Implementation Of A Single Instance Class

by John Chaytor

This article demonstrates a technique which can be used to ensure that no more than one instance of a class can be created in an application. This is achieved by making use of a little used Delphi feature, class methods. An example of where this would be useful is the `TSession` class, defined in DB.PAS. This object is created automatically for you and is accessed though the global variable `Session`. The documentation for this class states that only one instance should be created in an application and instructs you to not create another, but there is nothing to actually stop you from doing so.

To demonstrate the technique, a demo application (SNGLINST) has been provided on this month's disk which defines a new class called `TSLSingle`, that I derived from `TStringList`, which ensures that only one instance can be created, no matter how many times `Create` is called. The app simply shows client views of data (loaded from a file). Whenever the data is amended, all client views are refreshed automatically via event procedures. Only a few methods have been overridden to illustrate the concepts, it is not intended to be a full implementation. To demonstrate the implications of deriving new classes from this 'single instance' class the application defines a second class called `TSLSingleNum` which simply places a restriction that all strings added must start with a digit. The rest of this article describes the techniques used. Extra comments are in the source code.

Class Methods

If you have never encountered class methods before they can be a bit confusing initially. A class method is a function or procedure which operates on a class rather than an individual instance of that class. That is, a class method will perform processing regardless of the number of instances which may exist – even if no objects exist. However, a restriction put upon class methods is that they cannot access any instance data (for example the field definitions in the `private` section of a class declaration). This is understandable as instance data does not exist until an object is created.

You may have already used class methods without knowing it. For example `ClassName`, which is defined in the class `TObject`, is available for all classes. You can call this method without needing to create any objects. The code below shows two class functions being used. `ClassParent` returns the parent class of `TForm` and `ClassName` returns the `Name` of that class:

```
ShowMessage(
  TForm.ClassParent.ClassName);
```

This will display `TScrollingWinControl` in a `MessageBox`. This is possible because these functions are accessing data set up by the compiler 'behind the scenes'. The class functions know how to access that data for you. We will use a similar technique when we implement the `Create` constructor for our class.

Constructors

As we all know, the first thing you must do when using an object is create the object. This is usually done by a call to `Create` using a class reference. Each call to `Create` will cause a new instance to be created on the heap. So given the following two lines of code:

```
MyObjectA := TMyClass.Create;
MyObjectB := TMyClass.Create;
```

two object instances will be created; `MyObjectA` and `MyObjectB` will contain different pointers. However, for our class we want a single instance to be created. Any subsequent calls to `Create` need to return the pointer to the *original* instance. Hence, in the above case `MyObjectA` and `MyObjectB` would point to the *same* object instance. The next section explains how this is done.

A Class Method Constructor

Listing 1 shows a typical type definition for a regular class (`TStandardClass`), along with a simplified type definition for our class (`TSLSingle`). The good news is that the changes aren't drastic, but there are some subtle points which need to be addressed.

The first point to note is that the `Create` for our class is not a constructor at all. It is, in fact, a class function which just happens to be called `Create`. This function returns a type of `TSLSingle`. Compare this to the normal constructor where no return type is specified (as the compiler knows this information).

Listing 2 shows a cut down version of the source on the disk. It shows all the important points we need to address here. See the source code for further discussion.

As you can see, our class function `Create` accesses two variables, `FInstance` and `FUsage`. As we are unable to access instance data in class functions these fields cannot be contained within the object. So where are they? These fields are defined as typed constants in the `implementation` part of the unit. As such, they are stored in the data segment, but are private to our unit.

The `Create` class function first increases the `FUsage` count, then if we have not already created the object, it calls the `protected`

constructor RealCreate to create it. This is possible as Class methods are allowed to call constructors and destructors. Since the RealCreate constructor is defined as protected, this means that nobody using our class can create an instance behind our back! Hence, we always know how many instances have been created.

The pointer to the object, returned from RealCreate, is stored in FInstance and passed back to the caller. If a subsequent call is made to Create (determined by the usage count), the function simply updates the FUsage count and returns the instance address stored in FInstance. This is analogous to the way DLLs work.

The code you put in the RealCreate constructor is exactly the same type of code you would put in a standard Create method, including calls to the inherited constructor (usually Create).

Destroying The Object

Again, as everyone knows, you call the Free method to destroy an object. Ordinarily, when you call Free, the Destroy destructor is called as long as the pointer is not nil. This is definitely not what we want! If we allowed this to happen we would generate GPFs whenever more than one instance had been created, as the data would be freed from the heap on the first request. To avoid this, the class explicitly defines both Free and Destroy methods. (Note: Destroy is a method, not a destructor). This means that the class destructor is only called when we explicitly call it.

