



Borland

Delphi



FACULDADE DE TECNOLOGIA DO ESTADO DE SÃO PAULO

As **Faculdades de Tecnologia do Estado de São Paulo (FATEC)** são instituições públicas de ensino superior pertencentes ao Centro Estadual de Educação Tecnológica Paula Souza (CEETEPS), autarquia de regime especial vinculada e associada, por força de lei estadual, à Universidade Estadual Paulista Júlio de Mesquita Filho (UNESP).

As FATEC's oferecem Cursos Superiores de Tecnologia em diversas áreas do conhecimento, com exceção as unidades de São Caetano do Sul, Ourinhos, Carapicuíba e Americana que oferecem as opções de Bacharelado e Licenciatura no curso de Análise de Sistemas e Tecnologia da Informação

HISTÓRICO

O marco inicial da trajetória da FATEC-SP foi a fundação, em 1969, do Centro Estadual de Educação Tecnológica de São Paulo pelo então Governador do estado Abreu Sodré, que tinha por objetivo a formação de técnicos de nível superior para atender a crescente demanda de profissionais de nível universitário. O CEETEPS foi instalado na praça Coronel Fernando Prestes no centro da capital paulista, utilizando o antigo campus da Escola Politécnica da USP.

Os cursos ministrados pela FATEC-SP são os mais antigos, tendo sido ministrados ininterruptamente desde de 1969. Naquele ano, foi fundada na cidade de Sorocaba a Faculdade de Tecnologia de Sorocaba, também com os mesmos objetivos. No ano de 1970, foram criados os Cursos Superiores de Tecnologia em Construção Civil, nas modalidades: Edifícios, Obras Hidráulicas e Movimento de Terra e Pavimentação, bem como os Cursos Superiores de Tecnologia em Mecânica, nas modalidades: Oficinas e Desenhista/Projetista. Em 1973, por meio de decreto estadual, o Centro passou a ser denominado Centro Estadual de Educação Tecnológica Paula Souza e seus cursos passavam a constituir a Faculdade de Tecnologia de São Paulo. Dessa forma, o CEETEPS se tornou o mantenedor de duas FATEC's, uma na cidade de São Paulo e a outra em Sorocaba. Em 1974, foi criado o curso superior de Tecnologia em Processamento de Dados.

Em 1976, o governo estadual, reuniu todos os seus estabelecimentos isolados de ensino superior por meio de lei, para constituir a Universidade Estadual Paulista Júlio de Mesquita Filho - UNESP. Como o CEETEPS não era uma instituição de ensino e sim o mantenedor de duas unidades universitárias, a lei que criou a UNESP estabeleceu que o CEETEPS integraria o conjunto universitário da nova universidade na condição de autarquia de regime especial, vinculado e associado à ela.

CURSO: TECNOLOGIA EM PROCESSAMENTO DE DADOS

Perfil Profissional do Tecnólogo em Processamento de Dados

Atuando nas áreas de Análise de Sistemas, é capaz de desenvolver e administrar projetos de Sistemas de Informação. Conhece características de equipamentos, estando apto a fazer estudos de necessidades e viabilidade técnico-econômica de equipamentos e Sistemas de Informação. Conhece administração de empresas, podendo vir a ser analista de organização ou mesmo assumir todos os níveis de chefias na área de Informática. Pode dedicar-se ao ensino e à pesquisa, dentro do seu campo profissional. Conhece linguagens de programação comerciais e científicas, podendo realizar manutenção de sistemas de computadores e, ainda, como analista de software, pesquisar a otimização e aplicação de sistemas. Pode atuar no dimensionamento, implantação e gerência de redes de teleprocessamento. Conhece estruturas de dados e pesquisas a arquivos, estando habilitado a atuar em áreas de Administração de Dados e Administração de Banco de Dados.

ÍNDICE

FACULDADE DE TECNOLOGIA DO ESTADO DE SÃO PAULO	2
HISTÓRICO	2
CURSO: TECNOLOGIA EM PROCESSAMENTO DE DADOS	3
ÍNDICE.....	4
CAPÍTULO 01 - O AMBIENTE DE DESENVOLVIMENTO DO DELPHI.....	10
1.1. A JANELA PRINCIPAL	10
1.2. A PALETA DE COMPONENTES.....	11
1.3. A BARRA DE FERRAMENTAS.....	11
1.4. A BARRA DE MENUS E OS MENUS DE ATALHO.....	12
1.5. OS FORMULÁRIOS	13
1.6. O EDITOR DE CÓDIGO	13
1.7. O CODE EXPLORER	14
1.8. O OBJECT INSPECTOR.....	15
1.9. NAVEGANDO PELO AMBIENTE	15
CAPÍTULO 02 - A ESTRUTURA DE UM APLICATIVO NO DELPHI	17
2.1. FORMULÁRIOS E UNITS.....	17
2.2. TRABALHANDO COM PROJETOS	17
2.3. O QUE É O ARQUIVO DE PROJETO	17
2.4. VISUALIZANDO E ENTENDENDO O ARQUIVO DE PROJETO	18
2.5. ALTERANDO O ARQUIVO DE PROJETO	19
2.6. USANDO O PROJECT MANAGER.....	19
2.7. USANDO PROJECT GROUPS	20
2.8. SALVANDO UM PROJECT GROUP.....	21
2.9. ADICIONANDO E REMOVENDO PROJETOS EM UM PROJECT GROUP	21
2.10. ATIVANDO UM PROJETO.....	22
2.11. COMPILANDO E EXECUTANDO PROJETOS	22
2.12. VERIFICANDO APENAS A SINTAXE	22
2.13. COMPILANDO PROJETOS	22
2.14. COMPILANDO PROJECT GROUPS.....	23
2.15. MOSTRANDO INFORMAÇÕES SOBRE A COMPILAÇÃO	23
2.16. EXECUTANDO O APLICATIVO GERADO.....	23
2.17. ARQUIVOS GERADOS PELO DELPHI	24
2.17.1. Arquivos .DFM	24
2.17.2. Arquivos .DCU.....	24
2.17.3. Arquivos .RES, .DOF e .CFG	25
CAPÍTULO 03 - COMPONENTES: CONCEITOS BÁSICOS.....	26
3.1. TIPOS DE COMPONENTES.....	26
3.2. COMPONENTES VISUAIS	26
3.3. PROPRIEDADES.....	26
3.4. ALTERANDO PROPRIEDADES EM TEMPO DE DESENVOLVIMENTO	26
3.5. ALTERANDO PROPRIEDADES EM TEMPO DE EXECUÇÃO.....	29
3.6. EVENTOS	29
3.7. ASSOCIANDO CÓDIGO A UM EVENTO	29
3.8. APAGANDO EVENTOS	30
3.9. MÉTODOS.....	30
3.10. ADICIONANDO COMPONENTES A UM FORMULÁRIO	31
3.11. MANIPULANDO COMPONENTES.....	32
3.12. SELECIONANDO E REDIMENSIONANDO COMPONENTES	32

3.13.	DUPPLICANDO E TRANSFERINDO COMPONENTES.....	34
3.14.	ALINHANDO, TRAVANDO E SOBREPONDO COMPONENTES	34
3.15.	ALINHANDO PELA GRADE.....	36
3.16.	TRAVANDO COMPONENTES	36
3.17.	CONTROLANDO A SOBREPOSIÇÃO DE COMPONENTES	37
CAPÍTULO 04 - FORMULÁRIOS E CAIXAS DE DIÁLOGO		38
4.1.	ADICIONANDO FORMULÁRIOS	38
4.2.	ESPECIFICANDO O FORMULÁRIO PRINCIPAL	39
4.3.	LIGANDO FORMULÁRIOS	39
4.4.	PROPRIEDADES DOS FORMULÁRIOS	41
4.5.	EVENTOS DOS FORMULÁRIOS	43
4.6.	MÉTODOS DOS FORMULÁRIOS.....	43
4.7.	MOSTRANDO FORMULÁRIOS.....	44
4.8.	A PROPRIEDADE <i>MODALRESULT</i>	45
4.9.	USANDO O TECLADO COM FORMULÁRIOS	45
4.10.	ALTERANDO A ORDEM DE TABULAÇÃO	45
4.11.	EVITANDO QUE UM COMPONENTE RECEBA O FOCO.....	46
4.12.	CONTROLANDO A CRIAÇÃO DOS FORMULÁRIOS	46
4.13.	IMPRIMINDO FORMULÁRIOS	48
4.14.	CAIXAS DE DIÁLOGO PREDEFINIDAS.....	48
4.15.	MESSAGE BOXES (CAIXAS DE MENSAGEM).....	49
4.16.	USANDO O COMANDO <i>SHOWMESSAGE</i>	49
4.17.	USANDO A COMANDO <i>MESSAGEDLG</i>	49
4.18.	USANDO A FUNÇÃO <i>INPUTBOX</i>	51
CAPÍTULO 05 - TRABALHANDO COM MENUS		53
5.1.	ADICIONANDO MENUS E ABRINDO O MENU DESIGNER.....	53
5.2.	CONSTRUINDO MENUS PRINCIPAIS	53
5.3.	INSERINDO, MOVENDO E APAGANDO COMANDOS	54
5.4.	ADICIONANDO SEPARADORES.....	55
5.5.	DEFININDO "TECLAS DE ACELERAÇÃO" E TECLAS DE ATALHO	55
5.6.	CRIANDO SUBMENUS.....	56
5.7.	CONSTRUINDO MENUS <i>POPUP</i>	57
5.8.	ASSOCIANDO CÓDIGO AOS COMANDOS DE UM MENU	57
5.9.	PROPRIEDADES IMPORTANTES DOS MENUS E COMANDOS	57
5.10.	TRABALHANDO COM VÁRIOS MENUS	58
CAPÍTULO 06 - COMPONENTES VISUAIS COMUNS		59
6.1.	PROPRIEDADES COMUNS.....	59
6.2.	EVENTOS COMUNS.....	61
6.3.	COMPONENTE <i>BUTTON</i>	61
6.3.1.	<i>Propriedades</i>	61
6.3.2.	<i>Eventos</i>	61
6.4.	COMPONENTE <i>EDIT</i>	62
6.4.1.	<i>Propriedades</i>	62
6.4.2.	<i>Eventos</i>	63
6.5.	COMPONENTE <i>LABEL</i>	63
6.5.1.	<i>Propriedades</i>	63
6.5.2.	<i>Eventos</i>	63
6.6.	COMPONENTE <i>MEMO</i>	63
6.6.1.	<i>Propriedades</i>	64
6.6.2.	<i>A propriedade Lines</i>	65
6.6.3.	<i>Eventos</i>	66
6.7.	COMPONENTE <i>LISTBOX</i>	66
6.7.1.	<i>Propriedades</i>	67
6.7.2.	<i>Eventos</i>	67

6.8.	COMPONENTE <i>COMBOBOX</i>	68
6.8.1.	<i>Propriedades</i>	68
6.8.2.	<i>Eventos</i>	69
6.9.	COMPONENTE <i>CHECKBOX</i>	69
6.9.1.	<i>Propriedades</i>	69
6.9.2.	<i>Eventos</i>	69
6.10.	COMPONENTE <i>RADIOBUTTON</i>	70
6.11.	COMPONENTE <i>RADIOGROUP</i>	70
6.12.	COMPONENTE <i>GROUPBOX</i>	71
6.13.	COMPONENTE <i>PANEL</i>	71
6.14.	COMPONENTE <i>BITBTN</i>	72
CAPÍTULO 07 - A LINGUAGEM OBJECT PASCAL.....		74
7.1.	USANDO O COMANDO DE ATRIBUIÇÃO	74
7.2.	ENTENDENDO IDENTIFICADORES	74
7.3.	DECLARANDO VARIÁVEIS	75
7.4.	TIPOS DE VARIÁVEIS	76
7.5.	DECLARANDO CONSTANTES	76
7.6.	TIPOS ESTRUTURADOS	77
7.7.	TIPOS ENUMERADOS	77
7.8.	TIPOS DE INTERVALO (<i>SUBRANGE</i>)	78
7.9.	ARRAYS	79
7.10.	ARRAYS MULTIDIMENSIONAIS.....	80
7.11.	STRINGS	80
7.12.	REGISTROS (<i>RECORDS</i>)	81
7.13.	CONTROLE DE FLUXO.....	82
7.14.	USANDO BLOCOS.....	82
7.15.	IF-THEN-ELSE	83
7.16.	USANDO <i>ELSE</i>	83
7.17.	USANDO BLOCOS COM O COMANDO <i>IF</i>	83
7.18.	ANINHANDO COMANDOS <i>IF</i>	84
7.19.	A ESTRUTURA <i>CASE</i>	85
7.20.	USANDO LOOPS.....	86
7.21.	O LOOP <i>FOR</i>	86
7.22.	O LOOP <i>WHILE</i>	88
7.23.	O LOOP <i>REPEAT</i>	89
7.24.	O COMANDO <i>BREAK</i>	90
7.25.	O COMANDO <i>CONTINUE</i>	91
7.26.	PROCEDURES E FUNCTIONS.....	91
7.27.	SINTAXE DAS PROCEDURES E FUNCTIONS	92
7.28.	ENTENDENDO PARÂMETROS	93
7.29.	DEFININDO O VALOR DE RETORNO DE UMA FUNCTION.....	93
7.30.	CHAMANDO PROCEDURES E FUNCTIONS	94
7.31.	ONDE CRIAR PROCEDURES E FUNCTIONS	94
7.32.	TRABALHANDO COM EXCEÇÕES	95
7.33.	ENTENDENDO O CÓDIGO DAS UNITS	97
7.34.	A ESTRUTURA BÁSICA DE UMA UNIT	97
7.34.1.	<i>Parte Interface</i>	98
7.34.2.	<i>Parte Implementation</i>	98
7.34.3.	<i>Parte Initialization</i>	98
7.34.4.	<i>Parte finalization</i>	99
7.35.	ROTINAS ÚTEIS	99
7.36.	ROTINAS PARA MANIPULAÇÃO DE STRINGS.....	99
7.37.	FUNÇÕES DE CONVERSÃO DE TIPO	100
CAPÍTULO 08 - O EDITOR DE CÓDIGO		101
8.1.	VISUALIZANDO ARQUIVOS	101

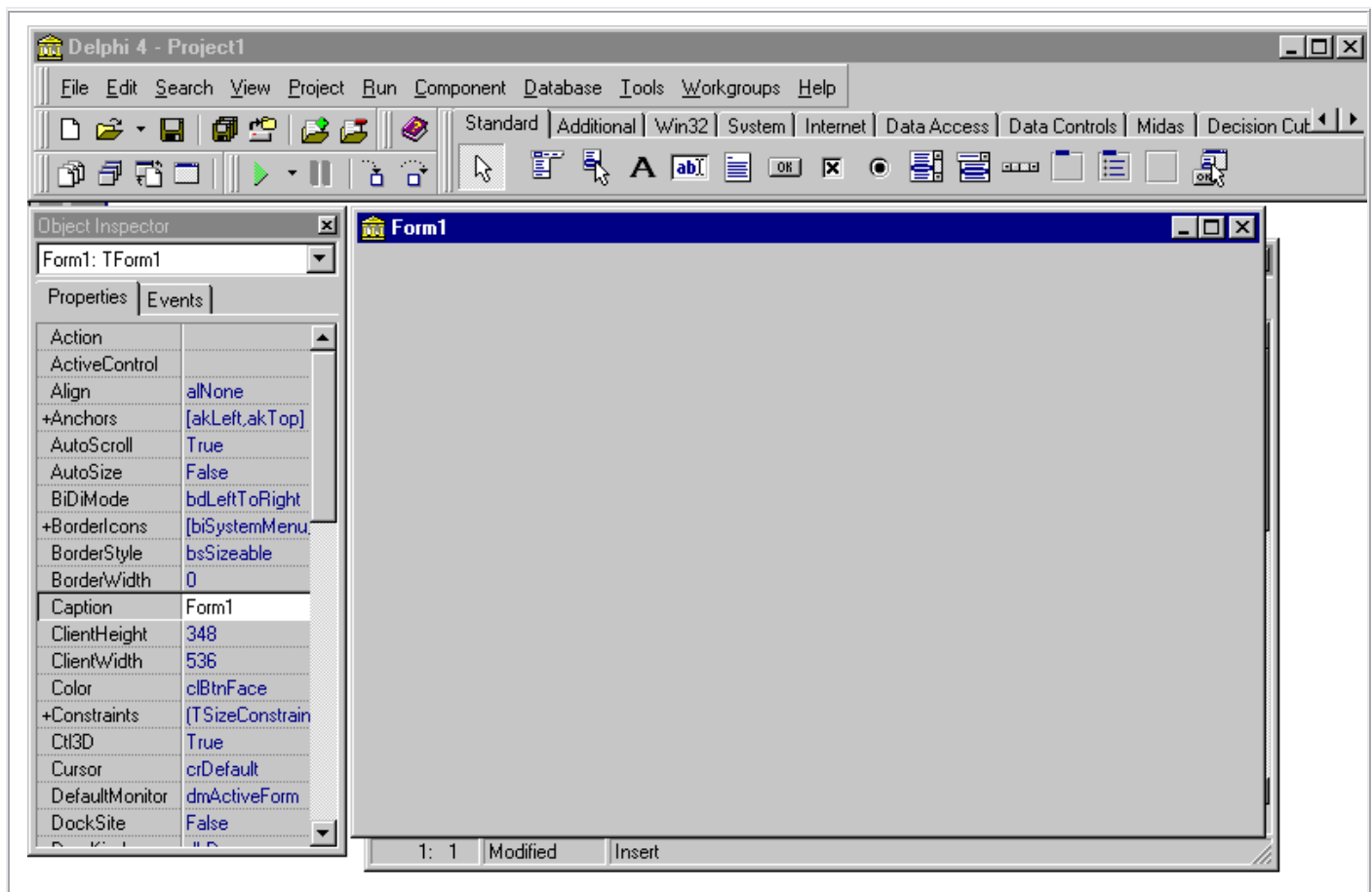
8.2.	TÉCNICAS DE NAVEGAÇÃO	103
8.3.	USANDO BOOKMARKS	103
8.4.	USANDO O RECURSO <i>CODE BROWSER</i>	104
8.5.	LOCALIZANDO E SUBSTITUINDO	105
8.6.	LOCALIZANDO TEXTOS NO ARQUIVO ATUAL	105
8.7.	LOCALIZANDO TEXTOS EM VÁRIOS ARQUIVOS	106
8.8.	OUTRAS TÉCNICAS PARA LOCALIZAÇÃO	107
8.9.	SUBSTITUINDO TEXTOS	107
8.10.	CONFIGURANDO O EDITOR DE CÓDIGO	108
8.11.	OPÇÕES IMPORTANTES DA PÁGINA <i>DISPLAY</i>	109
8.12.	OPÇÕES IMPORTANTES DA PÁGINA <i>COLORS</i>	109
8.13.	USANDO OS RECURSOS <i>CODE INSIGHT</i>	111
8.13.1.	<i>Code Completion</i>	111
8.13.2.	<i>Code Parameters</i>	111
8.13.3.	<i>Code Templates</i>	112
8.14.	CONFIGURANDO OS RECURSOS DO <i>CODE INSIGHT</i>	114
CAPÍTULO 09 - O DEPURADOR INTEGRADO		116
9.1.	ERROS E TIPOS DE ERROS	116
9.2.	UTILIDADES DO DEPURADOR	116
9.3.	EXECUTANDO UM PROGRAMA ATÉ A POSIÇÃO DO CURSOR	117
9.4.	EXECUTANDO UM PROGRAMA LINHA POR LINHA	118
9.5.	O COMANDO <i>TRACE INTO</i>	118
9.6.	O COMANDO <i>STEP OVER</i>	118
9.7.	USANDO OS DOIS COMANDOS	119
9.8.	INTERROMPENDO A EXECUÇÃO	119
9.9.	USANDO <i>BREAKPOINTS</i>	119
9.10.	DEFININDO <i>BREAKPOINTS</i>	120
9.11.	MOSTRANDO OS <i>BREAKPOINTS</i> DEFINIDOS	120
9.12.	VERIFICANDO VARIÁVEIS E EXPRESSÕES	121
9.13.	TRABALHANDO COM <i>WATCHES</i>	121
9.14.	AVALIANDO E MODIFICANDO EXPRESSÕES	122
CAPÍTULO 10 - TRABALHANDO COM BANCOS DE DADOS NO DELPHI: UMA VISÃO GERAL		124
10.1.	A ARQUITETURA DE ACESSO A BANCOS DE DADOS	124
10.2.	COMPONENTES BÁSICOS	125
10.3.	A PÁGINA <i>DATA ACCESS</i>	125
10.4.	A PÁGINA <i>DATA CONTROLS</i>	126
10.5.	ACESSANDO BANCOS DE DADOS: UMA INTRODUÇÃO	127
10.6.	UM EXEMPLO TÍPICO	127
10.7.	USANDO O <i>DATABASE DESKTOP</i>	128
10.8.	DEFININDO UM <i>ALIAS</i>	129
10.9.	ALTERANDO O DIRETÓRIO DE TRABALHO	131
10.10.	CRIANDO TABELAS	131
10.11.	ADICIONANDO DADOS A UMA TABELA	133
10.12.	ALTERANDO A ESTRUTURA DE UMA TABELA	134
CAPÍTULO 11 - TRABALHANDO COM DATASETS		136
11.1.	ABRINDO E FECHANDO <i>DATASETS</i>	136
11.2.	ESTADOS DE UM <i>DATASET</i>	136
11.3.	NAVEGANDO EM UM <i>DATASET</i>	137
11.4.	MODIFICANDO <i>DATASETS</i>	140
11.5.	MODIFICANDO CAMPOS	140
11.6.	ADICIONANDO REGISTROS	141
11.7.	APAGANDO REGISTROS	142
11.8.	CONFIRMANDO E CANCELANDO MUDANÇAS	142
11.9.	INSERINDO E MODIFICANDO REGISTROS INTEIROS	143

11.10.	LOCALIZANDO REGISTROS COM <i>LOCATE</i>	145
11.11.	FILTRANDO <i>DATASETS</i>	146
11.12.	USANDO <i>FILTER</i> E <i>FILTERED</i>	147
11.13.	USANDO O EVENTO <i>ONFILTERRECORD</i>	147
11.14.	EVENTOS DOS <i>DATASETS</i>	147
11.15.	CONTROLANDO A ATUALIZAÇÃO DE COMPONENTES.....	148
CAPÍTULO 12 - COMPONENTES <i>DATASOURCE</i> E <i>TABLE</i>		150
12.1.	USANDO O COMPONENTE <i>DATASOURCE</i>	150
12.2.	PROPRIEDADES DO COMPONENTE <i>DATASOURCE</i>	150
12.3.	EVENTOS DO COMPONENTE <i>DATASOURCE</i>	150
12.4.	USANDO O COMPONENTE <i>TABLE</i>	151
12.5.	CONECTANDO-SE A UMA TABELA DE BANCO DE DADOS.....	151
12.6.	CONTROLANDO O ACESSO A UMA TABELA	151
12.7.	TRABALHANDO COM <i>RANGES</i>	152
CAPÍTULO 13 - COMPONENTES <i>TFIELD</i>.....		154
13.1.	CRIANDO CAMPOS PERSISTENTES	154
13.2.	TIPOS DE CAMPOS PERSISTENTES.....	156
13.3.	CAMPOS CALCULADOS	157
13.4.	CAMPOS LOOKUP	159
13.5.	PROPRIEDADES DOS COMPONENTES <i>TFIELD</i>	161
CAPÍTULO 14 - O COMPONENTE <i>BATCHMOVE</i>		162
14.1.	CONFIGURAÇÃO BÁSICA	162
14.2.	MODOS DE OPERAÇÃO	163
14.3.	EXECUTANDO A OPERAÇÃO DE TRANSFERÊNCIA.....	163
14.4.	LIDANDO COM ERROS NA TRANSFERÊNCIA	164
CAPÍTULO 15 - COMPONENTES <i>DATA CONTROLS</i>.....		166
15.1.	PROPRIEDADES <i>DATASOURCE</i> E <i>DATAFIELD</i>	166
15.2.	OUTRAS PROPRIEDADES E RECURSOS COMUNS.....	166
15.3.	COMPONENTE <i>DBEDIT</i>	167
15.4.	COMPONENTE <i>DBTEXT</i>	167
15.5.	COMPONENTE <i>DBMEMO</i>	167
15.6.	COMPONENTE <i>DBCHECKBOX</i>	168
15.7.	COMPONENTE <i>DBRADIOGROUP</i>	168
15.8.	COMPONENTE <i>DBIMAGE</i>	169
15.9.	COMPONENTES <i>DBLISTBOX</i> E <i>DCOMBOBOX</i>	169
15.10.	COMPONENTES <i>DBLOOKUPLIST</i> E <i>DBLOOKUPCOMBO</i>	170
15.11.	COMPONENTE <i>DBNAVIGATOR</i>	170
15.12.	COMPONENTE <i>DBGRIID</i>	172
CAPÍTULO 16 - O COMPONENTE <i>QUERY</i>.....		174
16.1.	CONFIGURANDO UM COMPONENTE <i>QUERY</i>	174
16.2.	ESPECIFICANDO A CONSULTA <i>SQL</i> A SER EXECUTADA	174
16.3.	TRABALHANDO COM PARÂMETROS.....	176
16.4.	EXECUTANDO CONSULTAS	177
CAPÍTULO 17 - A LINGUAGEM <i>LOCAL SQL</i>		178
17.1.	AS DUAS PARTES DE <i>SQL</i>	178
17.2.	O COMANDO <i>SELECT</i>	179
17.3.	USANDO <i>IN</i> E <i>BETWEEN</i>	180
17.4.	USANDO <i>LIKE</i> E CARACTERES "CURINGA"	180
17.5.	USANDO FUNÇÕES DE AGREGAÇÃO	181
17.6.	O COMANDO <i>INSERT</i>	181
17.7.	O COMANDO <i>UPDATE</i>	182

17.8.	O COMANDO DELETE.....	182
17.9.	A LINGUAGEM DE DEFINIÇÃO DE DADOS	182
17.10.	O COMANDO CREATE TABLE	182
17.11.	O COMANDO ALTER TABLE.....	183
17.12.	O COMANDO DROP TABLE	184
17.13.	O COMANDO CREATE INDEX	184
17.14.	O COMANDO DROP INDEX	184
CAPÍTULO 18 - COMPONENTES QUICKREPORT.....		185
18.1.	CRIANDO UM RELATÓRIO SIMPLES.....	185
18.2.	O COMPONENTE <i>QUICKREP</i>	188
18.3.	BANDAS E TIPOS DE BANDAS	188
18.4.	ALTERANDO A FORMATAÇÃO GERAL DO RELATÓRIO	190
18.5.	PROPRIEDADES COMUNS A TODAS AS BANDAS	191
18.6.	OUTROS COMPONENTES IMPORTANTES.....	191
18.7.	O COMPONENTE <i>QRLABEL</i>	191
18.8.	O COMPONENTE <i>QRDBTEXT</i>	192
18.9.	O COMPONENTE <i>QRSYSDATA</i>	192
18.10.	O COMPONENTE <i>QREXPR</i>	192
18.11.	CRIANDO RELATÓRIOS COM AGRUPAMENTO	193
18.12.	IMPRIMINDO E PRE-VISUALIZANDO RELATÓRIOS.....	195

CAPÍTULO 01 - O AMBIENTE DE DESENVOLVIMENTO DO DELPHI

O Delphi oferece um ambiente visual de desenvolvimento rico e versátil. Os que nunca usaram outro ambiente visual de desenvolvimento como o Delphi (Visual Basic, Visual C++, PowerBuilder, etc.) podem estranhar inicialmente a organização visual do ambiente. Diferentemente de aplicativos comuns, como o Word e o Excel da Microsoft, o Delphi não é uma aplicativo MDI (com uma janela principal capaz de conter várias janelas secundárias). O ambiente do Delphi é composto por várias janelas independentes que podem ser sobrepostas e reorganizadas livremente na tela.



O ambiente de desenvolvimento do Delphi

Neste capítulo, apresentaremos as partes principais do ambiente de desenvolvimento do Delphi.

1.1. A Janela Principal

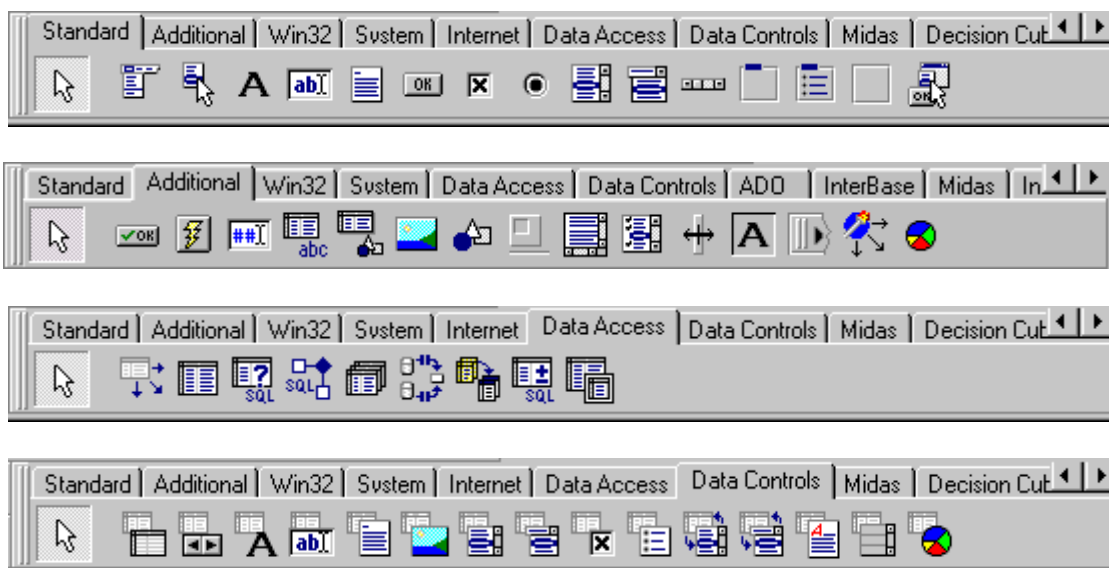
Na Janela Principal, localizada na parte de cima da tela, estão a Barra de Menus, a Paleta de Componentes e a Barra de Ferramentas (todas descritas a seguir). A janela principal é uma janela comum, apesar de ter uma aparência diferente do normal. Portanto é possível minimizá-la, maximizá-la e movê-la normalmente. Minimizar a janela principal, minimiza todas as outras janelas do Delphi – não é necessário minimizar ou fechar cada uma das outras janelas

primeiro. Quando a janela principal do Delphi é fechada, naturalmente, todo o ambiente (o Delphi inteiro) é fechado.

1.2. A Paleta de Componentes

A paleta de componentes é uma das partes mais utilizadas do Delphi. É a partir dessa paleta que se pode escolher componentes e adicioná-los a formulários. A paleta de componentes é dividida em várias *páginas*. Cada página contém um conjunto de componentes relacionados.

As páginas mais usadas são a *Standard* (com componentes básicos, como botões e caixas de texto), *Additional* (com alguns componentes especiais), e as páginas *Data Access* e *Data Controls* (para acesso e manipulação de bancos de dados). Essas páginas são ilustradas a seguir. Para passar de uma página para outra, clique na "aba" correspondente, na parte de cima da paleta de componentes.

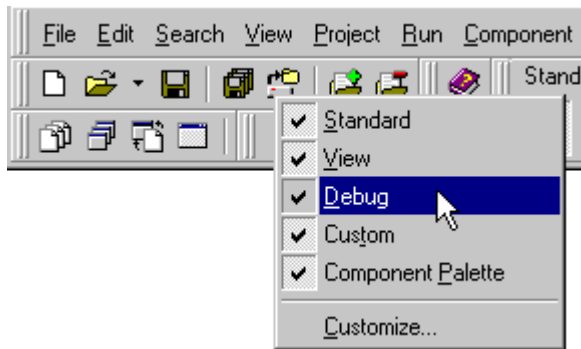


1.3. A Barra de Ferramentas



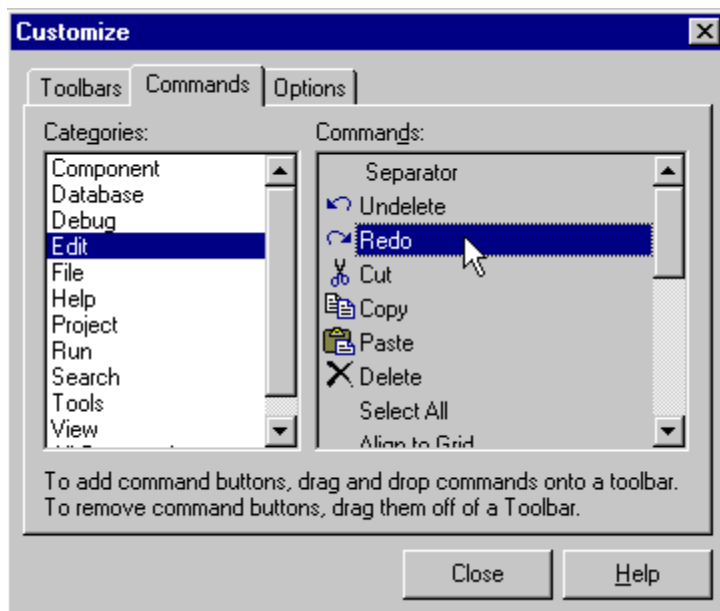
A Barra de Ferramentas oferece acesso rápido aos comandos mais comuns do Delphi, como criar formulários, abrir e salvar arquivos, ou compilar e executar aplicativos. A barra de ferramentas pode ser configurada de várias formas. Pode-se, por exemplo, adicionar botões para comandos chamados freqüentemente, retirar botões da barra ou reordená-los.

A barra de ferramentas é formada por várias partes que podem ser reorganizadas livremente. Para esconder/mostrar uma dessas partes, clique com o botão direito em qualquer local dentro da barra e desative/ative a opção correspondente (veja a figura abaixo).



Mostrando/escondendo partes da barra de ferramentas

Para adicionar um botão, clique na barra com o botão direito e escolha o comando **Customize**. Na caixa de diálogo que aparece, clique na aba "Commands". Aqui são mostrados todos os comandos do Delphi. Para criar um botão para um comando, arraste o comando para a barra.



Adicionando botões à barra de ferramentas

Ainda com a caixa de diálogo "Customize" aberta, você pode alterar a ordem dos botões arrastando-os para outras posições da barra. Para remover um botão, simplesmente arraste-o para fora da barra de ferramentas.

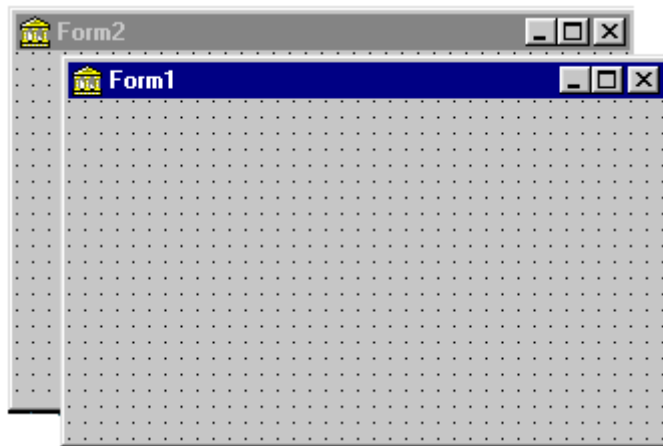
Finalmente, para fazer com que partes da barra de ferramentas voltem à sua configuração original (a mesma logo depois da instalação do Delphi), chame o comando **Customize**, clique na aba "Toolbar", selecione a parte que deseja alterar e clique no botão "Reset".

1.4. A Barra de Menus e os Menus de atalho

Na barra de menus estão todos os comandos que podem ser chamados no Delphi. A barra de menus é usada, normalmente, apenas para comandos pouco comuns. Os comandos mais comuns (como salvar um arquivo ou compilar um projeto, por exemplo), podem ser chamados mais rapidamente usando teclas de atalho, ou usando os botões da barra de ferramentas.

Os menus de atalho (chamados com o botão direito do mouse) são muito usados no Delphi. Há menus de atalho associados a praticamente todas as partes do ambiente de desenvolvimento.

1.5. Os Formulários

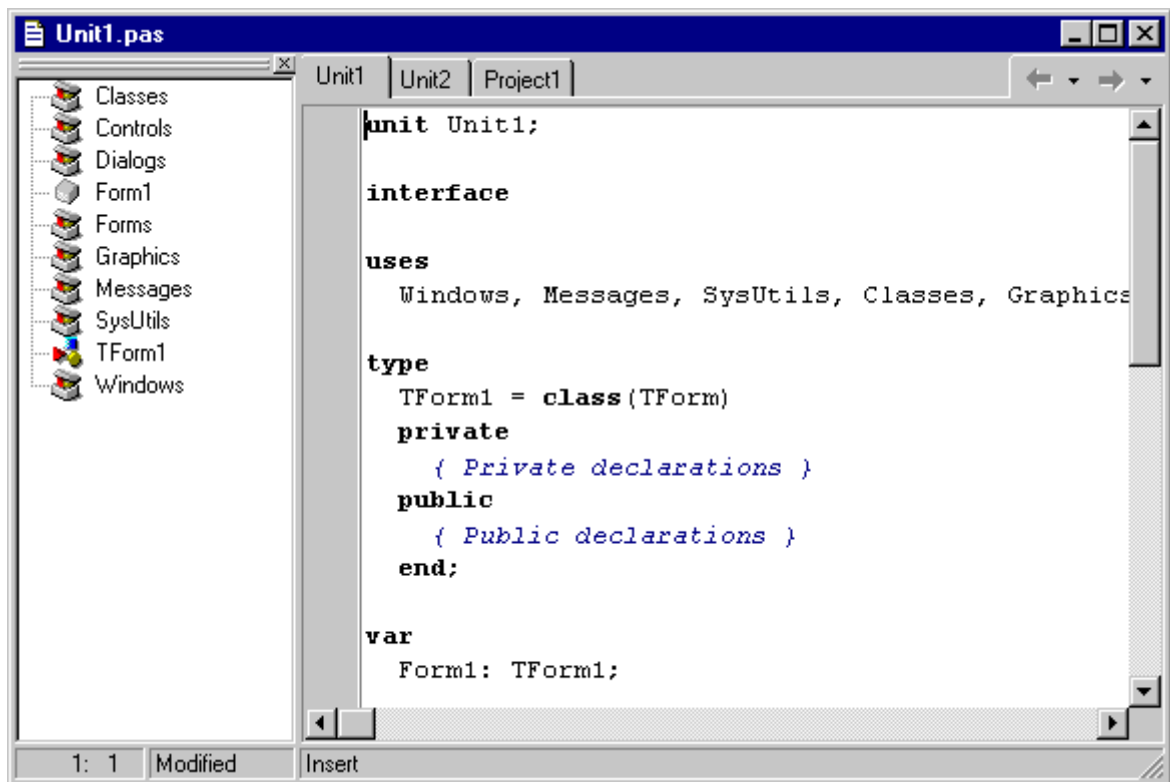


Os formulários (e o código associado a eles, as *Units*) constituem o coração de um aplicativo no Delphi. Quando você entra no Delphi, ou cria um novo projeto, um formulário vazio é criado automaticamente.

Os formulários são as partes visíveis de um aplicativo no Delphi. Neles são inseridos componentes como botões, listas, etc. Formulários podem ser usados com muitas funções diferentes em um aplicativo. Um formulário pode, por exemplo, ser desde a janela principal até uma pequena caixa de mensagem.

1.6. O Editor de Código

A todo formulário, é associado um programa – chamado de Unit, no Delphi – que controla como os componentes dos formulários reagem às ações do usuário (os eventos). As Units são exibidas no **Editor de Código** (*Code Editor*). O Editor de Código pode mostrar várias Units ao mesmo tempo. Pode-se mudar de uma Unit para outra, usando-se as abas na parte de cima da janela (veja a figura). Veremos detalhes sobre o Editor de Código mais adiante.



O Editor de Código e o Code Explorer

1.7. O Code Explorer

O Code Explorer, introduzido no Delphi 4, permite a navegação rápida entre partes de uma Unit aberta no Editor de Código. O Code Explorer é geralmente posicionado à esquerda do Editor de Código (veja a figura anterior). Nele é exibido um diagrama hierárquico, com todas as variáveis, métodos, tipos, classes e propriedades definidas na Unit.

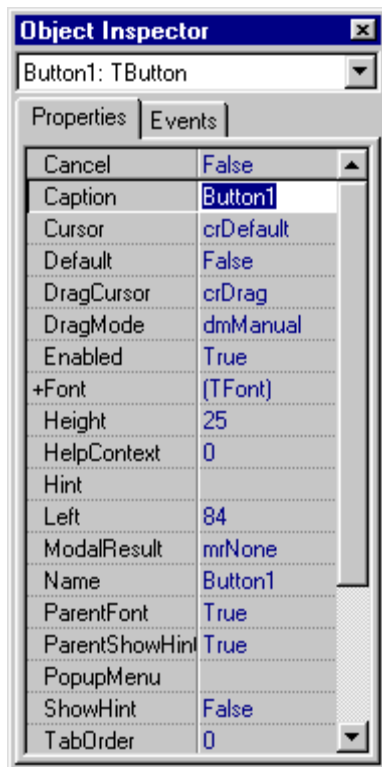
Para navegar para um item específico do código da Unit (a definição de um método ou variável, por exemplo), simplesmente clique duas vezes no nome do item. O Code Explorer pode ser usado também para renomear itens, ou para adicionar novos itens.

Para renomear um item, clique no nome do item com o botão direito e escolha o comando "Rename". Para adicionar um novo item clique com o botão direito em um item de mesmo tipo e escolha o comando "New".

Para esconder o Code Explorer, clique no pequeno "x" no seu canto direito. Para voltar a exibí-lo, use o comando **View | Code explorer** ou pressione CTRL+SHIFT+E. A combinação CTRL+SHIFT+E também pode ser usada para alternar entre o Code Explorer e o Editor de Código.

DICA: Como padrão, o Code Explorer é exibido sempre que o Delphi é inicializado, ou quando um projeto é aberto. Para que o Delphi não exiba o Code Explorer automaticamente, chame o comando **Edit|Environment options**, mude para a página "Explorer" e desative a opção "Automatically show Explorer".

1.8. O Object Inspector



O *Object Inspector* (tradução literal: "Inspetor de objetos") é usado para definir propriedades e eventos para os componentes. As listas de propriedades e eventos exibidas no Object Inspector mudam de acordo com o componente selecionado.

Outra função importante do Object Inspector é selecionar componentes por nome. Uma lista com o nome e o tipo de todos os componentes do formulário ativo são exibidos na parte de cima do Object Inspector. Para selecionar um componente, basta escolher seu nome nessa lista.

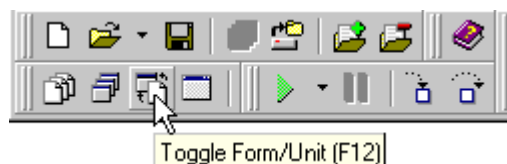
Para exibir/esconder o Object Inspector, use o comando **View | Object Inspector**, ou pressione **F11**.

Veremos como usar o Object inspector em detalhe, mais adiante no curso.

1.9. Navegando pelo ambiente

Durante o desenvolvimento de um aplicativo no Delphi, são criadas muitas janelas. O Delphi oferece vários recursos para navegar entre essas janelas. Veja os mais importantes a seguir.

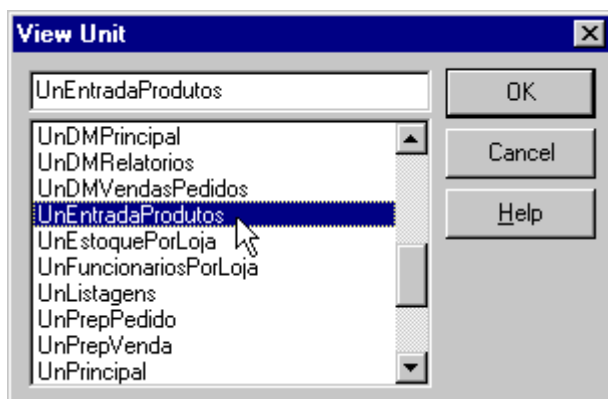
- Para alternar entre um formulário e a Unit associada, realize uma das seguintes ações:
- Pressione **F12**, ou
- Escolha o comando **View | Toggle Form /Unit**, ou
- Clique no botão **Toggle Form/Unit** na barra de ferramentas (veja figura a seguir).



- Para mostrar o *Object Inspector* ou trazê-lo para a frente das outras janelas:
- Pressione **F11**, ou
- Escolha o comando **View | Object Inspector**.
- Para trazer uma janela que não está visível para a frente:

1. Escolha o comando **View | Window List**

1. Clique duas vezes no nome da janela.
- **Para exibir uma lista das Units no projeto e exibir uma delas no Editor de Código:**
 - Escolha o comando **View | Units** (ou pressione **CTRL+F12**) e clique duas vezes no nome da Unit para exibi-la (veja a figura).



- **Para exibir uma lista dos formulários no projeto e exibir um deles:**
- Escolha o comando **View | Forms** (ou pressione **SHIFT+F12**) e clique duas vezes no nome do formulário.

CAPÍTULO 02 - A ESTRUTURA DE UM APLICATIVO NO DELPHI

2.1. Formulários e Units

Toda a parte visual e toda a interação com o usuário de um aplicativo criado no Delphi é baseada nos formulários. Para cada formulário adicionado a um aplicativo, o Delphi cria uma Unit associada.

Uma Unit (tradução literal: "Unidade") é um programa completo em Object Pascal, a linguagem de programação usada no Delphi. As Units contêm as declarações (o nome e o tipo) de cada componente no formulário, e o código para os eventos definidos para esses componentes.

NOTA* Apesar de ser comum criar aplicativos que só tenham Units associadas a formulários, pode-se criar **Units independentes**, com código que é compartilhado por vários formulários, ou até vários aplicativos.

Boa parte do código das Units é gerado e mantido automaticamente pelo Delphi. Quando um componente é adicionado, por exemplo, o Delphi acrescenta a sua declaração ao código da Unit. Quando um componente é removido, sua declaração é removida também.

Na maioria das vezes não se deve alterar diretamente o código gerado pelo Delphi. O código gerado pode ser alterado indiretamente, usando o Object Inspector e comandos de menu, ou manipulando os componentes diretamente no formulário.

2.2. Trabalhando com Projetos

Projeto é o nome dado pelo Delphi ao conjunto de todos os formulários, *Units* e outros objetos, usados na criação de um aplicativo. Em outras palavras, um projeto é o *código fonte* usado para gerar um aplicativo no Delphi.

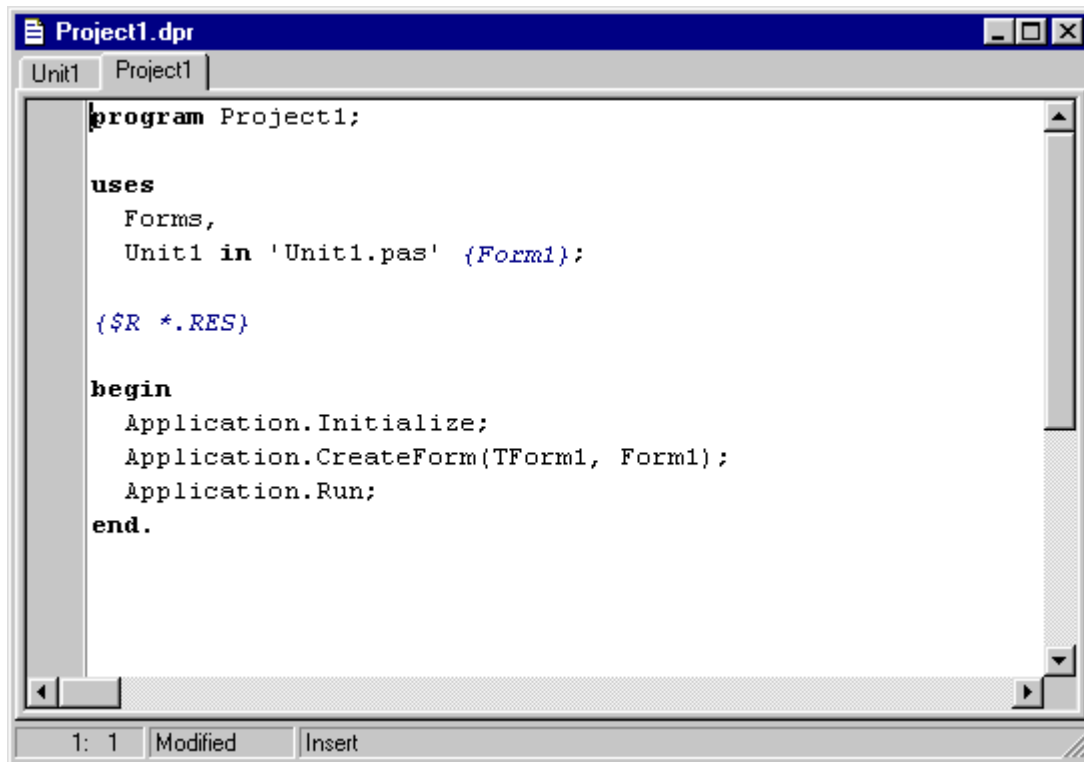
2.3. O que é o arquivo de projeto

Para cada projeto, o Delphi mantém um **arquivo de projeto**. O arquivo de projeto é um arquivo texto comum escrito em Object Pascal, que contém informações sobre todas as partes de um projeto. Um arquivo de projeto é criado automaticamente quando o Delphi é inicializado, ou quando um novo projeto é criado. Para cada projeto, por maior que seja, há apenas *um* arquivo de projeto.

Quando você compila um projeto no Delphi, o arquivo de projeto é o primeiro a ser compilado. A partir dele, o Delphi sabe exatamente quais formulários, Units e outros objetos devem ser compilados também, para gerar o arquivo final do aplicativo (em geral, um arquivo executável). Outra função importante do arquivo de projeto é controlar a *ordem de criação* dos formulários, como veremos depois.

2.4. Visualizando e entendendo o arquivo de projeto

O arquivo de projeto não é mostrado automaticamente no Editor de Código. Para visualizar o arquivo de projeto do projeto atual, use o comando **Project | View Source**.



A ilustração mostra o arquivo de projeto *mínimo*, aquele que é criado junto com um novo projeto "vazio". Na barra de título, na extremidade de cima da janela do Editor de Código, é mostrado o *nome* do arquivo de projeto. O nome mostrado na ilustração ("Project1.dpr") é gerado automaticamente pelo Delphi. A extensão .DPR vem de *Delphi Project*. (os próximos nomes gerados, caso sejam criados outros projetos, são "Project2.dpr", "Project3.dpr", e assim por diante). Lembre-se de alterar o nome do arquivo de projeto para um nome mais sugestivo na hora de salvar.

A primeira linha do arquivo de projeto contém o *nome do projeto*. Este é o nome que identifica o projeto para o Delphi, e é geralmente o nome do arquivo, retirada a extensão. A palavra "program" é usada para manter compatibilidade com versões anteriores do Turbo Pascal (o precursor do Delphi). Claramente, o uso da palavra "project" seria mais adequado no Delphi.

A segunda linha: "**Uses...**" (chamada de **cláusula uses**) indica quais Units são usadas pelo projeto. Aqui estão referenciados todos os formulários usados no projeto e suas Units correspondentes, além de Units que não estão associadas a nenhum formulário. Na figura é indicado o uso de uma única Unit, com o nome *Unit1*. A palavra-chave **in** é usada para indicar o arquivo que contém o código da Unit ("Unit1.pas"). Logo em seguida, entre chaves, é indicado o nome do formulário associado à Unit (no caso, *Form1*).

NOTA: Os nomes dados às Units e formulários são os mesmos nomes dos arquivos correspondentes. Por exemplo, se uma Unit for salva com o nome "CadastroProduto.pas", esta Unit será identificada pelo nome "CadastroProduto" no Delphi. No entanto, se forem usados

espaços no nome do arquivo, ou se houver letras acentuadas, os nomes não serão iguais. Espaços e letras acentuadas são simplesmente retirados (sem avisos) do nome do arquivo. Se, por exemplo, você chamou um formulário de "Relatório de preços.pas", o nome gerado pelo Delphi será "Relatriodepreos". Evite, portanto, usar acentos ou espaços nos nomes dos seus arquivos.

Voltando ao arquivo de projeto, a linha **{\$R ...}** indica o arquivo de recursos (*Resource file*) que está associado ao projeto. Esta linha nunca deve ser removida – sem ela seu aplicativo não será compilado corretamente. O arquivo de recursos contém o ícone do aplicativo, além de outras informações importantes. Este arquivo é gerado automaticamente pelo Delphi e raramente precisa ser alterado manualmente.

NOTA: as chaves indicam **comentários** no Delphi. A única exceção é o caso descrito acima. Uma chave seguida por um cifrão (**{ \$ }**) indica o início de uma **diretiva de compilação** - um comando especial para o compilador. Diretivas de compilação são um recurso avançado do Delphi que raramente é usado diretamente pelo programador.

A última parte do arquivo de projeto é o **bloco principal**, delimitado pelas palavras-chave **begin** e **end**. A primeira linha dentro do bloco principal, **Application.Initialize** realiza as tarefas de inicialização do aplicativo, como carregar ou ler arquivos de inicialização. O comando **Application.CreateForm** cria um formulário na memória, deixando-o pronto para a exibição na tela. Finalmente, o último comando do bloco principal, **Application.Run** executa o aplicativo.

2.5. Alterando o arquivo de projeto

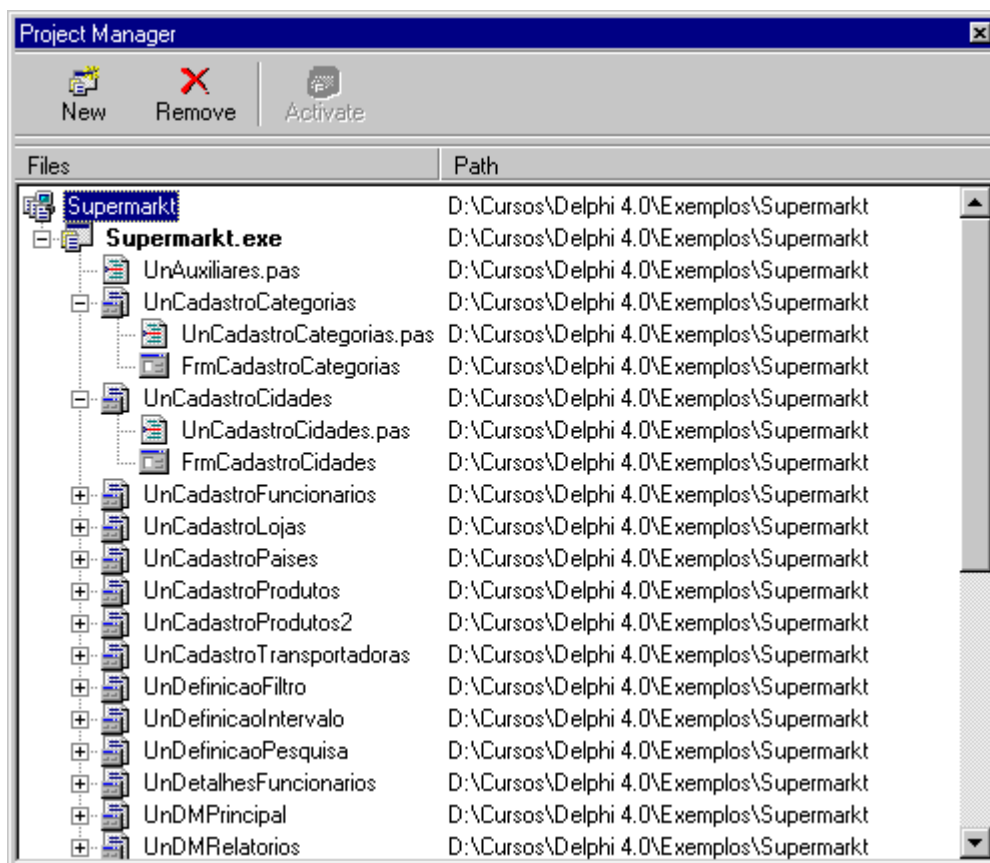
A não ser em casos especiais, o arquivo de projeto não deve ser alterado manualmente. Ele é atualizado automaticamente pelo Delphi quando são feitas alterações no projeto (quando um formulário é adicionado, por exemplo).

As alterações no arquivo de projeto, quando necessárias, devem ser feitas através do **Project Manager**, ou usando o comando **Project | Options**. Há um grande número de detalhes e opções que podem ser definidas.

2.6. Usando o Project Manager

O **Project Manager** (Gerenciador de Projetos) é uma ferramenta útil que dá acesso a todos os formulários e Units de um projeto. O uso do Project Manager é especialmente importante para projetos complexos, com muitas Units e formulários.

Para exibir o Project Manager, use o comando **View | Project Manager**. A ilustração a seguir mostra o Project Manager para um projeto de tamanho médio.



Na parte esquerda da janela do Project Manager, são exibidas as Units que fazem parte do projeto. As Units são agrupadas com os formulários a elas associados. Para exibir o nome do formulário associado a uma Unit, simplesmente clique no "+" ao lado do nome da Unit. Na parte direita do Project Manager, é mostrado o *path* das Units (arquivos .PAS) e dos arquivos de descrição dos formulários (arquivos .DFM).

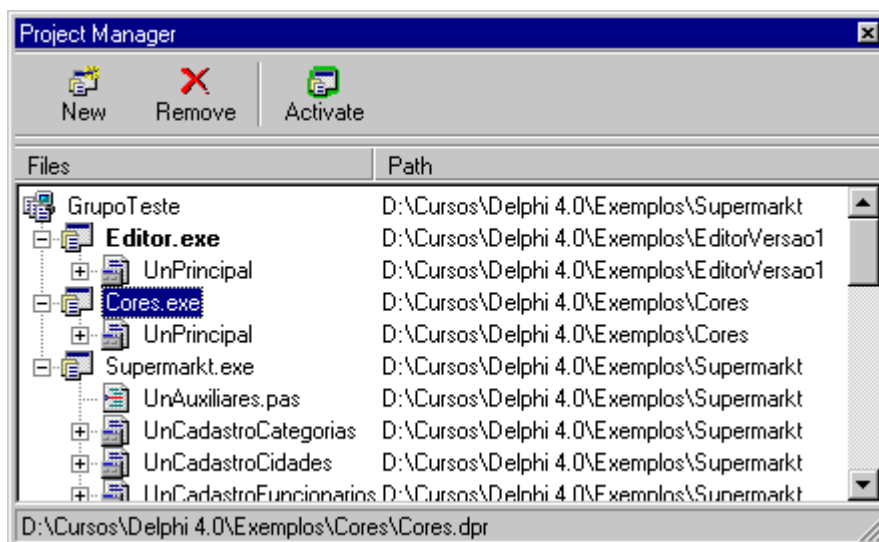
No exemplo ilustrado, a primeira Unit ("UnAuxiliares") não está associada a um formulário e portanto aparece de forma diferente na listagem. O primeiro item da listagem ("Supermarkt") é o nome do *Project Group* ao qual pertence o projeto (veremos como trabalhar com Project Groups mais adiante).

Uma das funções mais importantes do Project Manager é a *navegação* entre os arquivos de um projeto. Para mostrar o código de uma Unit no Editor de Código, ou para exibir um formulário, clique no seu nome duas vezes.

2.7. Usando Project Groups

O Delphi 4 introduziu os **Project Groups** (Grupos de Projetos), que permitem trabalhar com vários projetos ao mesmo tempo. Os Project Groups são úteis para aplicativos complexos, com vários arquivos executáveis, ou com várias DLLs (bibliotecas de funções).

Os Project Groups facilitam a compilação e a depuração de conjuntos de projetos relacionados. Código de um projeto, por exemplo, pode ser copiado facilmente para outro projeto do mesmo Project Group.



