UFSM

Caderno Didático

LAZARUS IDE

Bruno B. Boniati

Frederico Westphalen, 2011

Sumário

CAPITULO I - Introdução ao Lazarus 01

O que é o Lazarus 01
Visão geral do Lazarus 01
Características do ambiente Lazarus e da linguagem Object Pascal 01
Características da Programação com Lazarus e Object Pascal 02
Antes de Começar... "O Projeto da Interface Gráfica" 02
Padronização 02
Esqueça a programação "modo caractere" ou "console" 02
Simplicidade 02
Nomenclatura 02
Como é formado uma Aplicação em Lazarus 03
Arquivos que Compõem uma Aplicação 03
Arquivos Gerados pela Compilação 03
Código fonte do arquivo Project (.LPR) 04
Código fonte do arquivo Unit (.PAS) 05
Ambiente de Desenvolvimento Lazarus 07
Janela Principal 07
Object Inspector 08
Form Designer e Program Editor 09

CAPITULO II – Programação Orientada a Objetos (Conceitos) 10

Introdução 10
Abstração de Dados 10
Classe 10
Propriedade 10
Métodos 10
Instâncias 10
Objeto 10
Visibilidade 13
Declaração, Instanciamento, Destruição e Referência a Objetos 13
Encapsulamento 15
Herança 15
Características / Benefícios 15
Implementação 16
Polimorfismo 16
Características / Benefícios 16

CAPITULO III - Object Pascal 17

Básico da Linguagem 17
Comentários 17
O Ponto-e-Vírgula e o Ponto 17
Tipos de Dados Padrão 18
Tipos Inteiros 18
Tipos Reais 18
Tipos Texto 18
Tipos Ordinais 18
Boolean 18
TDateTime 18
Variant 19
Tipos de Definidos pelo Usuário 19
Strings Limitadas 19
Sub-Faixa 19
Enumerações 19
Ponteiros 20
Records 20
Arrays 20
Sets 20
Elementos Fundamentais de Controle 21
Operadores 21
Comandos de DECISÃO 21
Comandos de LOOPING 22
Procedimentos e Funções 24
Funções e Comandos Internos (principais) 26
Tratamento de Strings 26
Tratamento de Ordinais 27
Tratamento de valores TDateTime 27
Conversões de Tipos de Dados 28
Strings de Formato 30
Outras Funções 31

CAPITULO IV – Classes e Objetos Padrões do Lazarus 32

Propriedades 32
Tipos de Propriedade 32
Propriedades Comuns 32
Eventos 33
Eventos Comuns 33
Métodos 33
Métodos Comuns 33
Biblioteca de Classes 34
Janelas 34
Componentes Padrão 34
Caixas de Diálogo 40
Menus 40
Classes Não Visuais 41
Exceções 45
Blocos Protegidos 45
Principais Exceções 46
Blocos de Finalização 46
Geração de Exceções 47

CAPÍTULO V – Banco de Dados 48

Considerações Iniciais 48
Configurando a conexão e o acesso aos dados 48
SQLdb – Componentes de Acesso aos Dados 48
DataControls – Componentes de Controle aos Dados 61

CAPÍTULO VI – Relatórios e Comandos de Impressão 63

LazReport – Conceitos Básicos 63
Utilização de variáveis do relatório 66
Utilizando scripts 66
Relatórios baseados em consultas de banco de dados 67
Visualização e Impressão 68

Referências Bibliográficas 69

Sites Especializados (em português) 69

CAPITULO I - Introdução ao Lazarus

O que é o Lazarus

O Lazarus é um ambiente de desenvolvimento integrado (IDE – *Integrated Development Environment*) para aplicações orientadas a objetos, que utiliza a linguagem de programação Object Pascal. O Lazarus simula o funcionamento do Delphi® (IDE comercial da Borland). Disponibiliza ferramentas de desenvolvimento que aumentam a produtividade, facilitando a programação.

O ambiente Lazarus é estável e permite a criação de aplicações gráficas, de console e também para web. Representa uma importante alternativa para permitir a comunidade de programadores Pascal e Delphi o desenvolvimento de aplicações comerciais para diferentes plataformas (MS Windows, Linux, MAC OS, etc.). É uma ferramenta livre, gratuita e de código aberto.

Visão geral do Lazarus

O Projeto Lazarus iniciou em 1999 com três colaboradores Cliff Baeseman, Shane Miller e Michael A. Hess. Ambos haviam se envolvido em um projeto denominado *Megido*, cujo objetivo era desenvolver um clone para o IDE Delphi, mas que acabou sendo dissolvido. Frustrados com a "morte" do projeto eles começaram o projeto Lazarus que teve um grande crescimento de apoiadores e desenvolvedores durante os anos seguintes. Marc Weustink e Mattias Gaertner que se juntaram posteriormente ao grupo têm sido os maiores contribuidores para o núcleo que faz o Lazarus existir. Lazarus é a conhecida figura bíblica que foi ressuscitada por Cristo. O projeto foi chamado de Lazarus porque ele se originou da morte do *Megido*.

O ambiente Lazarus provê um editor de código personalizável, além de um ambiente de criação visual de formulários (janelas) acompanhado de um gerenciador de pacotes, depurador (*debugger*) e completa integração da interface gráfica com o compilador Free Pascal. Em função da produtividade que garante ao desenvolvedor pode ser classificado como uma ferramenta RAD (*Rapid Application Development*). Uma de suas principais vantagens é a linguagem de programação utilizada: Object Pascal (uma evolução do Pascal padrão). O Pascal surgiu no final dos anos 60 e, até hoje, é usada como uma das primeiras linguagens de programação para estudantes de computação, devido a sua facilidade de aprendizado.

A LCL (*Lazarus Component Library*) é a biblioteca que agrega o conjunto de classes e componentes visuais e não visuais que funcionam em múltiplas plataformas chamando as funções de diferentes bibliotecas de interfaces de usuário. Tanto a LCL do Lazarus, quanto à FCL do Free Pascal são licenciadas sobre a LGPL, o que permite o desenvolvimento de aplicações que podem ser comercializadas. A IDE Lazarus, contudo, é licenciada sobre a licença GPL, ou seja, pode ser modificada, mas não pode ser vendida (deve ser disponibilizada para a comunidade).

Diferentemente da plataforma Java, onde o código é escrito e compilado uma única vez e pode ser executado em qualquer plataforma (sem a necessidade de uma nova compilação), o Lazarus e o Free Pascal permitem que o código seja escrito uma única vez e compilado para qualquer plataforma. Não é necessário alterar o código para que ele produza o mesmo efeito em diferentes plataformas, mas é necessário recompilá-lo.

No IDE Lazarus, a criação de aplicativos começa normalmente com a montagem de componentes gráficos em janelas, também permite a utilização de componentes desenvolvidos por terceiros ou o desenvolvimento de componentes próprios. A conexão com banco de dados também é suportada por componentes específicos que acompanham o IDE.

Características do ambiente Lazarus e da linguagem Object Pascal

- Geração de um executável verdadeiro, independente de *run-time*.
- Utilização de um dialeto da linguagem Object Pascal para escrever os procedimentos do programa.
- Utilização de componentes, que são definidos como objetos, o que permite a herança.

- Possibilidade de criação de novos componentes na própria linguagem e incorporação dos mesmos ao IDE.
- Acesso facilitado a banco de dados.
- Ambiente de depuração integrado.

Características da Programação com Lazarus e Object Pascal

- Desenvolvimento orientado pelo desenho de formulários (janelas).
- Interface com usuário feita através de componentes visuais.
- Disponibilização de controles pré-desenvolvidos que dão acesso às características do sistema.
- Cada controle ou componente dispõe de propriedades, métodos e pode responder a eventos.
- As propriedades podem ter seus valores definidos em tempo de desenvolvimento e alterados em tempo de execução.
- Os eventos são as mensagens que cada componente pode responder, tendo associado a eles um procedimento de evento.

Antes de Começar... "O Projeto da Interface Gráfica"

Padronização

- Considere aplicações existentes, os padrões de sucesso.
- Utilize botões e controles de acesso rápido.
- Utilize teclas de atalho.
- Enfatize imagens e ícones.

Esqueça a programação "modo caractere" ou "console"

- Não utilize múltiplos níveis de menus.
- Não exagere em cores diferentes.
- Utilize a tecla TAB para passar de um campo para outro.
- Utilize a tecla ENTER ou um botão para executar ações.

Simplicidade

- Procure facilidades de uso.
- Permita liberdade de ação.
- Permita diferentes maneiras de se alcançar o mesmo resultado.
- Procure as soluções mais intuitivas.
- Procure adotar os símbolos que o usuário já esteja acostumado.

Nomenclatura

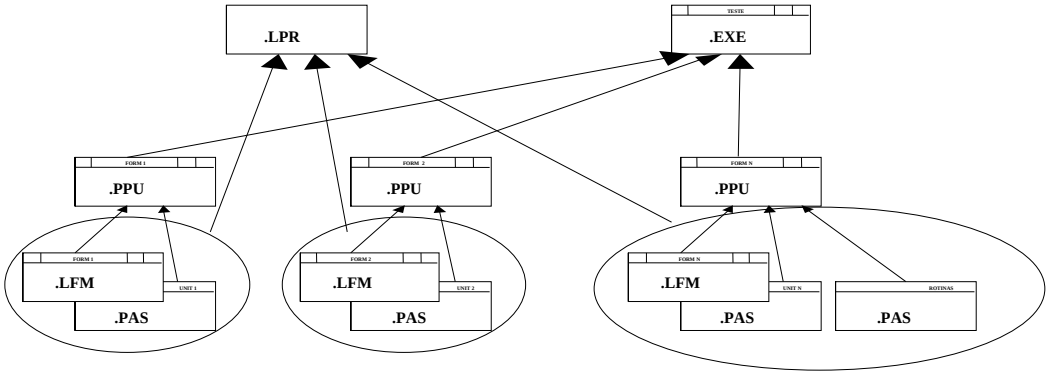
Normalmente os componentes são nomeados utilizando-se uma convenção, onde as primeiras letras, minúsculas e sem as vogais, identificam o tipo do componente e o restante identifica a função deste, assim, btnSair, seria o botão de sair.

Se a função do componente for um nome composto esse nome deve ser escrito com os primeiros nomes abreviados e com letras maiúsculas e minúsculas, como em btnRelVendas, que identifica o botão do relatório de vendas ou btnRelVenProduto, que identifica o botão do relatório de vendas por produto.

Como é formado uma Aplicação em Lazarus

Quando um projeto no Lazarus é criado, ele já mostra uma UNIT (unidade básica do código fonte) com várias linhas de código. Um projeto Lazarus tem, inicialmente, duas divisórias: uma UNIT, que é associada a um Form (formulário da interface gráfica), e outra UNIT que contem o Project (projeto) e que organiza todos os formulários (Form) e códigos (Unit) da aplicação.

No Lazarus temos: o Project, os Forms e as Units. Para todo Form temos pelo menos uma Unit (Código do Form), mas temos Units sem Form (códigos de procedures, funções, etc).



Arquivos que Compõem uma Aplicação

Extensão	Definição	Função
.LPR	Arquivo do Projeto (Lazarus Project)	Código fonte em Pascal do arquivo principal do projeto. Lista todos os formulários e units no projeto, e contém código de inicialização da aplicação.
.LPI	Arquivo XML com Informações sobre o Projeto (Lazarus Project Information)	Contém informações sobre o tipo de projeto, situação da área de trabalho, opções de ambiente específicas a este projeto, etc.
.PAS	Código fonte da Unit (Object Pascal)	Um arquivo .PAS é gerado para cada formulário que o projeto contém. Um projeto pode conter um ou mais arquivos .PAS associados com algum formulário. Contem todas as declarações e procedimentos incluindo eventos de um formulário.
.LFM	Arquivo gráfico do formulário (Lazarus Form)	Arquivo que contém as propriedades do desenho de um formulário contido em um projeto. Um .LFM é gerado conjuntamente com um arquivo .PAS para cada formulário do projeto.
.RC	Arquivo de Recursos do Compilador	Arquivo que contém o ícone, mensagens da aplicação e outros recursos usados pelo projeto.
.LRS	Arquivo de Recursos (Lazarus Resource)	Arquivo que contém ícones, mensagens e outros recursos utilizados pelo projeto e pelos formulários.

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

Arquivos Gerados pela Compilação

Extensão	Definição	Função
.EXE	Arquivo compilado executável	Este é um arquivo executável pelo qual a aplicação será distribuída. Este arquivo incorpora todos os arquivos .PPU gerados quando sua aplicação é compilada.
.PPU	Código objeto da Unit	A compilação cria um arquivo .DCU para cada .PAS no projeto.

Estes arquivos podem ser apagados da máquina do desenvolvedor para economizar espaço em disco.

Código fonte do arquivo Project (.LPR)

Neste arquivo está escrito o código de criação da aplicação e seus formulários. O arquivo Project tem apenas uma seção formada pelo seguinte código:

PROGRAM - Define o Projeto;
USES - Cláusula que inicia uma lista de outras unidades.
Forms = É biblioteca do Lazarus que define as classes e tipos para utilização de formulários
Interfaces = Biblioteca que determina qual interface deve ser utilizada
LResources = Biblioteca com operações para manipulação de recursos (arquivo .LRS)
Unit1 = A unidade do formulário principal
BEGIN
//Código de inicialização da aplicação
END

Abaixo o código padrão de uma aplicação recém-criada.

```
program Project1;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Interfaces, // this includes the LCL widgetset
  Forms, Unit1, LResources
  { you can add units after this };

{$IFDEF WINDOWS}{$R project1.rc}{$ENDIF}

begin
  {$I project1.lrs}
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Código fonte do arquivo Unit (.PAS)

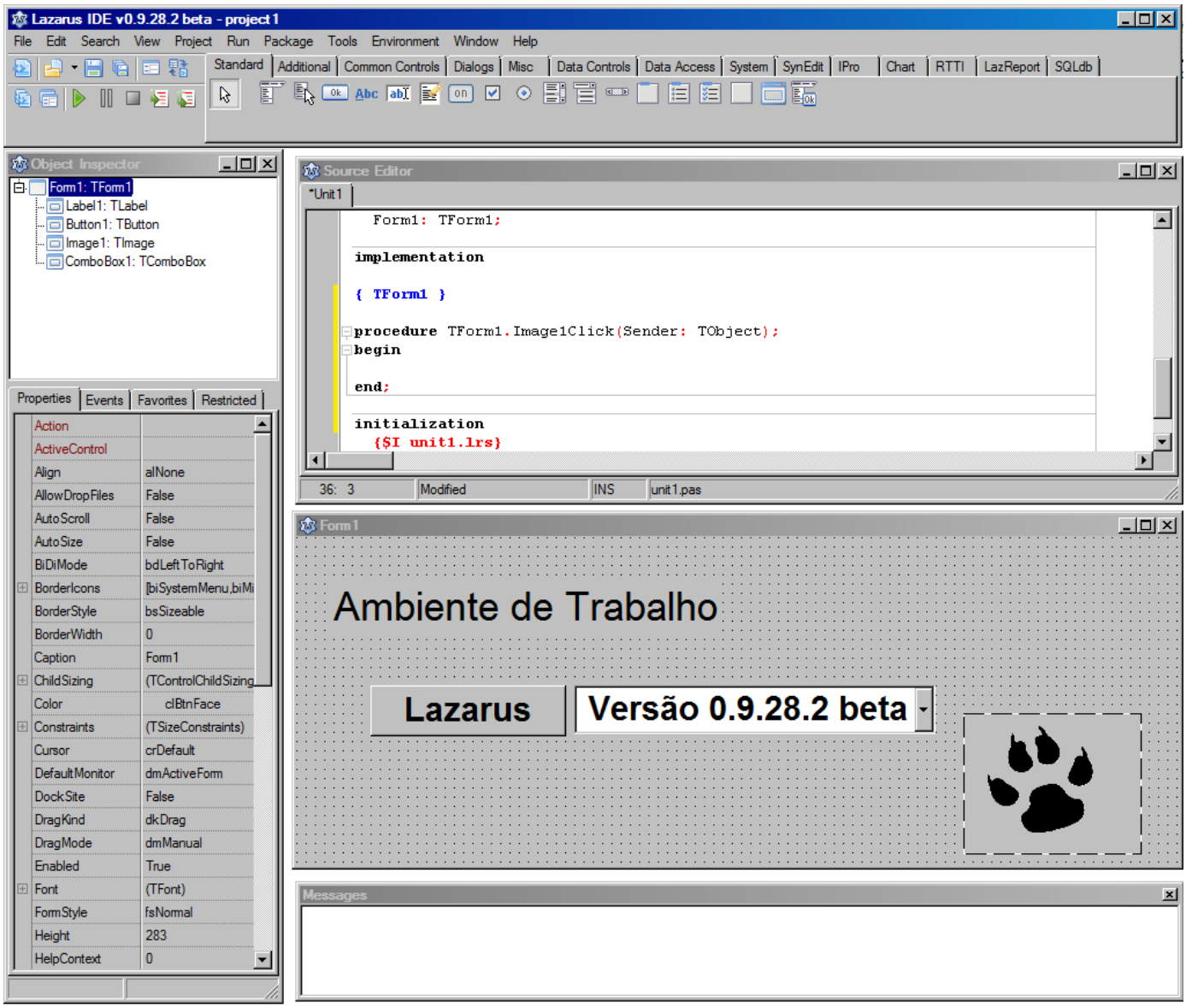
Um arquivo .PAS é um código object pascal. Dentro de um projeto cada arquivo .PAS representa uma Unit. Algumas Units são associadas a um formulário, neste caso há um arquivo .PAS e outro .LFM com o mesmo nome. O código de uma unit associada a um formulário é manipulado pelo IDE à medida que componentes são adicionados ao formulário correspondente. A seguir as principais seções de uma Unit:

- **Seção Unit:** indica o nome da unit.
- **Seção Interface**: Declarações de constantes, tipos de variáveis, funções e procedures gerais da Unit/Form. As declarações desta seção são visíveis por qualquer Unit.
- **Seção Uses:** Contém as units acessadas por este arquivo. Se um nome de uma unit aparecer dentro da cláusula uses, todas as declarações da parte interface da unit poderão ser utilizados na unit ou programa atual. Algumas units geralmente utilizadas na seção uses:
  - *Classes:* elementos de baixo nível do sistema de componentes;
  - *SysUtils:* utilitários do sistema, funções de uso comum (strings, data/hora);
  - *FileUtil*: funções utilitárias para tratamento de arquivos
  - *LResources = Biblioteca com operações para manipulação de recursos (arquivo .LRS)*
  - *Forms:* componentes de forma e componentes invisíveis de aplicativos;
  - *Controls:* elementos de nível médio do sistema de componentes;
  - *Graphics:* elementos gráficos;
  - *Dialogs:* componentes de diálogo comuns.
- **Seção Type:** Declara os tipos definidos pelo usuário. Subseções: Private, declarações privativas da Unit. Public declarações publicas da Unit.
- **Seção Var:** Declara as variáveis privadas utilizadas.
- **Seção Implementation:** Contém os corpos das funções e procedures declaradas nas seções Interface e Type. Nesta seção também estão definidos todos os procedimentos dos componentes que estão incluídos no Form. As declarações desta seção são visíveis apenas por ela mesma.
- **Inicialization:** Nesta seção, que é opcional, pode ser definido um código para proceder às tarefas de inicialização da Unit quando o programa começa. Ela consiste na palavra reservada inicialization seguida por uma ou mais declarações para serem executadas em ordem. Normalmente essa seção contém uma diretiva de compilação que inclui as informações do arquivo de recursos associado à unit: {$I nome_unit.lrs}.
- **Finalization:** Esta seção, também opcional, faz o serviço oposto da inicialization. Todo o código dela é executado após o programa principal ter terminado.

Abaixo veja como fica a unit quando você abre um projeto novo:

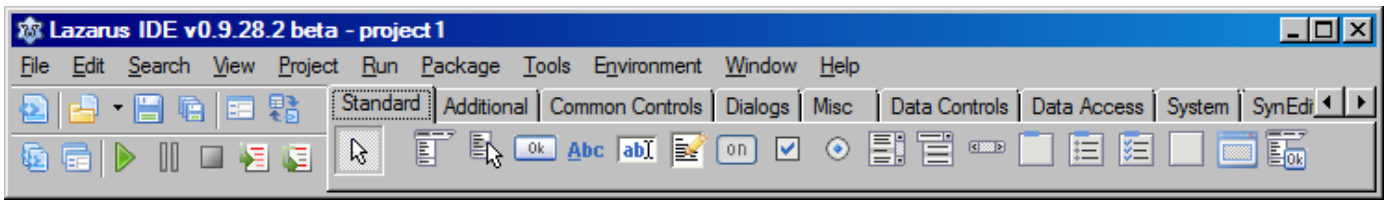
```
unit Unit1;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, LResources, Forms, Controls, Graphics, Dialogs;

