

Nesta seção você encontra artigos para iniciantes na linguagem Delphi

## Escrevendo e trabalhando com DLLs

Saiba como desenvolver e usar dlls no Delphi



Algumas das minhas grandes dúvidas no início de minha carreira como desenvolvedor Delphi, certamente foram: O que são *dll's*, como criá-las, como funcionam e onde aplicá-las? Realmente o tema sempre foi pouco discutido entre os desenvolvedores e até hoje conheço veteranos que tem certa dificuldade em trabalhar com esse formato de arquivo. Mas na verdade desenvolver e usar *dll* no dia-a-dia é mais fácil do que se pensa.

Para os iniciantes, *DLL* significa *Dynamic Link Library* ou *Biblioteca de Linkedição Dinâmica*. Mas o que vem a ser isso? Bem, precisamos entender como funciona exatamente um programa, um software para entendermos as *DLL's*. Dessa forma conseguiremos entender bem o papel delas no dia-a-dia.

O computador, em se tratando de *hardware* e como é conhecido de todos, possui uma coisa chamada memória *RAM* ou *Random Access Memory* – Memória de

### Resumo DevMan

Nem sempre um sistema trabalha sozinho, aliás, diria que nunca trabalha sozinho. Quando geramos um produto em Delphi, estamos criando um aplicativo que depende, ligeiramente, de outros programas para que funcione corretamente no sistema operacional. Algumas dessas dependências são as *DLL's*. O Windows está repleto delas por todo o sistema.

O interessante disso tudo, é que tanto o sistema operacional quanto nosso próprio sistema podem fazer uso de *DLL* para encapsular determinadas funcionalidades do software. Veremos nesse artigo como criar uma *DLL* e usá-la na prática no dia-a-dia.

### Nesse artigo veremos

- Desenvolvimento de *DLL*;
- Carregamento implícito;
- Carregamento explícito;

### Qual a finalidade

- Demonstrar como fazer o desenvolvimento de *DLL's*, como utilizá-las e onde aplicá-las;

### Quais situações utilizam esses recursos?

- Há situações em que é necessário compartilhar determinadas funcionalidades entre mais de um produto. Com o desenvolvimento de *dlls*, isso é perfeitamente possível;



### Adriano Santos

(falecom@adrianosantos.pro.br)

é desenvolvedor Delphi desde 1998, professor e programador PHP, bacharel em Comunicação Social pela Universidade Cruzeiro do Sul, SP, é editor geral das revistas ClubeDelphi e WebMobile. Mantém o blog Delphi to Delphi ([www.delphitodelphi.blogspot.com](http://www.delphitodelphi.blogspot.com)) com dicas, informações e tudo sobre desenvolvimento Delphi e é membro fundador do DUG-SP Delphi Users Group São Paulo ([www.dug-sp.com](http://www.dug-sp.com))

*Acesso Aleatório* - também chamada de *Memória Volátil*. É na memória RAM que os programas armazenam suas informações e posteriormente (dependendo da informação) enviam para o disco rígido. Portanto, quando abrimos um software qualquer como o *Word* ou *Excel*, partes do aplicativo são gravadas nessa memória. Acontece que em determinados casos, diversos programas podem compartilhar das mesmas funcionalidades, principalmente quando fazem parte do mesmo grupo (empresa) que os desenvolveu. Exemplo: *Microsoft Word*, *Excel*, *PowerPoint*, *Windows* e demais produtos. Isso significa que não é necessário termos várias instâncias do mesmo objeto em memória para executar a mesma funcionalidade. Por isso, muitas vezes se faz necessário a criação de *dll's* para que possamos disponibilizar uma ou mais funções que podem ser usadas por mais de um sistema. Esse é o principal propósito desses arquivos.

Em uma das edições passadas da *ClubeDelphi*, falei sobre *API's* (*Application Programming Interface*) e esse é um exemplo claro de como e onde usar *dll's*. Nesse artigo veremos como desenvolver uma *DLL*, as diferentes formas de carregamento e como utilizar as funções que nós mesmos desenvolveremos. Veremos o quanto é fácil integrar nosso sistema a esse tipo de arquivo e como eles podem nos ajudar no dia-a-dia. Mão na massa!