Um Project Group com três projetos, exibido no Project Manager

2.8. Salvando um Project Group

Para cada Project Group, o Delphi mantém um arquivo com a extensão .BPG. Este arquivo, no entanto, só é escrito no disco quando é salvo explicitamente pelo programador. Para salvar o arquivo do Project Group, abra o Project Manager (**View | Project Manager**), clique com o botão direito no nome do Project Group (o primeiro nome exibido) e escolha o comando **Save Project Group**.

NOTA: Se você usar o comando **File | Save all** para salvar todos os arquivos de um projeto, o arquivo do Project Group será salvo também.

2.9. Adicionando e removendo projetos em um Project Group

Pode-se adicionar novos projetos, ou projetos já existentes a um Project Group. Para adicionar um novo projeto, use o comando **Project | Add New Project**. Este comando abre a caixa de diálogo **New Items**, onde pode ser escolhido o tipo de projeto. Para adicionar um projeto existente, use o comando **Project | Add Existing Project** e escolha o projeto a ser adicionado (arquivo .DPR).

Para remover um projeto de um Project Group, é necessário usar o Project Manager (não há comando de menu para isso). No Project Manager, simplesmente clique no nome do projeto a ser removido e pressione DELETE. Outra maneira é usar o botão **Remove** da barra de ferramentas do Project Manager.

Note que um projeto não é apagado quando é removido de um Project Group. Vale notar, também, que um mesmo projeto pode fazer parte de vários Project Groups diferentes.

2.10. Ativando um projeto

Um Project Group pode conter vários projetos, mas apenas um deles pode estar **ativo**. O projeto ativo é aquele que é compilado ou executado com os comandos do menu **Project**, ou com as teclas de atalho (veja a seguir).

O projeto ativo aparece em negrito no Project Manager. Para ativar um projeto, clique no seu nome duas vezes, ou use o botão **Activate** da barra de ferramentas do Project Manager.

2.11. Compilando e executando projetos

Pode-se compilar e executar um projeto a qualquer momento durante o desenvolvimento (contanto que não haja erros no código, é claro). Há várias maneiras de compilar um projeto no Delphi. A execução também pode ser realizada de várias formas.

2.12. Verificando apenas a sintaxe

É possível verificar se os arquivos de um projeto contêm erros de sintaxe. Esse procedimento é muito rápido, porque o Delphi não precisa gerar um arquivo final.

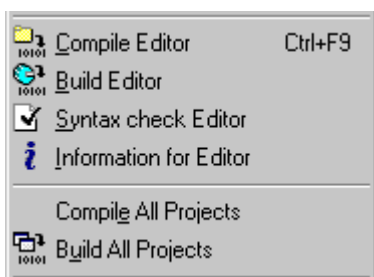
- **Para verificar a sintaxe de todos os arquivos do projeto atual:**
- Escolha o comando **Project | Syntax Check [nome do projeto]**

O Delphi verifica a sintaxe e mostra os erros resultantes em uma nova janela. Se não houver erros, volta ao estado anterior. Não é exibida nenhuma mensagem.

2.13. Compilando projetos

A compilação, além de verificar a sintaxe dos programas, gera o código executável do aplicativo. Há dois comandos para realizar a compilação no Delphi: **Compile** e **Build**, ambos no menu **Project**.

O comando **Project | Compile [nome do projeto]** compila somente os arquivos do projeto que foram alterados desde a última compilação. O comando **Project | Build [nome do projeto]** compila **todos** os arquivos do projeto, alterados ou não.



Os comandos de compilação (para um projeto chamado "Editor")

O comandos **Compile** e **Build** realizam a compilação do *projeto ativo*. Para compilar outro projeto do mesmo Project Group, é necessário antes *ativar* o projeto, usando o Project Manager.

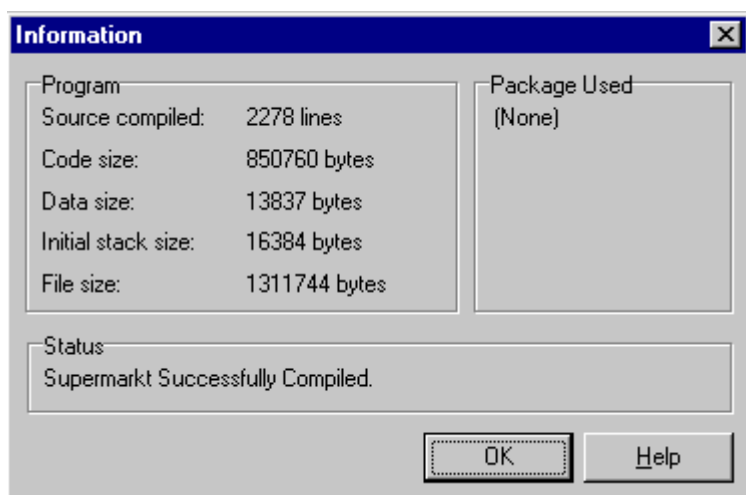
2.14. Compilando Project Groups

É possível compilar todos os projetos de um Project Group. Para isso, use os comandos **Project | Compile All Projects** e **Project | Build All Projects**. Esses dois comandos funcionam de forma semelhante aos comandos **Project | Compile** e **Project | Build**, com a diferença de que compilam todos os projetos do Project Group, e não apenas o projeto ativo.

2.15. Mostrando informações sobre a compilação

Como o compilador do Delphi é muito rápido, por padrão não são exibidas informações sobre o andamento da compilação. Essas informações estão disponíveis, no entanto, e podem ser mostradas durante a compilação. Para isso, escolha o comando **Tools | Environment Options**, mude para a página "Preferences", e ative a opção "Show Compiler Progress".

Pode-se também, exibir as informações sobre a compilação, a qualquer momento (depois da primeira compilação). Para isso, use o comando **Project | Information for [nome do projeto]**.



Exemplo de informações de compilação para um projeto

2.16. Executando o aplicativo gerado

Depois de compilar um projeto, você pode executar o aplicativo gerado de dentro do Delphi, para testá-lo. Você pode também executar o aplicativo de fora do Delphi, usando o Explorer do Windows, por exemplo. A vantagem de executar o aplicativo de dentro do Delphi, é que o Delphi é capaz de identificar e interceptar os erros que ocorrerem durante a execução do aplicativo.

- **Para executar o aplicativo ativo a partir do Delphi:**
- Escolha o comando **Run | Run**, ou pressione **F9**.

Se o projeto tiver sido alterado depois da última compilação, o Delphi compila novamente o projeto. Se forem encontrados erros, eles são exibidos na parte de baixo do Editor de Código, e o aplicativo não é executado.

2.17. Arquivos gerados pelo Delphi

Um projeto no Delphi é constituído de arquivos de vários tipos. Nas seções anteriores, mencionamos dois dos mais importantes: o arquivo de projeto (extensão .DPR) e as Units (extensão .PAS). Outros arquivos muito importantes gerados são os arquivos de descrição de formulário (extensão .DFM).

Os arquivos .DPR, .PAS e .DFM constituem o verdadeiro "código fonte" do programa e são (geralmente) os únicos arquivos imprescindíveis para a geração do aplicativo final. Vários outros arquivos são gerados na primeira compilação do projeto, como os arquivos .RES, .CFG e .DOF.

2.17.1. Arquivos .DFM

Arquivos .DFM (arquivos de descrição de formulário) são criados para cada formulário. O Delphi usa o mesmo nome da Unit associada, acrescentando apenas a extensão .DFM. Os arquivos .DFM contêm informações detalhadas sobre o formulário e seus componentes. São armazenados em um formato binário, legível apenas a partir do Delphi, mas podem ser facilmente convertidos para arquivos texto e exibidos no Editor de Código.

- **Para mostrar o arquivo DFM associado a um formulário:**
- Clique no formulário com o botão direito e escolha o comando **View as Text**.

O arquivo é exibido dentro do Editor de Código.

- **Para voltar à visão normal do formulário:**
- Clique com o botão direito dentro do Editor de Código (em cima do texto do arquivo DFM) e escolha o comando **View as Form**.

2.17.2. Arquivos .DCU

Quando um projeto é compilado no Delphi, os arquivos .DFM dos formulários e as units (arquivos .PAS) são processados e transformados em arquivos compilados intermediários, com a extensão .DCU (de "Delphi Compiled Unit"). Estes arquivos são gravados em formato binário e não podem (nem devem) ser abertos diretamente. Os arquivos DCU tornam mais rápidas a compilação e a linkagem de aplicativos no Delphi.

2.17.3. Arquivos .RES, .DOF e .CFG

Os arquivos .RES, são gerados automaticamente pelo Delphi. Esses arquivos contêm, por exemplo, o ícone do aplicativo, imagens, cursores e outros recursos relacionados ao Windows. A alteração e o uso direto dos arquivos .RES está além do escopo deste curso básico.

Os arquivos .DOF e .CFG, também gerados automaticamente, são usados para manter informações sobre opções do compilador e do linkeditor. Esses arquivos são alterados indiretamente usando o comando **Project | Options**. Os arquivos .DOF e .CFG nunca devem ser alterados diretamente.

CAPÍTULO 03 - COMPONENTES: CONCEITOS BÁSICOS

Os componentes são a parte mais importante da maioria dos aplicativos criados no Delphi. O Delphi oferece uma grande quantidade de componentes. Componentes podem ser usados para construir aplicativos completos rapidamente, algumas vezes com poucos cliques do mouse.

3.1. Tipos de componentes

Componentes podem ser divididos em dois tipos básicos: *componentes não-visuais* e *componentes visuais (controles)*.

3.2. Componentes visuais

Os **componentes visuais**, ou controles, são componentes com os quais o usuário pode interagir diretamente. Exemplos de controles são botões, caixas de texto, barras de rolagem, etc. Os componentes visuais têm a mesma aparência em tempo de desenvolvimento e em tempo de execução (salvo raras exceções).

3.3. Propriedades

Cada componente no Delphi apresenta um conjunto de **propriedades** que determinam o comportamento e a aparência do componente. Propriedades podem ser definidas durante o desenvolvimento, ou alteradas durante o tempo de execução.

3.4. Alterando propriedades em tempo de desenvolvimento

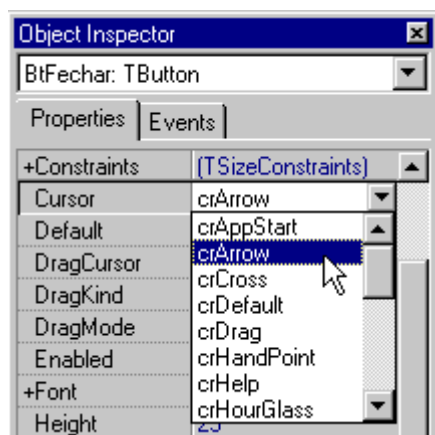
As propriedades de um componente que podem ser alteradas em tempo de desenvolvimento são listadas na parte esquerda do *Object Inspector*. Na parte direita do *Object Inspector*, são listados os *valores* para cada propriedade.

Há vários tipos de propriedades no Delphi. As propriedades mais comuns são as com **valores numéricos ou strings**. Para alterar esse tipo de propriedade, simplesmente digite um novo valor ou string, ao lado do nome da propriedade.



Propriedade numéricas e de strings

Algumas propriedades são restritas a uma **lista de valores**. As mais simples destas são as propriedades booleanas, com apenas os valores "True" ou "False" permitidos. Algumas propriedades permitem a escolha a partir de listas extensas. Para escolher um valor para uma propriedade desse tipo, simplesmente clique ao lado do nome da propriedade no Object Inspector, e escolha uma das opções da lista exibida (veja a figura a seguir).

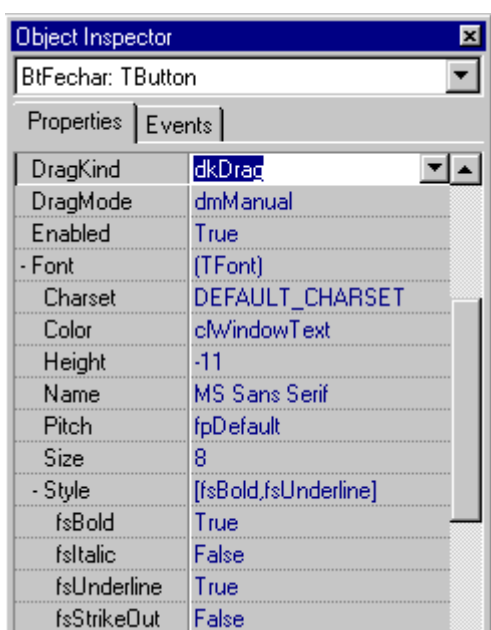


Uma propriedade com uma lista de opções

DICA: para propriedades com listas de valores, pode-se clicar duas vezes ao lado da propriedade, para exibir o próximo valor da lista. Esse recurso é especialmente útil para propriedades com poucos valores possíveis, como as propriedades booleanas.

Outro tipo de propriedade é a propriedade **composta**. Propriedades compostas, contêm várias **subpropriedades**. Os valores das subpropriedades determinam o valor final da propriedade composta. As propriedades compostas aparecem no Object Inspector precedidas de um sinal "+" (quando não expandidas).

Para definir estes valores para uma propriedade composta, primeiro clique duas vezes no nome da propriedade para abrir sua lista de subpropriedades. Em seguida altere os valores das subpropriedades desejadas. As subpropriedades podem ser de qualquer tipo (inclusive compostas).

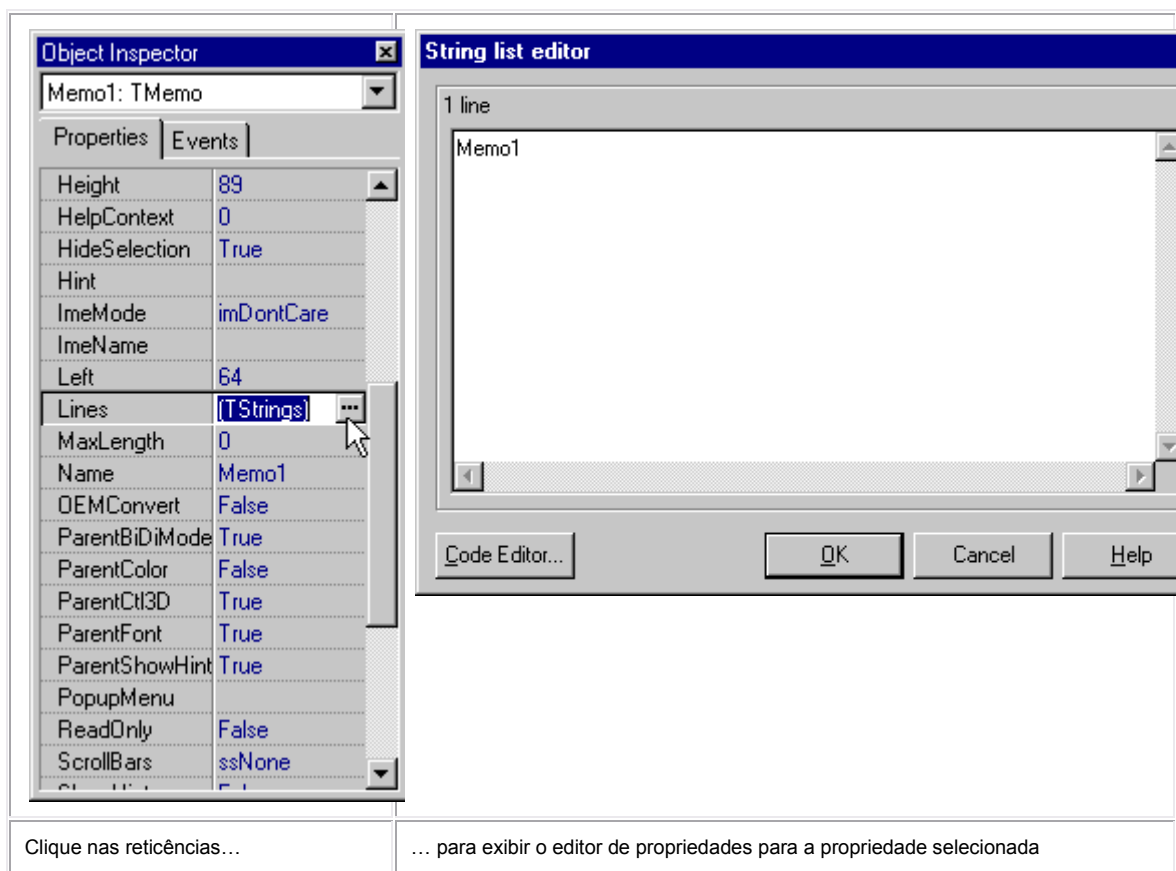


Exemplos de propriedades e subpropriedades compostas

Para algumas propriedades complexas, com muitas combinações de valores possíveis, o Delphi oferece **editores de propriedades** (*property editors*). Os editores de propriedades são comuns para propriedades compostas (como a propriedade *Font*, ilustrada acima). Nesse caso, o editor de propriedades é simplesmente outra forma de alterar a propriedade, aumentando a conveniência para o programador.

Para outras propriedades mais complexas, no entanto, os editores de propriedades são a única opção. Propriedades que podem ser alteradas com um editor de propriedades aparecem com **reticências**, quando são selecionados no Object Inspector. Para abrir o editor de propriedades, basta clicar nas reticências, ou clicar duas vezes ao lado da propriedade.

O editor de propriedades varia muito dependendo do tipo de propriedade. Veja um exemplo comum (para um componente *Memo*) na figura a seguir.



DICA: pode-se alterar propriedades de **vários componentes ao mesmo tempo**. Quando vários componentes são selecionados em um formulário, o Object Inspector mostra as propriedades *comuns* a todos eles. Se uma propriedade for alterada com vários componentes selecionados, todos esses componentes passarão a ter o mesmo valor para a propriedade alterada.

3.5. Alterando propriedades em tempo de execução

As propriedades de um componente podem também ser alteradas em tempo de execução. Essas alterações devem ser feitas usando programação. Há algumas propriedades que só podem ser alteradas usando programação. Essas propriedades não são listadas no *Object Inspector* e portanto só estão acessíveis a partir de programas.

Alterar uma propriedade de um componente usando programação é simples. O seguinte trecho de código altera o título de um formulário chamado "Form1" para "Principal" e depois desabilita um botão chamado "Button1", alterando sua propriedade *Enabled* para *False*.

```
Form1.Caption := 'Principal';
```

```
Button1.Enabled := False;
```

Comandos desse tipo são os comandos mais simples e mais usados no Delphi. São chamados de **atribuições** (como em outras linguagens de programação). Nesse caso, o lado esquerdo da atribuição define o componente e a propriedade que será alterada (separados por pontos). No lado direito é especificado o valor. Note que o símbolo para a atribuição de Object Pascal é diferente da maioria das outras linguagens (:=).

3.6. Eventos

Os eventos determinam a parte *dinâmica* de um aplicativo. Eles definem como o aplicativo reage às ações do usuário (ou de outros componentes). Cada componente oferece um conjunto de eventos específicos.

Para cada evento, pode-se associar um trecho de código que é executado quando o evento acontece. O código para os eventos de todos os componentes em um formulário é armazenado na Unit associada ao formulário.

3.7. Associando código a um evento

Os eventos disponíveis para um componente são listados na página **Events** do Object Inspector. Para muitos componentes, a lista de eventos é extensa. No entanto, somente poucos eventos são usados com frequência.

Há três formas básicas para **adicionar eventos** a um componente:

- **Clicar duas vezes no componente:** isso altera o **evento padrão** do componente. O evento padrão é geralmente o mais usado. Para o componente *Button*, por exemplo, o evento padrão é *OnClick*, que ocorre quando o botão é clicado. O Delphi mostra o Editor de Código, já com um "esqueleto" do código para o evento. Veja um exemplo desse esqueleto:

```
procedure TForm1.Button1Click(Sender: TObject);  
  
begin  
|  
  
end;
```

Entre o **begin** e o **end** é digitado o código a ser executado quando o usuário clica no botão (o Delphi já posiciona o cursor nesse local automaticamente). No exemplo a seguir, dois comandos foram adicionados para o evento. Esses serão os comandos executados quando o componente chamado "Button1" for clicado uma vez.

```
procedure TForm1.Button1Click(Sender: TObject);  
  
begin  
  
Form1.Caption := 'Principal';  
  
Button1.Enabled := False;  
  
end;
```

A maneira descrita aqui funciona somente para os *eventos padrão* dos componentes. Para adicionar código para qualquer outro evento, deve-se realizar uma das ações descritas a seguir.

- **Clicar duas vezes ao lado do evento desejado** (no Object Inspector): mostra o código para o evento no Editor de Código. A partir daí, pode-se proceder como antes para digitar o código associado ao evento.
- **Escolher um procedimento já definido para um evento**: clique do lado direito do nome do evento, no Object Inspector e escolha o nome do procedimento da lista que aparece. Faça isso quando for necessário associar o mesmo código a eventos de componentes diferentes. (Se ainda não houver nenhum código definido para eventos no formulário, a lista aparece vazia).

3.8. Apagando eventos

Pode-se apagar eventos definidos para um componente, para corrigir um erro no desenvolvimento por exemplo. Um evento pode ser apagado em "dois níveis". O evento pode ser simplesmente desligado de um componente, permanecendo ligado a outros componentes, ou pode ser removido inteiramente do código.

- **Para desligar um evento de um componente**, selecione o componente e clique no lado do evento a ser removido, no Object Inspector. Isso seleciona o nome do evento. Em seguida, apague (pressione DELETE e depois ENTER).
- **Para remover um evento inteiramente do código**, apague todo o código dentro do evento, deixando apenas o código gerado automaticamente pelo Delphi (a primeira linha, o **begin** e o **end**). Em seguida, salve o arquivo, ou compile o projeto. O Delphi remove o evento e todas as ligações a esse evento.

3.9. Métodos

Métodos (basicamente) são operações oferecidas por componentes do Delphi. Os métodos de um componente, acessíveis somente através de programação, são usados para ler ou alterar propriedades de um componente, ou para realizar tarefas especiais.

Para ver a lista de métodos oferecidos por um componente, você precisa usar o recurso de ajuda do Delphi (os métodos não são listados no Object Inspector). Para isso, selecione o

componente no formulário e pressione F1. Em seguida clique na opção *Methods*, na parte de cima da janela de ajuda. (Note que alguns componentes não oferecem nenhum método).

3.10. Adicionando componentes a um formulário

Há várias técnicas para adicionar componentes a um formulário. Apresentamos todas elas a seguir.

- **Para adicionar um componente a um local específico do formulário:**

1. Escolha o componente a ser adicionado, na paleta de componentes (mude para a página da paleta que contém o componente, se necessário).
1. Clique no formulário, no local onde deseja adicionar o componente.

O componente é adicionado no local clicado.

- **Para adicionar um componente ao centro do formulário:**
- Na paleta de componentes, clique duas vezes no componente.

Se já houver outros componentes no formulário, outros componentes adicionados usando esse método são colocados um pouco abaixo e à direita do últimos adicionados. Isso garante que os componentes não "desapareçam" uns por trás outros.

- **Para adicionar várias cópias de um mesmo componente:**

1. Na paleta de componentes, clique no componente *segurando a tecla SHIFT*.
1. Clique no formulário (várias vezes) nos locais desejados.

Não é necessário selecionar novamente o componente, antes de cada clique.

2. Quando terminar, clique novamente no componente (na paleta de componentes) para desativá-lo.

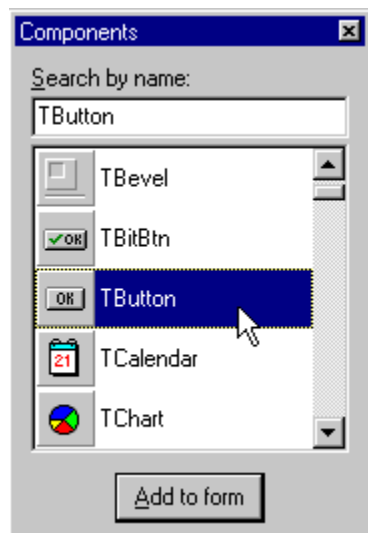
- **Para definir o tamanho de um componente no momento em que é adicionado:**

1. Na paleta de componentes, clique no componente.
1. No formulário, arraste para definir o tamanho desejado para o componente.

- **Para adicionar um componente sem usar a paleta de componentes**

1. Chame o comando **View | Component List**.

Uma caixa com todos os componentes disponíveis, listados em ordem alfabética é exibida (veja a figura).



2. Clique duas vezes no nome do componente, ou selecione o componente e clique no botão "Add to form".
3. Repita a operação para cada componente que deseja adicionar. Ao final, feche a caixa.

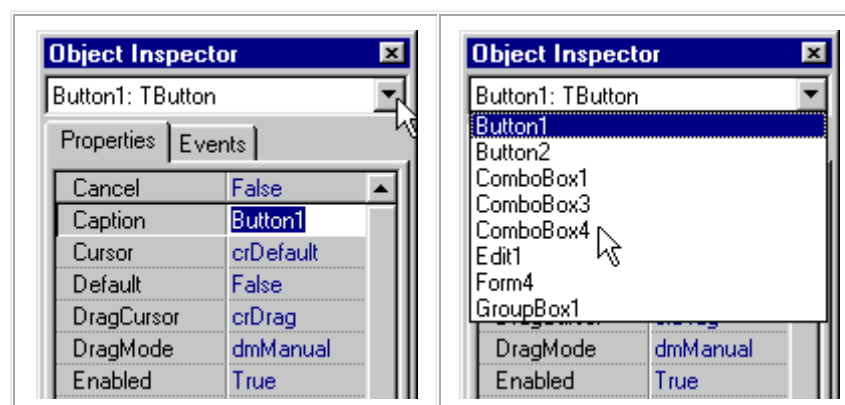
3.11. Manipulando componentes

O Delphi oferece várias técnicas para a organização precisa de componentes em um formulário. Veja as mais importantes dessas técnicas a seguir.

3.12. Selecionando e redimensionando componentes

Para fazer alterações em um componente, é necessário primeiro selecioná-lo. Veja as maneiras mais comuns.

- **Para selecionar um único componente em um formulário, realize uma das seguintes operações:**
- Clique no componente
- Clique em uma área vazia do formulário e em seguida use TAB para alternar para o componente
- Selecione o nome do componente na lista localizada na parte de cima do Object Inspector (veja a ilustração a seguir).

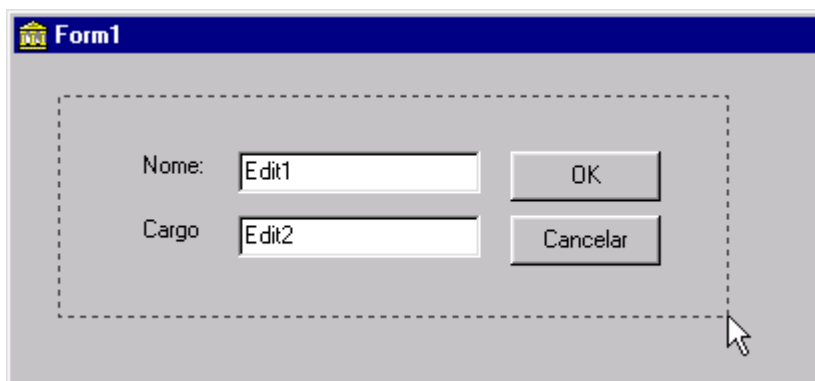


Na parte de cima do Object Inspector são listados os nomes de todos os componentes no formulário.

Selecione um componente da lista para selecioná-lo no formulário

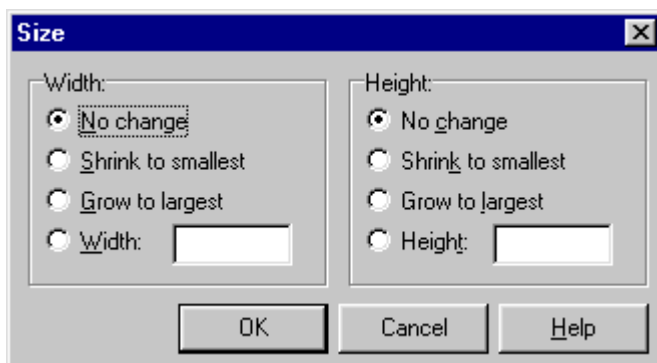
- **Para selecionar vários componentes, um de cada vez:**
- Segure a tecla SHIFT e clique em cada um dos componentes.
- **Para selecionar vários componentes em uma mesma área:**
- Clique em uma área vazia do formulário próxima aos componentes e arraste.

Um retângulo pontilhado é exibido enquanto você arrasta (veja a figura abaixo). Todos os componentes que estiverem dentro deste retângulo, ou *parcialmente* dentro dele, serão selecionados.



Se os componentes estiverem dentro de um *GroupBox* ou de um *Panel* mantenha a tecla CTRL enquanto arrasta. (Caso contrário, o *GroupBox* ou *Panel* é movido).

- **Para selecionar todos os componentes em um formulário:**
 - Use o comando **Edit | Select All**.
 - **Para redimensionar (alterar o tamanho de) um único componente:**
 - Selecione o componente e arraste uma das alças (pequenos quadrados) que aparecem em volta dele.
 - **Para redimensionar vários componentes ao mesmo tempo:**
1. Selecione os componentes e escolha o comando **Edit | Size** (ou clique em um dos componentes com o botão direito e escolha o comando **Size**).



1. Na caixa de diálogo exibida (figura acima), defina as novas dimensões para os componentes. As opções disponíveis são explicadas a seguir. (Note que as opções são as mesmas para as duas dimensões).

No change	"Nenhuma mudança": escolha essa opção para alterar somente a outra dimensão.
Shrink to smallest	"Reduzir para o menor": reduz o tamanho de todos os componentes selecionados para o tamanho do menor componente selecionado.
Grow do largest	"Aumentar para o maior": aumenta o tamanho de todos os componentes, para o tamanho do maior componente selecionado.
Width	"Largura": Especifique aqui a largura em pixels de todos os componentes selecionados.
Height	"Altura": A altura de todos os componentes selecionados.

3.13. Duplicando e transferindo componentes

No Delphi, não há um comando que permita *duplicar* um componente diretamente. Para fazer isso, é necessário usar os comandos **Copy**, **Cut** e **Paste** do menu **Edit**, ou as teclas de atalho correspondentes.

Quando um componente é duplicado dessa maneira, a cópia resultante mantém as mesmas propriedades que o componente original e continua associada aos mesmos eventos. A única diferença entre a cópia e o original é a sua propriedade *Name*, que é mudada automaticamente pelo Delphi quando o componente é copiado. (Dois componentes não podem ter o mesmo nome em um formulário).

- **Para duplicar um ou mais componentes em um formulário:**

1. Selecione os componentes e escolha o comando **Edit | Copy** (ou pressione **CTRL+C**).
1. Clique em um local vazio do formulário e escolha o comando **Edit | Paste** (**CTRL + V**).

Os componentes são copiados e são posicionados ligeiramente abaixo e à direita dos componentes originais

Pode-se também usar os comandos do menu **Edit** para *transferir* componentes de um formulário para outro, ou para dentro de um componente de agrupamento (como um *GroupBox*, ou *Panel*).

- **Para transferir componentes:**

1. Selecione os componentes e escolha o comando **Edit | Cut** (ou pressione **CTRL+X**).
1. Para mover os componentes para outro formulário clique no formulário. Para movê-los para dentro de um componente de agrupamento (*GroupBox*, *Panel*, etc.) clique dentro desse componente.
2. Escolha o comando **Edit | Paste** (**CTRL+V**).

3.14. Alinhando, travando e sobrepondo componentes

Para manter um visual uniforme e organizado nos seus formulários, muitas vezes é necessário *alinhar* componentes um pelo outro. Você pode alinhar componentes de três maneiras básicas no Delphi:

- Usando a *Alignment palette* (Paleta de alinhamento)
- Usando a caixa de diálogo *Align*, ou
- Alinhando pela grade (*grid*).











- **Para alinhar componentes com a *Alignment palette***

1. Selecione os componentes a serem alinhados
1. Escolha o comando **View | Alignment palette** para exibir a seguinte janela:



Essa paleta é uma "barra de ferramentas flutuante". Você pode movê-la para qualquer local na tela.

2. Clique no botão apropriado na paleta. Os botões da paleta são descritos na tabela abaixo:

	Alinhar pelo lado esquerdo dos componentes		Alinhar pelo centro do formulário, verticalmente.
	Alinhar pelo topo dos componentes		Espaçar igualmente, na horizontal
	Alinhar pelos centros dos componentes, horizontalmente		Espaçar igualmente, na vertical
	Alinhar pelos centros dos componentes, verticalmente		Alinhar pelo lado direito dos componentes
	Alinhar pelo centro do formulário, horizontalmente		Alinhar pela base dos componentes

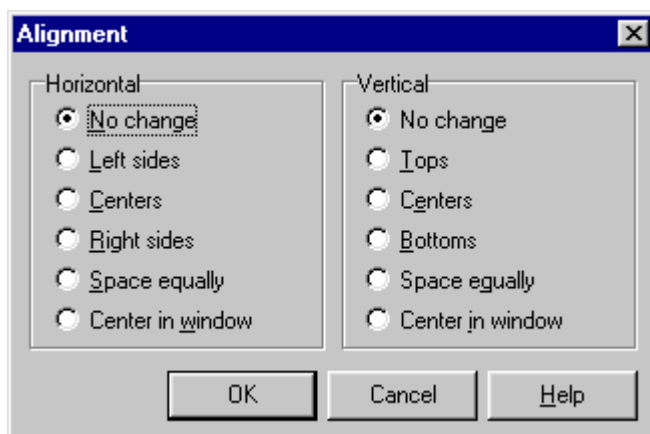
- **Para alinhar componentes com a caixa de diálogo *Alignment*:**

1. Selecione os componentes.
1. Clique com o botão *direito* em cima de um dos componentes selecionados e escolha o comando **Align**.

Ou escolha o comando **Edit | Align**.

2. Na caixa *Alignment* (figura abaixo), defina as opções de alinhamento desejadas e clique em OK.

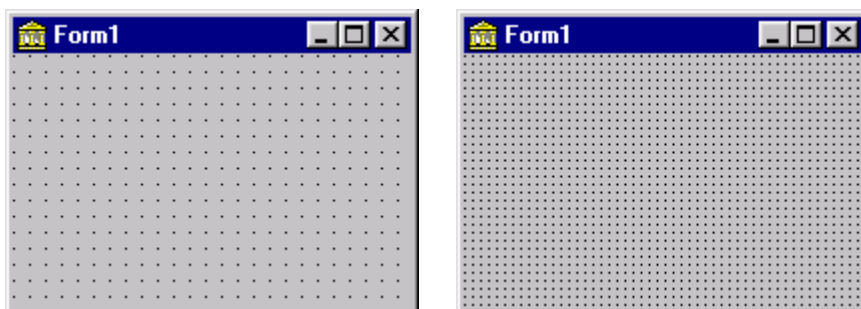
As opções são equivalentes às apresentadas na *Alignment palette*.



3.15. Alinhando pela grade

Os pontos mostrados em um formulário, durante o tempo de desenvolvimento, são a **grade** (*grid*) do formulário. A grade "atrai" componentes para posições predefinidas, facilitando o alinhamento.

Você pode desativar a grade, ou alterar seu espaçamento. Uma grade com um espaçamento menor permite o alinhamento mais preciso (e mais flexível) dos componentes.



Exibição da grade com dois valores para o espaçamento

- Para alterar o espaçamento da grade ou desativá-la:

1. Escolha o comando **Tools | Environment options**.
1. Para *alterar o espaçamento* da grade, na página **Preferences**, altere os valores das opções **Grid size X** (espaçamento horizontal) e **Grid size Y** (espaçamento vertical). Para *desativar* a grade, desligue a opção **Snap to grid**.

Você pode também *esconder* a grade, sem desativá-la. Para isso, desligue a opção **Display grid**.

3.16. Travando componentes

Depois de organizar e alinhar os componentes, você pode travá-los, para evitar que sejam desalinhados acidentalmente.

- Para travar a posição dos componentes:
- Escolha a comando **Edit | Lock controls**.

Isso trava a posição dos componentes em *todos* os formulários do projeto. Além disso, novos componentes adicionados ficam travados também (o que não é muito prático!).

Para *destravar* a posição dos componentes, escolha o comando **Edit | Lock controls** novamente.

3.17. Controlando a sobreposição de componentes

Componentes podem ser sobrepostos em um formulário. Pode-se alterar a ordem de sobreposição, trazendo um componente para frente ou enviando-o para trás.

A ordem de sobreposição padrão é a ordem em que os componentes são adicionados ao formulário. Os componentes adicionados mais recentemente são colocados "por cima" dos mais antigos.

- **Para trazer um componente para frente ou mandar um componente para trás:**

1. Selecione o componente e clique nele com o botão direito do mouse.
1. Escolha o comando **Bring to front** (Trazer para frente) ou **Send to back** (Mandar para trás).

NOTA: componentes gráficos, como os componentes *Label*, *Speedbutton* e *Bevel*, não podem ser colocados na frente de componentes "ajanelados" (a maioria dos demais).

CAPÍTULO 04 - FORMULÁRIOS E CAIXAS DE DIÁLOGO

Uma interface com o usuário no Delphi é geralmente composta por um formulário principal e vários outros formulários, como caixas de diálogo, janelas secundárias, etc.

Neste capítulo, veremos várias maneiras para criar novos formulários e como organizá-los e conectá-los. Veremos também como definir a ordem em que os formulários são criados e como usar as caixas de diálogos predefinidas do Delphi.

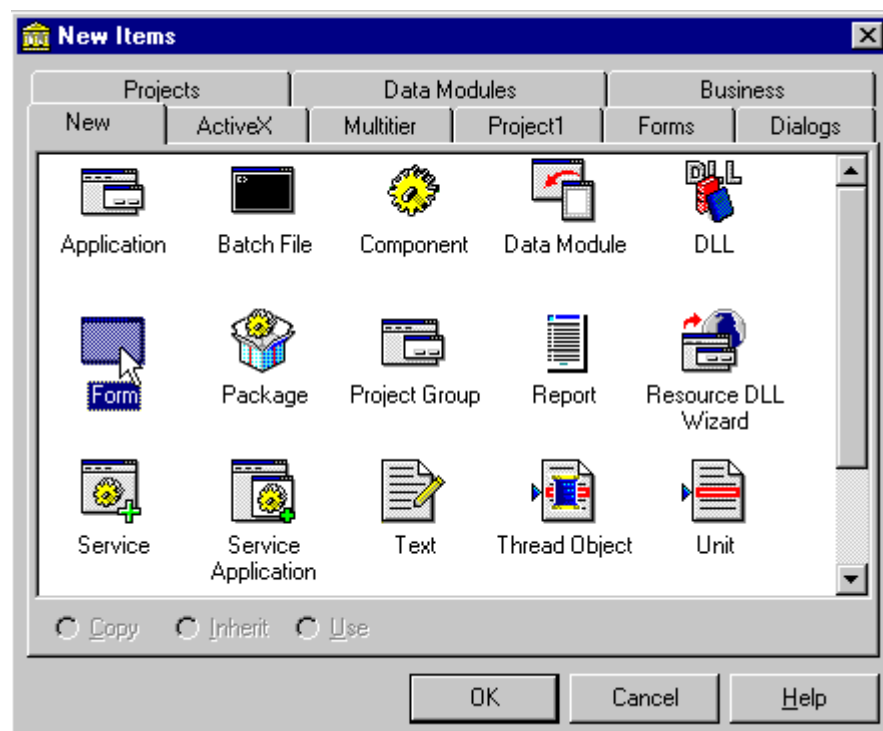
4.1. Adicionando formulários

Há várias maneiras de adicionar um formulário vazio a um projeto no Delphi. Veja as mais importantes:

- Escolha o comando **File | New form**.
- Na barra de ferramentas, clique no botão chamado **New form** (veja abaixo):



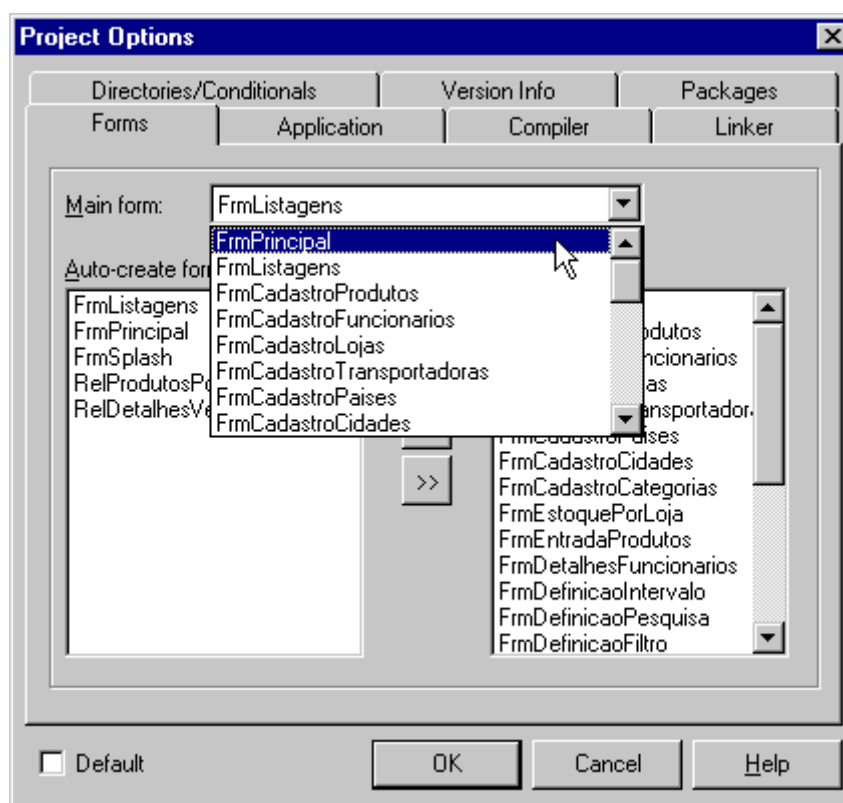
- Escolha o comando **File | New** e, na página "New", clique duas vezes no ícone "Form":
-



4.2. Especificando o formulário principal

Todo aplicativo criado no Delphi deve ter um de seus formulários definidos como **formulário principal**. O formulário principal é o formulário que é criado e exibido primeiro no aplicativo. É esse formulário, geralmente, o que contém a barra de menus, barras de ferramentas e outros componentes importantes que são usados por vários formulários secundários. Como padrão, o Delphi define o primeiro formulário adicionado como o formulário principal, mas isso pode ser alterado facilmente.

- **Para definir um formulário como o formulário principal:**
 1. Escolha o comando **Project | Options**.
 1. Na página "Forms", escolha o formulário da lista ao lado de "Main form" (veja figura).



DICA: durante o desenvolvimento, você pode tornar o formulário com que está trabalhando no formulário principal. Isso garante que o formulário seja o primeiro a ser exibido, facilitando na hora dos testes.

4.3. Ligando formulários

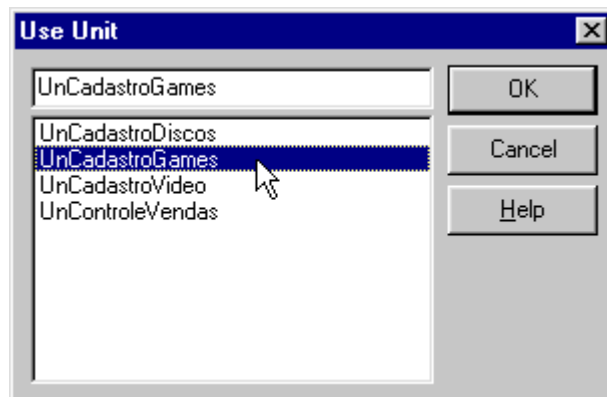
No trabalho com vários formulários, muitas vezes será necessário *ligar* formulários para que o código de um formulário possa ter acesso aos componentes do outro.

Quando se usa uma caixa de diálogo para a entrada de informações, por exemplo, deve ser possível verificar os valores entrados na caixa a partir do formulário principal. Para isso, é necessário *ligar* o formulário principal à caixa de diálogo.

Você cria uma ligação entre formulários adicionando referências aos formulários nas cláusulas **uses** das Units. Essas referências podem ser adicionadas manualmente, ou através de um comando de menu (o que geralmente é mais fácil).

- **Para ligar um formulário a outro formulário:**

1. Selecione o formulário que precisa se referir ao outro formulário.
1. Escolha o comando **File | Use unit** e escolha o formulário a ser referenciado (veja a figura a seguir).



Quando você liga um formulário a outro, o Delphi acrescenta o nome da Unit do outro formulário à cláusula **uses** do formulário ativo. Na figura, por exemplo, a Unit chamada "UnCadastroGames" foi escolhida. O Delphi altera o código para o seguinte:

```

unit UnControleEstoque;

interface

uses

Windows, Messages, SysUtils, Classes, Graphics, Controls,

Forms, Dialogs;

type

TControleEstoque = class(TForm)

end;

var

ControleEstoque: TControleEstoque;

implementation

uses UnCadastroGames;

{$R *.DFM}

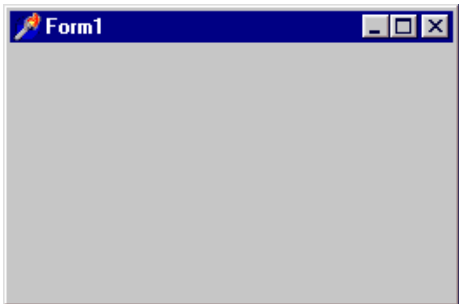

```


end.

4.4. Propriedades dos formulários

Além das propriedades que todos os componentes apresentam, como *Height* e *Width*, os formulários apresentam várias propriedades exclusivas, que permitem alterar sua aparência e seu comportamento. Veja a seguir descrições dessas propriedades.

Propriedade	Descrição
<i>ActiveControl</i>	Esta propriedade determina qual controle recebe o foco quando o formulário é mostrado. Quando o usuário clica em uma área vazia no formulário, é o controle definido aqui que recebe o foco.
<i>Caption</i>	O texto exibido da barra de título do formulário.
<i>Height, Width</i>	A altura e a largura totais do formulário, em pixels,.
<i>ClientHeight</i>	A distância (em pixels) entre a parte <i>interna</i> da borda de baixo do formulário até sua barra de título.
<i>ClientWidth</i>	A distância horizontal (em pixels) entre as partes <i>internas</i> das bordas esquerda e direita do formulário.
<i>BorderIcons</i>	Os controles que são exibidos na Barra de título do formulário. Você pode definir quatro opções para <i>BorderIcons</i> : <i>biSystemMenu</i> : mostra o Menu de Controle (também chamado "Menu de Sistema"). <i>BiMinimize</i> : mostra o botão minimizar no formulário. <i>BiMaximize</i> : mostra o botão maximizar no formulário. <i>biHelp</i> : botão de ajuda (interrogação). Só aparece se <i>biMaximize</i> ou <i>biMinimize</i> estiver em <i>False</i> . Esse botão é usado para exibir a página de ajuda definida para o formulário (se houver).
<i>BorderStyle</i>	Define a aparência das bordas do formulário e se o formulário pode ser redimensionado ou não. As seguintes opções estão disponíveis: <i>bsDialog</i> : o. Mostra somente o botão Fechar . O formulário não pode ser redimensionado. Este é o padrão para as <i>caixas de diálogo</i> . <i>bsSingle</i> : O formulário não pode ser redimensionado e é mostrado <i>com bordas</i> com linhas simples. <i>bsNone</i> :. O formulário não pode ser redimensionado <i>sem bordas</i> . <i>bsSizeable</i> : (opção padrão). O formulário pode ser

	<p>redimensionado e é mostrado com bordas.</p> <p><i>bsToolWindow</i>: como <i>bsSingle</i>, mas o formulário é mostrado com uma barra de título mais estreita. Essa opção pode ser usada para criar "barras de ferramentas flutuantes", por exemplo. O formulário não pode ser redimensionado.</p> <p><i>bsSizeToolWin</i>: como <i>bsToolWindow</i>, mas o formulário pode ser redimensionado.</p>
	<div>  <p><i>BorderStyle = bsSizeable</i></p> </div> <div>  <p><i>BorderStyle = bsSizeToolWin</i></p> </div>
<i>Position</i>	<p>Determina a posição e o tamanho do formulário em tempo de execução. As opções para <i>Position</i> são as seguintes:</p> <p><i>poDesigned</i>: a posição e o tamanho do formulário são os mesmos definidos em tempo de desenvolvimento. (Esta é a opção padrão).</p> <p><i>poDefault</i>: a posição e o tamanho do formulário são determinados pelo Windows. O Delphi não controla nem o tamanho nem a posição.</p> <p><i>poDefaultPosOnly</i>: somente a posição do formulário é definida pelo Windows. O tamanho é o que foi definido em tempo de desenvolvimento.</p> <p><i>poDefaultSizeOnly</i>: somente o tamanho é definido pelo Windows. A posição é a que foi definida em tempo de desenvolvimento.</p> <p><i>poScreenCenter</i>: o tamanho do formulário é mantido igual ao definido em tempo de desenvolvimento, mas a posição é no centro da tela. Se vários monitores estiverem sendo usados (no Windows 98), a posição pode ser ajustada para que o formulário caiba inteiro em um dos monitores.</p> <p><i>poDesktopCenter</i>: semelhante à opção <i>poScreenCenter</i>, mas a posição não do formulário não é ajustada para aplicativos que usam vários monitores.</p>
<i>WindowState</i>	<p>Determina como o formulário será exibido inicialmente no aplicativo (minimizado, maximizado ou restaurado). Opções:</p> <p><i>wsNormal</i>: mostra o formulário com tamanho normal (nem maximizado nem minimizado).</p>

	<p><i>wsMaximized</i>: mostra o formulário maximizado.</p> <p><i>wsMinimized</i>: mostra o formulário minimizado.</p>
--	---

4.5. Eventos dos formulários

Há dezenas de eventos para os formulários. Muitos deles são usados apenas raramente. A seguir apresentamos os eventos mais usados.

Evento	Descrição
<i>OnCreate</i>	Chamado quando o formulário é criado (automaticamente ou manualmente). Este evento é muito usado para realizar a inicialização de variáveis ou outras operações que devem ser realizadas logo no início da execução (como a conexão a bancos de dados, por exemplo).
<i>OnShow</i>	Chamado quando o formulário é mostrado na tela (geralmente usando <i>Show</i> ou <i>ShowModal</i>).
<i>OnClose</i>	Ocorre quando o formulário é fechado – pelo usuário, ou usando o comando <i>Close</i> . Este evento pode ser usado para liberar variáveis, ou fechar tabelas de bancos de dados, por exemplo. nota: o evento <i>OnCloseQuery</i> (se existir) é chamado antes do evento <i>OnClose</i> , para verificar se o formulário realmente pode ser fechado.
<i>OnCloseQuery</i>	Este evento é chamado, logo depois do comando <i>Close</i> , ou quando o usuário tenta fechar o formulário (clicando no 'X' por exemplo). Esse evento pode ser usado, por exemplo, para solicitar uma confirmação antes de fechar o formulário. O parâmetro CanClose do evento determina se o formulário será fechado ou não. Altere este parâmetro para <i>False</i> , dentro do código do evento, para evitar o fechamento do formulário. Se <i>CanClose</i> não for alterado, ou tiver o valor <i>True</i> no final do código do evento, o formulário é fechado e o evento <i>OnClose</i> é chamado.

4.6. Métodos dos formulários

Veja a seguir uma descrição dos métodos mais comumente usados para os formulários.

Método	Descrição
<i>Close</i>	O método <i>Close</i> fecha o formulário. Se o formulário for o principal o aplicativo é encerrado.

<i>FocusControl</i>	Use FocusControl (Nome_do_Componente) para mover o foco para o componente especificado no formulário. (O componente com foco é aquele que pode ser escolhido diretamente com o teclado).
<i>Show</i>	Mostra o formulário na tela. A execução do aplicativo continua (não é bloqueada).
<i>ShowModal</i>	Mostra o formulário na tela, mas bloqueia a execução do aplicativo até que o formulário seja fechado.

4.7. Mostrando formulários

Formulários podem ser mostrados de forma **modal** (usando o método *ShowModal*) ou de forma **não-modal** (usando *Show*).

Quando um formulário é mostrado usando *ShowModal*, a execução do aplicativo é interrompida e só continua quando o usuário fecha o formulário (cancelando ou confirmando as opções escolhidas, por exemplo). *ShowModal* é o método mais usado para as **caixas de diálogo**, formulários usados para solicitar informações do usuário, mostrar resultados, ou para pedir confirmações para ações realizadas.

Quando um formulário é mostrado usando o método *Show*, a execução do aplicativo não é interrompida. Isso permite que o usuário realize outras operações no aplicativo, mesmo enquanto o formulário está sendo exibido.

O trecho de código a seguir ilustra o uso dos métodos *Show* e *ShowModal*, para um formulário chamado "Caixa".

```

procedure TForm1.BtModalClick(Sender: TObject);

begin

    Caixa.ShowModal;

    {aqui, o código pára de ser executado, até que o usuário
    feche o formulário}

    ...

end;

procedure TForm1.BtNaoModalClick(Sender: TObject);

begin

    Caixa.Show;

    {aqui, o código continua a ser executado, mesmo antes do
    formulário ser fechado}

```

```
...  
  
end;
```

4.8. A propriedade *ModalResult*

Quando se mostra formulários de forma **modal** (usando *ShowModal*), pode-se alterar a propriedade *ModalResult* de botões no formulário para que eles realizem automaticamente ações comuns. Pode-se, por exemplo, adicionar um componente *Button* ao formulário e alterar sua propriedade *ModalResult* para *mrClose*. Isso faz com que o formulário seja fechado quando o componente *Button* é clicado.

O método *ShowModal* retorna o valor definido na propriedade *ModalResult*. Esse recurso pode ser usado para fazer decisões de acordo com o que foi escolhido no formulário, como mostra o trecho de código abaixo:

```
...  
  
Resultado := CaixaConfirmacao.ShowModal;  
  
if Resultado = mrOK then  
    {fazer alguma coisa}  
  
if Resultado = mrCancel then  
    {fazer outra coisa}  
  
...
```

4.9. Usando o teclado com formulários

A maioria dos aplicativos para Windows permite o uso do teclado para a navegação entre componentes de um formulário. As teclas mais comuns de navegação são a tecla TAB e as setas do teclado.

4.10. Alterando a ordem de tabulação

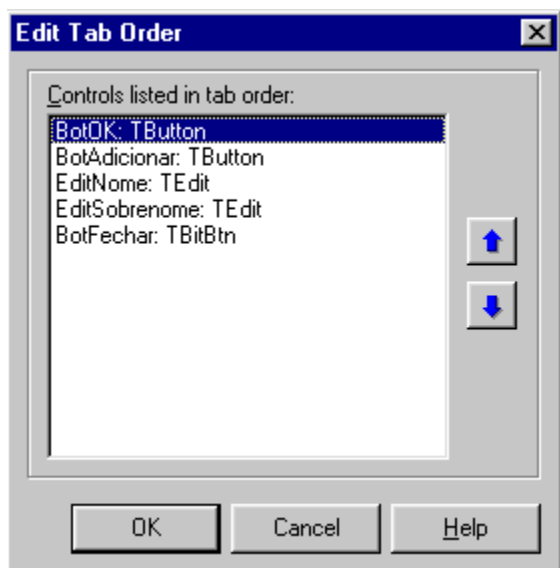
A **ordem de tabulação** é a ordem em que os componentes de um formulário recebem o **foco** (tornam-se ativos) quando é pressionada a tecla TAB. O componente que está com o foco pode ser alterado, ou ativado com o teclado. Pode-se, por exemplo, usar a tecla TAB para mover o foco para um botão no formulário e depois pressionar ENTER, o que equivale a clicar no botão.

NOTA: alguns componentes não podem receber o foco – não se pode usar a tecla TAB para chegar até eles com o teclado. Os componentes *Label* e *SpeedButton* são os exemplos mais comuns.

A ordem de tabulação pode ser definida de várias maneiras no Delphi. A maneira mais direta é alterar a propriedade *TabOrder* para cada componente. O valor de *TabOrder* deve ser um valor

inteiro e não deve haver valores duplicados no mesmo formulário (o Delphi não permite). Por exemplo, para que um componente seja o segundo a receber o foco quando o usuário pressiona a tecla TAB, faça `TabOrder = 1` (a contagem de `TabOrder` começa de zero).

Há outra maneira de alterar a ordem de tabulação. Faça o seguinte: clique com o botão direito em uma área vazia do formulário e escolha o comando **Tab Order**. Na caixa que aparece, são listados todos os componentes que podem receber foco do formulário. São listados o nome e o tipo de cada componente na ordem de tabulação atual (veja a ilustração a seguir).



Para alterar a ordem de tabulação de um componente, selecione o componente da lista e use as setas para cima ou para baixo.

NOTA: a ordem de tabulação só tem efeito em tempo de execução. Em tempo de desenvolvimento, a ordem de tabulação depende somente da ordem em que os componentes foram adicionados. Para testar a ordem de tabulação, você precisa executar o aplicativo.

4.11. Evitando que um componente receba o foco

Muitas vezes, não é necessário que todos os componentes em um formulário sejam capazes de receber o foco (usando a tecla TAB). Alguns componentes, como os componentes *Panel* e *GroupBox* geralmente não precisam receber foco.

- **Para evitar que um componente receba o foco:**
- Altere sua propriedade `TabStop` ("Parada de tabulação") para *False*.

4.12. Controlando a criação dos formulários

Quando você adiciona formulários a um aplicativo, o Delphi faz com que todos esses formulários sejam criados na memória do computador, assim que o aplicativo é executado. Essa é a configuração padrão, mas você pode fazer com que formulários não sejam criados

automaticamente (para poupar memória, por exemplo, ou para reduzir o tempo de inicialização do aplicativo).

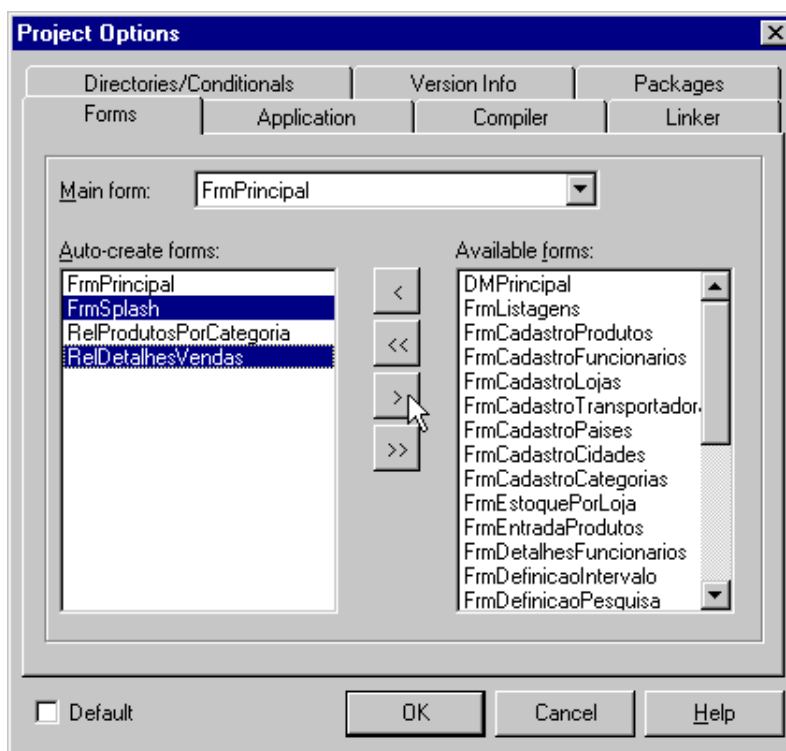
Outros casos em que é melhor não criar um formulário automaticamente é quando esse formulário não é usado com frequência, ou consome muitos recursos da máquina (acessando um banco de dados, por exemplo).

Além de definir se um formulário deve ser criado ou não, você pode definir a *ordem* em que os formulários são criados.

- **Para especificar quais formulários devem ser criados em tempo de execução:**

1. Escolha o comando **Project | Options** e mude para a página "Forms".

Os nomes de todos os formulários no projeto são exibidos na lista "Auto-create forms". Os nomes dos formulários disponíveis, mas que não serão criados automaticamente, aparecem na lista "Available forms".



1. Na lista da esquerda selecione os formulários que você *não* deseja que sejam criados durante a execução. Em seguida, clique no botão > .

