How to run a single instance of an application (part 3 of 4)

An object oriented solution

An object oriented solution

Overview

In this section we present an object oriented, extensible, solution to our problem. Most of the solution's code is stored in a unit named `USingleInst.pas`.

Our solution hinges around a singleton object, of base type *TSingleInst*, that determines whether or not an application is the only instance running. If there is already an instance of the application running when a second copy is started the object passes the second copy's command line to the original instance before terminating the second copy. The object also names the main form's window class and can supply a watermark to be used to test the integrity of *WM_COPYDATA* messages.

Users are expected to override *TSingleInst* in order to provide a suitably strong window class name for the main form and, optionally, to provide a watermark. We must register the newly derived class so that a singleton of the correct type can be created. We will provide a routine, *RegisterSingleInstClass*, to perform the registration. Our singleton is always accessed via the *SingleInst* function, which creates the object the first time the function is called. It is apparent, therefore, that we must register our *TSingleInst* descendant before we access the singleton for the first time. The safest way to accomplish this is by placing our call to *RegisterSingleInstClass* in the initialization section of the unit where the derived class is implemented.

Minimal modifications must still be made to the application's project file and the main form. In each case we simply call one of the singletion object's methods.

To summarise then, the procedure for enabling single application instance support in a project using `USingleInst.pas` is as follows:

- Add `USingleInst.pas` to the project.
- Create a new class derived from *TSingleInst* and override the methods that return the window class name and the watermark.
- Register the new class as the one to be used when creating the *SingleInst* singleton object by adding a call to *RegisterSingleInstClass* in the initialization section of the derived class's unit.
- Modify the main form and project sources to call relevant methods in the *SingleInst* object.

In the rest of this section we will first review the `USingleInst` unit, then look at how to derive and register a *TSingleInst* descendant class. Finally we will describe the changes to be made to the project and main form source files.

The USingleInst unit

The full source of the unit is included in the *demo project* that accompanies this article.

Let us begin with an examination of *Listing 15*, where the interface part of the unit is presented.

```
unit USingleInst;

interface

uses
  Windows, Controls, Messages;
```

```
type
  TSingleInstParamHandler =
    procedure(const Param: string) of object;

  TSingleInstClass = class of TSingleInst;

  TSingleInst = class(TObject)
  private
    fOnProcessParam: TSingleInstParamHandler;
    fEnsureRestoreMsg: UINT;
  protected
    function WdwClassName: string; virtual;
    function WaterMark: DWORD; virtual;
    function FindDuplicateMainWdw: HWND; virtual;
    function SendParamsToPrevInst(Wdw: HWND): Boolean;
      virtual;
    function SwitchToPrevInst(Wdw: HWND): Boolean;
    procedure EnsureRestore(Wdw: HWND; var Msg: TMessage);
      dynamic;
    procedure WMCopyData(var Msg: TMessage); dynamic;
  public
    constructor Create;
    procedure CreateParams(var Params: TCreateParams);
    function HandleMessages(Wdw: HWND;
      var Msg: TMessage): Boolean;
    function CanStartApp: Boolean;
    property OnProcessParam: TSingleInstParamHandler
      read fOnProcessParam write fOnProcessParam;
  end;

function SingleInst: TSingleInst;

procedure RegisterSingleInstClass(Cls: TSingleInstClass);
```

Listing 15

TSingleInstParamHandler is the type of the *OnProcessParam* event discussed below. *TSingleInstClass* is the class type of *TSingleInst* and any derived classes.

Moving on to the *TSingleInst* class, we have two fields:

1. *fOnProcessParam* – stores the user defined handler for the *OnProcessParam* event. This event is triggered once for each parameter that was passed to the application via a *WM_COPYDATA* message.
2. *fEnsureRestoreMsg* – stores the id of a custom message used to ask an application to restore and display its main window. The message id is provided by Windows and guaranteed to be unique. Unfortunately, because the message is not a constant we can't use a Delphi message method to handle it. Instead we have to resort to inercepting messages from the message loop (see later).

The methods of *TSingleInst* will be discussed in detail as we review their implementation.

Following the *TSingleInst* declaration, we declare two functions, both of which we have already discussed in the Overview:

1. *SingleInst* – used to return an instance of the required *TSingleInst* (or descendant) singleton object.
2. *RegisterSingleInstClass* – called to register the *TSingleInst* descendant class that is to be used to ceate the singleton.

The source code for these functions, along with the declaration of two global variables required by the functions, is shown in *Listing 16*.

```
implementation

uses
  SysUtils, Forms;

var
  // Globals storing singleton and class of SingleInst
  gSingleInst: TSingleInst = nil;
  gSingleInstClass: TSingleInstClass = nil;

function SingleInst: TSingleInst;
```

```
begin
  if not Assigned(gSingleInst) then
  begin
    if Assigned(gSingleInstClass) then
      gSingleInst := gSingleInstClass.Create
    else
      gSingleInst := TSingleInst.Create;
  end;
  Result := gSingleInst;
end;

procedure RegisterSingleInstClass(Cls: TSingleInstClass);
begin
  gSingleInstClass := Cls;
end;
```

Listing 16

Class variables

In later versions of Delphi *gSingleInst* and *gSingleInstClass* could be made into class variables of *TSingleInst* the and the functions could become class methods.