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

var
  Form1: TForm1;

implementation

initialization
  {$I unit1.lrs}
end.
```

Ambiente de Desenvolvimento Lazarus



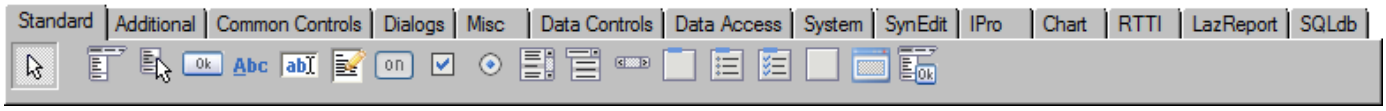
Janela Principal

A janela principal do Lazarus é composta pela barra de menus, Speed Bar e paleta de componentes:



Os Speed Buttons, são atalhos para diversos comandos de menu. Oferecem o recurso das Tool Tips, que são pequenas caixas de texto amarelas, exibidas quando o ponteiro do mouse é deixado por sobre um botão durante algum tempo e exibem a ação que será acionada quando ele for clicado.

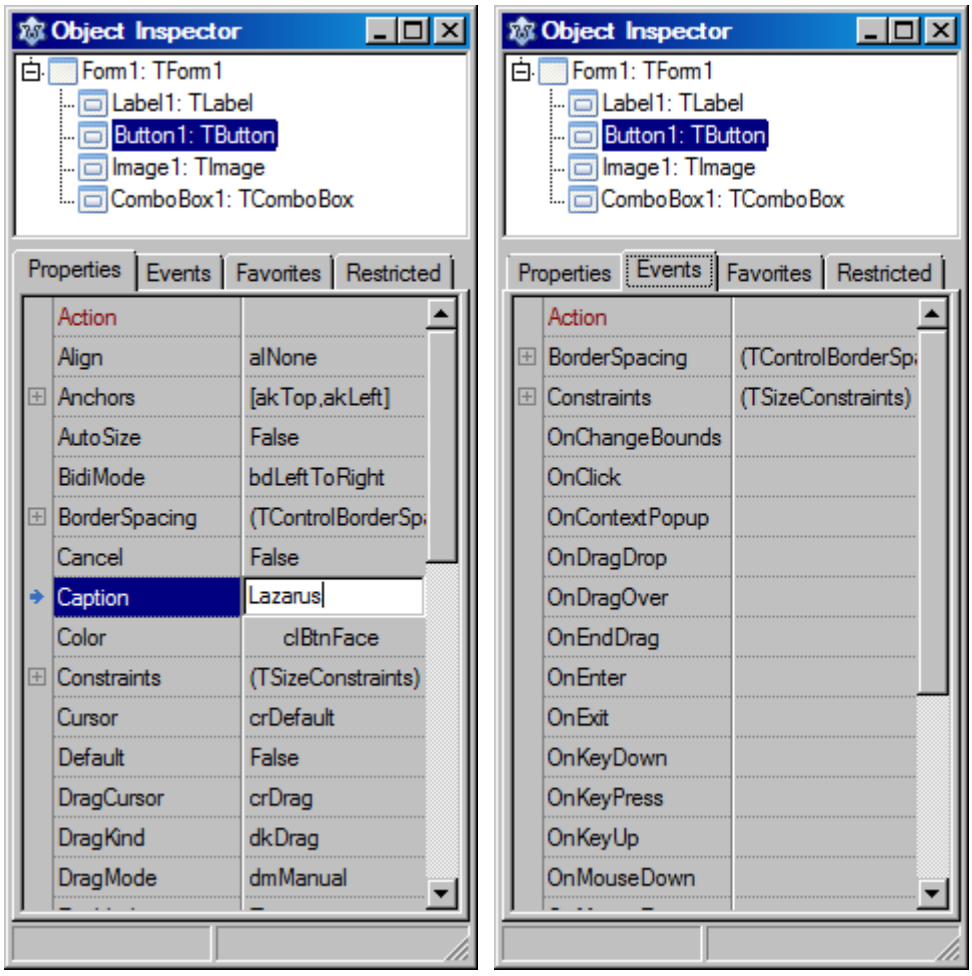
O restante da área principal é formado pela paleta de componentes:



Ela mostra todos os controles, tanto visíveis como não visíveis, que podem ser colocados em um formulário. O Lazarus utiliza a construção da paleta de componentes em abas, onde cada uma reúne componentes de diferentes categorias, agrupados por funcionalidade. Por exemplo, os componentes mostrados na figura anterior são todos componentes padrão (aba *Standard*).

Object Inspector

O *object inspector* é uma janela que aparece à esquerda da janela do *form designer* por padrão e pode ser visualizado de qualquer parte do projeto pressionando a tecla **F11**. Compõe-se de quatro abas, sendo as duas mais importantes as abas *Properties* (Propriedades) e *Events* (Eventos).



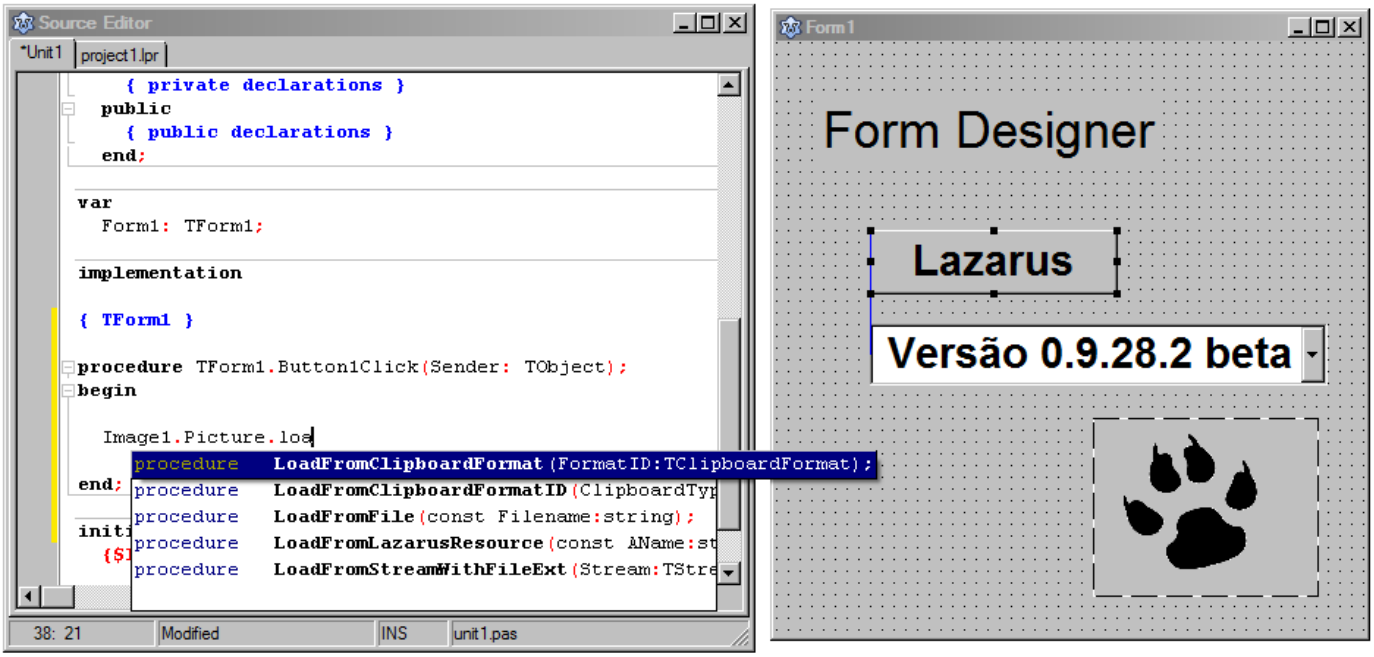
- **Propriedades:** Refere-se a aspectos e características de um determinado componente que podem ser alterados tanto em tempo de design como em tempo de execução. Essencialmente, propriedades são atributos que controlam a aparência de um componente e a maneira como ele age e reage.

- **Eventos**: Eventos são grupos de código que são fornecidos e que determinam a ação a ser tomada em tempo de execução para uma determinada mensagem ou evento do sistema operacional. Clicando sobre o evento o Lazarus associa um procedimento a ele e abre a janela de edição de código para que seja feita a programação.

O *object inspector* do Lazarus oferece ainda duas outras abas: ***Favorities*** que destaca características (propriedades e eventos) específicas do componente em questão e ***Restricted*** que indica possíveis restrições do componente em diferentes bibliotecas gráficas entre plataformas de sistemas operacionais.

Form Designer e Program Editor

O *Form Designer* é basicamente o(s) formulário(s) onde montamos nossa aplicação e inserimos os componentes. Já o Program Editor, é o editor de código, onde a programação é efetuada. Compõe-se de abas superiores quer permitem acesso ao código de outras UNITs, além do recurso de Complemento de código, que exibe automaticamente as propriedades e métodos dos objetos.



CAPITULO II – Programação Orientada a Objetos (Conceitos)

Introdução

A Metodologia Orientada a Objetos é relativamente nova, pois por volta de 1970 surgiram as primeiras publicações, mas o seu "boom" se deu nos anos 90, quando se tornou a principal metodologia de desenvolvimento de software. A orientação a objetos permite modelar de forma mais natural o mundo real, pois as estruturas de dados são vistas como objetos, ou seja, têm características e funções. Seu maior objetivo é aumentar a produtividade do desenvolvimento de software através de uma maior expansibilidade e reutilização de código, além de controlar a complexidade e o custo da manutenção do mesmo.

Quando a metodologia orientada a objetos é utilizada, a fase de projeto do desenvolvimento do software está mais intimamente ligada à fase de implementação. Um dos pontos chaves da metodologia orientada a objetos é centralização das atenções nas Estruturas de Dados, ao contrário da metodologia estruturada, onde a atenção era centralizada nos procedimentos. Na orientação a objetos há uma maior aproximação entre dados e procedimentos, pois procedimentos são definidos em termos dos dados.

Abstração de Dados

É o processo de criar novos tipos de dados, ou seja, a capacidade de abstrairmos algo, porém reconhecendo todas as suas propriedade básicas.

Classe

Tipo de dados definido pelo usuário (uma especificação para um tipo abstrato), que tem um estado (sua representação) e algumas operações (seu comportamento). Uma classe possui alguns dados internos e alguns métodos, na forma de procedimentos ou funções, e normalmente descreve as características genéricas de vários objetos semelhantes. O programador irá se preocupara no que a classe possui e **não** naquilo que ela significa.

Propriedade

Define as características dos objetos de uma classe.

Métodos

São os serviços oferecidos por uma classe, ou seja, são as funções criadas ou utilizadas pelo programador. Os métodos definem o comportamento dos objetos de uma classe.

Instâncias

Elementos de dados individuais são instâncias de uma classe. Instanciar uma classe significa **criar um objeto**.

Objeto

É uma instância de uma classe (da mesma forma que uma variável numérica nada mais é do que uma instância do tipo inteiro). Objeto é **uma variável do tipo de dados definido pela classe**. Os objetos são entidades reais, ocupam memória, quando o programa é executado. O relacionamento entre objeto e classe é o mesmo que o existente entre variável e tipo.

Exemplo:

```
unit Datas;

interface

type
    TData = Class(TObject)
      private
        Dia, Mes, Ano : Integer;
      public
       constructor Init (d,m,a : integer);
       procedure DefVal (d,m,a : integer);
       function  AnoBis : boolean;
       procedure Incrementa;
       procedure Decrementa;
       procedure Adiciona (NumDeDias : integer);
       procedure Subtrai (NumDeDias : integer);
      private
       function DiasNoMes : Integer;
    end;

implementation

constructor TData.Init (d,m,a : integer);
begin
  dia := d; Mes := m; ano := a;
end;

procedure TData.DefVal (d,m,a : integer);
begin
  dia := d; Mes := m; ano := a;
end;

function TData.AnoBis : boolean;
begin
  if (ano mod 4 <> 0) then AnoBis := false
  else
     if (ano mod 100 <> 0) then AnoBis := true
     else
        if (ano mod 400 <> 0) then AnoBis := False
        else AnoBis := True;
end;

function TData.DiasNoMes : integer;
begin
  case Mes of
    1,3,5,7,8,10,12 : DiasNoMes := 31;
    4,6,9,11        : DiasNoMes := 30;
    2               : if (AnoBis) then DiasNoMes := 29
                      else DiasNoMes := 28;
 end;
end;
```

Definição da Classe

Definição dos Métodos e Propriedades

```
procedure TData.Incrementa;
begin
if (dia < DiasNoMes) then inc(dia) {se não for o último dia do Mes}
  else
     if (Mes < 12) then
     begin
         inc(Mes);
         dia := 1;
     end
     else {se for o dia de ano novo}
     begin
         inc(ano);
         Mes := 1;
         dia := 1;
     end;
end;

procedure TData.Decrementa;
begin
  if (dia > 1) then Dec(dia) {se não for o primeiro dia do mês}
  else
     if (Mes > 1) then {se não for o primeiro dia do ano}
     begin
         Dec(Mes);
         dia := DiasNoMes;
     end
     else
     begin
         Dec(ano);
         Mes := 12;
         dia := DiasNoMes;
     end;
end;

procedure TData.Adiciona (NumDeDias : integer);
var n : integer;
begin
  for n := 1 to NumDeDias do
    Incrementa;
end;

procedure TData.Subtrai (NumDeDias : integer);
var n : integer;
begin
  for n := 1 to NumDeDias do
    Decrementa;
end;

end.
```

Utilização

```
var x : TData;
begin
    x:=TData.Create;
    x.Init(10,10,1981)
end;
```

Visibilidade

Define quem tem permissão de acessar e alterar os atributos e métodos da classe. Em uma mesma classe pode existir atributos e métodos com visibilidade diferentes.

Visibilidade	*Descrição*
Public	Os atributos e métodos podem ser manipulados por qualquer classe.
Private	Os atributos e métodos só podem ser manipulados pela própria classe.
Protected	Os atributos e métodos podem ser manipulados pela própria classe ou por qualquer subclasse desta e por demais classes declaradas na mesma UNIT.
Published	Semelhante a visibilidade public sendo que permite o acesso em tempo de projeto.

Declaração, Instanciamento, Destruição e Referência a Objetos

Um objeto é declarado da mesma maneira que uma variável. Para que um objeto possa ser utilizado, este deve ser instanciado e após o seu uso o mesmo deve ser liberado.

```
<Objeto> : Classe;              {Declarando}

<Objeto> = Classe.Create;       {Instanciando}

<Objeto>.Free;                  {Destruindo}

<Objeto>.Identificador          {Referenciando}
```

Identificador representa uma propriedade ou um método. A referência a Dados é idêntica a referência a código.

Método Construtor

```
constructor <identificador> ( <parâmetros formais> );
```

Aloca memória e inicializa o objeto, baseado nos parâmetros passados. Normalmente a primeira ação é invocar o construtor da classe base, através da instrução:

```
inherited <construtor> ( <parâmetros reais > );
```

Método Destrutor

```
destructor <identificador> ( <parâmetros formais> );
```

Destrói o objeto, baseado nos parâmetros passados, e libera a memória alocada para ele. Normalmente a última ação é invocar o destrutor da classe base, através da instrução:

```
inherited <destrutor> ( <parâmetros reais > );
```

O Parâmetro Self

Representa um parâmetro invisível passado a todos os métodos de uma classe e representa a instância da classe que esta chamando o método. É utilizado para evitar conflitos de nomes de objetos

Métodos Estáticos

Os métodos declarados numa classe são por default estáticos. Tem suas referências determinadas em tempo de compilação

Métodos Virtuais

O objetivo dos métodos virtuais é a possibilidade de substituí-los por novos métodos, contendo os mesmo parâmetros, das classes descendentes. Para tornar um método virtual, basta acrescentar no final de sua declaração na classe, a palavra virtual. Um método virtual pode ser substituído em uma classe descendente através de sua redeclaração seguida da diretiva override;

Métodos Dinâmicos

Métodos dinâmicos são basicamente idênticos a métodos virtuais sendo declarados com o uso da diretiva dynamic.

OBS.: Os métodos dinâmicos favorecem o tamanho do código enquanto os métodos virtuais favorecem a velocidade.

Métodos Abstratos

São métodos que não fazem nada, servem apenas para definir a estrutura de uma hierarquia. Para tornar um método abstrato, basta acrescentar no final de sua declaração na classe, a palavra abstract. Para um método ser abstrato, é necessário que ele seja também virtual.

Propriedades

Representa um mecanismo para encapsular os campos de uma Classe sobrecarregando as operações de leitura e escrita. São uma extensão natural às variáveis de instância de uma classe, pois permitem que o desenvolvedor pareça esta trabalhando com estas, enquanto na realidade está executando chamadas a métodos.

Para utilizar as propriedades os campos (atributos) devem ser declarados como private, os métodos como protected, e as propriedades como public.

```
property Identificador : TIPO
[read  MétodoDeLeitura]
[write MétodoDeEscrita];
```

Identificador representa a propriedade, TIPO o tipo da propriedade, MétodoDeLeitura o método associado à leitura da propriedade, MédodoDeEscrita o método associado à escrita da propriedade.

Verificação de Tipo

Verifica, em tempo de execução, se o objeto é uma instância da classe ou de alguma subclasse desta. Retorna true ou false.

```
<objeto> is <classe>
```

Conversão de Tipo

Converte, se possível, uma referência para o objeto de uma classe base em uma referência para objeto da subclasse.

```
(<objeto> as <classe>).<Métodos ou Propriedade>
```

Encapsulamento

A encapsulação consiste em ocultarmos ao usuário o funcionamento interno de uma classe. Desta forma todos os detalhes de um objeto, que são insignificantes ao programador não serão aparentes. A principal vantagem da encapsulação, é permitir que os implementadores de classes mudem a implementação de uma classe sem que precisem alterar algum código gerado. Desta forma podemos alterar os dados de um produto usando a mesma classe já previamente criada para a alteração de outro tipo de cadastro.

O principal objetivo é tornar o objeto independente de sua implementação interna, para isso a implementação das suas propriedades e métodos são "escondidas" de forma que o usuário precise apenas conhecer a interface do objeto para poder utilizá-lo. Desta forma o desenvolvedor poderá alterar tranquilamente a implementação de um objeto (Classe) sem causar transtornos aos usuários.

Herança

O objetivo básico deste conceito, é permitir a criação de uma nova classe a partir de outras existentes sem contudo duplicar código algum. A nova classe, chamada de derivada, se comportará de maneira análoga a classe que lhe deu origem, comportando-se diferentemente apenas naquilo que for alterado em relação à classe original.

A herança, é a responsável direta pela grande reusabilidade usual nos sistemas baseados em OOP. A herança permite que avancemos gradualmente em relação aos objetivos, sem perder consistências (alvos) anteriormente atingidos. Assim pode-se adaptar o comportamento de um componente sem mudar o próprio componente. O sistema pode ser alterado de forma incremental sem comprometimento do código.

Características / Benefícios

- Permite que uma nova classe seja descrita a partir de outra classe já existente (Reutilização).
- A subclasse herda as características e o comportamento da superclasse.
- A subclasse pode adicionar novas características e comportamentos aos herdados da superclasse.
- A subclasse pode ter um comportamento diferente da superclasse, redefinindo o método herdado.
- A subclasse é uma especialização da superclasse.
- Toda instância da subclasse é também uma instância da superclasse.
- Resultado de uma sequência de heranças é uma hierarquia de classes.

Implementação

Segue abaixo um exemplo de herança onde criamos uma subclasse (**TNewData**), a partir da classe **TData**, já existente, incluindo o método **GetText**.

```
type
    TNewData = Class(TData)
      public
       function  GetText : string;
    end;

function TData.GetText : string;
var d, m, a : string;
begin
  d := IntToStr(dia);
  case Mes of
    1 : m := 'Janeiro';
    2 : m := 'Fevereiro';
    3 : m := 'Março';
    4 : m := 'Abril';
    5 : m := 'Maio';
    6 : m := 'Junho';
    7 : m := 'Julho';
    8 : m := 'Agosto';
    9 : m := 'Setembro';
    10: m := 'Outubro';
    11: m := 'Novembro';
    12: m := 'Dezembro';
  end;
  a := IntToStr(ano);
  GetText := d + ', ' + m + ' de ' + a;
end;
```

Polimorfismo

Este conceito permite que sejam declaradas diferentes classes que por sua vez podem definir métodos de mesmo nome e comportamentos diferentes. Isto é importante na medida em que permite escrever rotinas que operem com qualquer classe que implementa os métodos necessários. Desta forma, as classes tornam-se mais independentes uma da outra e assim pode-se criar novas classes não gerando qualquer impacto sobre aquelas anteriormente existentes.

Características / Benefícios

- É a capacidade de tratarmos objetos de diferentes tipos de uma mesma maneira desde que eles tenham um ancestral em comum
- Objetos de classes diferentes podem ter métodos com mesmo nome e cada objeto responderá adequadamente de acordo com seu método.
- Métodos da mesma classe podem ter o mesmo nome, desde que possuam quantidade ou tipo de parâmetros diferentes.
- Métodos da classe derivada podem ter nomes iguais aos da classe base, inclusive com parâmetros iguais.

CAPITULO III - Object Pascal

Básico da Linguagem

Programa, procedures e funções são todos compostos de instruções que são, cada uma, uma linha de código executável. Instruções podem ser simples ou compostas. Veja alguns exemplos:

```
x := 13; //atribui o valor 13 à variável x.

y := x + 12; //atribui o resultado da soma à variável y.