The Free method simple calls the Destroy method, where all the logic resides. This is a safeguard against people who call Destroy instead of Free! The Destroy method makes use of the FUsage variable. Each time Destroy is called, the FUsage count is decreased. Only when the value reaches zero do we actually destroy the object by calling the RealDestroy destructor. When we finally free the object we set the FInstance variable to nil. The code you put in the RealDestroy destructor is exactly the same type of code

➤ *Listing 1*

```
type
  TStandardClass = class(TStringList)
    public
      constructor Create;
      destructor Destroy; override;
  end;
  TSLSingle = class(TStringList)
  protected
    constructor RealCreate; virtual;
    destructor RealDestroy; virtual;
  public
    class function Create: TSLSingle;
    procedure Destroy;
    procedure Free;
  end;
```

➤ *Below: Listing 2*

```
unit Snglins9;
interface
type
  TSLSingle = class(TStringList)
    FOnChangedList: TList;
  protected
    procedure Changed; override;
    constructor RealCreate; virtual;
    destructor RealDestroy; virtual;
  public
    class function Create: TSLSingle;
    procedure Destroy; virtual;
    procedure Free; virtual;
    procedure RegisterOnChangedEvent(Routine:
      TDataChangedEvent); virtual;
    procedure UnRegisterOnChangedEvent(Routine:
      TDataChangedEvent); virtual;
    { This class function is for demonstration
      purposes only }
    class function Usage: Integer;
  end;
implementation
const
  FUsage: Integer = 0;
  FInstance: TSLSingle = nil;
class function TSLSingle.Create;
begin
  Inc(FUsage);
  If FUsage = 1 then begin
    FInstance := RealCreate;
    ...
  end;
  Result := FInstance
end;
constructor TSLSingle.RealCreate;
begin
  inherited Create;
  Sorted := True;
  Duplicates := dupAccept;
  FOnChangedList := TList.Create;
  LoadFromFile(ExtractFilePath(Application.ExeName)+
    '\TESTDATA.TXT');
end;
procedure TSLSingle.Free;
begin
  Destroy;
end;
procedure TSLSingle.Destroy;
  procedure Error;
  begin
    raise Exception.Create(Format(
     'The Destroy method of the TSLSingle class was '+
     'called using a pointer value'#13#10'of %p. This '+
     'is invalid, it should be %p', [Pointer(Self),
     Pointer(FInstance)]));
  end;
begin
  if Self <> nil then begin
    { Error if there is not currently an instance of
      this object or the pointer passed is invalid }
    if (not Assigned(FInstance)) or
      (Self <> FInstance) then
      Error;
    Dec(FUsage);
    if FUsage = 0 then begin
      RealDestroy;
      FInstance := nil;
      ...
    end;
  end;
end;
destructor TSLSingle.RealDestroy;
begin
  { Put all standard destructor code here }
  inherited Destroy;
end;
end.
```

you would put in a normal `Destroy` method, including the call to the inherited destructor (usually `Destroy`).

Events

Standard classes have event properties (such as `OnChanged`) which can be set to method addresses and are invoked when the event occurs. In our class, there is only one 'real' object instance, but there may be multiple event handlers which need to be called. Therefore, the standard method of storing the address is not sufficient. One way to handle this is to amend the definition for the event write property to call a property access method which would store each address in a list. When the event occurs, the class simply calls each event procedure in turn. However, a problem with this is, if you wish to set the event property to `nil` how does the class know which of the possible event procedures should be removed from the list?

To simplify the situation, and to highlight the difference in approach for this class, I have implemented `RegisterOnChangedEvent` and `UnRegisterOnChangedEvent` methods. The class maintains a list of all event procedures which need to be called when the event occurs. It calls them in the registration order.