Você pode também clicar no botão >> para mover todos os formulários para a segunda lista.

2. Para **alterar a ordem** em que os formulários são criados (somente para a lista da esquerda), arraste o nome do formulário para outra posição da lista.

NOTA: as mudanças descritas alteram também o arquivo de projeto. As linhas com os comandos **Application.CreateForm(...)** dos formulários colocados na segunda lista ("Available Forms") são removidas do arquivo de projeto. A ordem dos comandos também é alterada de acordo com a ordem definida na caixa **Project Options** (ilustração anterior).

Quando um formulário é movido para a lista "Available Forms", ele não é mais criado automaticamente quando o aplicativo é inicializado. Para exibir o formulário, você deve antes criá-lo na memória. Uma maneira simples de fazer isso é através do mesmo comando gerado automaticamente pelo Delphi, **Application.CreateForm**. O exemplo a seguir usa esse comando para criar um formulário (chamado "FrmProdutos") antes de mostrá-lo.

```
procedure TFrmPrincipal.ProdutosClick(Sender: TObject);

begin

    Application.CreateForm(TFrmProdutos, FrmProdutos);

    FrmProdutos.ShowModal;

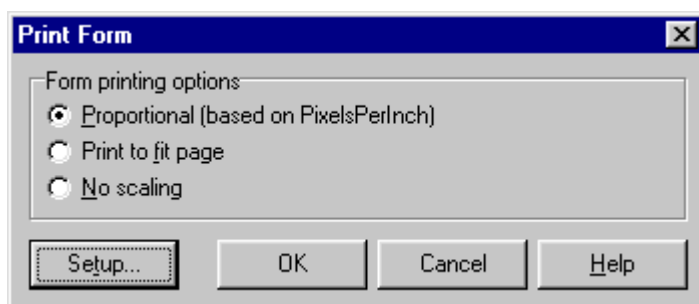
end;
```

4.13. Imprimindo formulários

Algumas vezes é útil imprimir os formulários de um aplicativo, para manter um controle organizado dos formulários que já estão prontos, por exemplo.

- **Para imprimir um formulário:**

1. Selecione o formulário e escolha o comando **File | Print**. A seguinte caixa é exibida.



1. Escolha uma das opções da caixa e clique em **OK**. As opções são descritas a seguir:

Proportional	("Proporcional") Ajusta o formulário proporcionalmente ao valor da propriedade <i>PixelsPerInch</i> do formulário. O valor inicial dessa propriedade depende da resolução atual e é calculada quando o Delphi é carregado.
Print to fit page	("Imprimir para caber na página") Ajusta o tamanho do formulário para que ocupe o maior espaço possível da página.
No scaling	("Sem ajuste de tamanho") Imprime o formulário com o mesmo tamanho que é mostrado na tela.

4.14. Caixas de diálogo predefinidas

O Delphi oferece vários comandos para a exibição de caixas de diálogo comuns, como caixas de mensagem (*Message Boxes*) e caixas de entrada (*Input Boxes*). Esses comandos permitem que aplicativos com recursos simples de entrada e saída sejam criados rapidamente.

4.15. Message Boxes (Caixas de Mensagem)

As caixas de mensagem são usadas para exibir resultados, erros, ou avisos para o usuário. Elas são caixas *modais* (bloqueiam a execução do programa quando são exibidas).

Há dois comandos para mostrar caixas de mensagem no Delphi. O mais simples é **ShowMessage**, que mostra apenas uma mensagem e um botão **OK**. Já o comando **MessageDlg** oferece mais opções, como símbolos e títulos diferentes.

4.16. Usando o comando ShowMessage

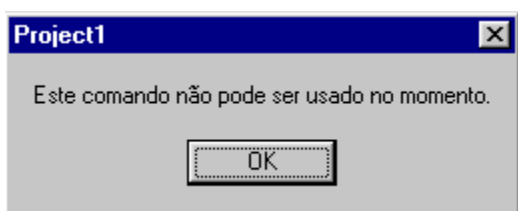
Use o comando *ShowMessage* para mostrar uma mensagem simples para o usuário. A mensagem é exibida em uma pequena caixa, com um botão **OK**. A caixa é fechada quando o botão OK é clicado.

No exemplo abaixo, uma caixa de mensagem é exibida quando o botão *Button1* recebe um clique:

```
procedure TForm1.Button1Click(Sender: TObject);  
  
begin  
  
    ShowMessage('Este comando não pode ser usado no  
    momento.');
```

```
end;
```

A caixa de mensagem exibida é a seguinte:





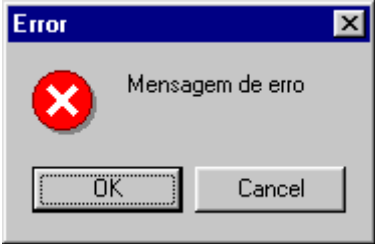


Como padrão, é exibido o nome do projeto na barra de título da caixa de mensagem.

4.17. Usando a comando MessageDlg

O comando *MessageDlg* mostra uma caixa de mensagem que pode conter símbolos especiais, botões adicionais e outros elementos. O comando tem vários parâmetros que devem ser especificados. Veja a sintaxe do comando *MessageDlg*:

```
MessageDlg(<Mensagem>, <Tipo da Caixa>, <Botões>, <Número de  
ajuda>);
```

<Mensagem> é um trecho de texto entre aspas simples, como para o comando *ShowMessage*. <Tipo da Caixa> determina o símbolo que será exibido na parte esquerda da caixa. Os cinco tipos disponíveis são ilustrados a seguir, além do comando usado para exibir cada tipo de caixa.

	
<p style="text-align: center;">mtInformation</p> <p>MessageDlg('Mensagem de informação', mtInformation, mbOKCancel, 0);</p>	<p style="text-align: center;">mtWarning</p> <p>MessageDlg('Mensagem de aviso', mtWarning, mbOKCancel, 0);</p>
	
<p style="text-align: center;">mtError</p> <p>MessageDlg('Mensagem de erro', mtError, mbOKCancel, 0);</p>	<p style="text-align: center;">mtConfirmation</p> <p>MessageDlg('Mensagem de confirmação', mtConfirmation, mbOKCancel, 0);</p>
	
<p style="text-align: center;">mtCustom</p> <p style="text-align: right;">MessageDlg('Outra mensagem', mtCustom, mbOKCancel, 0)</p>	

Note que os títulos das caixas de mensagem são sempre o *nome do tipo* da caixa, com exceção da opção *mtCustom* que exibe o nome do projeto como título (e não mostra símbolo especial). Infelizmente, os títulos são sempre em **inglês**. Não há como especificar outros títulos para as caixas usando o comando *MessageDlg*.

O terceiro parâmetro – <Botões> – é usado para definir o conjunto de botões que será exibido na parte de baixo da caixa de mensagem. Há alguns conjuntos de botões predefinidos. Um deles foi usado nos exemplos anteriores: **mbOKCancel**, que mostra os botões **OK** e **Cancel**. Os conjuntos predefinidos são resumidos a seguir:

mbYesNoCancel

Mostra os botões **Yes**, **No** e **Cancel**.

mbAbortRetryIgnore	Mostra os botões Abort , Retry e Cancel
mbOKCancel	Mostra os botões OK e Cancel .

Pode-se também adicionar botões específicos, listando o nome dos botões entre colchetes. Os botões disponíveis são:

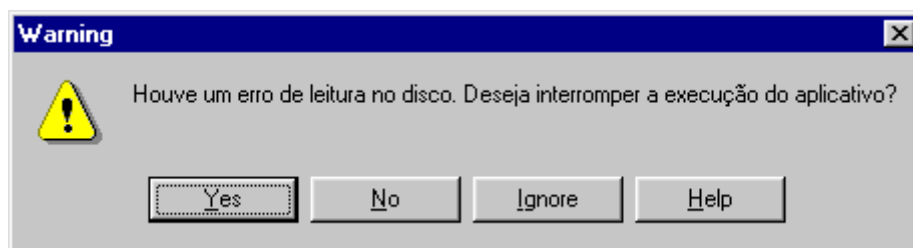
mbYes, mbNo,, mbOK, mbCancel, mbHelp, mbAbort, mbRetry, mbIgnore, mbAll

Por exemplo, para mostrar uma caixa de mensagem de aviso com os botões, **Yes**, **No**, **Ignore** e **Help**, use um comando como o seguinte (a caixa é mostrada logo depois):

```
MessageDlg('Houve um erro de leitura no disco. Deseja
interromper a

execução do aplicativo?', mtWarning, [mbYes, mbNo,

mbIgnore, mbHelp], 0);
```



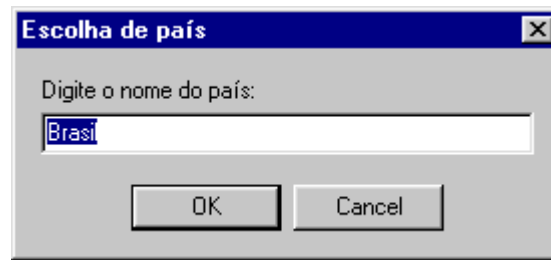
4.18. Usando a função *InputBox*

Para a entrada de informações simples, você pode usar a função *InputBox*. Essa função mostra um caixa simples com um campo para a entrada (um componente *Edit*) de dados e os botões **OK** e **Cancelar**. *InputBox* retorna um *String* com o texto digitado pelo usuário. A função *InputBox* recebe três parâmetros:

```
InputBox(<Título da Caixa>, <Texto do prompt>, <Texto padrão>);
```

<Título da Caixa> define o texto que é exibido na barra de título da caixa. <Texto do prompt> é o texto exibido na parte interna da caixa. <Texto Padrão> é o texto padrão exibido dentro do campo de entrada. Este texto aparece inicialmente selecionado. Para não mostrar um texto padrão use um string vazio (""). Veja um exemplo a seguir:

```
InputBox('Escolha de país', 'Digite o nome do país:', 'Brasil');
```



Para recuperar o que foi digitado na *InputBox*, use uma variável para armazenar o valor. No exemplo a seguir, é usada uma variável 'NomeDoPais', do tipo String:

```
procedure TForm1.Button1Click(Sender: TObject);  
  
  var  
    NomeDoPais: String;  
  
  begin  
    NomeDoPais := InputBox('Escolha de país', 'Digite o nome  
do  
país:', 'Brasil');  
  
    ShowMessage('Nome do país escolhido: ' + NomeDoPais);  
  
  end;
```

O exemplo lê um nome de país, usando a função *InputBox*, e exibe esse nome em uma caixa de mensagem.

CAPÍTULO 05 - TRABALHANDO COM MENUS

Os Menus são parte importante da maioria dos aplicativos para Windows. Com os recursos do *Menu Designer* do Delphi, é possível criar menus completos rapidamente, com comandos, teclas de atalho, separadores e outras opções. Dois tipos básicos de menus podem ser criados: menus principais e menus popup.

5.1. Adicionando menus e abrindo o Menu Designer

Menus no Delphi são criados usando os componentes *MainMenu* e *PopupMenu*. Estes são os primeiros componentes da página *Standard* da paleta de componentes (veja a figura abaixo).



Os componentes de menus

Os dois componentes de menus são exibidos como ícones no formulário. É a partir desses ícones que são construídos os menus principais e popup, usando o *Menu Designer*, uma ferramenta específica para a criação de menus. Para mostrar o Menu Designer para um componente de *MainMenu* ou *PopupMenu*, faça o seguinte:


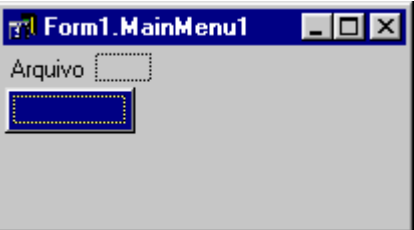
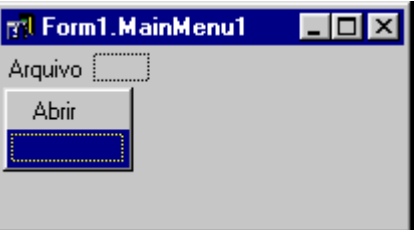
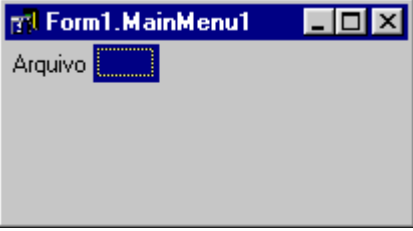
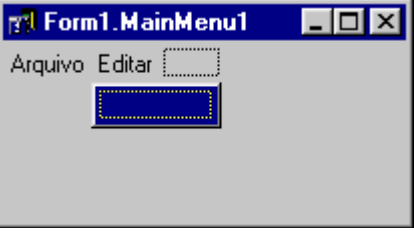
- Clique duas vezes no componente de menu, **ou**
- Clique com o botão direito no componente de menu e escolher o comando **Menu Designer**.

O Delphi abre o *Menu Designer* dentro de uma janela separada.

5.2. Construindo menus principais

Menus principais são os menus que aparecem no topo das janela principal do aplicativo. Para criar um menu principal em um formulário, adicione um componente *MainMenu* e clique no componente duas vezes para mostrar o Menu Designer.

A sequência de figuras a seguir ilustra o procedimento básico para a construção de um menu principal.

		
<p>O Menu Designer é aberto com o primeiro menu selecionado. Digite um título para o menu. O texto digitado aparece na propriedade <i>Caption</i>, no Object Inspector.</p>	<p>Depois de digitar um título, pressione ENTER. O posição do primeiro comando do menu é selecionada. Agora digite o texto para o comando.</p>	<p>Pressione ENTER outra vez para passar para o próximo comando. Continue digitando os comandos da mesma maneira.</p>
		
<p>Para passar para o próximo título de menu, pressione a seta para direita no teclado. Você pode também clicar no próximo título.</p>	<p>Digite o novo título e pressione ENTER. Continue da mesma forma para adicionar os outros comandos.</p>	

Depois de fechar o Menu Designer, você já pode visualizar o menu no formulário, mesmo em tempo de desenvolvimento.

5.3. Inserindo, movendo e apagando comandos

Depois de criar um menu, você pode alterá-lo facilmente com o *Menu Designer*. Você pode inserir, apagar, ou remover comandos, por exemplo. Clique duas vezes no componente *MainMenu* para abrir o *Menu Designer* e siga os passos a seguir para cada operação.

- **Para inserir um novo comando:**
- Mova o cursor (a barra azul) para o comando *antes* da posição que deseja para o novo comando. Pressione a tecla **INSERT** (ou **Ins**).

Um novo comando vazio é criado logo depois do cursor. Basta digitar o texto para o comando.

- **Para apagar um comando:**
- Mova o cursor para o comando e pressione **DELETE** (ou **Del**).

Se houver um submenu associado ao comando, este submenu também é apagado.

- **Para apagar um menu inteiro:**
- Mova o cursor para o título do menu e pressione **DELETE**.

Todos os comandos (e submenus) do menu são apagados.

- **Para mover um comando para outra posição:**
- Arraste o comando para outra posição.

Você pode mover o comando para outra posição do mesmo menu, ou para dentro de outro menu (ou submenu).

Se o comando movido contiver um submenu, o submenu é movido junto com o comando.

5.4. Adicionando separadores

Os separadores são usados para organizar menus extensos, agrupando comandos relacionados. Um separador aparece como uma linha horizontal entre os comandos do menu.

- **Para adicionar um separador a um menu:**
- No *Menu Designer*, digite um hífen(-) em vez de um nome para o comando e pressione ENTER.

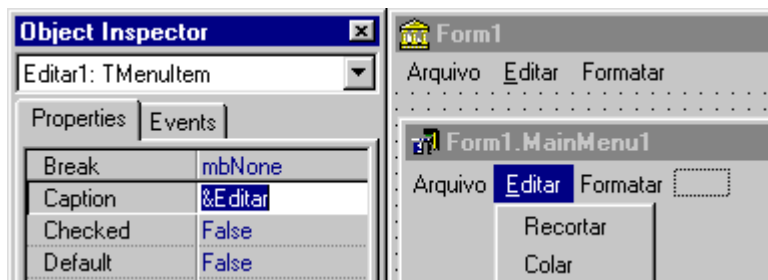
O Delphi acrescenta um linha horizontal (o separador) ao menu. O espaço ocupado pelo separador é menor quando o aplicativo é executado, como mostra a figura abaixo.



5.5. Definindo "teclas de aceleração" e teclas de atalho

Para que comandos de menus possam ser escolhidos rapidamente usando o teclado, o Delphi permite que sejam definidas **teclas de aceleração** e **teclas de atalho**. As teclas de aceleração são as letras que aparecem sublinhadas nos menus. Um comando ou título de menu com uma tecla de aceleração pode ser escolhido, pelo teclado, usando ALT com a tecla definida.

Para definir teclas de aceleração para um título de menu ou comando, digite um "&" dentro de sua propriedade *Caption*, antes do caractere a ser definido como tecla de aceleração. Na figura a seguir, a letra "E" do menu "Editar" foi definida como tecla de aceleração. Note que o "E" aparece sublinhado no menu. Assim, é possível escolher o menu com o teclado, usando **ALT + E**.



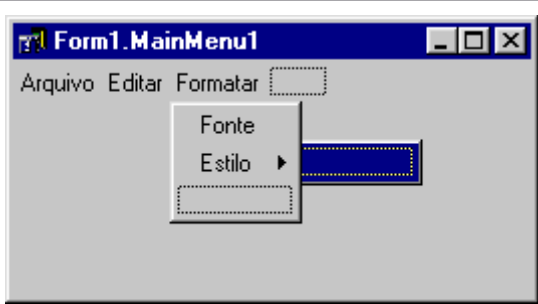
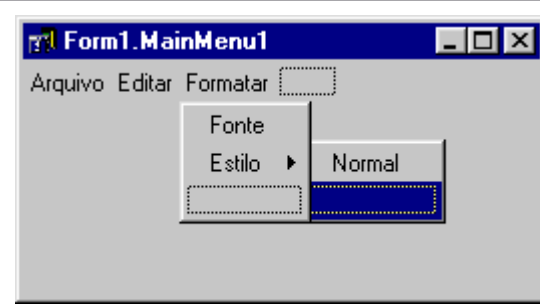
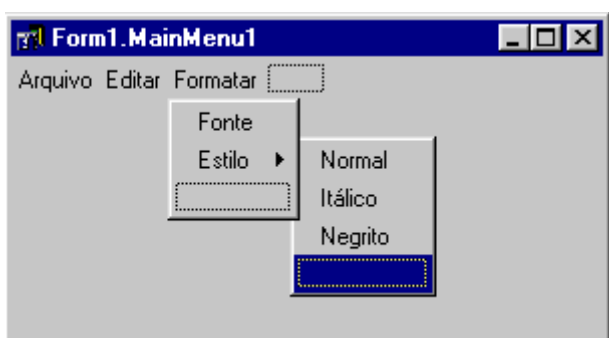
Definindo uma tecla de aceleração

As **teclas de atalho** são usadas para escolher comandos de menu rapidamente, sem a necessidade de abrir os menus. As teclas de atalho são, na verdade, *combinações* de teclas, como CTRL+C, CTRL+SHIFT+1, etc.

Para definir **teclas de atalho** para um comando de menu, selecione o menu e altere a propriedade *Shortcut*. Pode-se escolher um item da lista ou digitar uma combinação diretamente. As teclas de atalho aparecem ao lado dos comandos, nos menus.

5.6. Criando submenus

Os submenus são simplesmente "menus dentro de menus". O procedimento para criar um submenu é explicado na ilustração abaixo:

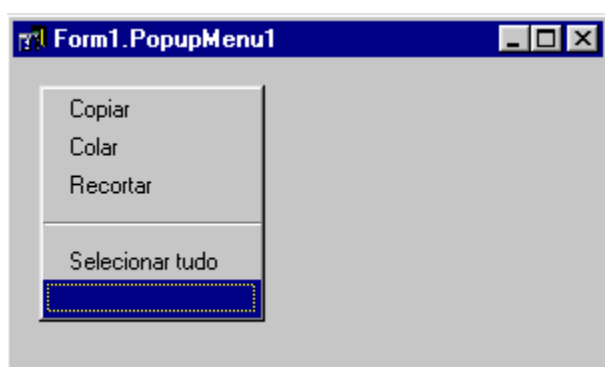
	
<p>Para adicionar um submenu a um comando, selecione o comando e pressione CTRL + Seta da Direita.</p>	<p>Digite o primeiro comando do submenu e pressione ENTER.</p>
	
<p>Continue digitando os novos comando seguidos por ENTER. Para voltar ao menu principal, use ESC.</p>	

5.7. Construindo menus *popup*

As técnicas usadas para a construção de menus *popup* são praticamente as mesmas que as usadas para os menus principais. O procedimento é até mais simples, pois um menu *popup* é formado por apenas um conjunto de comandos (e submenus).

Os menus *popup* podem ser associados a muitos tipos de componentes, inclusive *Buttons*, *Memos* e formulários. Para associar um menu *popup* a um componente, você deve adicionar um componente *PopupMenu* ao formulário que contém o componente. Depois, basta alterar a propriedade *PopupMenu* do componente para o nome do componente *PopupMenu* adicionado.

Para construir um menu *popup* (adicionar itens e submenus) proceda da mesma maneira que para os menus principais. Clique duas vezes no componente *PopupMenu* para abrir o *Menu Designer*. Em seguida, digite o nome de cada comando, seguido de ENTER. Para acrescentar outras opções como separadores ou teclas de atalho use as mesmas técnicas descritas anteriormente. A figura abaixo mostra um exemplo de um menu *popup* no *Menu Designer*.



Um menu popup no Menu Designer

5.8. Associando código aos comandos de um menu

Para associar código a um comando de menu (ou de submenu), dê um duplo clique no comando, no *Menu Designer*. Isso gera o código para o evento *OnClick* do comando. Neste evento, digite o código a ser executado quando o comando for escolhido no aplicativo.

Note que os comandos de menus, assim como os próprios menus, são tratados como componentes no Delphi. Eles possuem propriedades e eventos, como os outros componentes comuns.

5.9. Propriedades importantes dos menus e comandos

Muitas das propriedades dos menus são definidas indiretamente, através do *Menu Designer*, mas há algumas propriedades que só podem ser alteradas diretamente com o *Object Inspector*. A seguir são descritas algumas propriedades importantes dos menus e dos comandos de menus. Para alterar essas propriedades, primeiro selecione um menu, submenu, ou comando de menu no *Menu Designer*.

Propriedade	Descrição
-------------	-----------

<i>Caption</i>	<i>Caption</i> determina o texto que aparece no título ou comando de um menu.
<i>Checked</i>	<i>Checked</i> é usado para comandos que podem estar ligados ou desligados (opções). Quando <i>Checked</i> = <i>True</i> , é exibido uma pequeno 'v' do lado esquerdo do comando.
<i>Default</i>	Altere <i>Default</i> para <i>True</i> para tornar o comando selecionado no <i>comando padrão</i> do menu. O comando padrão aparece em negrito no menu e é o comando escolhido quando se clica duas vezes no título do menu. (Este é um recurso disponível, mas pouco usado no Windows).
<i>Enabled</i>	Altere <i>Enabled</i> para ativar ou desativar o comando. Um menu desativado (<i>Enabled</i> = <i>False</i>) aparece acinzentado em tempo de execução e não pode ser escolhido. Esta propriedade é usada geralmente alterada somente em tempo de execução, para desativar ou ativar um comando, dependendo do que está sendo feito pelo usuário.
<i>RadioItem</i>	<i>RadioItem</i> é semelhante <i>Checked</i> , mas mostra um pequeno círculo no lado esquerdo do comando. Comandos do mesmo grupo com <i>RadioItem</i> = <i>True</i> são mutuamente exclusivos: quando um comando do grupo é ligado, todos os outros são desligados.
<i>Shortcut</i>	As teclas de atalho usadas para executar o comando diretamente pelo teclado.
<i>Visible</i>	Altere <i>Visible</i> para exibir ou esconder um comando ou menu. Para esconder, faça <i>Visible</i> = <i>False</i> .

5.10. Trabalhando com vários menus

Um formulário pode conter vários menus, inclusive vários menus principais. O *Menu Designer* permite passar de um menu para outro em um formulário, rapidamente.

- **Para passar para um menu para outro, dentro do Menu Designer:**
- Clique com o botão direito e escolha o comando **Select Menu** para exibir a lista de menus do formulário. Depois clique duas vezes em um item da lista.

Apesar de ser possível acrescentar vários menus principais (componentes *MainMenu*) a um formulário, somente um menu principal pode estar ativo de cada vez. O menu principal usado em um formulário é determinado pela propriedade *Menu* do formulário. Essa propriedade pode ser alterada durante a execução do aplicativo. Por exemplo, a linha de código abaixo,

```
FormPrincipal.Menu := MainMenu2
```

altera o menu principal do formulário "FormPrincipal" para o menu "MainMenu2".

CAPÍTULO 06 - COMPONENTES VISUAIS COMUNS

Os **componentes visuais**, também chamados de **controles**, são usados para a interação direta com o usuário. Neste capítulo, veremos detalhes sobre os componentes visuais mais importantes e mais comumente usados

6.1. Propriedades comuns

Os componentes visuais possuem várias propriedades comuns entre si. Descrevemos a seguir as mais importantes dessas propriedades. (Algumas propriedades descritas não estão disponíveis para *todos* os componentes).


















NOTA: de agora em diante chamaremos os componentes visuais de apenas "componentes".

Propriedade	Descrição				
<i>Align</i>	<p><i>Align</i> determina o alinhamento do componente em relação ao formulário (ou a outro componente em que esteja contido). Opções:</p> <p><i>alNone</i>: sem alinhamento específico.</p> <p><i>alTop/alBottom</i>: alinhamento pelo topo/base do componente pai. A <i>largura</i> do componente se torna a mesma do componente pai.</p> <p><i>alLeft/alRight</i>: alinhamento pela esquerda/direita do componente pai. A <i>altura</i> do componente se torna a mesma do componente pai.</p> <p><i>alClient</i>: alinhamento pela área <i>cliente</i> do componente pai. A área cliente para um formulário, por exemplo, é toda a área do formulário menos a barra de título, as bordas e as barras de rolagem.</p>				
<i>Color</i>	<p><i>Color</i> determina a cor da parte interna do componente. As cores disponíveis dividem-se em <i>cores predefinidas</i>, que não dependem do sistema, e <i>cores do sistema</i>, que dependem da configuração definida no <i>Painel de Controle</i> do Windows. As cores predefinidas aparecem no início da lista. Elas são apresentadas na tabela a seguir.</p>				
<i>clAqua</i>	Azul piscina	<i>clFuchsia</i>	Roxo claro	<i>clGray</i>	Cinza escuro
<i>clGreen</i>	Verde escuro	<i>clLime</i>	Verde limão	<i>clLtGray</i>	Cinza claro
<i>clMaroon</i>	Vinho	<i>clNavy</i>	Azul marinho	<i>clOlive</i>	Verde oliva
<i>clPurple</i>	Roxo escuro	<i>clRed</i>	Vermelhão	<i>clSilver</i>	Cinza médio
<i>clTeal</i>	Azul-cinza	<i>clWhite</i>	Branco	<i>clYellow</i>	Amarelão
<i>clBlack</i>	Preto	<i>clBlue</i>	Azul puro		

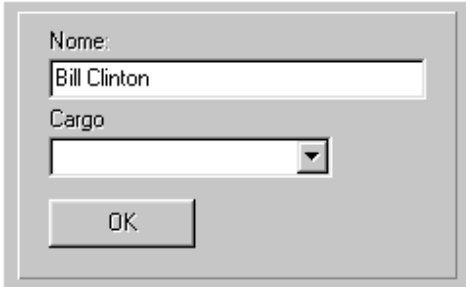
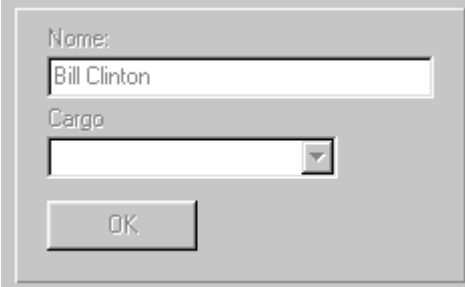
As cores predefinidas do Delphi

<i>Constraints</i>	<p>Esta propriedade determina as dimensões mínimas e máximas permitidas para o componente. <i>Constraints</i> é composta pelas seguintes subpropriedades:</p> <p><i>MinHeight</i>, <i>MaxHeight</i>: as alturas mínima e máxima, em pixels.</p> <p><i>MinWidth</i>, <i>MaxWidth</i>: as larguras mínima e máxima em pixels.</p>
<i>Cursor</i>	<p>Use esta opção para definir o tipo de cursor exibido, quando o mouse é colocado sobre o controle. A opção <i>crDefault</i> usa o cursor padrão do Windows (geralmente <i>crArrow</i>). A opção <i>crNone</i> faz com que não seja mostrado um cursor. Os outros tipos de cursores são mostrados a seguir.</p>

	<i>crArrow</i>		<i>crCross</i>		<i>crIBeam</i>
---	----------------	---	----------------	---	----------------

 crSize	 crNESW	 crNS
 crNWSE	 crWE	 crUpArrow
 crHourGlass	 crDrag	 crNoDrop
 crHSplit	 crVSplit	 crMultiDrag
 crSQLWait	 crNo	 crAppStart
 crHelp	 crHandPoint	

Tipos de cursores

Propriedade	Descrição	
<i>Enabled</i>	<p>Altere <i>Enabled</i> para <i>False</i> para <i>desativar</i> o componente. Um componente desativado não permite interação com o usuário (ele pára de funcionar). Para um componente <i>Edit</i>, por exemplo, o texto aparece mais claro e não pode ser editado. Para um componente <i>Button</i>, o texto se torna acizentado e o componente não aceita mais cliques.</p> <p>NOTA: alguns componentes (como <i>Buttons</i>) só aparecem desativados em tempo de execução.</p>	
	 <p><i>Enabled = True</i></p>	 <p><i>Enabled = False</i></p>
<i>Font</i>	A fonte utilizada no controle. Essa propriedade – uma propriedade composta – pode ser alterada no Object Inspector, ou usando a caixa de fonte padrão do Windows (clcando nas reticências).	
<i>Height, Width</i>	Determinam a <i>altura</i> e a <i>largura</i> do componente, em pixels.	
<i>Hint</i>	Defina aqui um texto para a "Dica" que será exibida quando o cursor do mouse pára sobre o controle por um momento. Para que a dica realmente apareça, você deve alterar a propriedade <i>ShowHint</i> para <i>True</i> .	
<i>Left, Top</i>	<i>Left</i> determina a distância, em pixels, do lado esquerdo do formulário à borda esquerda do componente. <i>Top</i> é semelhante, mas para a distância vertical.	
<i>Name</i>	O nome que identifica o componente, tanto no Object Inspector como em todos os programas.	
<i>PopupMenu</i>	Defina aqui um menu <i>popup</i> para o componente.	
<i>ShowHint</i>	Habilita a exibição de "dicas" para o componente. Se esta for <i>True</i> , o texto definido na propriedade <i>Hint</i> é exibido em um pequena caixa amarela (a "dica"), quando o ponteiro do mouse é deixado por alguns momentos sobre o controle.	
<i>TabOrder, TabStop</i>	<i>TabOrder</i> define a <i>ordem de tabulação</i> do componente. O valor de <i>TabOrder</i> só faz sentido se a propriedade <i>TabStop</i> estiver definida como <i>True</i> . (Veja mais sobre a ordem de tabulação no capítulo "Formulários e Caixas de Diálogo").	
<i>Tag</i>	Um número inteiro, sem significado específico, que pode ser usado como identificador do componente. Pode-se usar a propriedade <i>Tag</i> (tradução:	

	"etiqueta", "identificação") em estruturas case , por exemplo (veja o capítulo sobre Object Pascal para mais sobre a estrutura case).
<i>Visible</i>	Determina se o componente está visível ou não. Para esconder um componente, defina esta propriedade como <i>False</i> . O valor padrão (visível) é <i>True</i> . Esta propriedade normalmente só é alterada (e só tem efeito) em tempo de execução.

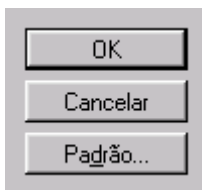
6.2. Eventos comuns

Alguns eventos são comuns à maioria dos componentes visuais. Veja detalhes sobre os mais importantes desses eventos a seguir.

Evento	Descrição
<i>OnClick</i>	Este é o evento mais usado para quase todos os componentes. O evento <i>OnClick</i> , ocorre quando o componente recebe um clique, com o botão esquerdo (botão padrão) do mouse.
<i>OnDbClick</i>	O evento <i>OnDbClick</i> ocorre quando o componente recebe um clique duplo, também com o botão esquerdo (botão padrão) do mouse.
<i>OnEnter</i>	<i>OnEnter</i> é chamado quando o componente <i>recebe o foco</i> . Um componente recebe o foco quando é ativado de alguma maneira – usando o mouse (com um ou mais cliques), ou usando o teclado (com a tecla TAB e as teclas de direção).
<i>OnExit</i>	<i>OnExit</i> é chamado quando um componente <i>perde o foco</i> . Um componente perde o foco, quando outro componente recebe o foco (somente um componente pode estar com o foco em um formulário).
<i>OnKeyPress</i>	<i>OnKeyPress</i> ocorre quando uma tecla é pressionada e soltada. O componente que "recebe" o evento é o que está com o foco. A tecla é especificada no parâmetro <i>Key</i> (do tipo <i>Char</i>) do evento.

6.3. Componente *Button*

O componente *Button* é, um dos componentes mais comuns em aplicativos para Windows.



Exemplos de componentes Button

6.3.1. Propriedades

Veja a seguir as propriedades importantes do componente *Button*.

Propriedade	Descrição
<i>Caption</i>	<i>Caption</i> determina o texto exibido no botão.
<i>Cancel</i>	Altere <i>Cancel</i> para <i>True</i> para que o botão seja ativado com a tecla ESC.
<i>Default</i>	Altere <i>Default</i> para <i>True</i> , para que o botão seja ativado com a tecla ENTER.

6.3.2. Eventos

O evento mais importante, e de longe o mais usado, para o componente *Button* é o evento **OnClick**, que ocorre quando o botão é clicado uma vez. *OnClick* é o evento padrão para esse componente.

6.4. Componente *Edit*

O componente *Edit* permite a entrada (e a exibição) de textos e valores. Esse componente suporta apenas uma linha de texto. Para textos com mais de uma linha, use o componente *Memo*, descrito em seguida.



Exemplos de componentes *Edit*

6.4.1. Propriedades

Seguem descrições das propriedades mais importantes do componente *Edit*.

Propriedade	Descrição
<i>AutoSelect</i>	Seleciona automaticamente todo o texto no <i>Edit</i> quando o componente recebe o foco no formulário.
<i>AutoSize</i>	Ajusta a altura do <i>Edit</i> automaticamente (no caso de mudanças de fonte ou tamanho, por exemplo). Somente a <i>altura</i> , mas não a largura, é ajustada.
<i>CharCase</i>	Controla a "caixa" (maiúsculas e minúsculas) do texto digitado no <i>Edit</i> . São três as opções disponíveis: <i>ecNormal</i> : (padrão) permite digitar textos de qualquer maneira (maiúsculas e minúsculas em qualquer ordem). <i>ecLowerCase</i> : restringe todo o texto a minúsculas . Todo o texto é transformado para minúsculas, mesmo se o usuário usar as teclas SHIFT e CAPS LOCK. <i>ecUpperCase</i> : tem efeito semelhante ao de <i>ecLowerCase</i> , mas restringindo o texto para maiúsculas
<i>MaxLength</i>	Define aqui a Quantidade máxima de caracteres que podem ser digitados dentro do <i>Edit</i> . <i>MaxLength</i> é ideal para limitar o tamanho da entrada para valores ou textos com tamanhos fixos, como telefones e códigos. O valor zero (o padrão) não impõe limites.
<i>PasswordChar</i>	Define o caractere exibido no componente quando o texto é digitado (normalmente asteriscos são utilizados). Tudo o que é digitado é substituído pelo caractere especificado em <i>PasswordChar</i> . O padrão é '#' que faz com que o texto digitado apareça normalmente.
<i>ReadOnly</i>	Altere <i>ReadOnly</i> para <i>True</i> para não permitir que o usuário altere o conteúdo do <i>Edit</i> . A opção <i>False</i> é a opção padrão, que permite a alteração normal do texto.
<i>SelStart</i> , <i>SelLength</i> , <i>SelText</i>	<i>SelStart</i> é a posição do primeiro caractere selecionado no <i>Edit</i> , ou a posição do cursor, se não houver texto selecionado. <i>SelLength</i> é o número de caracteres selecionados. <i>SelText</i> é o texto selecionado no <i>Edit</i> . Você pode ler os valores de <i>SelStart</i> ou <i>SelLength</i> para determinar o trecho selecionado no <i>Edit</i> , ou alterá-los para selecionar um trecho diferente. <i>SelText</i> pode ser usado para substituir o trecho selecionado por outro trecho de texto.

6.4.2. Eventos

Além dos eventos comuns, como *OnClick* e *OnDbClick*, o componente *Edit* oferece o evento **OnChange**. O evento *OnChange* ocorre quando o texto no *Edit* é alterado. *OnChange* é o evento padrão para o componente *Edit*.

6.5. Componente *Label*

O componente **Label** é usado, geralmente, para identificar componentes em um formulário. O texto de um *Label* é alterado através da sua propriedade **Caption**.

6.5.1. Propriedades

A seguir, são descritas as propriedades mais importantes do componente *Label*.

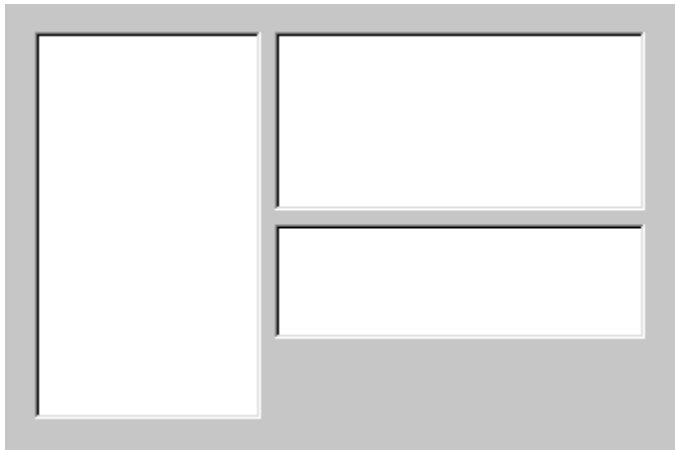
Propriedade	Descrição
<i>Alignment</i>	O alinhamento do texto do <i>Label</i> em relação à área que lhe foi reservada
<i>AutoSize</i>	Determina se o <i>Label</i> deve ter seu tamanho ajustado automaticamente ou não, quando o comprimento do seu texto (valor da propriedade <i>Caption</i>) é alterado.
<i>FocusControl</i>	O componente ao qual o <i>Label</i> está associado. Quando o usuário pressiona a tecla de atalho definida para o <i>Label</i> , o componente recebe o foco. (<i>Labels</i> não podem receber o foco). Para definir uma tecla de atalho para um <i>Label</i> , acrescente um caractere "&" dentro da propriedade <i>Caption</i> , como em "Pro&duto", ou "&Abrir".
<i>Layout</i>	<i>Layout</i> determina o alinhamento vertical do texto do <i>Label</i> em relação à área reservada para ele. Opções: <i>tlTop/tlCenter/tlBottom</i> : alinham o texto do <i>Label</i> pelo topo/centro/base da área reservada.
<i>WordWrap</i>	Determina se o texto do <i>Label</i> é quebrado em várias linhas quando não há espaço suficiente para exibi-lo em uma linha só. Use <i>True</i> , para permitir as quebras de linhas e <i>False</i> para evitá-las.

6.5.2. Eventos

Os eventos de um *Label* são raramente utilizados. Algumas vezes, os eventos **OnClick** e **OnDbClick** são utilizados para realizar alguma operação especial no componente associado ao *Label*.

6.6. Componente *Memo*

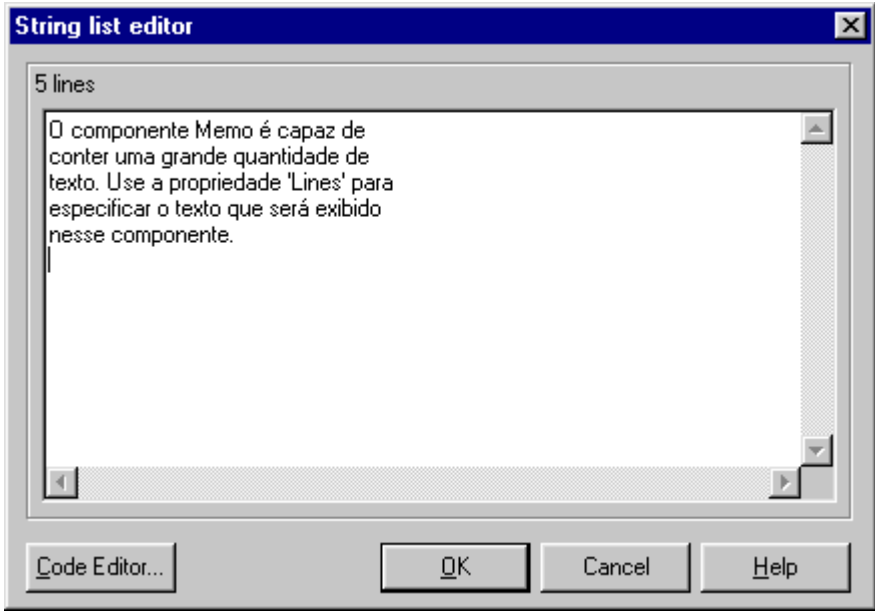
O componente *Memo* é uma extensão do componente *Edit* que permite a edição de várias linhas de texto. Ele apresenta todas as propriedades do componente *Edit* e mais algumas específicas para permitir o trabalho com trechos extensos de texto.



Exemplos de componentes Memo

6.6.1. Propriedades

Veja a seguir detalhes sobre as mais importantes propriedades (exclusivas) do componente *Memo*. Para as propriedades comuns ao componente *Edit* veja a seção sobre esse componente.

Propriedade	Descrição
<i>Lines</i>	<p>A propriedade <i>Lines</i> determina o texto exibido no <i>Memo</i>. Clique duas vezes ao lado dessa propriedade no Object Inspector (ou clique nas reticências) para exibir a janela "String list editor" (ilustração a seguir). Digite o texto normalmente na janela. Pode-se usar ENTER, TAB e várias outras teclas comuns de edição. (Veja mais sobre a propriedade <i>Lines</i> a seguir).</p>  <p>O String list editor</p>
<i>Alignment</i>	<p>Define como é alinhado o texto dentro do componente <i>Memo</i>. São três as opções disponíveis: <i>taLeftJustify</i> (alinhar à esquerda), <i>taCenter</i> (alinhar pelo centro), e <i>taRightJustify</i> (alinhar pela direita).</p>
<i>ScrollBars</i>	<p>Determina quais barras de rolagem são exibidas para o <i>Memo</i>. A opção padrão é <i>ssNone</i>, que não exibe nenhuma das duas barras de rolagem. As opções restantes são:</p>

	<p><i>ssVertical</i>: mostra somente a barra de rolagem vertical, do lado direito.</p> <p><i>ssHorizontal</i>: mostra somente a barra de rolagem horizontal, na parte de baixo.</p> <p><i>ssBoth</i>: mostra as duas barras de rolagem, à direita e abaixo do <i>Memo</i>.</p>
<i>WantReturns</i>	<p>Altere <i>WantReturns</i> para <i>True</i> para permitir que os usuários criem novas linhas dentro do <i>Memo</i>, usando a tecla ENTER (ou RETURN). Defina <i>WantReturns</i> como <i>False</i>, para passar a digitação da tecla ENTER para o formulário.</p> <p>Por exemplo, se um formulário tem um botão padrão que é ativado com a tecla ENTER (<i>Default = True</i>), e a propriedade <i>WantReturns</i> está definida como <i>False</i>, o botão padrão é escolhido – não é inserida uma nova linha no texto.</p> <p>NOTA: mesmo com <i>WantReturns</i> em <i>False</i>, pode-se digitar novas linhas no <i>Memo</i>, usando CTRL+ENTER.</p>
<i>WantTabs</i>	<p>Esta propriedade é semelhante a <i>WantReturns</i> (acima), só que para a tecla TAB. Se <i>WantTabs</i> estiver em <i>False</i>, os TABs digitados vão para o formulário, passando o foco para outros componentes (da forma usual).</p> <p>Já se <i>WantTabs</i> estiver em <i>True</i>, os TABs vão para dentro do <i>Memo</i>. Caracteres de tabulação são inseridos na posição do cursor.</p> <p>NOTA: mesmo com <i>WantTabs</i> em <i>False</i>, pode-se digitar TABs em um <i>Memo</i> usando CTRL+TAB.</p>
<i>WordWrap</i>	<p>Determina se as linhas de texto dentro do <i>Memo</i> são quebradas automaticamente ou não. O valor <i>True</i> ativa a quebra automática de linhas.</p>

6.6.2. A propriedade *Lines*

A propriedade *Lines*, que determina o conteúdo do componente *Memo*, é do tipo *TStrings*, um conjunto indexado de strings. Vários outros componentes, como o *ListBox* e o *ComboBox* também usam propriedades deste tipo.

Propriedades do tipo *TStrings* podem ser manipuladas de várias maneiras. Pode-se, por exemplo adicionar ou remover linhas, verificar ou alterar a linha com um determinado índice, ou apagar todas as linhas.

Para adicionar linhas, use o método **Add** ou **Append**, como em:

```
Memo1.Lines.Add('Uma linha')
```

ou

```
Memo1.Lines.Append('Outra linha')
```

A diferença entre *Add* e *Append* é que *Add* retorna um valor: o índice da linha adicionada. *Append* não retorna um valor.

Para remover linhas, use o método **Delete (Índice)**. O exemplo a seguir apaga as primeiras cinco linhas de um componente *Memo*. (Note que a propriedade *Lines* é indexada a partir de zero.)

```
procedure TFormTesteMemo.BtApagarClick(Sender: TObject);
var
  I: Integer;
begin
```

```

for I := 0 to 4 do

Memo1.Lines.Delete(I);

end;

```

Para remover todas as linhas, use o método *Clear*, como em Memo1.Lines.Clear. Outra maneira mais simples é usar o método *Clear* do *Memo*, como em Memo1.Clear. As duas maneiras são completamente equivalentes.

Para ler/alterar uma linha específica, use a propriedade *Lines* seguida pelo índice da linha, entre colchetes. Memo1.Lines[9] retorna a décima linha, por exemplo. No exemplo a seguir, as linhas de um *Memo* são copiadas, na ordem inversa, para outro (a propriedade **Count** retorna o número de linhas).

```

procedure TFormTesteMemo.BtInvLinhasClick(Sender:
TObject);

var

I: Integer;

begin

for I := Memo1.Lines.Count-1 downto 0 do

Memo2.Lines.Add(Memo1.Lines[I]);

end;

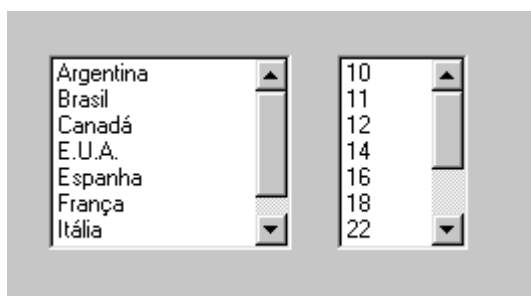
```

6.6.3. Eventos

Os eventos mais importantes do componente Memo são os mesmos do componente Edit e têm o mesmo significado. O evento **OnChange** acontece quando o texto do Memo é alterado. O evento **OnClick** ocorre quando o usuário clica em qualquer local do interior do Memo.

6.7. Componente *ListBox*

O componente *ListBox* exibe uma lista de itens que podem ser escolhidos com um clique do mouse. Os itens exibidos são definidos alterando a propriedade *Items*.



Exemplos de componentes ListBox

6.7.1. Propriedades

As propriedades mais importantes do componente *ListBox* são descritas a seguir.

Propriedade	Descrição
<i>Items</i>	A propriedade <i>Items</i> determina os itens exibidos no <i>ListBox</i> . <i>Items</i> é exatamente equivalente à propriedade <i>Lines</i> do componente <i>Memo</i> .
<i>Columns</i>	Determina o número de colunas do <i>ListBox</i> . Se o número de colunas for maior que um, os itens do <i>ListBox</i> são dispostos em colunas e é exibida uma barra de rolagem horizontal.
<i>MultiSelect</i> , <i>ExtendedSelect</i>	<p>A propriedade <i>ExtendedSelect</i> é usada em associação com a propriedade <i>MultiSelect</i> para permitir a seleção de vários itens no <i>ListBox</i>. Quando ambas <i>MultiSelect</i> e <i>ExtendedSelect</i> estão em <i>True</i>, vários itens podem ser selecionados na lista usando as teclas CTRL (para selecionar itens não-contíguos) e SHIFT (para selecionar uma sequência de itens).</p> <p>Se <i>ExtendedSelected</i> estiver em <i>False</i> (com <i>MultiSelect</i> em <i>True</i>) é possível selecionar vários itens (basta clicar), mas as teclas CTRL e SHIFT não funcionam. Finalmente, se <i>MultiSelect</i> estiver em <i>False</i>, o valor de <i>ExtendedSelect</i> não importa, pois somente um item pode ser selecionado na lista.</p>
<i>IntegralHeight</i>	Se <i>IntegralHeight</i> for definida como <i>True</i> , o último item da lista só será exibido se couber inteiramente na parte de baixo da lista (não aparecendo cortado). Com <i>IntegralHeight</i> em <i>False</i> , o último item é mostrado mesmo que apareça cortado no final da lista.
<i>Sorted</i>	Defina <i>Sorted</i> como <i>True</i> para que os itens do <i>ListBox</i> sejam ordenados alfabeticamente (ou numericamente). <i>False</i> lista as opções na ordem em que foram adicionadas.
<i>ItemIndex</i> , <i>TopIndex</i>	<p>Leia <i>ItemIndex</i> (em tempo de execução) para determinar o índice do item selecionado no <i>ListBox</i>. O primeiro item do <i>ListBox</i> tem índice zero. Se mais de um item estiver selecionado, o índice do primeiro item selecionado é retornado. Se nenhum item estiver selecionado <i>ItemIndex</i> retorna -1.</p> <p>Leia <i>TopIndex</i> (também em tempo de execução) para determinar o índice do primeiro item visível no <i>ListBox</i>. Altere o valor de <i>TopIndex</i>, para definir o primeiro item exibido no <i>ListBox</i>.</p>
<i>SelCount</i>	<i>SelCount</i> retorna o número de itens selecionados no <i>ListBox</i> .
<i>Selected</i>	<p>A propriedade <i>Selected</i> é um array de valores booleanos, com elementos para todos os itens do <i>ListBox</i>.</p> <p>O seguinte código lê os valores em <i>Selected</i> para copiar todos os itens selecionados em um <i>ListBox</i> para um componente <i>Memo</i>:</p> <pre> procedure TForm1.BtMoverClick(Sender: TObject); var I: Integer; begin for I := 0 to ListBox1.Items.Count-1 do if ListBox1.Selected[I] then Memo1.Lines.Add(ListBox1.Items[I]); end; </pre>

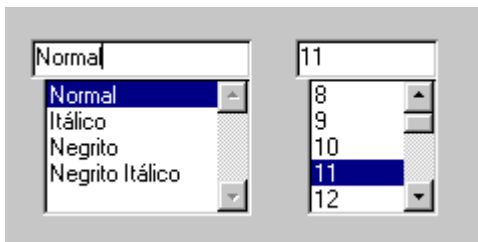
6.7.2. Eventos

O evento **OnClick** para um *ListBox* é chamado quando um item da lista é selecionado com o mouse.

O evento **OnDbClick** é chamado quando um item do *ListBox* é clicado duas vezes. Esse evento é geralmente usado para realizar ações imediatas, como adicionar o item selecionado a outro componente, por exemplo.

6.8. Componente *ComboBox*

O componente *ComboBox* é uma combinação de um componente *Edit* com um componente *ListBox*. O *ComboBox* mostra uma lista de opções (o *ListBox*) e uma área onde se pode digitar livremente (o *Edit*).

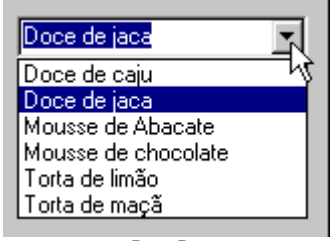
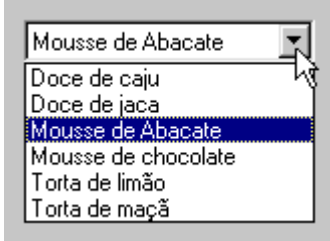
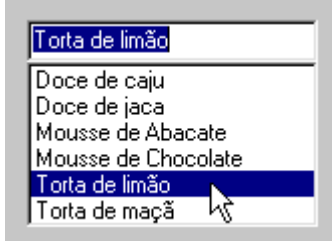


Exemplos de componentes *ComboBox*

6.8.1. Propriedades

O componente *ComboBox* apresenta quase todas as propriedades dos componentes *ListBox* e *Edit*. Veja as mais importantes propriedades a seguir.

Propriedade	Descrição
<i>DropDownCount</i>	Determina o número de itens listados (o padrão é 8)
<i>ItemIndex, TopIndex</i>	Leia <i>ItemIndex</i> para determinar o índice do item selecionado no <i>ComboBox</i> . Leia/altere <i>TopIndex</i> para determinar/alterar o primeiro item exibido na lista de opções (parte de baixo do <i>ComboBox</i>).
<i>Items</i>	Os itens listados na parte de baixo no <i>ComboBox</i> . Esta propriedade é do tipo <i>TStrings</i> e é idêntica à propriedade <i>Lines</i> do <i>Memo</i> e à propriedade <i>Items</i> do <i>ListBox</i> .
<i>MaxLength</i>	O número máximo de caracteres que pode ser digitado na parte de cima do <i>ComboBox</i> (a área de digitação). Se <i>MaxLength</i> for zero (o valor padrão), o número é ilimitado.
<i>MultiSelect, ExtendedSelect, SelCount, Selected</i>	Estas propriedades, usadas para a seleção de múltiplos itens, são idênticas às de mesmo nome para o componente <i>ListBox</i> (veja a seção sobre este componente).
<i>Sorted</i>	Altere <i>Sorted</i> para <i>True</i> para que os itens do <i>ComboBox</i> sejam ordenados alfabeticamente.
<i>Style</i>	Esta é uma das propriedades mais importantes. Ela determina a aparência e o comportamento (o estilo) do <i>ComboBox</i> . As opções seguintes são as mais usadas: <p><i>csDropDown</i>: com esta opção, o <i>ComboBox</i> permite que seja escolhido um valor da lista ou que seja digitado um valor na parte de cima.</p> <p><i>csDropDownList</i>: com esta opção, a parte de cima da caixa não pode ser alterada. O usuário fica restrito somente às opções da lista.</p> <p><i>csSimple</i>: é uma opção usada nas caixas "Abrir" e "Salvar" da maioria dos aplicativos para Windows. Aqui a lista de opções é sempre exibida (o número de itens exibidos depende do tamanho do <i>ComboBox</i>). Para essa opção, enquanto você digita na parte de cima, a lista se ajusta automaticamente, fazendo o item mais próximo do que está sendo digitado passar a ser a primeira da lista.</p>

			
	<i>csDropDown</i>	<i>csDropDownList</i>	<i>csSimple</i>
Text	O texto na parte de cima do <i>ComboBox</i> .		

6.8.2. Eventos

O evento **OnChange** é chamado quando o texto na parte de cima da lista é alterado. O evento **OnClick** é chamado quando o usuário clica em algum item da lista.

6.9. Componente *CheckBox*

O componente *CheckBox* é usado para permitir a escolha de opções não mutuamente exclusivas (várias opções podem ser escolhidas ao mesmo tempo). Para cada *CheckBox* há um título associado. Um *CheckBox* pode estar em três estados: marcado (checked), desmarcado (*unchecked*), ou acizentado (*grayed*).

6.9.1. Propriedades

As propriedades mais importantes do componente *CheckBox* são descritas a seguir.

Propriedade	Descrição
<i>Alignment</i>	O alinhamento do título do <i>CheckBox</i> .
<i>Caption</i>	O título que identifica o <i>CheckBox</i> .
<i>AllowGrayed</i>	Altere <i>AllowGrayed</i> para <i>True</i> para permitir mais um estado para o <i>CheckBox</i> : o estado <i>acizentado</i> . O significado desse estado depende do aplicativo. Ele pode, por exemplo, indicar que uma opção está em um estado intermediário: nem ligada nem desligada.
<i>Checked</i>	O estado inicial do <i>CheckBox</i> . <i>True</i> marca o <i>CheckBox</i> ; <i>False</i> desmarca.
<i>State</i>	Use <i>State</i> para alterar ou ler o estado do <i>CheckBox</i> . Os três estados possíveis são <i>cbUnchecked</i> (desmarcado), <i>cbChecked</i> (marcado) e <i>cbGrayed</i> (acizentado). A opção <i>cbGrayed</i> só tem efeito se a propriedade <i>AllowGrayed</i> for <i>True</i> . Veja abaixo a aparência de um <i>CheckBox</i> para cada um dos três estados: <div data-bbox="432 1576 1257 1680"> <input type="checkbox"/> Impressão em cores <input checked="" type="checkbox"/> Impressão em cores <input type="checkbox"/> Impressão em cores </div> <i>cbUnchecked cbChecked cbGrayed</i>

6.9.2. Eventos

O evento **OnClick** é o evento mais usado para o componente *CheckBox*. O evento *OnClick* ocorre quando o *CheckBox* recebe um clique. Um clique também automaticamente marca (ou desmarca) o *CheckBox*.

NOTA: você não precisa usar o evento *OnClick* para verificar se um *CheckBox* está marcado ou não. Para isso, verifique a propriedade *Checked*.

6.10. Componente RadioButton

O componente *RadioButton* é usado geralmente para a escolha entre opções *mutuamente exclusivas* (uma opção cancela a outra). Um *RadioButton* só pode estar em dois estados: ligado ou desligado. Altere a propriedade *Checked* do botão para *True* para ligá-lo e para *False* para desligá-lo.



Exemplos de RadioButtons

As **propriedades** do componente *RadioButton* são praticamente as mesmas que para o componente *CheckBox*.

Os **eventos** do componente *RadioButton* são exatamente os mesmos que os do componente *CheckBox*. Como para o componente *CheckBox*, o evento mais usado é o evento **OnClick**.

O componente *RadioGroup* (descrito a seguir) é geralmente usado em vez do componente *RadioButton*.

6.11. Componente RadioGroup

O componente *RadioGroup* é uma versão especial do componente *GroupBox* (descrito a seguir), que só pode conter *RadioButtons*. Para adicionar *RadioButtons*, altere a propriedade *Items* (da mesma forma que para os componentes *ListBox* e *ComboBox*).

Para organizar os *RadioButtons* em mais de uma coluna, altere a propriedade *Columns*. A propriedade *ItemIndex* determina qual *RadioButton* está selecionado.



Exemplos de RadioGroups

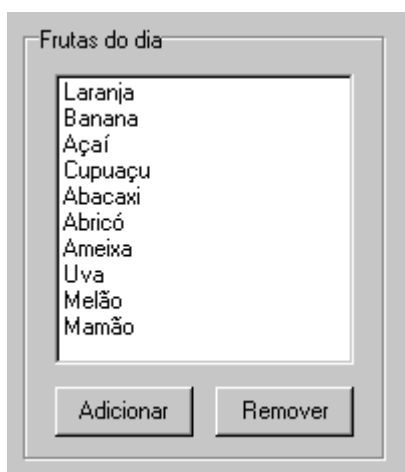
Os *RadioButtons* dentro de um *RadioGroup* não podem ser alterados diretamente. Para alterar os títulos dos *RadioButtons*, ou apagar ou acrescentar *RadioButtons*, altere a propriedade

Items. Para alterar o espaçamento entre cada *RadioButton*, aumente ou diminua o tamanho do *RadioGroup*. O espaçamento é ajustado automaticamente.