TestaArquivo('config.sys'); //executa o procedimento TestaArquivo
```

Comentários:

{ } Tudo o que está entre as chaves é ignorado;

(* *) Tudo o que está entre * é ignorado;

// Tudo após o marcador do comentário é ignorado até o final da linha.

O Ponto-e-Vírgula e o Ponto:

Como regra geral, todas as instruções em um programa Pascal devem ser separadas por um ponto-e-vírgula. Neste caso, note que instrução refere-se tanto a instruções simples como instruções compostas. É por esta razão que não existe um ponto-e-vírgula após um BEGIN e significa também que um ponto-e-vírgula antes de qualquer END é opcional. Assim, no código a seguir, a instrução de atribuição não possui um ponto-e-vírgula após ela porque é imediatamente seguida por um END, mas o END deve ter um ponto-e-vírgula após, porque ele é o final da instrução composta:

```
if x = 13 then
begin
       ShowMessage('X vale Treze`);
       x:=14   // aqui o ponto e vírgula é opcional
end;  //aqui o ponto e vírgula é obrigatório pois termina a instrução composta.
```

→ Na maioria dos caso o par **BEGIN** ... **END** estão sempre juntos. Entretanto, existem poucos casos onde você tem um **END** sem **BEGIN**. A ocorrência mais comum é a instrução **CASE**.

→ O **END** final do programa é seguido de um ponto e não de um ponto e vírgula.

Tipos de Dados Padrão

Observe as tabelas abaixo com a descrição dos tipos de dados:

Tipos Inteiros

São tipos numéricos exatos, sem casas decimais. O tipo ***integer*** é o tipo inteiro padrão.

Tipo	Tam. (bytes)	Valor Mín.	Valor Máx.
ShortInt	1	-128	127
SmallInt ou **Integer**	2	-32768	32767
Longint	4	-2147483648	2147483647
Byte	1	0	255
Word	2	0	65535
Cardinal	4	0	2147483647

Tipos Reais

São tipos numéricos exatos com casas decimais. O tipo ***Double*** é o tipo real padrão.

Tipo	Tam. (bytes)	Valor Mín.	Valor Máx.	Dígitos Significativos
Real	6	10^{-39}	10^{38}	11-12
Single	4	10^{-45}	10^{38}	7-8
Double	8	10^{-324}	10^{308}	15-16
Extended	10	10^{-4932}	10^{4932}	19-20
Comp	8	-10^{18}	10^{18}	19-20
Currency	8	-10^{12}	10^{12}	19-20

Tipos Texto

Os tipos texto podem operar com caracteres simples ou grupos de caracteres. O tipo ***String*** é o tipo texto padrão.

Char	Um único caractere ASCII
String *ou* ***shortstring***	Conjunto de caracteres de tamanho 1 a 256 bytes, sendo que o primeiro byte guarda o comprimento do conjunto.
Ansistring	Conjunto de caracteres em limite de tamanho, é terminada em nulo (#0)

Tipos Ordinais

Tipos ordinais são tipos que tem uma sequência incremental, ou seja, pode-se dizer qual o próximo valor ou qual o valor anterior a um determinado valor ordinal. São tipos ordinais o Char, os tipos inteiros, o Boolean e os tipos enumerados.

Boolean

Variáveis do tipo Boolean podem receber os valores lógicos True ou False, verdadeiro ou falso. Uma variável Boolean ocupa 1 byte de memória.

```
x := true;
```

TDateTime

O tipo TDateTime guarda data e hora em uma estrutura interna igual ao tipo Double, onde a parte inteira é o número de dias desde 31/12/1899 e a parte decimal guarda a hora, minuto, segundo e milissegundo. As datas podem ser somadas ou subtraídas normalmente.

Variant

Tipo genérico, que pode atribuir e receber valores de qualquer outro tipo. Deve-se evitar a utilização de variáveis do tipo Variant, pois seu uso pode prejudicar a performance do programa, além de diminuir a legibilidade do código fonte e a integridade do executável, no trecho de código abaixo se observa como esse tipo de variável tem um comportamento estranho.

```
var
  V1, V2, V3: Variant;
begin
  V1 := True;
  V2 := 1234.5678;
  V3 := Date;
  ShowMessage(V1 + V2 + V3);
end;
```

Tipos de Definidos pelo Usuário

Na linguagem object pascal permite a declaração de tipos que não foram inicialmente disponibilizados pela linguagem de programação. Essas declarações são feitas na seção type, da interface ou implementation, sendo que na implementation esses tipos não poderão ser usados em outras Units.

Strings Limitadas

No caso de se desejar limitar o número de caracteres de uma string, podemos criar um tipo de string limitada:

```
TEstado = string[2];

x := 'RS';
```

Se, por exemplo, for tentado atribuir o valor: 'RGS' a uma variável do tipo TEstado, a mesma receberá apenas os dois primeiros caracteres ... RG.

Sub-Faixa

É um subconjunto de um tipo ordinal e possui as mesmas propriedades do tipo original:

```
TMaiusculas = 'A' .. 'Z';
TMes = 1 .. 12;

x := 'C';
y := 7;
```

Enumerações

Sequência de identificadores como valores válidos para o tipo. A cada elemento da lista de identificadores é associado internamente um número inteiro, iniciando pelo número 0.

```
TDiaSemana = (Dom, Seg, Ter, Qua, Qui, Sex, Sab);

x := Ter;  // podemos usar o comando CASE para testar atribuições
```

Ponteiros

Ponteiros armazenam endereços de memória. Todas as classes em Object Pascal são implementadas como ponteiros, mas raramente o programador vai precisar usá-los como tal.

```
TPonteiroInt = ^Integer;
```

Records

O tipo record é uma forma de criar uma única estrutura com valores de diferentes tipos de dados. Cada um dos dados de um record é chamado de campo.

```
type
 TData = record
  Ano: Integer;
  Mes: TMes;
  Dia: Byte;
 end;

var
  Festa: TData;
begin
  Festa.Ano := 1997;
  Festa.Mes := Mai;
  Festa.Dia := 8;
end;
```

Arrays

Arrays fornecem uma forma de criar variáveis que contenham múltiplos valores, como em uma lista ou tabela, cujos elementos são do mesmo tipo. Abaixo alguns exemplos de arrays de dimensões variadas.

```
TTempDia = array [1..24] of Integer; // uma dimensão - vetor
TTempMes = array [1..31, 1..24] of Integer; // duas dimensões - matriz
TTempAno = array [1..12, 1..31, 1..24] of Integer; // tridimensional
```

Um array pode ser definido como constante tipada, onde todos os seus elementos devem ser inicializados.

```
Fatorial: array[1..7] of Integer = (1, 2, 6, 24, 120, 720, 5040);
```

O tipo dos elementos de um array pode ser qualquer um, você pode ter uma array de objetos, de conjuntos, de qualquer tipo que quiser, até mesmo um array de arrays.

```
TTempMes = array [1..31] of TTempDia;
TBtnList = array [1..10] of TButton;
```

Sets

São conjuntos de dados de um mesmo tipo, sem ordem, como os conjuntos matemáticos. Conjuntos podem conter apenas valores ordinais, o menor que um elemento pode assumir é zero e o maior, 255.

```
TBorderIcons = set of TBorderIcon;

BorderIcons := [biSystemMenu, biMinimize];

if MesAtual in [Jul, Jan, Fev] then
  ShowMessage('Férias');
```

Elementos Fundamentais de Controle

Operadores

1. **Atribuição e Igualdade**: o operador de atribuição em Object Pascal é o símbolo “dois pontos igual” (:=). É diferente do sinal igual (=) usado para testa a igualdade e definição de tipos.

2. **Numéricos**: observe a tabela a seguir:

Operador	Operação	Tipos de Operandos	Tipos de Resultado
+	Adição	Integer, real	Integer, real
-	Subtração	Integer, real	Integer, real
*	Multiplicação	Integer, real	Integer, real
/	Divisão	Integer, real	Real, real
DIV	Divisão de Inteiros	Integer	Integer
MOD	Resto de Divisão	Integer	Integer

3. **Relacionais**: a tabela a seguir contém os operadores relacionais que retornam dois valores booleanos (true e false):

Operador	Operação
=	Igual
<>	Não Igual
<	Menor que
>	Maior que
<=	Menor ou igual
>=	Maior ou igual
NOT	Negação
AND	E lógico
OR	OU lógico

Comandos de DECISÃO

IF ... THEN ... ELSE ... (SE ... ENTÃO ... SENÃO)

O formato básico da instrução IF ... THEN é mostrado abaixo:

```
if <expressão lógica> then
       <instrução>;
```

Esta sintaxe básica pode ser incrementada pela adição da cláusula ELSE:

```
if <expressão lógica> then
       <instrução>  // não vai ponto-e-vírgula
else
       <instrução>;
```

Para utilizar instruções compostas, a sintaxe básica deve ser incrementada pelos comandos de bloco de código:

```
if <expressão lógica> then
begin
       <instrução 1>;
       <instrução 2>;
       <instrução 3>;
end   // não vai ponto-e-vírgula
else
begin
       <instrução 1>;
       <instrução 2>;
       <instrução 3>;
end;
```

Obs.: Não existe ponto-e-vírgula após o **END** que aparece antes do **ELSE**;

CASE ... OF (ESCOLHA ... DE)

A instrução IF ... THEN ... ELSE mostrada anteriormente funciona bem para uma pouca quantidade de opções, mas torna-se inviável se você tiver muitas mais. Nesta situação, a construção CASE é mais fácil para se escrever. A sintaxe geral para a instrução CASE é:

```
case <variável> of
       <valor 1> : <instrução 1>;
       <valor 2> : <instrução 2>;
       <valor n> : <instrução n>;
else
       <instrução>;
end;
```

A instrução CASE toma uma variável e a compara com os valores possíveis. Múltiplos valores podem aparecer sob o mesmo caso, separados por vírgulas. Observe:

```
case x of
       'A' .. 'Z', 'a' .. 'z' : ShowMessage('Letra');
       '0' .. '9' : ShowMessage('Número');
       '+', '-', '*', '/' : ShowMessage('Operador');
else
       ShowMessage('Caractere Especial');
end;
```

Comandos de LOOPING

WHILE ... DO (ENQUANTO ... FAÇA)

O Loop while executa instruções até que a condição do loop torne-se falsa. Observe sua sintaxe:

```
while <expressão lógica> do
       <instrução>;
```

Como nas outras estruturas, aqui você também pode ter uma instrução simples ou composta após o DO, basta para isso utilizar o par BEGIN END. Observe:

```
i:=10;
while i > 0 do
begin
       i:=i-1;
       ShowMessage('Passando pelo loop ' + IntToStr(i));
end;
```

FOR ... DO (DE ... FAÇA)

O Loop FOR difere do while por executar uma quantidade específica de vezes a ação, determinada pela variável de controle, sua sintaxe básica é:

```
For <variável de controle> := <valor inicial> to <valor final> do
       <instrução>;
```

Usando a cláusula DOWNTO ao invés de TO, a variável de controle sofre um decremento de valor. Observe os exemplos:

```
For i:=10 to 100 do
    ShowMessage(IntToStr(i));
```

```
For i:=100 downto 10 do
    ShowMessage(IntToStr(i));
```

REPEAT ... UNTIL (REPITA ... ATÉ QUE)

A terceira construção de loop é o REPEAT ... UNTIL, ele executa todas as instruções dentro do loop até que a expressão lógica torne-se verdadeira (o contrário do WHILE). Observe a sintaxe geral:

```
repeat
       <instrução 1>;
       <instrução 2>;
       <instrução n>;
until <expressão lógica>;
```

Note que essa construção pode conter diversas instruções e não requer o par BEGIN .. END. Observe o exemplo:

```
j:=0;
repeat
       j:=j+1;
       ShowMessage('Looping');
until j>= 10;
```

BREAK e CONTINUE (PARE e CONTINUE)

São palavras reservadas que podem ser usadas em qualquer um dos laços de repetição. Observe o exemplo:

```
while true do
begin
       readln(inputFile,i);
       if strtoint(i) = 5 then
             break;
       ShowMessage(i);
end;
```

Neste exemplo, o loop é executado eternamente (uma vez eu true é sempre true). O loop continuará a ler a entrada de dados de um arquivo e exibirá um caixa de mensagem até que seja lido o número 5, pois o comando BREAK interrompera o laço.

O outro modificador de execução do loop é a instrução CONTINUE. Ela ignorará o restante do loop e voltará a avaliar a condição. Assim, se algo for alterado que altere a condição, você pode utilizar CONTINUE para pular para o topo do loop e avaliar novamente a condição. Observe:

```
while i<10 do
begin
       readln(inputFile,i);
       if strtoint(i) < 5 then
            continue;
       ShowMessage('Este número será maior que 5:' + i);
end;
```

Neste caso, CONTINUE impedirá que qualquer número menor que 5 seja impresso. Este comando é útil para ser usado com entradas inválidas dentro de um loop.

Procedimentos e Funções

Procedimentos e funções são blocos básicos de programas Object Pascal. Para tornar o programa mais fácil de lidar, é melhor quebrar a funcionalidade em blocos ao invés de enormes programas monolíticos, onde o código reside no arquivo de programa principal.

PROCEDURE

Procedures são as sub-rotinas mais comuns em Object Pascal, para criar módulos deste tipo, observe a sintaxe:

```
procedure NomeDaProcedure(<parâmetro 1> : <tipo 1>; <parâmetro n> : <tipo 2>);
const
       <nome da constante> =  <valor literal>;
var
       <nome da variável> : <tipo>;
begin
       <instrução 1>
       <instrução n>
end;
```

Observe um exemplo de declaração de procedure:

```
procedure MostraEstrelas(NumEstrelas : integer);
var
       i : integer;
       s : string;
begin
       for i:=1 to NumEstrelas do
            s:= s + '*';

       ShowMessage(s);
end;
```

Neste exemplo, uma caixa de mensagem com ‘n’ asteriscos será exibida. A quantidade de asteriscos é passada como parâmetro quando a procedure é chamada, sendo assim para exibir 20 asteriscos faz-se a seguinte chamada:

```
MostraEstrelas(20);
```

A passagem de parâmetros em um procedimento não é obrigatória, a seguir um exemplo:

```
procedure Mostra10Estrelas;
const   NumEstrelas = 10;
var     i : integer;
        s : string;
begin
        for i:=1 to NumEstrelas do
              s:= s + ‘*’;

        ShowMessage(s);
end;
```

Neste caso, para fazer a chamada do procedimento, utiliza-se o nome do mesmo: `Mostra10Estrelas;`

FUNCTION

Funções são basicamente o mesmo que procedimentos, porém com uma grande diferença: a função retorna um valor de algum tipo. Para tanto, ao declarar uma função, declara-se também qual o tipo de valor que ela retorna. A sintaxe básica de uma função é a seguinte:

```
function NomeDaFuncao(<param. 1> : <tipo>; <param. n> : <tipo>): <tipo retorno>;
const
       <nome da constante> =  <valor literal>;
var
       <nome da variável> : <tipo>;
begin
       <instrução 1>
       <instrução n>
end;
```

O valor de retorno de uma função pode ser especificado de duas formas: fazendo a atribuição diretamente ao nome da função:

```
function Cubo(base : integer): integer;
begin
       Cubo:=base*base*base;
end;
```

Ou usando a palavra reservada **RESULT** dentro da função para receber o resultado de retorno:

```
function Quadrado(base : integer): integer;
begin
       Result:=base*base;
end;
```

Quando a função é chamada, o valor de retorno deve ser atribuído a uma variável de mesmo tipo:

```
       x:=Cubo(2);  // x será igual a 8
       y:=Quadrado(3);  // y será igual a 9
```

Funções e Comandos Internos (principais)

Tratamento de Strings

- **CompareText(**texto1, texto2**)**
  Compara 2 strings sem sensitividade de maiúsculo-minúsculas e devolve um valor inteiro. Se este valor for menor que zero, o **texto1** é menor que o **texto2**, caso o valor for maior que zero, o **texto2** é menor que o **texto1** e os dois são iguais se o valor de retorno for zero.
  Ex.:
```
x:=CompareText('Casa','casa'); // x = 0
x:=CompareText('apartamento','casa'); // x < 0
x:=CompareText('casa','apartamento'); // x > 0
```

- **CompareStr(**texto1, texto2**)**
  Compara 2 strings com sensitividade de maiúsculo-minúsculas e devolve um valor inteiro. Se este valor for menor que zero, o **texto1** é menor que o **texto2**, caso o valor for maior que zero, o **texto2** é menor que o **texto1** e os dois são iguais se o valor de retorno for zero.
  Ex.:
```
x:=CompareStr('Casa','casa'); // x < 0
x:=CompareStr('casa','casa'); // x = 0
x:=CompareStr('casa','apartamento'); // x > 1
```

- **LowerCase(**texto**)**
  Converte todas as letras do texto passado como parâmetro para minúsculas.
  Ex.:
```
y:=LowerText('CASA'); // y = 'casa'
```

- **UpperCase(**texto**)**
  Converte todas as letras do texto passado como parâmetro para maiúsculas.
  Ex.:
```
y:=UpperText('casa'); // y = 'CASA'
```

- **Copy(**texto, pos_inicial, num_posições**)**
  Retorna um subtexto de um texto passado como parâmetro, de acordo com a posição e a quantidade de caracteres predefinidos.
  Ex.:
```
y:=Copy('Universidade Pública',14,7); // y = 'Pública'
```

- **Delete(**variável, pos_inicial, num_posições**)**
  Remove um número determinado de caracteres a partir de uma posição inicial.
  Ex.:
```
y:='Universidade Pública';
Delete(y,7,6); // y = 'Univer Pública'
```

- **Insert(**texto, variavel_destino, posição**)**
  Insere um subtexto em um texto, em uma posição preestabelecida.
  Ex.:
```
y:='Universidade Pública';
Insert('UFSM – ',y,1); // y = 'UFSM - Universidade Pública'
```

- **Length(**texto**)**
  Retorna o número de caracteres do texto passado como parâmetro.
  Ex.:
```
x:=length('UFSM'); // x = 4
```

- **Pos(**texto_pesquisado, texto_objeto**)**
  Retorna a posição do texto pesquisado dentro do texto objeto da pesquisa. Se a função retornar zero, o texto pesquisado não existe no texto objeto
  Ex.:
```
x:=Pos('S','UFSM'); // x = 3
```

- **Trim(**texto**)**
  Remove os espaços em branco à direita e à esquerda do texto passado como parâmetro
  Ex.: `y:=Trim('    FRED. WESTPHALEN – RS   '); // y:='FRED. WESTPHALEN – RS'`

- **TrimLeft(**texto**)**
  Remove os espaços em branco à esquerda do texto passado como parâmetro
  Ex.:
```
y:=Trim('    FRED. WESTPHALEN   '); // y:= 'FRED. WESTPHALEN   '
```

- **TrimRight(**texto**)**
  Remove os espaços em branco à direita do texto passado como parâmetro
  Ex.: `y:=Trim('    FRED. WESTPHALEN   '); // y:= '    FRED. WESTPHALEN'`

Tratamento de Ordinais

- **Dec(**ordinal, quantidade**)**
  Decrementa um valor ordinal em uma quantidade determinada de unidades;
  Ex.:
  ```
  x:=10;
  Dec(x); // x = 9
  Dec(x,6); // x = 4
  ```

- **Inc(**ordinal, quantidade**)**
  Incrementa um valor ordinal em uma quantidade determinada de unidades;
  Ex.:
  ```
  x:=10;
  Inc(x); // x = 11
  Inc(x,6); // x = 14
  ```

- **Odd(**ordinal**)**
  Testa se o ordinal passado como parâmetro é impar (neste caso retorna true), ou se é par (retornando false).
  Ex.:
  ```
  z:=Odd(10); // z = false
  z:=Odd(5); // z = true;
  ```

- **Pred(**ordinal**)**
  Retorna o predecessor do ordinal;
  Ex.: `x:=Pred('b'); // x = 'a';`

- **Succ(**ordinal**)**
  Retorna o sucessor do ordinal;
  Ex.: `x:=Succ('b'); // x = 'c';`

- **Ord(**ordinal**)**
  Retorna a ordem do valor ordinal na faixa de valores do tipo a qual ele faz parte;
  Ex.:
  ```
  x:=Ord('A'); // x = 65 (valor de A na tab. ASCII);
  z:=Odd(5); // x =5
  ```

- **Low(**ordinal**)**
  Valor mais baixo da faixa de valores do tipo a qual o ordinal faz parte;
  Ex.:
  ```
  x:=10; // inteiro;
  x:=Low(x); // x = -2147483648
  ```

- **High(**ordinal**)**
  Valor mais alto da faixa de valores do tipo a qual o ordinal faz parte;
  Ex.:
  ```
  x:=10; // inteiro;
  x:=Low(x); // x = 2147483647
  ```

Tratamento de valores TDateTime

- **Date**
  Retorna a data atual do sistema, no formato preestabelecido no painel de controle.
  Ex.: `x:=date; // x = 13/03/2001`

- **Time**
  Retorna a hora atual do sistema, no formato preestabelecido no painel de controle.
  Ex.: `x:=time; // x = 09:41:57`

- **Now**
  Retorna a data e a hora atual do sistema.
  Ex.: `x:=now; // x = 13/03/2001 09:45:57`

- **DayOfWeek(**data**)**
  Retorna o dia da semana de uma data especificada
  Ex.: `x:=DayOfWeek(13/03/2001); // x = 3 ... terça-feira`

- **DecodeDate(**DateTime, ano, mês, dia**)**
  Decodifica um valor TDateTime em três variáveis word ... dia, mês e ano.
  Ex.:
```
DecodeDate(now,a,m,d);  // a = 2001
                        // m = 03
                        // d = 12
```

- **DecodeTime(**DateTime, hora, minutos, segundos, milissegundos**)**
  Decodifica um valor TDateTime em quatro variáveis word ... hora, minutos, segundos, milissegundos.
  Ex.:
```
DecodeTime(now,h,m,s,ms);     // h  = 09
                              // m  = 45
                              // s  = 57
                              // ms = 23
```

- **EncodeDate(**ano, mês, dia**)**
  Retorna uma data a partir de três variáveis word ... dia, mês e ano.
  Ex.: `x:=EncodeDate(a,m,d);  // x  = 13/03/2001`

- **EncodeTime(**hora, minutos, segundos, milissegundos**)**
  Retorna o valor do horário a partir de quatro variáveis word ... hora, minutos, segundos, milissegundos.
  Ex.: `x:=EncodeTime(h,m,s,ms);  // x  = 09:45:57`

Conversões de Tipos de Dados

A tabela abaixo ilustra algumas conversões possíveis entre tipos de dados:

Orig. /Dest.	Byte	Char	String	Integer	Real	DateTime	Date	Time
		Chr	Chr	**direto***	**direto***			
Char	Ord		**direto***	Ord	Ord			
String	StrToInt	Variável[índice]		StrToInt	StrToFloat	StrToDateTime	StrToDate	StrToTime
Integer	**direto***		IntToStr		**direto***			
Real	Round / Trunc		FloatToStr	Round / Trunc				
DateTime			DateTimeToStr					
Date			DateToStr					
Time			TimeToStr					

As células com fundo preto representam conversões inexistentes. As conversões diretas, como Integer para Byte levam em consideração que o tamanho do dado da origem não é maior que o máximo suportado pelo dado de destino.

- **Chr(**var_byte**)**
  Converte uma variável Byte em Char. É utilizada para saber o caractere ASCII de um valor.
  Ex.: `x:=Chr(65); // x = 'A'`

- **StrToInt(**var_string**)**
  Converte uma variável String em Integer.
  Ex.: `x:=StrToInt('10');  // x = 10`

- **IntToStr(**var_integer**)**
  Converte uma variável Integer em String.
  Ex.: `x:=IntToStr(10);  // x = '10'`

- **StrToIntDef(**var_string, valor_padrão**)**
  Converte uma variável String em Integer e em caso de erro atribui o valor padrão (inteiro)
  Ex.:
```
x:=StrToIntDef('abc',0);      // x  = 0
x:=StrToIntDef('112',0);      // x  = 112
```

- **IntToHex(**var_integer, num_casas**)**
  Converte uma variável Integer em um valor String Hexadecimal
  Ex.:
```
x:= IntToHex(1100,5)    // x  = '0044C'
```

- **Round(**var_real**)**
  Arredonda um número real em um Integer.
  Ex.:
```
x:= Round(50.8)   // x  = 51  ... maior que 5
x:= Round(50.5)   // x  = 50  ... menor ou igual a 5
```

- **Trunc(**var_real**)**
  Trunca um número real em um Integer, pega a parte inteira do valor real.
  Ex.:
```
x:= Trunc(50.8)   // x  = 50
x:= Trunc(50.5)   // x  = 50
```

- **StrToFloat(**var_string**)**
  Converte uma variável String em Real.
  Ex.:
```
x:=StrToInt('10.12');   // x  = 10,12
```

- **FloatToStr(**var_real**)**
  Converte uma variável Real em String.
  Ex.:
```
x:=IntToStr(10.12);     // x  = '10,12'
```

- **DateToStr(**data**)**
  Converte uma variável do tipo TDateTime em string de data, de acordo com as configurações regionais do sistema operacional.
  Ex.:
```
x:=DateToStr(date);     // x  = '13/03/2001'
```

- **StrToDate(**var_string**)**
  Converte uma variável do tipo string de data em TDateTime.
  Ex.:
```
x:=StrToDate('13/03/2001');   // x  = 13/03/2001
```

- **TimeToStr(**hora**)**
  Converte uma variável do tipo TDateTime em string de hora.
  Ex.:
```
x:=TimeToStr(time);     // x  = '09:45:57'
```

- **StrToTime(**var_string**)**
  Converte uma variável do tipo string de hora em TDateTime.
  Ex.:
```
x:=StrToTime('09:45:57');     // x  = 09:45:57
```

- **DateTimeToStr(**hora**)**
  Converte uma variável do tipo TDateTime em string de data e hora.
  Ex.:
```
x:=DateTimeToStr(time); // x  = '13/03/2001 09:45:57'
```

- **StrToDateTime(**var_string**)**
  Converte uma variável do tipo string de data e hora em TDateTime.
  Ex.:
```
x:=StrToDateTime('13/03/2001 09:45:57'); // x  = 13/03/2001 09:45:57
```

- **Val(**texto, variavel_destino, var_erro**)**
  Converte uma string passada como parâmetro em um valor numérico. Caso o conteúdo da string não seja numérico, a variável de erro estará armazenando um número diferente de zero.
  Ex.:
```
Val('teste', x, erro); // x  = 0    erro = 1
Val('100', x, erro);   // x  = 100  erro = 0
```

- **Str(**valor, variavel_destino**)**
  Comando que converte um valor numérico em string e armazena o resultado em uma variável destino.
  Ex.:
```
Str(950.25:3:2,x);      // x  = '950.25'
```

Strings de Formato

Algumas funções permitem criar máscaras de saída a partir de alguns códigos, alguns exemplos:

- **Format(**texto_p_formatar,[argumentos]**)**
  Formata uma string com uma série de argumentos de vários tipos:

  d – Número Decimal (inteiros); **e** – Número Científico
  f – Número Fixado **g** – Número Geral
  m – Formato Dinheiro (money) **s** – String
  x – Hexadecimal (existem outros menos utilizados)

  Ex.:

```
Format('O numero %d é a parte inteira do número %f.', [10,10.5]);
   Resultado: 'O numero 10 é a parte inteira do número 10,50.'

Format('Este texto %s foi formatado %s usando o caractere #%d.',[#13,#13,13]);
                                Resultado:

                                Este texto
                               foi formatado
                            usando o caractere #13

Format('O preço do livro %s é %m.',['Como programar em Pascal',50.7]);
   Resultado: 'O preço do livro Como Programar em Pascal é R$ 50,70'

Format('O numero %d equivale ao valor %x em hexadecimal',[10,10]);
   Resultado: 'O número 10 eqüivale ao valor A em hexadecimal'
```

- **FormatFloat(**texto_p_formatar,[argumentos]**)**
  Converte um número real em string, utilizando uma string de formato com as seguintes especificações:

  0 – Dígitos obrigatórios **,** – Separador de milhar
  # – Dígitos opcionais **E –** Notação Científica
  . – Ponto flutuante

  Ex.: `x:= FormatFloat('###,##0.00',1210.25)  // x = 1.210,25`

  Observe mais algumas máscaras e seus respectivos resultados

Format string	1234	-1234	0.5	0
	1234	-1234	0.5	0
0	1234	-1234	1	0
0.00	1234,00	-1234,00	0,50	0,00
#.##	1234	-1234	,5	
#,##0.00	1.234,00	-1.234,00	0,50	0,00
#,##0.00;(#,##0.00)	1.234,00	(1.234,00)	0,50	0,00
#,##0.00;;Zero	1.234,00	-1.234,00	0,50	Zero
0.000E+00	1,234E+03	-1,234E+03	5,000E-01	0,000E+00
#.###E-0	1,234E3	-1,234E3	5E-1	0E0

Outras Funções

As bibliotecas do Lazarus e Free Pascal implementam ainda diversas outras funções veja mais alguns exemplos úteis:

- **Int(**var_real**)**
  Retorna a parte inteira do argumento real e devolve o valor como real também
  Ex.:
```
x:= Int(50.8)     // x  = 50,0
x:= Int(50.5)     // x  = 50,0
```

- **Frac(**var_real**)**
  Retorna a parte fracionária do argumento real e devolve o valor como real também
  Ex.:
```
x:= Frac(14.56)   // x  = 0,56
```

- **Abs(**numero**)**
  Fornece o valor absoluto do argumento numérico. O resultado pode ser inteiro ou real.
  Ex.:
```
x:=Abs(-4.21);    // x  = 4,21
```

- **Pi**
  Fornece o valor do π, com 14 casas decimais.
  Ex.:
```
x:=Pi;      // x  = 3,14159265358979
```

- **Sqr(**numero**)**
  Eleva o valor passado como parâmetro ao quadrado e retorna um valor real.
  Ex.:
```
x:=Sqr(5);  // x  = 25
```

- **Sqrt(**numero**)**
  Fornece a raiz quadrada do valor passado como parâmetro e retorna um valor real.
  Ex.:
```
x:=Sqrt(9); // x  = 3
```

- **ExtractFileName(**caminho**)**
  Extrai o nome do arquivo da string passada como parâmetro.
  Ex.:
```
x:=ExtractFileName('c:\temp\cafw.exe');   // x  = 'cafw.exe'
```

- **ExtractFilePath(**caminho**)**
  Extrai o caminho do arquivo da string passada como parâmetro.
  Ex.:
```
x:=ExtractFilePath('c:\temp\cafw.exe');   // x  = 'c:\temp\'
```

- **ExtractFileDrive(**caminho**)**
  Extrai o drive do caminho da string passada como parâmetro.
  Ex.:
```
x:=ExtractFilePath('c:\temp\cafw.exe');   // x  = 'c:\'
```

- **ExtractFileExt(**caminho**)**
  Extrai a extensão do arquivo da string passada como parâmetro.
  Ex.:
```
x:=ExtractFilePath('c:\temp\cafw.exe');   // x  = '.exe'
```

- **FileExists (**caminho**)**
  Testa se o arquivo especificado no caminho existe.
  Ex.:
```
x:=FileExists('c:\command.com');    // x  = true
```

- **Random -**
  Retorna um número (pseudo)-randômico. A sequência de valores randômicos sempre será a mesma.

- **Randomize**
  É utilizado antes do comando **Random**, para forçar o uso do relógio e daí sim gerar número randômicos em ordem que não se repete.

- **Sleep(**tempo**)**
  Efetua uma pausa nas operações. O tempo de duração da pausa é determinado pelo parâmetro tempo. Os valores informados para a pausa são em milissegundos.

- **Beep**
  Procedimento que emite o som padrão do sistema operacional definido no painel de controle.

CAPITULO IV – Classes e Objetos Padrões do Lazarus

Propriedades

As propriedades são características dos componentes. Para alterar propriedades em código use a sintaxe de ponto, observe:

```
Objeto.Propriedade := valor;
```

Tipos de Propriedade

- **Tipo String**
```
Button1.Caption := 'Fechar';
Label1.Caption := Edit1.Text + '/' + Edit2.Text;
```

- **Tipo Numérico**
```
Button2.Height := Button2.Height * 2;
Width := Button1.Width + Button2.Width + 12;
```

- **Tipo Enumerado**
```
BorderStyle := bsDialog;
Panel1.Color := clWindow;
```

- **Propriedades Aninhadas de Classe**
```
Memo1.Lines.Text := 'E agora, José?';
Label1.Font.Color := clBlue;
```

- **Propriedades Aninhadas de Conjunto**
```
BorderIcons := [biSystemMenu, biMaximize];
Label1.Font.Style := [fsBold, fsItalic];
```

Propriedades Comuns

Propriedade	Descrição
Align	Determina o alinhamento do componente
Canvas	Superfície de desenho, do tipo TCanvas, onde pode se desenhar a imagem do componente
Caption	Legenda do componente (& indica tecla de atalho para alguns componentes)
Color	Cor do componente
ComponentCount	O número de componentes possuídos
Components	Matriz de componentes possuídos
Enabled	Define se o componente está ativo, se pode ser usado
Font	Fonte utilizada no componente
Height	Altura
HelpContext	Número utilizado para chamar o Help on-line
Hint	String utilizada em dicas instantâneas
Left	Posição esquerda
Name	Nome do componente
PopupMenu	Menu de contexto do componente
ShowHint	Define se o Hint será mostrado
TabOrder	A ordem de tabulação do componente, usada quando o usuário tecla TAB
TabStop	Indica se o componente será selecionado quando o usuário teclar TAB
Tag	Propriedade não utilizada pelo Lazarus, que pode ser usada como propriedade personalizada
Top	Posição superior
Visible	Define se o componente está visível
Width	Largura

Eventos

Os Eventos acontecem em resposta a uma ação do usuário ou do próprio sistema, ao programar um método de evento, deve-se considerar que o código só será executado quando o evento acontecer. Uma das tarefas mais importantes na programação baseada em eventos é determinar quais eventos serão usados e qual a ordem desses eventos, por exemplo, quando o usuário clicar em um botão, qual evento acontecerá primeiro, OnEnter, OnMouseDown ou OnClick?

Os códigos dos eventos podem ser compartilhados entre componentes, dessa Forma, você pode ter um botão na barra de ferramentas que faz a mesma coisa que uma opção de menu. Para isso, basta escolher o evento na lista em vez de clicar duas vezes no Object Inspector.

É permitido mudar os métodos de evento através de código, uma vez que os eventos também são propriedades e podem ser usados como tal. Pode-se atribuir um evento de outro componente ou diretamente o nome do método:

```
Button1.OnClick := Edit1.OnExit;
Button2.OnClick := Edit2Click;
```

Eventos Comuns

Evento	Descrição
OnChange	O conteúdo do componente é alterado
OnClick	O componente é acionado
OnDblClick	Duplo-clique no componente
OnEnter	O componente recebe o foco
OnExit	O componente perde o foco
OnKeyDown	Tecla pressionada
OnKeyPress	Uma tecla é pressionada e solta
OnKeyUp	Tecla é solta

Métodos

Os métodos realizam ações definidas pelo componente, veja os exemplos abaixo e atente para os parâmetros passados. Note que podemos chamar os métodos de evento como qualquer outro método e que os métodos de evento pertencem ao Form, não aos componentes.

