoção da arquitetura .NET, voltada a desenvolvimento para a Web.

MyBase (vide [dbExpress](#)):

- Apropriado para aplicativo de banco de dados monousuário local.
- Para executar qualquer aplicativo que use ClientDataSet, você precisa implementar a biblioteca midas.dll, (vide [Midas](#)) que pode apresentar conflitos de versões.
- A partir do Delphi 6, a midas.dll pode ser ligada diretamente ao executável pela unidade MidasLib (vide Midas).
- O componente ClientDataSet suporta os formatos **CDS** (padrão) e XML (Extended Markup Language).

MDAC (Microsoft Data Access Components): componentes de acesso a dados da Microsoft

- É um guarda-chuva para as tecnologias de bancos de dados da Microsoft e inclui **ADO**, **OLE DB**, **ODBC** e **RDS** (Remote Data Services – serviços de dados remotos).
- O MDAC é lançado independentemente e disponível para download gratuito e distribuição praticamente gratuita.
- O Delphi 7 vem com o MDAC 2.6.
- O MDAC SDK faz parte do Platform SDK do Windows, que pode ser baixado gratuitamente.
- Estão inclusos no MDAC os seguintes provedores: Drivers **ODBC**, Jet 3.5 (Access 97), Jet 4.0 (Access e outros DB), SQL Server, Oracle, Serviços OLAP (Online Analytical Processing – processamento analítico on-line), CSV e Texto Simples.

XML (Extensible Markup Language – Linguagem de Marcação Extensível):

- É uma versão simplificada da **SGML**.
- É extensível porque permite marcas (tags) livres, em contraste com **HTML**, que tem tags prédefinidas.

Existem duas técnicas normais para gerenciar documentos **XML** em programas Delphi:

- a) **DOM** (Document Object Model – modelo de objetos documento): carrega o documento inteiro em uma árvore hierárquica de nós, permitindo leitura e manipulação.

b) **SAX** (Simple API for XML – API simples para XML): analisa o documento, chamando um evento para cada elemento desse, sem construir qualquer estrutura na memória.

BDE (Borland Database Engine):

- Acesso a bancos de dados locais: Paradox, dBase e servidores SQL, bem como tudo o que possa ser acessado por meio de drivers [ODBC](#).
- A Borland a considera obsoleta, tende a sair do ambiente Delphi.
- A paleta [BDE](#) do Delphi 7 ainda mantém os componentes Table, Query, StoredProc e outros do [BDE](#).

CORBA (Common Objects Request Broken Architecture): arquitetura de agente de requisições de objetos comuns.

- Até o Delphi 6, era um mecanismo de transporte para os aplicativos [DataSnap](#). Devido a questões de compatibilidade com as versões mais recentes da solução CORBA da Visibroker da Borland, esse recurso foi descontinuado no Delphi 7.
- Essa arquitetura foi redirecionada através da transformação de pacotes de dados em [XML](#) e entrega dos mesmos a um navegador Web (cria-se uma camada extra: o servidor Web recebe os dados da camada intermediária e os distribui para o cliente). É a arquitetura [Internet Express](#).

Internet Express: é parte da plataforma [WebSnap](#).

- Contém um componente-cliente chamado **XMLBroker**, que pode ser usado em lugar de um conjunto de dados-cliente para recuperar dados de um programa [DataSnap](#) de camada intermediária e torná-los disponíveis para um tipo específico de produtor de página, denominado **InetPageProducer**.
- É uma arquitetura de quatro camadas: **servidor SQL**, servidor de aplicativos (o servidor [DataSnap](#)), **servidor Web** com um aplicativo personalizado e o **navegador Web**.
- É uma tecnologia para a construção de clientes baseados em navegador, a qual permite enviar o conjunto de dados inteiro para o computador cliente, com o código **HTML** e algum código **JavaScript** para manipular o arquivo [XML](#) e exibi-lo na interface com o usuário definida pelo código HTML. É o código **JavaScript** que permite ao navegador exibir os dados e manipulá-los.

CGI (Common Gateway Interface):

- Protocolo para a programação de servidores Web, permitindo a geração de páginas **HTML** dinâmicas.
- É independente de plataforma, permitindo ao navegador enviar e requisitar dados. Quando o servidor detecta uma requisição de página do aplicativo CGI, ele lança o aplicativo, passa dados de linha de comando da requisição de página para o aplicativo e, em seguida, envia a saída-padrão do aplicativo de volta para o computador-cliente.
- Os aplicativos CGI podem ser gerados através de várias ferramentas e linguagens, inclusive o Delphi.
- É limitado aos sistemas baseados no processador Intel.

APIs:

- Alternativas ao CGI são o uso das **APIs** de servidor Web: **ISAPI** (Internet Server API, da Microsoft); **NSAPI** (Netscape Server API) ou API **Apache**. Essas APIs permitem escrever uma biblioteca que o servidor carrega em seu próprio espaço de endereçamento e mantém na memória. Após carregar a DLL, **o servidor API pode executar requisições individuais por meios de linhas de execução dentro do processo principal, em vez de iniciar um novo arquivo EXE para cada requisição, como no caso de aplicativos CGI**, sendo ainda mais rápidas que estes.
- No Delphi, essas tecnologias são implementadas através do [WebBroker](#).