OnClick é o evento mais usado para *RadioGroups*. Dentro do código para o evento *OnClick*, verifique a propriedade *ItemIndex* para determinar qual *RadioButton* foi selecionado.

6.12. Componente *GroupBox*

O componente *GroupBox* é muito útil para a organização de formulários complexos. Componentes adicionados a um *GroupBox* passam a *fazer parte* do *GroupBox*. Se o *GroupBox* for movido ou apagado, os componentes contidos nele são movidos ou apagados também.



Exemplo de GroupBox

Para adicionar um componente a um *GroupBox*, simplesmente clique dentro do *GroupBox* ao adicionar o componente. Depois de adicionado a um *GroupBox*, um componente só pode ser movido para fora dele usando os comandos **Cut** e **Paste**.

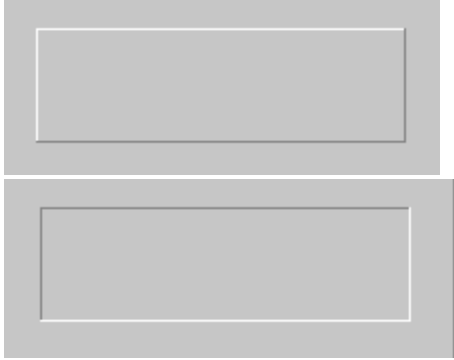
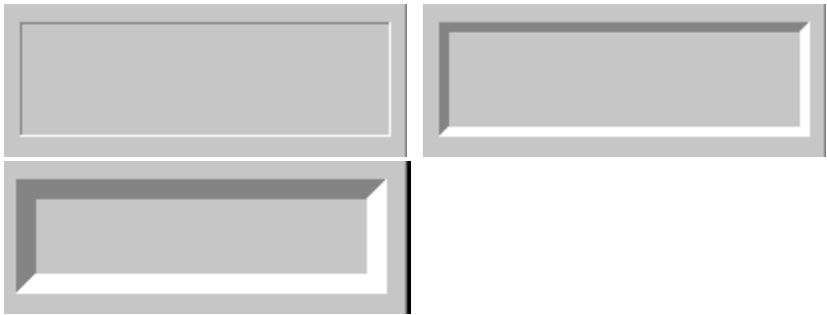
Para alterar o título de um *GroupBox*, altere sua propriedade *Caption*. Se você apagar todo o texto dessa propriedade, a caixa de grupo se transforma apenas em um contorno retangular (isso é usado com frequência).

O Delphi oferece vários **eventos** para os componentes *GroupBox*, mas estes são raramente usados. Para lidar com eventos dos componentes dentro de um *GroupBox*, use os eventos desses componentes.

6.13. Componente *Panel*

O componente *Panel* é usado, como o *GroupBox*, para agrupar componentes relacionados. Uma vantagem do *Panel* em relação ao *GroupBox* é a sua versatilidade, principalmente nos seus recursos visuais.

A maioria das propriedades do componente *Panel* afetam aparência do componente no formulário. Veja a seguir as propriedades usadas com mais frequência.









Propriedade	Descrição
<i>Alignment</i>	Determina o alinhamento do texto do Panel (se houver). O texto é determinado pela propriedade <i>Caption</i> .
<i>BevelInner</i>	Especifica a maneira como a parte <i>interna</i> do Panel é exibida. A opção <i>bvLowered</i> faz com que a parte interna pareça "abaixada". <i>bvRaised</i> tem o efeito contrário (aparência levantada). A opção <i>bvNone</i> (a opção padrão) retira o efeito tridimensional da parte interna do <i>Panel</i> .
	 <p><i>BevelInner = bsRaised BevelInner = bsLowered</i></p>
<i>BevelOuter</i>	Idêntico à propriedade <i>BevelInner</i> , só que para a parte <i>externa</i> do <i>Panel</i> .
<i>BevelWidth</i>	A intensidade do efeito tridimensional (veja as figuras).  <p><i>BevelWidth = 1 BevelWidth = 5 BevelWidth = 10</i></p>
<i>BorderStyle</i>	Determina se é exibida ou não uma moldura em volta do <i>Panel</i> .
<i>BorderWidth</i>	A distância entre os efeitos tridimensionais interno e externo. Só tem efeito quando a propriedade <i>BevelInner</i> tem o valor <i>bvLowered</i> ou <i>bvRaised</i> .
<i>Caption</i>	O texto exibido dentro do <i>Panel</i> .

6.14. Componente *BitBtn*

O componente *BitBtn* é uma versão especializada do componente *Button*, com a capacidade de exibir imagens e realizar algumas ações comuns. Este componente é muito usado para a criação de caixas de diálogo.

Há vários tipos predefinidos de componentes *BitBtn*. Para escolher o tipo do *BitBtn*, altere a propriedade *Kind*. A aparência do botão (a imagem e o texto que aparecem nele), bem como a ação realizada, dependem do valor escolhido para a propriedade *Kind*. Os tipos mais comuns de *BitBtns* são ilustrados a seguir.

Valor de <i>Kind</i>	Aparência do	Valor de <i>Kind</i>	Aparência do
----------------------	--------------	----------------------	--------------

	<i>BitBtn</i>		<i>BitBtn</i>
bkAbort	 Abort	bkNo	 No
bkCancel	 Cancel	bkOK	 OK
bkClose	 Close	bkRetry	 Retry
bkIgnore	 Ignore	bkYes	 Yes

CAPÍTULO 07 - A LINGUAGEM OBJECT PASCAL

A linguagem Object Pascal é a linguagem por trás de quase todas as partes de um aplicativo no Delphi. Os arquivos de projeto e as Units, como já vimos, são escritos em Object Pascal. O código usado para criar os componentes predefinidos do Delphi é também praticamente todo nessa linguagem.

Nos capítulos anteriores usamos várias vezes pequenas partes da linguagem Object Pascal (dentro do código para os eventos, por exemplo). Neste capítulo veremos como usar recursos mais avançados da linguagem e como criar programas que não dependam somente da interface com o usuário.

7.1. Usando o comando de atribuição

O comando de atribuição é um dos mais usados em muitas linguagem de programação. O comando de atribuição é usado para a alterar valores de variáveis e propriedades no Delphi. Já mostramos vários exemplos que usam atribuição. Aqui está mais um:

```
procedure TForm1.Button1Click(Sender: TObject);  
  
begin  
  
    Edit1.Color := clRed;  
  
    Edit1.Text := 'Novo texto';  
  
    Memo1.ReadOnly := True;  
  
end;
```

O comando de atribuição, como se vê é formado por duas partes, a parte esquerda é um nome de uma variável ou de uma propriedade; a parte direita é o novo valor que será *atribuído* à variável ou propriedade. Esse valor pode ser outra variável ou propriedade, uma constante, ou outra expressão. Os tipos dos dois lados devem ser os mesmos, ou pelo menos compatíveis; caso contrário, o Delphi gera erros na compilação.

7.2. Entendendo identificadores

Identificadores (*Identifiers*) são os nomes que identificam os elementos de um programa em Object Pascal. Exemplos de identificadores são nomes de variáveis, constantes, procedures, functions e componentes. Em Object Pascal todos os identificadores usados devem ser **declarados**. Quando você declara um identificador, você está definindo um *tipo* para ele (como inteiro, String, etc.).

Um identificador deve seguir as seguintes regras básicas:

- Um identificador pode ter até 63 caracteres. Qualquer caractere que passe desse limite é ignorado pelo compilador do Delphi.
- Identificadores devem começar sempre com letras, ou com o caractere de sublinhado (_). Um identificador não pode começar com um número. Os outros caracteres de um identificador (do segundo em diante) podem ser letras, números ou sublinhados. Nenhum outro caractere é permitido.

- Um identificador não pode ser uma palavra reservada da linguagem Object Pascal (como **begin**, **if** ou **end**, por exemplo).

7.3. Declarando variáveis

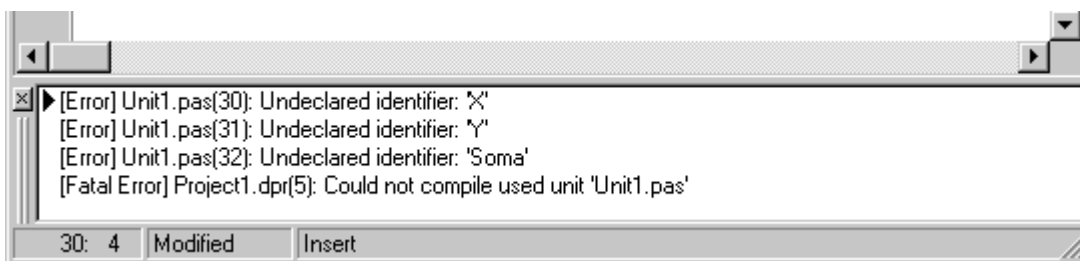
A declaração de uma variável define o **nome** e o **tipo** da variável. Declarações de variáveis devem ser precedidas pela palavra-chave **var**. As declarações, junto com a palavra-chave **var** constituem o chamado **bloco var**.

Declarações devem ser agrupadas no início do programa, ou no início de uma *procedure* ou *function* (antes da palavra-chave **begin**).

O programa (trivial) abaixo mostra um exemplo de declaração de variáveis. São declaradas três variáveis inteiras (**X**, **Y** e **Soma**). Assume-se que há dois componentes *Edit* no formulário, onde são digitados os dois valores a serem somados. As funções **StrToInt** e **IntToStr** convertem o tipo de *String* para o *Integer* (Inteiro) e de *Integer* para *String*, respectivamente. Um caixa de mensagem é exibida no final, com o resultado da soma.

```
procedure TForm1.Button1Click(Sender: TObject);  
  
var  
  
    X,Y: Integer;  
  
    Soma: Integer;  
  
begin  
  
    X := StrToInt(Edit1.Text);  
  
    Y := StrToInt(Edit2.Text);  
  
    Soma := X + Y;  
  
    ShowMessage('Soma = ' + IntToStr(Soma));  
  
end;
```

Se as variáveis desse programa não fossem declaradas, os seguintes erros seriam mostrados na compilação (na parte de baixo do Editor de Código):



Portanto, lembre-se sempre de declarar todas as variáveis usadas nos seus programas!

7.4. Tipos de variáveis

O tipo declarado para uma variável define o conjunto de valores permitidos para aquela variável.

Há uma grande quantidade de tipos disponíveis em Object Pascal. Há vários tipos muito parecidos entre si, que são usados apenas para manter compatibilidade com versões anteriores da linguagem Pascal. Abaixo listamos os tipos mais comumente usados.

Tipo	Descrição	Tamanho
Integer	Números inteiros (sem parte fracionária) que podem variar de -2.147.483.647 até 2.147.483.647.	4 bytes
Double	Números reais de alta precisão, com até 16 dígitos significativos. Usado para números muito grandes ou muito precisos. Podem variar de 5.0×10^{-324} até 1.7×10^{308} .	8 bytes
Real	Números reais de alta precisão. Idêntico a <i>Double</i> .	8 bytes
Currency	Números reais de alta precisão, com quatro casas decimais. O tipo ideal para valores monetários (use este tipo em vez de <i>Double</i> , para esses valores). Uma variável com esse tipo pode variar de -922337203685477.5808 até 922337203685477.5807 (mais de 900 trilhões).	8 bytes
Boolean	Valores booleanos só podem ter valores <i>True</i> (verdadeiro) ou <i>False</i> (falso).	1 byte
Char	Caracteres ANSI (padrão)	1 byte
String	Cadeias de caracteres com tamanhos que podem variar dinamicamente. Strings podem ter tamanho praticamente ilimitado.	Variável

7.5. Declarando constantes

Constantes são usadas em programas para armazenar valores que não podem (ou não devem) ser alterados em um programa. Declarações de constantes devem ser precedidas pela palavra-chave **const**. As declarações das constantes, junto com a palavra-chave **const** constituem o chamado **bloco const**.

Ao contrário das variáveis, as constantes devem ter seus valores definidos logo na declaração. Além disso, os tipos das constantes não são definidos explicitamente. Eles são deduzidos pelo Delphi, de acordo com o valor definido para cada constante. Veja alguns exemplos:

```
const

  Altura = 100;

  Largura = 200;

  VelocidadeMaxima = 225.17;

  Titulo = 'Ecossistemas';
```

Aqui, as constantes "Altura" e "Largura" são armazenadas com tipo *Integer*, "VelocidadeMaxima" com tipo *Real* (ou *Double*) e "Titulo" com tipo *String*.

Note como é usado o símbolo "=" para definir os valores para as constantes e não o símbolo de atribuição: ":=".

7.6. Tipos estruturados

Pode-se criar novos tipos em Object Pascal, baseando-se nos tipos predefinidos. Tipos criados a partir dos tipos básicos são chamados **Tipos Estruturados**. O exemplo mais comum de tipo estruturado é o **Array**, uma sequência de valores de mesmo tipo. A linguagem Object Pascal permite também criar **Tipos enumerados**, **Registros** e vários outros.

7.7. Tipos enumerados

A declaração de um tipo enumerado lista diretamente todos os valores que uma variável pode conter. Você define novos tipos usando a palavra-chave **type**, que deve ser colocada antes das definições dos tipos (tal como **var** e **const**). O **bloco type** deve ser posicionado antes dos blocos **var** e **const**. Veja alguns exemplos:

type

```
TDiaUtil = (Segunda, Terca, Quarta, Quinta, Sexta);
```

```
TFabricante = (Chevrolet, Ford, Volkswagen);
```

```
TSexo = (Masculino, Feminino);
```

Depois de definir os novos tipos, você pode usá-los normalmente nas declarações das variáveis, como se fossem tipos predefinidos. Veja exemplos de declarações que usam os tipos definidos acima:

var

```
DiaInicial, DiaFinal: TDiaUtil;
```

```
Fab: TFabricante;
```

```
Sexo: TSexo;
```

Nesse caso, por exemplo, a variável Fab só pode ter três valores diferentes. Qualquer valor diferente de "Chevrolet", "Ford", ou "Volkswagen" não é permitido.

Você atribui valores a uma variável de tipo enumerado normalmente. Veja alguns exemplos:

```
Fab := Ford;
```

```
Sexo := Masculino;
```

```
DiaDaSemana := Terca;
```

Note como não são usadas aspas para os valores. Isso acontece porque os valores de um tipo enumerado são armazenados como *valores inteiros*, na sequência em que são definidos (começando a partir de zero). Para o tipo *TFabricante*, por exemplo, temos Ford = 0,

Chevrolet = 1 e Volkswagen = 2. O uso de tipos enumerados em vez de inteiros pode tornar um programa muito mais legível.

7.8. Tipos de intervalo (*Subrange*)

Os tipos de intervalo são *seqüências de valores* dos tipos *Boolean*, *Char*, *Integer*, ou de tipos enumerados. Tipos de intervalo são úteis quando é necessário limitar uma variável a um intervalo contínuo de valores, com/ de 1 a 100, ou de "A" a "Z". Veja a seguir exemplos de definições de tipos de intervalo:

```
type

TGraus = 0..360;

THoraValida = 0..23;

TMaiuscula = 'A'..'Z';

var

Hora: THoraValida;

Angulo: TGraus;
```

No exemplo, a variável "hora" só pode estar entre 0 e 23 e a variável "angulo" deve estar entre 0 e 360.

Quando uma variável é definida com um tipo de intervalo, o Delphi verifica, durante a compilação, as alterações nesta variável e gera erros de compilação quando os limites do intervalo são violados. O seguinte exemplo gera um erro de compilação, na linha destacada.

```
procedure TFormTesteRange.Button1Click(Sender: TObject);

type

TGrau = 0..360;

var

Temp: TGrau;

begin

for Temp := 0 to 720 do {Erro de compilação!}

begin

... {código omitido}

end;

end;
```

A mensagem de erro gerada é "*Constant expression violates subrange bounds*", indicando que os limites do intervalo foram violados (a variável *Temp* passaria do limite, 360, no loop **for**).

No caso anterior, o Delphi pôde detectar o erro, porque foram usados valores constantes para a variável. Quando são usadas expressões mais complexas (cálculos por exemplo), o Delphi só pode detectar o erro em *tempo de execução*. O exemplo a seguir passaria na compilação, mas a linha destacada geraria uma exceção do tipo **ERangeError** (*Temp* receberia o valor 400, maior que o máximo permitido pelo tipo *TGrau*: 360).

```
procedure TFormTesteRange.Button2Click(Sender: TObject);

type

    TGrau = 0..360;

var

    Temp: TGrau;

begin

    Temp := 200;

    Temp := Temp*2; {Erro de execução!}

    ShowMessage(IntToStr(Temp));

end;
```

NOTA: O comportamento padrão do Delphi é *não* verificar erros de intervalo em tempo de execução (como o erro do exemplo anterior). Para que o Delphi realmente verifique se as variáveis estão dentro dos limites, escolha o comando **Project | Options**, mude para a página **Compiler** e ative a opção **Range Checking**.

7.9. Arrays

Os Arrays são usados com frequência, praticamente em qualquer programa. Arrays são seqüências indexadas de valores do mesmo tipo. Veja alguns exemplos de declarações de arrays:

```
var

    Nomes: array[1..100] of String;

    Salarios: array[1..50] of Real;
```

Aqui, a variável "Nomes" é uma seqüência de 100 *Strings* e "Salarios" é uma seqüência de 50 valores do tipo *Real*.

Os elementos de um array são **indexados**: para acessar um valor de um array, digite o nome do array, seguido do **índice** entre colchetes. Por exemplo: Nomes[32]. O índice pode ser qualquer expressão que tenha como resultado um valor inteiro.

O exemplo a seguir declara um array "quadrados" com 100 inteiros e depois preenche esse array com os quadrados dos números de -50 a 250:

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var

I: Integer;

Quadrados: array[1..100] of Integer;

begin

for I := -50 to 250 do

    Quadrados[I] := I*I;

end;
```

7.10. Arrays multidimensionais

Arrays multidimensionais são simplesmente "arrays de arrays". Os arrays que vimos até agora têm apenas uma dimensão. Arrays com duas dimensões (bidimensionais) são usados para armazenar **tabelas** de valores. Pode-se também definir arrays de 3 ou mais dimensões, mas estes são pouco usados e ocupam muita memória. Veja exemplos de definições de arrays multidimensionais:

```
var

TabelaGrafico: array[1..100, 1..100] of Double

ModeloEdificio: array[1..500, 1..500, 1..500] of
Integer;
```

O número de elementos de um array multidimensional é o resultado da multiplicação de todas as suas dimensões. O array "TabelaGrafico", no exemplo, tem $100 \times 100 = 10.000$ elementos, e o array "ModeloEdificio" tem $500 \times 500 \times 500 = 125$ milhões de elementos.

Para acessar um elemento de um array multidimensional, são usados vários índices (um para cada dimensão), separados por vírgulas. Por exemplo:

- TabelaGrafico[56,78]
- ModeloEdificio[150,115,82]

7.11. Strings

Strings em Object Pascal são armazenados internamente como **arrays de caracteres** (arrays do tipo Char). Há dois tipos de Strings em Object Pascal: **Strings longos** e **Strings curtos**.

Strings longos têm o comprimento praticamente ilimitado (2 GigaBytes). Eles são declarados usando somente a palavra-chave **string**. **String curtos** têm o comprimento limitado em 255 caracteres. Para declarar um String curto, use a palavra-chave **string** seguida pelo tamanho desejado, entre colchetes. Veja a seguir exemplos de declarações dos dois tipos de Strings:

```
var

Nome: String[30]; {um String curto com no máximo 30
caracteres}
```



```
Texto: String; {um String longo}
```

Como um String é um array de caracteres, pode-se acessar cada caractere do String individualmente, usando índices. Por exemplo, Nome[5] retorna o quinto caractere do String "Nome" (os caracteres são indexados a partir de 1).

7.12. Registros (*Records*)

Os registros são conjuntos de variáveis de tipos diferentes. Um registro pode ser manipulado como um único elemento e é definido usando a palavra-chave **record** (registro em inglês). Veja um exemplo que define um registro para armazenar dados sobre produtos.

type

```
TProduto = record  
  
    Codigo: String[12];  
  
    Descricao: String[50];  
  
    Preco: Currency;  
  
    Fornecedor: String[50];  
  
    Quantidade: Integer;
```

end;

Depois de definir um registro, você pode declarar uma variável com o novo tipo de registro ou até definir um array com o novo tipo, como no exemplo abaixo:

var

```
Produto: TProduto;  
  
Estoque: array[1..500] of TProduto;
```

Os elementos de um registro são chamados de *campos*. Os campos do registro "TProduto" definido acima, por exemplo, são *Codigo*, *Descricao*, *Preco*, *Fornecedor* e *Quantidade*. Para acessar um campo de uma variável do tipo registro, use o nome da variável seguido por um ponto e o nome do campo

O trecho de código abaixo, por exemplo, define valores para cada um dos campos da variável produto:

```
Produto.Preco := 259,95;  
  
Produto.Codigo := '005-002x';
```

```
Produto.Fornecedor := 'Eletrosul';

Produto.Descricao := 'Liquidificado industrial
Walita 2001';

Produto.Quantidade := 8;
```

Para evitar a repetição do nome do registro em casos como este, pode-se usar o comando **with**:

```
with produto do

begin

Preco := 259,95;

Codigo := '005-002x';

Fornecedor := 'Eletrosul';

Descricao := 'Liquidificado industrial Walita
2001';

Quantidade := 8;

end;
```

7.13. Controle de fluxo

Os programas mostrados até aqui não fazem nada além de definir novos tipos, declarar variáveis e alterá-las. Para que programas realmente "façam alguma coisa", eles devem ser capazes de tomar decisões de acordo com as entradas do usuário, ou repetir comandos, até que uma certa condição seja satisfeita. São essas operações que constituem o **controle de fluxo** de um programa.

7.14. Usando blocos

Um bloco é um conjunto de comandos delimitado pelas palavras chave **begin** e **end**. Já usamos blocos várias vezes nos exemplos mostrados até aqui. No arquivo de projeto, por exemplo, um bloco contém o código principal do projeto (onde, entre outras coisas, são criados os formulários do aplicativo). Todo programa em Object Pascal deve ter pelo menos um bloco: o bloco principal.

Quando você precisa executar ou não um *conjunto* de comandos dependendo de uma condição, você deve usar blocos para delimitar esses conjuntos. Outro uso importante de blocos é para a repetição de um conjunto de comandos várias vezes, como nos *loops for*, **while** e **repeat**, que veremos a seguir.

Blocos podem ser *aninhados*, isto é pode haver blocos dentro de blocos. Isso é útil em estruturas complexas, como **ifs** dentro de **ifs**, ou *loops* dentro de *loops*.

A estruturação em blocos da linguagem Object Pascal é uma de suas características mais elegantes e poderosas.

7.15. If-then-else

O comando mais usado para fazer decisões simples é o comando **if**. O comando **if** verifica uma condição e executa um comando ou bloco de comandos somente se a condição for verdadeira. O comando **if** é sempre usado com a palavra-chave **then**. Há várias maneiras de usar o comando **if**. Veja a mais simples a seguir:

```
if condição then  
  
    comando;
```

Aqui a *condição* é qualquer expressão que tenha valor booleano (*True* ou *False*). Condições são geralmente comparações, como **a > b**, **x = 1**, **total <= 1000**, etc. Pode-se também usar o valor de uma *propriedade* como condição. O exemplo abaixo, ativa um botão (chamado "Button2") somente se ele estiver desativado (com *Enabled=False*). Caso contrário, nada acontece.

```
if Button2.Enabled = False then  
  
    Button2.Enabled = True;
```

Há também várias funções predefinidas que retornam valores booleanos e que também podem ser usadas como condições.

7.16. Usando else

Um comando **if** pode também apresentar uma segunda parte, delimitada pela palavra-chave **else**. O comando da segunda parte (depois de **else**) é executado quando a condição é falsa. O exemplo a seguir mostra uma mensagem diferente dependendo do valor da variável "x":

```
if x > limite then  
  
    ShowMessage('Limite ultrapassado!');  
  
else  
  
    ShowMessage('Sem problemas.');
```

7.17. Usando blocos com o comando if

A versão simples do comando **if** usada nos exemplos acima tem uso muito limitado. Na maioria das vezes, é necessário executar *mais de um comando* se uma certa condição for verdadeira. Para isso você deve usar *blocos* de comandos. Veja um exemplo completo:

```
procedure TForm1.Button1Click(Sender: TObject);  
  
var  
  
    Dividendo, Divisor, Resultado: Real;  
  
begin  
  
    Dividendo := StrToFloat(Edit1.Text);  
  
    Divisor := StrToFloat(Edit2.Text);
```

```
if Divisor = 0 then  
  
begin  
  
Color := clRed;  
  
ShowMessage('Divisor inválido');  
  
end  
  
else begin  
  
Color := clSilver;  
  
Resultado := Dividendo/Divisor;  
  
ShowMessage('Resultado = ' + FloatToStr(Resultado));  
  
end;  
  
end;
```

Note os dois blocos de comandos: um para o **if-then** e outro para o **else**. O programa calcula a divisão de dois valores do tipo *Real* (lidos a partir de dois componente *Edit*). Se o divisor for zero, o primeiro bloco do **if** é executado: o formulário muda para vermelho e uma mensagem é exibida. Se o divisor não for zero, o segundo bloco (depois do **else**) é executado, mudando a cor do formulário para cinza.

As funções **StrToFloat** e **FloatToStr** convertem de Strings para números reais e de números reais para Strings, respectivamente.

Note como o **end** depois do primeiro bloco não é seguido de ponto-e-vírgula (;). Essa é uma regra obrigatória da linguagem Object Pascal: um **end** antes de um **else** nunca deve ser seguido por ";". Somente o último **end** do comando **if** deve terminar com ponto-e-vírgula.

7.18. Aninhando comandos *if*

Muitas vezes, um programa deve decidir entre mais de duas opções. Para isso, pode-se usar comandos **if** aninhados (um **if** dentro de outro). Veja um exemplo:

```
procedure TForm1.Button1Click(Sender: TObject);  
  
var  
  
Nota: Real;  
  
begin  
  
Nota := StrToFloat(Edit1.Text);  
  
if nota < 5.0 then  
  
ShowMessage('Reprovado por média')  
  
else
```

```
if (Nota >= 5.0) and (Nota < 7.0) then  
  
    ShowMessage('Aprovado na final')  
  
else  
  
    if Nota > 8.5 then  
  
        ShowMessage('Aprovado com excelência')  
  
    else  
  
        ShowMessage('Aprovado simplesmente')  
  
end;
```

O exemplo lê uma nota digitada pelo usuário e mostra uma mensagem que varia de acordo com o valor da nota. Acompanhe o fluxo do programa: se a nota for menor que 5, a primeira mensagem é exibida; caso contrário (qualquer valor maior ou igual a 5), o programa checa a segunda condição: se a nota estiver entre 5 e 7, a segunda mensagem é exibida. Se não, o programa verifica se a nota é maior que 8.5 e, se for, mostra a terceira mensagem. A mensagem depois do último **else** só é exibida se *nenhuma* das condições anteriores for verdadeira (isso acontece se a nota estiver entre 7 e 8.5).

NOTA: no programa acima foi usada a palavra chave **and** para criar uma condição mais complexa. Há mais duas palavras-chaves semelhantes: **or** e **not**. Pode-se usar qualquer combinação de **not**, **and** e **or** para criar novas condições. Lembre-se, entretanto de usar parênteses para agrupar as condições. Os parênteses usados no programa anterior, por exemplo, são todos obrigatórios.

7.19. A estrutura *case*

Quando é necessário decidir entre muitas opções em um programa, a estrutura **case** da linguagem Object Pascal é em geral mais clara e mais rápida que uma sequência de **ifs** aninhados. Uma estrutura **case** só pode ser usada com valores **ordinais** (inteiros, caracteres, tipos de intervalo e tipos enumerados). Na estrutura **case** uma variável é comparada com vários valores (ou grupos de valores) e o comando (ou bloco de comandos) correspondente é executado.

Veja a seguir um exemplo simples, que mostra mensagens diferentes, de acordo com o número entrado em um *Edit*.

```
procedure TForm1.Button1Click(Sender: TObject);  
  
var  
  
    Numero: Integer;  
  
begin  
  
    Numero := StrToInt(Edit1.Text);  
  
    case Numero of  
  
        1, 2, 3, 5, 7 :
```

```
begin

    ShowMessage('Número primo menor que 10');

    Color := clBlue;

end;

4, 6, 8: ShowMessage('Número par menor que dez');

9: ShowMessage('Número ímpar menor que dez');

10..1000: ShowMessage('Número entre 10 e 1000');

else

    ShowMessage('Outro número');

end; // final do case

end; // final da procedure
```

Uma estrutura **case** deve ser terminada com **end** e pode ter uma parte opcional **else**. Os comandos depois do **else** só são executados se a variável não for igual a nenhum dos valores especificados (se não houver a parte **else**, a estrutura **case** é pulada inteiramente). No exemplo acima, a mensagem "Outro número" só é mostrada se o valor da variável "numero" for maior que 1000 ou menor que 1.

7.20. Usando loops

Os loops ("laços") são estruturas usadas para repetir várias vezes sequências de comandos. Há três tipos de loops em Object Pascal. O loop **for** é usado para realizar um número fixo de repetições. Os loops **while** e **repeat** repetem um bloco de comandos até que uma condição se torne falsa ou verdadeira, respectivamente.

7.21. O loop *for*

O loop **for** é o mais rápido e mais compacto dos três tipos de loops. Esse loop usa um *contador*, uma variável inteira que é aumentada (*incrementada*) automaticamente cada vez que o loop é executado. O número de repetições do loop **for** é fixo. Ele depende somente do valor inicial e do valor final do contador, que são definidos no início do loop. Veja um exemplo:

```
procedure TForm1.Button1Click(Sender: TObject);

var

    I: Integer;

    Numero: Integer;

begin

    Numero := StrToInt(Edit1.Text);
```

```
for I := 1 to 50 do  
  
    ListBox1.Items.Add(IntToStr(Número*I));  
  
end;
```

Este programa mostra os primeiros 50 múltiplos do número digitado em um *Edit*. Os múltiplos são adicionados a um *ListBox*. O loop **for** usado aqui contém um único comando. Por isso não é necessário usar **begin** e **end**.

O exemplo a seguir é mais complexo. Nele são usados loops **for** aninhados para multiplicar 100 valores por 20 taxas diferentes, gerando uma tabela – um array bidimensional – com o "cruzamento" de todos os valores e taxas. É calculado também o total dos valores da tabela gerada. Vários outros loops **for** simples são usados. Note que uma mesma variável pode ser usada várias vezes em loops **for** diferentes.

```
procedure TForm1.Button1Click(Sender: TObject);  
  
var  
  
    i,j: Integer;  
  
    ValorInicial, Total: Currency;  
  
    Taxas: array[1..20] of Double;  
  
    Valores: array[1..100] of Currency;  
  
    Tabela: array[1..20, 1..100] of Currency;  
  
begin  
  
    ValorInicial := StrToFloat(Edit1.Text);  
  
    {Calcular taxas}  
  
    for i:= 1 to 20 do  
  
        Taxas[i] := i/100;  
  
        {Calcular valores}  
  
        for i:= 1 to 100 do  
  
            Valores[i] := ValorInicial + i*100;  
  
            {Multiplicar valores por taxas e somar resultados}  
  
            Total := 0;  
  
            for i:= 1 to 100 do  
  
                for j := 1 to 20 do  
  
begin
```

```
Tabela[i][j] := Valores[i]*Taxas[j];  
  
Total := Total + Tabela[i][j];  
  
end;  
  
end;
```

7.22. O loop *while*

O loop **while** é um loop mais versátil que o **for**, embora um pouco mais complexo. O loop **while** repete um comando, ou um bloco de comandos, até que a condição especificada se torne falsa. O número de repetições não é preestabelecido como no loop **for**.

No exemplo a seguir, o valor digitado em um *Edit* é lido, convertido e colocado em uma variável "x". Em seguida, é subtraído 10 do valor de "x" e adicionado o resultado (convertido em um string) a um *ListBox*, repetindo o processo até que "x" seja menor ou igual a 0.

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
  x: Integer;  
begin  
  x := StrToInt(Edit1.Text); {Ler valor inicial para x}  
  ListBox1.Items.Clear; {Limpar a lista de valores}  
  while x > 0 do  
  begin  
    x := x - 10;  
    ListBox1.Items.Add(IntToStr(x));  
  end;  
end;
```

No loop **while**, a condição é testada no início de cada repetição. Quando a condição é falsa, o programa sai do loop e executa o comando imediatamente depois. Se a condição testada for verdadeira, o loop é executado novamente.

Note que é possível que os comandos de um loop **while** não sejam executados **nenhuma** vez. Isso acontece quando a condição é *falsa* já na entrada do loop. No programa acima, por exemplo, um valor inicial negativo para a variável "x" faria com que o **while** fosse pulado inteiramente na primeira vez.

Outra característica importante do loop **while**, é que a condição usada no loop deve se tornar falsa em algum momento. Caso contrário, o loop é executado para sempre. O programa entra em um "loop infinito". Veja um exemplo de um loop infinito com **while**:

```
x:= 1;  
  
while x > 0 do  
  
  x := x + 1;
```

Aqui, a condição **x > 0** nunca se tornará falsa (x será sempre maior que zero), e o programa ficará preso para sempre dentro do **while**.

Esse é um erro cometido com frequência, principalmente em loops **while** com condições mais complexas. Tenha cuidado, portanto, de alterar pelo menos uma das variáveis envolvidas na condição, *dentro* do **while**.

7.23. O loop repeat

O loop **repeat** é uma versão "invertida" do loop **while**. Nele, a condição é verificada somente no **final** do loop. Isso faz com que o loop seja sempre executado, **pelo menos uma vez**. A condição do loop **repeat** aparece no final, ao lado da palavra-chave **until** ("até"). Além disso, devido à estrutura simples do loop **repeat**, os comandos do loop não precisam ser delimitados por **begin** e **end**. Veja um exemplo:

```
procedure TForm1.Button1Click(Sender: TObject);

var

Entrada: String;

Valor: Integer;

begin

repeat

Entrada:= (InputBox('', 'Entre com um número de 1 a
100', ''));

Valor:= StrToInt(entrada);

until (Valor >= 1) and (Valor <= 100);

ShowMessage('Entrada aceita.');
```

```
end;
```

O código mostra uma *InputBox* várias vezes, até que a entrada seja válida (entre 1 e 100). Note que a *InputBox* é exibida pelo menos uma vez, pois é usado um loop **repeat**.

Se **while** tivesse sido usado em vez de **repeat**, o código seria mais longo e menos elegante:

```
procedure TForm1.Button1Click(Sender: TObject);

var

Valor: Integer;

Entrada: string;

begin

Entrada:=InputBox('', 'Entre com um número de 1 a 100', '');

Valor := StrToInt(entrada);

while (Valor < 1) or (Valor > 100) do
```

```
begin

Entrada := InputBox('', 'Entre com um número de 1 a
100', '');

Valor := StrToInt(Entrada);

end;

ShowMessage('Entrada aceita.');
```

```
end;
```

Este código tem exatamente o mesmo efeito que o anterior. Mas note que a condição usada no loop **while** é o oposto da condição do loop **repeat**. Além disso, há uma repetição das linhas de código que lêem a entrada. Isso é necessário porque a variável "valor" é testada no início do loop e precisa ser inicializada primeiro.

Na verdade, qualquer programa que use **repeat** pode ser convertido em um programa usando **while**, fazendo-se mudanças semelhantes às mostradas aqui. O loop **repeat** é apenas uma conveniência, que pode algumas vezes tornar o código mais simples.

7.24. O comando *break*

O comando **break** é usado para sair imediatamente de um loop (**while**, **for** ou **repeat**). Veja um exemplo simples:

```
var

A,B: array[1..1000] of Integer;

i: Integer;

...

for i:= 1 to 1000 do

if A[i] <> B[i] then break;

if i = 1000 then

ShowMessage('Todos os elementos são iguais')

else

ShowMessage('O elemento com índice ' +

IntToStr(i) + ' é diferente');

...


```

O exemplo compara os elementos de dois arrays **A** e **B**, de 1000 elementos cada um (o cálculo para os valores dos arrays não é mostrado). Quando é encontrada a primeira diferença nos arrays, o comando **break** interrompe o loop **for** imediatamente e os outros elementos dos arrays não são checados. Se todos os elementos forem iguais, o valor da variável **i** no final do loop será 1000 e a primeira mensagem é mostrada. Se o loop for quebrado antes de terminar,

com o comando **break**, o valor de **i** será menor que 1000 e a segunda mensagem será mostrada.

7.25. O comando *continue*

O comando **continue** é usado dentro de loops para forçar a próxima execução do loop. Quando o comando **continue** é encontrado, o programa volta imediatamente para o início do loop e os comandos depois de **continue** não são executados.

Veja um exemplo que adiciona todos os valores diferentes de zero de um array a um *ListBox*.

```
var  
  
A: array[1..100] of Integer;  
  
i: Integer;  
  
...  
  
for i:= 1 to 100 do  
  
begin  
  
if A[i] = 0 then continue  
  
ListBox1.Items.Add(IntToStr(i));  
  
end;
```

Quando o elemento **A[i]** do array for zero, o comando **continue** faz com que o loop seja repetido imediatamente (voltando para o começo do loop). Com isso, o segundo comando dentro do loop só é executado quando **A[i]** é diferente de zero.

7.26. Procedures e Functions

As **procedures** (procedimentos) e **functions** (funções) são os blocos básicos dos programas criados em Object Pascal. Procedures e functions são blocos independentes de código, que podem ser chamados várias vezes, em vários pontos de um programa. Procedures e functions são às vezes chamadas de **rotinas**. A diferença entre procedures e functions é simples: functions *retornam um valor*; procedures não.

O exemplo mais comum de procedure no Delphi é o código gerado para os eventos de um componente. Todos os eventos são procedures. Veja mais um exemplo de um evento:

```
procedure TFormCadastro.BtCadastrarClick(Sender: TObject);  
  
begin  
  
if (EditNome.Text='') or (EditSobrenome.Text='') then  
  
ShowMessage('Campos não podem ser vazios')  
  
else begin
```

```
...  
  
ShowMessage('Sucesso no cadastro.');
```

end;

end;

Uma procedure ou function pode ter zero ou mais parâmetros. No caso dos eventos, o parâmetro mais comum é o parâmetro "Sender", que indica o componente que gerou o evento.

7.27. Sintaxe das procedures e functions

A sintaxe para uma procedure é a seguinte:

procedure *nome da procedure* (*param1, param2, ...:tipo1; param1,...:tipo2; ...*);

type

{definições de tipos}

var

{declarações de variáveis }

const

{definições de constantes}

begin

{ corpo }

end;

Para functions, a sintaxe é praticamente idêntica. A única diferença é que o tipo do valor retornado deve ser especificado no final da primeira linha.

function *nome da function* (*param1, param2, ...:tipo1; param1,...:tipo2; ...*): *tipo de retorno*;

type

{definições de tipos}

var

{declarações de variáveis }

const

{definições de constantes}

begin

{corpo}

end;

A primeira linha, chamada de **cabeçalho**, deve sempre terminar com ";". Os parâmetros, se existirem, devem ter seus tipos especificados e devem ser separados por vírgulas. Parâmetros do mesmo tipo podem ser agrupados; esses grupos devem ser separados por ";".

Depois do cabeçalho, vêm os blocos de declaração **type**, **var** e **const** (não necessariamente nessa ordem). Todos estes são *opcionais*, pois a procedure ou functions pode não precisar de nenhuma variável além das especificadas nos seus parâmetros. Em seguida, vem o bloco principal (ou **corpo**) da procedure, delimitado por **begin** e **end**. É neste bloco que fica o código executável da procedure.

7.28. Entendendo parâmetros

Os parâmetros são variáveis passadas para as procedures e functions (rotinas). Os parâmetros de uma rotina são opcionais. São comuns rotinas (principalmente procedures) sem nenhum parâmetro.

Quando são usados parâmetros, estes podem ser passados **por valor** ou **por referência**. A forma como os parâmetros são passados é definida no cabeçalho da rotina.

Quando uma variável é passada **por valor**, a rotina recebe apenas um *cópia* da variável. A variável passada se comporta como uma variável local. Alterações realizadas na variável não têm efeito depois da rotina terminar (ou retornar). A passagem por valor é o tipo padrão de passagem de parâmetros na linguagem Object Pascal.

Quando uma variável é passada **por referência**, a rotina recebe uma *referência* à variável passada ou, em outras palavras, a *própria variável*. Com esse tipo de passagem de parâmetros, a rotina pode alterar *diretamente* a variável passada. Para especificar a passagem por referência, use a palavra-chave **var** antes do nome do parâmetro, no cabeçalho da rotina.

Veja um exemplo que usa os dois tipos de passagem de parâmetros:

```
procedure Descontar(Taxa: Real; var Preco: Currency);  
  
begin  
  
    Preco := Preco - Preco*(Taxa/100);  
  
end;
```

Esta procedure recebe dois parâmetros: *Taxa* e *Preco*. O parâmetro *Taxa* é passado *por valor*. O valor de *Taxa* pode ser alterado dentro da procedure, mas as alterações não terão efeito depois de a procedure retornar.

O parâmetro *Preco* é passado por referência (usando-se a palavra **var** antes do parâmetro). *Preco* é alterado dentro da procedure e essa alteração é permanente.

7.29. Definindo o valor de retorno de uma function

O valor retornado por uma function pode ser definido de duas maneiras: atribuindo um valor para o nome da function, ou alterando a variável especial **Result**. As duas funções a seguir são exatamente equivalentes:

```
function Dobro1(Numero: Integer): Integer;  
  
begin  
  
    Dobro := Numero*2;  
  
end;
```

```
function Dobro2(Numero: Integer): Integer;  
  
begin  
  
    Result := Numero*2;  
  
end;
```

7.30. Chamando procedures e functions

Depois de definir uma procedure ou function, você pode chamá-la diretamente no seu código, em qualquer lugar da Unit em que foi definida.

Para chamar uma procedure ou function, basta especificar o seu nome e valores para seus parâmetros (se houver). O seguinte trecho de código mostra como chamar a procedure *Descontar* (definida acima) para realizar um desconto de 10% sobre um preço *p*. (O valor de *p* no final será 450.0).

```
...  
  
p := 500.0;  
  
Descontar(10, p);  
  
...
```

7.31. Onde criar procedures e functions

Há regras para o posicionamento das procedures no código: as procedures devem ser colocadas somente depois da parte *implementation* da Unit (veja a seguir), depois das declarações das variáveis globais.

O exemplo a seguir usa a procedure *Descontar* e uma variação da function *Dobrar*, dos exemplos anteriores. Os valores de cem preços, armazenados no array *Precos*, são alterados no evento *OnClick* do componente *Button1*. (O programa, apesar de completo, foi mantido deliberadamente simples, para ilustrar melhor os conceitos envolvidos).

```
{Código inicial da Unit}  
  
...  
  
implementation
```

```
var

Precos: array[1..100] of Currency;

procedure Descontar(Taxa: Double; var Preco: Currency);

begin

Valor := Preco - Preco*(taxa/100);

end;

function Dobrar(Preco: Currency): Currency

begin

Result := Preco*2;

end;

procedure TForm1.Button1Click(Sender: TObject);

var

i: Integer;

begin

// Descontar em 25% os primeiros 20 produtos...

for i := 1 to 20 do

Descontar(25, Precos[i]);

//...descontar em 50% os próximos 30...

for i := 21 to 50 do

Descontar(50, Precos[i]);

//...e dobrar o preço dos 50 produtos restantes

for i := 51 to 100 do

Precos[i] := Dobrar(Precos[i]);

end;

end.
```

7.32. Trabalhando com Exceções

As exceções são um mecanismo poderoso para lidar com erros nos seus programas. O uso de exceções permite uma separação entre o código normal de um programa e o código usado para lidar com erros que podem surgir.

Quando ocorre um erro em um programa, o Delphi *levanta* uma exceção. Se uma exceção que foi levantada não for *tratada*, o programa é interrompido e é mostrada uma caixa de diálogo com uma descrição da exceção. Uma exceção não-tratada pode causar danos aos seus dados, deixar o programa em uma situação instável, ou até levar o programa a "travar".

Para evitar que isso aconteça, você deve tratar as exceções, **protegendo** blocos de código que contenham comandos que possam causar erros. Para proteger um bloco de código cerque-o com as palavras-chave **try** e **end**. Dentro de um **bloco protegido**, pode-se usar as comandos **except** ou **finally**, para tratar as exceções. Há dois de tipos blocos protegidos:

try

{comandos que podem levantar exceções }

except

{comandos executados quando uma exceção é levantada}

end;

try

{comandos que podem levantar exceções }

finally

{comandos sempre executados, havendo exceções ou não}

end;

O primeiro tipo de bloco protegido é chamado de bloco **try-except**. O segundo, de bloco **try-finally**.

No bloco **try-except**, os "comandos protegidos" ficam entre as palavras-chave **try** e **except**. Quando uma exceção ocorre em um desses comandos, o programa pula imediatamente para o primeiro comando depois de **except**.

No caso do bloco **try-finally**, os comandos depois de **finally** são executados sempre (mesmo quando exceções não são levantadas). Pode-se usar esses comandos para realizar operações de "limpeza", como destruir componentes que não são mais necessários, por exemplo.

Há vários tipos de exceções. Veja a seguir um exemplo que trata a exceção **EDivByZero**, gerada quando é feita uma divisão por zero (o divisor aqui é o valor contido no componente *Edit1*). Assume-se que os valores dos arrays *resultado* e *valores* foram declarados e inicializados antes.

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
Divisor: Real;
```

```
begin
```



```

Divisor:= StrToFloat(Edit1.Text);

for i:= 1 to 50 do

try

Resultado[i] := Valores[i]/Divisor;

except

on EDivByZero do Resultado[i] := 0;

end;

end;

```

Note o uso da palavra-chave **on** dentro do bloco **except**. **On** é usada para especificar o *tipo* de exceção a ser tratado. Um bloco **except** pode lidar com vários tipos de exceção, definindo-se um bloco

```
on [tipo de exceção] do [comandos]
```

para cada tipo.

Na tabela abaixo, listamos alguns tipos comuns de exceções:

Tipo de exceção	Quando ocorre
<i>EDivByZero</i>	Quando se tenta dividir um inteiro por zero.
<i>EZeroDivide</i>	Quando se tenta dividir um número real (de ponto flutuante) por zero.
<i>EConvertError</i>	Quando é feita uma conversão ilegal de tipos.
<i>EInOutError</i>	Quando há um erro de entrada ou saída.
<i>EDatabaseError</i>	Quando ocorre um erro geral de banco de dados.
<i>EDBEngineError</i>	Quando há um erro no BDE (<i>Borland Database Engine</i>).

7.33. Entendendo o código das Units

As Units contêm praticamente todo o código de um aplicativo no Delphi. As Units geralmente são associadas a formulários, mas podem ser criadas de forma totalmente independente. Nesta seção veremos detalhes sobre cada parte de uma Unit.

7.34. A estrutura básica de uma Unit

Todas as Units têm a mesma estrutura básica:

```

unit <nome da unit>

interface

```

```
uses <lista de Units>

implementation

uses <lista de Units>

{código para os procedures e functions}

initialization {opcional}

{código para inicialização}

finalization {opcional}

{codigo para finalização}

end.
```

A primeira linha da Unit identifica o **nome** da Unit (que deve ser um identificador válido na linguagem Object Pascal). Veja a seguir descrições sobre cada uma das partes.

7.34.1. Parte Interface

A parte **interface** começa com a palavra **interface** e termina imediatamente antes da palavra-chave **implementation**. Esta parte contém a definição de todos os tipos, constantes, variáveis, procedures e funnctions que devem ser "visíveis" (ou acessíveis) para outras Units que se referenciem a esta. Somente o que estiver definido na parte **interface** pode ser acessado por outras Units.

A primeira cláusula **uses** fica dentro da parte **interface**. As Units listadas nessa cláusula são geralmente adicionadas pelo próprio Delphi, para fazer referência a Units do sistema (predefinidas). É rara a necessidade de adicionar manualmente nomes de Units a essa cláusula **uses**. Para se referenciar a outras Units, altera-se a cláusula **uses** na parte **implementation**.

7.34.2. Parte Implementation

A parte **implementation** contém todo o código das procedures e functions da Unit. Esta parte pode conter também declarações de variáveis e constantes, mas estas só serão "visíveis" pelo código desta Unit. Nenhuma outra Unit pode ter acesso às variáveis e constantes declaradas aqui.

A parte **implementation** contém a segunda cláusula **uses**. As Units listadas nessa cláusula são geralmente adicionadas pelo programador, manualmente, ou usando o comando **File | Use unit**.

7.34.3. Parte Initialization

A parte **initialization** é opcional. Você pode usar a parte **initialization** para declarar e inicializar variáveis, por exemplo. O código nesta parte é executado antes de qualquer outro código na Unit. Se um aplicativo tiver várias Units, a parte **initialization** (se houver) de *cada* Unit é executada antes de qualquer outro código na Unit.

7.34.4. Parte finalization

A parte **finalization** também é opcional. O código nessa parte é executado logo antes do término do aplicativo. Essa parte é geralmente usada para realizar "operações de limpeza", como recuperar memória e outros recursos, ao final da execução do aplicativo.

7.35. Rotinas úteis

Nas seções a seguir, apresentaremos algumas rotinas (functions e procedures) úteis na programação em Object Pascal.

7.36. Rotinas para manipulação de strings

A manipulação de strings é uma tarefa muito comum em programação. A seguir, listamos as principais rotinas de manipulação de strings oferecidas pelo Delphi. (As rotinas cujos nomes começam com "Ansi" são capazes de lidar com caracteres acentuados e, portanto, são muito úteis para strings em português).

Função	Descrição
AnsiCompareStr (S1, S2: string): Integer AnsiCompareText (S1, S2: string): Integer	A função AnsiCompareStr compara S1 to S2, levando em conta maiúsculas e minúsculas. A função retorna um valor menor que zero se $S1 < S2$, zero se $S1 = S2$, ou um valor maior que 0 se $S1 > S2$. A função AnsiCompareText é semelhante, mas ignora maiúsculas e minúsculas.
AnsiLowerCase (S: string): string	A função AnsiLowerCase retorna o string S convertido para <i>minúsculas</i> (inclusive para letras acentuadas).
AnsiUpperCase (S: string): string	A função AnsiUpperCase retorna o string S convertido para <i>maiúsculas</i> (inclusive para letras acentuadas).
AnsiPos (Substr: string ; S: string): Integer	A função AnsiPos retorna um inteiro com a posição do string <i>Substr</i> no string S. Se <i>Substr</i> não for encontrado, a função retorna 0.
Copy (S: string ; Indice, Comp: Integer): string	A função Copy retorna um string contendo <i>Comp</i> caracteres, começando com <i>S[Indice]</i> .
Delete (var S: string ; Indice, Comp: Integer)	A procedure Delete remove o <i>substring</i> com <i>Comp</i> caracteres do string S, começando com <i>S[Index]</i> . O string resultante é retornado na variável S.
Length (S: string): Integer	A função Length retorna o número de caracteres no string S.
Trim (S: string): string	A função Trim remove espaços à esquerda e à direita do string S.

7.37. Funções de conversão de tipo

A linguagem Object Pascal é especialmente exigente com relação aos tipos usados em expressões, e nas chamadas de procedures e functions. Se o tipo usado não for compatível com o tipo esperado, é gerado um erro de compilação. **Funções de conversão de tipo** devem ser usadas para converter valores para os tipos adequados.

As principais funções de conversão de tipo são listadas na tabela a seguir. Se uma conversão for ilegal, essas funções levantam a exceção **EconvertError**.

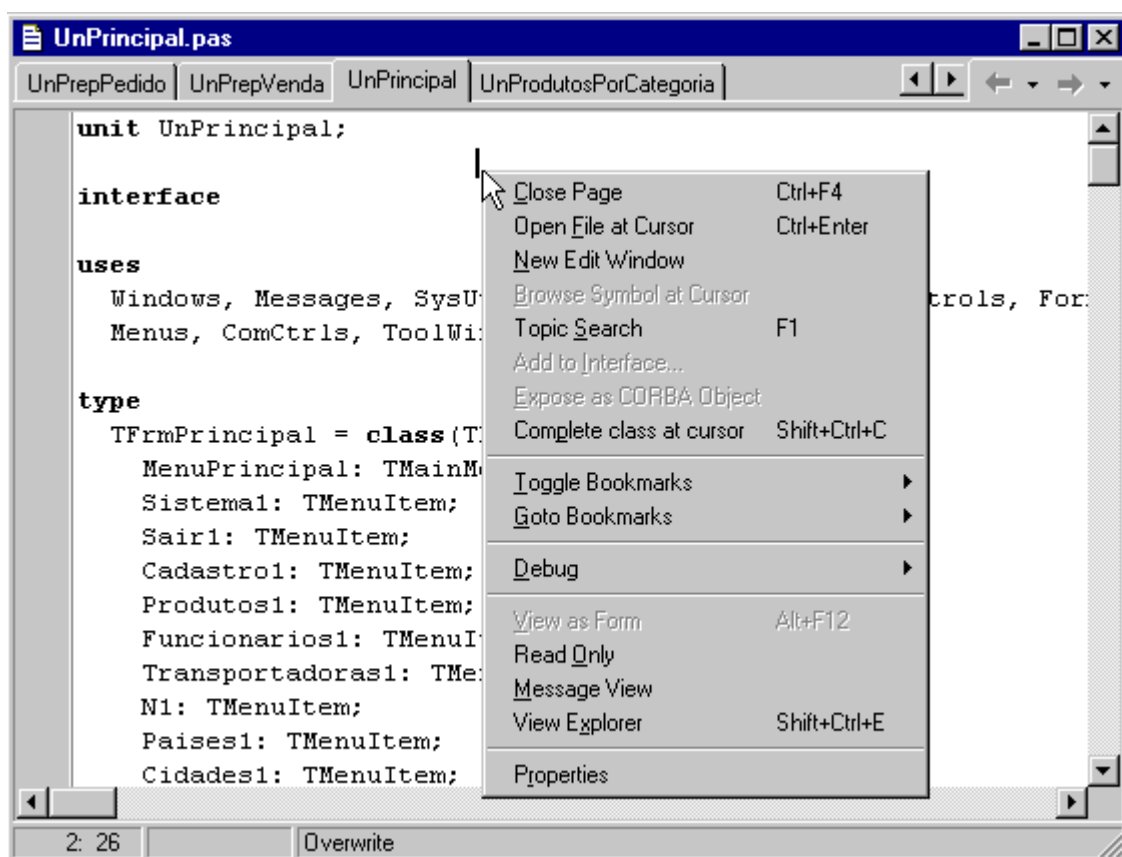
Função	Tipo de origem	Tipo de destino
<i>StrToCurr</i>	String	Currency
<i>StrToDate</i>	String	TDate
<i>StrToDateTime</i>	String	TDateTime
<i>StrToFloat</i>	String	Real
<i>StrToInt</i>	String	Integer
<i>StrToTime</i>	String	TTime
<i>IntToStr</i>	Integer	String
<i>CurrToStr</i>	Currency	String
<i>DateToStr</i>	TDate	String
<i>TimeToStr</i>	TTime	String
<i>DateTimeToStr</i>	TDateTime	String

CAPÍTULO 08 - O Editor de Código

O Editor de Código (*Code Editor*) é o editor padrão do Delphi. Neste capítulo, veremos como configurar e usar os recursos desse versátil editor.

8.1. Visualizando arquivos

Cada arquivo aberto no Editor de Código tem seu título exibido em uma "aba" no topo da janela do editor. Pode-se alternar de um arquivo aberto para outro facilmente, clicando na aba correspondente. A janela do Editor de Código é mostrada abaixo, com vários arquivos abertos. É mostrado também o menu de atalho do Editor de Código, que contém vários comandos úteis. Para abrir esse menu de atalho, clique com o botão direito na parte interna (parte do texto) do Editor de Código.



Pode-se também usar o teclado para passar de um arquivo para outro. Use **CTRL+TAB** para mover para o próximo arquivo (na ordem das abas) e **CTRL+SHIFT+TAB** para mover para o arquivo anterior.

Além das Units e outros arquivos internos do Delphi, o Editor de Código é capaz de abrir qualquer arquivo texto, até outros escritos em outras linguagens. (Mas nesse caso, é claro, o arquivo não significará nada para o Delphi e não poderá ser compilado, ou alterado automaticamente).

Há várias maneiras para abrir arquivos no Editor de código. Já conhecemos algumas, mas todas as maneiras são descritas aqui brevemente, por conveniência.

- Para abrir ou visualizar um arquivo no Editor de Código, realize uma das seguintes operações:

- Escolha o comando **File | Open** para abrir um **arquivo existente**.

Os arquivos padrão especificados são as Units (**.PAS**) e os arquivos de projeto (**.DPR**). Se for necessário abrir outro tipo de arquivo, especifique seu tipo em "Arquivos do tipo".

- Clique com o botão direito em cima do **nome de um arquivo** (dentro do Editor de Código) e escolha o comando **Open File at Cursor** ("Abrir arquivo na posição do cursor").

Esse comando é útil para código que trabalha diretamente com arquivos. Você pode usá-lo para verificar se o arquivo realmente existe com o nome e local indicado no código. O nome do arquivo deve estar especificado com o caminho (*path*) completo; caso contrário, o Delphi pode não conseguir encontrá-lo.

- Para visualizar **o código associado a um formulário** (a Unit associada), selecione o formulário e pressione **F12**. Pressione **F12** novamente para voltar ao formulário.
- Para visualizar **o arquivo de projeto** escolha o comando **View | Project Source**.

O Editor de código cria mais uma *página* para mostrar o arquivo de projeto (não é aberta uma nova janela).

A seguir, listamos algumas outras operações básicas que podem ser realizadas no Editor de código.

- **Para fechar um arquivo no Editor de Código:**

1. Mude para a página do arquivo e clique dentro dela com o botão direito.
1. Escolha o comando **Close Page**. Há também um atalho para isso: **CTRL + F4**. Se o arquivo não tiver sido salvo, o Delphi mostra uma caixa de confirmação.

Note que não é possível fechar uma página simplesmente clicando na sua aba com o botão direito. Para fechar uma página no Editor de Código, você deve mostrá-la primeiro.

- **Para abrir uma nova janela para um mesmo arquivo:**
- Mude para a página do arquivo, clique com o botão direito e escolha o comando **New Edit Window**.

É criada uma nova janela com uma cópia do arquivo. As duas janelas (a original e a cópia) são sincronizadas. Alterações em uma janela afetam a outra também. Esse comando é útil quando é necessário visualizar ou trabalhar com partes diferentes de um mesmo arquivo.

- **Para proteger o texto de um arquivo contra alterações:**
- Mude para a página que contém o arquivo, clique com o botão direito e escolha o comando **Read Only**.

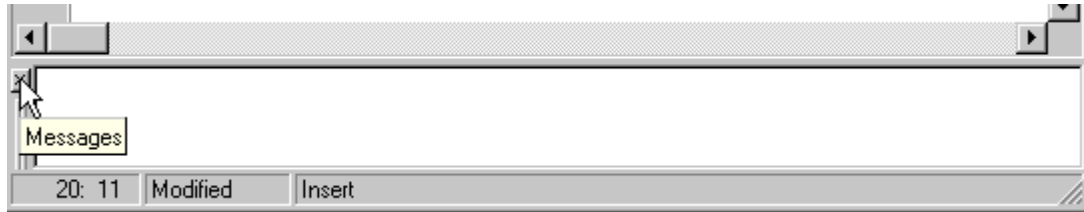
Isso faz com que o arquivo se torne "Somente leitura", não permitindo alterações acidentais. Para desproteger o arquivo, escolha o mesmo comando novamente.

- **Para exibir uma janela de mensagens na parte de baixo do Editor de Código:**
- Clique com o botão direito em qualquer local de dentro do Editor de Código e escolha o comando **Message View**.

Essa janela é exibida durante a compilação para exibir mensagens de erro ou avisos.

- **Para esconder a janela de mensagens:**

Clique no pequeno "x" no canto esquerdo superior da janela de mensagens (veja figura).



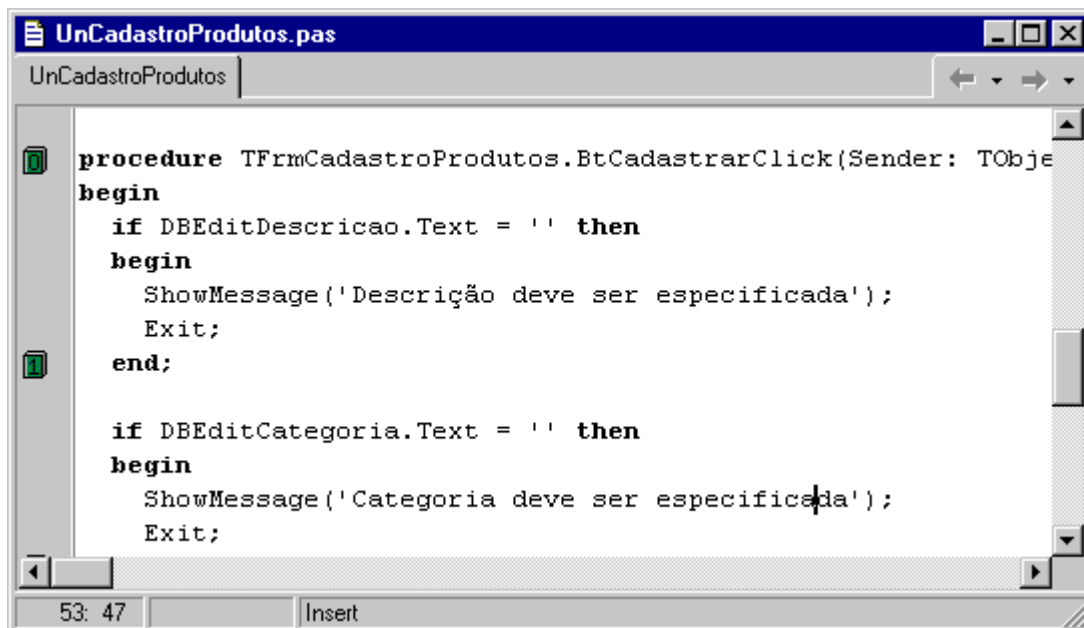
8.2. Técnicas de navegação

Há algumas técnicas que tornam mais rápido o trabalho com várias Units relacionadas, ou com Units extensas no Editor de Código. Veja as mais importantes dessas técnicas a seguir:

8.3. Usando Bookmarks

Você pode marcar linhas do código com *bookmarks*, para depois retornar a essas linhas rapidamente, usando o teclado ou comandos de menu. Pode-se definir até dez bookmarks para cada Unit.

Linhas com um bookmark definido aparecem com o número do bookmark (dentro de um quadrado verde) na margem esquerda do Editor de Código (veja figura).



O Editor de Código com dois bookmarks definidos

Veja como realizar as operações básicas com bookmarks:

- **Para definir um bookmark para uma linha de código**
- Coloque o cursor na linha e pressione CTRL+SHIFT+número, com *número* entre 0 e 9 (como CTRL+SHIFT+0 ou CTRL+SHIFT+5).

(Se já houver um bookmark com o mesmo número definido, ele é movido para a linha atual)

- **Para pular para um linha com um bookmark definido**
- Pressione CTRL+número, onde *número* é o número do bookmark para onde você deseja pular.

(Se não houver um bookmark com este número, nada acontece).

- **Para remover um bookmark**
- Mova o cursor para a linha com o bookmark e pressione CTRL+SHIFT+número, onde *número* deve ser o número do bookmark a ser removido.

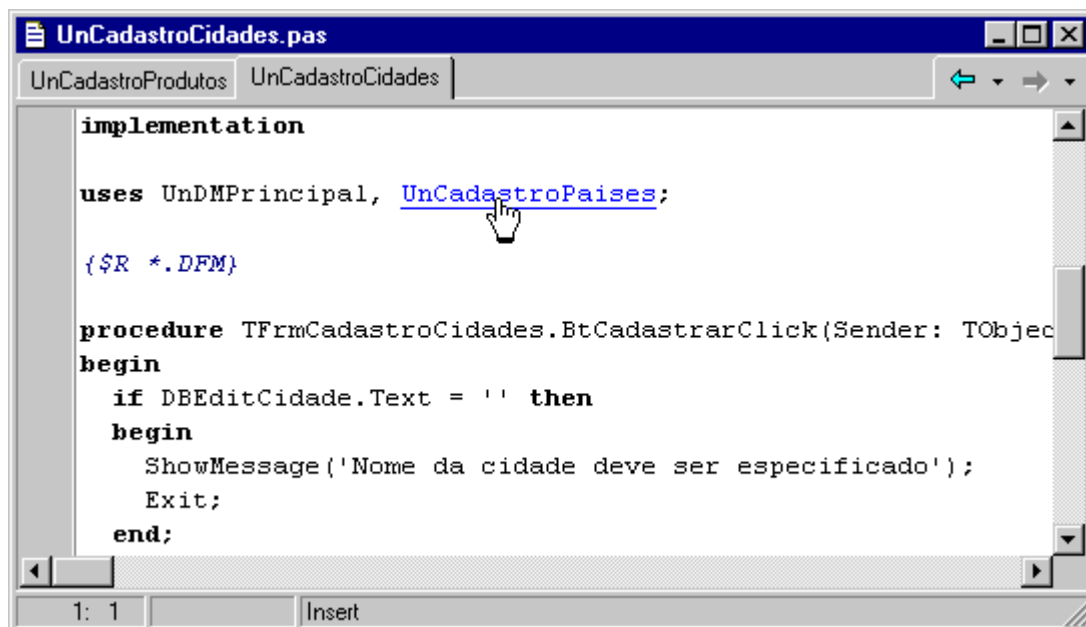
(Se for usado um número diferente, um novo bookmark é definido "por cima" do anterior).

NOTA: pode-se definir bookmarks também usando comandos do menu de atalho do Editor de código. Clique com o botão direito dentro do editor e use os comandos **Toggle Bookmarks** e **Goto bookmarks**.

8.4. Usando o recurso *Code Browser*

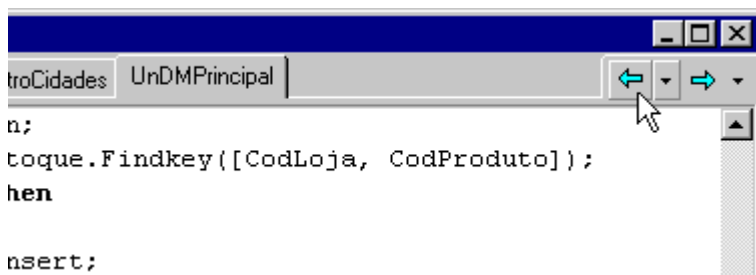
O recurso *Code Browser* permite navegar rapidamente entre vários arquivos relacionados. Você pode, por exemplo, abrir o arquivo que tenha a definição de uma procedure ou function, com apenas um clique do mouse.

Para usar esse recurso, no Editor de Código, pressione e mantenha pressionada a tecla CTRL e mova o cursor. Quando o cursor apontar para uma parte "navegável" do código, como o nome de uma Unit ou formulário, essa parte aparece sublinhada em azul, como um *link* na Internet (veja a figura).



Clique na parte sublinhada para exibir o trecho de código relacionado. Se o código estiver em outro arquivo (e o Delphi puder encontrá-lo), o arquivo é automaticamente aberto no Editor de Código. Se a parte sublinhada for uma variável ou nome de componente, o Delphi move o cursor para a declaração da variável ou componente.

Quando você navega para vários arquivos ou locais diferentes, são ativados os controles no canto superior direito da janela de Editor de Código (veja figura).



Os controles de navegação do Editor de Código

Use a seta para a esquerda para pular para o *último* local visitado e a seta para a direita para o *próximo* (a seta para a direita só é ativada se a seta para a esquerda for usada ao menos uma vez). O funcionamento é semelhante aos botões "Back" e "Forward" dos browsers para a Internet.

Você também pode pular para um local visitado clicando ao lado das setas, e escolhendo um local a partir da lista exibida.

8.5. Localizando e substituindo

O Delphi oferece um conjunto de recursos poderosos para a localização e a substituição de trechos de código em seus programas.

8.6. Localizando textos no arquivo atual

O Delphi oferece todas as opções comuns para localização de textos. Você define o texto a ser localizado e várias outras opções usando o comando **Search | Find**, que exibe a seguinte caixa de diálogo:

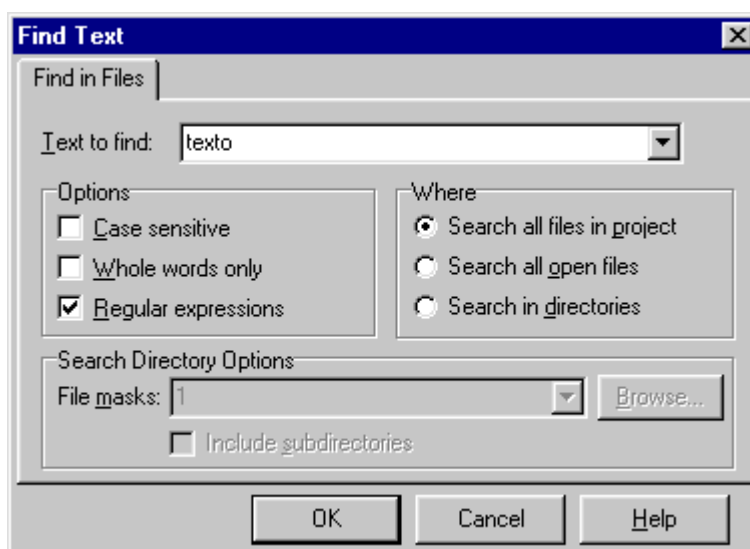


Para localizar um trecho de texto no arquivo atual, digite o trecho em "Text to find", defina as opções necessárias e escolha OK. As opções são explicadas a seguir.

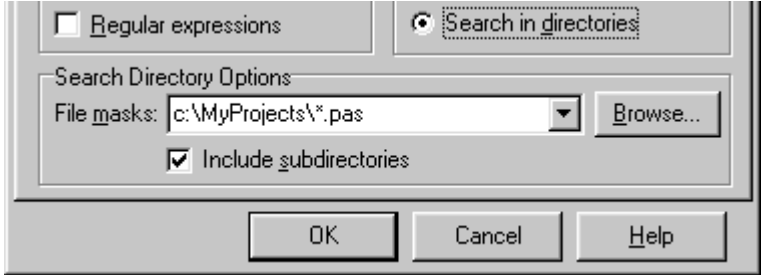
Ative essa opção...	Para obter o seguinte efeito:
Case sensitive	O Delphi diferencia entre maiúsculas e minúsculas na localização.
Whole words only	A palavra especificada só é localizada se ela aparecer "inteira" no texto (com espaços antes e depois). A palavra não pode ser <i>parte</i> de outra. (Há vários outros caracteres especiais – consulte o <i>Help</i>)
Regular expressions	O Delphi permite que sejam usados caracteres especiais para especificar o texto a ser localizado. Um asterisco (*) vale por qualquer número de caracteres; um ponto (.) vale por um único caractere, por exemplo.
Forward	A localização é feita da posição do cursor para a frente.
Backward	A localização é feita da posição do cursor para trás.
Global	A localização é feita em todo o arquivo, na direção especificada.
Selected text	A localização é restrita à área selecionada. Se nada estiver selecionado, ocorre um erro no Delphi!
From cursor	A localização é feita a partir do cursor.
Entire scope	A localização é feita na área selecionada, ou em todo o documento (se a opção <i>Global</i> estiver marcada)

8.7. Localizando textos em vários arquivos

É possível localizar textos em vários arquivos, até em arquivos que não estão abertos no Editor de Código. Para localizar textos em vários arquivos, escolha o comando **Search | Find in files**, ou escolha a página **Find in files** da caixa de diálogo **Find** (veja a seção anterior). A seguinte caixa é exibida:

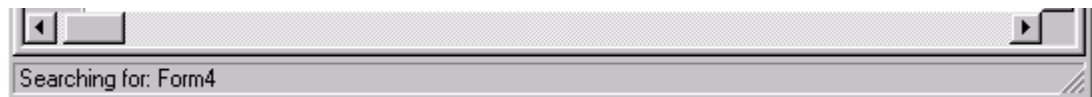


Aqui, da mesma forma que para o comando **Search | Find**, você digita o texto a ser procurado e escolhe as opções necessárias. As opções na área "Options" da caixa são idênticas às descritas anteriormente. Na área "Where", você especifica em quais arquivos o texto será procurado:

Opção	Descrição
Search all files in project	Procura em todos os <i>arquivos do projeto</i> , estejam eles abertos ou não.
Search all open files	Procura em todos os <i>arquivos abertos</i> no Editor de código.
Search in directories	Procura o texto no diretório especificado na parte de baixo da caixa de diálogo. No exemplo ilustrado a seguir, o texto será procurado nos arquivos com extensão .pas e que estejam no diretório c:\MyProjects , ou em subdiretórios deste. 

8.8. Outras técnicas para localização

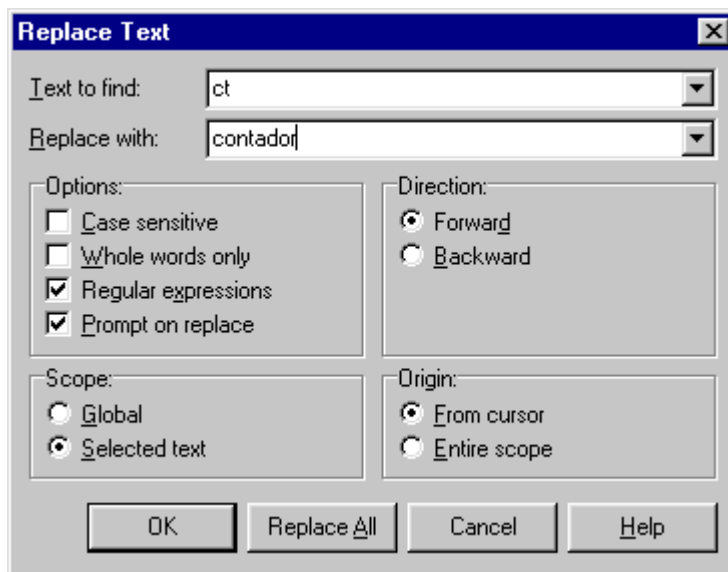
- Depois de localizar um trecho de texto, com os comandos descritos, você **pode localizá-lo novamente** sem usar comandos de menu. Use o comando **Search | Search again**, ou pressione **F3** para localizar as próximas ocorrências do texto.
- Com o comando **Search | Incremental Search**, o Delphi localiza o texto a medida que você digita. Depois de escolher esse comando, o texto digitado aparece na barra de status, na parte de baixo do Editor de Código:



Se o texto que está sendo digitado for encontrado, o texto é *selecionado* no arquivo. Continue digitando até localizar exatamente o texto desejado. Para parar a localização, pressione ENTER, ou qualquer seta do teclado.

8.9. Substituindo textos

Muitas vezes é necessário substituir palavras ou trechos de texto em um arquivo (para corrigir o nome de uma variável que aparece várias vezes, por exemplo). O Delphi permite fazer substituições apenas em um arquivo de cada vez. Para substituir um trecho de texto por outro, use o comando **Search | Replace**. Especifique o texto a ser localizado na primeira caixa ("Text to find") e o texto que será usado para substituí-lo na segunda caixa ("Replace with").

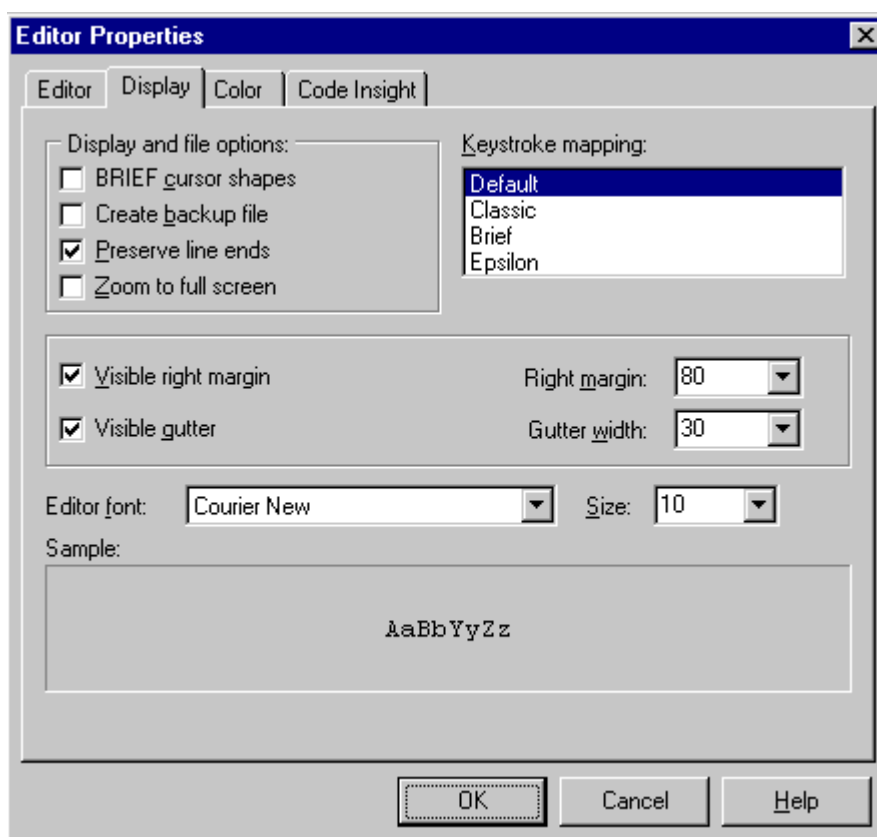


A opção **Prompt on replace** faz com que cada substituição seja confirmada. Desative-a para que o Delphi não mostre confirmações.

Para substituir **todas** as ocorrências do texto especificado, clique no botão **Replace All**.

8.10. Configurando o Editor de Código

O Editor de Código pode ser configurado para se adaptar melhor à sua maneira de programar, ou a costumes antigos que foram adquiridos com a experiência de programação. Há uma quantidade enorme de opções que podem ser alteradas para o Editor de Código. Veremos aqui as mais importantes e úteis.



Configuração do Editor de Código: página Display.

Para configurar o Editor de Código, clique dentro da janela do Editor com o botão direito e escolha **Properties**. A caixa de diálogo **Environment Options**, com apenas quatro páginas de opções, é exibida. A primeira página – **Editor** – é usada para configurações avançadas do editor. São opções pouco usadas. As próximas duas páginas – **Display** e **Colors** – apresentam várias opções úteis que afetam a aparência e os recursos de edição do Editor de Código. A página **Code Insight** é usada para configurar os recursos "Code Insight", de auxílio à programação, que veremos na próxima seção.

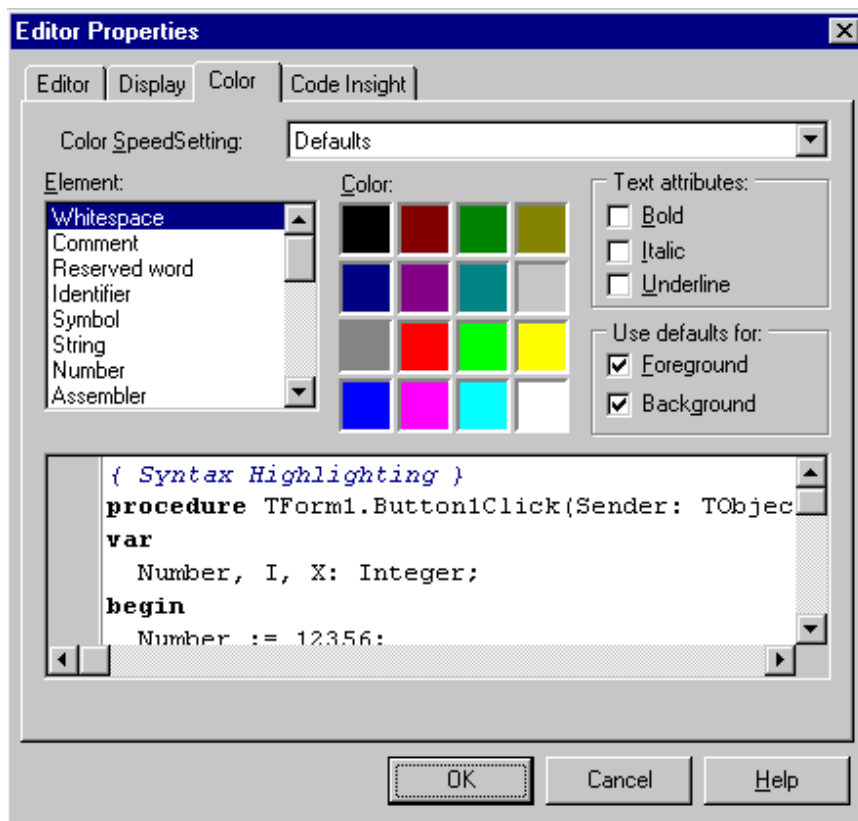
8.11. Opções importantes da página Display

Na página **Display**, você define a aparência e a organização da tela do Editor de Código, alterando as seguintes opções:

Opção	Descrição
Create backup file	Ative essa opção para criar sempre arquivos de backup quando um arquivo é salvo a partir do Editor de Código.
Zoom to full screen	Ative essa opção para que o Editor de Código ocupe toda a área da tela , quando sua janela é maximizada. Se essa opção estiver desativada (o padrão), a janela do Editor de Código não sobrepõe a janela principal do Delphi.
Visible right margin	Exibe ou não a linha da margem do lado direito do Editor de Código.
Right Margin	Determina a distância da margem direita, em caracteres.
Visible Gutter	Exibe ou não a faixa cinza na parte esquerda do Editor de Código.
Gutter Width	Determina a largura da faixa cinza, em caracteres.
Editor Font / Size	A fonte usada no Editor. As fontes que você pode escolher estão restritas a somente fontes monoespaçadas – fontes nas quais todos os caracteres ocupam a mesma largura. Size é o tamanho da fonte, em pontos.

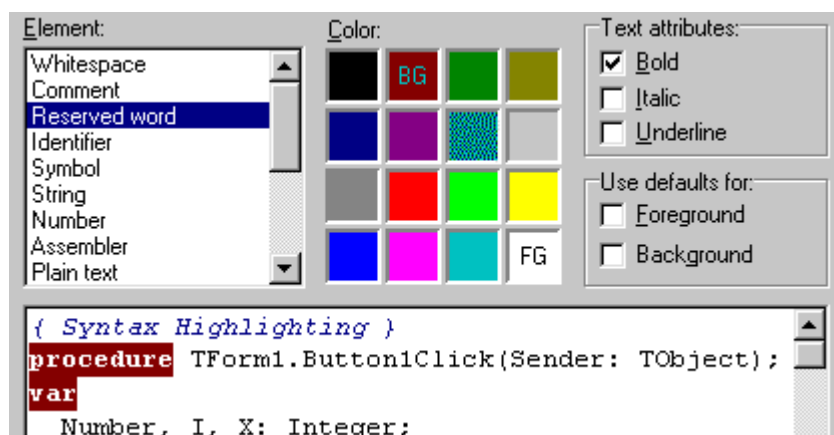
8.12. Opções importantes da página Colors

Na página **Colors**, você define as cores usadas para o efeito de **Syntax highlighting**. É o efeito de Syntax Highlighting que faz as palavras mudarem de cor ou de estilo durante a digitação do código. Este efeito altera as cores de elementos de programas, dependendo da função ou significado desses elementos (exemplos de elementos são comentários, palavras-chave, constantes, variáveis, etc.). Cada elemento pode ser mostrado com duas cores: a cor de frente (*Foreground*) e a cor de fundo (*Background*). Você pode também escolher entre vários padrões de cores predefinidos.



Configuração do Editor de Código: página Colors

- Para alterar as opções de **Syntax Highlighting** para um elemento:
 1. Na lista "Element", escolha o elemento para o qual deseja alterar a cor. Por exemplo, para alterar a cor dos comentários, escolha o elemento "Comments" na lista.
 1. Na paleta de cores, para alterar a cor de frente, clique com na cor com o botão esquerdo (aparecem as letras "FG" dentro do quadrado de cor correspondente); para alterar a cor de fundo, clique com o botão direito (aparecem as letras "BG" dentro do quadrado de cor). Veja um exemplo na seguinte figura:



Para desfazer uma mudança de cor, escolha uma das opções **Foreground** ou **Background**, na área "Use defaults for". Isso faz com que o elemento selecionado volte para a cor normal.

2. Caso necessário, defina atributos (Negrito, Itálico, ou Sublinhado) para o texto do elemento, em "Text attributes".

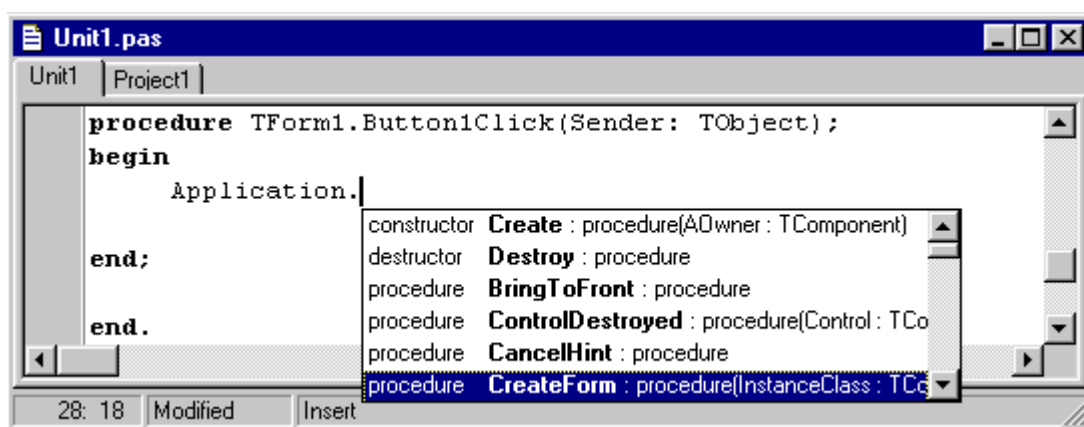
As mudanças que você faz para cada elemento são mostradas em um exemplo de programa, na parte de baixo da caixa de diálogo.

8.13. Usando os recursos *Code Insight*

Os recursos *Code Insight* facilitam muito a preparação de programas no Delphi. Há três muito úteis: **Code Completion**, **Code Parameters** e **Code Templates**.

8.13.1. Code Completion

Esse recurso mostra, durante a digitação do código, uma lista de propriedades, procedures e functions. A lista exibida depende do que está sendo digitado no momento. No exemplo abaixo, a palavra "Application" acaba de ser digitada, seguida de um ponto:

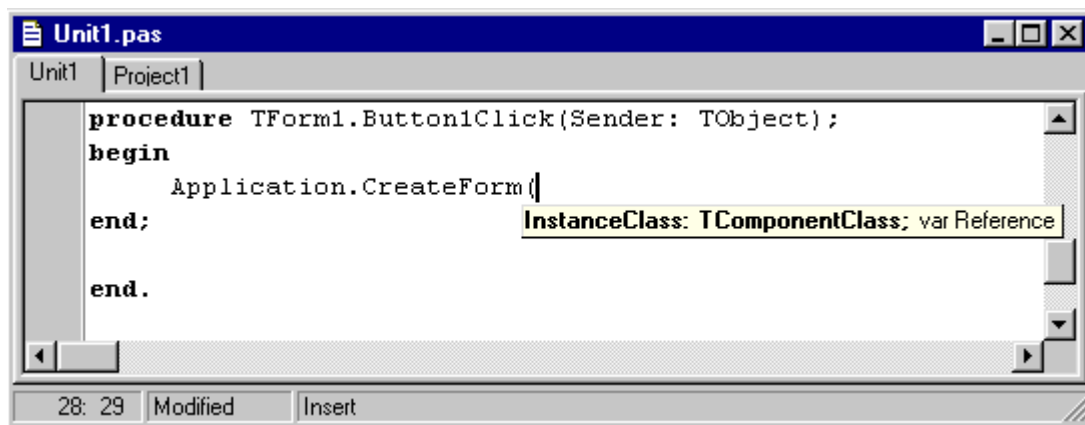


Assim que o ponto é digitado, o Delphi verifica o tipo de objeto digitado e exibe a lista automaticamente (depois de alguns momentos). Pode-se também pressionar **CTRL + Barra de Espaços** para mostrar a lista imediatamente.

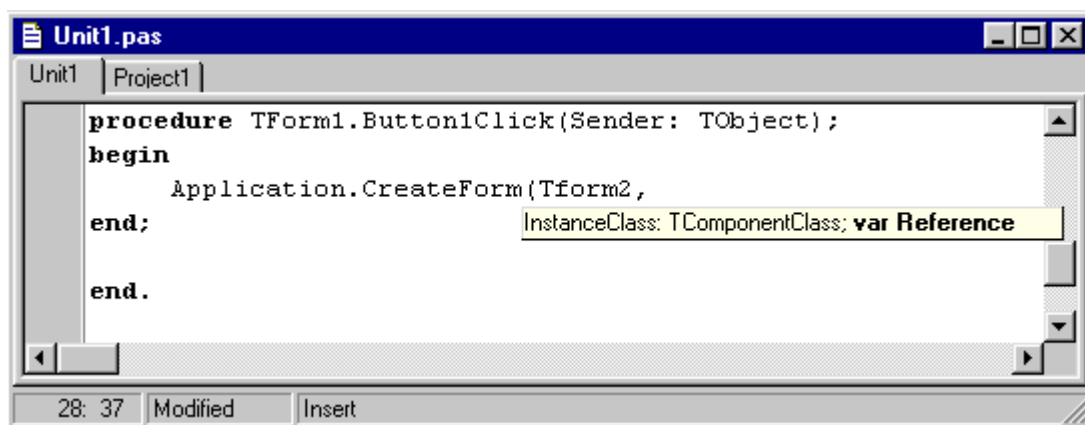
Você pode escolher um item da lista com o teclado ou o mouse, ou continuar digitando normalmente. (Na figura acima, a procedure **CreateForm** foi escolhida). Com isso, o Delphi insere o item escolhido depois do ponto e fecha a lista de opções.

8.13.2. Code Parameters

O recurso de **Code Parameters** ("Parâmetros do Código") mostra a posição e os tipos de parâmetros definidos para uma procedure ou function (que tenha parâmetros!). As informações sobre os parâmetros são exibidos em uma pequena caixa amarela, abaixo da linha que está sendo digitada (veja a figura abaixo).



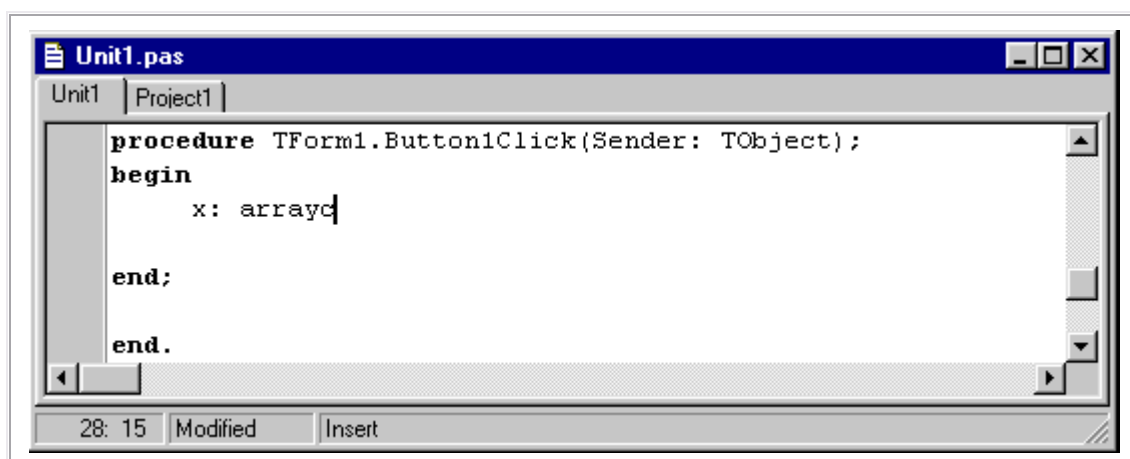
As informações continuam sendo exibidas até que os parênteses da procedure/function sejam fechados. Além disso, enquanto os parâmetros são digitados, o Delphi realça (em negrito) o parâmetro atual. Veja um exemplo na figura a seguir, com o primeiro parâmetro já digitado:



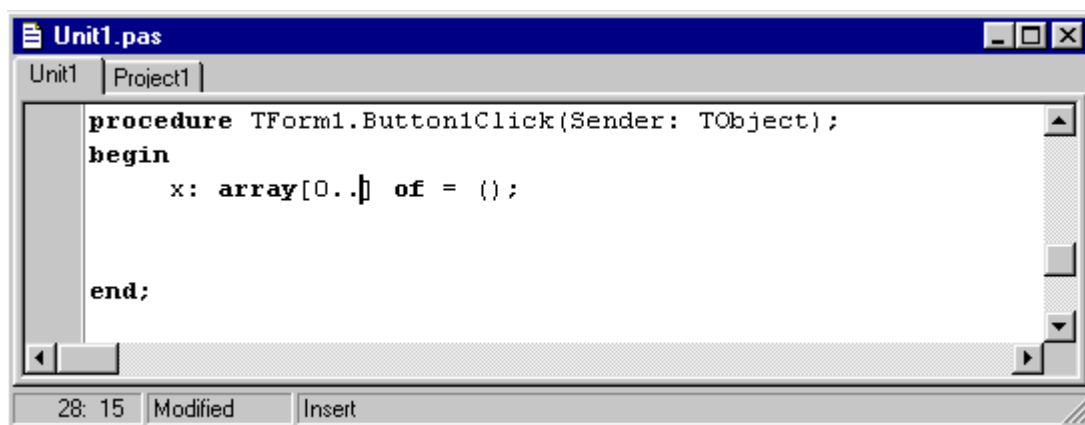
8.13.3. Code Templates

Os **Code Templates** ("Modelos de Código") são "esqueletos" de comandos comuns que podem ser inseridos no código rapidamente, através da digitação de uma palavra ou código.

O Delphi já define vários *Code Templates* para trechos comuns de programas, como estruturas **if-then-else**, declarações de **arrays**, etc. Para inserir um code template, digite o seu nome e pressione CTRL + J. O Delphi insere o template e move o cursor automaticamente para a posição onde se deve começar a digitar (veja as figuras a seguir).



Digite o nome do *template* e pressione CTRL+J...



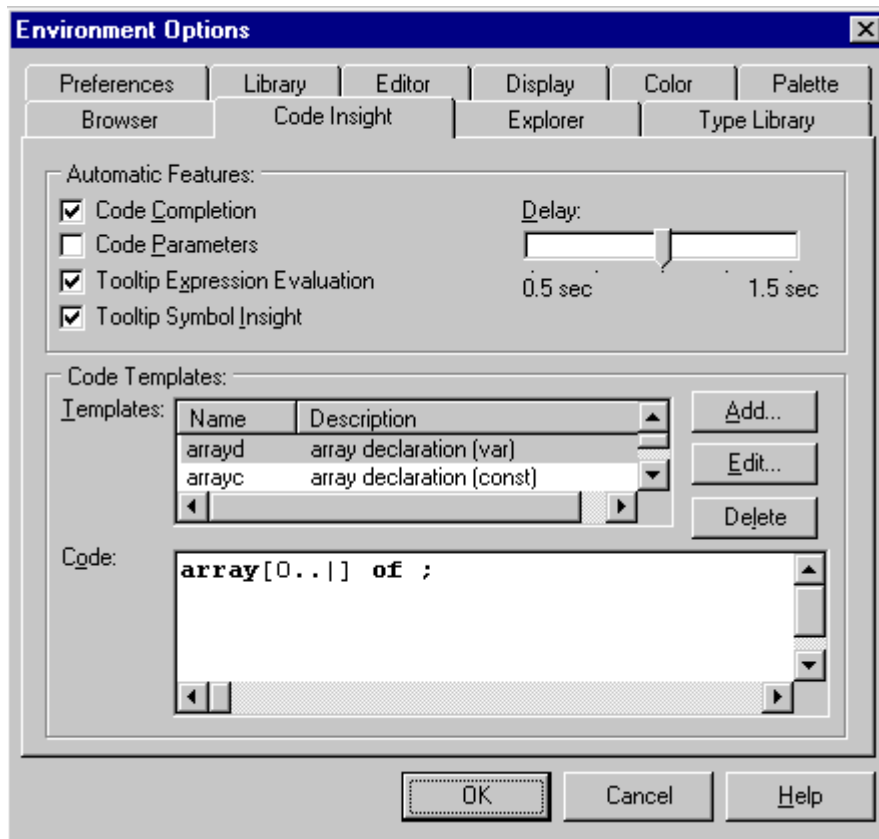
... o código é inserido e o cursor é colocado na posição apropriada.

Veja a seguir alguns exemplos de *code templates* predefinidos do Delphi (as barras verticais no código indicam a posição onde é colocado o cursor, quando o *code template* é inserido).

Este nome...	É substituído por...	Este nome...	É substituído por...
cases	case of :: :: end;	whileb	while do begin end;
forb	for := to do begin end;	fors	for := to do
function	function (): ; begin end;	ife	if then begin end else
procedure	procedure (); begin end;	trye	try except end;

8.14. Configurando os recursos do *Code Insight*

É possível desativar ou ativar cada um dos recursos do *Code Insight*, ou alterar opções gerais para esses recursos. Para isso, escolha o comando **Tools | Environment Options** e mude para página **Code Insight**, ilustrada a seguir:

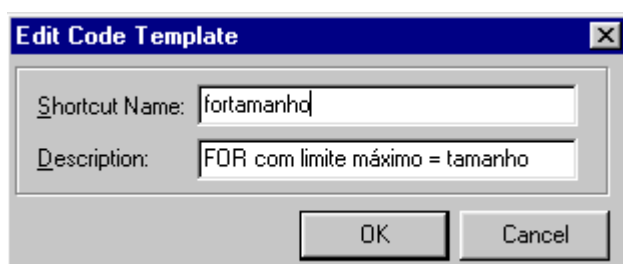


Pode-se ativar ou desativar cada recurso *Code Insight* na parte de cima da caixa, além de alterar o tempo de espera para que os recursos sejam ativados durante a digitação. Para alterar o tempo de espera, mova a pequena barra abaixo de "Delay".

Pode-se também definir novos *Code Templates* ou alterar os já existentes. Os procedimentos abaixo devem ser realizados a partir da página **Code Insight** ilustrada anteriormente.

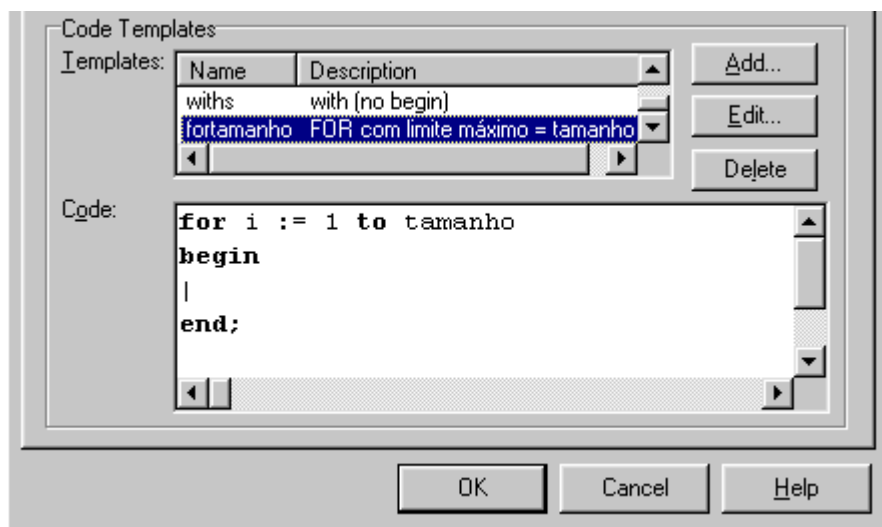
- **Para definir um novo *Code Template*:**

1. Clique no botão **Add**. Na caixa que aparece, digite um nome para o *code template* e uma descrição para ele:



1. Clique em **OK** para fechar a caixa de diálogo e digite o código para o *code template* na área de baixo (ao lado de "Code"). Digite uma barra vertical (|) para indicar a posição

onde o cursor deve ser colocado quando o código é inserido. (Veja um exemplo na figura a seguir).



2. Clique em **OK** para salvar o *code template*. Agora você pode usá-lo da mesma forma que os *code templates* predefinidos.

- **Para alterar um *code template*:**
- Selecione o nome do *code template* na lista de *code templates* e altere o código na parte de baixo da caixa (ao lado de "Code").
- **Para apagar um *template*:**
- Selecione o nome do *code template* e clique no botão **Delete**.

CAPÍTULO 09 - O DEPURADOR INTEGRADO

O Depurador Integrado (*Integrated Debugger*) do Delphi é uma ferramenta que facilita a verificação e a correção dos erros em seus programas. Com o depurador, você pode, por exemplo, executar programas passo-a-passo e verificar o valor de variáveis e expressões durante a execução.

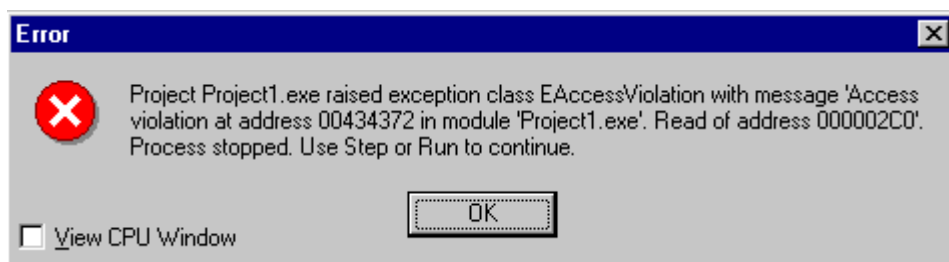
9.1. Erros e tipos de erros

Há dois tipos de erros básicos que podem ocorrer (e quase sempre ocorrem) durante o desenvolvimento de um aplicativo: erros de compilação e erros de execução.

Os **erros de compilação** aparecem durante a *compilação* do aplicativo no Delphi. Os erros de compilação são geralmente causados por erros de sintaxe, ou erros de tipos. Esses erros, geralmente são os mais fáceis (ou menos difíceis) de corrigir. Muitas vezes esses erros são causados por simples erros de digitação. Um erro de compilação é fácil de corrigir porque o compilador mostra exatamente a linha em que o erro ocorreu e às vezes até sugere uma maneira de corrigi-lo.

Os **erros de execução** ocorrem durante a *execução* do aplicativo. Estes são bem mais difíceis de corrigir. Como os aplicativos do Delphi são compilados, o código que é executado é convertido antes em **código binário**. O código binário puro não guarda ligação direta com as linhas do código fonte (o código nas Units) – o código binário só reconhece **posições de memória**.

Por isso, quando ocorre um erro em tempo de execução, não é mostrada a linha onde ocorreu o erro, mas sim a posição de memória (um número em hexadecimal!!) Veja um exemplo de uma mensagem de erro de execução:



A mensagem ilustrada foi causada por um erro comum: tentar mostrar um formulário que ainda não foi criado em memória. Mas outras mensagens de erros de execução têm geralmente o mesmo formato. A mensagem começa com o *nome do arquivo executável* onde ocorreu o erro, o *nome da exceção* que foi levantada, e as *posições de memória* onde os erros aconteceram.

A não ser para os programadores do tipo "escovador de bits", esse tipo de mensagem não ajuda muito a identificar o erro. Mesmo que se reconheça o tipo de erro que aconteceu (a partir do nome da exceção), não há como saber diretamente *onde* o erro ocorreu no programa.

É nesse ponto que entra o Depurador Integrado do Delphi.

9.2. Utilidades do Depurador

Quando você executa um aplicativo no Delphi, ele é executado automaticamente *dentro do depurador*. Quando ocorre um erro de execução no aplicativo, o depurador interrompe o

programa e assume o controle. O depurador consegue "assumir o controle" mesmo se houver um erro grave (como um erro de *hardware*). Nesse caso extremo, o depurador não pode ajudar muito, mas ele pelo menos evita que os erros travem o computador ou o coloquem em um estado instável.

O depurador é mais útil, entretanto, quando os erros são causados pelo próprio código do aplicativo e não por razões externas. Quando há erros de lógica em seus programas, por exemplo, o Depurador (além da sua experiência) é a sua única ajuda para detectá-los.

NOTA: Erros de lógica são problemas na estrutura ou lógica de um programa que fazem com que o programa não execute como esperado. Um erro de lógica comum é, por exemplo, usar uma condição incorreta em um loop **while** o **repeat**: o programa pode ficar preso dentro do loop para sempre. Erros de lógica, claramente, não podem ser indicados pelo compilador.

9.3. Executando um programa até a posição do cursor

No Editor de Código, você pode posicionar o cursor em qualquer linha de um programa e executar o programa até aquela linha. Esse recurso é especialmente útil para programas extensos. Você pode executar rapidamente uma parte do programa para depois se concentrar na parte "suspeita", que contém os erros.

- **Para executar um programa até a posição do cursor:**

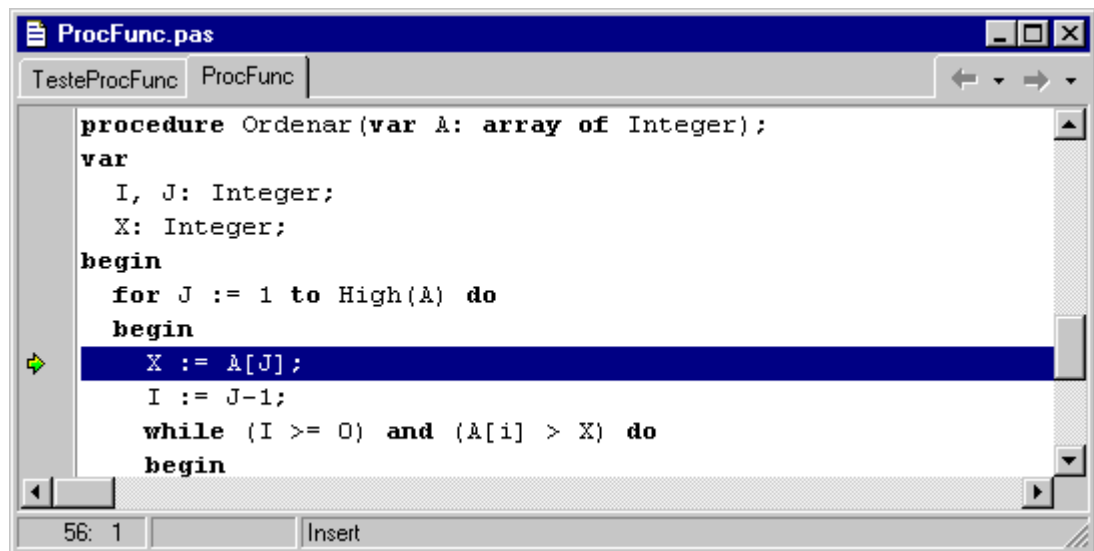
1. No Editor de Código, posicione o cursor na linha até onde o programa deve ser executado.

1. Realize uma das ações a seguir:

- Escolha o comando **Run | Run to Cursor**
- Clique com o botão direito no Editor de Código e escolha o comando **Debug** e depois **Run to Cursor**.
- Pressione **F4**

O Delphi executa normalmente todas as linhas do programa até a linha anterior à linha do cursor. Depois o depurador interrompe a execução e assume o controle, realçando a próxima linha a ser executada (a linha do cursor).

A próxima linha a ser executada é chamada **ponto de execução**. O ponto de execução é realçado em azul, no Editor de Código. É exibida também uma pequena seta do lado esquerdo da linha. A ilustração abaixo mostra o ponto de execução logo depois de um programa ser executado com o comando **Run to cursor**.



NOTA: você pode mostrar o ponto de execução no Editor de Código, se ele não estiver aparecendo por alguma razão, usando o comando **Run | Show execution point**.

9.4. Executando um programa linha por linha

Um programa pode ser executado linha por linha no Delphi, para que se possa, por exemplo, entender a execução em detalhe, ou verificar precisamente onde os erros estão acontecendo.

O depurador do Delphi oferece dois comandos para executar um programa linha por linha: **Trace Into** e **Step Over**.

9.5. O comando Trace Into

O comando **Trace Into** executa uma linha de um programa de cada vez. Se a linha a ser executada for uma chamada a uma rotina (function ou procedure), o código da rotina é executado também, linha por linha. Em outras palavras, o comando **Trace Into**, "entra" nas procedures e functions.

- Para usar o comando **Trace Into**, realize uma das seguintes operações:
- Escolha o comando **Run | Trace Into**
- Pressione **F7**
- Clique no botão **Trace Into** na barra de ferramentas (figura a seguir).



O botão Trace Into

9.6. O comando Step Over

O comando **Step Over** também executa os comandos de um programa, uma linha de cada vez. Mas, ao contrário do comando **Trace Into**, ele não "entra" no código das procedures e

functions. Estas são executadas diretamente, em um único passo, e o ponto de execução é movido para a linha depois da chamada à função.

Você pode usar o comando **Step Over** para "pular" o código de uma função que não precisa ter seu código testado linha por linha. Isso pode tornar a depuração muito mais rápida.

- Para usar o comando **Trace Into**, realize uma das seguintes operações:
- Escolha o comando **Run | Step Over**
- Pressione **F8**
- Clique no botão **Step over** na barra de ferramentas (veja a figura).



9.7. Usando os dois comandos

Para programas complexos, você geralmente vai precisar usar os dois comandos, **Trace Into** e **Step Over**, alternadamente. Se, durante a depuração, você achar necessário "entrar" no código de uma procedure ou function, use o comando **Trace Into**. Para pular o código da procedure ou function, executando-a rapidamente use o comando **Step Over**. Você pode usar um comando ou o outro na ordem que desejar.

9.8. Interrompendo a execução

Muitas vezes durante a depuração, é necessário interromper a execução do aplicativo. Isso pode ser necessário, por exemplo, quando se passa do erro que estava sendo procurado, ou quando o programa fica em uma situação instável, por conta de uma alguma exceção que ocorreu.

- Para interromper o programa que está sendo executado:
- Escolha o comando **Run | Program reset**, ou
- Pressione **CTRL+F2**

Na verdade, o comando **Program reset** faz mais que interromper o programa. Ele é extremamente poderoso e pode ser usado para lhe tirar das situações mais difíceis. Se o programa travar, devido a um erro de hardware, por exemplo, o comando **Program reset** consegue "matar" o programa e fazer tudo voltar ao normal. São muito raros os casos em que um erro em um programa faz o próprio Delphi travar irreversivelmente.

O comando **Program reset** fecha todos os arquivos abertos pelo seu programa, libera todos os recursos que estejam sendo usados e limpa o valor de todas as variáveis.

9.9. Usando Breakpoints

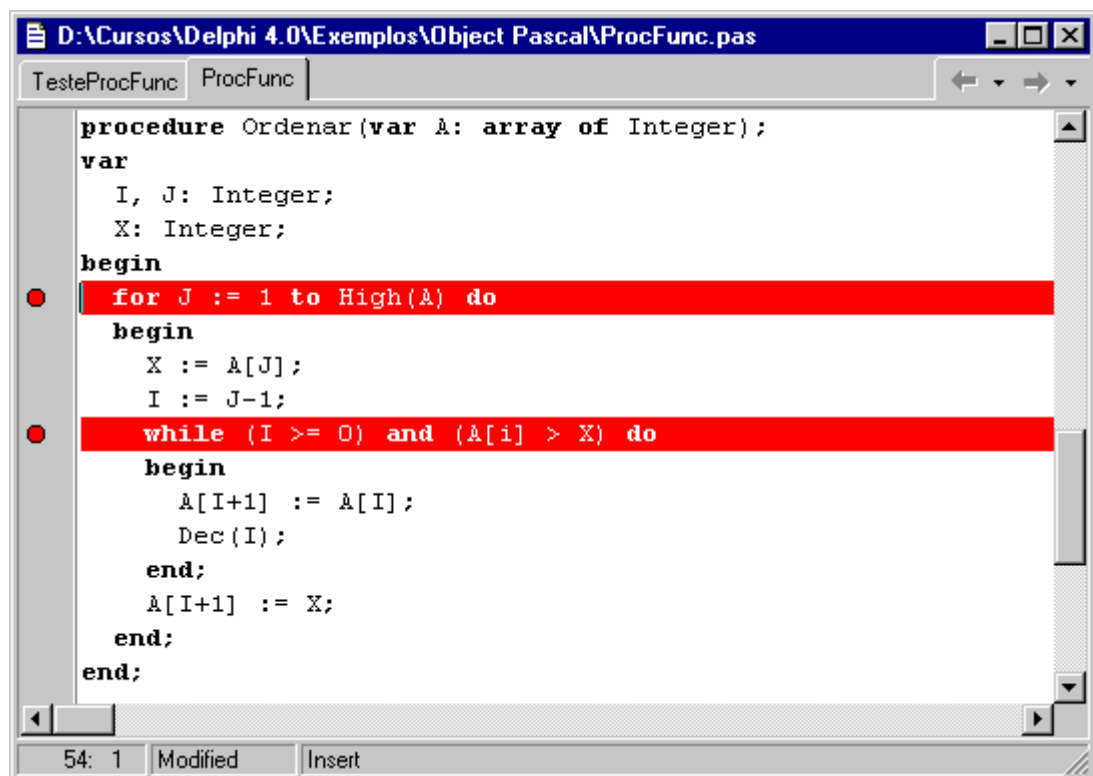
Você usa **breakpoints** ("pontos de parada") para que o depurador pare em pontos específicos de um programa. Vários breakpoints podem ser definidos no mesmo programa. Com isso, você pode fazer a execução parar em partes "delicadas" ou complexas de um programa e depois executar essas partes linha por linha.

9.10. Definindo breakpoints

Os *breakpoints* ("pontos de parada") permitem forçar a parada da execução em linhas específicas de um programa. Pode-se definir breakpoints para várias linhas de um mesmo programa, desde que estas sejam linhas executáveis (não se pode definir breakpoints para linhas de comentários, por exemplo). Veja como definir um breakpoint:

- Para definir um breakpoint em uma linha de código:
- Clique na faixa cinza do Editor de Código, ao lado da linha, ou
- Mova o cursor para a linha de código e pressione **F5**.

A linha é realçada em vermelho e aparece um pequeno círculo do lado esquerdo da linha (veja a figura).



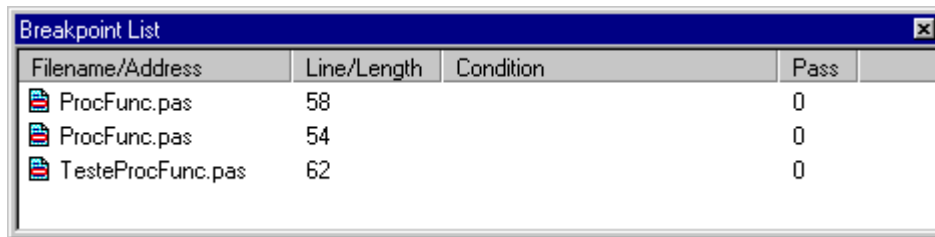
Breakpoints definidos no Editor de Código

Quando encontra um breakpoint, o Delphi pára imediatamente o programa e mostra a linha onde parou – a linha do breakpoint – no Editor de Código. Pode-se então continuar executando o programa passo-a-passo, com **F7** ou **F8**, ou continuar a execução até o final do programa (ou até o próximo breakpoint), usando **F9**.

9.11. Mostrando os breakpoints definidos

Durante a depuração de um programa extenso, o número de breakpoints definidos pode crescer muito. Para facilitar o trabalho com breakpoints, você pode mostrar uma janela com a lista de todos os breakpoints definidos.

- Para mostrar todos os breakpoints definidos em um aplicativo:
- Escolha o comando **View | Debug Windows > Breakpoints**.

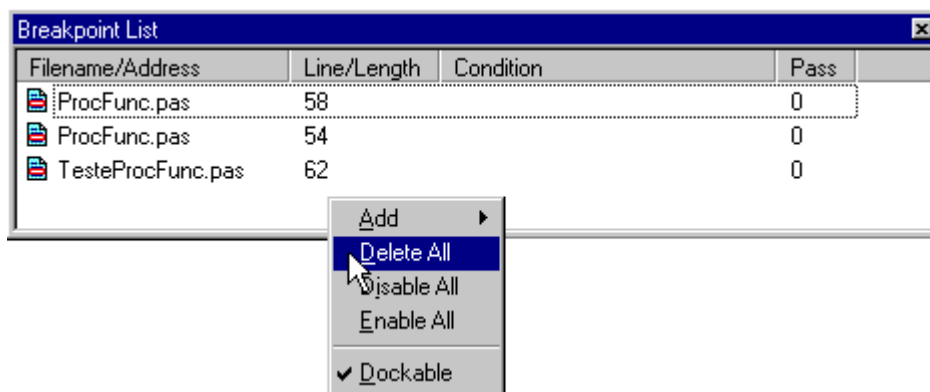


A janela exibe o nome do arquivo que contém cada breakpoint, o número da linha do breakpoint e outras informações mais avançadas. A partir desta janela, você pode pular diretamente para qualquer breakpoint definido, clicando duas vezes no nome do breakpoint.

Você pode também **desativar** um breakpoint temporariamente, ou **apagá-lo** de forma permanente. Você pode também apagar todos os breakpoints de uma vez. Note, no entanto, que um breakpoint desativado pode ser reativado, mas que não é possível voltar atrás depois de apagar um breakpoint.

- **Para desativar/reactivar um breakpoint:**
- No Editor de Código, clique com o botão direito do lado esquerdo do breakpoint e escolha o comando **Enabled**.
- **Para apagar um breakpoint:**
- No Editor de Código, clique no breakpoint e pressione **F5**, ou
- Clique do lado esquerdo do breakpoint (na faixa cinza do Editor de Código).
- **Para apagar/desativar/ativar todos os breakpoints:**

Abra a lista de breakpoints (**View | Debug Windows > Breakpoints**) e clique com o botão direito dentro da janela exibida, mas fora da lista de breakpoints (veja a figura a seguir).



Escolha o comando **Delete All** para **apagar** todos os breakpoints, ou **Disable All** para **desativar** todos os breakpoints, ou **Enable All** para **ativar** todos os breakpoints.

9.12. Verificando variáveis e expressões

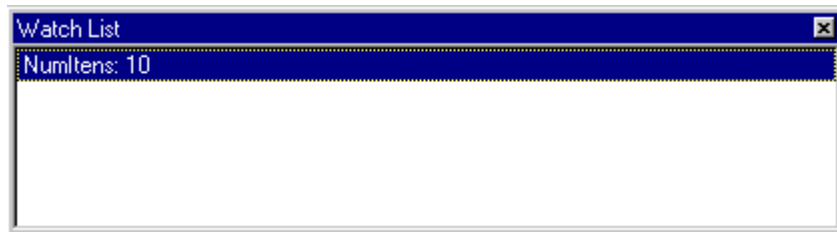
A execução linha por linha de um programa pode não ser suficiente para descobrir a causa de um erro. Às vezes é necessário verificar o *valor* de variáveis ou expressões durante a execução do programa. O Delphi oferece vários recursos para realizar essas tarefas.

9.13. Trabalhando com Watches

Watches ("sentinelas") são usados para monitorar os valores de uma variável ou expressão, durante a execução de um aplicativo no Delphi. Os *watches* definidos são atualizados automaticamente durante a execução.

- **Para definir um watch:**

1. No Editor de Código, selecione a variável ou expressão cujo valor deseja monitorar (este será o *watch*).
1. Clique com o botão direito em cima do trecho selecionado e escolha o comando **Debug > Add Watch at Cursor** no menu de atalho que aparece. O Delphi cria o *watch* e mostra a lista de *watches* definidos:



- **Para visualizar a lista de watches definidos:**
- Escolha o comando **View | Watch**.

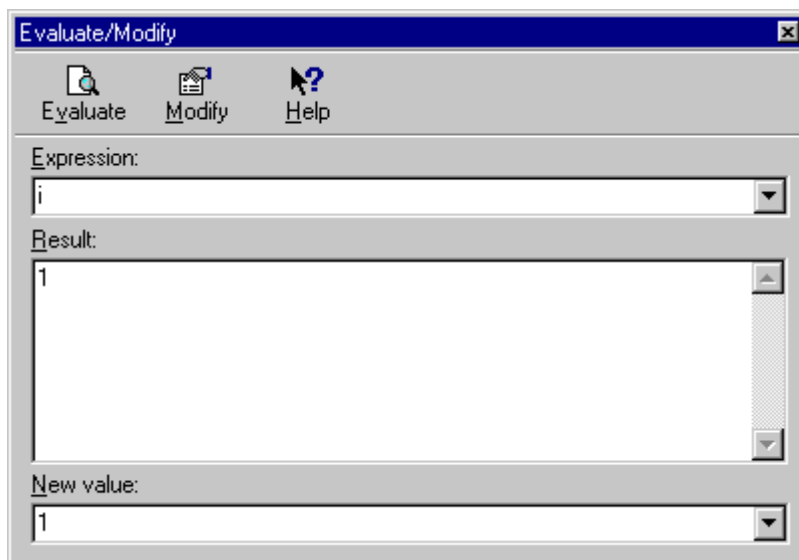
O Delphi exibe a janela **Watch List** (ilustração anterior) com os nomes e os valores de cada *watch*.

9.14. Avaliando e modificando expressões

A caixa de diálogo **Evaluate/Modify** permite avaliar expressões e alterar variáveis, durante a depuração.

- **Para avaliar e modificar expressões:**

1. Escolha o comando **Run | Evaluate/ Modify**
2. O Delphi mostra a seguinte caixa de diálogo:



1. Para **avaliar** uma expressão, digite a expressão a ser avaliada em "Expression". A expressão deve ser formada por constantes, ou por variáveis acessíveis no momento. Em seguida clique no botão **OK**. O valor da expressão é mostrado em "Result".

2. Para **modificar** o valor de uma variável durante a execução, especifique o nome da variável em "Expression", especifique um valor em "New value", e clique no botão **Modify**. Não é possível modificar uma expressão com mais de uma variável.

CAPÍTULO 10 - TRABALHANDO COM BANCOS DE DADOS NO DELPHI: UMA VISÃO GERAL

O Delphi oferece recursos poderosos para a criação de aplicativos com acesso a bancos de dados. Aplicativos criados no Delphi podem ter acesso a dezenas de tipos de bancos de dados, locais ou remotos.

Para os bancos de dados mais populares, como Oracle, Sybase, DB2, Access, etc., o Delphi oferece acesso **nativo**. Toda a comunicação entre o Delphi e esses SGBDs é feita internamente no ambiente do Delphi. O acesso nativo é geralmente muito mais rápido.

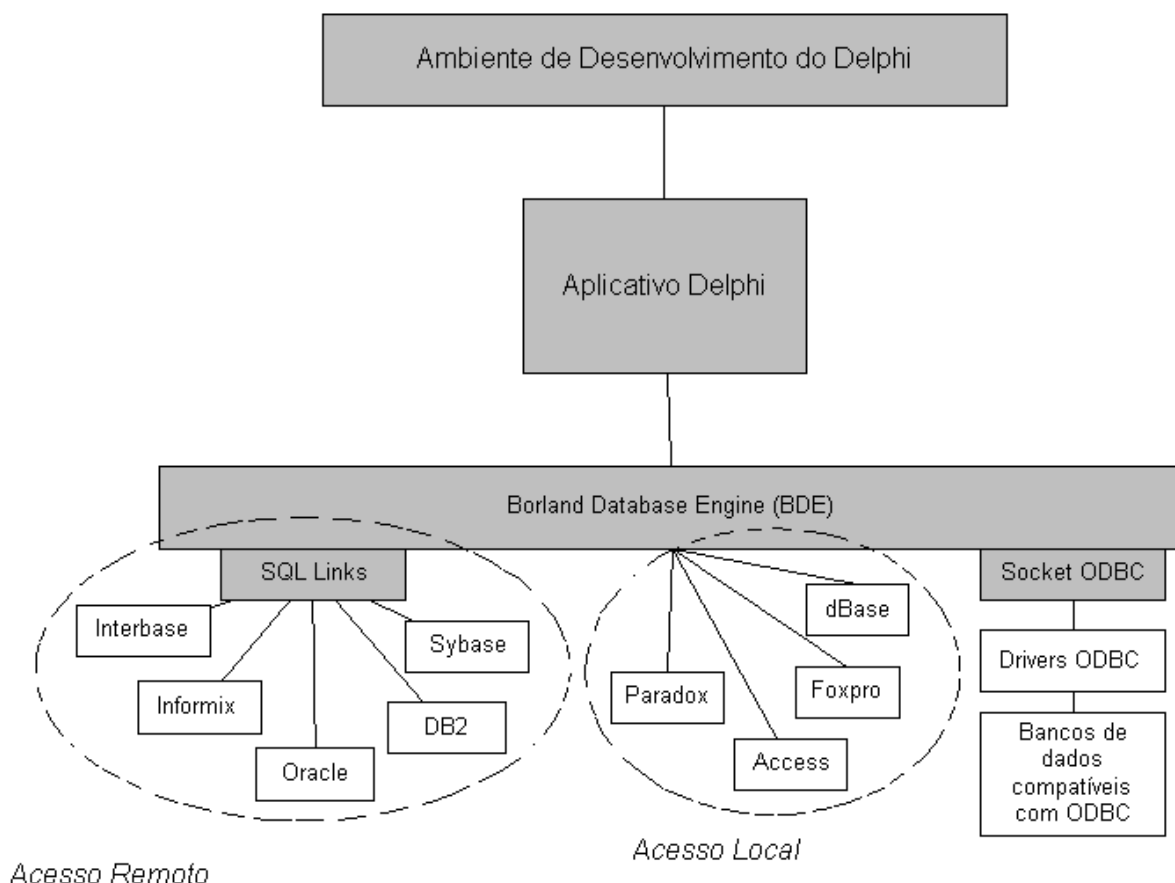
Para bancos de dados menos populares, o Delphi oferece acesso via *ODBC (Open Database Connectivity)*. Praticamente todos os bancos de dados profissionais são compatíveis com a tecnologia ODBC. Um banco de dados compatível com ODBC oferece *drivers ODBC* que podem ser instalados no Windows. Com o driver ODBC correspondente instalado, um banco de dados pode ser acessado facilmente a partir do Delphi (o acesso é bem mais lento que o acesso nativo, no entanto).

10.1. A arquitetura de acesso a bancos de dados

Todo o acesso a bancos de dados a partir de um aplicativo Delphi, sejam esses bancos remotos ou não, é feito através do *BDE (Borland Database Engine)*. O BDE é uma interface padrão que gerencia a parte técnica por trás do acesso e manipulação de bancos de dados.

Além disso, o BDE *uniformiza* a comunicação entre o aplicativos e os bancos de dados – pode-se usar os mesmos componentes e comandos para acessar um banco de dados Oracle, ou um banco Paradox, por exemplo.

Os bancos locais, como Paradox e dBase são acessados diretamente pelo BDE. Já os bancos remotos SQL, como Oracle e Sybase, precisam também do driver **SQL Links**, instalado automaticamente com o Delphi *Client/Server*. É esse driver que garante a uniformização do acesso a bancos de dados SQL, gerando consultas SQL automaticamente, no dialeto de SQL usado pelo banco. Com o driver **SQL Links**, é possível trabalhar com bancos SQL da mesma forma que com bancos locais.



A arquitetura de acesso a bancos de dados no Delphi

10.2. Componentes básicos

Há dezenas de componentes no Delphi para o trabalho com bancos de dados. São duas páginas da paletas de componentes exclusivamente dedicadas a bancos de dados: as páginas **Data Access** e **Data Controls**.

10.3. A Página Data Access





A página **Data Access** contém os componentes usados para realizar a conexão aos bancos de dados e acessar suas informações. Os componentes mais importantes, e de longe os mais usados, são os três primeiros: **DataSource**, **Table** e **Query**.



A página Data Access

Há também vários outros componentes avançados usados, por exemplo, para a criação de aplicativos cliente/servidor. Esses componentes são vistos no curso de Delphi avançado.

Veja a seguir uma descrição breve dos componentes mais importantes da página **Data Access** (para os fins desse curso básico).

Componente	Descrição
 <i>DataSource</i>	Funciona como um intermediário entre os componentes <i>Table</i> e <i>Query</i> e os componentes da paleta <i>Data Controls</i> .
 <i>Table</i>	Usado para realizar o acesso a tabelas de um banco de dados.
 <i>Query</i>	Usado para realizar o acesso a tabelas geradas por consultas SQL (<i>queries</i>).
 <i>BatchMove</i>	Usado para mover grandes quantidades de dados de uma tabela para outra, para copiar tabelas inteiras, ou para converter tabelas de um tipo para outro.







10.4. A página Data Controls







A página **Data Controls** contém componentes que podem ser ligados diretamente a campos e tabelas de um banco de dados. Muitos desses componentes são apenas versões mais poderosas dos componentes na página *Standard*.



A página Data Controls

Os componentes *DBEdit* e *DBMemo*, por exemplo são equivalentes aos componentes *Edit* e *Memo*, mas com o recurso adicional de acesso a bancos de dados. Veja a seguir uma descrição breve dos componentes mais importantes da página *Data Controls*.

Componente	Descrição
 <i>DBGrid</i>	Exibe dados em formato de tabela. Este componente é usado geralmente para exibir os dados de um componente <i>Table</i> ou <i>Query</i> .
 <i>DBNavigator</i>	Usado para navegar os registros de uma tabela, permitindo que dados sejam apagados, inseridos ou alterados.
 <i>DBText</i>	Exibe o texto de um campo de forma não-editável. Semelhante a um componente <i>Label</i> .
 <i>DBEdit</i>	Exibe o texto de um campo e permite editá-lo. Semelhante a um componente <i>Edit</i> .
 <i>DBMemo</i>	Exibe o texto de um campo com várias linhas de texto (do tipo "BLOB" ou "Memo"). Semelhante a um componente <i>Memo</i> .
 <i>DBImage</i>	Exibe imagens armazenadas em um campo do tipo BLOB, ou "Image".

 DBListBox	Exibe uma lista de itens a partir da qual, pode ser escolhido um valor para um campo. Semelhante ao componente <i>ListBox</i> .
 DBComboBox	Semelhante ao componente <i>DBListBox</i> , mas permite a digitação direta de um valor, além da escolha de um item listado. Semelhante ao componente <i>ComboBox</i> .
 DBCheckBox	Exibe um <i>CheckBox</i> que pode ser usado para exibir ou alterar o valor de um campo booleano.
 DBRadioGroup	Exibe um conjunto de valores mutuamente exclusivos para um campo.
 DBLookupListBox	Exibe uma lista de itens extraída de outra tabela relacionada. Somente elementos da lista podem ser escolhidos. Tem a mesma aparência e o mesmo funcionamento básico do componente <i>ListBox</i> .
 DBLookupComboBox	Exibe uma lista de itens extraída de outra tabela relacionada. Pode-se digitar um valor diretamente. Tem a mesma aparência e o mesmo funcionamento básico do componente <i>ComboBox</i> .

10.5. Acessando bancos de dados: uma introdução

O acesso a bancos de dados a partir de um aplicativo no Delphi é baseado em três componentes básicos: **DataSource**, **Table** e **Query**. Os três são componentes *não-visuais*. Eles não são exibidos durante a execução. Apenas controlam a conexão com o banco de dados e o processamento dos dados.

A conexão com um banco de dados é geralmente feita através de um componente **Table** ou **Query**. Esses componentes são tipos de **DataSets**. Os **DataSets** são conjuntos de dados armazenados em formato de tabela (em linhas e colunas). Veremos mais sobre **DataSets** no próximo capítulo.

Ambos os componentes **Table** e **Query** têm as propriedades *DatabaseName*, que indica o banco de dados associado ao componente. Um componente **Table** é ligado a uma tabela de um banco de dados através de sua propriedade *TableName*. Um componente **Query** contém o texto de uma *query* em SQL. Essa *query* pode ser executada, gerando uma nova tabela de dados (temporária) como resultado.

O componente **DataSource** é usado como um intermediário obrigatório entre os componentes *Data Controls* em um formulário, e os dados de um componente **Table** ou **Query**. A propriedade *DataSet* de um componente **DataSource** é usada para realizar a ligação entre os componentes.

A maioria dos componentes da página *Data Controls* apresenta a propriedade *DataSource*, onde é especificado o componente **DataSource** a ser conectado ao componente. Os dados exibidos nos componentes *Data Controls* são lidos a partir do componente **DataSource** especificado.

10.6. Um exemplo típico

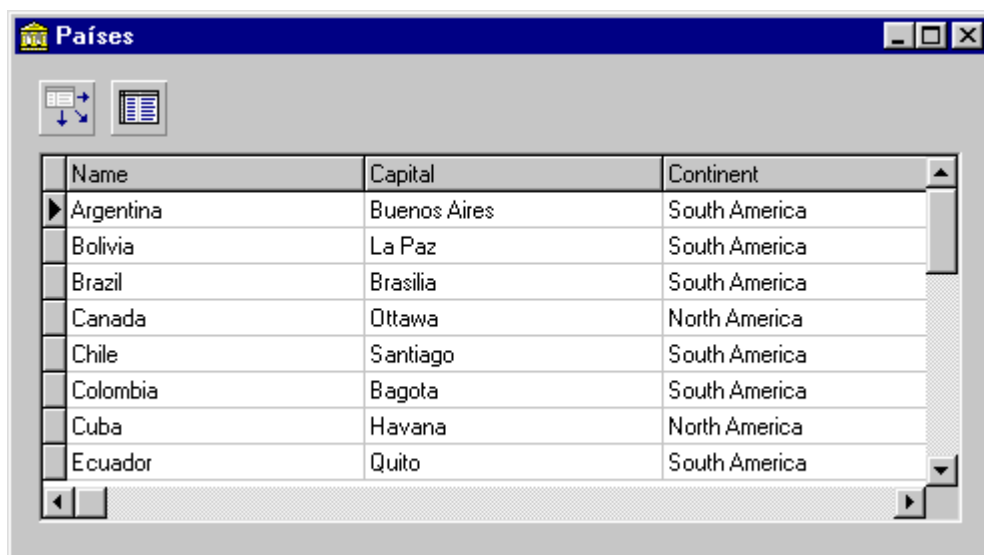
Veremos agora como criar um formulário básico com acesso a bancos de dados. No exemplo, usaremos um componente **Table** para realizar a conexão com uma tabela de um banco de

dados. Os dados serão exibidos em um componente **DBGrid**. (O componente *DBGrid* mostra os dados em um *DataSet* em formato de tabela).

- **Para criar um formulário simples com acesso a dados:**

1. Em um novo formulário, adicione um componente **Table** e um componente **DataSource**, ambos localizados na página *Data Access* da paleta de componentes do Delphi.
2. Adicione também um componente **DBGrid**, da página *Data Controls*.
3. Altere a propriedade *DatabaseName* do componente **Table** para o nome do banco de dados a ser acessado. Altere também a propriedade *TableName* para a tabela do banco de dados cujos dados você deseja exibir. (No nosso exemplo, escolhemos o banco de dados **DBDEMOS** e a tabela **Country**, que contém informações sobre os países do continente americano.)
4. Altere a propriedade *DataSet* do componente *DataSource* para o nome do componente *Table* ("Table1" é o nome padrão).
5. Altere a propriedade *DataSource* do componente **DBGrid** para o nome do componente **DataSource** ("DataSource1" é o nome padrão).
6. Para visualizar os dados imediatamente no componente **DBGrid**, altere a propriedade *Active* do componente **Table** para *True*.

Veja a aparência final do formulário, no exemplo:



10.7. Usando o DataBase Desktop

O *DataBase Desktop* é um aplicativo independente que é instalado junto com o Delphi. Ele pode ser acessado diretamente, através do menu **Iniciar**, ou mesmo de dentro do Delphi.

Com o *Database Desktop* você pode criar tabelas de bancos de dados do tipo Paradox/dBASE. Essas tabelas são muito úteis para a criação de aplicativos com bancos de dados pequenos ou médios, usados por um ou poucos computadores. Outra função importante do *DataBase Desktop* é a definição de "Aliases".

- **Para entrar no Database Desktop, realize uma das seguintes operações:**
- De fora do Delphi, clique no botão **Iniciar** e escolha **Programas > Borland Delphi 4 > DataBase Desktop**.
- De dentro do Delphi, escolha o comando **Tools | DataBase Desktop**.

10.8. Definindo um *Alias*

Antes de definir as tabelas de um banco de dados, você deve definir o *local* onde os arquivos do banco de dados serão armazenados. Esse local é identificado por um nome, chamado **Alias**. Os *Aliases* são usados dentro do Delphi como um tipo de "apelido" para um banco de dados. Definir um Alias oferece várias vantagens:

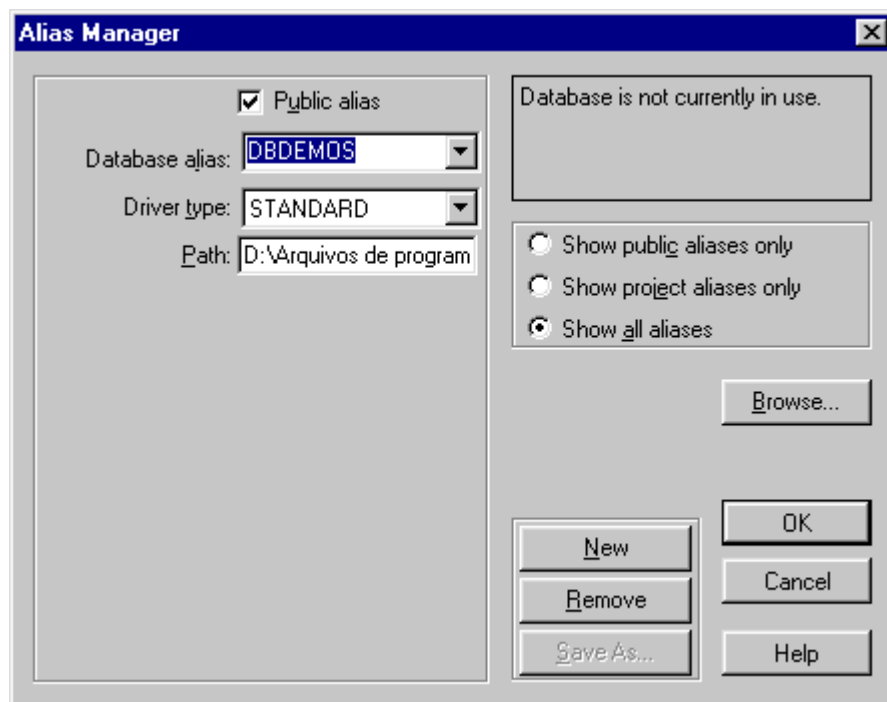
- Não é necessário digitar o caminho completo (*drive* e diretório) do banco de dados. Basta digitar o nome do Alias.
- Pode-se alterar o diretório associado a uma Alias sem a necessidade de alterar o código do aplicativo. Com a mudança, todos os componentes que se referem ao Alias passam a apontar para o novo diretório.

Por exemplo, você pode usar um diretório temporário (digamos, C:\DADOS) para armazenar suas tabelas durante o desenvolvimento e usar outro diretório quando o aplicativo é instalado no computador do cliente (digamos, D:\SISTEMA\DADOS). Se um Alias tiver sido definido na hora do desenvolvimento, basta alterar o diretório do Alias na hora da instalação. O código do aplicativo não precisa ser alterado (nem recompilado).

- **Para criar um *Alias*:**

1. Escolha o comando **Tools | Alias Manager**.

O *DataBase Desktop* exibe a seguinte caixa:



1. Clique no botão **New** e digite um nome para o Alias, em "Database Alias". Em "Driver Type", escolha o tipo do *Driver*.

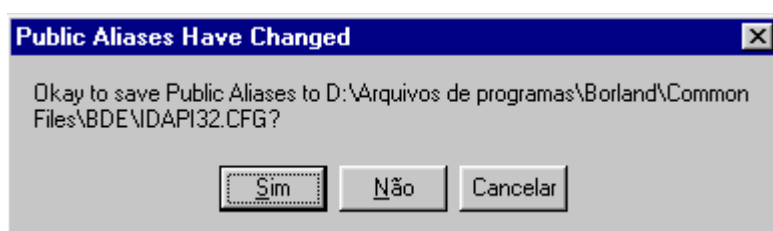
O tipo do driver é geralmente identificado pelo nome do banco de dados (como *Oracle*, *Access*, *Informix*, etc). O tipo *Standard* é o tipo padrão. Ele é usado para acessar bancos *Paradox* e *dBase*.



Quando você escolhe um tipo diferente de *Standard*, novos campos aparecem na caixa de diálogo. A maioria desses campos têm valores padrão que podem ser usados como estão. Algumas vezes, entretanto, é necessário alterar os valores desses novos campos. Consulte a documentação do banco de dados que você está usando (ou o *Help* do Delphi) para verificar quais mudanças são necessárias.

2. Na área "Path" digite o diretório para o Alias. O diretório deve ser um diretório existente (crie um novo diretório antes de criar o Alias, caso necessário).
3. Clique em **OK** para confirmar a criação do Alias. (Ou clique no botão **Keep New** para continuar criando outros Aliases).

É exibida a seguinte caixa de diálogo de confirmação:

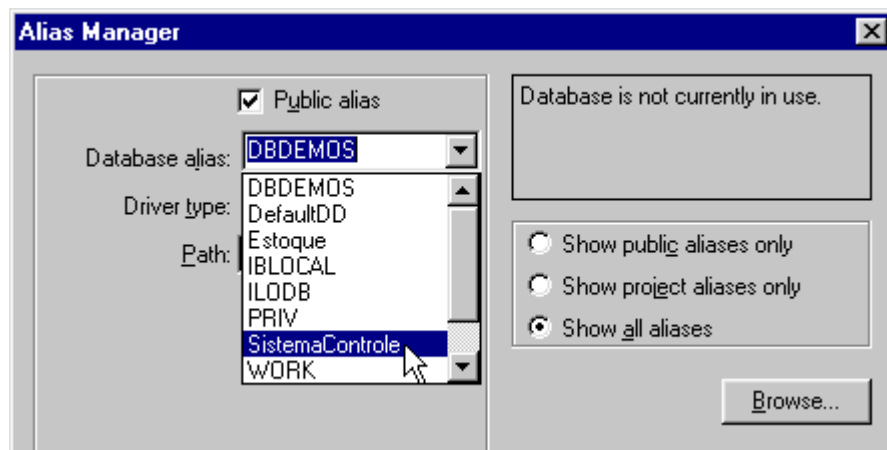


Os novos Alias (públicos) criados são armazenados no arquivo **IDAPI32.CFG**, no local indicado na caixa. A caixa de diálogo apenas confirma essa operação. Na maioria dos casos, basta clicar no botão **Sim** para confirmar.

Depois de criar um Alias, ele se torna disponível nas propriedades *DatabaseName* de vários componentes, tais como *Table*, *Query* e *Database*.

- **Para alterar um Alias:**

1. Escolha o comando **Tools | Alias Manager**.
1. Na parte de cima da caixa, escolha o Alias a ser alterado (figura abaixo).

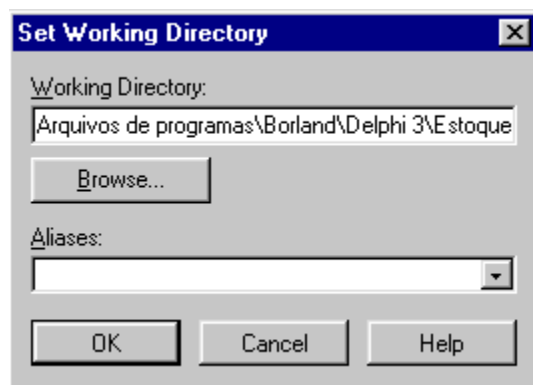


2. Faça as alterações necessárias e clique em **OK**.
3. Na caixa de diálogo que aparece, confirme as alterações clicando novamente em **OK**.

10.9. Alterando o diretório de trabalho

É recomendável definir um **diretório de trabalho** (*working directory*) durante o trabalho com o *DataBase Desktop*. Todos os arquivos criados no *DataBase Desktop* (como tabelas e índices) são armazenados nesse diretório de trabalho. O diretório de trabalho deve ser, geralmente, o Alias do banco de dados com que se está trabalhando no momento.

- **Para alterar o diretório de trabalho:**
1. Escolha o comando **File | Working directory**.



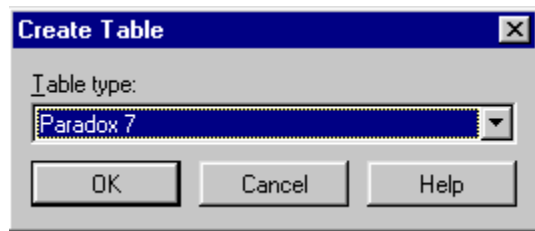
1. Na caixa de diálogo exibida, digite um caminho completo, ou escolha um Alias na parte de baixo da caixa.

10.10. Criando tabelas

O *DataBase Desktop* permite que sejam criadas tabelas de vários tipos de bancos de dados. Você cria uma tabela primeiro definindo a sua **estrutura**. A estrutura de uma tabela é, basicamente, o **nome**, o **tamanho** e o **tipo** de cada campo da tabela.

- **Para criar uma tabela:**
1. Escolha o comando **File | New > Table**.

A seguinte caixa de diálogo é exibida:



1. Escolha o tipo da tabela e clique em **OK** para começar a definir a estrutura. Os próximos passos assumem que foi escolhido o tipo de tabela **Paradox 7** (o tipo mais comum de tabela definido com o *DataBase Desktop*).
2. Há quatro colunas que devem ser preenchidas. Preencha cada coluna como descrito a seguir:

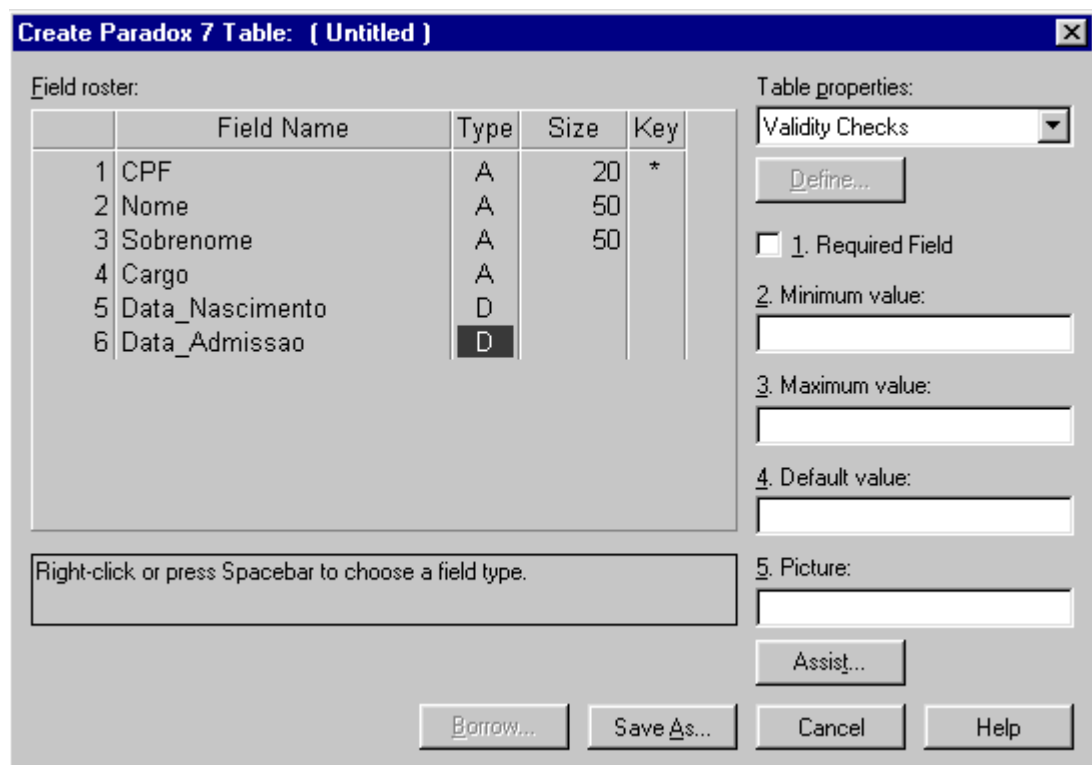
Field Name: o *nome* do campo.

Type: o *tipo* do campo (se é inteiro, monetário, booleano, etc.) Pressione a barra de espaços com o cursor nessa coluna e escolha um tipo com as setas, ou com o mouse.

Size: o *tamanho* do campo (em dígitos ou caracteres). O tamanho deve ser um valor entre 1 e 255. Essa coluna não pode ser alterada para alguns tipos (ela fica vazia, nesses casos).

Key: um asterisco (*) nessa coluna determina se o campo é ou não uma **chave**. Pressione a barra de espaços para mostrar ou esconder o asterisco. Vários campos podem ser chaves ao mesmo tempo (formando uma *chave composta*).

Use as setas do teclado, TAB ou ENTER para mover o cursor (quando possível), ou use o mouse.



Definindo a estrutura de uma tabela

4. Depois de definir o tipo e o tamanho de cada campo, clique no botão **Save As** para salvar a nova tabela no disco.

É mostrada um caixa do tipo "Salvar como" onde você deve digitar o nome do arquivo a ser salvo. A tabela é salva no *diretório de trabalho* (veja a seção anterior).

10.11. Adicionando dados a uma tabela

Uma vez criadas as tabelas de um banco de dados, você pode preenchê-las com dados dentro do próprio *Database Desktop*. (Esse recurso é geralmente usado somente para adicionar alguns dados de amostra, pois os dados são normalmente adicionados a partir dos aplicativos).

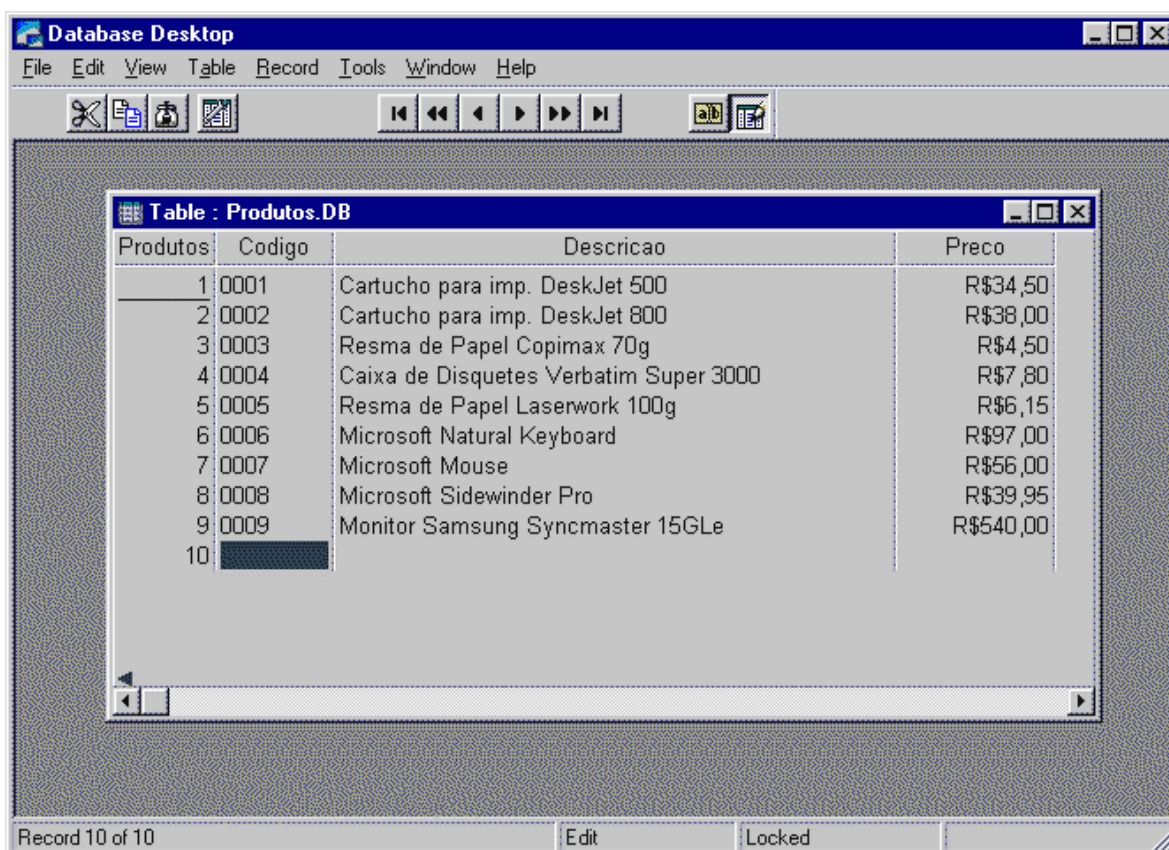
- **Para adicionar dados a uma tabela (ou alterar dados):**

1. Escolha o comando **File | Open > Table**.

1. Na caixa que aparece, escolha a tabela a ser alterada.

Os arquivos exibidos na caixa são os do *diretório de trabalho*. Escolha um Alias na parte de baixo para mostrar arquivos de outro diretório.

2. A tabela é aberta e exibida em uma janela. Se a tabela contiver dados, estes são exibidos também. (Na figura a seguir, é mostrada uma pequena tabela no *Database Desktop*, com três campos e alguns dados já digitados).



3. Quando a tabela é exibida, seus dados são inicialmente protegidos contra alterações. Para fazer alterações ou adicionar novos dados, escolha o comando **Table | View Data**, ou pressione **F9**.

4. Faça as alterações ou digite novos dados normalmente. Use o mouse, a tecla TAB, ou as setas do teclado para mover o cursor e depois digite os valores desejados.
5. As alterações são salvas automaticamente a cada registro (linha da tabela) digitado. Para terminar a edição dos dados, simplesmente feche a janela onde é exibida a tabela.

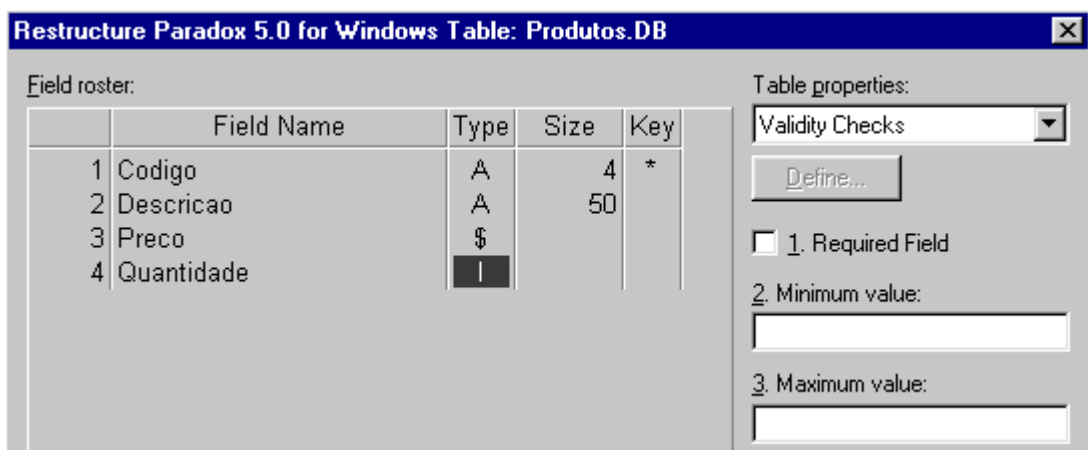
10.12. Alterando a estrutura de uma tabela

Algumas vezes, é necessário adicionar campos ou retirar campos de uma tabela, alterando sua estrutura. Você pode fazer isso facilmente no *DataBase Desktop*.

- **Para alterar a estrutura de uma tabela:**

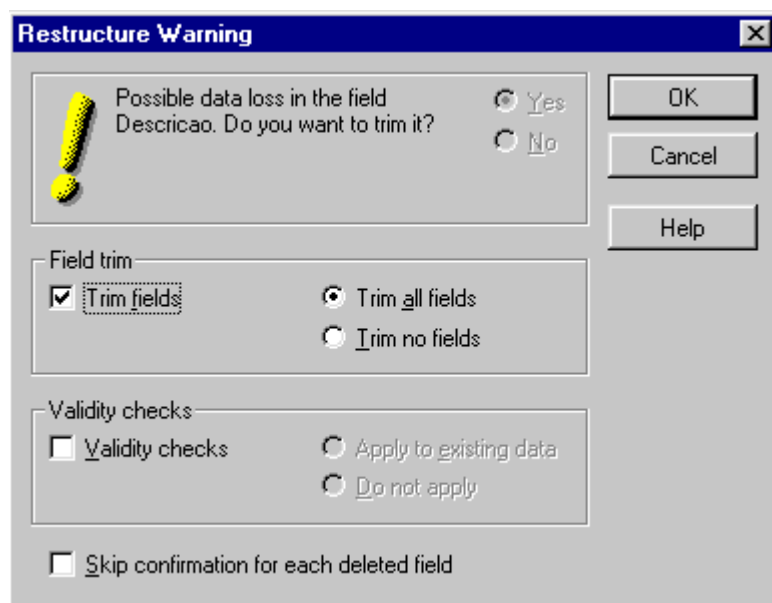
1. Se a tabela não estiver aberta, use o comando **File | Open > Table** para abri-la.
1. Escolha o comando **Table | Restructure**.

A caixa de diálogo com a estrutura da tabela é exibida (figura a seguir).



2. Pode-se adicionar ou retirar campos da tabela, ou alterar o nome, o tamanho ou o tipo de campos já existentes. Pode haver perdas nesse processo. Veja o que pode acontecer:
 - Se um campo for retirado, todos os dados contidos naqueles campos são apagados permanentemente.
 - Se o tamanho de um campo for reduzido, valores com comprimento maior que o novo tamanho serão cortados.

Clique no botão **Save** para salvar as alterações. Se houver algum perigo de perdas de dados, é exibida um caixa de aviso como a seguinte:



1. A caixa ilustrada pode ser exibida várias vezes, uma vez para cada campo com problemas. Geralmente, basta clicar em **OK** para confirmar as alterações na estrutura da tabela.

CAPÍTULO 11 - TRABALHANDO COM DATASETS

Um DataSet é um conjunto de dados organizado em forma de tabela (em linhas e colunas). As colunas são os **campos** e as linhas são os **registros**. Todo o acesso a bancos de dados no Delphi é feito através de DataSets. Os componentes *Table* e *Query* são os tipos principais de DataSets. Neste capítulo, veremos as propriedades, eventos e métodos dos DataSets. Tudo que veremos aqui vale para os componentes *Table* e *Query*.

11.1. Abrindo e fechando DataSets

Para alterar ou ler os dados em uma DataSet, você deve primeiro abrir o DataSet.

- **Para abrir um DataSet, realize uma das seguintes operações:**
- Altere a propriedade *Active* do DataSet para *True*.

Isso pode ser feito em tempo de desenvolvimento no Object Inspector, ou em tempo de execução. O seguinte comando abre o componente chamado "Table1":