```
Edit1.Clear;
Form2.Show;
Close;
ScaleBy(110, 100);
Button1.ScrollBy(10, 10);
Button1.OnClick(Sender);
Button1Click(Self);
Form2.Button1Click(Sender);
```

Métodos Comuns

Método	Descrição
Create	Cria um novo Objeto de uma Classe
Free	Destrói um Objeto e libera a memória ocupada por ele
Show	Torna o componente visível
Hide	Torna o componente invisível
SetFocus	Coloca o foco no componente
Focused	Determina se o componente tem o foco
BringToFront	Coloca o componente na frente dos outros
SendToBack	Coloca o componente atrás dos outros
ScrollBy	Move o componente
SetBounds	Muda a posição e o tamanho do componente

Biblioteca de Classes

Janelas

Janelas são os elementos básicos no desenvolvimento sobre o qual um aplicativo é construído no Lazarus. O tipo TForm é usado pelo Lazarus como classe base para todas as janelas, abaixo algumas propriedades, eventos e métodos dessa classe.

Propriedade	**Descrição**
Active	Indica se o Form está ativo
ActiveControl	Determina o controle que receberá o foco por default
AutoScroll	Adiciona barras de rolagem automaticamente, quando necessário
BorderIcons	Define quais ícones de controle serão visíveis, quais botões vão aparecer na barra de título
BorderStyle	Estilo da borda do Form
FormStyle	Tipo de Form, normal, MDI pai, MDI filho ou sempre visível
Icon	Ícone do Form
Menu	Indica qual o menu do Form
Position	Permite controlar a posição e tamanho do Form na exibição
WindowState	Estado do Form, maximizada, minimizada ou normal
Evento	**Descrição**
OnCreate	Quando o Form é instanciado
OnDestroy	Quando o Form é liberado da memória
OnShow	Exatamente antes de mostrar o Form
OnCloseQuery	É chamada para validar se o Form pode ser fechado
OnClose	Quando o Form é fechado
OnActivate	Quando o Form recebe o foco
OnDeactivate	Quando o Form perde o foco
OnResize	Quando o Form muda de tamanho
Método	**Descrição**
ShowModal	Ativa o Form modal, que o usuário tem que fechar para poder continuar a usar a aplicação
Show	Mostra o Form
Close	Fecha o Form

Componentes Padrão

- **TButton**
  Botão padrão para executar ações.

Propriedade	**Descrição**
Cancel	Dispara o evento OnClick do botão quando a tecla ESC é pressionada em qualquer controle
Default	Dispara o evento OnClick do botão quando a tecla ENTER é pressionada em qualquer controle
ModalResult	Associa o botão a opção de fechamento de um Form modal
Método	**Descrição**
Click	Ativa o evento OnClick do botão

- **TBitBtn**
  Botão especializado, com Bitmap.

Propriedade	**Descrição**
Glyph	Bitmap exibido pelo botão
LayOut	Posição do Bitmap no Botão
Margin	Indica o espaço entre a borda do botão e o Bitmap
Spacing	Indica o espaço entre o Bitmap e o texto do botão
Kind	Seleciona um tipo padrão para o botão, mudando várias propriedades, como Glyph e ModalResult

- **TSpeedButton**
  Botão com Bitmap, normalmente utilizado em barras de ferramentas.

Propriedade	**Descrição**
Down	Estado do botão (Pressionado ou não)
GroupIndex	Indica quais botões pertencerão ao mesmo grupo
AllowAllUp	Permite que todos os botões de um grupo possam ficar não pressionados
Flat	Define se a borda do botão deve aparecer apenas quando ele for apontado

- **TLabel**
  Utilizado para exibir rótulos

Propriedade	**Descrição**
Alignment	Alinhamento do texto no componente
AutoSize	Define se o tamanho do componente será automaticamente ajustado ao tamanho do Caption
WordWrap	Retorno automático de linha
Transparent	Define se o componente será transparente
FocusControl	Componente que receberá o foco quando a tecla de atalho do Caption (&) for pressionada
ShowAccelChar	Indica se o caractere & será usado para definir tecla de atalho

- **TEdit**
  Utilizado para entrada de texto em uma única linha.

Propriedade	**Descrição**
Text	Texto do componente
AutoSelect	Indica se o texto será ou não selecionado quando o componente receber o foco
MaxLength	Número máximo de caracteres permitidos
CharCase	Define se as letras aparecerão em maiúsculo, minúsculo ou normal
PasswordChar	Caractere utilizado para esconder o texto digitado (Senhas)
ReadOnly	Define se será permitido alterar o texto
Método	**Descrição**
Clear	Limpa o conteúdo do componente
ClearSelection	Limpa o texto selecionado no componente

- **TMemo**
  Permite entrada de dados texto em múltiplas linhas. Contém propriedades e métodos do TEdit.

Propriedade	**Descrição**
Lines	Propriedade do tipo TStrings que armazena as linhas de texto do componente
WantReturns	Define se a tecla ENTER será tratada como quebra de linha
WantTabs	Define se a tecla TAB será tratada como espaço de tabulação
ScrollBar	Define as barras de rolagem

- **TMaskEdit**
  Permite entrada de dados texto em uma linha, utilizando uma máscara de edição. Possui todas as propriedades do componente TEdit.

Propriedade	**Descrição**
EditMask	Máscara de edição

Máscaras

Uma máscara é composta por três partes, a primeira parte é a máscara propriamente dita, a segunda parte indica se os caracteres literais serão salvos e a terceira parte indica qual o caractere utilizado para representar os espaços a serem digitados no texto.

Estes são os caracteres especiais que podem compor a máscara de edição:

Caractere	Descrição
!	Espaços em branco não serão considerados no texto
>	Todos os caracteres seguintes serão maiúsculos até que apareça o caractere <
<	Todos os caracteres seguintes serão minúsculos até que apareça o caractere >
\	Indica um caractere literal
L	Somente caractere alfabético
L	Obrigatoriamente um caractere alfabético
A	Somente caractere alfanumérico
A	Obrigatoriamente caractere alfanumérico
9	Somente caractere numérico
0	Obrigatoriamente caractere numérico
C	Permite um caractere
C	Obrigatoriamente um caractere
#	Permite um caractere numérico ou sinal de mais ou de menos, mas não os requer.
:	Separador de horas, minutos e segundos
/	Separador de dias, meses e anos

Alguns exemplos de máscaras:

!90/90/00;1;_ Máscara de data. Ex.: 10/10/1981
!90:00;1;_ Máscara de hora. Ex.: 12:00
!\(999\)\ 000-0000;1;_ Máscara de Telefone. Ex.: (055) 332-7100
00000\-999;1;_ Cep - Brasil

- **TStrings**

Muitos componentes, como o TMemo, possuem propriedades do tipo TStrings, essa classe permite armazenar e manipular uma lista de Strings. Toda propriedade do tipo TStrings permite acesso indexado aos itens da lista.

Propriedade	**Descrição**
Count	Número de strings
Text	Conteúdo do memo na Forma de uma única string
Método	**Descrição**
Add	Adiciona uma nova string no final da lista
Insert	Insere uma nova string numa posição especificada
Move	Move uma string de um lugar para outro
Delete	Apaga uma string
Clear	Apaga toda a lista
IndexOf	Retorna o índice do item e - 1 caso não encontre
LoadFromFile	Carrega texto de um arquivo
SaveToFile	Salva texto para um arquivo

- **TCheckBox**
  Utilizado para obter informações de checagem.

Propriedade	Descrição
AllowGrayed	Determina se o checkbox terá três possibilidades de estado
Checked	Determina se o checkbox está marcado
State	Estado atual do checkbox

- **TRadioButton**
  Usado em grupo, pode ser utilizado para obter informações lógicas mutuamente exclusivas, mas é recomendado usar o RadioGroup em vez de RadioButtons.

- **TRadioGroup**
  Componente que agrupa e controla RadioButtons automaticamente.

Propriedade	Descrição
Columns	Número de colunas de RadioButtons
Items	Lista de strings com os itens do RadioGroup, cada item da lista representa um RadioButton
ItemIndex	Item selecionado, iniciando em 0

- **TPanel**
  Componente Container utilizado para agrupar componentes em um painel.

Propriedade	Descrição
BevelInner	Estilo da moldura interna do painel
BevelOuter	Estilo da moldura externa do painel
BevelWidth	Largura das molduras
BorderStyle	Estilo da Borda
BorderWidth	Largura da borda, distância entre as molduras interna e externa

- **TScrollBox**
  Container com barras de rolagem automáticas.

- **TGroupBox**
  Componente container com um título e borda 3D.

- **TBevel**
  Moldura ou linha com aparência 3D.

Propriedade	Descrição
Shape	Tipo de moldura a ser desenhada
Style	Define alto ou baixo relevo para a linha

- **TListBox**
  Utilizado para exibir opções em uma lista.

Propriedade	Descrição
Columns	Número de colunas de texto da lista
MultiSelect	Define se será permitida a seleção de múltiplos itens
ExtendedSelect	Define se a seleção poderá ser estendida pelo uso das teclas Shift e Ctrl
IntegralHeight	Define se os itens poderão aparecer parcialmente ou somente por completo
Items	Lista de strings com os itens da lista
ItemIndex	Índice do item selecionado, começando em 0
Selected	De acordo com o índice indica se um item em particular esta selecionado
SelCount	Indica quantos itens estão selecionado
Sorted	Define se os itens aparecerão ordenados

- **TComboBox**
  Caixa combinada com lista suspensa.

Propriedade	Descrição
Items	Lista de strings com os itens da lista
DropDownCount	Número de itens visíveis da lista suspensa
Style	Estilo do ComboBox, os principais estilos são csDropDown, csDropDownList, csSimple

- **TImage**
  Componente usado para exibir figuras.

Propriedade	Descrição
Center	Determina de a figura será centralizada no componente
Picture	Figura a exibida, pode ser BMP, ICO, WMF ou EMF
Stretch	Define se o tamanho da figura deve ser ajustada ao do componente

- **TPicture**
  Classe usada para guardar ícones, bitmaps, fotos e figuras definidas pelo usuário.

Método	Descrição
LoadFromFile	Carrega figura de um arquivo
SaveToFile	Salva figura para um arquivo

- **TPageControl**
  Usado para criar controles com múltiplas páginas, que podem ser manipuladas, em tempo de projeto, através do menu de contexto. Cada página criada é um objeto do tipo TTabSheet.

Propriedade	Descrição
ActivePage	Página ativa
Evento	**Descrição**
OnChange	Após uma mudança de página
OnChanging	Permite a validação de uma mudança de página
Método	**Descrição**
FindNextPage	Retorna a próxima página
SelectNextPage	Seleciona a próxima página

- **TTabSheet**
  Página de um PageControl.

Propriedade	Descrição
PageIndex	Ordem da página
TabVisible	Define se a aba da página é visível

- **TShape**
  Gráfico de uma Forma geométrica.

Propriedade	Descrição
Brush	Preenchimento da figura, objeto do tipo Tbrush
Pen	Tipo da linha, objeto do tipo Tpen
Shape	Forma geométrica

- **TTimer**
  Permite a execução de um evento a cada intervalo de tempo.

Propriedade	**Descrição**
Interval	Tempo em milissegundos quando o componente irá disparar o evento OnTimer
Evento	**Descrição**
OnTimer	Chamado a cada ciclo de tempo determinado em Interval

- **TStatusBar**
  Utilizado para criar barras de status para exibir informações.

Propriedade	**Descrição**
SimplePanel	Indica se haverá apenas um panel
SimpleText	Texto exibido caso SimplePanel seja True
SizeGrip	Define se a alça de redimensionamento padrão deve ser mostrada
Panels	Propriedade do tipo TStatusPanels, com os painéis do StatusBar

- **TStatusPanels**
  Lista de panels de um StatusBar.

Propriedade	**Descrição**
Count	Número de panels
Items	Lista de panels, cada panel é um objeto do tipo TStatusPanel
Método	**Descrição**
Add	Adiciona um novo panel à lista

- **TStatusPanel**
  Panel de um StatusBar.

Propriedade	**Descrição**
Text	Texto do panel
Width	Largura em pixels
Bevel	Moldura do panel
Alignment	Alinhamento do texto de um panel

- **TProgressBar**
  Barra de progresso.

Propriedade	**Descrição**
Max	Valor máximo do progresso (ponto de chegada)
Min	Valor mínimo (ponto de partida)
Position	Posição atual (progresso atual)
Step	Avanço padrão

Caixas de Diálogo

Grupo de caixas de diálogo comuns da maioria dos aplicativos.

Método	Descrição
Execute	Mostra a caixa de diálogo e retorna True caso o usuário clique em Ok

- **TOpenDialog / TSaveDialog**
  Caixas de diálogo para abrir e salvar arquivos.

Propriedade	Descrição
FileName	Nome do arquivo
DefaultExt	Extensão padrão para os arquivos
Filter	Filtro, com os tipos de arquivos que serão abertos ou salvos
FilterIndex	Índice do filtro default
InitialDir	Pasta inicial
Title	Título da janela
Options	Define características gerais do diálogo

- **TFontDialog**
  Caixa de diálogo de escolha de fonte.

Propriedade	Descrição
MinFontSize	Tamanho mínimo da fonte
MaxFontSize	Tamanho máximo da fonte
Options	Define características das fontes
Evento	**Descrição**
OnApplyClicked	Ocorre após o usuário pressionar o botão Aplicar, antes de a janela fechar

Menus

No Lazarus os menus serão desenhados no Menu Editor, que pode ser acessado com um duplo clique sobre o componente de menu.

- **TMainMenu**
  Menu principal de um Form.

Propriedade	Descrição
Items	Itens de menu, essa propriedade guarda todas as alterações feitas no Menu Designer

- **TPopUpMenu**
  Menu de contexto de um componente. Cada componente tem uma propriedade PopUpMenu, que indica seu menu de contexto.

- **TMenuItem**
  Item de menu.

Propriedade	Descrição
Checked	Indica se o item está marcado ou não
GroupIndex	Índice do grupo do item, semelhante ao SpeedButton
RadioGroup	Indica se o item pode ser mutuamente exclusivo com outros itens do mesmo grupo
ShortCut	Tecla de atalho do item

Classes Não Visuais

- **TApplication**

Todo programa tem um objeto global nomeado Application, do tipo TApplication, esse objeto representa a aplicação para o sistema operacional.

Propriedade	**Descrição**
ExeName	Caminho e nome do arquivo executável
MainForm	Form principal da aplicação
Hint	Hint recebido pela aplicação
Title	Título da aplicação
HelpFile	Caminho e nome do arquivo help
Evento	**Descrição**
OnHint	Quando um hint é recebido pela aplicação
OnException	Quando ocorre uma exceção
OnHelp	Quando acontece uma solicitação de help
Método	**Descrição**
MessageBox	Apresenta um quadro de mensagem
Run	Executa a aplicação
Terminate	Finaliza a aplicação normalmente

Quadros de Mensagem

O método **Application.MessageBox** mostra quadros de mensagem com chamadas a funções da API do sistema operacional. Os flags de mensagem mais usados e os valores de retorno desse método são mostrados abaixo (**deve-se utilizar na cláusula USES a biblioteca LCLType**).

```
Application.MessageBox('Texto','Título',Flags):integer;
```

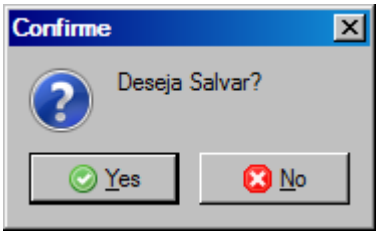
Flag	**Valor**	**Item Mostrado**
MB_OK	0	Botão de Ok
MB_OKCANCEL	1	Botões de Ok e Cancelar
MB_ABORTRETRYIGNORE	2	Botões de Anular, Repetir e Ignorar
MB_YESNOCANCEL	3	Botões de Sim, Não e Cancelar
MB_YESNO	4	Botões de Sim e Não
MB_RETRYCANCEL	5	Botões de Repetir e Cancelar
MB_ICONERROR	16	Ícone de erro
MB_ICONQUESTION	32	Ícone de pergunta
MB_ICONEXCLAMATION	48	Ícone com ponto de exclamação
MB_ICONINFORMATION	64	Ícone com letra i, usada para mostrar informações
Valor de Retorno		**Botão Escolhido**
IDOK	1	Ok
IDCANCEL	2	Cancelar
IDABORT	3	Anular
IDRETRY	4	Repetir
IDYES	6	Sim
IDIGNORE	5	Ignorar
IDNO	7	Não

Esses quadros são usados quando se deseja uma resposta simples do usuário, principalmente numa confirmação ou pergunta para o usuário, observe o código abaixo:

```
    if Application.MessageBox('Deseja Salvar?', 'Confirme', MB_ICONQUESTION +
MB_YESNO) = IDNO then Close;
```

Observe que este outro código também teria o mesmo resultado:

```
if Application.MessageBox('Deseja Salvar?', 'Confirme', 36) = 7 then Close
```



Além do método MessageBox da classe Application, existem ainda outras funções utilitárias para construção rápida de janelas de aviso. Abaixo estão descritos alguns e suas características:

- **MessageDlg**

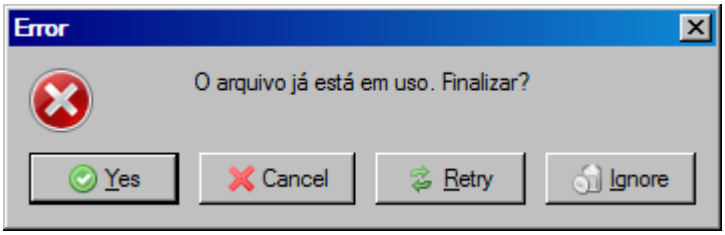
A função MessageDlg implementa basicamente a mesma funcionalidade do Método MessageBox, porém permite que sejam colocados mais que três botões. Compõe-se de 4 parâmetros: a mensagem que será mostrada, o tipo de bitmap usado para ilustrar o diálogo, os botões que serão utilizados e o índice de help que a janela deverá chamar caso seja solicitada, em caso de não uso, deve-se colocar zero.

```
MessageDlg(Mensagem,Tipo,Botoes,Indice_Help):integer;
```

Tipo	Descrição
mtWarning	Ponto de exclamação amarelo
mtError	Um sinal de ‘pare’ vermelho
mtInformation	Um ‘i’ azul
mtConfirmation	Um ponto de interrogação azul
MtCustom	Sem bitmap
Botões	**Descrição**
mbYes	Sim
mbNo	Não
mbCancel	Cancelar
mbHelp	Ajuda
mbAbort	Anular
mbRetry	Repetir
mbIgnore	Ignorar
mbAll	Todos
Retorno	**Descrição**
mrNone	Nenhum
mrYes	Sim
mrNo	Não
mrCancel	Cancelar
mrHelp	Ajuda
mrAbort	Anular
mrRetry	Repetir
mrIgnore	Ignorar
mrAll	Todos

Observe o código abaixo e o resultado visual:

```
if MessageDlg('O arquivo já está em uso. Finalizar?', mtError,
       [mbYes,mbCancel,mbRetry,mbIgnore],0) = mrYes then Table1.Close;
```



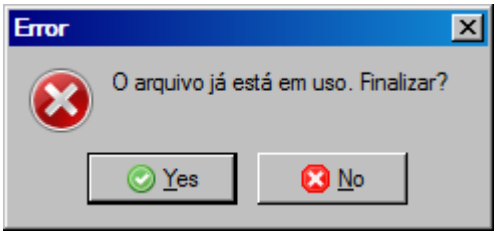
- **MessageDlgPos**

É semelhante à função MessageDlg, porém permite escolher a posição onde será mostrado o quadro de mensagem

```
MessageDlg(Mensagem,Tipo,Botoes,Indice_Help,X,Y):integer;
```

A diferença é que ela implementa dois parâmetros: X e Y que representam as coordenadas do canto superior esquerdo do quadro da mensagem (em pixels, referentes à tela).

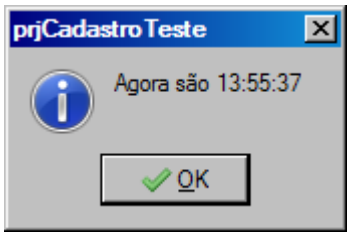
```
if MessageDlgPos('O arquivo já está em uso. Finalizar?', mtError,
       [mbYes,mbNo],0,120,150) = mrYes then Table1.Close;
```



- **ShowMessage**

A função ShowMessage simplesmente mostra um quadro de mensagem contendo apenas a mensagem passada como parâmetro e o botão de Ok.

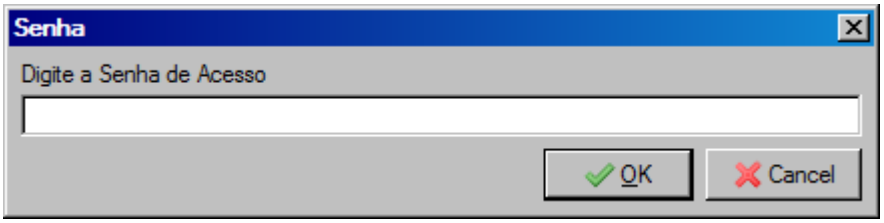
```
ShowMessage(Mensagem);

ShowMessage('Agora são ' + TimeToStr(time));
```



- **InputBox**

A função InputBox permite ao usuário informar valores, é uma função de entrada de dados. Compõe-se de três parâmetros: título, pergunta e valor default. Observe:

```
InputBox(Título,Pergunta,ValorPadrão):string

InputBox('Senha','Digite a Senha de Acesso','');
```



- **TScreen**

O Lazarus automaticamente cria a variável Screen do tipo TScreen, essa variável guarda características do vídeo, como mostrado abaixo.

Propriedade	**Descrição**
ActiveForm	Form com o foco
FormCount	Número de Forms no vídeo
Cursor	Cursor do mouse
Forms	Lista dos Forms disponíveis
Fonts	Lista de Fontes de tela disponíveis
PixelsPerInch	Número de pixels por polegada da Fonte usada pelo sistema
Height	Altura da tela em pixels
Width	Largura da tela em pixels

- **TList**

Estrutura de dados polimórfica que pode gerenciar uma lista de objetos de qualquer classe e possui métodos semelhantes aos de TStrings.

- **TStringList**

Lista de strings descendente de TStrings usada para manter listas de strings independentes de qualquer componente.

- **TCanvas**

Um objeto da classe TCanvas é uma superfície de desenho, onde podem ser usados vários métodos de plotagem gráfica. Todos os controles visíveis possuem uma propriedade Canvas, do tipo TCanvas, que geralmente é usada nos Forms e no objeto Printer.

Propriedade	**Descrição**
Brush	Padrão de preenchimento, propriedade do tipo Tbrush
Pen	Estilo de linha, propriedade do tipo Tpen
Font	Fonte usada nas plotagens de texto
Método	**Descrição**
TextOut	Desenha texto na superfície
Ellipse	Desenha uma elipse
Polygon	Desenha um polígono
Rectangle	Desenha um retângulo

Exceções

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

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

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

Blocos Protegidos

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

```
try

  A := StrToFloat(EdtA.Text);
  B := StrToFloat(EdtB.Text);
  ShowMessage(Format('%f / %f = %f', [A, B, A + B]));

except

  ShowMessage('Números inválidos.');

end;
```

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

```
try

  Soma := StrToFloat(EdtSoma.Text);
  NumAlunos := StrToInt(EdtNum.Text);
  ShowMessage(Format('Média igual a %f.', [Soma / NumAlunos]));

except

  on EConvertError do
    ShowMessage('Valor inválido para soma ou número de alunos.');

  on EZeroDivide do
    ShowMessage('O número de alunos tem que ser maior que zero.');

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

end;
```

Principais Exceções

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

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

Blocos de Finalização

Blocos de finalização são executados sempre, haja ou não uma exceção. Geralmente os blocos de finalização são usados para liberar recursos.

```
FrmSobre := TFrmSobre.Create(Application);
try
  FrmSobre.Img.LoadFromFile('Lazarus.png');
  FrmSobre.ShowModal;
finally
  FrmSobre.Release;
end;
```

Pode-se usar blocos de proteção e finalização aninhados

```
FrmOptions := TFrmOptions.Create(Application);
try

  FrmOptions.ShowModal;

  try
         Soma := StrToFloat(EdtSoma.Text);
         NumAlunos := StrToInt(EdtNum.Text);
         ShowMessage(Format('Média igual a %f.', [Soma / NumAlunos]));
  except
         on EConvertError do
           ShowMessage('Valor inválido para soma ou número de alunos.');
         on EZeroDivide do
           ShowMessage('O número de alunos tem que ser maior que zero.');
         else
           ShowMessage('Erro na operação, verifique os valores digitados.');
  end;

finally

  FrmOptions.Release;

end;
```

Geração de Exceções

Uma exceção pode ser provocada usando a cláusula raise.

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

Também é possível criar tipos próprios de exceções.

```
type
  EInvalidUser = class (Exception);

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

Para que uma exceção continue ativa, mesmo depois de tratada, deve-se utilizar a cláusula raise dentro do bloco de tratamento da exceção. Geralmente isso é feito com exceções aninhadas.

```
try
  SQLQuery1.Edit;
  SQLQuery1.FieldByName('contador').asInteger := StrToInt(Edit1.Text);
  SQLQuery1.Post;
except
  ShowMessage('Erro ao alterar contador.');
  raise;
end;
```

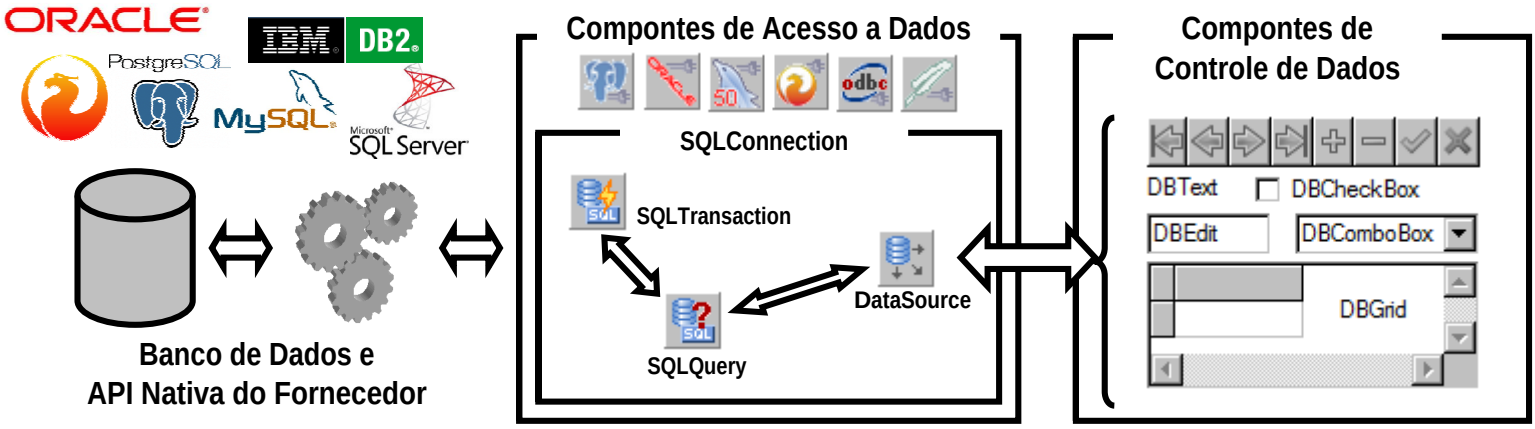
CAPÍTULO V – Banco de Dados

Considerações Iniciais

Uma característica importante do Lazarus é o seu suporte prático a uma diversidade de bancos de dados. O ambiente define um componente com comportamento padrão que é especializado por cada fornecedor de banco de dados para oferecer conexão e acesso nativo aos dados (sem a necessidade de softwares intermediários). Além do suporte que é oferecido com a instalação padrão do Lazarus, existem ainda alguns pacotes de componentes para acesso a banco de dados, ZeosLib é um exemplo.

