

1. O DEPURADOR INTEGRADO (DEBUGGER).....	2
BREAKPOINTS, STEP OVER E TOOL TIP SYMBOL INSIGHT.....	2
TRACE INTO E RUN UNTIL RETURN	3
MAIS OPÇÕES DE BREAKPOINTS	3
WATCH LIST, LOCAL VARIABLES E EVALUATE/MODIFY	4
CALL STACK	5
DEBUG INSPECTOR.....	5
2. EXCEÇÕES EM DELPHI.....	5
TRATAMENTO DE EXCEÇÃO NO DELPHI	5
TRY FINALLY	6
TRATAMENTO DE EXCEÇÃO.....	6
OPÇÕES DO DEBUGGER	7
TRY EXCEPT	7
PROPAGANDO EXCEÇÕES - RAISE	9
CALL STACK E EXCEÇÕES.....	10
TRATAMENTO GLOBAL DE EXCEÇÃO	11
TRY EXCEPT FINALLY.....	11
CUSTOM EXCEPTIONS	12
INSTÂNCIA DO OBJETO EXCEÇÃO	12
EXCEÇÕES SILENCIOSAS - EABORT	13
3.PROGRAMAÇÃO ORIENTADA A OBJETOS	13
EXEMPLO PRÁTICO.....	13
CRIANDO MÉTODOS	15
HERANÇA.....	15
MÉTODOS ESTÁTICOS	16
MÉTODOS VIRTUAIS E DINÂMICOS - POLIMORFISMO.....	17
INHERITED	17
MÉTODOS ABSTRATOS.....	18
MAIS POLIMORFISMO	19
CONSTRUTORES E DESTRUTORES.....	19
TYPE CASTING E RTTI.....	20
PROPRIEDADES	21
DEFAULT	21
MÉTODOS GET/SET.....	21
OVERLOAD	22
4. COMPONENTES E PACOTES.....	22
CRIANDO UM PACOTE.....	23
CRIANDO UM COMPONENTE.....	23
CRIANDO UMA PROPRIEDADE.....	23
INSTALANDO O PACOTE.....	24
CRIANDO UM ÍCONE PARA O COMPONENTE	24
TESTANDO	24
LIBRARY PATH.....	25
DE CLASSES A COMPONENTES	25
REGISTRANDO COMPONENTES	25
POLIMORFISMO E ABSTRAÇÃO EM COMPONENTES	26
NOTIFICATION.....	27
EVENTOS.....	27
OPÇÕES DO PACOTE.....	28
CONTAINS E REQUIRES	28
USANDO PACOTES EM TEMPO DE EXECUÇÃO.....	30

PACOTES DE DESIGN-TIME E DE RUNTIME	31
HERANÇA VISUAL DE FORMULÁRIO (VISUAL FORM INHERITANCE)	31
CRIANDO COMPONENTES EM TEMPO DE EXECUÇÃO (RUN-TIME)	33
MODELMAKER BÁSICO	33
A VCL	35
5. THREADS	35
MULTITHREAD - PROGRAMAÇÃO CONCORRENTE	35
THREADS E PROCESSOS	35
PROCESSOS	35
CRIANDO UM CONTADOR	36
O THREAD PRINCIPAL	36
CRIANDO THREADS	37
A CLASSE TTHREAD DO DELPHI	37
SYNCHRONIZE	39
MUTEX	40
SEMÁFOROS	41
SEÇÕES CRÍTICAS	42

1. O Depurador Integrado (Debugger)


BreakPoints, Step Over e Tooltip Symbol Insight

Certamente o recurso mais simples do *Debugger* é o uso de *BreakPoints*, ou pontos de interrupção. Você pode definir um ponto de parada no código fonte de sua aplicação, de modo que a execução inteira do programa é suspensa quando determinada linha de código marcada está pronta para ser executada. Isto nos permite monitorar detalhadamente o fluxo de execução do código, examinando valores de variáveis e chamadas de procedures.

Como exemplo, inicie uma nova aplicação, dando o nome de “FrmExemplo” ao formulário e salvando-o na unit “uFrmFormulario.pas”. Dê ao projeto o nome de “Debugger.dpr”.

no evento *OnClick* de um botão coloque o código abaixo, o qual monta uma string de 10 letras randomizadas ao acaso, como ‘ABCDEFGHIJ’.

```
procedure TForm1.Button1Click(Sender: TObject);
var
    loop : integer;
    str : string;
begin
    randomize;
    for loop:=1 to 10 do
        str:=str+chr(Random(26)+65); // adiciona uma letra de A a Z em str
    end;
```

Imagine agora que quiséssemos observar a execução deste código a cada iteração do loop for. Para isso podemos clicar ao lado esquerdo da linha de código (na parte cinza) onde queremos marcar a interrupção. Isto adiciona um *BreakPoint*, marcando a linha em vermelho por padrão. Ao executar a aplicação e clicar no botão, o *Debugger* interrompe a execução do programa e indica com uma seta verde a posição de parada, sobre o *BreakPoint*. Vamos executar a operação passo a passo, para observar o fluxo de execução do loop. Para isso, escolha a opção *Run|Step Over*  do menu do Delphi ou simplesmente pressione F8. Assim, o *Debugger* passa para a próxima linha compilada, indicando com uma linha azul por padrão. Agora você pode pressionar a tecla F8 mais algumas vezes, e observar a montagem da variável *str*. Para continuar a execução do programa ou passar para o próximo *BreakPoint*, pressione F9.

Para saber o valor atual de *str*, basta repousar o mouse sobre qualquer parte do código que contenha uma referência à variável. Neste caso, seu valor é mostrado em um “hint”. Este recurso é conhecido como *Tooltip symbol insight*.

[g1] Comentário: Depurado r dos probes - ShowMessage('Olá passei por aqui!');

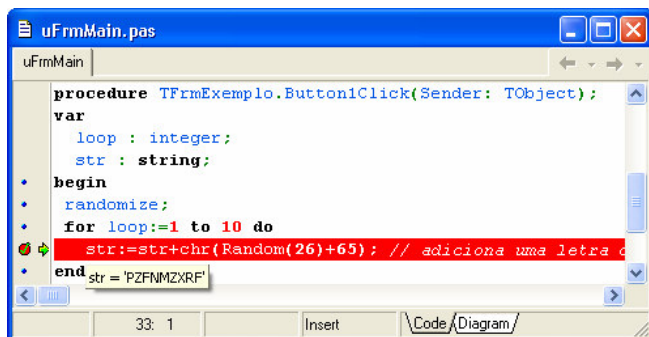


Figura. Usando um BreakPoint e observando o valor de variáveis no hint suspensão

Utilize este recurso para observar não somente o valor de uma variável simples, mas valores de *TFields* de *datasets* em bancos de dados, instruções SQL, propriedades de componentes visuais, etc.

Atenção: Observe que após compilar a aplicação, o depurador indica com algumas marcações em azul a esquerda, quais linhas foram usadas para montar o código compilado final. Alguns códigos não possuem esta marcação, significando que foram removidos pelo otimizador do compilador. Você não pode marcar BreakPoints sobre estes códigos que não foram compilados.



Trace Into e Run Until Return

Para examinar estas opções, modifique nosso código colocando a atribuição da variável *str* em um procedure separado, como mostra o código a seguir:


```
procedure MontarStr (var str : string);
begin
  str:=str+chr(Random(26)+65);
end;
```

Substitua a linha de atribuição de *str* por:

```
MontarStr(str);
```

Deixe o *BreakPoint* sobre *MontarStr* abaixo do *for* e execute a aplicação. Pressione o botão para observarmos o fluxo do código. Se você executar passo a passo com F8, notará que quando passamos pela instrução *MontarStr*, o *Debugger* não entra no corpo do procedure. Para que isso aconteça, quando a linha de execução estiver sobre a instrução *MontarStr*, pressione F7 (ou use o menu *Run|Trace Into* ) , fazendo com que o *Debugger* entre no procedure ou função. Você pode agora continuar pressionando F8 para ter uma execução passo a passo. Em um grande sistema com vários procedures, você vai querer que o *Debugger* entre em alguns trechos de código e outros não. Se você acidentalmente pressionar F7 ao invés de F8, e entrar em um procedure, não precisará ficar pressionando F8 até chegar ao final do procedure. Para isto basta retornar a execução ao ponto de chamada, usando o menu *Run|Run Until Return* , ou pressionando Shift+F8.

Mais opções de BreakPoints

Se você estiver depurando uma grande aplicação, talvez fique difícil manipular muitos *BreakPoints* e localizá-los em meio a várias units. Para isso, o Delphi oferece um *BreakPoint List* no menu *View|Debug Windows|BreakPoints* , que você pode usar para gerenciar os *BreakPoints* do seu programa.

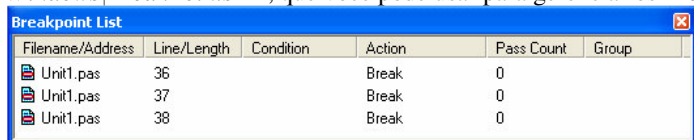


Figura. Visualizando os BreakPoints da aplicação usando o BreakPoint List

Um recurso muito interessante do *Debugger* é permitir a personalização dos *BreakPoints* (isso foi introduzido no Delphi 5, versão que trouxe muitas inovações nos recursos do *Debugger*). Para personalizar

um *BreakPoint*, dê um clique de direita sobre ele no *BreakPoint List* e escolha *Properties*. Se você estiver no editor de código fonte, clique de direita sobre a marcação de *BreakPoint* a esquerda do código, escolhendo *BreakPoint Properties*. Como exemplo, pegue o código que usamos anteriormente. Imagine agora que por algum motivo você quisesse depurar o loop do *for* somente a partir da quinta iteração, ignorando as primeiras, pois constatou que algum bug poderia estar acontecendo somente naquela situação. Então você pode editar as propriedades do *BreakPoint* e digitar “loop>=5” na opção *Condition*, como mostra a figura:

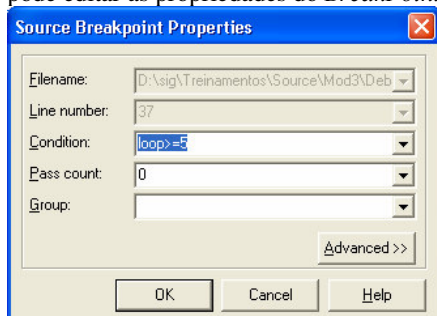


Figura. Editando as propriedades de um BreakPoint

Desta forma, quando o depurador analisar o *BreakPoint*, fará um teste na condição deste ponto de interrupção. Se o teste falhar, continuará o fluxo de execução, senão, interrompe a execução para a depuração. Você também pode usar a propriedade *Pass Count* do *BreakPoint* para indicar em qual passagem o depurador deve parar para iniciar a depuração.

Ainda temos opções mais avançadas neste editor de propriedades, como uso de grupos de pontos de interrupção, personalização da ação do *BreakPoint*, ativar / desativar *BreakPoints* ou grupos subsequentes. Para visualizar estas opções, clique no botão *Advanced* da caixa de propriedades.

Watch List, Local Variables e Evaluate/Modify

O depurador inclui a *Watch List* que é uma janela especial utilizada para visualizar valores de variáveis. Para visualizá-la, clique em *View|Debug Windows|Watches*. Agora você pode dar um clique de direita sobre a janela e escolher a opção *Add Watch* para adicionar a variável, componente ou objeto o qual você deseja monitorar. Usando o exemplo anterior, adicionei a variável *str* a janela *Watch*. Com o *BreakPoint* ativado e executando a aplicação passo a passo, podemos observar os valores de *str* durante toda a sua vida útil dentro do escopo do *procedure*.

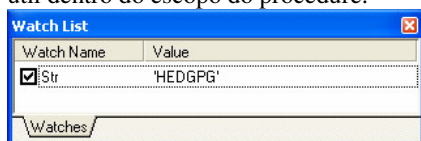


Figura. Usando uma Watch List para monitorar a variável str

Dica: Quando você estiver no editor de código, basta clicar de direita sobre a variável que quiser adicionar a *Watch List*, escolhendo a opção *Debug|Add Watch at Cursor*

Outra janela do depurador é a *Local Variables*, que você pode usar para facilmente visualizar os valores de todas as variáveis locais ao *procedure* que está sendo depurado. Isto é muito útil em um *procedure* que possui muitas variáveis. Para visualizá-la, clique em *View|Debug Windows|Local Variables*. A figura a seguir mostra as variáveis do evento *OnClick*. *Sender* (que é *Button1*) e *Self* (que é o *Form1*) não estão acessíveis ao depurador neste ponto do código devido a otimização.

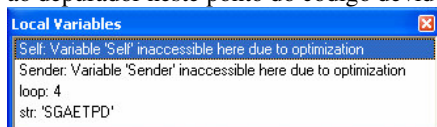


Figura. Usando Local Variables para monitorar as variáveis locais

Você pode simular situações em tempo de execução através da troca manual de valores de variáveis, propriedades, objetos, etc. No exemplo anterior, adicionamos uma *string* no final de *str* para observar o seu comportamento durante o término da execução. Para modificar o valor de uma variável, durante a depuração,

clique de direita sobre o código que referencia a variável e escolha a opção *Debug|Evaluate/Modify*. Entre com o novo valor na opção *New value* (o valor deve ser compatível com o tipo da variável).

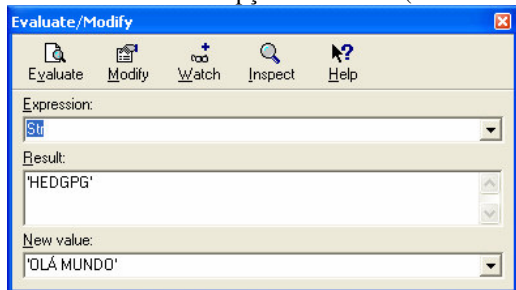


Figura. Modificando valores usando Evaluate/Modify

Call Stack

Uma procedure contém um bloco de código que executa instruções, que podem incluir a chamada de um outro procedure ou função, que conseqüentemente pode chamar outro procedure, e assim por diante. No final teremos uma pilha (*stack*) de chamadas de procedures, que você pode desejar monitorar para ver exatamente que procedures e endereços de chamada estão sendo empilhados. Para isso use o menu *View|Debug Windows|Call Stack*. Como exemplo, mostrei a janela *Call Stack* no momento em que o procedure *MontarStr* era chamado, e podemos observar que o procedure *Button1Click* já está na pilha de chamada, juntamente com *Debugger* que é o programa principal. Quando a chamada de procedures sobrecarrega a pilha, ou ocorre a chamada de um procedure de forma recursiva e infinita, acontece um erro conhecido como “Stack Overflow”.