```
Table1.Active := True;
```

- Use o método **Open** no DataSet, como abaixo:

```
Query1.Open;
```

Quando um DataSet é aberto os dados conectados a ele são lidos e exibidos automaticamente (se houver componentes onde os dados possam ser exibidos, é claro). No exemplo do capítulo anterior, abrimos um componente *Table* em um formulário para que os dados fossem exibidos imediatamente em um componente *DBGrid*.

Você deve sempre *fechar* um DataSet depois de usá-lo, para liberar recursos do sistema.

- **Para fechar um DataSet, realize uma das seguintes operações:**
- Altere a propriedade *Active* do DataSet para *False*.
- Use o método **Close** no DataSet, como em `Table1.Close`

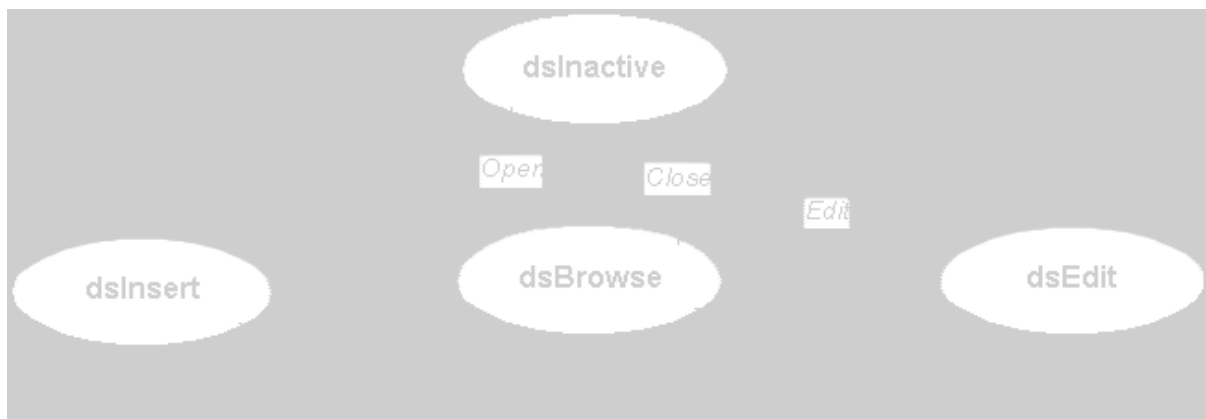
11.2. Estados de um DataSet

Um DataSet pode estar em vários *estados* diferentes. O estado de um DataSet determina o que pode ser feito (ou está sendo feito) com o DataSet. O valor da propriedade *State* de um DataSet determina o seu estado atual. Veja a seguir uma descrição breve dos estados mais importantes em que pode estar um DataSet.

Estado (valor de <i>State</i>)	Significado
<i>dsInactive</i>	O Dataset está fechado. Seus dados não estão disponíveis (não podem ser lidos nem alterados).
<i>dsBrowse</i>	O Dataset está aberto. Seus dados podem ser visualizados, mas não podem ser alterados. Este é o estado padrão de um DataSet.

<i>dsEdit</i>	O DataSet está aberto. O registro atual pode ser modificado.
<i>dsInsert</i>	O DataSet está aberto. Um novo registro acaba de ser inserido.

O estado **dsBrowse** é o estado padrão. Quando um DataSet é aberto, ele é colocado automaticamente neste estado. Vários **métodos** de um DataSet podem ser usados para alterar o seu estado. Na ilustração a seguir, são mostrados os quatro estados mais importantes e os métodos que podem ser usados para passar de um estado para outro.



Estados de um DataSet

Para usar um dos métodos da ilustração, simplesmente use o nome do DataSet seguido pelo nome do método. O trecho de código abaixo, por exemplo, altera cinco vezes o estado de um componente *Table*.

```

procedure TForm1.Button1Click(Sender: TObject);

begin

    Table1.Open; // O estado muda para dsBrowse ...

    Table1.Edit; //... muda para dsEdit...

    Table1.Insert; //... muda novamente para dsInsert ...

    Table1.Post; //... volta a dsBrowse ...

    Table1.Close; //... e finalmente muda para dsInactive

end;

```

11.3. Navegando em um DataSet

Os DataSets teriam pouca utilidade se não fosse possível percorrer e consultar (navegar) os seus registros. Há vários métodos e propriedades úteis para a navegação de DataSets.

Para permitir a navegação de seus registros, todo DataSet contém um **cursor** que indica o **registro atual** (ou linha atual) do DataSet. É no registro atual que são feitas alterações, ou onde são inseridos (ou removidos) registros. Todos os métodos de navegação alteram a *posição do cursor*. Veja uma descrição breve desses métodos na tabela a seguir:

Método	Descrição
<i>First</i>	Move o cursor para o primeiro registro do DataSet.
<i>Last</i>	Move o cursor para o último registro do DataSet.
<i>Next</i>	Move o cursor para o próximo registro do DataSet (imediatamente <i>depois</i> do registro atual). Se o cursor já estiver no último registro, nada acontece.
<i>Prior</i>	Move o cursor para o registro anterior do DataSet (imediatamente antes do registro atual). Se o cursor já estiver no primeiro registro, nada acontece.
<i>MoveBy(num)</i>	<p>Move o cursor o número de registros especificado em <i>num</i>. Um valor positivo move o cursor para frente; um valor negativo move-o para trás. Por exemplo, <code>Table1.moveBy(-10)</code> move o cursor 10 registros para trás na tabela <i>Table1</i>.</p> <p>Se o número de registros especificado for maior do que o número que se pode mover, o cursor é movido para o primeiro ou o último registro, dependendo da direção do movimento.</p>

Além dos métodos descritos acima, há duas propriedades que indicam se o cursor chegou ao final ou ao início de um DataSet: *BOF* e *EOF*.

Propriedade	Descrição
BOF	<p>BOF é alterado para <i>True</i> quando o cursor está no primeiro registro do DataSet. BOF é a abreviação de <i>Begin of File</i> – "Início do Arquivo".</p> <p>Quando o cursor estiver em qualquer registro que não seja o primeiro do DataSet, o valor de BOF é <i>False</i>.</p>
EOF	<p>EOF é alterado para <i>True</i> quando o cursor está no último registro do DataSet. EOF é a abreviação de <i>End of File</i> – "Final do Arquivo".</p> <p>Quando o cursor estiver em qualquer registro que não seja o último do DataSet, o valor de EOF é <i>False</i>.</p>

Veja a seguir dois exemplos que usam os métodos e propriedades vistas acima. O primeiro exemplo percorre uma query "Query1" do primeiro registro até o último, somando todos os valores no campo "Quantidade". `FieldValues[NomeDoCampo]` é usado para obter o valor de cada campo. Uma mensagem com o valor total é mostrada no final.

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
Total: Double;
```

```
begin  
  
Query1.Open; {Abrir a Query}  
  
Query1.First; {Mover para o primeiro registro}  
  
Total := 0;  
  
while not (Query1.EOF) do  
  
    begin  
  
        Total := Total + Query1['Quantidade'];  
  
        Query1.Next; {Mover para o próximo registro}  
  
    end;  
  
    ShowMessage('Quantidade total: ' + FloatToStr(Total));  
  
    Query1.Close {Fechar a Query}  
  
end;
```

Este segundo exemplo percorre os registros do componente *Table* "Table1" do último até o primeiro, calculando o valor total do estoque: a soma de todos os preços multiplicados pelas quantidades dos produtos. O exemplo é semelhante ao anterior, a não ser pela ordem inversa e o uso de um componente *Table* em vez de um *Query*. Note que obviamente a ordem inversa não é obrigatória. Ela foi usada somente para ilustrar *BOF* e os métodos *Last* e *Prior*.

```
procedure TForm1.Button2Click(Sender: TObject);  
  
var  
  
    Valor: Currency;  
  
begin  
  
    Table1.Last; {Mover para o último registro}  
  
    Valor := 0;  
  
while not (Table1.BOF) do  
  
    begin  
  
        Valor := Valor + Table1['Quantidade'] * Table1['Preco'];  
  
        Table1.Prior; {Mover para o registro anterior}  
  
    end;  
  
    ShowMessage('Valor do estoque: ' + FloatToStr(Valor));  
  
end;
```

11.4. Modificando Datasets

Pode-se alterar um DataSet diretamente, modificando valores campo a campo, ou adicionando e removendo registros inteiros. Os seguintes métodos permitem fazer essas alterações:

Método	Descrição
<i>Edit</i>	Coloca o DataSet no estado <i>dsEdit</i> . Isto permite a alteração dos valores do registro atual. Muitos componentes chamam esse método implicitamente para permitir a alteração direta dos valores de um DataSet. O componente <i>DBGrid</i> , por exemplo, entra no estado <i>dsEdit</i> usando o método <i>Edit</i> , quando se dá um duplo clique em um dos registros.
<i>Append</i>	Adiciona um registro vazio ao final do DataSet. O estado do DataSet muda para <i>dsInsert</i> .
<i>Insert</i>	Adiciona um registro vazio na posição atual do cursor. O estado do DataSet muda para <i>dsInsert</i> (como para o método <i>Append</i>).
<i>Post</i>	<p>Tenta enviar o novo registro ou o registro alterado para o banco de dados. Se tudo correr bem, o DataSet é colocado no estado <i>dsBrowse</i>. Caso contrário, o estado do DataSet não é alterado. O comando <i>Post</i> é um tipo de confirmação da última entrada.</p> <p>Muitos componentes chamam <i>Post</i> automaticamente (quando se passa de um registro para outro em um <i>DBGrid</i>, por exemplo).</p>
<i>Cancel</i>	Cancela a última operação (uma alteração em um registro, por exemplo) e coloca o DataSet no estado <i>dsBrowse</i> .
<i>Delete</i>	Apaga o registro atual e coloca o DataSet no estado <i>dsBrowse</i> .

11.5. Modificando campos

O valor de um campo do registro atual de um DataSet pode ser alterado de várias maneiras:

- Especificando o nome do campo diretamente:

```
Tabela['Preco'] := 54.43;
```

- Usando a propriedade *FieldValues*:

```
Tabela.FieldValues['Preco'] := 54.43;
```

- Usando o método *FieldByName* e propriedades de conversão:

```
Tabela.FieldByName('Preco').AsCurrency := 54.43;
```

Para que seja possível alterar os valores dos campos, o DataSet deve estar no estado *dsEdit* ou *dsInsert* (use *Edit*, *Insert* ou *Append* para colocá-lo em um desses estados). Depois de realizar as alterações, você deve confirmá-las, usando o método *Post*. Veja um exemplo que usa *FieldValues*:

```
procedure TForm1.Button1Click(Sender: TObject);  
  
var  
  
NovaDesc: String;  
  
begin  
  
NovoNome := Edit1.Text;  
  
with TabProdutos do  
  
begin  
  
Edit; //Preparar para alteração  
  
FieldValues['Descricao'] := NovaDesc;  
  
Post; //Confirmar alteração  
  
end;  
  
end;
```

No exemplo, o campo "Descricao" do registro atual do componente *Table* "TabProdutos" é alterado para o valor digitado no componente *Edit1*. O método *Edit* é usado para colocar a tabela no estado *dsEdit*, antes de fazer a alteração. O método *Post*, confirma a alteração, tornando-a permanente.

11.6. Adicionando registros

Os métodos *Insert* e *Append* são usados para adicionar novos registros a um *DataSet*. *Insert* adiciona um registro vazio na posição atual do cursor; *Append* adiciona um registro vazio depois do último registro do *DataSet*. Os dois métodos colocam o *DataSet* no estado *dsInsert*.

NOTA: Para tabelas *indexadas* do tipo *Paradox* e *dBASE*, os comandos *Insert* e *Append* têm o mesmo efeito. Eles adicionam um novo registro na posição determinada pelo *índice* da tabela.

O exemplo a seguir insere um novo registro depois do décimo registro da tabela. *Moveby(10)* move o cursor 10 registros adiante; *Insert* insere o novo registro na posição do cursor. Em seguida, valores são definidos para os campos "Descricao", "Preco" e "Quantidade" e as alterações são confirmadas com *Post*.

```
procedure TForm1.Button1Click(Sender: TObject);  
  
begin  
  
with TabProdutos do  
  
begin
```

```
First;  
  
MoveBy(10);  
  
Insert;  
  
FieldByName('Codigo').AsInteger := 78;  
  
FieldByName('Descricao').AsString := 'Televisão ToNaBoa';  
  
FieldByName('Preco').AsCurrency := 754.00;  
  
FieldByName('Quantidade').AsInteger := 24;  
  
Post;  
  
end;  
  
end;
```

11.7. Apagando registros

Pode-se apagar registros inteiros rapidamente, com o método *Delete*. *Delete* apaga o registro atual, sem confirmações, e coloca o DataSet no estado *dsBrowse*. O registro imediatamente depois do registro apagado se torna o registro atual.

11.8. Confirmando e cancelando mudanças

Depois de inserir um registro, ou modificar os valores dos seus campos, você pode usar o método *Post* para tornar as mudanças permanentes, ou usar o comando *Cancel* para cancelar essas mudanças.

Post é chamado automaticamente, quando o cursor é movido para outro registro, usando os métodos *First*, *Last*, *Prior*, *Next*, ou *MoveBy*. Os métodos *Insert* e *Append* também chamam o método *Post*, tornando permanentes as mudanças que haviam sido feitas antes do registro ser adicionado.

Durante a alteração de um registro, ou logo depois da inserção de um novo registro, é possível desfazer alterações recentes, usando o método *Cancel*. Na maioria dos componentes *Data Controls*, o método *Cancel* é chamado automaticamente quando o usuário pressiona a tecla ESC, durante a edição de um campo. Quando o método *Cancel* é chamado, os valores dos campos do registro atual reverterem para os valores antes da alteração. No caso em que um novo registro acabou de ser inserido, o método *Cancel* remove o novo registro.

O exemplo abaixo adiciona um novo registro a uma tabela com quatro campos (chamada "TabProdutos"). Os valores inseridos são lidos a partir de quatro *Edits*. As mudanças só têm efeito, entretanto, depois da confirmação do usuário (feita com a função *MessageDlg*). Se o usuário confirmar a alteração, é usado o método *Post* para enviar os dados ao banco de dados; se não, o método *Cancel* cancela todas as mudanças.

```
procedure TForm1.Button1Click(Sender: TObject);  
  
var  
  
NovoCodigo: Integer;
```

```
NovaDescricao: String;

NovoPreco: Currency;

NovaQuant: Integer;

begin

    // Lê os valores nos quatro Edits

    NovoCodigo := StrToInt(EditCodigo.Text);

    NovaDescricao := EditDescricao.Text;

    NovoPreco := StrToCurr(EditPreco.Text);

    NovaQuant := StrToInt(EditQuant.Text);

    with TabProdutos do

        begin

            Open; // Abre a tabela, colocando-a no estado dsBrowse

            Append; // Cria um novo registro no final (estado dsInsert)

            // Altera os valores de cada campo

            FieldValues['Codigo'] := NovoCodigo;

            FieldValues['Descricao'] := NovaDescricao;

            FieldValues['Preco'] := NovoPreco;

            FieldValues['Quantidade'] := NovaQuant;

            // Confirma as alterações

            if MessageDlg('Confirma alteração?', mtConfirmation,

                mbYesNoCancel,0) = mrYes

            then Post

            else Cancel;

            end;

        end;
```

11.9. Inserindo e Modificando registros inteiros

Há três métodos que permitem inserir ou alterar registros inteiros: *InsertRecord*, *AppendRecord* e *SetFields*.

Método	Descrição
<i>InsertRecord([valor1, valor2, ...])</i>	<p>Insere um novo registro, na posição atual do cursor, com os valores especificados. A lista de valores deve estar entre colchetes e os valores devem ser separados por vírgulas. Pode-se usar nil para especificar um valor nulo (vazio) para um ou mais campos.</p> <p>A ordem em que os valores são especificados deve ser a ordem dos campos no DataSet. Se o número de valores especificados for menor que o número de campos na tabela, o restante dos campos do novo registro são preenchidos com nil. O método <i>InsertRecord</i> chama <i>Post</i> implicitamente no final.</p>
<i>AppendRecord([valor1, valor2, ...])</i>	<p>Idêntico ao método <i>InsertRecord</i>, mas o registro é adicionado ao final do DataSet. Se o DataSet for uma tabela indexada, os métodos <i>AppendRecord</i> e <i>InsertRecord</i> têm exatamente o mesmo efeito – a posição do registro adicionado depende do índice da tabela. O método <i>AppendRecord</i> também chama <i>Post</i> implicitamente, tornando as alterações permanentes.</p>
<i>SetFields([valor1, valor2, ...])</i>	<p>Altera os valores dos campos correspondentes do registro atual. O DataSet deve estar no estado <i>dsEdit</i> para que <i>SetFields</i> possa ser usado (use o método <i>Edit</i>). Para alterar apenas alguns campos do registro atual, use nil como argumento para cada campo que <i>não</i> deseja alterar.</p> <p>O método <i>Post</i> não é chamado implicitamente por <i>SetFields</i>. Deve-se chamar <i>Post</i> depois do método para tornar as alterações permanentes.</p>

O exemplo a seguir usa o método *InsertRecord* para inserir um novo registro na tabela *TabProdutos* (esta tabela tem os campos *Codigo*, *Descricao*, *Preco* e *Quantidade*). O exemplo altera somente os dois primeiros campos (usando os valores lidos) e o último campo, assumindo uma quantidade fixa de 12 para este. Para o terceiro campo (o *preço*) é especificado como **nil**. Isso deixa o campo vazio (ou inalterado).

Em seguida, o programa pergunta se o campo *Preco* deve ser preenchido ou não. Caso positivo, é usado o método *SetFields* para alterar *somente o preço* do registro atual (note como **nil** é usado para todos os outros campos).

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
NovoCodigo: Integer;
```

```
NovaDescricao: String;
```



```
NovoPreco: Currency;  
  
begin  
  
NovoCodigo := StrToInt(EditCodigo.Text);  
  
NovaDescricao := EditDescricao.Text;  
  
with TabProdutos do  
  
begin  
  
Open;  
  
//Insere um novo registro  
  
InsertRecord([NovoCodigo, NovaDescricao, nil, 12]);  
  
// Confirma a entrada do preco  
  
if MessageDlg('Entrar com preço agora?',  
mtConfirmation, [mbYes, mbNo], 0) = mrYes then  
  
//Realiza a alteracao do preco (o campo estava vazio)  
  
begin  
  
NovoPreco := StrToCurr(InputBox('', 'Digite o preço', ''));  
  
Edit; //Coloca o Table no estado dsEdit (essencial)  
  
SetFields([nil, nil, NovoPreco, nil]); //Altera o preco  
  
Post; //Confirma a alteração  
  
end;  
  
end;  
  
end;
```

11.10. Localizando registros com *Locate*

O método *Locate* é maneira mais versátil e mais rápida de **localizar registros** em um DataSet. Como argumentos do método *Locate*, você especifica o nome dos campos a serem consultados, o valor desejado para cada campo especificado e um conjunto de opções de localização (que pode ser vazio). Veja a seguir um exemplo de uma chamada ao método *Locate*:

```
TabClientes.Locate('Nome;Sobrenome', VarArrayOf(['Maria', 'S']),[loPartialKey]);
```

O exemplo localiza o primeiro registro da tabela *TabClientes*, com nome "Maria" e sobrenome começando com "S".

O primeiro parâmetro de *Locate* é uma lista de campos separados por ponto-e-vírgula, entre aspas simples. O segundo parâmetro é um array especial que deve ser construído com o comando *VarArrayOf* (para o caso de uma pesquisa em mais de um campo). Os valores aqui devem "casar" com os campos especificados no primeiro parâmetro. O terceiro parâmetro especifica opções de procura.

Há duas opções de procura disponíveis para o método *Locate*:

<i>loPartialKey</i>	Permite que os valores de procura especificados no segundo parâmetro sejam <i>parciais</i> (parte de um nome, por exemplo).
<i>loCaseInsensitive</i>	Faz com que <i>Locate</i> ignore maiúsculas e minúsculas. Com essa opção "maria", "Maria" e "MARIA" são considerados iguais, por exemplo.

Se houver algum registro que case com os valores especificados, o método *Locate* retorna *True* e move o cursor para aquele registro. Se nenhum registro for encontrado, *Locate* retorna *False*.

Se a localização for baseada em apenas um campo, a sintaxe de *Locate* é mais simples. Veja um exemplo, que localiza o cliente com código igual a 100.

```
TabClientes.Locate('Codigo', 100, []);
```

Agora veja um exemplo que lê os valores em dois componentes *Edit* ("EditNome" e "EditSobrenome") e localiza o primeiro registro que casa com esses valores, mostrando uma mensagem com o resultado da localização.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NomeProc, SobrenomeProc: String; //Valores procurados
  Achou: Boolean; //Resultado da pesquisa
begin
  NomeProc := EditNome.Text;
  SobrenomeProc := EditSobrenome.Text;
  Achou := TabFunc.Locate('Nome;Sobrenome',
    VarArrayOf([NomeProc,SobrenomeProc]),
    [loPartialKey, loCaseInsensitive]);
  if Achou then
    ShowMessage('Registro encontrado.')
  else
    ShowMessage('Registro não encontrado.');
```

11.11. Filtrando DataSets

Algumas vezes é necessário exibir apenas um subconjunto dos registros de uma *DataSet* que atendam a alguma condição. Isso pode ser feito usando uma consulta SQL, ou aplicando um *filtro*.