## Entendendo melhor as DLL's

Acredito que uma das vantagens mais importantes do uso de *DLL* chama-se: *encapsulamento*. Vimos em outros artigos que *encapsular* significa isolar determinadas partes do software em *um componente*,



### Nota do DevMan

*Application Programming Interfaces* são arquivos *DLL* que possuem métodos e funções em comum que podem ser utilizadas por qualquer linguagem de programação. Isso significa que quando pedimos ao *Windows* para exibir as propriedades de um determinado arquivo, uma função está sendo chamada e essa mesma função pode ser invocada usando qualquer linguagem de programação.

#### Listagem 1. Criando uma DLL

```
library Project1;
(* Important note about DLL memory management: ShareMem must be the
first unit in your library's USES clause AND your project's (select
Project-View Source) USES clause if your DLL exports any procedures or
functions that pass strings as parameters or function results. This
applies to all strings passed to and from your DLL--even those that
are nested in records and classes. ShareMem is the interface unit to
the BORLNDMM.DLL shared memory manager, which must be deployed along
with your DLL. To avoid using BORLNDMM.DLL, pass string information
using PChar or ShortString parameters. *)
uses
  SysUtils, Classes;
{$R *.res}
begin
end.
```

uma classe ou mesmo em uma *DLL*. Isso porque não precisamos (do ponto de vista do programador/codificador) saber exatamente como que funciona tal mecanismo. Precisamos apenas saber como chamar e quais parâmetros, quando necessário, passar para a função ou método desejado. Em outras palavras, basta que saibamos da existência do método e de seus requisitos para execução. Exemplificando melhor: quando precisamos executar um programa externo via codificação *Delphi*, podemos simplesmente utilizar a função *WinExec* presente na *Unit Windows* que por sua vez faz referência a *DLL* do *Kernel32.dll* do sistema operacional. Para que *WinExec* execute o programa que desejamos, passamos para ela dois parâmetros, que são:

- *lbCmdLine*: caminho completo do programa;
- *uCmdShow*: como o programa será aberto.

A linha de comando utilizada para executar a calculadora do *Windows*, por exemplo, é:

```
WinExec('C:\Windows\System32\Calc.exe',
  SW_SHOWNORMAL);
```

Muito bom, mas você sabe me dizer exatamente o que o método *WinExec* faz para executar um programa? Não é possível saber, isso porque a função *WinExec* está *encapsulada* na *DLL Kernel32.dll* como mencionei anteriormente. Mesmo porque, qual seria nossa necessidade em saber isso? Praticamente nenhuma, o mais importante é entendermos e sabermos como usar essa funcionalidade da *DLL*.

Em nosso exemplo, faremos o desenvolvimento de uma *DLL* para uso em nosso sistema. Colocaremos algumas funções e as usaremos na prática.

## Construindo a DLL

A primeira coisa que devemos saber é como funciona a estrutura de desenvolvimento de uma *DLL*. Vejamos na prática como isso funciona. Crie uma nova aplicação utilizando o *Template* para criação de *DLL's*. Você encontrará isso no menu *File>New>Other>New>DLL Wizard* no *Delphi 7* ou *File>New>Other>Delphi Projects>DLL Wizard* em versões superiores, como o *RAD Studio 2007*, por exemplo. O projeto então será criado e um código semelhante a *Listagem 1* será automaticamente gerado pelo *Delphi*.

Perceba que há uma diferença bastante grande se compararmos a um projeto de software utilizando a opção *File>New>Application*, usado para criar aplicações *Win32*. Aqui vemos que não temos um *form* e ao invés de encontrarmos *program* na primeira linha do programa, o que vem escrito é *library*, indicando que o projeto é uma *DLL*. Além disso, é claro, há outras diferenças como a ausência das sessões *Interface* e *Implementation*.

Muito bem, vamos construir uma *DLL* que contenha alguns métodos para que possamos entender como funciona o seu uso. Vamos criar métodos para converter textos em maiúsculas ou minúsculas conforme a necessidade e mais uma função para converter números inteiros em texto. É claro que esses métodos já existem no próprio *Delphi*, mas vamos encarar como uma novidade em nosso sistema.

A primeira coisa que devemos fazer é inserir o código da *Listagem 2* logo após a diretiva de compilação *{\$R \*.res}*. Certifique-se de que estará acima do *begin...end* ao final da *Unit*. Se tirarmos o bloco de comentário gerado pelo *Delphi*, nossa *DLL* ficará como na *Listagem 3*.