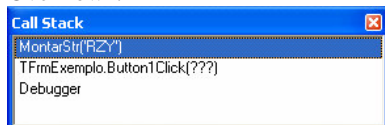


Figura. Monitorando a pilha de chamada de procedures

Debug Inspector

Este é um importante recurso do *Debugger*, semelhante a um *Object Inspector* em tempo de execução. Nele você pode observar uma variável ou componente da VCL, acessando todas as suas propriedades (inclusive as herdadas e não publicadas) e endereços de métodos. Você pode também trocar valores de variáveis. Como exemplo chamei a janela *Inspect* para o formulário *FrmExemplo*, simplesmente clicando de direita sobre sua declaração no editor de código, e escolhendo a opção *Debug|Inspect*.

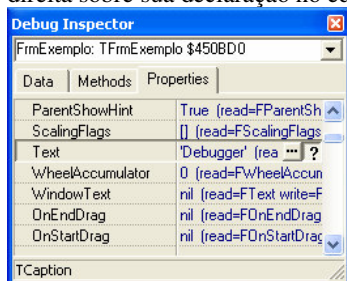


Figura. Usando a janela Inspect para o FrmExemplo

A introdução ao debugger vista anteriormente foi baseada no artigo de Guinther P. – Revista ClubeDelphi nº 28

2. Exceções em Delphi

Tratamento de exceção no Delphi

- Manipulação de exceção para proteção de recursos (*try finally*);
- Manipulação de erros de tempo de execução (*try except*);

[g2] Comentário: Falar que antigamente se usava `IORESULT`

Try Finally

```
try
  <instrução>;
  <instrução>;
finally
  <instrução>;
  <instrução>;
end;
```

Atenção: Não importa o que aconteça dentro do bloco *try*, o código contido em *finally* será sempre executado, mesmo que nenhuma exceção ocorra.

Blocos *try finally*, além de serem usados para proteger a alocação de recursos, podem ser usados para devolver o estado original a um objeto (código de limpeza). Observe no exemplo a seguir como proteger a criação de um formulário para que sempre seja destruído:

Antes:

```
Form1:=TForm1.Create(Application);
Form1.ShowModal; // pode causar uma exceção
Form1.Free; // pode nunca ser executado
```

Depois:

```
Form1:=TForm1.Create(Application);
try
  Form1.ShowModal; // pode causar uma exceção
finally
  Form1.Free; // sempre será executado
end;
```

No módulo 1 usamos o comando *DisableControls* de *TDataSet* sem usar *try finally*. Se alguma exceção ocorrer durante a varredura, os controles nunca seriam habilitados. Aqui está a versão protegida do mesmo código:

```
procedure TDM.AumentarMensalidadeAlunos(APercentual: single);
begin
  with cds_alunos do
  begin
    DisableControls;
    try
      // loop
    finally
      EnableControls;
    end;
  end;
end;
```

Exercício: Se uma exceção ocorrer na instrução 2, que outras instruções serão executadas?

```
procedure Foo;
begin
  Instrução1;
  try
    Instrução2;
    Instrução3;
  finally
    Instrução4;
    Instrução5;
  end;
  Instrução6;
end;
```

Tratamento de Exceção

Para exemplificar o tratamento de exceções, faça um pequeno exemplo. Crie uma nova aplicação e construa o exemplo abaixo (salve o projeto como “TryExcept.dpr” e o formulário como “uFrmMain.pas”):

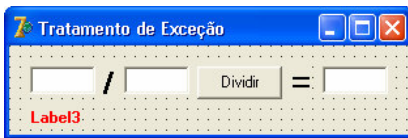


Figura. Exemplo simples mostra como tratar exceções

```
procedure TFrmMain.Button1Click(Sender: TObject);
var
  x, y, r : single;
begin
  x:=StrToFloat(Edit1.Text);
  y:=StrToFloat(Edit2.Text);
  r:=x/y;
  Edit3.Text:=FloatToStr(r);
end;
```

Execute a aplicação. Entre com uma divisão por zero.

Opções do Debugger

Você pode habilitar/desabilitar as mensagens do depurador através do menu *Tools|Debugger Options>Language Exceptions>Stop On Delphi Exceptions*. Você pode também desabilitar todo o depurador.

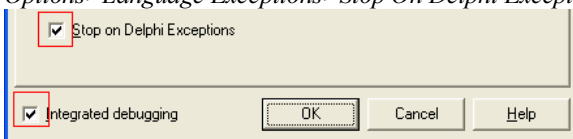


Figura. Opções do Depurador

Por exemplo, se *Stop on Delphi Exceptions* estiver selecionado, o ambiente irá interceptar as exceções antes que sua aplicação o faça. Você verá uma caixa de diálogo gerada pelo *Debugger* do Delphi, como esta:

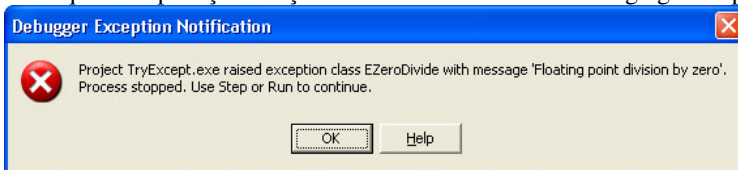


Figura. Notificação do depurador do Delphi

Em seguida, você verá a caixa de diálogo a seguir (caso você não tenha feito ainda o tratamento). Mas caso *Stop on Delphi Exceptions* esteja desmarcada, você verá somente a próxima caixa de diálogo:

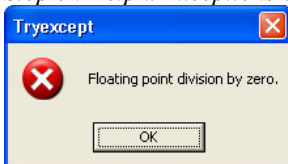


Figura. Manipulador de exceção da Aplicação

Note que isto se aplica somente a aplicações sendo executadas dentro do ambiente do Delphi. Caso seja executado por fora do Delphi, você verá somente a segunda caixa de diálogo.

Para este exemplo deixe essa opção **desmarcada**.

Try Except

Faz o tratamento (manipulação) de exceções.

```
try
  <instrução>;
  <instrução>;
except
  <instrução>;
end;
```

Agora vamos tratar (com *try except*) o código do botão:

```

procedure TFrmMain.Button1Click(Sender: TObject);
var
  x,y,r : single;
begin
  try
    x:=StrToFloat(Edit1.Text);
    y:=StrToFloat(Edit2.Text);
    r:=x/y;
  except
    on E : EConvertError do
      begin
        r:=0;
        Label3.caption:='Valores inválidos';
      end;
    on E : EZeroDivide do
      begin
        r:=0;
        Label3.caption:='Divisão por zero';
      end;
    on E : EMathError do
      begin
        r:=0;
        Label3.caption:='Erro matemático geral';
      end;
    end;
  end;
  Edit3.text:=FloatToStr(r);
end;

```

“E” é a instância da exceção levantada.

Dica: O nome da instância pode ser mudado, não precisa necessariamente ser “E”.

Dica: Você pode usar **else** no *except*.

Usando **On**, testamos se a instância E é de determinado tipo e fazemos o tratamento específico para aquele tipo de erro.

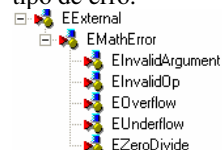


Figura. Classes de Exceção EMathError

Atenção: Sempre trate tipos mais específicos primeiro e mais básicos por último.

Execute a aplicação, digite um valor 0 para o denominador. Aperte o botão *Dividir*. Você verá a mensagem de erro do depurador (porque *Stop On Delphi Exceptions* está ativado). Seu usuário final nunca vai ver esta mensagem. Aperte F9 para continuar o fluxo de execução e você deverá receber a mensagem de erro tratada, a qual o usuário final receberia:

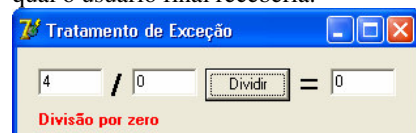


Figura. Exceção tratada

Exercícios:

Se uma exceção do tipo *EConvertError* ocorrer na instrução 2, que outras instruções serão executadas?

```

procedure Foo;
begin
  Instrução1;
  try
    Instrução2;
    Instrução3;
  except
    Instrução4;
  end;
end;

```



```
Instrução5;
end;
```

Se uma exceção do tipo *EConvertError* ocorrer na instrução 2, que outras instruções serão executadas?

```
procedure Foo;
begin
  Instrução1;
  try
    Instrução2;
    Instrução3;
  except
    on E : EMathError do
      Instrução4;
    end;
  Instrução5;
end;
```

Propagando Exceções - Raise

- *raise* pode ser utilizado para gerar uma exceção;
- *raise* pode propagar uma exceção para a rotina chamadora;

Em nosso exemplo, após assegurarmos um valor válido para o resultado, poderíamos levantar a exceção para que fosse manipulada na rotina chamadora.

```
except
  on E : EConvertError do
  begin
    r:=0;
    raise; // label3.caption:='Valores inválidos';
  end;
```

Exercício – inicie uma nova aplicação Delphi, salve o projeto com “RaiseEx.dpr” e a unit como “uFrmMain.pas”. Coloque um botão com o *Caption* “Raise” e um *Memo*.

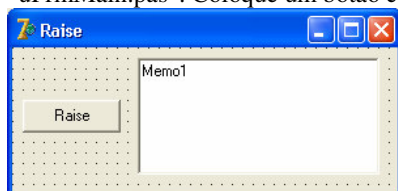


Figura. Exemplo com raise

Declare três métodos (*ProcA*, *ProcB*, *ProcC*, *ProcD*) na seção *public* do formulário. Aperte Shift+Ctrl+C para gerar os cabeçalhos.

```
public
  procedure ProcA;
  procedure ProcB;
  procedure ProcC;
  procedure ProcD;
```

Em nosso exemplo, *ProcD* chama *ProcC*, que chama *ProcB* que chama *ProcA*. Implemente os métodos como mostrado a seguir:

```
procedure TForm1.ProcA;
begin
  Memo1.Lines.Add('ProcA levantou exceção');
  raise Exception.create('ERRO!');
end;

procedure TForm1.ProcB;
begin
  try
    ProcA; // chama A
  except
    Memo1.Lines.Add('ProcB tratou e propagou'); // trata
    raise; // propaga
  end;
end;
```

```

procedure TForm1.ProcC;
begin
  try
    ProcB; // chama B
  except
    Memo1.Lines.Add('ProcC tratou e propagou'); // trata
    raise; // propaga
  end;
end;

procedure TForm1.ProcD;
begin
  try
    ProcC; // chama C
  except
    Memo1.Lines.Add('ProcD tratou e NÃO propagou'); // trata e não propaga
  end;
end;

```

No evento *OnClick* chame *ProcD*:

```

try
  ProcD;
except;
  Memo1.Lines.Add('Button1Click tratou e NÃO propagou');
end;

```

Execute a aplicação.

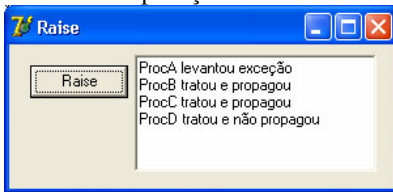


Figura. Propagando exceções

Observe que nunca uma exceção chega em *Button1Click*, pois *ProcD* não gera nem propaga nenhuma exceção. Faça testes agora, retirando e colocando *raise* no tratamento das exceções para examinar como a propagação de exceções funciona. Como exemplo, propague a exceção em *ProcD* e também em *Button1Click*, e siga os passos a seguir.

Call Stack e Exceções

Agora veremos como a propagação de exceções se comporta na pilha de chamadas (*Call Stack*). Coloque um *breakpoint* ao lado do *raise* em *ProcA*.

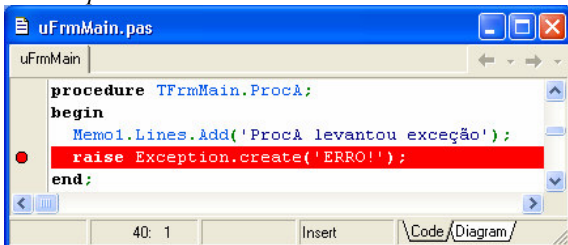


Figura. Gerando uma exceção com raise

Execute a aplicação e clique no botão. Quando o *breakpoint* for disparado clique em *View|Debug Windows|Call Stack*. Observe que as chamadas estão “empilhadas”.

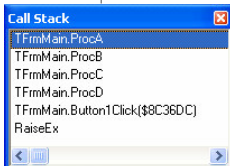


Figura. Observando a exceção na pilha

Como uma exceção foi gerada manualmente em *ProcA*, e exceção vai começar a “descer” na pilha até que alguém faça o seu tratamento. O procedimento de chamada pode tratar e “matar” a exceção para que não desça mais na pilha, ou fazer um tratamento local e propagar.

Aperte F7 para ver a exceção descendo pela pilha e sendo tratada.

Como todos os procedimentos da pilha estão propagando a exceção após o tratamento, a pilha chega ao fim e a exceção não foi “morta”. Dessa forma, a exceção chega até a aplicação que executará o tratamento global (padrão).



Figura. Exceção chega até a aplicação

Tratamento Global de exceção

Você pode definir o comportamento global de uma exceção usando o componente *ApplicationEvents* da paleta *Additional*. Primeiro crie um novo formulário (com o nome de “FrmOnEx” – unit “uFrmOnEx.pas”) e nele coloque uma *Label*. Volte ao primeiro formulário e aperte Alt+F11, escolhendo a unit do segundo formulário. Coloque um *ApplicationEvents* no formulário e no seu evento *OnException* digite:

```
procedure TFrmMain.ApplicationEvents1Exception(Sender: TObject; E: Exception);
begin
  FrmOnEx.Label2.Caption:=E.Message;
  FrmOnEx.ShowModal;
end;
```

Onde *E* é a exceção passada como parâmetro para o evento.

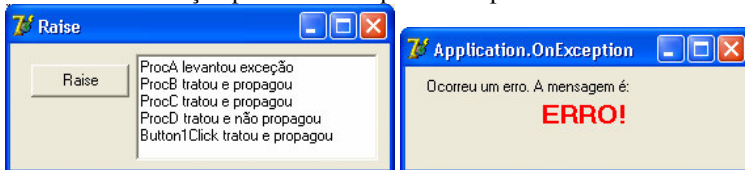


Figura. Tratamento de erro global em Application.OnException

Você pode usar essa técnica para centralizar (padronizar) a saída de erros para a interface de usuário, ou ainda fazer um gerador de logs, etc.