As consultas SQL são mais poderosas, mas também bem mais complexas. Elas serão vistas mais adiante. Os filtros são mais simples e podem ser aplicados diretamente a um componente *Table*, ou a um componente *Query* (e a outros *DataSets*).

Filtros podem ser definidos de duas maneiras: usando as propriedades *Filter* e *Filtered*, ou usando o evento *OnFilterRecord*.

11.12. Usando Filter e Filtered

Para filtrar um *DataSet*, defina uma condição para a propriedade *Filter* e altere a propriedade *Filtered* para *True*. Isso pode ser feito em tempo de desenvolvimento ou em tempo de execução. A condição na propriedade *Filter* deve especificar uma expressão lógica, com os nomes dos campos entre aspas simples. Veja alguns exemplos de condições:

- 'Estado' = 'PE';
- 'Quantidade' < 100;
- 'Resposta' <> False
- ('Salario' > 10000) AND ('Salario' <= 50000)
- ('Estado' = 'RJ') OR ('Estado' = 'SP')

Se o *DataSet* estiver aberto (propriedade *Active* = *True*), a filtragem dos registros é feita imediatamente, mesmo em tempo de desenvolvimento. Somente os registros para os quais a condição é verdadeira são mostrados. Para componentes que acessam uma tabela filtrada, é como se outros registros não existissem.

11.13. Usando o evento OnFilterRecord

Usando o evento *OnFilterRecord*, você pode definir filtros muito mais elaborados, usando os recursos da linguagem Object Pascal. O evento *OnFilterRecord* recebe o parâmetro booleano *Accept* cujo valor, no final do código do evento, determina se o registro será exibido ("aceito") ou não. Uma referência ao *DataSet* que está sendo filtrado também é passada como parâmetro.

Para cada registro no *DataSet*, o evento *OnFilterRecord* é chamado e o parâmetro *Accept* é verificado. Se *Accept* for *True* (o valor padrão) o registro é exibido; se *Accept* for *False*, o registro não é exibido (ele é filtrado). Veja um exemplo:

```
procedure TForm1.TabProdutosFilterRecord(DataSet: TDataSet;  
  
var Accept: Boolean);  
  
begin  
  
if DataSet['Descricao'] >= 'M' then  
  
if DataSet['Preco'] * DataSet['Quantidade'] < 2000.00 then  
  
Accept := True;  
  
end;
```

11.14. Eventos dos DataSets

Os *DataSets* oferecem vários eventos que podem ser usados para verificar e validar mudanças antes de se tornarem permanentes. Há eventos associados a todos os métodos mais importantes, como *Open*, *Close*, *Insert* e *Edit*. Para a maioria desses métodos há dois eventos:

um que ocorre antes de o método ser chamado (prefixo "Before"), e outro que ocorre depois (prefixo "After"). Veja uma descrição breve de cada um desses eventos:

Estes eventos...	...ocorrem:
<i>BeforeOpen, AfterOpen</i>	Antes e depois de o DataSet ser aberto.
<i>BeforeClose, AfterClose</i>	Antes e depois de o DataSet ser fechado.
<i>BeforeInsert, AfterInsert</i>	Antes e depois de o DataSet entrar no estado <i>dsInsert</i> .
<i>BeforeEdit, AfterEdit</i>	Antes e depois do DataSet entrar no estado <i>dsEdit</i> .
<i>BeforeCancel, AfterCancel</i>	Antes e depois do comando <i>Cancel</i> ser chamado (implicitamente ou explicitamente).
<i>BeforePost, AfterPost</i>	Antes e depois de as mudanças em um DataSet serem enviados para o Banco de Dados (comando <i>Post</i>).
<i>BeforeDelete, AfterDelete</i>	Antes e depois de um registro ser apagado.
<i>OnNewRecord</i>	Quando um novo registro é adicionado. Usado geralmente para definir valores padrão para alguns campos.
<i>OnCalcFields</i>	Quando os campos calculados do DataSet são calculados.

11.15. Controlando a atualização de componentes

Componentes associados a dados de um banco de dados, como o *DBGrid* e a maioria dos outros componentes *Data Controls*, são atualizados automaticamente quando o cursor é movido de um registro para outro. Muitas vezes, essa atualização não é desejada como, por exemplo, quando o programa faz uma pesquisa em uma tabela, ou faz atualizações em vários registros.

A atualização dos componentes pode ser desabilitada temporariamente usando o método *DisableControls* e depois reabilitada com *EnableControls*. Geralmente, esses métodos são usados em um bloco **try-finally** para que os componentes sejam reabilitados mesmo se ocorrer uma exceção no processamento do DataSet. Veja a seguir um exemplo que usa esse dois métodos.

O exemplo aumenta, em 8%, o salário de todos os funcionários cadastrados no *Table* "TabFuncionarios". Os componentes associados ao *Table* são desabilitados antes do processamento da tabela e são reabilitados no final, na parte **finally**, mesmo se houver exceções durante o processamento.

```

procedure TForm1.Button1Click(Sender: TObject);

begin

    with TabFuncionarios do

        begin

            try

```

```
Open;  
  
Edit;  
  
DisableControls; //Desabilitar atualização  
  
while not EOF do  
  
begin  
  
FieldValues['Salario'] := FieldValues['Salario']*1.08;  
  
Next;  
  
end;  
  
finally  
  
EnableControls; //Habilitar atualização  
  
end;  
  
end;  
  
end;
```

Há mais um método relacionado à atualização de componentes: o método **Refresh**. Esse método *força a atualização* dos componentes, fazendo com que os dados sejam buscados novamente no banco de dados. *Refresh* é útil quando são feitas alterações no registro atual por outro usuário, por exemplo. Uma chamada a *Refresh* garante que os dados exibidos são os mais atuais.

CAPÍTULO 12 - COMPONENTES *DATASOURCE* E *TABLE*

12.1. Usando o componente *DataSource*

O componente *DataSource* funciona como um "canal de comunicação" entre *DataSets* e os componentes *DataControls*. Todo *DataSet* (*Table*, *Query*, etc.) deve ser associado a um componente *DataSource* para que seus dados possam ser exibidos em componentes *Data Controls*.

12.2. Propriedades do componente *DataSource*

O componente *DataSource* é um componente simples, com apenas cinco propriedades, duas das quais (*Name* e *Tag*) são comuns a todos os componentes do Delphi. Veja uma descrição das outras três:

Propriedade	Descrição
<i>AutoEdit</i>	Determina se o <i>DataSet</i> ligado ao <i>DataSource</i> entra no estado <i>dsEdit</i> automaticamente quando o usuário clica no componente associado (um <i>DBGrid</i> , por exemplo). Lembre-se que um <i>DataSet</i> no estado <i>dsEdit</i> pode ter seus registros alterados. O padrão para essa propriedade é <i>True</i> . Se for especificado o valor <i>False</i> , o <i>DataSet</i> deve ser colocado no estado <i>dsEdit</i> explicitamente, usando o método <i>Edit</i> , para que seja possível fazer alterações nele.
<i>DataSet</i>	Contém o nome do <i>DataSet</i> associado ao <i>DataSource</i> .
<i>Enabled</i>	Determina se o <i>DataSource</i> está ou não ativo (<i>True</i> = ativo). Quando <i>Enabled</i> = <i>False</i> (desativado) todos os componentes ligados ao <i>DataSource</i> aparecem vazios (não exibem dados).

12.3. Eventos do componente *DataSource*

Há apenas três eventos para o componente *DataSource*:

Evento	Descrição
<i>OnDataChange</i>	Este evento ocorre quando o cursor do <i>DataSet</i> é movido para outro registro, depois de alterações no <i>DataSet</i> . Isso acontece, por exemplo, quando os métodos <i>First</i> , <i>Last</i> , <i>Next</i> ou <i>Prior</i> são chamados. Você pode usar esse evento para sincronizar os dados exibidos com os dados de um banco de dados (no caso em que componentes comuns são usados para exibir dados, por exemplo).
<i>OnStateChange</i>	Este evento ocorre quando o <i>estado</i> do <i>DataSet</i> associado ao <i>DataSource</i> é alterado.
<i>OnUpdateData</i>	Ocorre imediatamente antes de uma atualização no <i>DataSet</i> associado – depois de um comando <i>Post</i> , mas antes dos dados serem realmente

<p>atualizados. O evento ocorre mesmo se o comando <i>Post</i> for chamado implicitamente por outro comando ou outro componente.</p>
--

12.4. Usando o componente *Table*

Um componente *Table* contém todas as informações de uma tabela de um banco de dados. Vimos anteriormente que um componente *Table* é um *DataSet*, portanto já conhecemos vários métodos e propriedades desse componente. Na verdade, por ser o tipo de *DataSet* mais comum, o componente *Table* foi o componente usado na maioria dos exemplos do capítulo anterior. Neste capítulo veremos mais detalhes sobre algumas propriedades e métodos já conhecidos. Veremos também como usar vários recursos que são exclusivos dos componentes *Table*.

12.5. Conectando-se a uma tabela de banco de dados

No capítulo "Trabalhando com bancos de dados no Delphi: uma visão geral", vimos um exemplo completo onde conectamos uma componente *Table* a um banco de dados e exibimos o seu conteúdo em um componente *DBGrid*. Veja aquele capítulo para uma descrição passo-a-passo sobre como realizar a conexão com um banco de dados.

O procedimento para a conexão precisa alterar três propriedades do componente *Table*. Veja a seguir uma descrição de cada uma dessas propriedades.

Propriedade	Descrição
<i>Active</i>	<p><i>Active</i> determina se a tabela está aberta ou não. Esta propriedade pode ser alterada em tempo de desenvolvimento para que os registros da tabela sejam exibidos, mesmo antes da execução do aplicativo.</p> <p>A propriedade <i>Active</i> deve estar em <i>False</i> para que seja possível alterar várias propriedades da tabela, inclusive as propriedades <i>DatabaseName</i> e <i>TableName</i>. Quando estas duas propriedades são alteradas, o Delphi altera automaticamente <i>Active</i> para <i>False</i>.</p>
<i>DatabaseName</i>	<p><i>DatabaseName</i> é o nome do banco de dados que contém a tabela associada. O nome especificado aqui é geralmente um Alias definido no <i>DataBase Desktop</i>.</p> <p>Para tabelas locais, como as dos bancos Paradox e dBASE, pode-se também especificar o <i>diretório</i> onde as tabelas do banco de dados estão localizadas.</p>
<i>TableName</i>	<p><i>TableName</i> é o nome da tabela do banco à qual o componente <i>Table</i> está associado. Quando a propriedade <i>DatabaseName</i> é definida primeiro, as tabelas do banco de dados são listadas automaticamente para essa propriedade.</p>

12.6. Controlando o acesso a uma tabela

Para alguns bancos de dados, principalmente os que usam SQL, há um controle rígido sobre a leitura e a escrita dos registros de uma tabela. Há três propriedades dos componentes *Table*

que permitem verificar permissões ou controlar o acesso às tabelas de um banco de dados. Veja a seguir uma descrição dessas propriedades:

Propriedade	Descrição
<i>CanModify</i>	Esta é uma propriedade somente leitura que indica se a tabela associada ao componente <i>Table</i> pode ser alterada ou não. Você pode verificar o valor de <i>CanModify</i> para se certificar que os dados da tabela podem ser alterados.
<i>ReadOnly</i>	Determina se o usuário pode visualizar e alterar os dados da tabela, ou apenas visualizar esses dados. O valor de <i>ReadOnly</i> é controlado pelo aplicativo e não pelo banco de dados, como acontece para a propriedade <i>CanModify</i> . Você pode alterar a propriedade <i>ReadOnly</i> para <i>True</i> (o padrão é <i>False</i>) para exibir os dados em uma tabela sem, no entanto, permitir alterações do usuário.
<i>Exclusive</i>	Esta propriedade é usada apenas para tabelas <i>Paradox</i> , <i>dBASE</i> ou <i>FoxPro</i> . Altere-a para <i>True</i> para que apenas um usuário de cada vez tenha acesso à tabela. Se o valor de <i>Exclusive</i> for <i>False</i> , será permitido o acesso à tabela por vários usuários ao mesmo tempo.

12.7. Trabalhando com *Ranges*

Algumas vezes, especialmente quando se está trabalhando com tabelas extensas, é útil restringir os registros exibidos a apenas um subconjunto dos registros de uma tabela. Uma maneira para fazer isso é através de *filtros*, como vimos no capítulo sobre *DataSets*. Outra maneira é usando **Ranges**, um recurso exclusivo dos componentes *Table*.

Os *Ranges* definem um valor inicial e um valor final para um ou mais campos de uma tabela. Para tabelas *Paradox* e *dBASE*, os campos usados devem ser *indexados*. Para tabelas SQL (*Oracle*, *SQL Server*, *Sybase*, etc.) pode-se usar quaisquer campos para os *Ranges*.

A maneira mais simples e direta para se definir um *Range* é usando o método *SetRange*. Este método tem como argumentos duas listas de valores: os valores *iniciais* e os valores *finais* do *Range*. Os valores devem ser listados na ordem dos campos da tabela e devem ser colocados entre colchetes ([]). Use a palavra-chave **nil** para omitir campos do início da tabela. Veja um exemplo do uso de *SetRange*, para uma tabela cujo primeiro campo é um código de quatro dígitos:

```
TabProdutos.SetRange(['0005'], ['0025']);
```


O comando restringe os registros da tabela a apenas aqueles com códigos *entre* "0005" e "0025".

Para cancelar um *Range* depois de usá-lo, use o método *CancelRange*, como em:

```
TabProdutos.CancelRange;
```

CAPÍTULO 13 - COMPONENTES

TField

Quando você associa um DataSet a uma tabela de um banco de dados e abre o DataSet, o Delphi cria automaticamente **campos dinâmicos**, um para cada campo da tabela. Os campos dinâmicos são criados na memória do computador, temporariamente, quando o DataSet é aberto e destruídos quando o DataSet é fechado. O tipo dos campos dinâmicos, as suas propriedades, e a ordem em que eles são exibidos dependem somente das tabelas às quais esses campos estão associados.

Para que se tenha mais controle sobre os campos de um DataSet, o Delphi oferece os componentes **TField**. Componentes TField podem estar associados a um campo de um DataSet, ou podem ser novos campos, derivados de consultas ou cálculos. Os componentes TField não podem ser adicionados diretamente a um formulário. Eles fazem *parte* de um DataSet.

Os campos definidos pelos componentes TField são também chamados de **campos persistentes**. Isso porque eles "persistem" durante a execução do aplicativo e não são destruídos cada vez que o DataSet é fechado (como acontece com os campos dinâmicos).

Com os componentes TField, você pode controlar quais campos são exibidos, o tipo e a formatação de cada campo e várias outras propriedades. Você pode também criar **campos calculados** e **campos lookup**. Esses dois tipos de campos são, talvez, a principal razão para a existência dos componentes TField.

13.1. Criando campos persistentes

Os componentes TField, ou campos persistentes, só podem ser criados *dentro de DataSets*. Na maioria das vezes, os campos persistentes são associados aos campos de uma tabela já existente, portanto o DataSet deve estar ligado a um banco de dados antes da criação dos campos.

Como os DataSets mais usados são os componentes *Table*, usaremos esse componentes como base para os exemplos do restante dessa seção. Os procedimentos apresentados a seguir, no entanto, valem também para outros tipos de DataSets, como o componente *Query*, por exemplo.

- **Para criar campos persistentes para um componente *Table*:**
 1. Clique duas vezes no componente *Table*, para exibir o *Fields Editor* (veja ilustração a seguir).



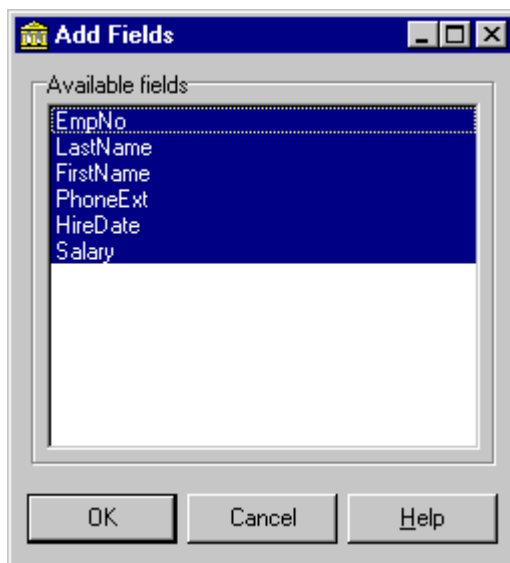
O Fields Editor

É a partir do *Fields Editor* que são feitas todas as operações com campos persistentes.

1. Clique com o botão direito dentro do *Fields Editor* e escolha o comando **Add Fields**.

Você pode também usar o comando **Add all Fields** para criar campos persistentes para todos os campos do DataSet, rapidamente (neste caso, o passo 3 não é necessário).

Uma janela com todos os campos da tabela é exibida:



2. Todos os campos são inicialmente selecionados. Clique em um campo para selecionar somente ele e use CTRL ou SHIFT para selecionar vários campos. Depois clique em OK para criar um campo persistente para cada campo selecionado.

O *Fields Editor* é exibido novamente, agora com os novos campos criados:



Os campos criados dessa maneira substituem os campos dinâmicos criados pelo Delphi. Não pode haver campos dinâmicos e persistentes em um mesmo DataSet. Se, por exemplo, você criar apenas alguns campos persistentes (e não todos os disponíveis), somente esses campos serão mostrados nos componentes ligados ao DataSet.

A ordem em que os campos exibidos nos componentes ligados ao DataSet é a ordem em que eles são listados no *Fields Editor*. Essa ordem pode ser alterada arrastando os nomes dos campos dentro do *Fields Editor*.

Pode-se também *apagar campos* usando DELETE, ou adicionar novos campos usando o comando **Add Fields** novamente.

13.2. Tipos de campos persistentes

Quando você cria campos persistentes para um DataSet, o Delphi adiciona, ao código da Unit que contém o DataSet, declarações para cada componente TField associado. Veja as declarações geradas para o exemplo anterior (destacadas em negrito).

```
unit Unit1;  
  
...  
  
type  
  
TForm1 = class(TForm)  
  
    TabEmp: TTable;  
  
    TabEmpEmpNo: TIntegerField;  
  
    TabEmpLastName: TStringField;  
  
    TabEmpFirstName: TStringField;  
  
    TabEmpPhoneExt: TStringField;  
  
    TabEmpHireDate: TDateTimeField;
```

```
TabEmpSalary: TFloatField;
```

```
end;
```

```
...
```

TIntegerField, *TStringField*, *TDateTimeField* e *TFloatField* são tipos de componentes *TField*. Esses tipos são definidos automaticamente pelo Delphi, dependendo dos campos associados no banco de dados. Por exemplo, um campo do tipo "Integer" em uma tabela Paradox, gera um componente do tipo *TIntegerField*; um campo do tipo "Alpha" gera um componente do tipo *TStringField*, e assim por diante. Veja uma lista dos tipos mais importantes de componentes *TField*:

Tipo de TField	Valores que podem ser armazenados
<i>TBooleanField</i>	Valores booleanos (<i>True</i> ou <i>False</i>)
<i>TBlobField</i>	Dados binários (figuras, por exemplo).
<i>TCurrencyField</i>	Números reais. Compatível com tipo <i>Currency</i> .
<i>TDateField</i>	Datas. Compatível com tipo <i>TDate</i> .
<i>TDateTimeField</i>	Datas e horas. Compatível com o tipo <i>TDateTime</i>
<i>TFloatField</i>	Números reais. Compatível com tipos <i>Float</i> , <i>Real</i> e <i>Double</i> .
<i>TIntegerField</i>	Números inteiros. Compatível com tipo <i>Integer</i> .
<i>TMemoField</i>	Textos longos. Compatível com tipo <i>String</i> .
<i>TStringField</i>	Strings pequenos (limitados a 8192 bytes).
<i>TTimeField</i>	Horas. Compatível com tipo <i>TTime</i> .

Os componentes *TField* (de vários tipos) podem ser manipulados diretamente no código. Eles possuem propriedades e eventos como os outros componentes. A diferença é que eles são componentes internos, que não aparecem diretamente nos formulários. Isso algumas vezes assusta o programador iniciante, que está acostumado a trabalhar *visualmente* com os componentes. Há vários outros componentes desse tipo no Delphi, mas eles são apenas usados para programação avançada.

13.3. Campos calculados

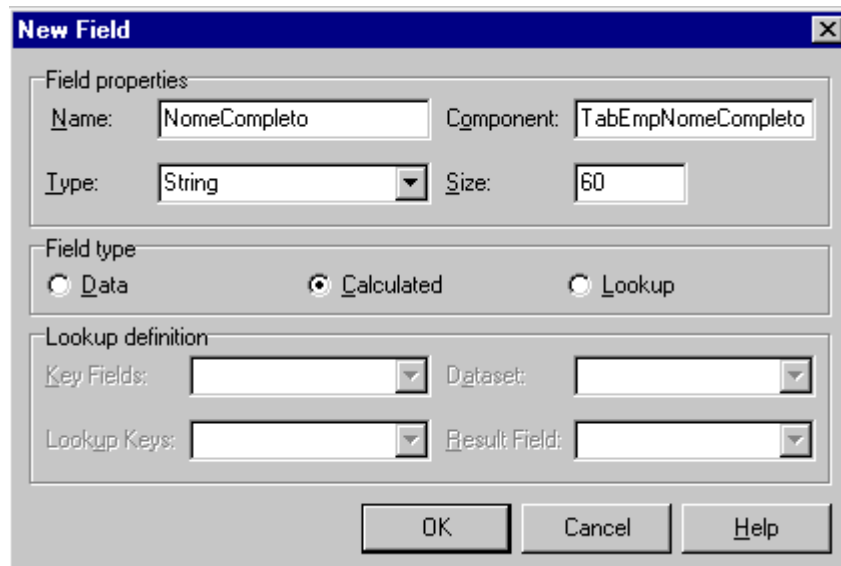
Os campos calculados exibem valores que são calculados durante a execução do aplicativo. Os cálculos são geralmente baseados em valores de outros campos, mas podem também ser completamente independentes (o que é raro).

- **Para criar um campo calculado para um DataSet:**
 1. Abra o *Fields Editor* para o DataSet. (Clique duas vezes no DataSet.)
 1. Dentro do *Fields Editor*, clique com o botão direito e escolha o comando **New Field**.

O Delphi exibe a caixa de diálogo **New Field**, onde se pode definir várias opções para o novo campo (veja ilustração a seguir).

2. Digite um nome para o campo na área "Name". Este é o nome pelo qual o componente será identificado no *Fields Editor*.

Enquanto o nome é digitado, o nome na área "Component" é atualizado automaticamente. Este nome é, por padrão, o nome do DataSet que contém o campo, seguido pelo nome definido na área "Name".



3. Defina um tipo para o campo. O tipo deve ser compatível com os valores usados para o cálculo do campo. Depois clique em OK para criar o campo.

O Delphi adiciona o nome do campo à lista de campos no Editor de Código e acrescenta uma linha à Unit do formulário, declarando um componente para o novo campo. O exemplo de código a seguir mostra (em negrito) a linha acrescentada. Note que o nome do componente é o nome que o Delphi define em "Component", na hora da criação do campo.

...

type

```
TForm1 = class (TForm)

TabEmp: TTable;

TabEmpEmpNo: TIntegerField;

TabEmpLastName: TStringField;

TabEmpFirstName: TStringField;

TabEmpPhoneExt: TStringField;

TabEmpHireDate: TDateTimeField;

TabEmpSalary: TFloatField;
```

```
TabEmpNomeCompleto: TStringField;
```

```
end;
```

```
...
```

Assim que é criado, um campo calculado contém apenas valores nulos. Para realizar os cálculos necessários para o campo, você deve digitar código para o evento **OnCalcFields** do DataSet que contém o campo. Esse código deve calcular um valor para o campo.

Veja um exemplo que "calcula" o nome completo de cada funcionário (concatenando o primeiro e último nomes).

```
procedure TForm1.TabEmpCalcFields(DataSet: TDataSet);  
  
begin  
  
    DataSet['NomeCompleto'] := DataSet['FirstName'] +  
  
    DataSet['LastName'];  
  
end;
```

Você pode definir vários campos calculados para um DataSet. O valor de todos os campos calculados deve ser calculado sempre dentro do código para evento *OnCalcFields* do DataSet.

No exemplo a seguir, é o realizado o cálculo de dois campos calculados e uma tabela de funcionários (chamada "TabFunc"): um para o salário anual e outro para o endereço completo.

```
procedure TForm1.TabFuncCalcFields(DataSet: TDataSet);  
  
begin  
  
    DataSet['SalarioAnual'] := DataSet['SalarioMensal'] * 12;  
  
    DataSet['EndCompleto'] := DataSet['Rua'] + ', ' +  
  
    DataSet['Numero'] + ', ' +  
  
    DataSet['Bairro'];  
  
end;
```

13.4. Campos lookup

Os campos lookup são campos especiais, usados para localizar automaticamente valores em um segundo DataSet, baseando-se em "valores-chave" de um primeiro DataSet.

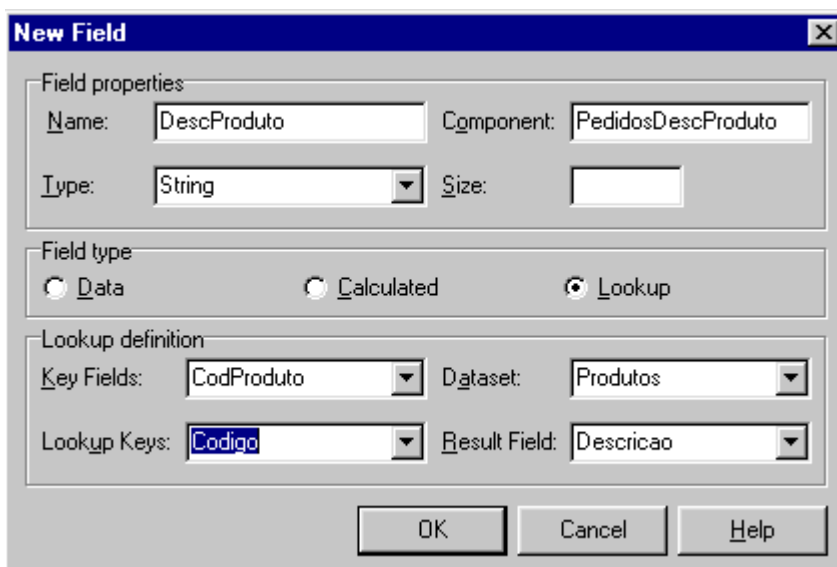
Por exemplo, suponha que o seu aplicativo use uma tabela **Produtos** com campos para o *código* e a *descrição* de cada produto, e outra tabela **Pedidos**, com campos para o *código do produto*, e a *quantidade* e a *data do pedido*. Nesse caso, você poderia definir um campo lookup na tabela Pedidos para mostrar a *descrição* do produto, baseando-se no *código do produto*. O campo lookup buscaria a descrição na tabela Produtos e a mostraria na tabela Pedidos (o *código do produto* poderia até ser escondido).

- **Para criar um campo lookup em um DataSet:**

1. Clique duas vezes no DataSet para exibir o *Fields Editor*.
1. Dentro do *Fields Editor*, clique com o botão direito e escolha o comando **New Field**:
2. Digite um nome para o novo campo em "Name" e defina um tipo para o campo em "Type".
3. Para "Field Type", escolha "Lookup"
4. Em "Lookup definition", defina valores para as opções descritas a seguir:

Opção	Descrição
<i>Dataset</i>	O DataSet de onde serão buscados os valores para o campo lookup.
<i>Key Fields</i>	Os campos chave que serão usados como base para realizar a busca. Estes são campos do DataSet que contém o campo lookup. Geralmente é usado apenas um campo aqui. Escolha um campo da lista ou digite diretamente o nome do campo. Para usar mais de um campo chave, separe os nomes dos campos com ponto-e-vírgula.
<i>Lookup Keys</i>	Os campos do DataSet que serão comparados com os campos especificados em <i>Key Fields</i> . Estes campos pertencem ao DataSet que contém os valores a serem buscados pelo campo lookup.
<i>Result Field</i>	O campo a ser retornado pela busca. O valor desse campo é o valor exibido no campo lookup.

A ilustração abaixo mostra a definição de um campo lookup que exibe a descrição de um produto, baseando-se no seu código. O campo lookup "casa" o campo *CodProduto* da tabela **Pedidos** com o campo *Codigo* da tabela **Produtos**, retornando a descrição do produto.



13.5. Propriedades dos componentes *TField*

Uma das maiores vantagens dos componentes *TField* (campos persistentes) sobre os campos dinâmicos (criados automaticamente pelo Delphi) é o fato de que você pode alterar as *propriedades* dos campos *TField*. Veja as propriedades mais úteis dos campos *TField* na tabela a seguir:

Propriedade	Descrição
<i>Alignment</i>	Determina o alinhamento do valor exibido no campo.
<i>DefaultExpression</i>	Defina aqui um valor padrão para o campo. O valor pode ser qualquer expressão válida em SQL, mas que não se refira a nomes de campos. O valor deve ser especificado entre aspas simples, a não ser que seja formado apenas por dígitos.
<i>DisplayLabel</i>	O título mostrado para o campo em componentes como o <i>DBGrid</i> .
<i>ReadOnly</i>	<p>Determina se o campo pode ou não ser alterado. Quando um campo é criado, <i>ReadOnly</i> é definido de acordo com o campo da tabela de banco de dados associado a ele.</p> <p>Você pode alterar <i>ReadOnly</i> para <i>True</i> para proteger um campo contra alterações, mesmo se o campo correspondente no banco de dados não estiver protegido.</p> <p>NOTA: Em um componente <i>DBGrid</i>, pressionar TAB pula os campos que têm a propriedade <i>ReadOnly</i> = <i>True</i>.</p>
<i>Required</i>	O valor de <i>Required</i> é definido automaticamente pelo Delphi quando um campo é criado. Essa propriedade determina se o campo pode ou não ser nulo. Faça <i>Required</i> = <i>True</i> para proibir valores nulos para um campo.
<i>Visible</i>	<p>Altere essa propriedade para <i>False</i> para esconder o campo em um <i>DBGrid</i>. O valor padrão é <i>True</i>.</p> <p>Essa propriedade é muito útil quando se usa campos calculados em uma tabela. Você pode esconder os campos que foram usados para os cálculos, mostrando apenas os campos com os resultados, por exemplo.</p>

Há várias outras propriedades importantes que não foram citadas aqui. Muitas delas são definidas automaticamente quando um campo é criado e raramente precisam ser alteradas. Outras são tratadas no curso avançado.

CAPÍTULO 14 - O COMPONENTE *BATCHMOVE*

O componente *BatchMove* é usado para realizar transferências de grandes quantidades de dados de uma tabela de banco de dados para outra. Com esse componente, pode-se copiar tabelas inteiras, ou criar novas tabelas com uma única linha de código (ou até em tempo de desenvolvimento).

Um dos usos mais comuns do componente *BatchMove* é para mover tabelas, ou partes de tabelas, de um banco de dados local, como Paradox, para um banco de dados SQL remoto (como Oracle, SQL Server, etc.) A conversão dos valores, tipos e estrutura das tabelas é realizada automaticamente pelo componente.

Outra utilidade do componente *BatchMove* é copiar dados entre uma tabela e outra de um mesmo banco de dados. Pode-se fazer isso, por exemplo, para criar uma tabela formada por registros de outra tabela que satisfaçam um condição especificada.

14.1. Configuração básica

Para realizar uma transferência de dados com o componente *BatchMove*, você deve definir quais são as tabelas de origem e de destino e o *mapeamento* que será feito entre os campos de cada tabela. Veja uma descrição das propriedades que precisam ser alteradas:

Propriedade	Descrição
<i>Destination</i>	O nome da tabela de <i>destino</i> , para onde serão transferidos os dados.
<i>Source</i>	O nome da tabela de <i>origem</i> , de onde serão extraídos os dados.
<i>Mappings</i>	<p>Define o <i>mapeamento</i> entre os campos da tabela de origem e de destino. Se as duas tabelas têm campos com nomes iguais, não é necessário alterar essa propriedade. Se os nomes forem diferentes, no entanto, você deve indicar aqui quais são os campos correspondentes em cada tabela.</p> <p>Clique duas vezes na área do valor de <i>Mappings</i> e digite uma linha para cada par de campos a serem "casados". Para campos com o mesmo nome, simplesmente digite o nome em comum. Para cada linha, use:</p> <p style="text-align: center;"><i>Campo da tabela de destino = Campo da tabela de origem</i></p> <p>Veja um exemplo:</p> <pre> Codigo Descricao Quant = Quantidade Valor = Preco </pre> <p>Aqui os campos <i>Codigo</i> e <i>Descricao</i> têm os mesmos nomes nas duas tabelas. Já o</p>

	campo <i>Quant</i> na tabela de destino corresponde ao campo <i>Quantidade</i> na tabela de origem. O mesmo acontece com os campos <i>Valor</i> e <i>Preço</i> .
<i>RecordCount</i>	O número de registros que serão transferidos de uma tabela para outra. Se <i>RecordCount</i> for zero, todos os registros são transferidos.

14.2. Modos de operação

A transferência de dados com um componente *BatchMove* pode ser realizada de várias maneiras, usando **modos** de operação diferentes. Pode-se criar acrescentar, copiar, atualizar, ou apagar dados. A propriedade **Mode** do componente *BatchMove* define qual a operação realizada. Veja os valores que essa propriedade pode assumir.

Valor de <i>Mode</i>	Efeito na transferência
<i>batAppend</i>	Adiciona os registros da tabela de origem no final de tabela de destino. A tabela de destino deve existir. (Esta é a opção padrão).
<i>batAppendUpdate</i>	Substitui os registros da tabela de destino pelos que "casam" com os registros da tabela de origem. O "casamento" é feito baseado nos índices (geralmente as chaves primárias) das tabelas. Registros que não casam são adicionados ao final da tabela de destino. As duas tabelas envolvidas devem existir.
<i>batCopy</i>	Cria uma nova tabela com a mesma estrutura e dados da tabela de origem. Se já existir uma tabela com o mesmo nome, ela é substituída.
<i>batDelete</i>	Apaga os registros da tabela de destino que "casam" com os da tabela de origem. As tabelas devem conter um índice para permitir o "casamento".
<i>batUpdate</i>	Substitui os registros da tabela de destino pelos que "casam" com os da tabela de origem. Registros que não casam <i>não são</i> transferidos (note a diferença de <i>batAppendUpdate</i>).

14.3. Executando a operação de transferência

A transferência de dados com um componente *BatchMove* pode ser realizada em tempo de desenvolvimento ou em tempo de execução.

Em *tempo de execução*, use o método *Execute* do componente *BatchMove*, como em:

```
BatchMove1.Execute;
```

Em *tempo de desenvolvimento*, faça o seguinte:

- Clique no componente *BatchMove* com o botão direito e escolha o comando *Execute*.

Veja um exemplo simples que usa um componente *BatchMove* para transferir todos os registros de uma tabela chamada *Produtos* para outra chamada *ProdutosNovos*.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
```

```
TabProdutos.Open;  
  
TabProdutosNovos.Open;  
  
with BatchMove1 do  
  
begin  
  
Source := TabProdutos;  
  
Destination := TabProdutosNovos;  
  
Mode := batAppendUpdate;  
  
Execute;  
  
end;  
  
TabProdutos.Close;  
  
TabProdutosNovos.Close;  
  
end;
```

Várias propriedades do componente *BatchMove* são definidas diretamente no exemplo, mas lembre-se que todas essas propriedades poderiam também ser definidas em tempo de desenvolvimento (veja a tabela de propriedades na página anterior).

14.4. Lidando com erros na transferência

Uma operação de transferência de dados entre tabelas pode causar muitos erros, principalmente quando a quantidade de dados a ser transferida é grande, ou quando há muitas diferenças entre as tabelas de origem e de destino. Erros que podem ocorrer são violações de chave (quando um registro adicionado contém a mesma chave que um registro existente) e erros de conversão (os tipos dos campos não são compatíveis).

O componente *BatchMove* pode criar novas tabelas durante a transferência, para permitir um controle maior sobre os registros que foram transferidos e sobre os registros que causaram problemas. Três tabelas (do tipo *Paradox*) podem ser criadas. As três tabelas contêm (respectivamente):

- Registros alterados durante a transferência.
- Registros que não puderam ser transferidos, devido a violações de chave ou restrições de integridade.
- Registros que tiveram campos cortados, devido à conversão dos tipos de cada campo, ou a campos com do mesmo tipo, mas com tamanhos diferentes.

A criação e a manipulação dessas tabelas é controlada pelas seguintes propriedades do componente *BatchMove*:

Propriedade	Descrição
<i>ChangedTableName</i>	O nome da tabela que conterá todos os registros alterados na transferência. A tabela é criada automaticamente pelo Delphi. Deixe essa propriedade vazia para não criar essa tabela.

<i>ChangedCount</i>	(Acessível somente através de programação) Retorna o número de registros na tabela especificada em <i>ChangedTableName</i> .
<i>KeyViolTableName</i>	O nome da tabela onde serão armazenados todos os registros que não puderam ser transferidos por causa de violações de chave ou de integridade . A tabela é criada automaticamente pelo Delphi. Deixe essa propriedade vazia, para evitar que a tabela seja criada
<i>KeyViolCount</i>	(Acessível somente através de programação). Retorna o número de registros na tabela especificada em <i>KeyViolTableName</i> .
<i>ProblemTableName</i>	O nome da tabela onde serão armazenados todos os registros que tiveram campos cortados na transferência. Um campo é cortado, quando o campo de destino é menor que o campo de origem (o valor original não cabe). A tabela é criada automaticamente pelo Delphi. Deixe essa propriedade vazia para não criar essa tabela.
<i>ProblemCount</i>	(Acessível somente através de programação) Retorna o número de registros na tabela especificada em <i>ProblemTableName</i> .

CAPÍTULO 15 - COMPONENTES *DATA CONTROLS*

Os componentes da página *Data Controls*, da paleta de componentes do Delphi, permitem acessar e alterar dados em um banco de dados diretamente. Já vimos uma introdução breve sobre esses componentes e já usamos um deles nos exemplos anteriores: o componente *DBGrid*. Neste capítulo, veremos detalhes sobre como funcionam e como usar os mais importantes componentes *Data Controls*.

Todos os componentes *Data Controls* têm a capacidade de exibir e alterar dados de um banco de dados. Muitas das tarefas necessárias para a leitura e a manipulação de dados são realizadas automaticamente por esses componentes. Algumas vezes, um aplicativo inteiro com acesso a bancos de dados pode ser desenvolvido usando componentes *Data Controls*, sem a necessidade de manipular dados diretamente com programação.

15.1. Propriedades *DataSource* e *DataField*

Há dois tipos básicos de componentes *Data Controls*: os que acessam *campos* de uma *DataSet*, como os componentes *DBEdit* e *DBText*, e os que acessam *registros inteiros* de um *DataSet*, como o componente *DBGrid*.

Os componentes que acessam os dados de um *DataSet* campo a campo têm duas propriedades importantes em comum: *DataSource* e *DataField*.

A propriedade ***DataSource*** determina o *DataSource* ao qual o componente está conectado. Este *DataSource* deve estar conectado a um componente *Table* ou *Query* (um *DataSet*), para que possa ter acesso ao banco.

A propriedade ***DataField*** indica o *campo* ao qual o componente está associado. Este é um dos campos do *DataSet* conectado ao *DataSource*.

Os componentes *DBGrid* e *DBNavigator* acessam dados registro por registro, e não campo por campo, como os outros componentes *Data Controls*. Portanto, esses componentes *não* apresentam a propriedade *DataField* (eles não estão associados a um campo específico).

15.2. Outras propriedades e recursos comuns

A maioria dos componentes *Data Controls* têm componentes correspondentes na página *Standard* da paleta de componentes. Por exemplo, o componente *DBEdit* corresponde ao componente *Edit*, e o componente *DBText* corresponde ao componente *Label*.

Na verdade, vários dos componentes *Data Controls* são apenas versões com acesso a bancos de dados dos componentes da página *Standard*. Por isso, a maioria das propriedades desses componentes são as mesmas, como aquelas que determinam a cor, o alinhamento, as dimensões, etc. Muitos eventos e métodos também são os mesmos.

Nas seções a seguir, veremos apenas as propriedades e eventos relevantes para o trabalho com bancos de dados (além de algumas outras propriedades essenciais). Veja os capítulos sobre os componentes comuns para mais detalhes sobre as outras propriedades e eventos.

15.3. Componente *DBEdit*

O componente *DBEdit* é a versão com acesso a banco de dados do componente *Edit*. Você pode usá-lo para exibir ou alterar o valor de um campo de um banco de dados diretamente. Veja algumas propriedades importantes desse componente.

Propriedade	Descrição
<i>Text</i>	(Acessível somente através de programação). Como o valor exibido nesse componente vem de um campo, não é possível alterar a propriedade <i>Text</i> em tempo de desenvolvimento, para definir um valor inicial a ser exibido. A propriedade <i>Text</i> , entretanto, pode ser lida e alterada em tempo de execução. Quando o valor de <i>Text</i> é alterado, o campo associado no banco de dados é alterado também.
<i>ReadOnly</i>	Determina se o valor exibido pode ou não ser alterado pelo usuário. Altere <i>ReadOnly</i> para <i>True</i> para não permitir alterações.
<i>MaxLength</i>	O número máximo de caracteres que pode ser digitado dentro do <i>DBEdit</i> . Use essa propriedade para trabalhar com campos de tamanho fixo, como os usados para códigos, por exemplo.

15.4. Componente *DBText*

O componente *DBText* corresponde ao componente *Label*. Esse componente é usado para exibir valores que não devem (nem podem) ser alterados. Use este componente para indicar que os campos são apenas para exibição.

Não há como alterar o que é exibido no componente *DBText* diretamente, nem com programação. (O componente não apresenta a propriedade *Caption*). O que é exibido no componente depende exclusivamente do campo associado no banco de dados, definido pelas propriedades *DataSource* e *DataField*.

15.5. Componente *DBMemo*

Este componente é semelhante ao componente *Memo*. Ele é usado para exibir trechos longos de texto, como os armazenados em campos do tipo "MEMO" dos bancos de dados. Há algumas novas propriedades interessantes:

Propriedade	Descrição
<i>AutoDisplay</i>	Define se o texto do campo associado é exibido imediatamente ou não. Se <i>AutoDisplay</i> for <i>False</i> , o texto só será exibido quando o usuário clicar duas vezes no componente. Isso torna o aplicativo mais rápido, especialmente quando os textos a serem exibidos são muito extensos. O valor padrão é <i>True</i> , que faz com o que texto seja exibido automaticamente, quando o usuário passa de um registro para outro.
<i>ReadOnly</i>	Determina se o texto exibido no componente pode ser alterado ou não pelo usuário.

15.6. Componente DBCheckBox

O componente *DBCheckBox* é uma versão especial, com acesso a bancos de dados, do componente *CheckBox*. Esse componente é geralmente ligado a um campo do tipo booleano, mas também pode ser ligado a outros tipos de campos com dois valores possíveis, como "Sim/Não", "Ligado/Desligado", etc. Os valores associados aos estados do *DBCheckBox* são definidos pelas propriedades *ValueChecked* e *ValueUnchecked*:

Propriedade	Descrição
<i>ValueChecked</i>	<p>Os valores que correspondem ao <i>DBCheckBox</i> quando ele está marcado. Pode-se especificar um único valor, ou uma lista de valores separados por ponto-e-vírgulas, como "Sim; Verdadeiro; True".</p> <p>Se o valor do campo associado for um dos valores especificados na lista, o componente <i>DBCheckBox</i> aparece marcado.</p> <p>Se o campo associado for um campo booleano, um valor <i>True</i> marca, e um valor <i>False</i> desmarca o <i>DBCheckBox</i>. Isso acontece mesmo se os valores <i>True</i> e <i>False</i> não forem especificados nas propriedades <i>ValueChecked</i> e <i>ValueUnchecked</i>.</p>
<i>ValueUnchecked</i>	<p>Os valores que correspondem ao <i>DBCheckBox</i> quando ele está desmarcado. Como para a propriedade anterior, pode-se especificar um único valor, ou uma lista de valores, como por exemplo: "Não; False; Falso".</p>

NOTA: se o valor do campo associado ao *DBCheckBox* não for nem *True*, nem *False*, nem um valor das propriedades *ValueChecked* ou *ValueUnchecked*, o componente aparece acinzentado (nem marcado, nem desmarcado).

15.7. Componente DBRadioGroup

Esse componente é semelhante ao componente *RadioGroup*. Ele é usado para listar opções que são mutuamente exclusivas (somente uma pode ser escolhida). Uma característica interessante desse componente é que o título exibido ao lado dos *RadioButtons* não precisa ser o mesmo valor que é lido ou armazenado no campo associado. A propriedade *Items* define os títulos que são exibidos e *Values* define os valores associados a cada título.

Propriedade	Descrição
<i>Caption</i>	O texto que aparece na parte de cima do <i>DBRadioGroup</i> . Usado para identificar o grupo de opções.
<i>Items</i>	Os itens exibidos ao lado de cada <i>RadioButton</i> . Clique duas vezes ao lado dessa propriedade para definir a lista de itens – digite um item para cada linha.
<i>ItemIndex</i>	(Apenas disponível através de programação). Retorna o índice do <i>RadioButton</i> que está selecionado no momento. O índice é um número inteiro. Zero corresponde ao primeiro <i>RadioButton</i> .
<i>Values</i>	Os valores associados aos itens exibidos. Os valores especificados são os valores que são lidos ou escritos no campo associado. Clique duas vezes ao lado da propriedade para definir a lista de valores. A ordem dos valores determina a associação com os itens especificados na propriedade <i>Items</i> .

	Se <i>Values</i> for deixada vazia vazia, os valores lidos e armazenados são os especificados na propriedade <i>Items</i> (os mesmos que aparecem na tela).
<i>Value</i>	(<i>Apenas disponível através de programação</i>). O valor correspondente ao <i>RadioButton</i> que está selecionado.

15.8. Componente *DBImage*

Este componente é usado para exibir imagens armazenadas em campos do do tipo BLOB (*Binary Large Object*). Veja as propriedades mais importantes do componente *DBImage*:

Propriedade	Descrição
<i>AutoDisplay</i>	Se <i>AutoDisplay</i> for <i>True</i> (o valor padrão), a imagem é atualizada automaticamente quando o campo associado é alterado. Se <i>AutoDisplay</i> for <i>False</i> , a imagem só é atualizada (recarregada) quando o usuário clicar duas vezes no componente <i>DBImage</i> .
<i>BorderStyle</i>	<i>BorderStyle</i> determina se é exibida uma linha em volta da imagem. O valor <i>bsNone</i> não mostra um linha; <i>bsSingle</i> exibe uma linha fina.
<i>Center</i>	Se <i>Center</i> for <i>True</i> (o padrão), a imagem é exibida <i>centralizada</i> no componente. Se <i>Center</i> for <i>False</i> , a imagem é exibida no canto esquerdo superior do componente.
<i>QuickDraw</i>	Altere <i>QuickDraw</i> para <i>True</i> para que imagens com 256 cores ou mais sejam exibidas mais rapidamente, mas com perda de qualidade. Altere <i>QuickDraw</i> para <i>False</i> para obter uma maior qualidade de exibição, mas com perda de velocidade.
<i>Stretch</i>	Determina se a imagem será ou não "esticada" para preencher todo o espaço do componente <i>DBImage</i> . Se <i>Stretch</i> for <i>True</i> , a imagem é esticada (o que geralmente causa distorções e perda de qualidade); se <i>Stretch</i> for <i>False</i> , o tamanho da imagem não é alterado.

15.9. Componentes *DBListBox* e *DComboBox*

Os componentes *DBListBox* e *DBComboBox* são as versões com acesso a bancos de dados dos componente *ListBox* e *ComboBox*, respectivamente.

Os valores listados nesses componentes devem ser adicionados diretamente, alterando a propriedade *Items*. Os valores não são trazidos do campo associado no banco de dados (isso não seria prático para tabelas com centenas ou milhares de valores diferentes, por exemplo).

Propriedade	Descrição
<i>Items</i>	<i>Items</i> determina os itens exibidos. Esses itens podem ser adicionados manualmente, em tempo de desenvolvimento, ou em tempo de execução. A propriedade <i>Items</i> é do tipo <i>TStrings</i> . Isso significa que você pode usar os métodos <i>Add</i> e <i>Append</i> para adicionar itens, e <i>Delete</i> para apagar itens (veja detalhes sobre o tipo <i>TStrings</i> na seção sobre o componente <i>Memo</i> , no capítulo "Componentes Visuais Comuns").
<i>Style</i>	(Somente para o componente <i>DBComboBox</i>). Determina o

comportamento e a aparência do componente <i>DBComboBox</i> . As opções são idênticas às do componente <i>ComboBox</i> – elas são descritas em detalhe na seção sobre este componente.
--

15.10. Componentes *DBLookupList* e *DBLookupCombo*

Estes dois componentes são semelhantes, na sua aparência, aos componentes *DBListBox* e *DBComboBox*, mas oferecem vários recursos a mais. Os dois componentes trabalham com dois *DataSets* e não um como acontece com a maioria dos outros componentes *Data Controls*.

O comportamento dos componentes *DBLookupList* e *DBLookupCombo* é parecido com o dos campos lookup, que vimos no capítulo sobre componentes *TField*. Eles "casam" dois campos especificados em *DataSets* diferentes e exibem um valor resultante.

Um componente *DBLookupList* ou *DBLookupCombo* pode ser ligado a um campo lookup. Nesse caso, basta definir as propriedades *DataSource* e *DataField* apropriadamente, para associar o campo lookup ao componente. O componente "reconhece" que o campo é o do tipo lookup e realiza as buscas automaticamente, exibindo os resultados.

Os componentes *DBLookupList* e *DBLookupCombo*, no entanto, são geralmente ligados a campos comuns. Eles têm a capacidade de fazer a busca dos valores automaticamente, sem a necessidade de usar campos lookup predefinidos. Para isso, é necessário ligá-los a campos de dois *DataSets* diferentes.

- **Para configurar componentes *DBLookupList* ou *DBLookupCombo*:**

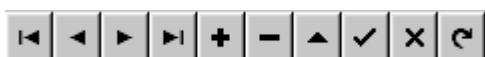
1. Na propriedade *DataSource*, escolha o *DataSource* a ser diretamente ligado ao componente. Os dados desse *DataSource* são os dados que serão *alterados* pelo componente. Na propriedade *DataField*, escolha o campo que será alterado.
1. Na propriedade *ListSource* defina o *DataSet* de onde serão *lidos* os valores, de acordo com o valor do campo especificado em *DataField*. Na propriedade *ListField* defina o campo de onde os valores serão lidos.

Os valores lidos a partir do campo definido em *ListField* são os valores exibidos no componente.

2. Finalmente, altere a propriedade *KeyField* para o campo que será comparado com o campo definido em *DataField*.







Isso termina a configuração. Quando os *DataSets* são ativados, o valor do campo *DataField* (do primeiro *DataSet*) é lido e comparado com os valores no campo *KeyField* (do segundo *DataSet*). O valor retornado é o valor do campo *ListField* (do segundo *DataSet*).

15.11. Componente *DBNavigator*

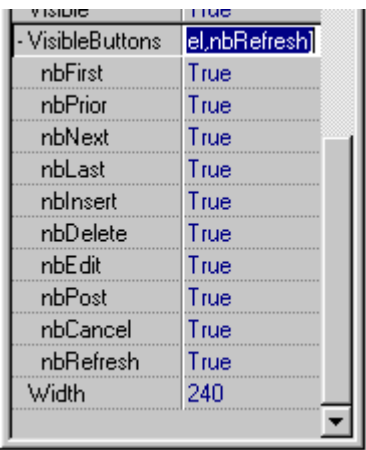


O componente *DBNavigator* permite realizar uma série de operações comuns em registros de um *DataSet*, como navegação, inserção e deleção. Um componente *DBNavigator* é ligado a um *DataSet* através de um componente *DataSource* (como os outros componentes *Data Controls*), mas o acesso é feito *registro por registro* e não campo por campo. É comum usar um componente *DBNavigator* em associação com um componente *DBGrid* para a navegação dos dados em um *DataSet*.

O componente *DBNavigator* é composto de um conjunto de botões. Cada botão executa um dos métodos do *DataSet*, como *First*, *Last*, *Prior*, *Next*, *Post*, *Edit*, etc. Tudo que pode ser feito com *DBNavigator* pode ser feito através de programação, usando esses métodos diretamente. O *DBNavigator* é apenas uma maneira simples e rápida de oferecer recursos para a navegação e alteração de um *DataSet* para o usuário, sem a necessidade de programação. Veja a seguir os métodos associados a cada botão do *DBNavigator*:

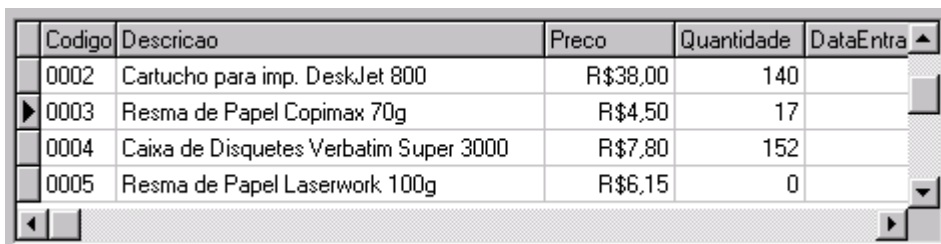
 <i>First</i>	 <i>Delete</i>	 <i>Prior</i>	 <i>Edit</i>	 <i>Next</i>
 <i>Post</i>	 <i>Last</i>	 <i>Cancel</i>	 <i>Insert</i>	 <i>Refresh</i>

Há várias propriedades importantes para o componente *DBNavigator*. Com elas, você pode escolher quais os botões são exibidos no componente (*VisibleButtons*), e controlar a aparência de "Dicas" para cada botão (*Hints* e *ShowHint*).

Propriedade	Descrição
<i>ConfirmDelete</i>	Se <i>ConfirmDelete</i> for <i>True</i> , uma caixa de confirmação é exibida quando o botão "Delete" é clicado. Caso contrário, o registro atual é apagado sem confirmações.
<i>DataSource</i>	Especifique aqui o <i>DataSource</i> ao qual o <i>DBNavigator</i> está ligado.
<i>Flat</i>	Determina se os botões do <i>DBNavigator</i> são exibidos com efeito tridimensional, ou não. Se <i>Flat</i> for <i>True</i> , o efeito tridimensional não é exibido e o componente se torna semelhante às barras de ferramentas do <i>Office 97</i> .
<i>Hints</i> , <i>ShowHint</i>	<p>Altere <i>Hint</i> para definir as "dicas" que aparecem para cada botão no <i>DBNavigator</i>. Clique duas vezes ao lado da propriedade e digite uma linha de texto para cada dica. As dicas só serão mostradas se a propriedade <i>ShowHint</i> for <i>True</i>.</p> <p>Se <i>Hints</i> for deixada vazia e <i>ShowHint</i> for <i>True</i>, as dicas exibidas são os próprios nomes dos botões (em inglês).</p>
<i>VisibleButtons</i>	<p>Especifique aqui quais botões do <i>DBNavigator</i> você deseja exibir. Esta propriedade é muito importante, pois na maioria das vezes o <i>DBNavigator</i> oferece mais botões do que é necessário.</p> <p>Para definir os botões a serem exibidos, clique duas vezes no nome da propriedade para abrir uma lista de subpropriedades.</p>
	 <p>É mostrada uma subpropriedade para cada botão que pode ser exibido (veja a figura ao lado). Para esconder um botão, altere a propriedade correspondente para <i>False</i>. Para exibir um botão altere a propriedade correspondente para <i>True</i>.</p>

15.12. Componente DBGrid

O componente *DBGrid* é um dos componentes mais usados para trabalhar com os dados de um *DataSet*. Da mesma forma que para o componente *DBNavigator*, o componente *DBGrid* trabalha com dados *registro por registro*. Veja um exemplo de um *DBGrid* em ação:



Codigo	Descricao	Preço	Quantidade	DataEntrada
0002	Cartucho para imp. DeskJet 800	R\$38,00	140	
0003	Resma de Papel Copimax 70g	R\$4,50	17	
0004	Caixa de Disquetes Verbatim Super 3000	R\$7,80	152	
0005	Resma de Papel Laserwork 100g	R\$6,15	0	

Um *DBGrid* é formado por *linhas*, *colunas* e *células*. Cada célula contém um valor de um campo de um *DataSet*. Como padrão, o *DBGrid* mostra também o nome de cada coluna e indica o registro atual, através de um pequeno triângulo no lado esquerdo.

A propriedade mais usada no componente *DBGrid* é a propriedade *Options*, que contém várias opções que podem ser ligadas (valor *True*) ou desligadas (valor *False*). Veja a seguir o significado das opções mais importantes:

Opção	Efeito quando ligada
<i>dgEditing</i>	Permite que o usuário altere dados dentro do <i>DBGrid</i> .
<i>dgAlwaysShowEditor</i>	Faz com que o <i>DBGrid</i> esteja sempre pronto para realizar alterações (no estado <i>dsEdit</i>).
<i>dgTitles</i>	Faz com que títulos sejam exibidos no topo de cada coluna.
<i>dgIndicator</i>	Exibe o indicador do registro atual (um pequeno triângulo no lado esquerdo do <i>DBGrid</i>).
<i>dgColumnResize</i>	Permite que colunas sejam movidas ou redimensionadas.
<i>dgColLins</i>	Exibe linhas entre as colunas do <i>DBGrid</i> .
<i>dgRowLines</i>	Exibe linhas entre as linhas do <i>DBGrid</i> .
<i>dgTabs</i>	Permite que o usuário use TAB e SHIFT+TAB para passar de uma célula do <i>DBGrid</i> para outra.
<i>dgRowSelect</i>	Permite que o usuário selecione linhas inteiras no <i>DBGrid</i> .
<i>dgAlwaysShowSelection</i>	Faz com que a célula ou linha selecionada permaneça selecionada, mesmo quando o <i>DBGrid</i> não está com o foco.
<i>dgConfirmDelete</i>	Faz com que uma caixa de confirmação apareça, quando o usuário usa CTRL+DELETE para apagar um registro.
<i>dgCancelOnExit</i>	Faz com que um novo registro inserido no <i>DBGrid</i> não seja enviado para o banco de dados, a não ser que o registro tenha sido alterado (não esteja vazio).

dgMultiSelect

Permite que mais de uma linha no DBGrid possa ser selecionada ao mesmo tempo.

CAPÍTULO 16 - O COMPONENTE *QUERY*

Neste capítulo, veremos como usar o componente *Query* para realizar consultas SQL em bancos de dados. O componente *Query* permite realizar consultas SQL em qualquer tipo de banco de dados suportado pelo Delphi, desde em bancos locais Paradox e dBASE até bancos SQL, como Oracle ou SQL Server.

Um componente *Query* contém o texto de uma consulta SQL e a especificação dos parâmetros usados na consulta. O componente *Query* se encarrega da execução da consulta SQL e do processamento dos dados retornados pelo banco de dados, para a consulta.

16.1. Configurando um componente *Query*

O componente *Query*, como o componente *Table*, é um tipo de *DataSet*. Portanto, muitas das propriedades e métodos que vimos no capítulo sobre *DataSets* também se aplicam ao componente *Query*.

Veja os passos básicos necessários para usar e configurar um componente *Query* em um formulário:

1. Adicione um componente *Query* ao formulário.
1. Altere a propriedade *DatabaseName* para o *Alias* do banco de dados a ser utilizado.
2. Na propriedade *SQL*, especifique a consulta SQL que a ser executada pelo componente.
3. Adicione um componente *DataSource* ao formulário e altere a sua propriedade *DataSet* para o nome do componente *Query*.
4. Para exibir os resultados gerados pelo componente *Query*, adicione um componente *DBGrid*, ou outro componente com acesso a dados. Altere a propriedade *DataSource* desse componente para o nome do componente *DataSource* que você acabou de adicionar.
5. Para executar a consulta SQL, altere a propriedade *Active* do componente *Query* para *True*. Se tudo correr bem, os resultados da consulta SQL são exibidos imediatamente no *DBGrid* (ou outro componente com acesso a dados).

16.2. Especificando a consulta SQL a ser executada

A consulta SQL de um componente *Query* é armazenada na sua propriedade *SQL*. A propriedade *SQL* é uma lista de strings (do tipo *TStrings*), contendo as linhas da consulta SQL. Quando um componente *Query* é executado, todas as linhas da propriedade *SQL* são concatenadas em uma só e enviadas ao banco de dados. Os resultados da consulta SQL podem, então, ser acessados através do componente *Query*.

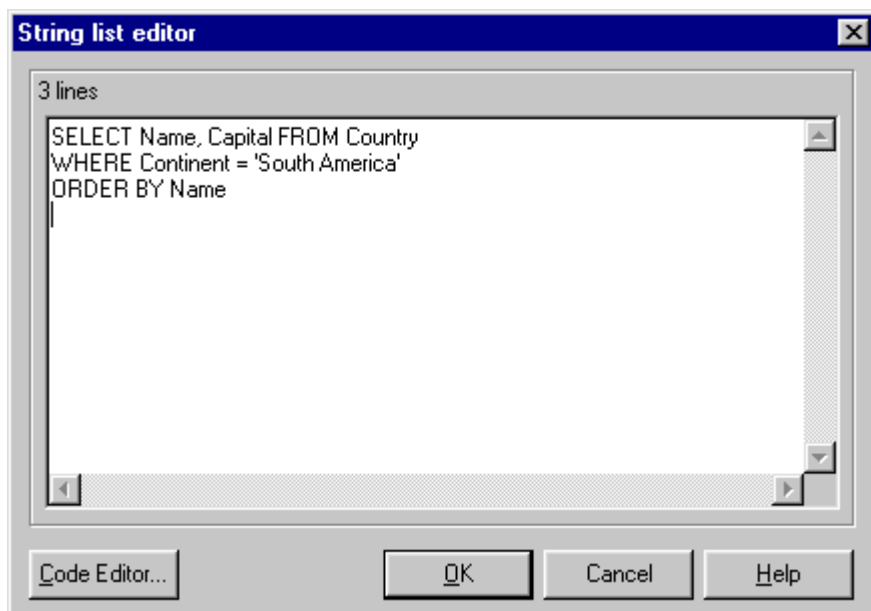
Pode-se especificar dois tipos de consultas SQL: **consultas estáticas** e **consultas dinâmicas**.

As **consultas estáticas** não podem ser alteradas em tempo de execução. Se os dados do banco de dados não forem alterados entre uma execução e outra da consulta, os dados retornados serão os mesmos.

As **consultas dinâmicas**, também chamadas de **consultas parametrizadas**, podem ser alteradas durante a execução do aplicativo. Consultas dinâmicas contêm **parâmetros**. Os parâmetros podem ser alterados diretamente no código, ou em tempo de desenvolvimento.

- **Para especificar uma consulta estática (sem parâmetros):**
- Em **tempo de desenvolvimento**, na propriedade *SQL* do componente *Query*, digite a consulta SQL a ser executada.

Para alterar o valor da propriedade *SQL*, clique duas vezes ao lado dessa propriedade no Object Inspector e digite o comando SQL na janela que aparece (veja a figura abaixo).

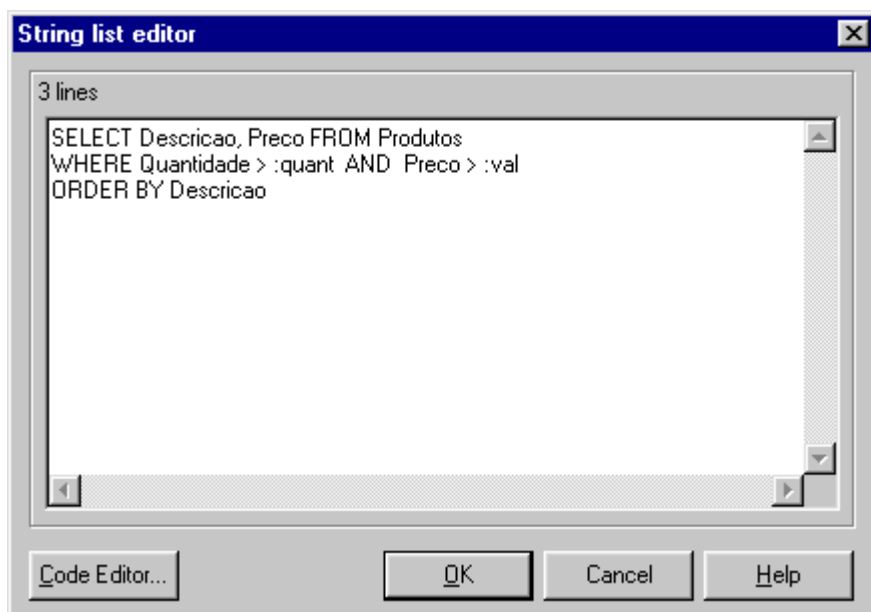


- Em **tempo de execução**, altere a propriedade *SQL* diretamente, usando os métodos *Add*, *Append*, *Clear*, etc. (a propriedade *SQL* é do tipo *TStrings*).

O exemplo a seguir lê o texto em um componente *Edit* e cria a consulta SQL no código, baseando-se no texto lido. *chr(39)* retorna o caractere "aspas simples" (39 é o código ASCII desse caractere). A consulta é executada alterando-se a propriedade *Active* do *Query* para *True*.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Continente: String;
begin
  Continente := Edit1.Text;
  Query1.SQL.Clear;
  Query1.SQL.Add('SELECT Name, Capital FROM Country ' +
    'WHERE Continent = ' + chr(39) + Continente + chr(39) +
    'ORDER BY Name');
  Query1.Active := True;
end;
```

- **Para especificar uma consulta dinâmica (parametrizada):**
- Faça o mesmo que para as consulta estáticas, mas em vez de especificar valores fixos na consulta, especifique *parâmetros*, identificadores antecidos por dois pontos (:). Veja um exemplo:

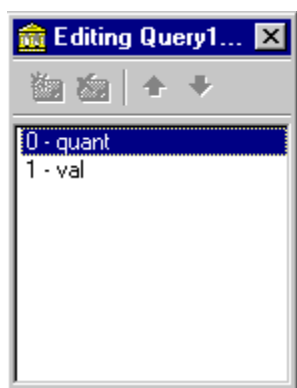


No exemplo, são definidos dois parâmetros: *quant* e *val*.