Repare que temos algumas novidades no código em relação a aplicações *Win32*. Uma delas é o retorno das funções. Estamos utilizando *ShortString* ao invés de *String* como de costume. Isso é uma medida adotada para compatibilizar nossa *DLL* com outras linguagens de programação. Em *C*, por exemplo, não existem tipos de *string* longos como no *Delphi*. Então, se algum programador *C* precisar utilizar nossa *DLL* não terá problemas de incompatibilidade.

A segunda novidade é a palavra reservada *exports*. Perceba que declaramos nessa sessão as três funções que criamos. É nesse momento que informamos a *DLL* quais funções estão acessíveis ao mundo externo. Veja, que criamos também uma *procedure* chamada *MensagemConfirmacao* que é chamada sempre que um método é executado. Essa *procedure* não possui a palavra reservada *stdcall* e também não

foi declarada em *exports*, ou seja, não será enxergada por processos externos.

Crie um diretório em seu micro e salve o projeto com o nome *MinhaDll.dpr*. Para facilitar, vamos criar um grupo de projetos e adicionar a ele os nossos exemplos. Clique em *View>Project Manager* e você verá uma tela semelhante a **Figura 1**.

Clique com o direito em *ProjectGroup1* e *Save Project Group As...* e dê o nome de *ProjetoDLL* ou o nome que preferir. Agora ficará mais fácil trabalharmos com mais de um projeto ao mesmo tempo sem precisar abrir mais de um *Delphi* ou ficar abrindo e fechando nossos projetos. Em teoria, nossa *DLL* está pronta. Pressione *Ctrl + F9* e veja se o *Delphi* compila normalmente nosso projeto.

## Chamando a DLL implicitamente

Há basicamente dois modos de se utilizar uma *DLL*: implícito e explícito. O

modo implícito se dá quando declaramos as funções/métodos a serem utilizados diretamente no software. Esse método é usado por “n” motivos e um deles é quando há necessidade obrigatória do uso da *DLL*, isso porque quando declaramos as funções a serem usadas diretamente no projeto o aplicativo carregará a biblioteca (*DLL*) no mesmo instante e a manterá em memória. Por isso precisamos que ela esteja no mesmo diretório do executável final de nossa aplicação.

Em outras palavras, o método implícito carrega a *DLL* para memória e a deixa lá até que o sistema que a usa seja fechado. Se não houver a necessidade do arquivo manter-se carregado o tempo todo, esse método deve ser desprezado. Vejamos como fazer uso desse método para carregar e utilizar os métodos criados em nosso projeto.

A primeira coisa é criarmos um projeto dentro de nosso grupo. Para isso clique com o botão direito no nome de nosso grupo no *Project Manager* e selecione *Add New Project*. Na aba *New* escolha *Application*. Desenhe uma tela semelhante a **Figura 2**.

Nós temos alguns *Labels* para dar nomes as controles *Edit* e *SpinEdit* em tela. E mais três *buttons*. Se preferir troque os nomes dos componentes, mas nesse exemplo mantive todos com os nomes *default*. Os

### Listagem 2. Código das funções da DLL

```
procedure MensagemConfirmacao(AMensagem: WideString);stdcall;
begin
  MessageDlg(AMensagem, mtInformation, [mbOk], 0);
end;
function MeuIntToStr(Numero: Integer): ShortString;stdcall;
begin
  Result := IntToStr(Numero);
  MensagemConfirmacao('Convertido com sucesso!');
end;
function MeuUpperCase(s: ShortString): ShortString; stdcall;
begin
  Result := UpperCase(s);
  MensagemConfirmacao('Texto convertido em Maiúsculas!');
end;
function MeuLowerCase(s: WideString): ShortString; stdcall;
begin
  Result := LowerCase(s);
  MensagemConfirmacao('Texto convertido em Minúsculas!');
end;
exports
  MeuUpperCase, MeuLowerCase, MeuIntToStr;
```