Try Except Finally

Você pode combinar os dois blocos de tratamento, *try finally* e *try except* em uma única estrutura. Observe o exemplo a seguir:

```
procedure TFrmMain.Button1Click(Sender: TObject);
var
  TD : TTransactionDesc;
  Query : TSQLQuery;
begin
  Query:=TSQLQuery.Create(self);
  Query.SQLConnection:=SQLConnection1;
  Try
    TD.TransactionID:=1;
    TD.IsolationLevel:=xilREADCOMMITTED;
    SQLConnection1.StartTransaction(TD);
    try
      Query.SQL.Text:='update TABELA1 set CAMPO=1';
      Query.ExecSQL;
      Query.SQL.Text:='delete from TABELA2';
      Query.ExecSQL;
      Query.ExecSQL;
      SQLConnection1.Commit(TD); // tenta comitar
    except
      SQLConnection1.Rollback(TD); // se der erro desfaz a transação e propaga exceção
      raise;
    end;
  end;
```

```
finally
    Query.free; // sempre libera a Query
end;
end;
```

Custom Exceptions

Inicie uma nova aplicação. Declare sua nova exceção de um tipo descendente de *Exception*.

```
type
    EInvalidImpar = class (Exception);
```

Crie o seguinte formulário:

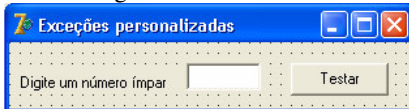


Figura. Testando exceções Personalizadas

No evento *OnClick* do botão digite:

```
if not odd(StrToInt(Edit1.Text)) then
    raise EInvalidImpar.Create('Não é um número ímpar');
```

Uma alternativa é usar *CreateFmt*:

```
raise EInvalidImpar.CreateFmt('%s não é um número ímpar', [Edit1.Text]);
```



Figura. Testando exceções Personalizadas

Instância do Objeto Exceção

Inicie uma nova aplicação. Crie o seguinte formulário:



Figura. Testando exceções Personalizadas

No evento *OnClick* do botão digite:

```
try
    StrToDate(Edit1.Text);
except
    on E : Exception do
        ShowMessage('O Erro gerado pelo Delphi foi : '+E.Message);
end;
```

Onde *E.Message* é a mensagem da exceção. Digite uma Data inválida no *edit* e aperte o botão.

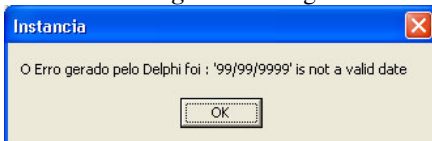


Figura. Tratando a exceção e capturando informações da instância

Atenção: Você não pode liberar a instância da exceção, pois ela é controlada pelo Delphi. Além disso, só pode ser usada abaixo do bloco *except*.

Exceções Silenciosas - EAbort

Use uma exceção silenciosa quando quiser propagar uma exceção pela pilha. A diferença é que ela não causa nenhum efeito na aplicação (final da pilha). Ou seja, nada é exibido para o usuário final. Seu objetivo principal é quebrar uma sequência de eventos.

Use o tipo *EAbort* para levantar uma exceção silenciosa, ou simplesmente use *Abort*.

Ex.: No evento *OnClose* de um novo formulário digite:

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  if MessageDlg('Deseja sair', mtconfirmation, [mbok, mbno], 0) <> mrOk then
    raise EAbort.Create('Aborting...');
end;
```

A linha do *raise* pode ser substituída simplesmente por:

```
Abort;
```

Observe no exemplo anterior que o *raise EAbort* **cancela** o fechamento do formulário.

3. Programação Orientada a Objetos

Abstração

- Habilidade da linguagem em modelar o mundo real;

Encapsulamento

- Uma classe deve ser o mais independente possível de outras classes;
- Classes devem ter uma única função, para permitir a reutilização de código;
- Classes devem ser unidas (ligadas) por um único ponto;

Herança

- Herdar características de outra classe e adicionar capacidades específicas;
- Reutilizar código, sem causar redundância;

Polimorfismo

- Capacidade de classes definirem um comportamento diferente para um método da classe base.

[g3] Comentário: Citar exemplo do vídeo-cassete e televisão, que você não gostaria de levar a TV para arrumar toda vez que estragasse o vídeo, ou que a TV ligasse sozinha toda vez que você abrisse a geladeira

[g4] Comentário: Citar exemplo dos vagões, que são ligados por um único ponto, facilitando a substituição

Exemplo prático

Inicie uma nova aplicação. Dê o nome de “FrmExemplo” ao formulário e salve-lo na unit “uFrmMain.pas”. Dê o nome de “ClassesObjetos” ao projeto. Crie o formulário mostrado na figura a seguir. Para os *Edits*, dê os nomes de *EdtCapCarro*, *EdtQuilmetragem*, *EdtCapAviao* e *EdtHorasVoo*, respectivamente (veja a figura).

Figura. Formulário exibe as propriedades dos objetos

Clique em *File|New|Unit*. Salve a nova unit com o nome de “uCarro.pas”. Usando o mesmo procedimento crie uma unit chamada “uAviao.pas”.

Aqui estão os códigos das units:

```
unit uCarro;

interface
```

```

type
  TCarro = class
    Descricao : string;
    Capacidade : integer;
    Quilometragem : integer;
  end;
implementation
end.

```

```

unit uAviao;

interface

type
  TAviao = class
    Descricao : string;
    Capacidade : integer;
    HorasVoo : integer;
  end;
implementation
end.

```

Volte ao formulário principal e adicione as units *uCarro* e *uAviao* à cláusula *uses* da interface. Declare duas variáveis na seção *public* do formulário.

```

public
  Carro : TCarro;
  Aviao : TAviao;

```

No botão *Criar* (do objeto Carro) digite:

```

procedure TFrmExemplo.BtnCriarCarroClick(Sender: TObject);
begin
  // cria o objeto e inicializa campos conforme valores dos edits
  Carro:=TCarro.Create;
  if EdtDescCarro.Text<>'' then
    Carro.Descricao:=EdtDescCarro.Text;
  if EdtCapCarro.Text<>'' then
    Carro.Capacidade:=StrToIntDef(EdtCapCarro.Text,0);
  if EdtQuilometragem.Text<>'' then
    Carro.Quilometragem:=StrToIntDef(EdtQuilometragem.Text,0);
end;

```

Agora vamos fazer o mesmo para objetos do tipo TAviao. No botão *Criar* (do objeto Avião) digite:

```

procedure TFrmExemplo.BtnCriarAviaoClick(Sender: TObject);
begin
  // cria o objeto e inicia campos conforme valores dos edits
  Aviao:=TAviao.Create;
  if EdtDescAviao.Text<>'' then
    Aviao.Descricao:=EdtDescAviao.Text;
  if EdtCapAviao.Text<>'' then
    Aviao.Capacidade:=StrToIntDef(EdtCapAviao.Text,0);
  if EdtHorasVoo.Text<>'' then
    Aviao.HorasVoo:=StrToIntDef(EdtHorasVoo.Text,0);
end;

```

[g5] Comentário: Falar do uso do "WITH"

No botão *Liberar* (do objeto Carro) digite:

```

procedure TFrmExemplo.BtnLiberarCarroClick(Sender: TObject);
begin
  Carro.Free;
end;

```

No botão *Liberar* (do objeto Avião) digite:

```

procedure TFrmExemplo.BtnLiberarAviaoClick(Sender: TObject);
begin
  Aviao.Free;

```

```
end;
```

Execute a aplicação e teste a criação e destruição dos objetos.

Criando métodos

Abra a declaração de *TCarro* e declare o método *Mover*, como mostrado a seguir:

```
TCarro = class
  Descricao : string;
  Capacidade : integer;
  Quilometragem : integer;
  procedure Mover; // com o cursor sobre essa linha aperte SHIFT+CTRL+C
end;
```

Implemente o método como mostrado a seguir:

```
procedure TCarro.Mover;
begin
  ShowMessage(Descricao+' entrou em movimento.');
```

Declare *dialogs* na cláusula *uses* da unit (*implementation*).

No formulário principal, nas opções do objeto carro, coloque um botão com o *caption* “Mover”. No seu evento *OnClick* digite:

```
procedure TFrmExemplo.BtnMoverCarroClick(Sender: TObject);
begin
  Carro.Mover;
end;
```

Agora repita os mesmos passos para classe *TAviao*. Veja a implementação do método *Mover* para *TAviao*:

```
procedure TAviao.Mover;
begin
  ShowMessage(Descricao+' está voando.');
```

Declara *Dialogs* na cláusula *uses* das duas units. Teste a aplicação.

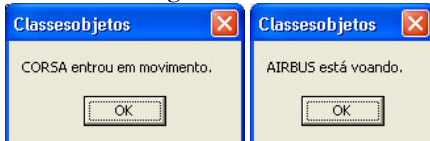


Figura. Chamando um método

Herança

Como você pode notar, muitas características estão presentes tanto em *TCarro* como em *TAviao*, como os campos *Descricao*, *Capacidade* e o método *Mover*. O que faremos agora é generalizar as classes *TCarro* e *TAviao*, criando uma classe base chamada *TMeioTransporte*, que conterá as características comuns a todos os meios de transporte. Dessa forma, *TAviao* e *TCarro* herdarão dessa classe e adicionarão funcionalidades específicas.

Clique em *File|New|Unit*. Salve a nova unit como “uMeioTransporte.pas”. Defina a nova classe como mostrado a seguir:

```
unit uMeioTransporte;

interface

type
  TMeioTransporte = class
    Descricao : string;
    Capacidade : integer;
    procedure Mover;
  end;
```

```
implementation
{ TMeioTransporte }

procedure TMeioTransporte.Mover;
begin

end;

end.
```

Abra a unit *uCarro.pas* e altere a definição da classe *TCarro* como mostrado a seguir:

```
unit uCarro;

interface

uses
  uMeioTransporte; // coloque essa unit p/ acessar a classe TMeioTransporte

type
  TCarro = class (TMeioTransporte) // observe que TCarro agora herda de TMeioTransporte
    Quilometragem : integer; // observe que retiramos os campos Capacidade e Descricao daqui
    procedure Mover;
  end;
  (...)
```

Os campos *Capacidade* e *Descricao* são agora herdados de *TMeioTransporte*. Mas e o método *Mover*, porque não foi removido? Porque nem todas as classes se movimentam da mesma forma, ou seja, cada descendente deve implementar da sua forma o método *Mover*. O que fizemos foi apenas declarar o método *Mover* na classe base para dizer que ele existe para todas as classes descendentes. Falaremos mais sobre isso a seguir, quando estudarmos o polimorfismo.

Repita os mesmos passos para a classe *TAviao*, fazendo a herança e removendo os campos *Capacidade* e *Descricao*. Execute a aplicação.

Métodos Estáticos

Vá até o formulário principal e localize a seguinte declaração, na seção *public* da classe:

```
public
  Carro : TCarro;
  Aviao : TAviao;
```

Substitua por:

```
public
  Carro,Aviao : TMeioTransporte;
```

Declare *uMeioTransporte* na cláusula *uses*. Se você compilar o programa agora, receberá alguns erros. Como a variável *Carro* agora é um *TMeioTransporte* ela não reconhece mais o campo *Quilometragem*. Faça então o seguinte *TypeCasting*:

```
TCarro(Carro).Quilometragem:=StrToIntDef(EdtQuilometragem.Text,0);
```

Faça o mesmo para a atribuição de *HorasVoo* de *TAviao*:

```
TAviao(Aviao).HorasVoo:=StrToIntDef(EdtHorasVoo.Text,0);
```

Falaremos mais sobre *TypeCasting* a seguir.

Execute a aplicação. Você verá que tudo funciona perfeitamente, menos o botão *Mover*. Quando você chama o método *Mover*, você está chamado o método da classe *TMeioTransporte*, que não possui implementação. Isso porque *Mover* é um método **estático** (o padrão), e dessa forma ele fica vinculado à classe que você declarou. Observe a declaração das variáveis *Aviao* e *Carro*:

```
public
  Carro,Aviao : TMeioTransporte;
```


Observe agora como foram criados *Carro* e *Aviao*:

```
Aviao:=TAviao.Create;
```

```
Carro:=TCarro.Create;
```

Como você viu é possível declarar uma variável de um tipo e instanciar a partir de outro tipo (desde que seja descendente). Isso é possível porque *TAviao* e *TCarro* são *TMeioDeTransporte*.

Quando você chama o método *Carro.Mover*, não importa se a instância atual faz referência a um *TCarro*. Como o objeto foi declarado com *TMeioTransporte*, o método *Mover* sempre será dessa classe (pois, novamente, ele é **estático**).

Métodos Virtuais e Dinâmicos - Polimorfismo

Abra a unit *uMeioTransporte* e altere a declaração do método *Mover* como mostrado a seguir:

```
procedure Mover; virtual;
```

Virtual define um método que pode ser sobrescrito e assim tomar múltiplas formas em classes descendentes.

Abra a unit *uCarro.pas* e altere o método *Mover* como mostrado a seguir:

```
procedure Mover; override;
```

Faça o mesmo para *TAviao.Mover*.

Override é a diretiva usada para **sobrescrever** um método virtual ou dinâmico.

Dessa forma, se você agora chamar o método *Mover* de um objeto instanciado como *TCarro* mas originalmente declarado como *TMeioTransporte*, fará uma chamada ao método *TCarro.Mover*, pois o método deixou de ser **estático** e passou a ser **virtual**.

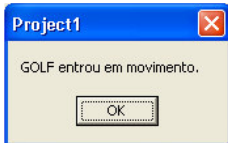


Figura. Chamando um método virtual

Inherited

Como você pode ver, o método *Mover* não possui implementação na classe base *TMeioTransporte*. Podemos implementar o comportamento padrão nessa classe e deixar que as classes descendentes chamem esse método, adicionando alguma funcionalidade mais específica.

Imagine que, por exemplo, que todo o veículo precisa ser ligado antes de entrar em movimento. Você não teria que implementar isso em todas as classes descendentes de *TMeioTransporte*, pois esse comportamento é padrão para todas.

Abra então a unit *uMeioTransporte.pas* e implemente o método *Mover* como mostrado a seguir:

```
procedure TMeioTransporte.Mover;
begin
  ShowMessage('Ligando ' +Descricao);
end;
```

Declare *Dialogs* na cláusula *uses*. Agora localize a classe *TCarro*, vá até a implementação do método *Mover* e adicione a diretiva **inherited** como mostrado a seguir:

```
procedure TCarro.Mover;
begin
  inherited;
```

```
ShowMessage(Descricao+' entrou em movimento');
end;
```

Inherited chama um método da classe base

Execute a aplicação. Adicione um carro à lista e clique no botão *Mover*. Como o método é *virtual* será disparado o método *Mover* de *TCarro*. A primeira linha da implementação desse método contém um *inherited*, o que faz uma chamada ao método *Mover* da classe base (*TMeioTransporte*), e logo em seguida é executado o restante do código. Por último, chame *inherited* também no método *Mover* de *TAviao*.

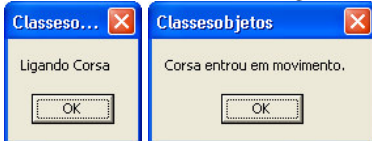


Figura. Inherited chama o método da classe base

Esse recurso é bastante poderoso, visto que podemos compartilhar uma mesma implementação herdada da classe base e adicionar alguma funcionalidade específica. Em uma linguagem não orientada a objetos, você provavelmente precisaria criar procedimentos diferentes e fazer vários testes condicionais com IF para saber que método deve ser chamado.

Métodos Abstratos

Mas agora imagine o seguinte: e se cada meio de transporte for ligado de uma forma? O que é comum nesse caso é a chamada do método, ou seja, todos devem ser ligados antes de entrarem em movimento. O que muda é a maneira como são ligados.

Então altere o código do método *Mover* mostrado anteriormente para:

```
procedure TMeioTransporte.Mover;
begin
  Ligar;
end;
```

Isso diz que um meio de transporte deve ser ligado sempre que entrar em movimento. Agora declare o seguinte na *interface* da classe *TMeioTransporte*:

```
procedure Ligar; virtual; abstract;
```

Abstract é sempre usado da diretiva **virtual** e indica que o método não possui implementação na classe em que é declarado.

Ligar não é implementado nessa classe, ou seja, ele é totalmente abstrato, só possui a definição para poder ser chamado. Usando recursos de polimorfismo vamos sobrescrever esse método na classe descendente.

Abra a classe *TCarro* e sobrescreva o método *Ligar*:

```
procedure Ligar; override;
```

E implemente-o da seguinte forma:

```
procedure TCarro.Ligar;
begin
  // repare que não vai inherited aqui pois não existe nada na classe base
  ShowMessage('Ligando o carro '+Descricao);
end;
```

Faça o mesmo para a classe *TAviao*.

Execute a aplicação e crie um carro. Veja o que acontece passo a passo:

1 - Clique no botão *Mover*. Será chamado *Mover* da classe *TMeioTransporte*. Como esse método é *virtual* será chamado *Mover* de *TCarro*.

2 – *Mover* de *TCarro* faz uma chamada a *inherited*, e invoca a classe base, *TMeioTransporte.Mover*.

3 – *TMeioTransporte.Mover* chama *Ligar* para ligar o meio de transporte.

SIG – Soluções em Informática e Gestão – Treinamento em Delphi – Módulo 3

Instrutor –guinther@sigbrasil.com.br – Copyright© SIG 2003 - Todos os direitos Reservados

4 – *Ligar* é um método *virtual*, logo será chamado *TCarro.Ligar*.

5 – Após a chamada à *inherited* retornamos a *TCarro.Mover*, que então executa o restante do código.

Mais polimorfismo

Faremos agora outro método *virtual*, semelhante ao método *Mover*. Declare o seguinte em *TMeioTransporte*:

```
function Consultar : string; virtual; abstract;
```

Sobrescreva esse método na classe *TCarro*. Declare a função na *interface* usando a diretiva *override*:

```
function Consultar: string; override;
```

Implemente o método como mostrado a seguir:

```
function TCarro.Consultar: string;
begin
    result:='Descrição='+Descricao+#13+ // atenção: use "=" na string ao lado, bem como está mostrado
    'Capacidade='+IntToStr(Capacidade)+#13+
    'Quilometragem='+IntToStr(Quilometragem);
end;
```

Implemente o mesmo método na classe *TAviao*, como mostrado a seguir:

```
function TAviao.Consultar: string;
begin
    result:='Descrição='+Descricao+#13+ // atenção: use "=" na string ao lado, bem como está mostrado
    'Capacidade='+IntToStr(Capacidade)+#13+
    'Horas de voo='+IntToStr(HorasVoo);
end;
```

Declare *SysUtils* na cláusula *uses* de ambas as units. Coloque dois botões, chamados “Consultar”, um para cada tipo de objeto. No evento *OnClick* digite:

```
Carrol.Consultar;
```

No outro botão, faça o mesmo para Avião. Teste a aplicação.

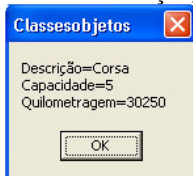


Figura. Chamando um método virtual

Atenção: Se você esquecer de declarar *override* ao sobrescrever um método virtual, receberá o seguinte warning do compilador: “Method ‘Consultar’ hides virtual method of base type ‘TMeioTransporte’”

Construtores e Destrutores

Um *construtor* é um método especial encarregado de alocar a memória para um objeto. Ao contrário, um *destrutor* libera a memória alocada para o objeto. Muitas vezes precisamos adicionar código personalizado a um construtor, por exemplo, para alocar memória para um objeto interno ou inicializar variáveis. Em nosso exemplo, vamos utilizar construtores para inicializar as variáveis do objeto. Declare o seguinte na classe *TCarro*:

```
constructor create;
```

Aperte Shift+Ctrl+C e implemente o construtor da seguinte forma:

```
constructor TCarro.create;
```

[g6] Comentário: Comentar o uso de construtores sobrecarregados

```
begin
  inherited; // chama o construtor da classe base
  Quilometragem:=0;
end;
```

Faça o mesmo para a classe *TAviao*:

```
constructor TAviao.create;
begin
  inherited;
  HorasVoo:=0;
end;
```

Podemos inicializar as variáveis comuns (como *Descricao* e *Capacidade*) no construtor da classe base:

```
constructor TMeioTransporte.create;
begin
  inherited;
  Capacidade:=0;
  Descricao:='Sem Nome';
end;
```

Para criar um destrutor, você deve declará-lo na interface da classe:

```
destructor destroy; override;

destructor TMeioTransporte.destroy;
begin
  // seu código de limpeza aqui
  inherited;
end;
```

Atenção: Observe que *destroy* é declarado como *virtual* em *TObject*, logo devemos dar um *override* ao sobrescrever. Ao contrário, o *constructor* é estático, e só passa a ser *virtual* a partir de *TComponent*.

Type Casting e RTTI

Podemos testar o tipo de um objeto em tempo de execução usando o operador RTTI **is**. Para fazer o *TypeCasting* do objeto para o tipo desejado usamos o operador **as**.

```
If Carro is TMeioTransporte then
  (Carro as TMeioTransporte).Mover;
```

Além do **as**, outra maneira de se fazer o *TypeCasting* é:

```
TMeioTransporte(Carro).Mover;
```

Você pode usar também o método *InheritsFrom* ao invés do **is** para saber se um objeto é ou descendente de um tipo:

```
If Carro.InheritsFrom(TMeioTransporte) then
```

Exercício: Criar um formulário com *Edits*. Quando um *Edit* receber o foco, o seu *Hint* deve ser mostrado em uma *StatusBar*. Usar para isso o recurso de RTTI.

Especificadores de Visibilidade

Até aqui nossas classes utilizaram um único especificador de visibilidade para as variáveis membro e métodos (**public**). Um especificador de visibilidade define como um campo ou método pode ser acessado (visto) por outras classes. Veja a seguir a declaração de uma classe com seus respectivos especificadores:

```
type
  TMeioTransporte = class(TObject)
  private
    ...
  protected
    ...
  public
    ...
```

```
published
end;
```

Private - Visível somente na classe atual;

Protected - Visível somente na classe atual e descendentes;

Public - Visível a partir de qualquer outra classe;

Published - Visível a partir de qualquer outra classe, ativando suporte a RTTI e Object Inspector;

Localize a declaração da classe *TMeioTransporte*, e altere como mostrado a seguir:

```
TMeioTransporte = class
private
  FCapacidade : integer;
  FDescricao : string;
protected
  procedure Ligar; virtual; abstract;
public
  function Consultar : string; virtual; abstract;
  constructor create;
  destructor destroy; override;
  procedure Mover; virtual;
end;
```

Observe que foi colocada a letra “F” no início dos campos privados, para indicar que se trata de um *Field*. Observe também que o método abstrato *Ligar* foi declarado na seção *protected*. Isso porque esse método não é chamado a partir de outra classe, é chamado internamente pela própria classe *TMeioTransporte* e é sobrescrito pelas classes descendentes.

O motivo de movermos as variáveis para a seção *private* foi encapsular o seu acesso. Porém, se tentarmos compilar a aplicação vamos ter alguns erros, pois variáveis privadas não são acessíveis fora do escopo da classe. Criaremos então propriedades para permitir o acesso às variáveis privadas.

Propriedades

Adicione então o último especificador de visibilidade à nossa classe, declarando duas propriedades que fazem o mapeamento para os campos privados. Utilize a diretiva *property* para definir uma propriedade:

```
(...)
published
  property Capacidade: integer read FCapacidade write FCapacidade;
  property Descricao: string read FDescricao write FDescricao;
end;
```

Altere a definição de *TCarro* e *TAviao*, criando propriedades e definindo os especificadores de visibilidade.

Default

Default é um especificador de armazenamento. Se o valor atual de propriedade de um componente corresponder ao mesmo valor indicado pela diretiva *default*, então o valor não será armazenado no arquivo DFM. *Default* não inicializa propriedades, para isso você deve usar o construtor.

Altere a definição da propriedade *Capacidade* como mostrado a seguir:

```
property Capacidade: integer read FCapacidade write FCapacidade default 0;
```

Atenção: *Default* só pode ser usado com tipos ordinais, como inteiros e booleanos.

Métodos Get/Set

Uma grande vantagem de se usar propriedades ao invés de campos, é a possibilidade de usar rotinas que fazem o mapeamento da atribuição e leitura das variáveis privadas.

Por exemplo, vá até a definição da classe *TCarro* e altere a definição da propriedade para:

```
property Quilometragem : integer read GetQuilometragem write SetQuilometragem;
```

Aperte Shift+Ctrl+C. Serão criados dois métodos na seção *private* (o *Get* e o *Set*):

```
function GetQuilometragem: integer;
procedure SetQuilometragem(const Value: integer);
```

Implemente os métodos da seguinte forma:

```
function TCarro.GetQuilometragem: integer;
begin
    result:=FQuilometragem;
end;

procedure TCarro.SetQuilometragem(const Value: integer);
begin
    if Value<0 then
        FQuilometragem:=0
    else
        FQuilometragem:=Value;
end;
```

Observe que nossa rotina *Get* apenas repassa o valor para a variável privada. Em nossa rotina *Set* testamos se o valor passado não foi negativo, e se for configuramos a variável privada como zero. Observe que o seguinte código:

```
Carro.Quilometragem:=-3;
```

Fará ser disparar a rotina *Set*, e o valor da *Quilometragem* passará a valer 0.

Veremos mais sobre propriedades, rotinas get/set e especificadores de visibilidade a seguir, quando criarmos um componente.

Clique em *File|Save All*.

Overload

[g7] Comentário: Falar que está fora do exemplo

Uma última diretiva que você pode usar ao declarar métodos ou rotinas é o *Overload*. Ele permite que um método (ou rotina) tenha diferentes versões, variando os parâmetros em tipo e quantidade. Ex.

```
procedure ShowText (const Number : Integer); overload;
begin
    ShowMessage(IntToStr(Number));
end;

procedure ShowText (const Text : string); overload;
begin
    ShowMessage(Text);
end;

procedure ShowText (const Text : string; const Number : integer); overload;
begin
    ShowMessage(Text+' - '+IntToStr(Number));
end;
```

Você pode usar o procedimento da seguinte forma:

```
ShowText(10);
ShowText('Olá Mundo');
ShowText('Olá Mundo',20);
```

4. Componentes e Pacotes

Um pacote é, basicamente, um conjunto de units que quase sempre definem componentes. Um pacote permite, entre outras coisas, que uma aplicação fique “modularizada”. Pacotes podem diminuir o tamanho do

seu executável final e auxiliar na distribuição de atualizações. Um pacote é um arquivo .BPL (um tipo especial de DLL), e pode ser de 3 tipos: *RunTime*, *Design-Time* e *Run-Time & DesignTime*.

Criando um Pacote

Clique em *File|New|Other>Package*.

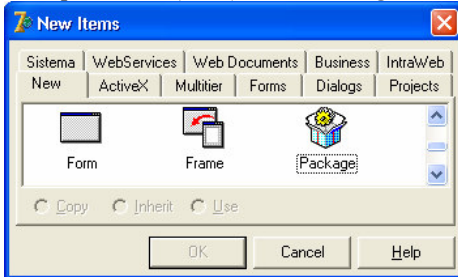


Figura. Criando um pacote

Salve o pacote em um novo diretório ou no mesmo diretório da aplicação anterior. Chame o pacote de “pkPessoa.dpk”. Veja na figura a seguir o editor do pacote.

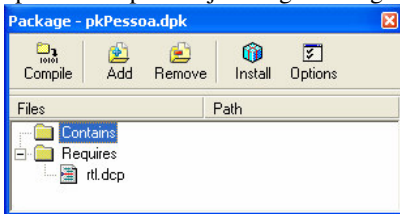


Figura. Editor do Pacote

Criando um Componente

Clique no botão *Add*. Na janela que aparece clique em *New Component*. Preencha os campos como mostrado na figura a seguir:

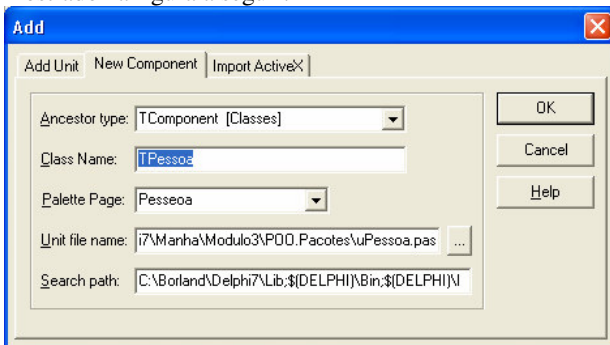


Figura. Criando um componente

Atenção: Em *Unit file name* especifique o nome do arquivo e o caminho completo onde deve ser salvo.

Aperte o botão *Ok*. Será criada uma nova unit com a estrutura básica de uma classe derivada de *TComponent*.

Criando uma propriedade

Na seção *published* declare o seguinte:

```
published
property Nome: string;
```

Aperte *Shift+Ctrl+C* (ou dê um clique de direita no editor e escolha *Complete class at cursor*). O Delphi criará automaticamente o campo privado para a propriedade e a rotina *Set*.

Atenção: Troque o ancestral da classe *TPessoa* para *TComponent*.

Instalando o pacote

De volta ao editor do pacote clique no botão *Install*. O arquivo *dpr* será compilado juntamente com as units do pacote, gerando um arquivo *.bpl* que é adicionado à IDE do Delphi. Você deve receber uma mensagem semelhante a mostrada a seguir. Observe que foi criada uma nova paleta no Delphi com o componente criado.

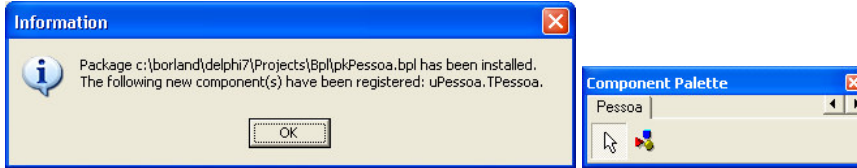


Figura. Instalando o pacote

Criando um ícone para o componente

Abra o *Image Editor* no menu *Iniciar\Programas\Borland Delphi 7* (ou use o menu *Tools* da IDE). Clique em *File\New\Component Resource File (.dcr)*. Salve o arquivo no mesmo diretório do pacote, como nome *pkIcones.dcr*. Clique de direita em *Contents* e escolha *New>Bitmap*. Defina as dimensões 24x24 e 256 cores. Renomeie *Bitmap1* para o mesmo nome da classe, tudo em minúsculo, neste caso *toperador*. Dê um duplo clique nesse item e desenhe o ícone no editor que aparece.

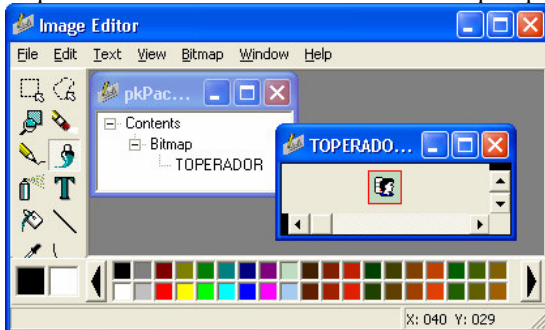


Figura. Criando um ícone para o componente

Salve o arquivo. Volte ao pacote no Delphi e clique em *Project\View Source*. Logo após a declaração do pacote inclua o arquivo *dcr*.