16.3. Trabalhando com parâmetros

Quando você define parâmetros na consulta SQL de um componente *Query*, o Delphi lê e registra automaticamente os parâmetros definidos. Antes de executar a consulta parametrizada, no entanto, você deve *configurar* os parâmetros, definindo o tipo e (possivelmente) um valor inicial para cada parâmetro.

- **Para configurar os parâmetros de um componente *Query*:**
 1. Selecione o componente *Query* e clique duas vezes ao lado da propriedade *Params*, no Object Inspector. Isso mostra o *Editor de Parâmetros* (figura abaixo)



O Editor de Parâmetros

2. Para cada parâmetro definido, clique no parâmetro e altere as propriedades *DataType* e *Value*, usando o Object Inspector. A propriedade *DataType* determina o tipo (*Real*, *Integer*, etc.) do parâmetro. *DataType* deve ser obrigatoriamente alterada.
3. Para confirmar as alterações, feche a janela do *Editor de Parâmetros*.

A propriedade *Value* de um parâmetro pode ser alterada para definir um valor inicial para um parâmetro, em tempo de desenvolvimento. Mas o mais comum é alterar os valores dos

parâmetros usando programação. A maneira mais simples é usando o método *ParamByName* do componente *Query*.

- **Para alterar os parâmetros de uma query em tempo de execução:**
- Use o método *ParamByName(nome do parâmetro)*. Use as propriedades de conversão *AsString*, *AsInteger*, *AsFloat*, etc. para converter valores na hora de atribuí-los aos parâmetros. Veja um exemplo:

```
procedure TForm1.Button1Click(Sender: TObject);  
  
begin  
  
with Query1 do  
  
begin  
  
Close;  
  
ParamByName('val').AsCurrency := StrToCurr(Edit1.Text);  
  
ParamByName('quant').AsInteger := StrToInt(Edit2.Text);  
  
Open;  
  
end;  
  
end;  
  
end;
```

O exemplo lê dois valores digitados em componentes *Edit* e atribui esses valores aos parâmetros *val* e *quant* (veja o exemplo da seção anterior). Em seguida a consulta SQL é executada usando o método *Open* do componente *Query*.

16.4. Executando consultas

Um componente *Query* deve ser "executado" para que sua consulta SQL seja executada. Quando você executa um componente *Query*, o Delphi interpreta o seu comando SQL, substitui os parâmetros pelos valores especificados e disponibiliza os dados resultantes da consulta.

Nos exemplos anteriores, usamos o método *Open* (ou a propriedade *Active*) do componente *Query* para executar consultas SQL. *Open*, no entanto, só pode ser usado para consultas com o comando *SELECT*.

As consultas SQL que não retornam dados, como as que usam os comandos *INSERT*, *UPDATE*, *DELETE*, etc. devem ser executadas usando o método *ExecSQL*, como em

```
Query1.ExecSQL;
```

CAPÍTULO 17 - A LINGUAGEM *LOCAL SQL*

A linguagem *Local SQL* é um subconjunto da linguagem SQL padrão (SQL 92 ANSI). SQL é uma linguagem universal usada na grande maioria dos bancos de dados relacionais.

A linguagem *Local SQL* contém os comandos mais básicos da linguagem SQL e é compatível também com os bancos locais, do tipo *Paradox* e *dBASE*, além dos bancos de dados SQL mais poderosos, como *Oracle* e *SQL Server*. A linguagem *Local SQL* é suficiente para realizar a maioria das consultas SQL necessárias em um banco de dados.

NOTA: como *Local SQL* é um subconjunto da linguagem SQL, usaremos apenas "SQL" de agora em diante.

17.1. As duas partes de SQL

A linguagem SQL se divide em duas partes independentes (chamadas também de linguagens): uma linguagem de **manipulação de dados**, usada para consultar e alterar os *dados* de bancos de dados, e uma linguagem de **definição de dados**, usada para alterar a *estrutura* dos bancos.

A linguagem de **manipulação de dados** é constituída pelos seguintes comandos principais:

Comando	Função
SELECT	Usado para recuperar dados de uma ou mais tabelas, baseando-se em condições especificadas. O comando SELECT é usado para realizar muitas operações diferentes e é muito poderoso.
INSERT	Usado para adicionar dados a uma tabela.
UPDATE	Usado para atualizar (modificar) dados existentes em uma tabela.
DELETE	Usado para apagar registros de uma tabela.

A linguagem de **definição de dados** é constituída pelos seguintes comandos principais:

Comando	Função
CREATE TABLE	Criar uma tabela.
ALTER TABLE	Alterar a estrutura de uma tabela.
DROP TABLE	Destruir uma tabela (removê-la do banco de dados).
CREATE INDEX	Criar um índice para uma tabela.
DROP INDEX	Remover um índice para uma tabela.

17.2. O comando SELECT

O comando SELECT é o mais poderoso e mais complexo dos comandos da linguagem SQL. Esse comando é usado para recuperar dados de uma ou mais tabelas. Os dados recuperados dependem das condições definidas no comando SELECT. A forma básica para o comando SELECT é a seguinte:

SELECT [*campos retornados*] **FROM** [*tabelas consultadas*]

WHERE [*condição*]

ORDER BY [*campos de ordenação*]

GROUP BY [*campos de agrupamento*]

Veja o que significa cada parte do comando:

Parte do comando SELECT	Descrição
<i>campos retornados</i>	Os nomes dos campos a serem retornados. Se um asterisco (*) for especificado, todos os campos das tabelas são retornados. Os campos devem pertencer a uma das tabelas especificadas em <i>tabelas consultadas</i> .
<i>tabelas consultadas</i>	As tabelas de onde serão extraídos os dados.
<i>condição</i>	Uma condição que restringe os valores que são retornados. A condição pode usar os operadores booleanos comuns de Object Pascal, como <, >, <>, =, AND, OR, NOT, entre outros.
<i>campos de ordenação</i>	Os campos usados como base para a ordenação dos valores retornados. O primeiro campo especificado tem prioridade. Os outros campos são usados como "critérios de desempate".
<i>campos de agrupamento</i>	Os campos usados para agrupar campos quando são usadas funções de agregação como COUNT, SUM e AVG.

Veja agora alguns exemplos do uso do comando SELECT:

Este comando...	Retorna...
SELECT * FROM Produtos	Todos os valores de todos os campos da tabela Produtos (a tabela inteira).
SELECT Nome, Preço FROM Produtos	Todos os nomes e os preços da tabela Produtos (mas nenhum outro campo).
SELECT Nome, Preço FROM Produtos WHERE Preço > 100.00	O Nome e o Preço de todos os produtos com Preço maior que 100.00.
SELECT Código, Preço FROM Produtos WHERE Quantidade > 12	O Código e o Preço de todos os produtos com Quantidade maior que 12. Note que os campos na parte WHERE não precisam estar na parte

	SELECT.
SELECT Nome, Quantidade FROM Produtos WHERE Preco > 100.00 AND Preco < 500.00	O Nome e a Quantidade de todos os produtos com Preço entre 100 e 500.
SELECT Nome FROM Produtos WHERE Quantidade <> 0 ORDER BY Nome	Somente o Nome dos produtos com Quantidade diferente de zero, <i>ordenados</i> pelo Nome do produto.
SELECT * FROM Pedidos, Produtos WHERE Pedidos.CodProduto = Produtos.Codigo	Todos os campos de ambas as tabelas Pedidos e Produtos, "casados" por código. (É feita uma <i>junção</i> das duas tabelas, baseada nos campos Codigo e CodProduto).
SELECT Nome, SUM(Preco*Quantidade) FROM Pedidos, Produtos WHERE Pedidos.CodProduto = Produtos.Codigo GROUP BY Nome, Codigo	O Nome e o valor total dos produtos (Preço vezes Quantidade), agrupados por código de produto. (Se houver um produto com o mesmo código em vários pedidos, o valor de todos os produtos pedidos é somado – o produto só aparece uma vez nos dados resultantes).

17.3. Usando IN e BETWEEN

O comando IN é usado em *condições* para determinar se um valor pertence a um conjunto especificado. O exemplo a seguir retorna o nome e o preço de todos os produtos com quantidades que estejam no conjunto {100, 200, 300, 400, 500}:

```
SELECT Nome Preco FROM Produtos
WHERE Quantidade IN (100, 200, 300, 400, 500)
```

O comando BETWEEN também é usado em *condições*, junto com a palavra AND, para determinar se um valor está dentro de um intervalo especificado. Veja dois exemplos:

```
SELECT Nome, Quantidade FROM Produtos
WHERE Preco BETWEEN 100 AND 1000
```

```
SELECT * FROM Pedidos
WHERE Codigo BETWEEN '0001' AND '0100'
```

17.4. Usando LIKE e caracteres "curinga"

Os comando LIKE é usado em condições, junto com os caracteres % e _ , para fazer "casamentos" parciais. O caractere % vale por um ou mais caracteres (como o * do DOS); o caractere _ vale por exatamente um caractere (semelhante ao ? do DOS).

Veja um exemplo:

```
SELECT * FROM Produtos
WHERE Nome LIKE 'Micro%'
```

Este exemplo retorna todos os produtos com o nome começando com "Micro".

17.5. Usando funções de agregação

As *funções de agregação* são usadas para realizar cálculos simples nos valores de um campo, como somas, médias e contagens. São cinco as funções de agregação:

Função	Retorna...
SUM	A soma de todos os valores numéricos em um campo.
AVG	A média de todos os valores não nulos em um campo.
MIN	O valor mínimo em um campo.
MAX	O valor máximo em um campo.
COUNT	O número de valores em um campo, ou o <i>número total de registros</i> retornados.

Veja alguns exemplos do uso de funções de agregação:

Comando SQL	Retorna
SELECT AVG(Preco) FROM Produtos	Retorna a média do preço de todos os produtos.
SELECT COUNT(*) FROM Produtos WHERE Preco < 100.00	Retorna o número de produtos com preço abaixo de 100.00. Um único valor é retornado.
SELECT SUM(Preco*Quantidade) FROM Produtos	Retorna a soma da multiplicação do Preço e da Quantidade de todos os produtos. Um único valor é retornado.

17.6. O comando INSERT

O comando INSERT é usado para inserir novos registros (com dados) em tabelas. Este comando é usado da seguinte forma:

INSERT INTO *Nome da tabela* (*Campo1, Campo2, ...*) **VALUES** (*Valor1, Valor2, ...*)

Alguns exemplos:

Comando	Efeito / Comentário
INSERT INTO Clientes (Nome, Sobrenome) VALUES ('Leonardo', 'Galvão')	Insere um novo registro na tabela Clientes com Nome = "Leonardo" e Sobrenome = "Galvão". Os outros campos do registro ficam vazios

	(nulos), se isso for permitido.
INSERT INTO Produtos VALUES ('0079','Arno 100',500,100,'12/10/97')	Insere um novo registro na tabela Produtos, preenchendo todos os valores do registro (Codigo, Nome, Valor, Quantidade e Data de entrada). Note que os campos não precisam ser especificados nesse caso.

17.7. O comando UPDATE

O comando UPDATE é usado para atualizar (modificar) registros em uma tabela. Esse comando é usado da seguinte forma:

UPDATE *Nome da Tabela* **SET** *Campo = Valor* **WHERE** *Condição*

Exemplos:

Comando	Efeito / Comentário
UPDATE Países SET Capital = 'Bratislava' WHERE País = 'Eslováquia'	Altera a capital do País "Eslováquia" para "Bratislava" (todas as ocorrências).
UPDATE Produtos SET Preço = Preço * 0.8	Reduz o preço de <i>todos</i> os produtos da tabela Produtos para 80% do preço anterior.

17.8. O comando DELETE

O comando DELETE é usado para apagar registros inteiros de uma tabela, que satisfaçam uma condição especificada.

DELETE FROM *Nome da tabela* **WHERE** *Condição*

O seguinte exemplo, apaga todos os registros da tabela Produtos, que têm o campo Quantidade = 0:

DELETE FROM Produtos WHERE Quantidade = 0

17.9. A linguagem de definição de dados

17.10. O comando CREATE TABLE

Usado para criar tabelas. Com esse comando, você especifica o nome da tabela a ser criada e o nome e o tipo de cada campo da nova tabela:

CREATE TABLE *Nome da Tabela* (

Campo1 TIPO1

Campo2 TIPO2

...

PRIMARY KEY (CampoChave)

)

A parte **PRIMARY KEY** define a chave primária da tabela. Veja um exemplo do uso do comando **CREATE TABLE**:

CREATE TABLE *Jogadores* (

Nome **CHAR**[40],

Sobrenome **CHAR**[60],

DataNasc **DATE**,

Clube: **CHAR**[40],

PRIMARY KEY (*Sobrenome*)

)

Para criar uma tabela do tipo Paradox ou dBASE adicione **.DB** ou **.DBF** ao final do nome da tabela e coloque o nome entre aspas, como no exemplo a seguir:

CREATE TABLE "*Clientes.db*" (

Codigo: **INTEGER**;

Nome **CHAR**[30],

Cidade **CHAR**[40],

)

17.11. O comando **ALTER TABLE**

O comando **ALTER TABLE** é usado para alterar a *estrutura* de uma tabela existente. Há duas versões para esse comando, uma para *adicionar* campos e outra para *removê-los*.

ALTER TABLE *Nome da tabela* **ADD** *NovoCampo1 Tipo1*, **ADD** *NovoCampo2 Tipo2*, ...

(Adiciona os campos especificados depois de **ADD**).

ALTER TABLE *Nome da tabela* **DROP** *Campo1*, **DROP** *Campo2*, ...

(Remove os campos especificados depois de **DROP**)

Veja alguns exemplos que usam o comando **ALTER TABLE**:

Comando	Efeito
ALTER TABLE Produtos ADD DataExpiracao DATE, ADD Fornecedor CHAR[60]	Adiciona dois campos (colunas) à tabela Produtos: "DataExpiracao" e "Fornecedor".
ALTER TABLE Jogadores DROP DataNasc	Remove o campo "DataNasc" da tabela <i>Jogadores</i> .

17.12. O comando DROP TABLE

O comando DROP TABLE é usado para remover tabelas inteiras de um banco de dados (A palavra "Drop" significa "Deixar cair", ou "Abandonar"). O uso desse comando é simples:

DROP TABLE *Nome da tabela*

17.13. O comando CREATE INDEX

O comando CREATE INDEX é usado para criar índices em tabelas. A estrutura do comando a é a seguinte:

CREATE INDEX *Nome do índice* **ON** *Nome da tabela* (*Campo1*, *Campo2*, ...)

Para tabelas dBASE essa é a única maneira de criar índices. Para tabelas Paradox esse comando só pode ser usado para a criação de índices secundários. O índice primário de uma tabela Paradox é o campo definido na parte "PRIMARY KEY" do comando CREATE TABLE.

17.14. O comando DROP INDEX

Usado para remover índices em uma tabela. Pode ser usado de duas formas:

DROP INDEX *Nome da tabela.Nome do índice*

(Remove da tabela o índice especificado)

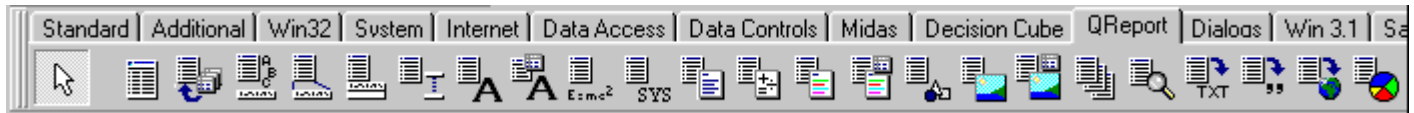
DROP INDEX *Nome da tabela.PRIMARY*

(Remove da tabela o índice primário - para tabelas Paradox)

CAPÍTULO 18 - COMPONENTES

QUICKREPORT

O Delphi oferece o conjunto de componentes QuickReport, para a criação de relatórios para impressão, baseados em bancos de dados. Os "componentes QuickReport", como os chamaremos, estão localizados na página *QReport* da paleta de componentes.



Os componentes QuickReport

Relatórios criados com os componentes QuickReport são baseados em **bandas**. Nas bandas podem ser adicionados **componentes imprimíveis**, como títulos de coluna, valores de campos de uma tabela, números de páginas e outros.

Há mais de dez tipos de bandas diferentes. Cada tipo de banda se comporta de forma distinta. Algumas bandas, por exemplo, se repetem em todas as páginas, outras são exibidas apenas uma vez no relatório. Há também bandas "inteligentes" que oferecem recursos para o agrupamento de dados, por exemplo.

Como a maioria dos componentes QuickReport são imprimíveis, são oferecidos vários recursos para a formatação detalhada e o posicionamento preciso dos componentes.

No restante deste capítulo veremos como criar dois tipos de relatórios comuns: listagens e relatórios com agrupamento. Veremos também detalhes sobre os componentes QuickReport mais importantes.

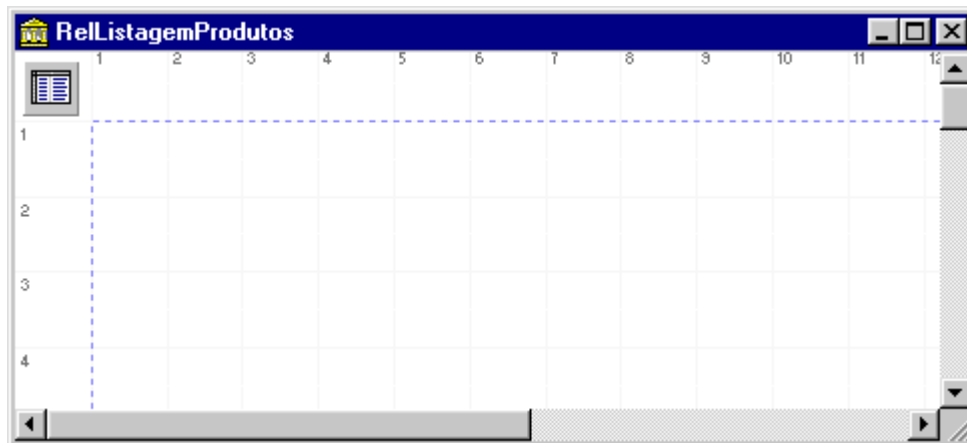
18.1. Criando um relatório simples

Apresentamos a seguir os passos necessários para a criação um relatório simples, que lista os registros de uma tabela ou query.

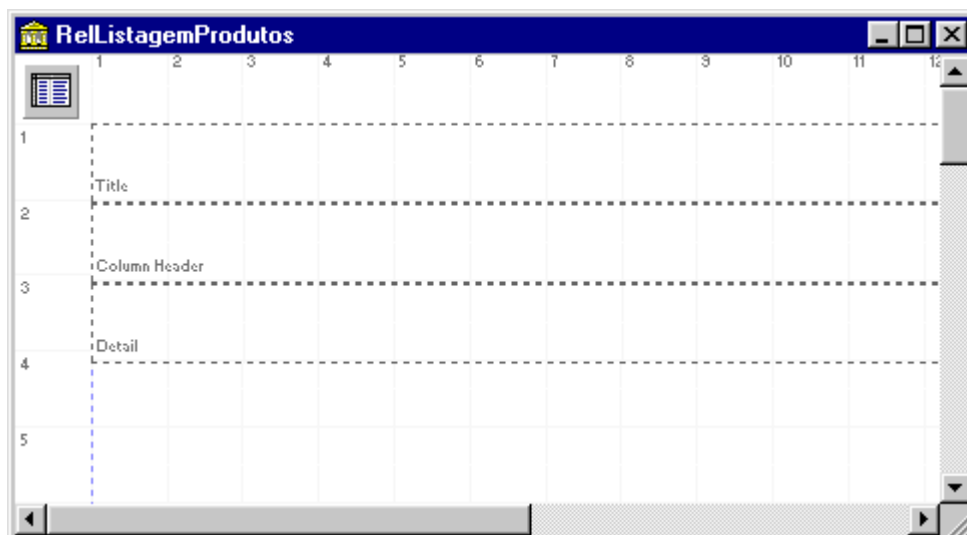
- **Para criar um relatório simples (do tipo "Listagem"):**

1. Crie um novo formulário e adicione a ele um componente *QuickRep*.
1. Ligue o componente *QuickRep* a um *DataSet* (*Table* ou *Query*), alterando a sua propriedade *DataSet*.

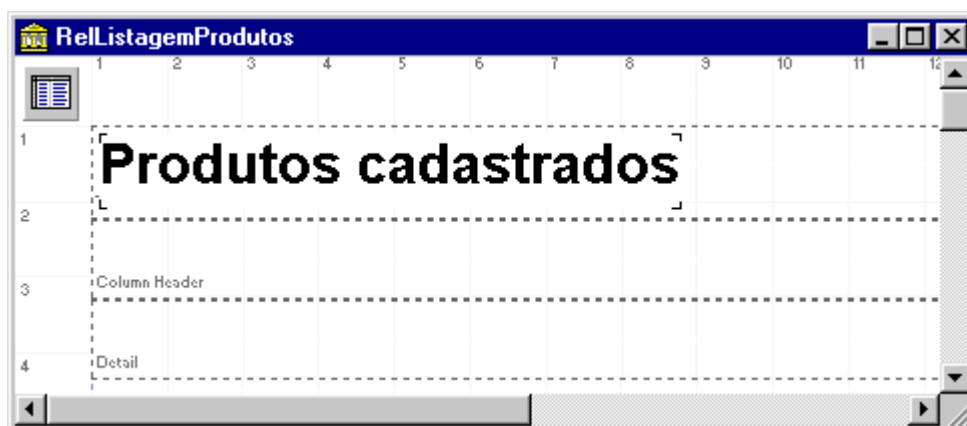
(Você pode adicionar um *Table* ou *Query* ao formulário do relatório)



2. Defina as bandas que devem ser exibidas no relatório (propriedade *Bands*). As três bandas mais importantes são "Title", "Column Header" e "Detail". Altere as propriedades *Bands.HasTitle*, *Bands.HasColumnHeader* e *Bands.HasDetail* para *True* para adicionar estas bandas ao relatório. (Veja a figura abaixo).

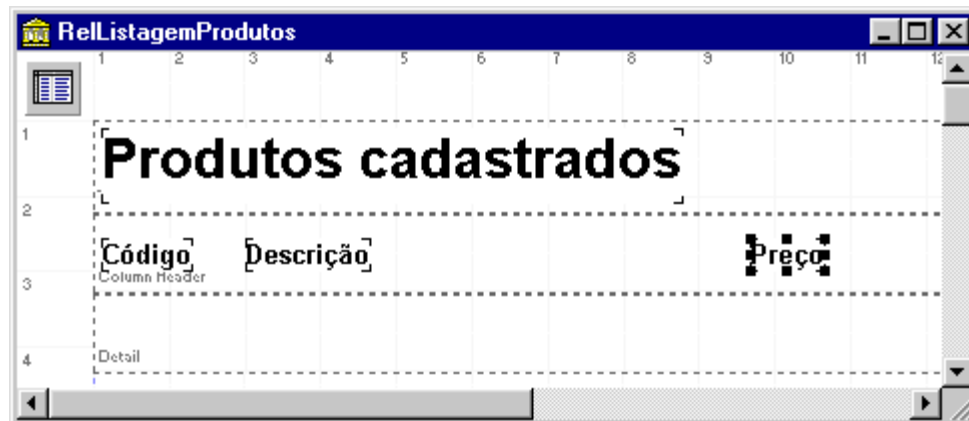


2. Na banda "Title", adicione um componente *QRLabel* e altere sua propriedade *Caption* para o título do relatório. Altere também a propriedade *Font* para destacar o título.



3. Na banda "Column Header", adicione componentes *QRLabel* para os títulos das colunas que deseja exibir no relatório. Altere a propriedade *Caption*, como antes, para definir o texto a ser exibido no componente.

(A banda "ColumnHeader" é repetida a cada nova página, logo depois do título do relatório).

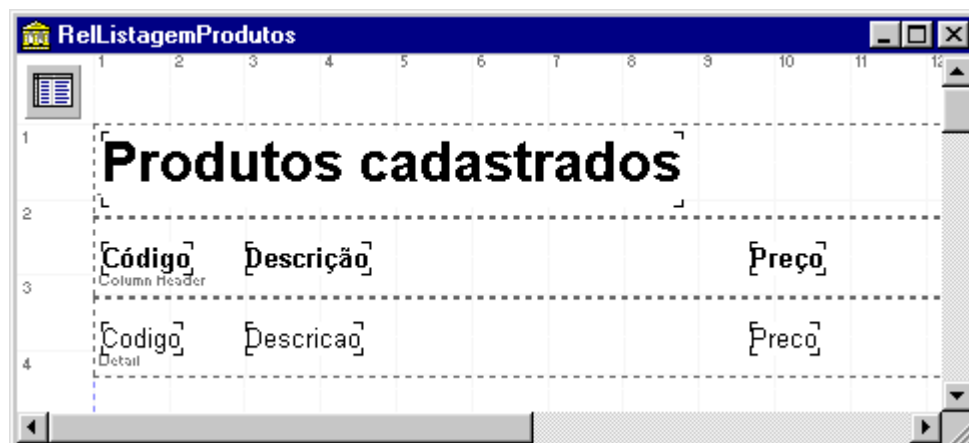


4. Na banda "Detail", adicione componentes *QRDBText* para cada campo do DataSet a ser exibido no relatório.

(A banda "Detail" é a mais importante do relatório. Esta banda é repetida para cada registro do DataSet ligado ao componente *QuickRep*).

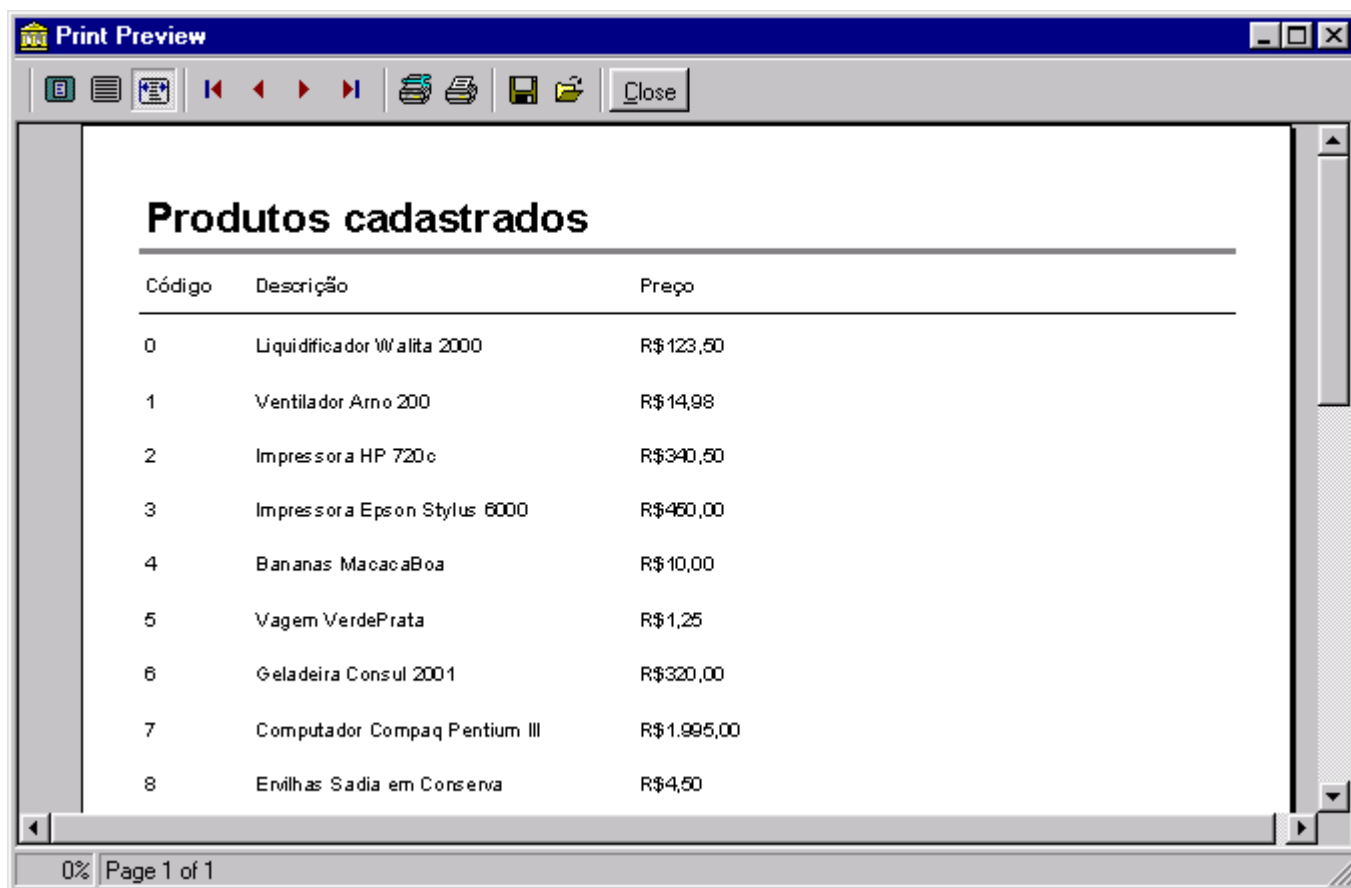
5. Ligue cada componente *QRDBText* aos campos adequados, alterando suas propriedades *DataSet* e *DataField*. Por exemplo, para exibir o campo "Descricao" de uma tabela chamada "TabProdutos", altere *DataSet* para "TabProdutos" e *DataField* para "Descricao".

Quando um componente *QRDBText* é ligado a um campo de um DataSet, o nome do campo passa a ser exibido dentro do componente (veja a figura a seguir).



6. Aplique os formatos necessários aos componentes e às bandas. Use a propriedade *Frame* para definir linhas de contorno, e a propriedade *Font* para definir tipos, estilos e cores de fonte.
7. Para visualizar o relatório em tempo de desenvolvimento, clique com o botão direito no componente *QuickRep* e escolha o comando "Preview".

É mostrada a janela "Print preview", com uma pré-visualização do relatório:



Código	Descrição	Preço
0	Liquidificador Walita 2000	R\$123,50
1	Ventilador Arno 200	R\$14,98
2	Impressora HP 720c	R\$340,50
3	Impressora Epson Stylus 8000	R\$450,00
4	Bananas MacacaBoa	R\$10,00
5	Vagem VerdePrata	R\$1,25
6	Geladeira Consul 2001	R\$320,00
7	Computador Compaq Pentium III	R\$1.995,00
8	Envilhas Sadia em Conserva	R\$4,50

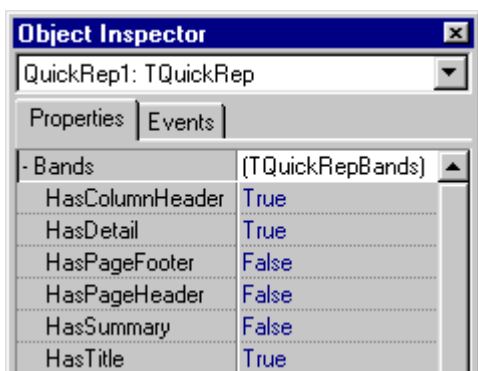
18.2. O Componente *QuickRep*

O componente *QuickRep* é o componente mais importante em um relatório. Este componente deve ser adicionado a um formulário vazio e ligado a um DataSet (Table ou Query). O DataSet pode ser adicionado ao mesmo formulário do relatório, ou colocado em um DataModule separado.

O formulário onde é adicionado o componente *QuickRep* passa a ser apenas um "recipiente" para o relatório – esse formulário não deve ser exibido diretamente. Para exibir ou imprimir um relatório, são usados os métodos *Preview* e *Print*, do componente *QuickRep*, como veremos adiante.

18.3. Bandas e tipos de bandas

Com a propriedade *Bands* do componente *QuickRep*, pode-se adicionar ou remover os seis tipos de bandas mais usados. Para adicionar uma banda, clique duas vezes na propriedade *Bands*, no Object Inspector, e altere para *True* a propriedade "Has..." correspondente ao tipo de banda desejado (veja a figura a seguir).



Segue uma descrição dos seis tipos de bandas que podem ser adicionados usando a propriedade *Bands*:

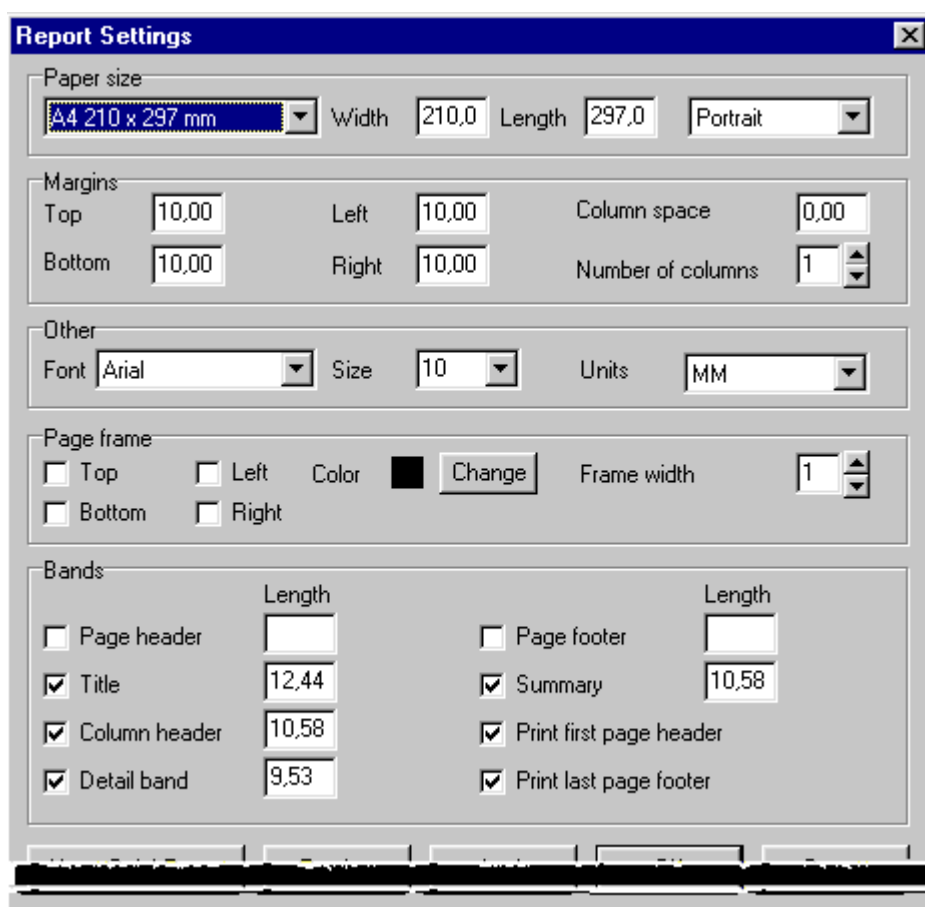
Banda	Descrição
<i>ColumnHeader</i>	A banda <i>ColumnHeader</i> é usada para os títulos das colunas do relatório. Nesta banda é geralmente colocado texto estático – como nomes de campos – usando o componente <i>QRLabel</i> . Esta banda se repete em todas as páginas do formulário, logo abaixo da banda <i>Title</i> .
<i>Detail</i>	A banda <i>Detail</i> ("Detalhe") é o "coração" do relatório. Esta banda é repetida para cada registro. Nesta banda, são colocados componentes com acesso a dados, principalmente os componentes <i>QRDBText</i> .
<i>PageFooter</i>	<p>A banda <i>PageFooter</i> ("Rodapé") é posicionada na extremidade de baixo de cada página do relatório (exceto, possivelmente, na última). Nesta banda podem ser colocados, por exemplo, o número da página, a data ou a hora.</p> <p>Para que a banda <i>PageFooter</i> não apareça na última página do relatório, altere para <i>False</i> a propriedade <i>Options.LastPageFooter</i> do componente <i>QuickRep</i>.</p>
<i>PageHeader</i>	<p>A banda <i>PageHeader</i> ("Cabeçalho") é posicionada na extremidade de cima de cada página do relatório (exceto, possivelmente, na primeira página). Pode-se colocar nessa banda, por exemplo, o título do relatório, a data, ou outras informações relevantes, que devem ser repetidas em todas as páginas.</p> <p>Para que a banda <i>PageHeader</i> não apareça na primeira página do relatório, altere para <i>False</i> a propriedade <i>Options.FirstPageHeader</i> do componente <i>QuickRep</i>.</p>
<i>Summary</i>	A banda <i>Summary</i> ("Resumo") é usada geralmente para exibir dados agregados, como somas, médias, ou contagens. Esta banda é exibida logo depois da última banda <i>Detail</i> (com os dados do último registro). Para realizar os cálculos necessários nessa banda, são usados componentes <i>QRExpr</i> .
<i>Title</i>	A banda <i>Title</i> ("Título") é usada para exibir o título do relatório. Esta banda aparece apenas uma vez, na primeira página, e é posicionada logo depois do cabeçalho da página (se este existir).

18.4. Alterando a formatação geral do relatório

O componente *QuickRep* oferece muitas opções de formatação, que podem ser alteradas através do Object Inspector (propriedades *Bands*, *Font*, *Frame* e *Page*), ou usando o comando **Report Settings** disponível no menu de atalho do componente. Clique com o botão direito no componente *QuickRep* e escolha o comando **Report Settings** (ou clique duas vezes no componente) para exibir a seguinte caixa de diálogo ilustrada a seguir.

Veja o que pode ser alterado em cada seção desta caixa:

- Na seção *Paper size*, você pode alterar o tipo e o tamanho do papel, e a orientação (*retrato* ou *paisagem*).
- Na seção *Margins*, as margens das páginas, em milímetros (ou em outra unidade, especificada na seção *Other*), podem ser alteradas com precisão. O número de colunas e o espaçamento entre colunas do relatório também podem ser alterados.



- Na seção *Other*, pode-se alterar a fonte e o tamanho de fonte usados como padrão no formulário. A fonte e o tamanho escolhidos aqui têm efeito sobre todos os componentes do relatório, exceto aqueles que tiveram suas fontes alteradas diretamente. A unidade de medida usada no relatório também pode ser alterada – a unidade escolhida determina como são medidas várias propriedades dos componentes do formulário, como a largura e a altura, por exemplo.
- Na seção *Page Frame*, pode-se definir bordas para as páginas do relatório. Bordas acima, abaixo, à esquerda ou à direita, e com várias cores e espessuras podem ser definidas. (A cor e a espessura valem para todas as bordas).
- Na última seção, *Bands*, é possível acrescentar ou remover bandas do relatório (ativando ou desativando as opções correspondentes). Pode-se também ajustar precisamente a altura de cada banda.

18.5. Propriedades comuns a todas as bandas

Há várias propriedades comuns a todas as bandas, como as que determinam a altura da banda ou linhas de contorno. Para alterar as propriedades de uma banda, deve-se primeiro selecioná-la (normalmente) com um clique. Veja a seguir uma descrição das propriedades mais importantes.

Propriedade	Descrição
<i>ForceNewPage</i>	Altere a propriedade <i>ForceNewPage</i> para <i>True</i> , para que seja iniciada uma nova página cada vez que a banda selecionada é impressa.
<i>ForceNewColumn</i>	Altere a propriedade <i>ForceNewColumn</i> para <i>True</i> , para que seja iniciada uma nova coluna cada vez que a banda selecionada é impressa. (Aplicável somente em relatórios com mais de uma coluna).
<i>Height</i>	A propriedade <i>Height</i> determina a altura da banda, na unidade de medida atual. Essa propriedade pode ser alterada de forma interativa no relatório – clique na banda e arraste os pequenos quadrados que aparecem acima e abaixo da banda.
<i>Frame</i>	<p>A propriedade <i>Frame</i> determina a exibição de <i>bordas</i> em volta da banda de vários formatos e cores, em volta das páginas do relatório. As opções são bastante intuitivas:</p> <p><i>Color</i>: A cor das bordas. (Se aplica a todas as quatro bordas).</p> <p><i>DrawBottom</i>, <i>DrawLeft</i>, <i>DrawRight</i>, <i>DrawTop</i>: Determinam quais bordas serão exibidas – abaixo, à esquerda, à direita e acima, respectivamente.</p> <p><i>Style</i>: o estilo da linha das bordas (sólido, tracejado, pontilhado, etc.)</p> <p><i>Width</i>: a espessura das bordas. (Se aplica a todas as quatro bordas).</p>

18.6. Outros componentes importantes

Veremos agora mais alguns detalhes importantes sobre os quatro componentes mais usados em relatórios: *QRLabel*, *QRDBText*, *QRSysData* e *QRExpr*.

18.7. O Componente *QRLabel*

O componente *QRLabel* é usado para textos estáticos, como títulos ou nomes de campos. A propriedade *Caption* determina o **texto** que será exibido no componente. A propriedade *Font* define as opções de **fonte** (fonte, estilo, tamanho e cor).

Três propriedades relacionadas determinam o **tamanho** e a **organização** do texto exibido no componente *QRLabel*. *AutoSize* faz com que o *QRLabel* altere sua *largura* automaticamente, de acordo com o comprimento do texto. *AutoStretch* faz com que a *altura* do componente seja ajustada automaticamente (para textos com mais de uma linha). *WordWrap* faz com que textos longos sejam quebrados em várias linhas. O efeito dessas três propriedades às vezes depende do valor das outras duas. *AutoStretch*, por exemplo, só faz sentido quando *WordWrap* for *True*.

O **alinhamento** de um componente *QRLabel* pode ser alterado usando as propriedades *Alignment* e *AlignToBand*. *Alignment* altera o alinhamento com relação ao tamanho do componente, da forma usual: *taLeftJustify* alinha pela esquerda, *taRightJustify* pela direita, e *taCenter* pelo centro. A propriedade *AlignToBand* faz com que o alinhamento seja feito em

relação à banda. Se *AlignToBand* for *True* e *Alignment* for *taRightJustify*, por exemplo, o *QRLabel* será alinhado pela extremidade direita da banda.

18.8. O Componente *QRDBText*

O componente *QRDBText* é uma versão, com acesso a bancos de dados, do componente *QRLabel*. Praticamente todas as propriedades do componente *QRLabel* se aplicam também ao componente *QRDBText*. Além dessas, há mais duas propriedades importantes: *DataSet* e *DataField*.

A propriedade *DataSet* determina o *Table* ou *Query* ligado ao componente *QRDBText*. A propriedade *DataField* determina o *campo* da *Table* ou *Query*, cujo conteúdo é exibido no componente. Note que diferentemente dos componentes *Data Controls* (como o *DBText* e o *DBEdit*), o acesso a dados é realizado diretamente e não através de um componente *DataSource*.

18.9. O Componente *QRSysData*

O componente *QRSysData* é usado para exibir informações úteis, como a data, a hora, ou o título do relatório. Este componente é muito semelhante ao componente *QRLabel* – as propriedades de formatação, por exemplo, são as mesmas. A propriedade *Data* determina a informação exibida no componente. Veja alguns valores que podem ser atribuídos à propriedade *Data*:

Altere <i>Data</i> para...	Para mostrar...
<i>qrsDate</i> , <i>qrsTime</i> , <i>qrsDateTime</i>	A data, a hora, ou a data e hora atuais, respectivamente.
<i>qrsPageNumber</i>	O número da página.
<i>qrsReportTitle</i>	O título do relatório (extraído da propriedade <i>ReportTitle</i> do componente <i>QuickRep</i>).

18.10. O Componente *QRExpr*

Com o componente *QRExpr*, é possível realizar cálculos, ou outras operações no relatório. Podem ser calculados totais ou médias de um campo, por exemplo. A propriedade *Expression* determina a operação realizada. A sintaxe das expressões segue uma sintaxe parecida com a da linguagem SQL. Veja algumas expressões válidas:

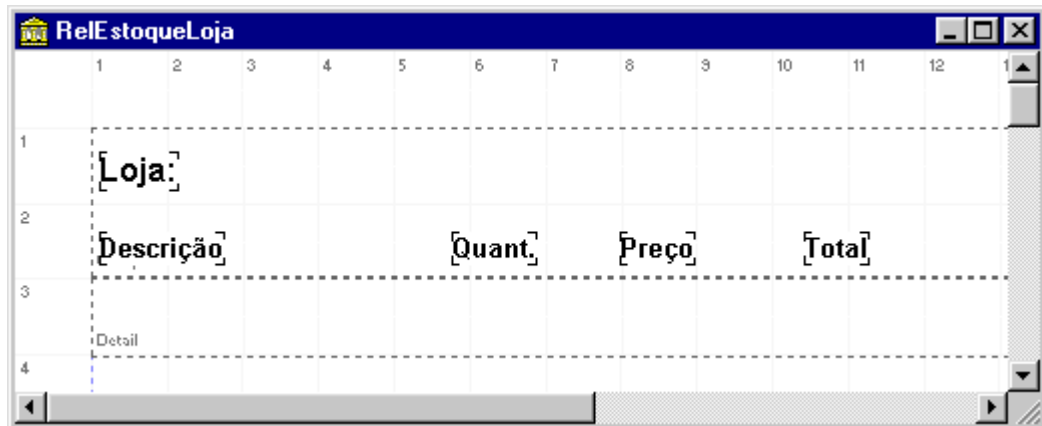
- `Preco*Quantidade`
- `SUM(Preco)`
- `MIN(TabEstoque.Quantidade)`
- `MAX(TabProdutos.Preco)`
- `AVERAGE(Preco*Quantidade)`
- `IF(Preco = 0, 'Em estoque', 'Fora de estoque')`

A propriedade *Expression* pode ser alterada diretamente, ou através do "Expression Wizard" (clique duas vezes ao lado do nome da propriedade para abri este "Wizard").

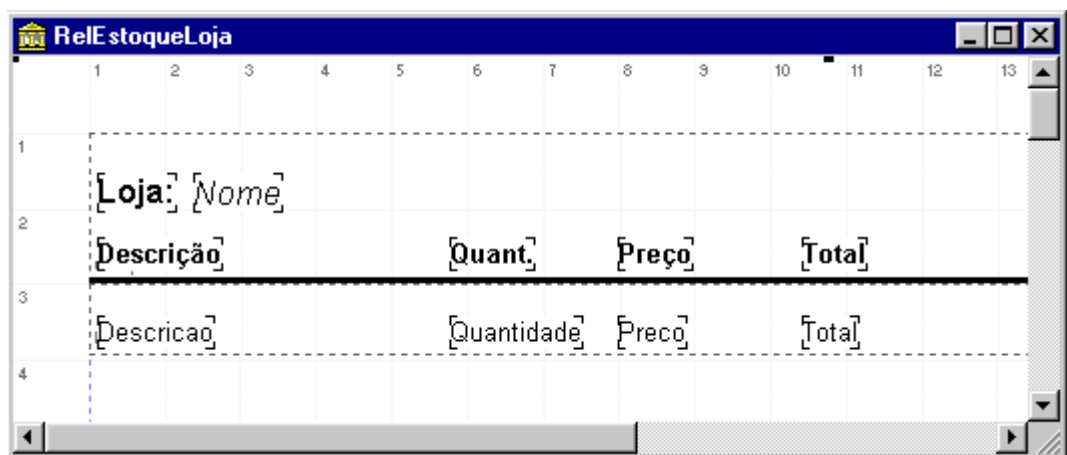
Os componentes *QRExpr* são freqüentemente colocados em bandas do tipo "Summary", para realizar cálculos agregados, como somas ou médias, mas podem ser colocados em qualquer tipo de banda.

CodLoja	CodProduto	Quantidade	Codigo	Descricao	Preco	Codigo	Nome
---------	------------	------------	--------	-----------	-------	--------	------

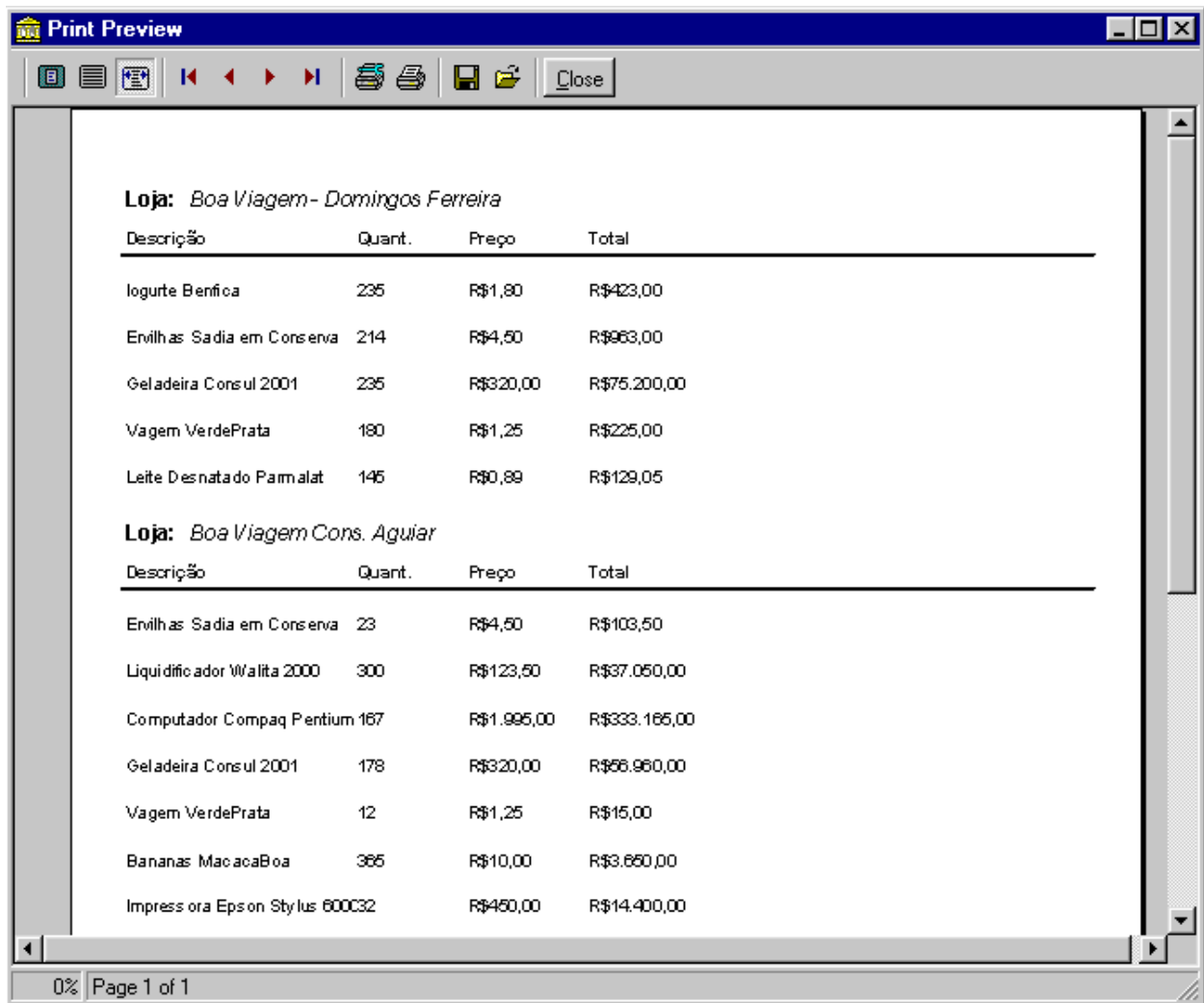
- Altere para *True* a propriedade *Active* do componente *Query*. Isso permite testar o relatório em tempo de desenvolvimento, usando o comando **Preview**.
- Ligue o componente *QuickRep* ao componente *Query* (propriedade *DataSet*).
- Na banda "Group Header" (o componente *QRGroup*) adicione componentes *QRLabel*, como na ilustração a seguir:



- Na banda "Detail", adicione componentes *QRDBText* para cada um dos campos a serem listados. Ligue cada um dos componentes à *Query* (alterando a propriedade *DataSet*) e altere as suas propriedades *DataField* para os nomes dos campos.
- Na banda "Group Header", ao lado de "Loja", adicione outro *QRDBText* e ligue-o ao campo "Nome" da *Query* (o nome da loja). Altere também a propriedade *Frame.DrawBottom* da banda para *True* e a propriedade *Width* para 2. A ilustração a seguir mostra os resultados das mudanças.



- Para realizar o agrupamento automático, altere a propriedade *Expression* da banda "Group Header" para "Nome" (sem as aspas). Isso faz com que os dados sejam agrupados por loja.
- Finalmente, para testar o relatório, clique nele com o botão direito e escolha o comando **Preview**. A ilustração a seguir mostra o relatório gerado, com alguns dados de amostra:



18.12. Imprimindo e pre-visualizando relatórios

Até agora vimos apenas como exibir relatórios em tempo de desenvolvimento, usando o comando **Preview**. Relatórios podem ser também exibidos e – o que é mais importante – *impressos*, em tempo de execução. É tudo muito simples.

Para **pre-visualizar** um relatório em tempo de execução, use o método *Preview* do componente *QuickRep* do relatório. Esse comando, claramente deve ser chamado a partir de um comando de menu, ou de algum outro componente, em um formulário separado (lembre-se que o formulário que contém o relatório não deve ser exibido).

O código a seguir mostra a pre-visualização do relatório em um formulário chamado "RelVendas":

```
procedure TFormRelatorios.BtPreviaClick(Sender: TObject);
begin
    RelVendas.QuickRep1.Preview;
end;
```

Note que foi usado o nome padrão do componente QuickRep ("QuickRep1"). Recomenda-se, no entanto que se dê um nome mais sugestivo a esse componente ("Relatorio", por exemplo).

Para **imprimir** um relatório, use o método *Print* do componente *QuickRep* do relatório, como no seguinte código:

```
procedure TFormPrincipal.RelatorioVendas1Click(Sender: TObject);  
  
begin  
  
    RelVendas.QuickRep1.Print;  
  
end;
```