### Listagem 3. Código completo da DLL

```
library MinhaDLL;
uses
  SysUtils, Classes;

{$R *.res}
procedure MensagemConfirmacao(AMensagem: WideString);stdcall;
begin
  MessageDlg(AMensagem, mtInformation, [mbOk], 0);
end;
function MeuIntToStr(Numero: Integer): ShortString;stdcall;
begin
  Result := IntToStr(Numero);
  MensagemConfirmacao('Convertido com sucesso!');
end;
function MeuUpperCase(s: ShortString): ShortString; stdcall;
begin
  Result := UpperCase(s);
  MensagemConfirmacao('Texto convertido em Maiúsculas!');
end;
function MeuLowerCase(s: WideString): ShortString; stdcall;
begin
  Result := LowerCase(s);
  MensagemConfirmacao('Texto convertido em Minúsculas!');
end;
exports
  MeuUpperCase, MeuLowerCase, MeuIntToStr;
end.
```

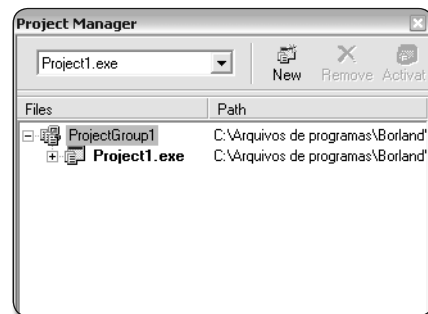


Figura 1. Project Manager



Figura 2. Exemplo de uso da DLL implicitamente

dois primeiros botões (*Maiúsculas* e *Minúsculas*) farão uma chamada aos métodos `MeuUpperCase` e `MeuLowerCase`, para transformar o texto do primeiro *Edit* em maiúsculas ou minúsculas. Já o terceiro botão fará uma chamada ao método `MeuIntToStr` e enviará o número convertido para o segundo *Edit*.

Para que possamos utilizar as funções mencionadas, teremos que declará-las no corpo da *Unit* de nosso *form*. Localize então a palavra reservada *Implementation* do formulário e logo acima da declaração do *form* digite o código da **Listagem 4**.

Aqui estamos informando ao executável, quem é a *DLL* que possui tal método e qual o nome dessa função. A digitação errada do *name* do método pode causar um erro de *Access Violation*, já que o executável não consegue encontrar a função declarada. Após isso, precisamos apenas fazer a chamada normalmente no evento *OnClick* de cada botão. Veja o código da **Listagem 5** atentamente, e implemente a chamada para cada *Button* em tela. Em seguida, compile o projeto e execute-o. Procure testar cada funcionalidade e verificar se está tudo de acordo.

Para provar que a *DLL* foi carregada para a memória, faça um teste. Abra o sistema e em seguida abra o *Windows Explorer* ou *Meu Computador* e navegue até a pasta em que salvou o projeto e a *DLL*. Tente apagar a *DLL* com o programa aberto. Verá que o *Windows* exibirá uma mensagem informando que não é possível excluir o arquivo, pois possivelmente está em uso.

Outro teste interessante de se fazer é apagar a *DLL* e então tentar abrir o sistema. Um erro como na **Figura 4** será exibido. Isso acontece devido ao que informei antes. Como estamos usando-a

**Listagem 4.** Declaração das funções da *DLL* no projeto

```
[...]
function MeuUpperCase(s: ShortString): ShortString; stdcall;
external 'MinhaDll.dll' name 'MeuUpperCase';
function MeuLowerCase(s: WideString): ShortString; stdcall;
external 'MinhaDll.dll' name 'MeuLowerCase';
function MeuIntToStr(Numero: Integer): ShortString; stdcall;
external 'MinhaDll.dll' name 'MeuIntToStr';
var
  Form1: TForm1;
[...]
```

A novidade aqui é que adicionamos o trecho a seguir depois da palavra reservada *stdcall* de cada função.

```
external 'MinhaDll.dll' name 'MeuUpperCase';
```

**Listagem 5.** Código dos botões de tela

```
procedure TForm1.Button4Click(Sender: TObject);
begin
  Edit1.Text := MeuUpperCase(Edit1.Text);
end;
procedure TForm1.Button5Click(Sender: TObject);
begin
  Edit1.Text := MeuLowerCase(Edit1.Text);
end;
procedure TForm1.Button3Click(Sender: TObject);
begin
  Edit2.Text := MeuIntToStr(SpinEdit1.Value);
end;
```

implicitamente em nosso projeto, o sistema a carrega logo que entra no ar.