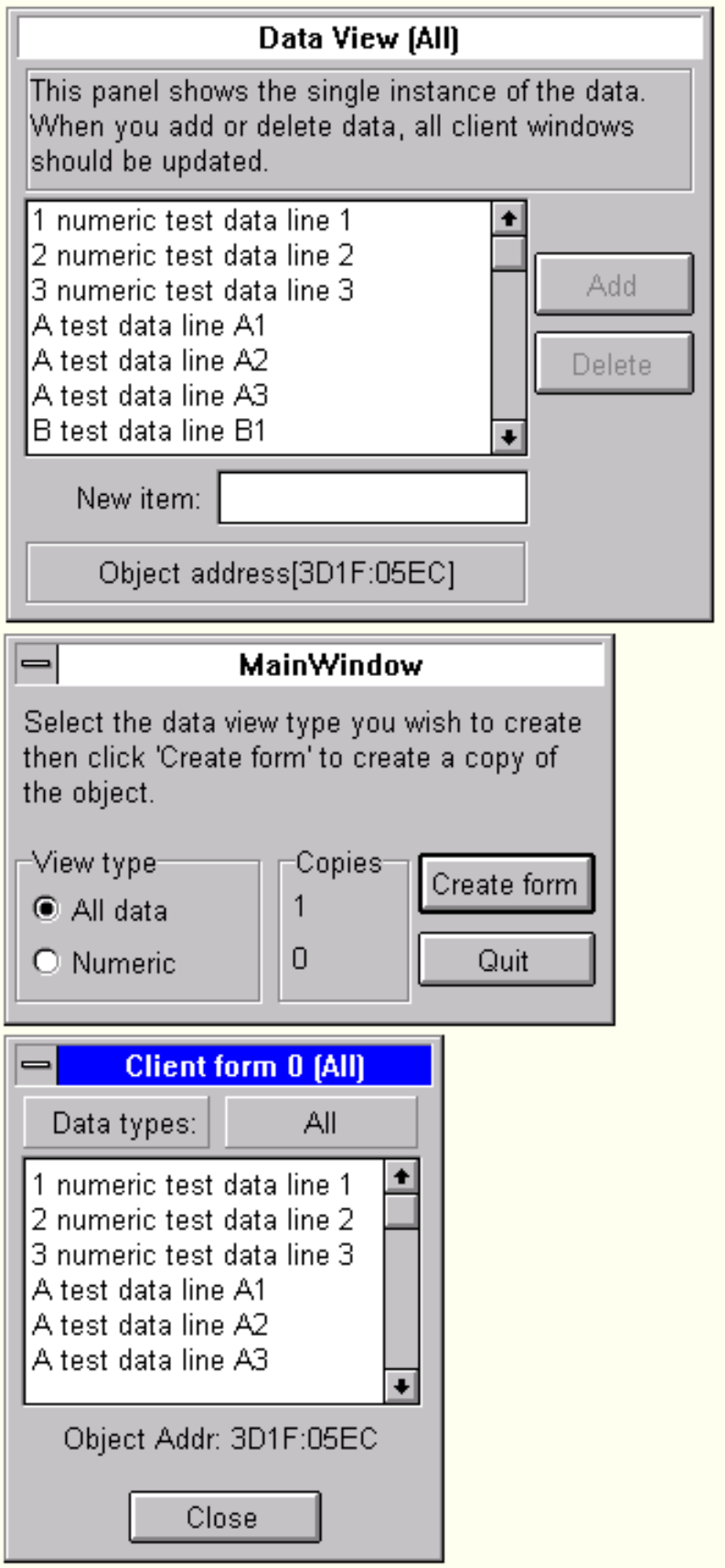
Considerations For Derived Classes

Deriving new classes from a single instance class is not a major problem (refer to `TSLSingleNum` for an example) as long as you consider the following.

Always create a new constructor as a class function and return the new class type.

Call the `RealCreate` constructor when this is the first instance created. In the `RealCreate` constructor, call the inherited `RealCreate` constructor.

Ensure that any usage and instance type variables are different from those used in the parent class. This will ensure that you don't overwrite these values in any application which creates both your new class and its parent class.



➤ *Figure 1*

Provide a `Free` method which simply calls `Destroy`.

Provide a `Destroy` method which calls the `RealDestroy` destructor only when the last instance of the object needs to be freed.

In the `RealDestroy` destructor, call the inherited `RealDestroy`.

Both the `Free` and `Destroy` methods should specify `override` in their definitions to provide polymorphism.

SNGLINST Sample Application

When you compile and execute the sample application (SNGLINST.DPR, see Figure 1) a single window will be displayed with a caption `MainWindow`. To create an instance of either class (`TSLSingle` or `TSLSingleNum`) select the required `View` type radio button then click `Create Form`. This will create a client form with the data type (All/Numeric) displayed in the window caption. This provides a read only view of the data and displays the address of the object.

When the first instance of each class type is created a second window `Data View` is displayed which provides update access to the same data. This form also shows the object address to allow you to ensure that they are referring to the same object. After creating the new instances the number of copies for each type is updated in the main window. In the `Data View` window you can amend the data; if you do so, all associated client forms will be refreshed.

When you close a client form, it destroys the instance and the window is destroyed. The number of copies for each type is updated in the main window. If the last instance is destroyed for the class, the `Data View` window is also destroyed.

Enhancements

As this class stands, there is no protection against multiple updates. This should be OK for Win16 applications but would cause problems for Win32 applications. Protection against concurrent updates needs to be implemented. This is beyond the scope of this article but can be implemented using the Win32 API.

John Chaytor lives in Brighton, England, and can be contacted via CompuServe as 100265,3642