O Lazarus permite que os dados sejam acessados por código ou através de componentes visuais, os chamados componentes data-aware. Os componentes data-aware estão ligados a fontes de dados por meio de propriedades e representam os campos de tabelas ou consultas.

Para acessar uma base de dados através do Lazarus é necessário antes estabelecer alguns conceitos importantes. A figura abaixo demonstra a infra-estrutura de acesso aos dados no Lazarus.



Configurando a conexão e o acesso aos dados

Para desenvolver uma aplicação em Lazarus conectada a um banco de dados é necessário no mínimo a utilização de três tipos de componentes:

- Componentes de **Acesso aos Dados** (conexão, transação, consulta), localizados na aba **SQLdb**;
- Componentes de **Controle de Dados** (caixas de texto, barras de navegação, grades), localizados na aba **Data Controls**;
- Componentes que fazem a ligação **entre aqueles que acessam e os que controlam**, onde o exemplo mais prático é o **DataSource** (pertencente a aba **Data Access**);

SQLdb – Componentes de Acesso aos Dados

- **TSQLConnection (PQConnection, OracleConnection, MySQL50Connection, IBConnection, ODBCConnection, SQLite3Connection, etc.)**

Componentes derivados da classe TSQLConnection representam a conexão com o banco de dados. O componente SQLConnection não existe como um componente que pode ser utilizados, deve-se portanto utilizar um componente que ofereça as capacidades específicas de conexão com o banco de

dados que se está utilizando. Em uma aplicação comercial, a eventual troca do fornecedor de banco de dados pode ser facilmente atendida através da alteração deste componente. Alguns exemplos de componentes que representam a conexão com um banco de dados são: PQConnection, OracleConnection, MySQL50Connection, IBConnection, ODBCConnection, SQLite3Connection, etc.

Propriedades	**Descrição**
Connected	Define se a conexão com o banco de dados está ativa
DatabaseName	Nome do banco de dados (nome que identifica o banco de dados que se deseja conectar)
KeepConnection	Define se a conexão com o banco de dados será mantida, mesmo sem DataSets abertos
LoginPrompt	Define se será mostrado o quadro de login solicitando a identificação do usuário (login e senha)
UserName	Identificação do usuário para servidor de banco de dados
Password	Senha do usuário
HostName	Nome ou endereço IP do servidor de banco de dados
Port	Número da porta de conexão com o servidor de banco de dados
CharSet	Codificação de caracteres utilizada pelo banco de dados
Transaction	Componente do tipo SQLTransaction que identifica uma transação no banco de dados
Params	Parâmetros adicionais que determinados bancos de dados podem exigir
Métodos	**Descrição**
Close	Encerra a conexão com o banco de dados, todos os DataSets serão fechados
CloseDataSets	Fecha todos os DataSets abertos, mas a conexão não é encerrada
Open	Abre a conexão com o banco de dados
Eventos	**Descrição**
OnLogin	Evento disparado durante o processo de login do banco de dados.

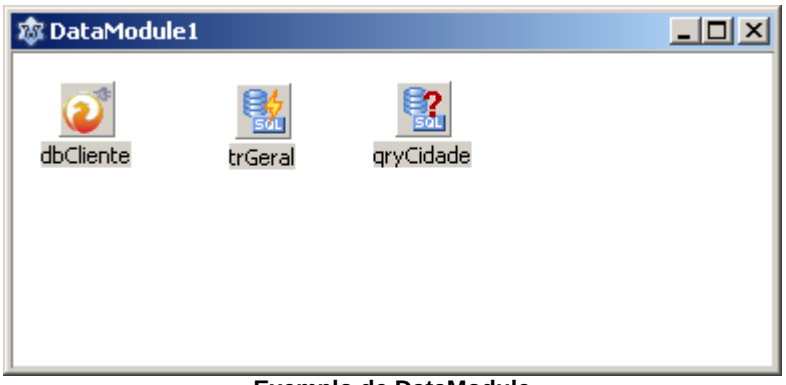
- **TSQLTransaction**

SQLTransaction é o componente que controla as transações no banco de dados. Em um banco de dados as operações ocorrem no contexto de uma transação. Uma transação é finalizada com sucesso através de um COMMIT. Um comando de ROLLBACK desfaz as operações realizadas anteriormente.

Propriedades	**Descrição**
Active	Indica o estado da transação (ativa ou inativa)
Action	Define o tipo de transação padrão
DataBase	Indica qual conexão essa transação está vinculada
Métodos	**Descrição**
StartTransaction	Inicia a transação (os comandos somente serão persistidos após um *commit* ou então todos serão cancelados através de um *rollback*)
Commit	Encerra a transação persistindo todas as alterações
CommitRetaining	Persiste todas as alterações sem encerrar a transação
Rollback	Desfaz as alterações e encerra a transação
RollbackRetaining	Desfaz as alterações mas não encerra a transação

- **TDataModule**

Um DataModule é como se fosse um Form invisível, onde serão inseridos os componentes de acesso a dados, como o SQLConnection, SQLTransaction, SQLQuery e DataSource. Uma vez que os DataModules são classes eles permitem a fácil implementação de modelos de objetos, permitindo herança, criação de métodos, dentre outros aspectos. Para inserir um DataModule em um projeto, escolha New... no menu File e escolha a opção Data Module (dentro da opção Module). Os Data Modules não gastam recursos do sistema, servem apenas para agrupar os componentes de acesso a dados e potencializar a organização do programador.



Exemplo de DataModule

- **TSQLQuery**

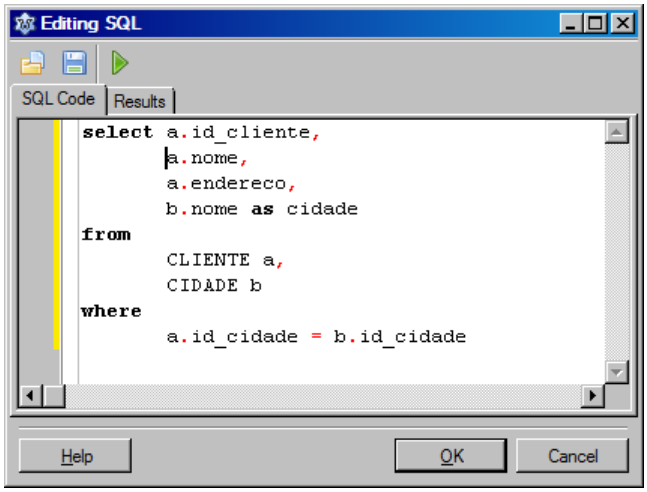
O componente SQLQuery é a principal via de acesso para interação com o banco de dados. É através dele que utilizamos instruções em linguagem SQL para representar acessos a tabelas e/ou consultas personalizadas no banco de dados. É um componente ancestral de uma classe denominada de TDataSet, que contém um conjunto de propriedades e métodos para representar o estado e as operações sobre um conjunto de dados.

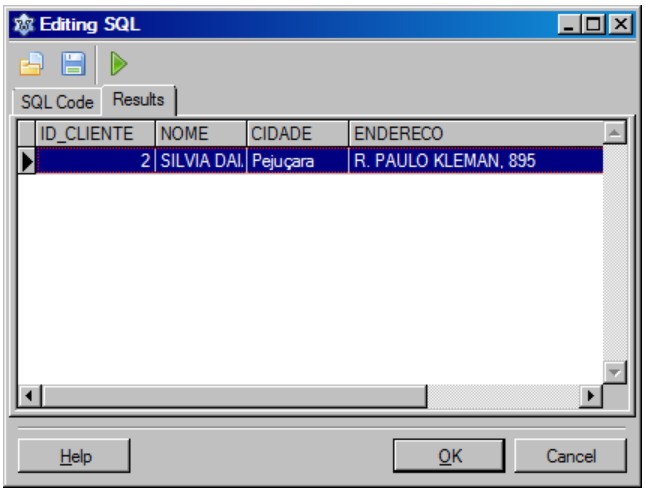
Propriedades	**Descrição**
Active	Define se a consulta está aberta ou fechada
AutoCalcFields	Indica se os campos calculados devem sofrer atualização automática
BOF	Informa se o cursor está no início do conjunto de dados
Database	Componente TSQLConnection que fornece a conexão para a consulta
DataSource	Define o DataSource para se obter os valores das variáveis na consulta.
DeleteSQL	Instrução SQL alternativa para execução de exclusões (delete). Exemplo de utilização: delete from clientes where codigo = :old_codigo and nome is not null; O prefixo ":old_" é utilizado para referenciar o valor que está sendo excluído.
EOF	Informa se o cursor está no final do conjunto de dados
FieldCount	Número de campos da consulta
FieldDefs	Lista com a definição dos campos da consulta
Fields	Lista de objetos do tipo TField, que representam os campos da consulta
Filter	String com uma condição de filtragem
Filtered	Define se a consulta está filtrada
IndexDefs	Lista com a definição dos índices
IndexFieldNames	Nome dos campos de índice, usados para ordenar os registros da consulta
IndexName	Nome do índice atual, vazia quando o índice for a chave primária
InsertSQL	Instrução SQL alternativa para execução de inclusões (insert). Exemplo de utilização: `insert into clientes (codigo, nome) values (:codigo, :nome);` Os nomes dos campos na cláusula values devem ser exatamente iguais aos nomes dos campos na tabela destino.
MasterFields	Campos usados no relacionamento com a tabela mestre
Modified	Define se o registro atual foi modificado
ReadOnly	Define se a consulta é somente leitura
RecNo	Número do registro atual
RecordCount	Número de registros
SQL	Instrução SQL normalmente utilizada para acessar o conteúdo de uma tabela (select * from tabela) ou para fazer consultas personalizadas.
State	Estado do conjunto de dados (edição, inserção, inativo, etc).
Transaction	Componente que controla as transações realizadas por essa consulta
UpdateSQL	Instrução SQL alternativa para execução de atualizações (update). Exemplo de utilização: update clientes set nome = :nome where codigo = :old_codigo and nome is not null; O prefixo ":old_" é utilizado para referenciar o valor anterior (antes da atualização).

Método	**Descrição**
Append	Entra em modo de inserção e, ao gravar, o registro será colocado no fim do conjunto
AppendRecord	Insere um registro no final do arquivo através de código
ApplyUpdates	O método ApplyUpdates faz com que as alterações feitas na cache do conjunto de dados (alterações apenas em memória) sejam enviadas para o banco de dados.
Cancel	Cancela as alterações feitas no registro atual
Close	Fecha a tabela
Delete	Exclui o registro corrente
DisableControls	Desabilita a atualização dos controles visuais
Edit	Permite a alteração dos campos do registro atual
EnableControls	Habilita os controles visuais
ExecSQL	Executa uma instrução, exceto SELECT
FieldByName	Acessa um campo, do tipo TField, pelo nome
First	Move para o primeiro registro
Insert	Entra em modo de inserção de um novo registro na posição atual
InsertRecord	Adiciona um novo registro, já com os dados, na posição atual
IsEmpty	Define se a tabela está vazia
Last	Move para o último registro
Locate	Procura um registro, usando ou não índices, de acordo com a disponibilidade
Lookup	Procura um registro e retorna valores dos campos deste
MoveBy	Move um número específico de registros
Next	Move para o próximo registro
Open	Abre a tabela
Post	Grava as alterações no registro atual
Prepare	Prepara uma instrução SQL para ser aberta ou executada
Prior	Move para o primeiro registro
Refresh	Atualiza a tabela com os dados já gravados
Evento	**Descrição**
AfterCancel	Após do método Cancel
AfterClose	Após o fechamento da consulta
AfterDelete	Após do método Delete
AfterEdit	Após do método Edit
AfterInsert	Após do método Insert
AfterOpen	Após do método Open
AfterPost	Após do método Post
AfterScroll	Após mudar de registro
BeforeCancel	Antes do método Cancel
BeforeClose	Antes do fechamento da consulta
BeforeDelete	Antes do método Delete
BeforeEdit	Antes do método Edit
BeforeInsert	Antes do método Insert
BeforeOpen	Antes do método Open
BeforePost	Antes do método Post
BeforeScroll	Antes de mudar o registro
OnCalcFields	Evento usado para calcular os valores dos campos calculados
OnDeleteError	Quando ocorre um erro ao chamar o método Delete
OnEditError	Quando ocorre um erro ao chamar o método Edit
OnFilterRecord	Evento usado com filtragem variável
OnNewRecord	Quando a consulta entra em modo de inserção (obs.: não deixa Modified igual a True)
OnPostError	Quando ocorre um erro ao chamar o método Post

Construindo uma consulta em tempo de projeto

Para construir uma consulta SQL em tempo de projeto, utiliza-se a propriedade SQL do componente TSQLQuery. Ao clicar no botão de reticências, o Lazarus abre a janela do Editing SQL, e nele a consulta é digitada normalmente, utilizando os comandos SQL padrão do banco de dados a ser utilizado:

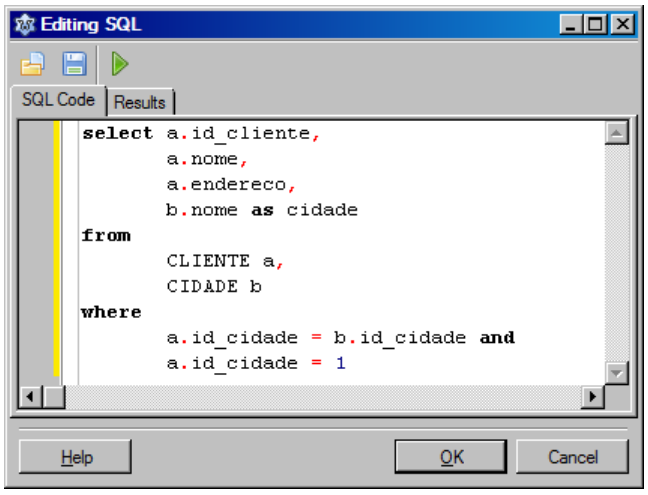


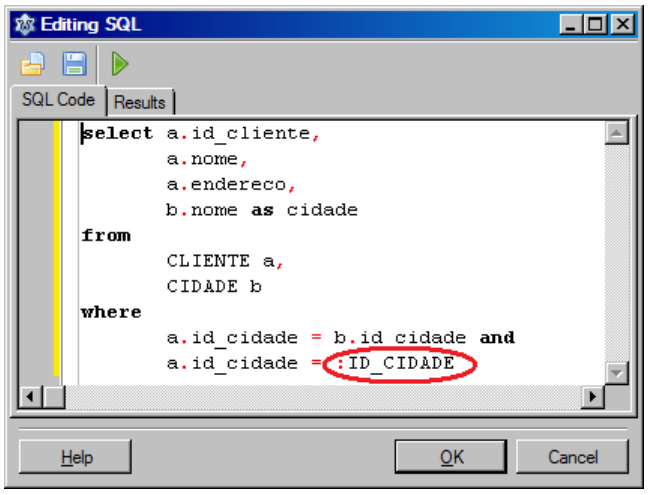


Caso a conexão (SQLConnection) ao qual a SQLQuery está conectada estiver ativa é possível testar a consulta e exibir o resultado na aba “Results”

Passagem de Parâmetros

Geralmente em uma instrução SQL genérica é necessário especificar alguns parâmetros, por exemplo: selecionar todos os clientes de uma determinada cidade. Para isto, o nome do parâmetro deve ser incluído na expressão, precedido pelo símbolo de “:” (dois pontos).





No primeiro caso o SQL é escrito em tempo de execução passando o valor 1 para o critério “`a.id_cidade = `” como condição filtrante da consulta. Já no segundo caso, a maneira como o parâmetro **:ID_CIDADE** foi escrito, permite alterá-lo em tempo de execução:

```
SQLQuery1.Close;
SQLQuery1.Params[0].AsInteger := StrToInt(Edit1.Text);
// ou Params.ParamByName('ID_CIDADE').asInteger := StrToInt(Edit1.Text);
SQLQuery1.Open;
```

Construindo uma consulta em tempo de execução

Para construir uma consulta SQL em tempo de execução a propriedade SQL do componente SQLQuery deve ser manipulado. O métod ADD parmite adicionar os comandos, um a um:

```
SQLQuery1.Close;
SQLQuery1.SQL.Clear;
SQLQuery1.SQL.ADD('select a.id_cliente, a.nome, a.endereco, b.nome as cidade');
SQLQuery1.SQL.ADD('from   CLIENTE a, CIDADE b');
SQLQuery1.SQL.ADD('where  a.id_cidade = b.id_cidade and a.id_cidade = 1');
SQLQuery1.Open;
```

Esta consulta faz a mesma coisa que aquela criada em tempo de projeto, porém permite alterar ou criar novas consultas em tempo de execução, via código. Da mesma forma, para alterar valores de parâmetros em tempo de execução utiliza-se a propriedade Params do componente SQLQuery.

```
SQLQuery1.Close;
SQLQuery1.SQL.Clear;
SQLQuery1.SQL.ADD('select * from cidade where id_cidade = :ID_CIDADE');
SQLQuery1. Params.ParamByName('ID_CIDADE').asInteger := StrToInt(Edit1.Text);
SQLQuery1.Open;
```

Executando um Comando SQL

Para **executar** um comando SQL (**insert**, **update**, **create table** ...), é necessário utilizar o método **execSQL**, diferente do comando **select**, onde apenas utiliza-se o método **open**, do componente SQLQuery.

```
SQLQuery1.SQL.Clear;
SQLQuery1.SQL.Add(‘create table senhas_telefone (sn_rgfunc char(6) not null,’);
SQLQuery1.SQL.Add(‘sn_nome varchar(40) not null, sn_ramal char(3),’);
SQLQuery1.SQL.Add(‘sn_senha char(5) not null);’);
SQLQuery1.execSQL;
```

Neste exemplo uma tabela é criada pelo comando CREATE TABLE e no exemplo seguinte são inseridos alguns registros na tabela recém criada, utilizando-se de dados de campos de formulário (Edits)

```
SQLQuery1.Close;
SQLQuery1.SQL.Clear;
SQLQuery1.SQL.Add('insert into senhas_telefone values (
                  ''' + Edit1.Text + ''',
                  ''' + Edit2.Text + ''',
                  ''' + Edit3.Text + ''',
                  ''' + Edit4.Text + ''')' );
SQLQuery1.execSQL;
```

Filtros

A propriedade Filter, permite filtrar os registros em cache de uma consulta usando uma expressão lógica (abaixo alguns exemplos). Para tornar um filtro ativo, a propriedade Filtered deve ser igual a True.

```
Data = '20/04/1998'
(Data = '20/04/1998') AND (Vendedor = 'Guilherme Augusto da Fonseca')
(Nome > 'A') AND (Nome < 'B')
```

Ao filtrar uma consulta, a propriedade RecordCount do componente SQLQuery mostra o número total de registros da consulta, ignorando se a consulta está com o filtro ativado ou não.

Alterando Registros

Para alterar registros através de código, a consulta deve ser colocada em modo de edição e só então o valor dos campos desejados devem ser alterados e posteriormente gravados (lembrando que as alterações são feitas nos dados em cache e precisam ser efetivadas no banco de dados através do método ApplyUpdates e do encerramento da transação).

```
SQLQuery1.Edit;
SQLQuery2.FieldByName('ID').asInteger := 1;
SQLQuery2.FieldByName('DESCRICAO').asString := 'Teste';
SQLQuery2.Post;
```

Inserindo Registros

Para inserir registros em código utiliza-se o método Insert e posteriormente a atribuição de valores aos campos utilizando o método FieldByName. Da mesma forma que a alteração a operação de inserção apenas coloca o DataSet da consulta em modo de inclusão e realiza alterações na cache do SQLQuery. Para efetivar as inclusões na base de dados o método ApplyUpdates precisa ser executado assim como a transação precisa ser finalizada.

```
SQLQuery1.Insert;
SQLQuery2.FieldByName('ID').asInteger := 10;
SQLQuery2.Post;
```

Localizando Registros

O método Locate é o mais indicado para fazer pesquisas na cache de dados da consulta, no exemplo abaixo é feita uma pesquisa exata.

```
if not SQLQuery3.Locate('ID_CIDADE', Edit1.Text, []) then
    ShowMessage('Cidade não encontrada.');
```

Também podem ser feitas pesquisas parciais e/ou sem sensitividade de caso usando o terceiro parâmetro, que é um conjunto de opções.

```
SQLQuery3.Locate('NOME', Edit1.Text, [loPartialKey, loCaseInsensitive]);
```

As pesquisas em mais de um campo também são permitidas, neste caso, os nomes dos campos devem ser separados por ponto e vírgula e a função VarArrayOf (unit variants) deve ser usada para criar um array com os valores que se deseja procurar.

```
if not SQLQuery3.Locate('Vendedor;Data', VarArrayOf([EdtVendedor.Text, EdtData.Text]),
  [loCaseInsensitive]) then
  ShowMessage('O vendedor não realizou nenhuma venda nessa data');
```

Indexação

A indexação é usada para ordenar os registros da tabela, para isso deve-se escolher os campos pelos quais se deseja ordenar na propriedade IndexFieldNames, inclusive em código, como mostrado abaixo, todos os campos devem ser indexados e separados por ponto e vírgula.

```
  SQLQuery3.IndexFieldNames := 'ID_CIDADE';
  SQLQuery3.IndexFieldNames := 'ID_CIDADE;NOME';
```

Verificando Alterações

Existem situações onde é necessário verificar se houveram alterações na cache de uma consulta, por exemplo, no evento OnClose de um Form de manutenção, pode-se usar a propriedade Modified, como mostrado no exemplo abaixo.

```
if SQLQuery1.Modified then

  if Application.MessageBox('Gravar alterações?', 'Dados Alterados', MB_ICONQUESTION
    + MB_YESNO) = IDYES then
  begin
    SQLQuery1.ApplyUpdates;
    SQLTransaction1.CommitRetaining;
  end
  else
    SQLQuery1.Cancel;
```

Percorrendo uma Consulta

O código semelhante a este mostrada abaixo pode ser utilizado para percorrer uma consulta do início ao fim.