```
package pkPacote;
{$R pkIcones.dcr} // inclusão do arquivo dcr
```

Salve e recompila o pacote.

Testando

Inicie uma nova aplicação e coloque no formulário nosso componente. Observe que a propriedade *Nome* (que é *Published*) está disponível no *Object Inspector*. Se você olhar o arquivo *.dfm*, verá que como nossa classe é um *TComponent*, derivado de *TPersistent*, tem suporte à persistência.

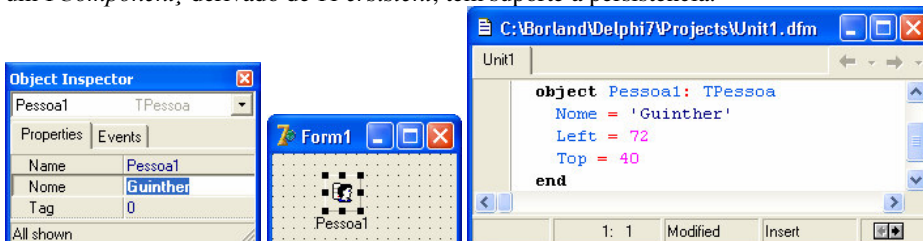


Figura. Usando o componente

Library Path

O Library é uma importante opção do Delphi que indica onde devem ser encontradas as bibliotecas compiladas utilizadas por uma aplicação (arquivos .dcu por exemplo). Para visualizar essa opção clique em *Tools/Environment Options>Library>Library Path*.

[g8] Comentário:

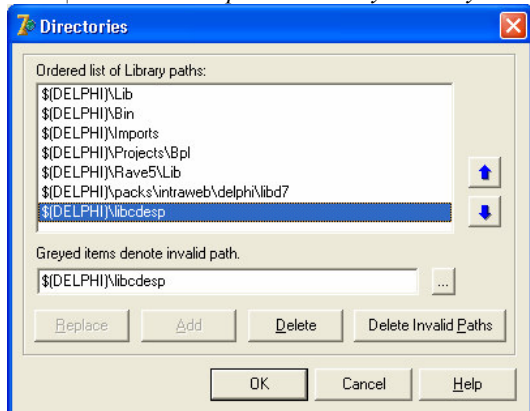


Figura. Library Path

Atenção: Se o caminho não estiver no *Library Path*, o Delphi dirá que não encontrou a unit do componente na cláusula *uses* quando você tentar compilar uma aplicação que utilize o componente.

De Classes a Componentes

O que faremos agora é transformar em componentes nossas três classes criadas no exemplo anterior (*TMeioDeTransporte*, *TCarro* e *TAviao*). Primeiro, no editor de pacotes, com o item *Contains* selecionado, clique no botão *Add*. Adicione as units *uMeioDeTransporte*, *uCarro* e *uAviao*. Falaremos mais sobre a opção *Contains* a seguir.

Abra a unit *uMeioTransporte* e defina *TComponent* como classe ancestral de *TMeioTransporte*. Adicione *Classes* à cláusula *uses* (veja o código a seguir):

```
uses
  classes;
type
  TMeioTransporte = class (TComponent)
```

Altere a definição do construtor como mostrado a seguir:

```
constructor create (AOwner : TComponent); override;
```

Altere essa definição também a implementação do construtor.

Faça a mesma alteração no construtor das classes *TCarro* e *TAviao*. Declare *Classes* na cláusula *uses* dessas units também.

Registrando componentes

Vá até a unit *uCarro* e declara o seguinte procedimento antes da cláusula *implementation*:

```
procedure Register;
```

Atenção: A linguagem Delphi não diferencia maiúsculas e minúsculas. A única exceção é o procedimento *Register*, que deve ter "R" maiúsculo.

Implemente esse procedimento na seção *implementation* da unit, logo após a cláusula *uses*.

```
procedure Register;
begin
  RegisterComponents('Pessoa', [TCarro]);
end;
```

Faça o mesmo para unit *uAviao*. Recompile o pacote e instale os novos componentes.

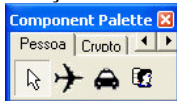


Figura. Pacote instalado

Observe que não precisamos fazer esse procedimento para o componente *TPessoa*, pois esse componente foi criado usando o wizard do pacote, o qual já inclui a cláusula *register* automaticamente.

Observe ainda que *TMeioTransporte* não foi registrado, pois esse componente não é usado diretamente na IDE, apenas serve de ancestral para *TCarro* e *TAviao*.

Polimorfismo e Abstração em Componentes

Agora imagine que um objeto *Pessoa* precise fazer uso de um *Carro* e de um *Aviao*. Poderíamos simplesmente declarar duas propriedades na classe *TPessoa*:

Atenção: não altere seu código, apenas observe o código a seguir:

```
property Carro : TCarro read FCarro write FCarro;
property Aviao : TAviao read FAviao write FAviao;
```

Pronto, uma pessoa poderá agora utilizar carros e aviões. Mas temos um problema. Se um dia for criado um novo meio de transporte, como *TBicicleta*, nosso componente *TPessoa* precisaria de uma nova propriedade. E se novos meios de transporte forem surgindo, serão necessárias novas alterações em *TPessoa*.

A solução é criar uma única propriedade que possa apontar para qualquer meio de transporte. Declare a seguinte propriedade na sessão *published* de *TPessoa*:

```
property Transporte : TMeioTransporte read FTransporte write FTransporte;
```

Aperte Shift+Ctrl+C para gerar a declaração do campo privado. Declare *uMeioTransporte* na cláusula *uses*. Recompile o pacote.

[g9] Comentário: Falar que não se deve sempre levar uma TV pro concerto quando estraga o vídeo cassete, nem que precisamos trocar o estofamento do carro quando se troca a bateria por uma mais potente

Nota: Você deve estar se perguntando por que declarar uma variável de um tipo ancestral e não do tipo descendente, como fizemos aqui. Isso permite que sua aplicação utilize os serviços de uma determinada classe sem saber até o momento de execução qual implementação está sendo utilizada. Dessa forma, podemos fornecer uma nova implementação de nossa classe sem que para isso tenha que se alterar outras classes que a referenciam. Um belo exemplo disso é a propriedade *DataSet* de um relatório do *QuickReport*. Quando o QR foi criado, ele definiu a propriedade *DataSet* do tipo *TDataSet*, de onde os dados para o relatório são retirados. A partir daí isso nunca mais foi mudado. E surgiram várias implementações de *TDataSet* através dos anos, como *TQuery*, *TSQLQuery*, *TIBTable*, *TSimpleDataSet*, e a classe *TQuickRep* pode gerar relatórios para qualquer um desses *TDataSets*.

Inicie uma nova aplicação e salve os arquivos do projeto. Coloque no formulário principal um *ValueListEditor* (da paleta *Additional*) e dois *Buttons* com os *Captions* de “Mover” e “Consultar”. Coloque um *Carro* e uma *Pessoa*. Defina as propriedades do objeto *Carro*. Aponte a propriedade *Transporte* da *Pessoa* para o objeto *Carro*.

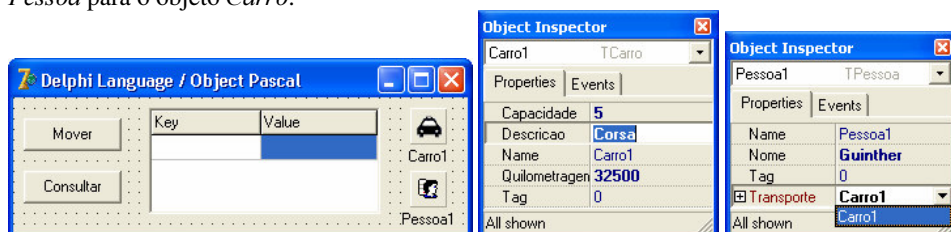


Figura. Classes transformadas em Componentes

No evento *OnClick* do botão *Mover* escreva:

```
Pessoa1.Transporte.Mover;
```

Execute a aplicação. Pronto, agora nossa pessoa já consegue colocar o carro em movimento.
No evento *OnClick* do botão *Consultar* escreva:

```
ValueListEditor1.Strings.Text:=Pessoa1.Transporte.Consultar;
```

Execute a aplicação e teste a chamada polimórfica ao método *Consultar*.

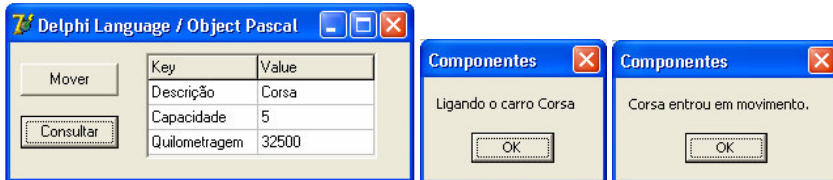


Figura. Chamando métodos do componente

Agora coloque um componente *Avião* no formulário, e configure suas propriedades. Aponte a propriedade *Transporte* de *Pessoa* para o objeto *Aviao*.

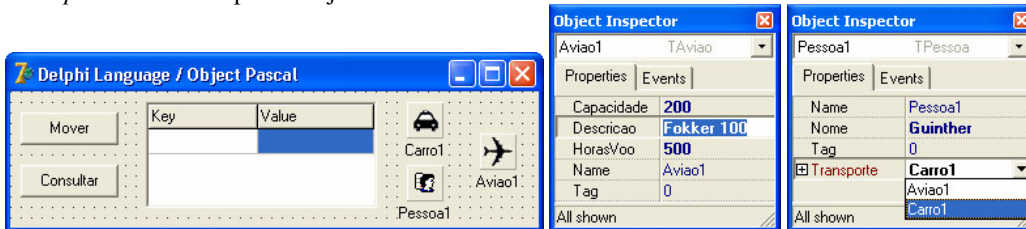


Figura. Substituindo o componente de transporte

Notification

Se você excluir o componente *Aviao* do formulário receberá uma mensagem de exceção e provavelmente tenha que fechar a IDE do Delphi. Isso porque o componente *Pessoa* estará apontando para uma posição de memória que não existe mais. Para resolver esse problema usamos o método *Notification*.

Declare o seguinte na seção *protected* da classe *TPessoa*:

```
procedure Notification(AComponent: TComponent; Operation: TOperation); override;
```

Aperte Shift+Ctrl+C e implemente-o da seguinte forma:

```
procedure TPessoa.Notification(AComponent: TComponent; Operation: TOperation);
begin
  inherited;
  if (Operation=opRemove) and
    (AComponent=Transporte) then
    Transporte:=nil;
end;
```

Eventos

Declare o seguinte na seção *published* da classe *TMeioTransporte*:

```
property BeforeLigar : TNotifyEvent;
```

Aperte Shift+Ctrl+C. Altere o método *Mover* da classe para:

```
procedure TMeioTransporte.Mover;
begin
  if Assigned(FBeforeLigar) then
    BeforeLigar(self);
  Ligar;
end;
```

Recompile o pacote. Volte ao exemplo anterior. Selecione o componente *Carro* e veja o novo evento no *Object Inspector*:

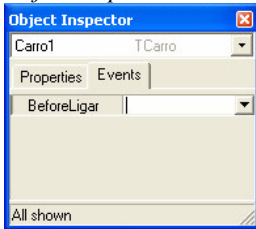


Figura. Criando um evento

Adicione o seguinte manipulador para o evento *BeforeLigar*:

```
procedure TFrmExemplo.Carro1BeforeLigar(Sender: TObject);
begin
  raise EAbort.create('Isso cancelará a chamada aos métodos Ligar e Mover');
end;
```

Selecione o componente *Aviao* e aponte seu evento *BeforeLigar* para o mesmo método. Dessa forma, ambos os componentes usam o mesmo manipulador. O parâmetro *Sender* dirá qual dos componentes está chamando o evento.

Observe a exceção silenciosa que foi levantada caso o usuário deseje cancelar a chamada. O *raise* fará com que a sequência de chamadas (*Call Stack*) seja quebrada e todos os métodos cancelados.

Opções do Pacote

Abra novamente o pacote *pkPessoa*. Clicando no botão *Options* temos acesso às opções do pacote, onde podemos definir a descrição, os diretórios onde devem ser salvos os arquivos .dcu, .dcp., .bpl., se o pacote é de Run-Time ou Design-Time, etc.

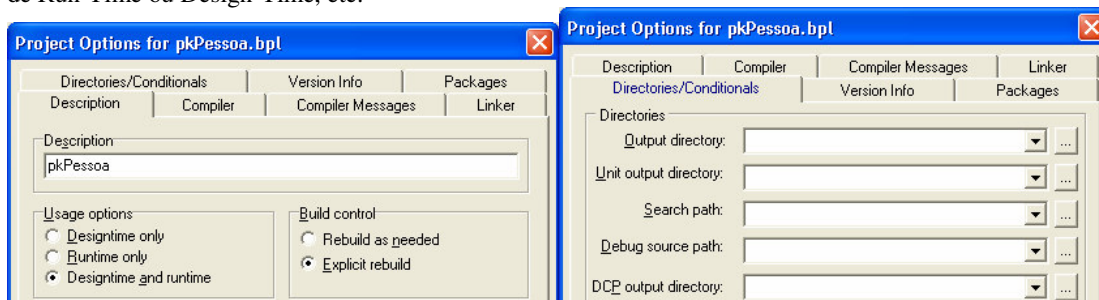


Figura. Opções de Pacote

Contains e Requires

Contains indica todas as *units* que estão contidas no pacote.

Para demonstrar o uso de das seções *Requires* e *Contains*, vamos separar nosso pacote em dois. Um que conterá *TPessoa* e o outro conterá os *TMeioTransporte*.

Nosso pacote *pkPessoa* possui 4 units, indicadas no item *Contains*. Remova então as units *uAviao*, *uCarro* e *uMeioTransporte*. Para isso dê um clique de direita na unit e escolha *Remove from Project*. Depois desinstale esse pacote da IDE, clicando em *Component\Install Packages*, selecionando o pacote *pkPessoa* e clicando em *Remove*.

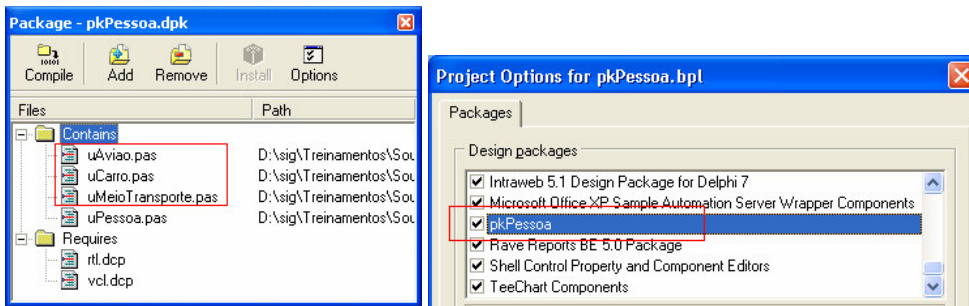


Figura. Removendo componentes do pacote

Clique em *File|New|Other>Package* e salve o pacote como “pkTransporte.dpk” no mesmo diretório do pacote *pkPessoa*. Clique no botão *Add* e adicione as 3 units removidas do pacote anterior. Clique em *Options>Description* e dê a descrição de “pkTransporte” ao pacote. Adicione o arquivo *pkIcones.dcr* ao arquivo .dpk, como fizemos anteriormente para *pkPessoas.dpk*.

Abra as units *uAviao* e *uCarro* e altere o nome do pacote no procedure *Register* para “Transporte”. Clique em *Install*. Pronto, temos um pacote só para os meios de transporte.

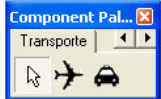


Figura. Classes transformadas em Componentes

Atenção: Se não for criada a nova paleta, desinstale e instale novamente o pacote ou simplesmente reorganize a paleta dando um clique de direita sobre ela e escolhendo *Properties*.

Abra o novamente o pacote *pkPessoa* e clique em *Install*. Você receberá a seguinte mensagem:

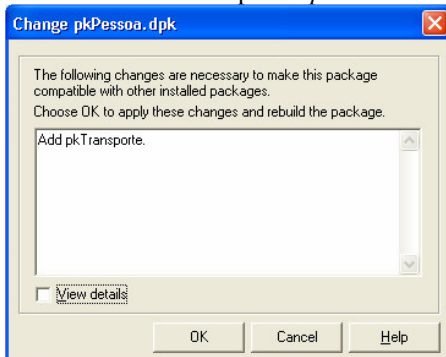


Figura. Dependência entre pacotes (requires)

Observe que nossa unit *pkPessoa* faz uma referência a unit *uMeioTransporte* na unit *uPessoa*:

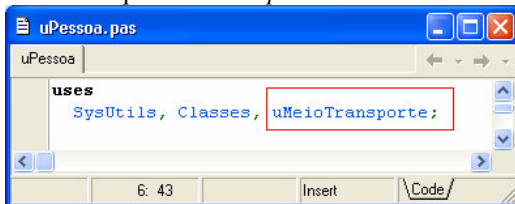


Figura. Unit de um pacote adicionada à cláusula uses de uma unit do outro pacote, causando a dependência entre pacotes

A IDE do Delphi percorre todos os pacotes instalados na IDE procurando se alguma BPL já possui essa unit instalada (nesse caso *uMeioTransporte*). Caso encontre, o Delphi ele adiciona esse pacote à cláusula *Requires* do pacote:

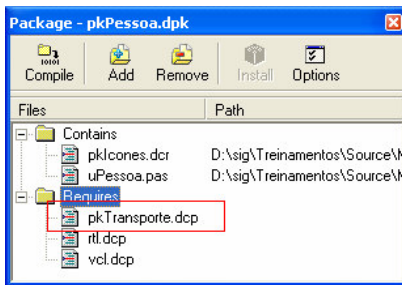


Figura. Requires indica a dependência entre pacotes

Atenção: *rtl.dcp* e *vcl.dcp* são pacotes definidos pelo Delphi e definem as classes básicas utilizadas pelos componentes do pacote.

Caso não encontre nenhuma BPL com essa unit, ou se o pacote *pkTransporte* não estiver na memória, o pacote *pkPessoa* fará uma importação implícita de *uMeioTransporte*, como se você tivesse colocado na seção *contains*.

Veja a simulação a seguir:

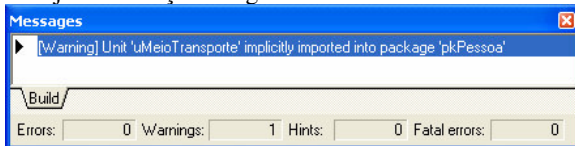


Figura. Classes transformadas em Componentes

Ao tentar carregar o pacote *pkTransporte* na IDE, você receberá um erro dizendo que *uMeioTransporte* já foi instalada no pacote *pkPessoa*.

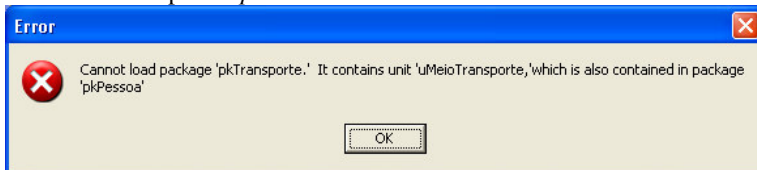


Figura. Classes transformadas em Componentes

Nesse caso desinstalar o pacote *pkPessoa* e instalar *pkTransporte* novamente. Portanto, tenha muito cuidado ao receber a mensagem de importação implícita.

Requires lista todos os pacotes requeridos pelo pacote corrente.

Isso é, para que seu pacote possa ser instalado na IDE os pacotes especificados no *Requires* devem estar instalados ou em um caminho que a IDE possa encontrar. Para refazer a lista *Requires*, exclua todas as entradas e compile o pacote novamente. Para adicionar novos pacotes clique de direita sobre o item *Requires* e escolha *Add*.

Usando pacotes em tempo de execução

Abra novamente nosso projeto exemplo feito anteriormente. Clique em *Project|Options|Packages*. Marque a opção *Build with runtime packages*. Especifique na lista quais pacotes você deseja utilizar em RunTime.

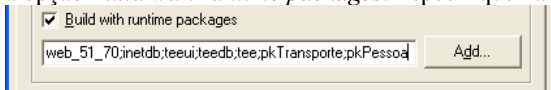


Figura. A opção Build with runtime packages

Em nosso exemplo, observe que agora estamos usando os pacotes *pkPessoa* e *pkTransporte* (entre outros). Dessa forma, ao compilar novamente o projeto, as units contidas nesses pacotes não serão compiladas junto com seu executável, mesmo que você esteja as referenciando na cláusula *uses* (veja a figura a seguir).

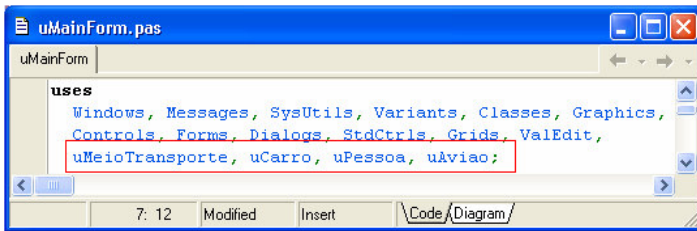


Figura. Units ficarão compiladas na BPL e não mais na aplicação

Com isso, o tamanho do seu executável ficará consideravelmente menor (porém você deve distribuir as BPL's). E se por algum motivo você precise alterar a implementação de alguma dessas units, você só precisará redistribuir o pacote. Lembrando que você pode utilizar essa técnica para distribuir pacotes do Delphi, como *vcl, rtl, dbexpress*, etc.

Pacotes de Design-Time e de RunTime

Pacotes podem ser de três tipos:

- *Runtime* (usado somente em tempo de execução)
- *Design-Time* (usado somente pela IDE do Delphi)
- *Runtime and Design-Time* (usado pela IDE do Delphi e em tempo de execução)

Você pode definir o tipo de pacote clicando em *Options>Description>Usage Options*:

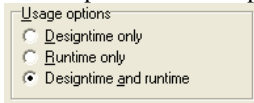


Figura. Tipos de pacote

Ao criar pacotes do tipo *Runtime* ou *Runtime and Design-time* você pode utilizar a opção mostrada no item anteriormente. Você usará pacotes de *Design-time* quando criar um editor de propriedade ou editor de componente, onde o código nunca vai ser utilizado pela aplicação final, apenas pela IDE do Delphi.

Herança Visual de Formulário (Visual Form Inheritance)

Um último tópico sobre orientação a objetos que não poderíamos deixar de falar aqui é o recurso de *Herança Visual de Formulário*. Para utilizar esse recurso, crie uma nova aplicação, salvando a unit do formulário principal como "uFrmMain.pas" (dê o nome de "FrmMain" ao formulário). Salve o projeto como "VFH.dpr". Crie um novo *DataModule* (dê o nome de DM e salve na unit uDM) e coloque nele dois *ClientDataSet*s, um chamado "cds_alunos" e outro chamado "cds_cursos". Crie um campo chamado NOME_ALUNO em um deles e no outro crie um campo chamado NOME_CURSO. Atribua a propriedade *FileName* desses componentes para "alunos.cds" e "cursos.cds" (semelhante ao que fizemos no módulo 1).



Figura. DataModule

Crie um novo formulário, salvando sua unit como "uFrmBase.pas" e dando-lhe o nome de "FrmBase". Coloque nesse formulário um *DataSource*, um *Panel*, um *DBNavigator* e três botões, configurando como mostrado na figura a seguir (aponte o *DataSource* do *DBNavigator* para *DataSource1*).



Figura. Formulário servirá como base para demais formulários

No evento *OnClick* do botão *Gravar* escreva:

```
DataSource1.DataSet.Post;
```

No evento *OnClick* do botão *Cancelar* escreva:

```
DataSource1.DataSet.Cancel;
```

No evento *OnClick* do botão *Inserir* escreva:

```
DataSource1.DataSet.Append;
```

Clique em *File|New|Other|(NomeDoProjeto)* e dê um duplo clique no formulário *FrmBase*.

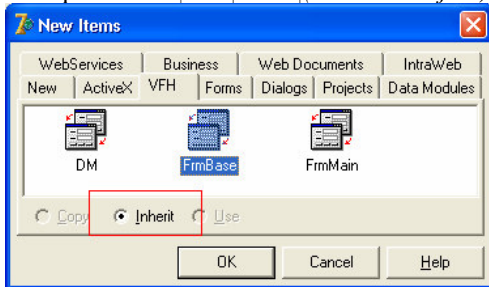


Figura. Object Repository permite criar um formulário herdado a partir de outro

Inherit significa que estamos criando um novo formulário herdado de *TFrmBase*

Após esse passo, será criado um novo formulário herdado de *TFrmBase*. Dê o nome de *TFrmAlunos* para este formulário. Observe a seguinte declaração:

```
TFrmAlunos = class(TFrmBase)
```

Isso indica que *TFrmAlunos* herda de *TFrmBase*. Aperte *Alt+F11* e escolha *uDM*. Aponte o *DataSource* para *cds_alunos*. Coloque um *DBEdit* e aponte para o campo *NOME_ALUNO*. Repita os mesmos passos e crie um formulário chamado *FrmCursos*. Veja na figura a seguir como ficaram os formulários:

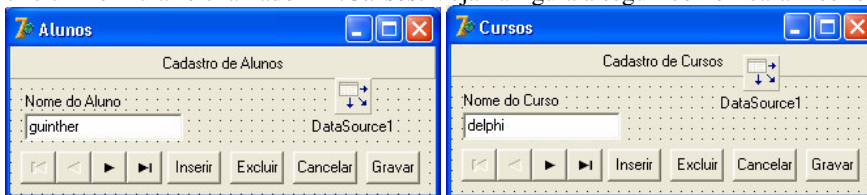


Figura. Classes transformadas em Componentes

Volte ao formulário principal e coloque dois botões, um para chamar o formulário de alunos e outro o de cursos. Execute e teste a aplicação.

Remova o formulário *TFrmBase* da lista de criação de formulários, ele não precisa ser instanciado pois apenas serve de base para os demais formulários

Agora o formulário base e escreva o seguinte no botão *Gravar*:

```
Validar;  
DataSource1.DataSet.Post;
```

Declare o método *Validar* na seção *protected* do formulário (essa seção não existe por padrão, você deve declará-la):

```
protected  
procedure Validar; virtual; abstract;
```

Esse método é chamado por *TFrmBase* antes da gravação dos dados, porém não possui implementação. Isso é feito nas classes descendentes usando *override*. Abra o formulário de alunos e declare:

```
protected
```



```
procedure Validar; override;
```

Implemente da seguinte forma:

```
procedure TFrmAlunos.Validar;
begin
  if DataSource1.DataSet.FieldByName('NOME_ALUNO').IsNull then
    raise Exception.Create('Nome do aluno não pode ficar em branco!');
end;
```

Criando componentes em tempo de execução (run-time)

Use a sintaxe a seguir para criar componentes em tempo de execução:

```
ClientDataSet:=TClientDataSet.Create(self);
```

ModelMaker básico

O ModelMaker é um utilitário que acompanha o Delphi 7 Enterprise e Architect, utilizado na modelagem de aplicações e frameworks, baseado em UML. Você pode utilizar o MM para criar suas classes, definir métodos, propriedades, criar diagramas, relações, aplicar *patterns*, etc.

O ModelMaker é discutido em detalhes no módulo 6

Após instalado o ModelMaker, o Delphi possui um item de menu (veja a figura a seguir).

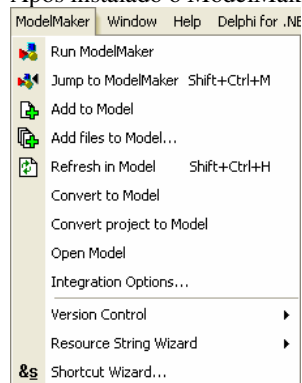


Figura. Menu do ModelMaker integrado à IDE do Delphi

Abra no editor as units *uPessoa*, *uMeioTransporte*, *uCarro* e *uAviao*. Clique no menu *Run|Model Maker*, e depois que o MM estiver aberto, volte ao Delphi e clique no menu *Add Files to Model*. Na janela que aparece clique no botão *Add All*. Clique em OK e volte ao MM (feche a janela *Messages*). Clique na aba *Diagram* e clique no botão *Add Diagram*. Dê o nome de *MeiosTransporte* para o diagrama.

Clique na aba *Classes* e arraste todas as classes para o interior do diagrama, partindo de *TMeioTransporte*.

Para exibir as classes com as propriedades e métodos, clique de direita no diagrama, escolha *Diagram Properties>SymbolStyle*, desmarque *Project member type filter*, marque todas as opções abaixo e em *Membre Visibility filter* escolha *private*, *protected*, *public* e *published*.

[g10] Comentário: Isso que fizemos é também chamado de *Reverse Engineer*.

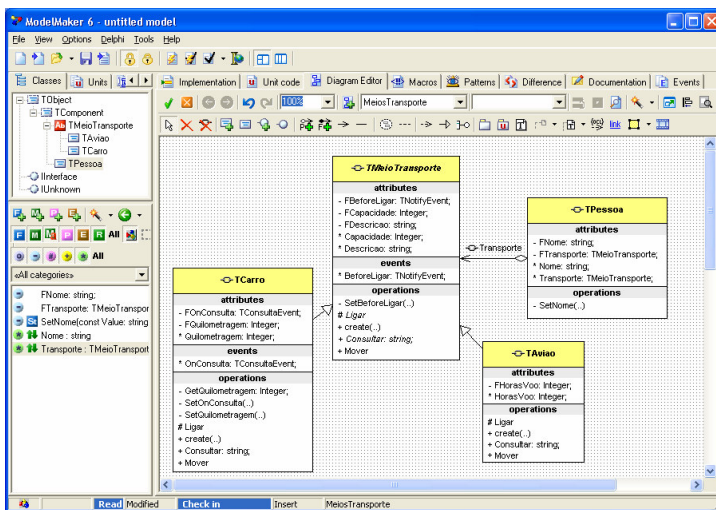


Figura. Diagrama de classes no ModelMaker

Selecione a classe *TMeioTransporte* e clique no botão *Add Method* . Preencha as opções conforme a figura a seguir:

TMeioTransporte method

Method | Documentation

Name:

Parameters:

Visibility: ☐ default ☐ public ☒ private ☐ published ☐ automated

Data type: ☒ Integer ☐ LongWord ☒ string ☐ Class ☐ Boolean ☐ AnsiString ☐ Variant ☐ Byte ☐ Extended ☐ Char ☐ OLEVariant ☐ Int64 ☐ Double ☐ PChar ☐ User defined ☐ Word ☐ Currency ☐ Pointer ☐ Void

Standard / More

Data type name:

Method type: ☐ constructor ☐ procedure ☒ function ☐ destructor

Method binding kind: ☐ Static ☐ Dynamic ☐ Message ☒ Virtual ☐ Override ☒ Abstract

Options and Directives: ☐ Class Method ☐ Reintroduce ☐ DISPID
☐ Call inherited ☐ Overload ☐ Hidden ☐ Instrumented
☐ Inheritance restricted ☐ Not inheritance restricted

Category:

Owner: the User

Figura. Adicionando um método

Clique em Ok. Clique em *Unlock code generation* e depois em *Enable Auto Generation* . Observe que o código foi atualizado no Delphi. Você pode usar o mesmo procedimento para criar propriedades.

```

procedure Ligar; virtual; abstract;
function Teste(str : string): string; virtual; abstract;
public
constructor create(AOwner : TComponent); override;
function Consultar: string; virtual; abstract;
  
```

Figura. Método gerado no código Delphi

A VCL

As classes básicas da VCL são mostradas no diagrama a seguir

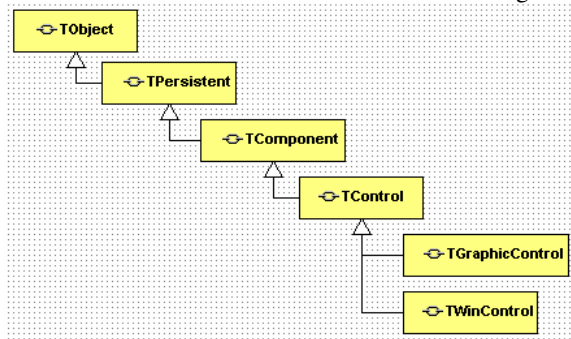


Figura. Classes base da VCL

5. Threads

Texto baseado no artigo de Guinther P – Revista ClubeDelphi nº 21

MultiThread - Programação concorrente

A programação multitarefa é um dos aspectos-chave na arquitetura interna da maioria dos sistemas operacionais. Mas o que é multitarefa? Sabemos que o sistema operacional Windows utiliza o conceito de multitarefa quando gerencia múltiplos aplicativos ao mesmo tempo. Podemos por exemplo digitar um documento no Word ao mesmo tempo em que estamos escutando música em nosso player, navegando pela Internet, fazendo download de arquivos ou lendo e-mails. O Windows percorre todos estes aplicativos dando um determinado tempo de execução para cada um, de acordo com sua prioridade, simulando assim a multitarefa. A execução simultânea real de dois ou mais aplicativos na verdade só será possível se o computador possuir mais de um processador.

Threads e Processos

Quando um desses aplicativos está na memória, utilizando recursos do processador, ele também é conhecido como processo. O Windows realiza multiprocessamento quando gerencia vários processos ao mesmo tempo. Um processo na verdade não executa código, ele contém um objeto interno que é responsável pela execução dos códigos. Este objeto interno é chamado de *Thread*. Cada processo possui pelo menos um *Thread*, chamado de *Thread primário* ou *principal*. Um aplicativo pode fazer multiprocessamento através da manipulação de vários *Threads*. Então uma aplicação *MultiThread* é aquela capaz de realizar multiprocessamento dentro de si própria.

Os Processos e *Threads* no Windows recebem um identificador quando executados (chamados respectivamente de *ThreadID* – identificador de *Thread*, e *ProcessID* – identificador de processo (PID)). *Threads* e Processos possuem prioridades de execução, como por exemplo: baixa, normal ou alta, que são utilizadas pelo Windows quando ele precisa fornecer fatias de tempo (chamadas *quantums*) para os *Threads* de cada processo que está na memória.

Processos

Seguindo os passos abaixo veremos seu aplicativo como um processo no Windows.

No Delphi, comece uma nova aplicação (*File|New|Application*). Dê o nome de *FrmMain* ao formulário e *Caption* de “MultiThread Demo”. Salve a unit com o nome de *uFrmMain*, e o projeto como Processo. Execute a aplicação pressionando F9. Abra o gerenciador de tarefas (*task manager*) do Windows pressionando Ctrl+Shift+Esc (Ctrl+Alt+Del no Windows 95). No caso do NT, você pode verificar que sua aplicação é um processo, faz uso de memória e CPU, além de possuir uma prioridade que pode ser setada em tempo de execução pelo próprio Windows (clique de direita sobre o processo). No windows 9x é possível somente visualizar o nome do processo.

Criando um contador

Neste exemplo criaremos um procedure simples, que executará um determinado bloco de código. Nosso objetivo agora será escrever uma tarefa, algo que utilize os recursos da máquina, utilize memória, processamento, faça o programa ficar ocupado. Poderíamos escrever código para acessar um determinado banco de dados, trazer dados de uma tabela, conectar ao modem, tocar alguma música. No entanto, criaremos uma tarefa bem simples, contar até 10. Não utilizaremos ainda o recurso de MultiThread. Vamos primeiro criar o problema, depois resolvê-lo.

Coloque um *TLabel* e um *TButton* no *Form*, e no evento *OnClick* do botão escreva o seguinte trecho de código:

```
procedure TForm1.Button1Click(Sender : TObject);
var
  cont : byte;
begin
  for cont:=1 to 10 do
  begin
    sleep(1000); // espera 1 segundo
    label1.caption:=inttostr(cont); // isso não funciona
  end;
end;
```

Execute a aplicação pressionando F9. Aperte o botão colocado no formulário.

O Thread principal

Quando clicamos no botão, o procedure acima é executado, em resposta ao evento *OnClick* do botão. O código foi executado dentro do *Thread principal* da aplicação (todo código que você escreve fica no *Thread principal* por default, se você não criar um *Thread* específico).

Você pode não saber, mas seu aplicativo já possui um *Thread* (o *Thread principal*). O fato de possuir um *Thread* não significa que não possamos criar mais *Threads* quando desenvolvemos uma aplicação. O *Thread principal* da sua aplicação Delphi é responsável pelo gerenciamento da UI (user interface – interface de usuário), onde estão os controles visuais (*formulários, botões, memos, edits*). A VCL pressupõe que toda a alteração do visual dos controles seja feita por meio deste *Thread principal*.

No código do exemplo acima, a função *sleep* (declarada na unit *SysUtils*) tem a função de parar o programa por um determinado tempo, passado em milésimos de segundo. Observe a linha que atualiza o *Caption* da *Label*. Lendo o código, diríamos que a cada segundo a *Label* receberia o valor de *cont*. Então ela iniciaria mostrando 1, e seria atualizada a cada segundo até chegar a 10. No entanto, não é isso que acontece. Quando você aperta o botão, o trecho de código é executado dentro do contexto do *Thread principal*. Como o *Thread principal* também é encarregado de trabalhar com a VCL, ele deixa de lado o tratamento de mensagens do formulário para processar a execução do procedure. Então somente ao final do código, o *Thread* retornará ao tratamento das mensagens da VCL, finalmente então colocando o valor 10 na *Label*. Note também que, durante a execução do procedure, o tratamento de eventos do *Form*, como mover ou redimensionar, não funcionarão. O programa simplesmente “trava”. Uma saída seria colocar a seguinte declaração logo abaixo da atribuição do *Caption* da *Label*:

```
Application.ProcessMessages;
```

Isso faz com que a aplicação processe as mensagens que estão esperando na fila. Neste caso, há uma mensagem *WM_SETTEXT* esperando para ser executada. *ProcessMessages* força a aplicação a tratar as mensagens da VCL, mesmo que a execução do procedure não esteja concluída.

Agora vamos visualizar o *Thread principal* usando o *Debugger* do Delphi:

Volte ao Delphi (mas mantenha seu aplicativo rodando), pressione *Ctrl+Alt+T*, ou vá até o menu *View|Debug Windows|Threads*.

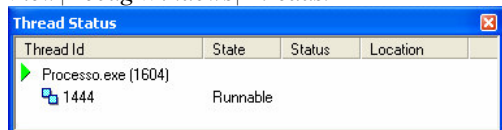


Figura. Classes transformadas em Componentes

Note que a aplicação possui um ID (é o *ProcessID*), e dentro dele há apenas um *Thread*, o principal, que também possui um ID (*ThreadID*).

Criando Threads

O que faremos agora é retirar o código de contagem, que está dentro do *Thread* principal da aplicação, colocando-o dentro de uma nova classe, derivada da classe *TThread* do Delphi. Criaremos então n instâncias de contadores de uma só vez, rodando simultaneamente, usando recursos da programação *MultiThread*. Faremos então um simulador de *Threads* (criando objetos *Thread* reais do *Kernel* do Windows).

Como você viu até agora neste exemplo, tínhamos uma *Label* que mostrava o andamento da execução da contagem até 10. Mas agora precisamos mostrar o andamento de n *Threads* na memória, o que seria difícil fazer com uma *Label* só. Para representar o armazenamento dos *Threads*, poderíamos criar um vetor de n posições, e através deste vetor gerenciar os *Threads* nele colocados, manipulando a contagem e a execução dos mesmos. Manipular vetores dinamicamente não é algo simples, apesar do Delphi dar grande suporte a arrays dinâmicos a partir da versão 4. Porém, necessitamos de outros recursos que se tornariam difíceis de implementar usando *arrays*.

Para tornar tudo mais fácil, e programar em alto nível e visualmente, usaremos neste exemplo um recurso muito eficiente e pouco conhecido, o componente *ClientDataSet* utilizado como vetor de memória, para guardar os dados sobre os *Threads*. Usando o *ClientDataSet* como vetor, poderemos utilizar as funções de inserção, localização, edição e exclusão. Inserir um *Thread* no vetor *ClientDataSet* é semelhante a incluir um registro em uma tabela. A localização do *Thread* no vetor também será fácil, porque temos a função *locate*.

Continuando no mesmo exemplo, vamos então criar a estrutura que armazenará as informações sobre os *Threads*.

Coloque no formulário o componente *ClientDataSet* da guia *DataAccess*, chamando-o de “*cds_threads*”. Adicione os seguintes *TFields* ao *ClientDataSet*:

```
THREAD_ID : String (20);
PRIORIDADE : String(20);
STATUS : String (20);
TEMPO_RESTANTE : Integer;
```

Agora clique de direita sobre o componente *ClientDataSet*, e escolha a opção *Create DataSet*. Isso cria nossa estrutura em memória, como um vetor. Coloque um *DataSource* no formulário e chame-o de “*ds_threads*”. Aponte sua propriedade *DataSet* para *cds_threads*. Coloque um *DBGrid* da guia *DataControls* no formulário. Aponte sua propriedade *DataSource* para *ds_threads*.

Agora precisamos criar nosso *Thread*, e colocar os dados das instâncias na estrutura do *ClientDataSet*. Antes de criarmos nossa classe descendente de *TThread*, vamos examinar o que é a classe *TThread*.

A classe TThread do Delphi

TThread é uma classe abstrata que encapsula a criação e manipulação de *Threads* em uma aplicação Windows. Utilizamos esta classe quando precisamos construir uma aplicação *MultiThread* no Delphi. A classe *TThread* está declarada na unit *classes* dessa forma :

```
// declaração simplificada
TThread = class
private
    // campos privados
protected
    procedure DoTerminate; virtual;
    procedure Execute; virtual; abstract;
    procedure Synchronize(Method: TThreadMethod);
    property ReturnValue: Integer read FReturnValue write FReturnValue;
    property Terminated: Boolean read FTerminated;
public
    constructor Create(CreateSuspended: Boolean);
    destructor Destroy; override;
    procedure Resume;
    procedure Suspend;
    procedure Terminate;
    function WaitFor: LongWord;
    property FreeOnTerminate: Boolean read FFreeOnTerminate write FFreeOnTerminate;
    property Handle: THandle read FHandle;
    property Priority: TThreadPriority read GetPriority write SetPriority;
    property Suspended: Boolean read FSuspended write SetSuspended;
```

```
property ThreadID: THandle read FThreadID;
property OnTerminate: TNotifyEvent read FOnTerminate write FOnTerminate;
end;
```

Priority - Indica a prioridade do Thread, e pode receber um dos valores da enumeração abaixo :

TThreadPriority = (*tpIdle, tpLowest, tpLower, tpNormal, tpHigher, tpHighest, tpTimeCritical*);

Suspend - Suspende a execução de um Thread (coloca em espera, mas não o termina);

Suspended - Indica se um Thread está em estado de espera. Um Thread pode estar nesse estado se for criado suspenso (passando TRUE no parâmetro do create), ou depois de uma chamada ao método suspend;

Terminate - Termina a execução de um Thread;

Terminated - Indica se um Thread foi finalizado através da chamada ao método Terminate;

Resume - Retoma a execução de um Thread que foi suspenso pela chamada ao método Suspend ou que foi criado suspenso;

ThreadId - É um número que identifica o Thread no Windows;

OnTerminate – Evento disparado quando a execução de um Thread termina;

FreeOnTerminate – Indica se o Thread vai ser liberado automaticamente quando sua execução terminar;

Execute – É um método abstrato (sem implementação na classe em que foi declarada). Ele deve ser sobrescrito (com a diretiva override) na classe que herdar de TThread. É nele que implementamos a tarefa a ser realizada pelo nosso Thread;

Synchronize – Faz comunicação(sincronismo) com o Thread principal da aplicação para atualizações de interface;

FMethod – É um ponteiro para um método do tipo TThreadMethod, o qual é utilizado como parâmetro na chamada ao método Synchronize. O tipo TThreadMethod é declarado da seguinte forma :

TThreadMethod = procedure of object;

Create – O constructor de TThread chama a função BeginThread, que por sua vez chama a função CreateThread da API para criar um objeto Thread do sistema operacional;

Agora que conhecemos melhor *TThread*, vamos criar um descendente, chamado de *TMyThread*.

Declare a classe *TMyThread* derivada de *TThread* logo depois do end da declaração da classe *TFrmMain*, como mostrado abaixo.