The private global variables *gSingleInst* and *gSingleInstClass* store a reference to the singleton object and the type of the singleton object respectively.

As already noted, *SingleInst* creates the singleton object the first time the function is referenced. If a derived class has been registered then this class is used to create the singleton, otherwise the base class is used. *RegisterSingleInstClass* simply stores the class reference passed to it.

Moving on to the class definition, we first look at the class constructor which simply registers the custom message with Windows, as *Listing 17* reveals.

```
constructor TSingleInst.Create;
begin
  inherited;
  fEnsureRestoreMsg := RegisterWindowMessage(
    'USINGLEINST_ENSURERESTORE'
  );
end;
```

Listing 17

There are various "entry points" into the class that must be called from either the main unit or the project file. *Listing 18* has the details:

```
function TSingleInst.CanStartApp: Boolean;
var
  Wdw: HWND;
begin
  Wdw := FindDuplicateMainWdw;
  if Wdw = 0 then
    Result := True
  else
    Result := not SwitchToPrevInst(Wdw);
end;

procedure TSingleInst.CreateParams(
  var Params: TCreateParams);
begin
  inherited;
  StrPLCopy(
    Params.WinClassName,
    WdwClassName,
    SizeOf(Params.WinClassName) div SizeOf(Char) - 1
  );
end;

function TSingleInst.HandleMessages(Wdw: HWND;
  var Msg: TMessage): Boolean;
begin
  if Msg.Msg = WM_COPYDATA then
  begin
    WMCopyData(Msg);
    Result := True;
  end
```

```
  else if Msg.Msg = fEnsureRestoreMsg then
  begin
    EnsureRestore(Wdw, Msg);
    Result := True;
  end
  else
    Result := False;
end;
```

Listing 18

The first of these "entry points" is *CanStartApp* which is called from the project file and indicates whether the application should be started or not. It works in a very similar way to the *CanStart* function.

The *CreateParams* method sets up the main form's window class name by updating the form's window creation parameters. The main form must override its inherited *TForm.CreateParams* method and call *SingleInst.CreateParams* from within that method.

The most complex of the "entry point" methods is the *HandleMessages* method. This intercepts *WM_COPYDATA* and the custom *fEnsureRestoreMsg* messages from the main form and handles the processing by delegating to *WMCopyData* and *EnsureRestore* respectively. This method returns true if it handles a message and false otherwise. The main form must override the *TForm.WndProc* method and call *SingleInst.HandleMessages* from there, calling its inherited *TForm.WndProc* method if *HandleMessages* returns false.

Let us now examine the protected helper methods. Firstly, *Listing 19* shows two methods that *should* be overridden in descendant classes:

```
function TSingleInst.WdwClassName: string;
begin
  Result := 'SingleInst.MainWdw';
end;

function TSingleInst.WaterMark: DWORD;
begin
  Result := 0;
end;
```

Listing 19

WdwClassName simply returns the name of the main form's window class. This *should* be overridden to return some unique value to the application. Similarly, *Watermark* returns 0. Again, this method *should* be overridden to return some unusual value if watermarks are to be used.

Next up, in *Listing 20*, are two methods that assist in handling messages intercepted in the main form:

```
procedure TSingleInst.EnsureRestore(Wdw: HWND;
  var Msg: TMessage);
begin
  if IsIconic(Application.Handle) then
    Application.Restore;
  if Assigned(Application.MainForm)
    and not Application.MainForm.Visible then
    Application.MainForm.Visible := True;
  Application.BringToFront;
  SetForegroundWindow(Wdw);
end;

procedure TSingleInst.WMCopyData(var Msg: TMessage);
var
  PData: PChar;
  Param: string;
begin
  if TWMCopyData(Msg).CopyDataStruct.dwData = WaterMark then
  begin
    PData := TWMCopyData(Msg).CopyDataStruct.lpData;
    while PData^ <> #0 do
    begin
      Param := PData;
      if Assigned(fOnProcessParam) then
        fOnProcessParam(Param);
      Inc(PData, Length(Param) + 1);
```

```
    end;
    Msg.Result := 1;
  end
  else
    Msg.Result := 0;
end;
```

Listing 20

Notice that these methods are based closely on the *UMEnsureRestored* and *WMCopyData* message handler methods that were discussed earlier. Note though that the methods are no longer message handlers – they are now dynamic methods. There are other changes to the methods that should be noted:

1. *EnsureRestore* is now passed a handle to the window to be brought to the foreground. (this will be the window handle of the main form).
2. *WMCopyData* now triggers the *OnProcessParam* event rather than calling a hard-wired method to process parameters.