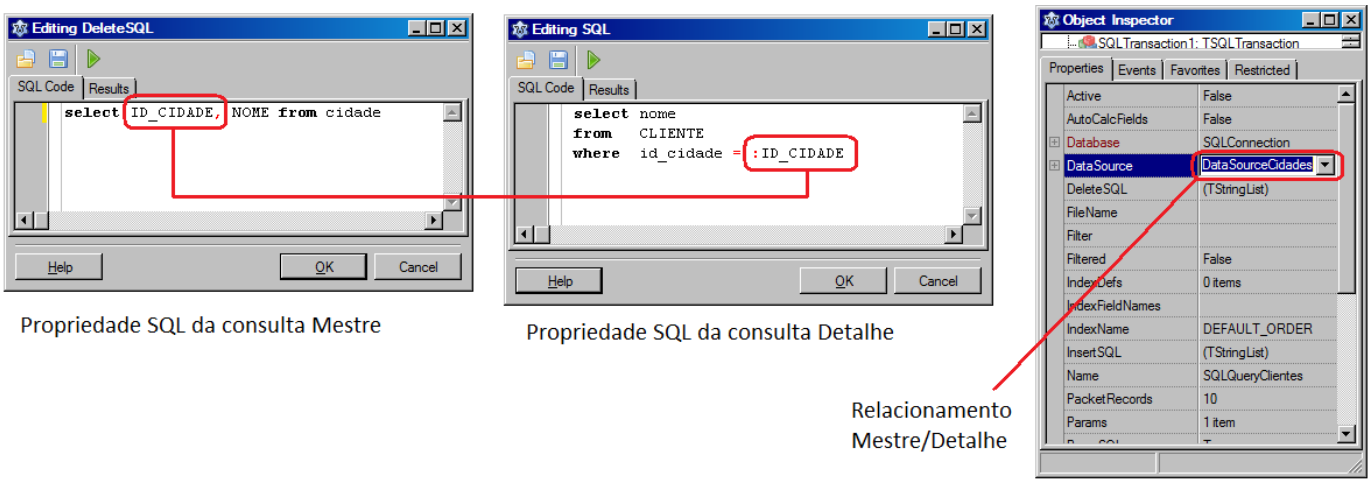
```
SQLQuery1.DisableControls;
Total := 0;
SQLQuery1.First;

while not SQLQuery1.EOF do
begin
  Total := Total + SQLQuery1.FieldByName('IGC_CONTINUO').asFloat;
  SQLQuery1.Next;
end;
SQLQuery1.EnableControls;
```

Mestre/Detalhe

Nos relacionamentos de 1 para N, uma tabela pode estar ligada a outra em uma relação Mestre/Detalhe, nesse tipo de relação os registros da tabela de ordem N são filtrados pelo campo de relacionamento com a tabela de ordem 1. Por exemplo, se o relacionamento de Clientes com Cidades for mestre/detalhe, só serão acessados em Clientes, os registros cujo campo Id_Cidade seja igual ao Id_Cidade da tabela de Cidades.

Para fazer esse tipo de relacionamento utilizamos a propriedade DataSource do componente SQLQuery que representa os dados da tabela detalhe (o valor da propriedade **DataSource** será o componente DataSource que representa a tabela Mestre). Na consulta da tabela detalhe os campos dependentes da tabela mestre devem ser especificados na cláusula where com parâmetros cujos nomes devem ser exatamente os mesmos contidos na cláusula SLQ da SQLQuery da tabela mestre (precedidos do sinal de “:”).

Abaixo uma relação mestre/detalhe que mostra todos os clientes de uma cidade. Na aplicação existem dois componentes SQLQuery, conforme consultas exibidas abaixo:



Marcando Dados

Os métodos para marcar são:

- GetBookmark - Pega a posição do registro e coloca em uma variável do tipo TBookmark;
- GotoBookmark - Leva o apontador para uma variável do tipo TBookmark;
- FreeBookmark - Desaloca uma variável do tipo TBookmark;

```
procedure DoSomething;
var
   Marca: TBookmark;
begin
   Marca  :=  SQLQuery1.GetBookmark; {Marca a posição}
   SQLQuery1.DisableControls; {Desabilita o controle de dados}
   try
       Table1.First;
       while not Table1.EOF do
            begin
               {Faz alguma coisa}
               Table1.Next;
            end;
    finally
       SQLQuery1.GotoBookmark(Marca);
       SQLQuery1.EnableControls;
       SQLQuery1.FreeBookmark(Marca); {Desaloca a variável}
    end;
end;
```

Fields Editor

Para criar objetos para os campos de uma consulta clica-se duas vezes no componente TSQLQuery. Na janela seguinte (Edit Fields) clica-se com o botão direito do mouse e escolhe-se Add Fields, na janela Add Fields, escolhe-se os campos que se deseja utilizar e confirma-se no botão Ok.

No Fields Editor pode-se remover os campos criados, alterar sua ordem de apresentação e usar suas propriedades e eventos no Object Inspector. Para cada campo é criado um objeto de um tipo descendente de TField, como TStringField, TIntegerField, TFloatField. As principais propriedades dos objetos TField estão listadas na tabela abaixo.

Se nenhum objeto TField for criado para uma determinada consulta, todos os campos da consulta estarão disponíveis, mas caso algum seja criado, somente os campos que estiverem criados estarão disponíveis.

- **TField**

A classe TField é usada como ancestral para todos as classes dos campos. Geralmente são utilizados objetos de classes descendentes de TField, mas em todos eles pode-se encontrar os itens mostrados abaixo.

Propriedades	**Descrição**
Alignment	Alinhamento do texto do campo nos controles visuais
AsBoolean	Valor do campo convertido para Boolean
AsCurrency	Valor do campo convertido para Currency
AsDateTime	Valor do campo convertido para DataTime
AsFloat	Valor do campo convertido para Double
AsInteger	Valor do campo convertido para Integer
AsString	Valor do campo convertido para string
AsVariant	Valor do campo convertido para Variant
Calculated	Indica se o campo é calculado em tempo de execução
CanModify	Indica se um campo pode ser modificado
ConstraintErrorMessage	Mensagem de erro se a condição de CustomConstraint não for satisfeita
CustomConstraint	Condição de validação do campo
DataSet	DataSet onde está o campo
DataSize	Tamanho do campo, em Bytes
DataType	Propriedade do tipo TFieldType, que indica o tipo do campo
DefaultExpression	Expressão com valor Default do campo para novos registros
DisplayLabel	Título a ser exibido para o campo
DisplayText	Texto exibido nos controles visuais associados ao campo
DisplayWidth	Número de caracteres que deve ser usado para mostrar o campo no controles visuais
FieldKind	Propriedade do tipo TFieldKind que indica o tipo do campo, como Calculado ou Lookup
FieldName	Nome do campo na tabela
FieldNo	Posição física do campo na tabela
Index	Posição do campo nos controles visuais
IsIndexField	Indica se um campo é válido para ser usado como índice
IsNull	Indica se o campo está vazio
KeyFields	Campo chave da tabela no relacionamento com LookupDataSet, usado em campos Lookup
Lookup	Indica se um campo é Lookup
LookupCache	Define se será usado cache para campos Lookup
LookupDataSet	DataSet onde está definido o valor do campo Lookup
LookupKeyFields	Campo chave do relacionamento em LookupDataSet
LookupResultField	Valor do campo, que será mostrado nos controles visuais
ReadOnly	Define se um campo é somente para leitura
Required	Define se o campo é obrigatório
Size	Tamanho físico do campo
Text	Texto de edição do campo
Value	Acesso direto ao valor do campo
Visible	Define se um campo é visível
Eventos	**Descrição**
OnChange	Chamado quando o valor do campo é mudado
OnSetText	Chamado pelos controles visuais para atribuir o texto digitado pelo usuário ao campo
OnGetText	Chamado para formatar o texto de exibição do campo
OnValidate	Validação do valor atribuído ao campo, caso o valor não seja válido, gere uma exceção
Método	**Descrição**
Assign	Atribui um valor de um campo a outro, inclusive nulo
FocusControl	Seta o foco para o controle visual ligado ao campo nos Forms
Clear	Limpa o conteúdo do campo

Abaixo estão listadas as classes que são manipuladas no tratamento dos campos de uma consulta, são classes descendentes de TField.

TStringField	TBlobField	TTimeField
TSmallintField	TIntegerField	TBytesField
TFloatField	TWordField	TVarBytesField
TCurrencyField	TAutoIncField	TGraphicField
TBooleanField	TBCDField	TMemoField
TDateField	TDateTimeField	

Em algumas dessas classes poderão ser encontradas outras propriedades e métodos que não estão presentes em TField.

Propriedades	**Descrição**
MaxValue	Valor máximo para o campo
MinValue	Valor mínimo para campo
DisplayFormat	Formato de apresentação do campo, como *,0.00" %"* ou *,0.##" Km"*
EditFormat	Formato de edição do campo
Currency	Define se um campo é monetário
DisplayValues	Usado com campos Boolean, define o texto para True e False, como *Sim;Não*
Métodos	**Descrição**
LoadFromFile	Carrega o conteúdo do campo de um arquivo
SaveToFile	Salva o conteúdo do campo para um arquivo

Para acessar os campo de uma tabela, existem várias abordagens, como mostrado abaixo..

- Usando o objeto TField ligado ao campo.

```
sqlQryDescri.Value := sqlQryVend.Value + ' em ' + sqlQryDta.asString;
```

- Através do método FieldByName

```
sqlQry.FieldByName('Descricao').Value :=
         sqlQry.FieldByName('Vendedor').Value + ' em ' +
              sqlQry.FieldByName('Data').AsString;
```

- Usando a lista Fields do TSQLQuery

```
sqlQry.Fields[5].Value := sqlQry.Fields[3].Value + ' em ' +
                                 sqlQry.Fields[4].AsString;
```

Conversão de Tipos

A conversão de tipo de um campo pode ser feita através as propriedades tipo As..., como AsString.

```
SQLQuery1.AsString := EdtData.Text;
SQLQuery1.AsInteger := 10;
SQLQuery1.AsFloat := StrToFloat(EdtValor.Text);
```

Validação

Para validar os valores de um campo, utiliza-se a propriedade CustomConstraint. Para garantir que a quantidade de um item seja maior que zero, por exemplo, utiliza-se em CustomConstraint Quantidade>0, e em CustomConstraint coloca-se a mensagem para o usuário caso a condição seja falsa. Outra forma, mais flexível, é usando o evento OnValidate, com um código como abaixo, onde é gerada uma exceção para cancelar a atribuição do valor ao campo.

```
if CampoQuantidade.Value <= 0 then
  raise Exception.Create('Quantidade deve ser maior que zero.');
```

Formatação Personalizada

Para fazer uma formatação personalizada do campo, pode usar os eventos OnGetText e OnSetText. Por exemplo, se existir um campo “Estado Civil”, e se desejar que quando o valor do campo for C seja mostrado Casado e S: Solteiro, no evento OnGetText utiliza-se um código como o abaixo.

```
if sqlQryEstado.Value = 'C' then
  Text := 'Casado'
else if sqlQryEstado.Value = 'S' then
  Text := 'Solteiro';
```

Como controle visual para o usuário escolher o valor do campo, você poderia usar o DBComboBox, com Solteiro e Casado na propriedade Items, e no evento OnGetText do campo o código mostrado abaixo.

```
if Text = 'Casado' then
  sqlQryEstado.Value := 'C'
else if Text := 'Solteiro' then
  sqlQryEstado.Value = 'S';
```

Campos Calculados

Para criar campos calculados, clica-se com o botão direito do mouse no Fields Editor e escolhe-se New Field, no quadro NewField, digita-se o nome do campo, o nome do objeto será automaticamente informado, o tipo do campo, seu tamanho e escolhe-se Calculated em Field type.

Para colocar um valor nesse campo utiliza-se o evento OnCalcFields do componente TSQLQuery, em nenhuma outra parte os valores desses campos podem ser alterados.

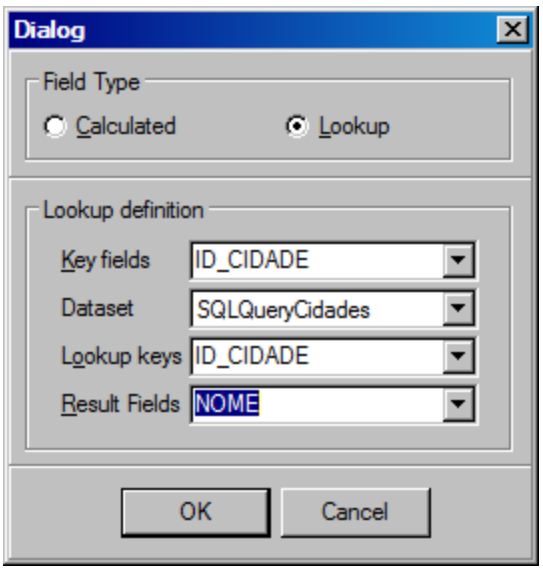
O código do evento OnCalcFields deve ser enxuto, pois este é chamado várias vezes durante a edição de um registro e um procedimento pesado pode comprometer a performance do sistema.

```
procedure TDtmAluno.SQLQueryCalcFields(DataSet: TDataSet);
begin
  if SQLQryFaltas.Value > SQLQryMaxFaltas.Value then
    SQLQrySituacao.Value := 'Evadido'
  else if SQLQryNota.Value >= 7 then
    SQLQrySituacao.Value := 'Aprovado'
  else
    SQLQrySituacao.Value := 'Retido'
end;
```

Campos Lookup

Para criar um relacionamento, às vezes é necessário criar um campo de descrição, por exemplo, na tabela de clientes guardamos o identificador da cidade, mas gostaríamos de exibir o nome da cidade, esses campos são chamados de campos Lookup. Para criar um campo Lookup, siga os passos abaixo, tomando como exemplo o caso da relação entre cidade e cliente.

- Abra o Fields Editor do SQLQuery onde o campo será adicionado, no caso a consulta de Clientes.
- Clique com o direito e escolha New
- No quadro New Field, em Field type, escolha Lookup.
- Em Key Fields escolha o campo da tabela que faz parte do relacionamento (neste caso ID_CIDADE)
- DataSet é a consulta (SQLQuery) onde está a descrição que se deseja buscar (consulta Cidades)
- Em Lookup Keys, escolha o campo de DataSet que faz parte do relacionamento (ID_CIDADE da consulta Cidades)
- Finalmente, escolha em Result Fields o campo de DataSet que vai ser mostrado para o usuário (o nome da cidade)



Essas opções correspondem a algumas propriedades do objeto TField gerado, que podem ser alteradas no Object Inspector, KeyFields, LookupDataSet, LookupKeyFields, LookupDataSet e LookupResultField.

Quando esses campo são exibidos em um DBGrid, por padrão é criado um botão de lookup que mostrará os valores da outra tabela uma lista. Para colocar esses campos em um Form, devemos usar o DBLookupComboBox, apenas com as propriedades padrão, DataSource e DataField.

- **TDataSource**

Componente usado para fazer a ligação entre um DataSet (SQLQuery) e os componentes visuais, é encontrado na guia Data Access.

Propriedade	**Descrição**
AutoEdit	Define se a consulta entrará em modo de edição assim que o usuário digitar novos valores nos controles
DataSet	DataSet (consulta) ao qual o TDataSource faz referência
Evento	**Descrição**
OnDataChange	Ocorre quando o DataSet é alterado, ao mudar de registro ou mudar os valores dos campos
OnStateChange	Ocorre quando o estado do DataSet é alterado
OnUpdateData	Ocorre antes de uma atualização

Obtendo o Estado de uma Consulta

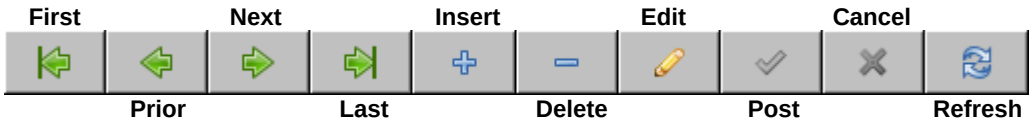
Manipulando o evento StateChange da DataSource podemos obter o estado de uma consulta em qualquer momento, atualizando-o quando houver alterações.

```
procedure TForm1.DataSource1.StateChange(Sender:TObject);
var S:String;
begin
   case SQLQuery1.State of
     dsInactive: S := 'Inactive'; //o data set está fechado
     dsBrowse: S := 'Browse';  //estado de espera (visualização/navegação)
     dsEdit: S := 'Edit'; //o data set está sendo editado
     dsInsert: S := 'Insert';  //o data set está recebendo um novo registro
   end;

   Label1.Caption := S;
end;
```

DataControls – Componentes de Controle aos Dados

- **TDBNavigator**

O DBNavigator permite que o usuário realize operações padrão de controle de dados. Cada um dos botões do DBNavigator chama um método do componente SQLQuery ao qual está ligado.



Podemos personalizar o DBNavigator usando as suas propriedades e eventos.

Propriedades	**Descrição**
VisibleButtons	Define os botões que serão visíveis
Hints	Hints exibidos pelos botões
ConfirmDelete	Define se será solicitado uma confirmação antes da exclusão
Eventos	**Descrição**
BeforeAction	Quando um botão do Navigator é pressionado, antes da ação ser executada
OnClick	Quando um botão do Navigator é pressionado, depois da ação ser executada

- **TDBGrid**

Mostra os registros de uma consulta na forma de uma grade ou tabela (cada coluna é um campo e cada linha um registro).

Propriedades	**Descrição**
Columns	Lista do tipo TGridColumns, com as colunas da Grid, cada item da lista é do tipo TColumn
Options	Set com as opções da Grid, como ConfirmDelete, MultiSelect, ColumnResize
SelectedField	Campo da coluna selecionada
SelectedIndex	Índice da coluna selecionada
SelectedRows	Lista do tipo TBookmarkList, com os registros selecionados em uma Grid com MultiSelect
TitleFont	Fonte do título das colunas
FixedColor	Cor Fixa, usada nas colunas e indicadores
Eventos	**Descrição**
OnCellClick	Ao clicar em uma célula da Grid
OnColEnter	Quando uma célula de alguma coluna da Grid recebe o foco
OnColExit	Quando uma célula de alguma coluna da Grid perde o foco
OnColumnMoved	Quando o usuário mover uma coluna
OnEditButtonClick	Ao clicar no botão de edição de uma célula, mostrado pela propriedade ButtonStyle da coluna
OnTitleClick	Ao clicar no título das colunas

- **TColumn**

Representa uma coluna do DBGrid, é um item da lista TGridColumns (propriedade Columns).

Propriedades	**Descrição**
ButtonStyle	Botão mostrado ao editar as células da coluna
FieldName	Nome do campo ligado à coluna
PickList	Tstrings com os itens da lista DropDown usada nas células da coluna
Title	Propriedade do tipo TColumnTitle com as opções do título da coluna

- **TDBText, TDBEdit, TDBMemo, TDBListBox, TDBComboBox, TDBImage, TDBCalendar**
  Controles genéricos ligados a um campo de uma tabela.

Propriedades	Descrição
DataField	Campo ao qual o controle está ligado

- **TDBCheckBox**
  Usado em campos que podem receber apenas dois valores, como campos lógicos.

Propriedades	Descrição
ValueChecked	Valor a ser armazenado quando está selecionado
ValueUnchecked	Valor a ser armazenado quando não está selecionado

- **TDBRadioGroup**
  Mostra algumas opções para o preenchimento de um campo.

Propriedades	Descrição
Values	Valor a ser armazenado para cada botão de rádio

- **TDBLookupListBox, TDBLookupComboBox**
  Preenche um campo com dados contidos em outra tabela.