## Chamando a *DLL* explicitamente

O modo explícito de se fazer o carregamento da *DLL* exige um pouco mais de explicações, pois faremos todo o processo dinamicamente. Podemos dizer que em determinadas situações é mais vantajoso usar esse método, pois a *DLL* não aloca espaço em memória desnecessário. Outra grande vantagem diz respeito à distribuição. É perfeitamente possível fazer a exclusão da biblioteca com o programa em execução, ou seja, caso você possua um programa que faz a atualização automática de versão de sua *DLL*, o programa que utiliza tal arquivo não precisa necessariamente ser fechado, a menos que esteja usando alguma funcionalidade da *DLL*.

O carregamento dinâmico é feito utilizando três *API's* do *Windows* presentes na *DLL Kernel32.dll*, são elas:

- *LoadLibrary*: responsável por carregar a *DLL* para a memória;
- *GetProcAddress*: recupera o endereço de

memória da função que precisamos usar;

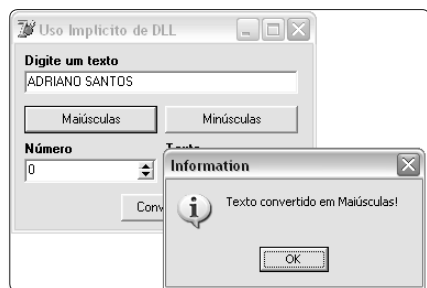
- *FreeLibrary*: elimina da memória a *DLL* carregada.

A título de curiosidade, a declaração de cada uma pode ser vista a seguir e estão declaradas na *Unit Windows* do *Delphi*.

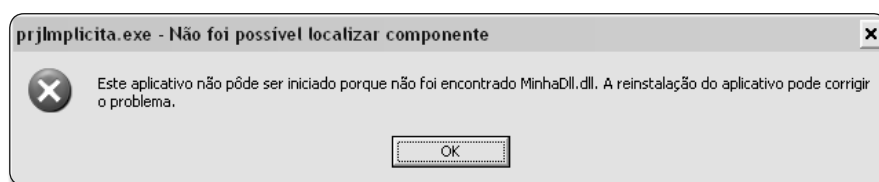
```
function LoadLibrary;
external kernel32 name 'LoadLibraryA';
function GetProcAddress;
external kernel32 name 'GetProcAddress';
function FreeLibrary;
external kernel32 name 'FreeLibrary';
```

Para que tenhamos um bom parâmetro de comparação entre os dois métodos, faça um clone da aplicação que criamos anteriormente. Clique novamente com o botão direito no grupo de projetos no *Project Manager* e escolha *Add New Project*. Salve o novo projeto como *prjExplicito.dpr* ou o nome que preferir. Abra o formulário do primeiro projeto que desenvolvemos, copie todos os objetos e cole-os no formulário desse novo projeto.

A primeira medida que devemos tomar é criar três classes que farão referência às funções em nossa *DLL*. Faça isso digitando o código a seguir antes da sessão *type* do formulário. Veja:



**Figura 3.** Teste no sistema carregando a *dll* implicitamente



**Figura 4.** Erro ao tentar executar o sistema sem a presença da *MinhaDll.dll* no diretório

**Listagem 6. Código para uso da função Maiúsculas (dinamicamente)**

```
procedure TForm2.Button4Click(Sender: TObject);
var
  Handle: THandle;
  mUpperCase: TMeuUpperCase;
begin
  Handle := LoadLibrary('MinhaDLL.dll');
  if Handle <> 0 then
  begin
    mUpperCase := GetProcAddress(Handle, 'MeuIntToStr');
    Edit2.Text := mUpperCase(Edit2.Text);
    FreeLibrary(Handle);
  end;
end;
```

**Listagem 7. Código do botão Minúsculas**

```
procedure TForm2.Button5Click(Sender: TObject);
var
  Handle: THandle;
  mLowerCase: TMeuLowerCase;
begin
  Handle := LoadLibrary('MinhaDLL.dll');
  if Handle <> 0 then
  begin
    mLowerCase := GetProcAddress(Handle, 'MeuLowerCase');
    Edit2.Text := mLowerCase(Edit2.Text);
    FreeLibrary(Handle);
  end;
end;
```