Finally we examine the three remaining protected methods:

```
function TSingleInst.FindDuplicateMainWdw: HWND;
begin
  Result := FindWindow(PChar(WdwClassName), nil);
end;

function TSingleInst.SendParamsToPrevInst(
  Wdw: HWND): Boolean;
var
  CopyData: TCopyDataStruct;
  I: Integer;
  CharCount: Integer;
  Data: PChar;
  PData: PChar;
begin
  CharCount := 0;
  for I := 1 to ParamCount do
    Inc(CharCount, Length(ParamStr(I)) + 1);
  Inc(CharCount);
  Data := StrAlloc(CharCount);
  try
    PData := Data;
    for I := 1 to ParamCount do
    begin
      StrPCopy(PData, ParamStr(I));
      Inc(PData, Length(ParamStr(I)) + 1);
    end;
    PData^ := #0;
    CopyData.lpData := Data;
    CopyData.cbData := CharCount * SizeOf(Char);
    CopyData.dwData := WaterMark;
    Result := SendMessage(
      Wdw, WM_COPYDATA, 0, LPARAM(@CopyData)
    ) = 1;
  finally
    StrDispose(Data);
  end;
end;

function TSingleInst.SwitchToPrevInst(Wdw: HWND): Boolean;
begin
  Assert(Wdw <> 0);
  if ParamCount > 0 then
    Result := SendParamsToPrevInst(Wdw)
  else
    Result := True;
  if Result then
    SendMessage(Wdw, fEnsureRestoreMsg, 0, 0);
end;
```

Listing 21

These methods are analogues of the routines of the same name in the previous section, with the following differences:

- *FindDuplicateMainWdw* gets the the name of the required window class from the *WdwClassName* method rather than a constant.
- *SendParamsToPrevInst* gets its watermark from the *Watermark* method rather than a constant.
- *SwitchToPrevInst* sends the registered message identified by *fEnsureRestoreMsg* to the main form window rather the hard-wired *UM_ENSURERESTORED* message.

Modifications to the project file

We must make a simple change to the project file so that it calls the *CanStartApp* method of the *SingleInst* singleton object to check whether the application can be started. First of all add *USingleInst* to the project then change the project file as shown in *Listing 22*:

```
...
begin
  if SingleInst.CanStartApp then
  begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
  end;
end.
```

Listing 22

Modifications to the main form

Slightly more work needs to be done in the main form unit. First, add the following declarations to the form class's protected section:

```
  protected
    procedure WndProc(var Msg: TMessage); override;
    procedure CreateParams(
      var Params: TCreateParams); override;
```

Listing 23

Now implement these methods to call into *SingleInst* as per *Listing 24*:

```
procedure TForm1.CreateParams(var Params: TCreateParams);
begin
  inherited;
  SingleInst.CreateParams(Params);
end;

procedure TForm1.WndProc(var Msg: TMessage);
begin
  if not SingleInst.HandleMessages(Msg) then
    inherited;
end;
```

Listing 24

CreateParams calls its ancestor method then calls *SingleInst.CreateParams*. This method sets the window class name.

The overridden *WndProc* method calls *SingleInst.HandleMessages* which checks if the message is one of those handled by *SingleInst*. The method returns true if the message was handled and false otherwise. If the message was not handled then the message is passed to the inherited *WndProc* method.

We also need to handle the *TSingleInst.OnProcessParam* event so the application gets notified when parameters are passed to it via the *WM_COPYDATA* message. We define a new method, *HandleParam* to handle this event. This method is declared in the form's protected section as per *Listing 25*:

```
  protected
    procedure HandleParam(const Param: string);
```

Listing 25

HandleParam is where we add whatever processing we wish to do with the parameters. In this example we just want to display the parameters in a list box, so we implement the method as follows:

```
procedure TForm1.HandleParam(const Param: string);
begin
  ListBox1.Items.Add(Param);
end;
```

Listing 26

We assign the event handler in *FormCreate*, where we also process any parameters passed to the program when it first starts:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  SingleInst.OnProcessParam := HandleParam;
  for I := 1 to ParamCount do
    HandleParam(ParamStr(I));
end;
```

Listing 27

Overriding TSingleInst

We now look at an example of how to override *TSingleInst*. The class was designed so that we only need to override the *WdwClassName* and, if desired, the *Watermark* methods to provide a unique windows class name and stronger watermark.

The whole unit (`UMySingleInst.pas`) is listed below. The only point of note is how the new class is registered so that it is used to create the *SingleInst* singleton object.

```
unit UMySingleInst;

interface

uses
  Windows,
  USingleInst;

type
  TMySingleInst = class(TSingleInst)
  protected
    function WdwClassName: string; override;
    function WaterMark: DWORD; override;
  end;

implementation

{ TMySingleInst }

function TMySingleInst.WaterMark: DWORD;
begin
  Result := $DE1F1DAB;
end;

function TMySingleInst.WdwClassName: string;
begin
  Result := 'DelphiDabbler.SingleInst.1';
end;

initialization

// Make sure we use this class for the singleton
RegisterSingleInstClass(TMySingleInst);
```

```
end.
```

Listing 28

That completes the presentation of the object oriented solution. In the *final section* we draw the article to a close and discuss the article's demo program.