Propriedades	Descrição
ListSource	DataSource que contém os valores a serem exibidos na lista
ListField	Campo de ListSource que será exibido
KeyField	Campo de ListSource usado no relacionamento

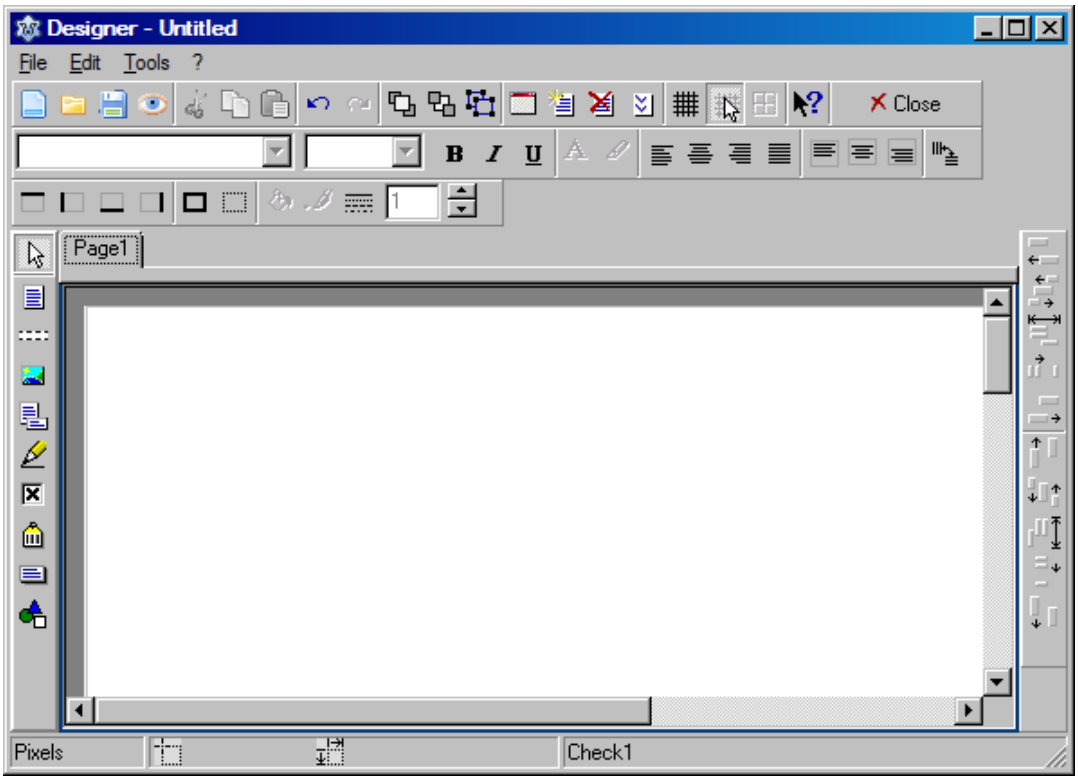
CAPÍTULO VI – Relatórios e Comandos de Impressão

Existem diferentes componentes para construção de relatórios no Lazarus. Normalmente a instalação padrão do Lazarus é acompanhada pelo pacote de componentes LazReport (Report Generator for Lazarus).

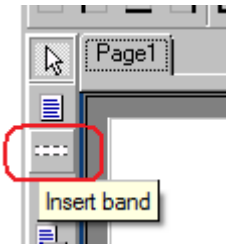
LazReport – Conceitos Básicos

O funcionamento do LazReport é relativamente simples, um arquivo template é gerado com o layout do relatório que é construído em tempo de execução e este mesmo arquivo é carregado no momento da exibição do relatório. Um relatório é organizado em um conjunto de faixas de informação (Band) sobre as quais são inseridos campos de informação variável (dados de um registro do banco de dados, campos calculados, texto fixo, expressões condicionais, etc.).

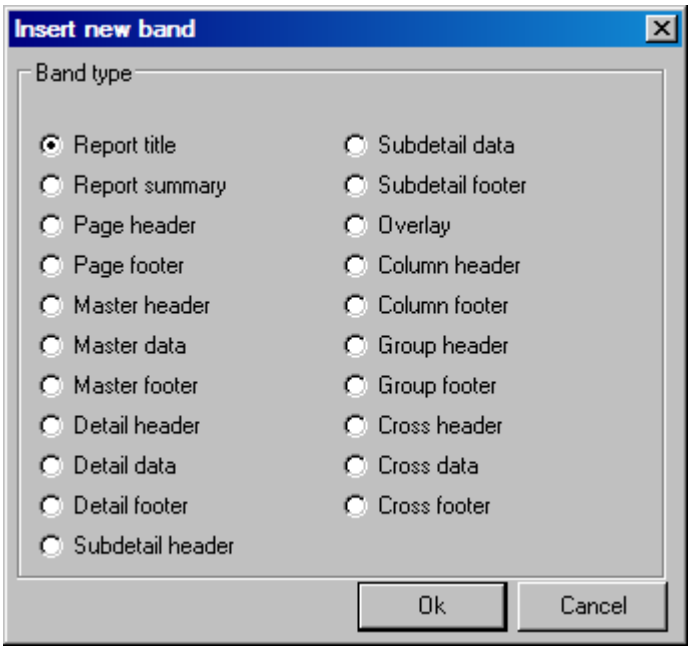
O principal componente para construção de relatórios usando LazReport é o componente TfrReport. Um relatório feito em LazReport é um arquivo com extensão (lrf – Lazarus Report Form), logo, não é necessária a utilização de vários componentes TfrReport em um mesmo projeto (um para cada relatório, por exemplo), uma vez que o relatório não fica armazenado no componente. Ao clicar duas vezes sobre o componente TfrReport será exibida a janela de Designer do relatório:



Um relatório recém-criado é basicamente uma folha em branco onde serão inseridas faixas de conteúdo e campos de dados. Uma faixa de conteúdo (Band) define uma área no relatório que seguirá um determinado comportamento padrão, por exemplo, será impressa no início de cada página, ou então, será impressa ao final do relatório. Para inserir uma faixa de conteúdo utilizamos a barra de componentes a esquerda, escolhendo o segundo botão:

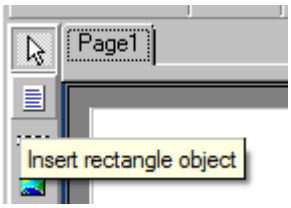


Cada Band (faixa de conteúdo) possui um comportamento específico, logo, não importa o local onde a mesma é adicionada na página ela irá obedecer prioritariamente seu comportamento. Antes de adicionar uma faixa de conteúdo é necessário indicar o seu tipo de comportamento, uma janela será exibida para que a escolha seja feita:

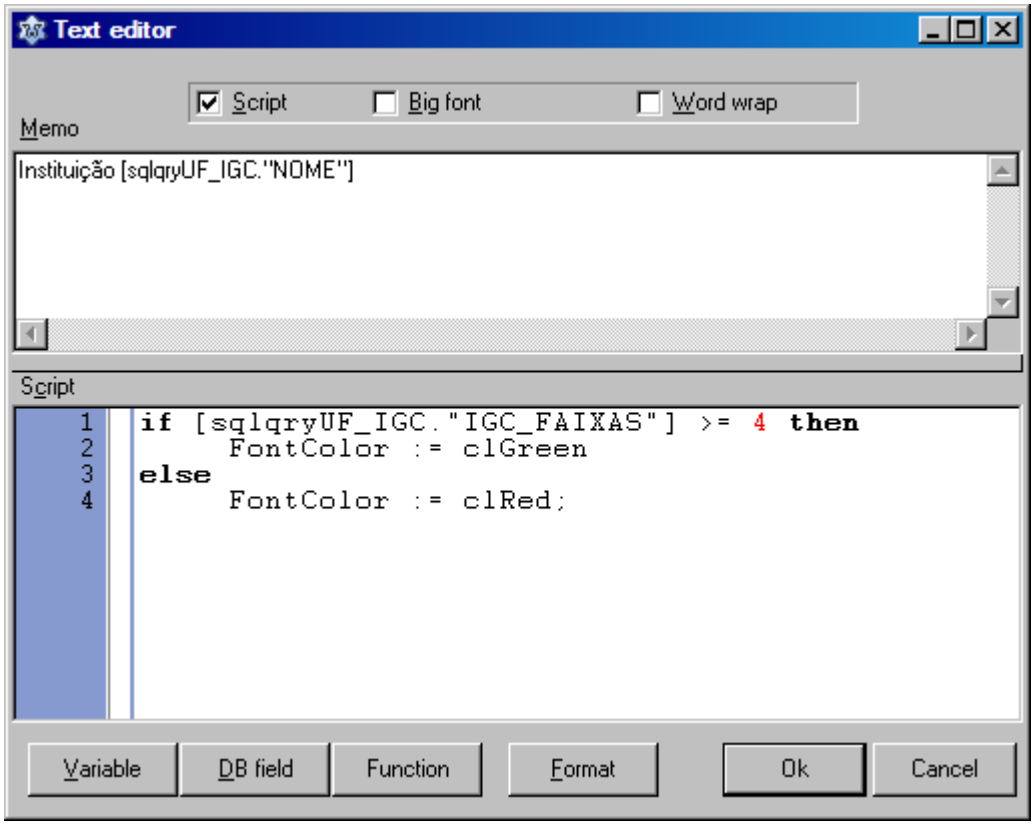


Tipo de faixa	Utilização/Comportamento
Report title	Impresso no início do relatório (apenas na primeira página)
Report summary	Impresso no final do relatório (apenas na última página e não necessariamente na parte inferior da mesma)
Page header	Impresso no início de cada página
Page footer	Impresso ao final de cada página
Master header	Impresso no início do primeiro nível de dados
Master data	Dados do primeiro nível (repetidos para cada registro mestre)
Master footer	Impresso ao final do primeiro nível de dados
Detail header	Impresso no início do segundo nível de dados
Detail data	Dados do segundo nível (repetidos para cada registro detalhe)
Detail footer	Impresso ao final do segundo nível de dados
Subdetail header	Impresso no início do terceiro nível de dados
Subdetail data	Dados do terceiro nível (repetidos para cada registro sub-detalhe)
Subdetail footer	Impresso ao final do terceiro nível de dados
Overlay	Impresso em cada página usando uma camada inferior da página (utilizado para marcas d'água)
Column Header	Título das colunas, utilizado geralmente em relatórios em formato de tabela onde é necessário indicar o título de cada coluna (impresso no início de cada página, após o cabeçalho da página)
Group Header	Banda agrupadora (agrupa registros ordenados por determinado campo). Impressa sempre que o valor do campo que é condição para o grupo é alterado.
Group Footer	Impresso após os dados agrupados, antes do valor do campo que é condição para o grupo ser alterado.
Cross Header	Esse grupo de bandas foi projetado para criar relatórios de tabulação cruzada, que têm uma quantidade variável de colunas na página.
Cross Data	
Cross Footer	

Sobre as Bands são adicionados objetos retângulo. Um objeto retângulo é normalmente um texto organizado com expressões fixas, variáveis e campos de uma consulta. A figura abaixo demonstra como inserir um objeto retângulo:



A janela a seguir é exibida logo que o objeto retângulo é incluído sobre a faixa de conteúdo:

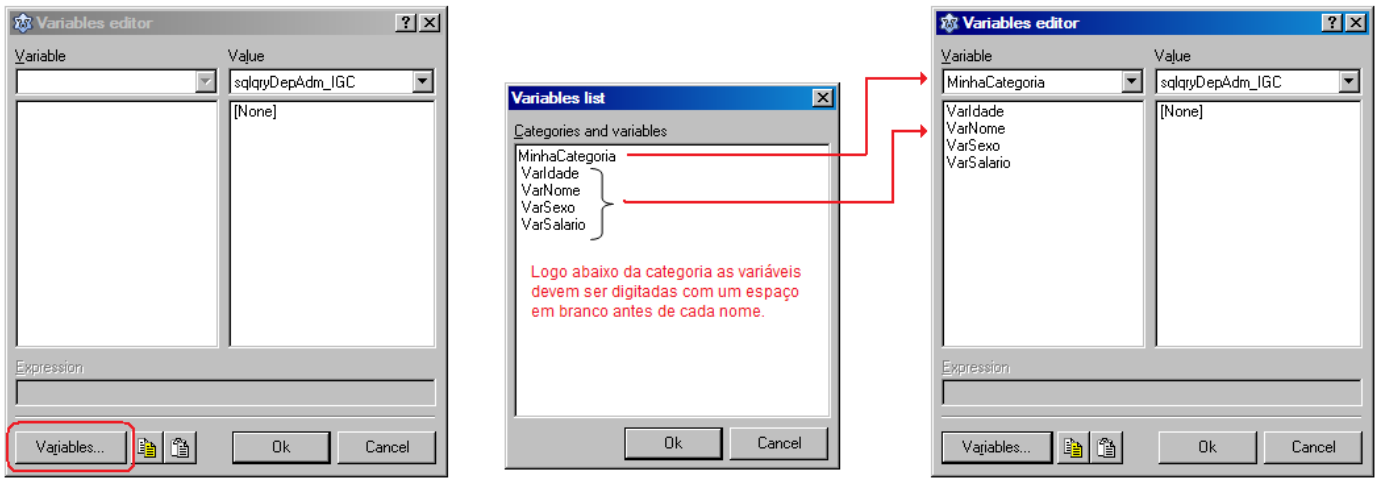


Podem compor o conteúdo de um objeto retângulo dados estáticos como a palavra “Instituição” na figura acima, campos de uma consulta como o campo “NOME” da consulta “sqlqryUF_IGC” além de variáveis pré-definidas:

Tipo de faixa	Utilização/Comportamento
[PAGE#]	Número da página
[DATE]	Data atual
[TIME]	Hora atual
[LINE#]	Número da linha (considerando o conjunto de dados que está sendo percorrido)
[LINETHROUGH#]	Numero da linha geral
[COLUMN#]	Número da coluna
[CURRENT#]	Numero da linha atual
[TOTALPAGES]	Total de páginas

Utilização de variáveis do relatório

Além de usar textos fixos, campos de uma consulta ou variáveis pré-definidas também é possível definir outras variáveis que podem ser controladas através de código. Para fazer isso, na tela de *Designer* do relatório escolhe-se o menu “File/Variables list ...”. A seguir, na janela que surge clica-se sobre o botão “Variables” e na janela “Variables List” digita-se inicialmente um nome de categoria para as variáveis seguido dos nomes de variáveis, sendo que cada nome deve obrigatoriamente iniciar por um espaço em branco. Depois de concluído, na janela de escolha das variáveis as mesmas aparecerão agrupadas por categoria.



Para informar os dados de uma variável utiliza-se o evento OnGetValue do componente TfrReport. Este evento recebe dois parâmetros: ParName (que se refere ao nome da variável que se quer obter o valor) e ParValue (que se refere ao valor que deverá ser atribuído a variável). O código atribui a variável VarNome o conteúdo do campo EditNome de um formulário de dados:

```
procedure TForm1.frReport1GetValue(const ParName: String; var ParValue: Variant);
begin

  if ParName = 'VarNome' then
      ParValue:= EdtNome.Text;

end;
```

Diferentemente dos scripts, os dados de variáveis definidas pelo usuário somente podem ser testados/visualizados em tempo de execução.

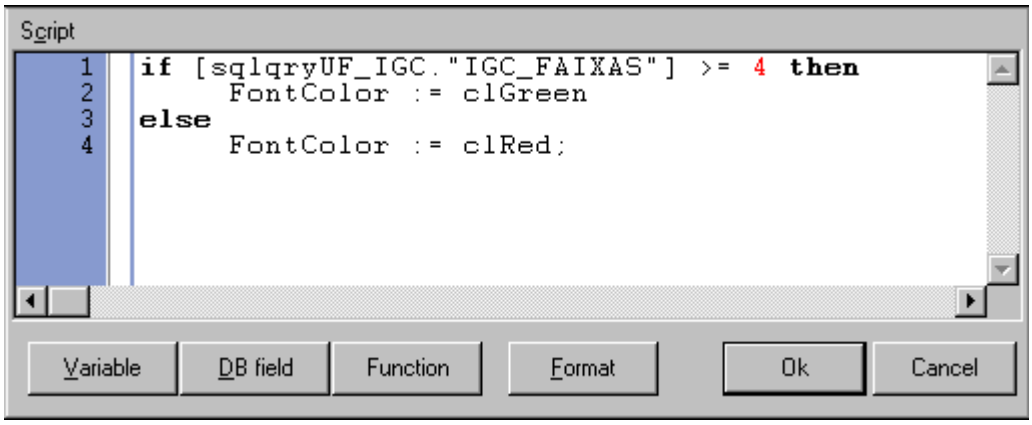
Utilizando Scripts

Um objeto retângulo pode ter suas propriedades modificadas por meio da utilização de scripts. Para utilizar um script a opção “Script” da janela “Text Editor” deve estar selecionada. Um script LazReport é baseado um Pascal e possui as seguintes capacidades:

- Operadores (>, <, =, >=, <=, <>);
- Atribuição (por meio do operador de atribuição (:=);
- Desvios condicionais (por meio de estruturas if ... then ... else)
- Laços de repetição (while ... do e repeat ... until)
- Delimitadores de blocos (begin e end);
- Utilização de variáveis temporárias não tipadas;

- Acesso a alguns objetos do LazReport e suas propriedades:
    - Visible - determina a visibilidade do objeto (boolean);
    - FillColor - determina a cor de fundo do objeto. Valores possíveis: clRed, clBlue, etc.;
    - Text - conteúdo do memo;
    - FontName, FontSize, FontColor - nome, tamanho e cor da fonte;
    - FontStyle - estilo da fonte (1 - itálico, 2 - negrito e 3 - sublinhado);
    - Adjust - alinhamento do texto (1 - direita, 2 - centro).

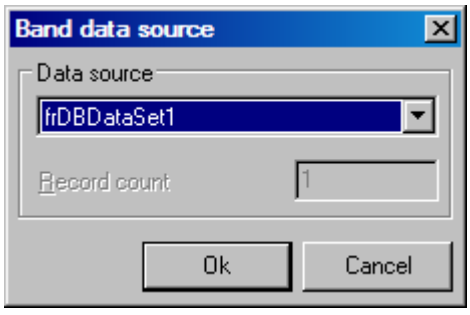
A cada vez que um objeto retângulo é impresso, o script associado a ele é executado. A janela abaixo demonstra um script LazReport que testa se o valor do campo IGC_Faixa for maior ou igual a 4 e neste caso troca a cor da fonte para Verde (clGreen) e em não sendo troca para vermelho).



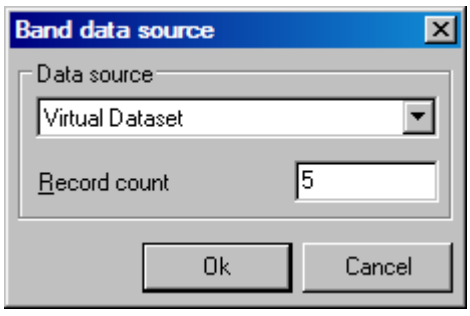
Relatórios baseados em consultas de banco de dados

Relatórios que acessam bancos de dados precisam de alguns cuidados especiais. O primeiro deles é a utilização de um componente TfrDBDataSet (guia LazReport). O componente TfrDBDataSet nada mais é do que uma extensão da consulta (TSQLQuery), é utilizado por motivos de compatibilidade com outras bibliotecas de relatórios. Cada consulta que será utilizada no relatório irá precisar de um TfrDBDataSet com a propriedade DataSource configurada.

Na tela de construção do relatório existem algumas faixas de conteúdo (Bands) com comportamentos específicos para utilização de dados de consultas: Master data, Detail data e Subdetail data. Essas faixas são impressas uma vez para cada registro das consultas aos quais estão ligadas. A ligação entre a faixa de conteúdo (Band) e a consulta é feita por meio da componente TfrDBDataSet. Ao adicionar uma faixa de conteúdo baseada em banco de dados será necessário indicar qual componente TfrDBDataSet irá controlar o comportamento da faixa:



Um tipo especial de DataSource pode ser utilizado quando não temos um específico, neste caso na tela de escolha “Band data source” deve-se indicar “” assim como a quantidade de linhas que deverão ser criadas para essa faixa no relatório.



Visualização e Impressão

Conforme detalhado anteriormente, um relatório feito em LazReport é salvo separadamente da aplicação como um arquivo (lrf) que precisa necessariamente ser distribuído junto com o arquivo executável. Para fazer a impressão de um relatório precisamos inicialmente carregar o arquivo que contém o relatório:

```
frReport1.LoadFromFile('relatorio1.lrf');
```

Para exibir um relatório já carregado pode-se utilizar o método ShowReport:

```
frReport1.ShowReport;
```

Referências Bibliográficas

CANTÚ, M. **Dominando o Delphi 2005**: **A Bíblia**. Prentice Hall, 2006.

CANTÚ, M. **Dominando o Delphi 5: A Bíblia**. Makron Books, 2000.

CANTÙ, M. **Essential Pascal**. 4ª Edição. Paperback, 2008.

EVARISTO, J. **Programando com Pascal**. 2ª Edição. Ed. Book Express, 2004.

MANZANO, J. A.; YAMATUMI, W. Y. **Free Pascal - Programação de Computadores**. Ed. Érica, 2007.

SOUZA, V. A. **Programando com o Lazarus**. Cerne Tecnologia e Treinamento Ltda, 2010.

Sites Especializados (em português)

http://lazarusbrasil.org

http://lazarus.codigolivre.org.br

http://professorcarlos.blogspot.com/

http://wiki.lazarus.freepascal.org/Lazarus_Tutorial/pt