**Listagem 8. Código do botão Converter**

```
procedure TForm2.Button3Click(Sender: TObject);
var
  Handle: THandle;
  mIntToStr: TMeuIntToStr;
begin
  Handle := LoadLibrary('MinhaDLL.dll');
  if Handle <> 0 then
  begin
    mIntToStr := GetProcAddress(Handle, 'MeuIntToStr');
    Edit3.Text := mIntToStr(SpinEdit1.Value);
    FreeLibrary(Handle);
  end;
end;
```

```
type
  TMeuUpperCase =
    function(s: ShortString): ShortString;
  TMeuLowerCase =
    function(s: ShortString): ShortString;
  TMeuIntToStr =
    function(Numero: Integer): ShortString;
```

Agora faremos a codificação do botão *Maiúsculas*. Clique duas vezes nele para acessarmos seu evento *OnClick* e então digite o código da **Listagem 6**. Vejamos o que temos aqui. Declaramos duas variáveis: *Handle* e *mUpperCase*. *Handle* receberá a referência da *DLL* em memória, ou seja, atribuímos a ela o resultado do método *LoadLibrary*, que é justamente o endereço de memória. Todo programa, quando iniciado(instanciado), recebe um endereço de memória que é a forma como o sistema operacional o mapeia.

A variável *mUpperCase* por sua vez, receberá o endereço de memória da função *MeuUpperCase* que está declarada em nossa variável. Isso significa, que poderemos acessar o resultado da função atribuindo-a a um objeto em tela, nesse caso a propriedade *Text* do *Edit2*. Por fim

chamamos o método *FreeLibrary* para liberar de memória nossa *DLL*. O interessante disso tudo é que nossa biblioteca não permanecerá em memória depois de sua utilização. Experimente agora codificar os demais botões conforme as **Listagens 7 e 8**, sendo a 7 é do botão *Minúsculas* e 8 do *Converter*. A explicação é exatamente a mesma.

Rode a aplicação e faça o mesmo teste que foi feito com nosso projeto anterior. Tente apagar a *DLL* do diretório com o software aberto ou iniciar ele sem a presença dela. O sistema somente acusará a falta da mesma ao clicarmos em um dos botões que fazem sua chamada. Na verdade, não será exibido nenhum erro. Os botões apenas não funcionarão como se não tivéssemos programado nenhuma funcionalidade para eles.

Há algumas formas de contornar esse pequeno problema. No evento *OnClick* dos botões podemos fazer uma checagem de rotina usando o método *FileExists*, que verifica se determinado arquivo existe. Veja:

```
if not FileExists(
  ExtractFilePath(Application.ExeName)+
  'MinhaDLL.dll') then
begin
  ShowMessage('DLL não encontrada.');
```

Podemos incluir esse código antes da chamada ao método *LoadLibrary*. Assim, caso não exista a *DLL* no diretório da aplicação o programa avisará e o fluxo do evento será parado (*Exit*).

Outra forma é incluir um *else* no *IF* no *Handle*. Se a variável *Handle* retornar 0 (zero) significa que não conseguiu carregar a *DLL* por algum motivo, então exibimos uma mensagem.

```
if Handle <> 0 then
begin
end
else
  ShowMessage(
    'Não foi possível carregar a DLL.');
```

## Considerações finais

O uso de *DLLs*, como pudemos ver, não é nenhum bicho de sete cabeças. Com esse estudo abrimos brecha para o desenvolvimento de diversas soluções para os mais variados problemas do dia-a-dia e que podem ser solucionados desenvolvendo uma simples *DLL*. Um bom exemplo disso seria criar um sistema de licenciamento do software através de *DLL*. É claro, que o uso desse recurso deve ser bastante pensado para evitar maiores problemas, principalmente quando se guarda dados confidenciais dentro desse tipo de arquivo.

## Conclusão

Como pudemos perceber, o uso de *DLL* é bastante simples e não requer nenhum conhecimento específico. O importante é entender o conceito de uso desses arquivos aplicá-lo corretamente.

Procure estudar mais o processo de desenvolvimento de *DLL* e quais tipos de dados são limitados no desenvolvimento. Um forte abraço e até a próxima. ●

**Dê seu feedback sobre esta edição!**

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

[www.devmedia.com.br/clubedelphi/feedback](http://www.devmedia.com.br/clubedelphi/feedback)