```
TMyThread = class (TThread)
protected
  procedure Execute; override;
public
  TempoRestante : integer;
  constructor Create (CreateSuspended : boolean);
  procedure AtualizarGrid;
end;
```

Repare a diretiva *override* sobrescrevendo o método *virtual Execute* definido por *TThread*. Pressione Shift+Ctrl+C para o Delphi gerar os cabeçalhos das implementações. Implemente o método *Execute* da seguinte forma :

```
procedure TMyThread.Execute;
var
  loop : byte;
begin
  // retire o "inherited" daqui
  FreeOnTerminate:=true;
  for loop:=10 downto 0 do
  begin
    Synchronize(AtualizarGrid);
    TempoRestante:=TempoRestante-1;
    sleep(1000);
  end;
end;
```

Implemente o método *AtualizarGrid* da seguinte forma :

```
procedure TMyThread.AtualizarGrid;
begin
```

```

with FrmMain.cds_threads do
begin
  if not Locate('THREAD_ID',inttostr(ThreadID),[]) then
  begin
    append;
    fieldbyname('THREAD_ID').AsString:=inttostr(ThreadID);
    fieldbyname('STATUS').AsString:='Aguardando';
    fieldbyname('TEMPO_RESTANTE').AsInteger:=TempoRestante;
    fieldbyname('PRIORIDADE').AsString:=
      GetEnumName(TypeInfo(TThreadPriority),integer(Priority)); //declare TypInfo no uses
    post;
  end
  else
  begin
    edit;
    fieldbyname('TEMPO_RESTANTE').AsInteger:=TempoRestante;
    if TempoRestante=0 then
      fieldbyname('STATUS').AsString:='Terminado'
    else
      fieldbyname('STATUS').AsString:='Rodando';
    post;
  end;
end;
end;

```

Este método procura pelo *Thread* no *ClientDataset* usando a função *locate*, que retorna TRUE se encontrou o registro que procuramos. Se não encontrou, então adiciona os dados do *Thread* no *ClientDataset*. Se encontrou, então atualiza os dados para mostrar os novos valores de contagem e status.

A função *GetEnumName* é uma função RTTI (*Run-Time Type Information*) do Delphi, que permite extrair a descrição em string de um tipo enumerado, neste caso *TThreadPriority*. Para usar essa função, declare *TypInfo* na cláusula *uses* da unit.

Synchronize

AtualizarGrid é o método que fará a comunicação com o formulário principal, atualizando o *DBGrid* que mostra todos os *Threads* que estão em memória no *ClientDataset*. Como falado anteriormente, um *Thread* secundário não pode modificar a interface principal do usuário (pelo menos em teoria). O tratamento da VCL só pode ser feito pelo *Thread* principal da aplicação. Por isso, esse método não é chamado diretamente no código, e sim junto com *Synchronize*. *Synchronize* manda uma mensagem ao *Thread* principal da aplicação passando o parâmetro *FMethod* que é o método a ser executado. Explicando em uma linguagem mais acessível, seria como nosso *Thread* falasse assim: “*Thread principal, eu tenho que modificar a grade no form através de AtualizarGrid, mas não posso chamar esse método porque ele atualiza os controles da VCL, e isso é só você quem pode fazer. Então eu te passo um ponteiro para esse método e você chama ele pra mim*”. De fato, uma olhada nas fontes da VCL na classe *TThread*, comprova que há comunicação entre os *Threads*. Abaixo segue a implementação do método *Synchronize* retirada das fontes da VCL. Observe a chamada *SendMessage* da API do Windows.

```

procedure TThread.Synchronize(Method: TThreadMethod);
begin
  FSynchronizeException := nil;
  FMethod := Method;
  SendMessage(ThreadWindow, CM_EXECPROC, 0, Longint(Self));
  if Assigned(FSynchronizeException) then raise FSynchronizeException;
end;

```

O passo final será criar um botão no formulário principal que instancie um ou mais objetos da classe *TMyThread*. Além disso, um *RadioGroup* indicará qual prioridade queremos dar ao *Thread* a ser criado.

Coloque um *RadioGroup* da guia Standard no formulário, dê o nome de *RdGrp_Prioridade*, coloque o valor 3 na propriedade *ItemIndex*, e na propriedade *Items* digite o seguinte (um item por linha): *tpIdle*, *tpLowest*, *tpLower*, *tpNormal*, *tpHigher*, *tpHighest*, *tpTimeCritical*

Localize a declaração do constructor de *TMyThread* e implemente-o da seguinte forma :

```

constructor TMyThread.create(CreateSuspended: boolean);
begin
  inherited create(CreateSuspended);
  TempoRestante:=10;
  Priority:=TThreadPriority(FrmMain.RdGrp_Prioridade.ItemIndex);

```

```
end;
```

CreateSuspended indica se a execução do *Thread* começará imediatamente ou ficará no aguardo até a chamada do método *Resume*. Note a atribuição da prioridade do *Thread*, onde fazemos um cast no Integer *ItemIndex* do *RadioGroup* e obtemos um valor do tipo enumerado *TThreadPriority*.

Coloque um botão no formulário com o *Caption* "Create Thread". No evento *OnClick* digite o seguinte :

```
TMyThread.create(false);
```

Rode a aplicação. Agora você pode clicar várias vezes no botão e assim instanciar vários *Threads* de uma só vez, dando a prioridade que quiser para cada um deles, e todos serão executados simultaneamente. Note que você pode mover a janela, minimizar, maximizar, coisas que não podiam ser feitas quando o contador residia no *Thread* principal da aplicação (como no início desse exemplo). Observe também a janela *Thread Status* (clique no menu *View|Debug Windows|Threads*), agora mostrando e comprovando que nosso aplicativo é *MultiThread* (nossos *Threads* são objetos reais do sistema operacional).

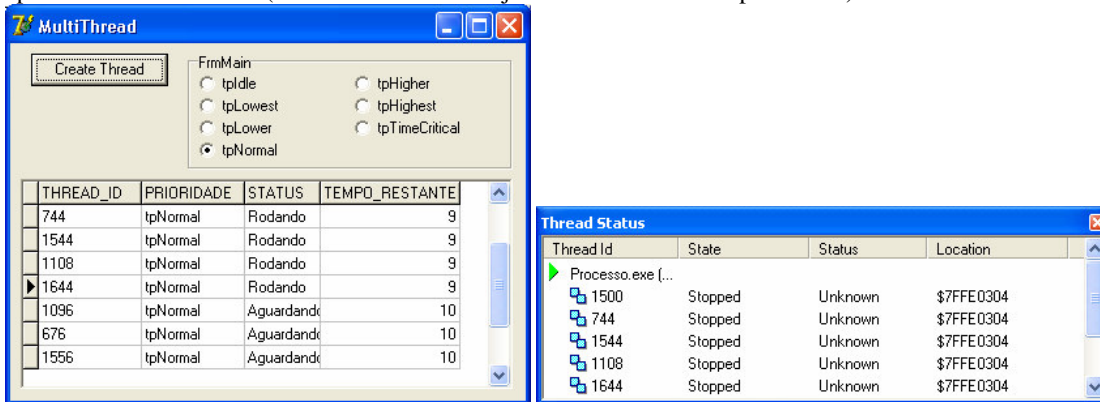


Figura. Classes transformadas em Componentes

Mutex

Note que todos os *Threads* executam o mesmo bloco de código do procedimento *Execute*, que faz o processo de contagem. Apesar de cada um dos *Threads* receber sua própria cópia do código da classe, existem determinadas ocasiões em que precisamos testar se algum *Thread* já está executando aquele determinado trecho, e esperar que ele saia do escopo.

Em nosso exemplo, vamos imaginar a seguinte situação: Digamos que um *Thread* alocasse recursos que não poderiam ser acessíveis por outro *Thread* enquanto este primeiro estiver em execução. Então só poderíamos ter um *Thread* contando de cada vez, e quando este terminar sua contagem, outro que está na fila começa a contagem (entrando então o de maior prioridade).

Para realizar tal tarefa, utilizaremos um recurso (objeto) do sistema operacional chamado *Mutex*.

Declare uma variável global chamada *MyMutexHandle*, junto com a declaração do *FrMMain*, na seção *var* da unit:

```
var
  FrmMain: TFrmMain;
  Obj : THandle;
```

Obj será a variável para manipular o *Mutex*.

No evento *OnShow* do formulário escreva o seguinte código :

```
Obj:=CreateMutex(nil, false, nil);
```

Isto cria um *Mutex* quando o formulário for mostrado (logo no início do programa)

Altere a implementação do método *Execute* como mostrado a seguir:


```

procedure TMyThread.Execute;
var
  loop : byte;
begin
  Synchronize (AtualizarGrid);
  FreeOnTerminate:=true;
  if WaitForSingleObject (Obj, INFINITE)=WAIT_OBJECT_0 then
    for loop:=10 downto 0 do
      begin
        Synchronize (AtualizarGrid);
        TempoRestante:=TempoRestante-1;
        sleep (1000);
      end;
      ReleaseMutex (Obj);
    end;
end;

```

O método *Execute* possui agora uma faixa de código protegida pelo *Mutex*. Essa faixa de código é conhecida também como uma faixa de código sincronizada ou protegida. Note que incluímos logo no início uma chamada a *Synchronize*, para que logo após um *Thread* ser criado, este imediatamente apareça na *DBGrid* em estado de espera. Observe também a função *WaitForSingleObject*, que indica o início da faixa de código sincronizada, e faz os *Threads* pararem enquanto um *Thread* estiver sendo executado, até alcançar a linha de liberação da faixa crítica, através de *ReleaseMutex*.

No evento *OnClose* do formulário escreva o seguinte código :

```
CloseHandle (Obj);
```

Isso libera o *Mutex* quando o formulário fechar. Execute a aplicação. Crie vários *Threads* com prioridades diferentes. Repare que somente um será executado, os outros permanecem na espera. Quando o primeiro *Thread* terminar sua execução, entra em execução não o segundo *Thread*, mas sim o de maior prioridade, e assim por diante.

Semáforos

Um Semáforo também funciona de maneira semelhante a um *Mutex*. A vantagem é que podemos especificar quantos objetos *Threads* podem passar por vez na faixa de código sincronizada. Observe a figura:

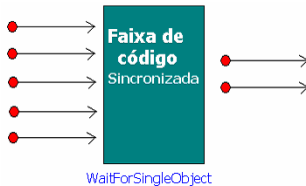


Figura. Classes transformadas em Componentes

Em nosso exemplo, vamos substituir o *Mutex* por um semáforo, que possibilitará que somente uma quantidade pré-determinada de *Threads* passe por vez na faixa de código sincronizada.

Substitua as seguintes linhas no código fonte (você pode comentar para ficar com os dois exemplos):

```

ReleaseMutex (Obj);
por
ReleaseSemaphore (Obj, 1, nil);

Obj:=CreateMutex (nil, false, nil);
por
Obj:=CreateSemaphore (nil, 2, 2, nil);

```

Observe que *CreateSemaphore* pede que o número máximo de *Threads* (que podem estar na faixa sincronizada) seja passado como parâmetro.

Execute a aplicação. Crie vários *Threads* com diferentes prioridades, e observe que somente dois deles serão executados por vez.

Seções Críticas

Uma seção crítica funciona como um *Mutex*, e também tem o objetivo de sincronizar blocos de códigos em espaços de execução crítica. Uma seção crítica faz com que um simples *Thread* possa bloquear a execução de outros *Threads*. Uma grande vantagem em usar uma seção crítica é que o Delphi encapsula as APIs de sincronismo em uma classe da VCL chamada *TCriticalSection*, que também está presente no *Kylix* (sabemos que códigos que usarem chamadas à API do Windows não são portáveis para o Linux). Assim, podemos construir aplicações *MultiThread Cross-Platform*, sem nos preocuparmos como o Windows ou o Linux criam e sincronizam de *Threads* internamente.

Substitua as seguintes linhas no código fonte:

```
Obj : THandle;
por
Obj : TCriticalSection;

WaitForSingleObject (Obj, INFINITE) = WAIT_OBJECT_0
por
Obj.Enter;

ReleaseSemaphore (Obj, 2, nil);
por
Obj.Leave;

Obj := CreateSemaphore (nil, 2, 2, nil);
por
Obj := TCriticalSection.create;

CloseHandle (Obj);
por
Obj.free;
```

Declare *SyncObjs* na cláusula *uses*.

As chamadas aos métodos da classe *TCriticalSection* equivalem a chamadas as seguintes funções da API do Windows: *InitializeCriticalSection*, *DeleteCriticalSection*, *EnterCriticalSection* e *LeaveCriticalSection*. O tipo de dados da API que equivale a *TCriticalSection* é *RTL_CRITICAL_SECTION*.

Pressione F9 e teste a aplicação.

Dica: Thread Manager – gerenciador de Threads para Delphi / Windows e Kylix / Linux
<http://codecentral.borland.com/codecentral/ccweb.exe/author?authorid=222668>