Optimising Form Handling

by Phil Brown

When a new form is designed in Delphi, the IDE very conveniently adds a variable declaration to the unit and makes the form AutoCreate. When the application is started an instance of that form is guaranteed to exist. The project source file handles the creation of these forms, by default adding a line of code calling Application.CreateForm for each form type as shown in Listing 1 below.

While this approach is very convenient (it frees the developer from the considerations of creating and releasing the form in a try..finally block each time a new instance of the form is required) there are a few disadvantages. The first is the memory consumed by each instance of a form: this is allocated once when the application starts and only released when the program terminates. For projects with just a few forms, this is quite acceptable, but for a large application the memory consumed can be significant. Furthermore, simply creating all these forms can have a noticeable impact on the time required to load the application. Finally, because a global variable exists in the unit for each form, careless developers may re-use that variable inappropriately (for example, to hold another instance of the same form). The use of AutoCreate for forms also has an undesirable side-effect: if a different scheme is required to generate forms (for example, if an application requires two instances of the same form, controlled locally within a routine), the application has a mixture of styles for form instance handling: AutoCreate and custom routines.

This eclectic mix leaves other developers not knowing which approach to use for a particular form in an application, a problem compounded by the fact that the form instance global variables in each unit have a nasty habit of lying around in the code, even if AutoCreate is disabled for that form, giving the impression that the variable has been initialised and is available for use when in fact it is not. Again, for multi-developer projects this lack of consistency is disadvantageous.

The standard approach in these cases is to turn AutoCreate off for all forms in the project and create each form locally as it is required. The Project|Options menu option has a tab called Forms which allows AutoCreate to be disabled for selected forms, alternatively the developer can edit the project source file and simply remove the Application.CreateForm entry for those forms where it is not required.

Fortunately Delphi ensures that these two alternatives both reflect any changes made in the other, so you may choose the code-oriented approach or the interactive one depending on personal preference. It is a very good idea to edit the source unit and delete the global form variable for each form that you remove from AutoCreate, this prevents the problems highlighted earlier where developers unwittingly use uninitialised variables (with disastrous results).

This approach does mean that you need to create a new instance of a form each time you want to use it. The standard way to do this is to use a local variable and a try..finally block as shown in Listing 2. Note that the parameter passed to the form constructor is nil, indicating that it has no parent. This is fine, but we must be sure to free up the instance when we have finished with it.

While this approach has the great advantages of minimum memory utilisation (only the forms that are required at any one time are created) and the use of the form is very visible (the frmDetails variable has a very short ‘span’, the entire collection of references to frmDetails occur in one procedure, rather than being scattered throughout the code), the technique is not without disadvantages. In particular, as the form instance is created each time it is

➤ *Listing 1: The project source file for a new application.*

```
program Project1;
uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

➤ *Listing 2:*
Creating a new instance of a form when a button is pressed.

```
implementation
{$R *.DFM}
procedure TfrmFind.btnDetailsClick(Sender: TObject);
var
  frmDetails: TfrmDetails;
begin
  frmDetails := TfrmDetails.Create (nil);
  try
    // initialise form here
    frmDetails.Caption := 'Test';
    // display form
    frmDetails.ShowModal;
  finally
    // release resources
    frmDetails.Free;
  end;
end;
```

used, there can be a short delay while the form is initialised, which can be quite noticeable on low-end hardware if the form is complex (if it has a page control with a few tabs, for example). This delay can give users a feeling of sluggishness or lack of response.

Ideally we would have a scheme in which we could combine the low memory requirements of dynamic form creation with the rapid response of global form variables. This article will show you how this can be done, and provided on the disk is a unit you can use to achieve exactly that.

Caching Forms

An elegant solution is to have a cache of forms that are frequently used and, when a new instance of a form is required, the cache is searched to see if there is one already created that we can use. If so, and it is not already in use, then we can simply return that instance and the elapsed time will be minimal. If an instance of the form is not available, we can create a new one and add this to the cache. When the form is freed we will not actually free the instance but will mark it as expired and available for reuse. In this scenario, although we will still incur a performance penalty when the form is created for the first time, the second and subsequent times that it is accessed will be virtually instantaneous.

The amount of memory consumed by this technique depends on the number of forms that are held in the cache. With a large cache the amount of memory saved will be small, although you will improve application load time, whilst with a small cache the memory savings will be greater, although more time will be spent actually constructing the forms. The beauty of the scheme is that the handling of new instances of all forms can be standardised and the size of cache adjusted to fit a particular application. The technique can also be extended so that the size of the cache can be adjusted dynamically at runtime (perhaps according to the amount of available memory on the host), so that the application minimises memory use on low-end hardware and maximises performance for larger machines.

Given that the cache will have some finite size, consideration must be given to the case when the cache has reached the required capacity. In this situation there will be no room to store the new instance. In these situations a new instance of the form is created on demand, and this instance

```
TCachedForm = class (TForm)
public
  class function CreateForm: TCachedForm;
  procedure FreeForm;
end;
```

➤ *Listing 3: Providing new CreateForm and FreeForm methods to be used as constructors.*

replaces the *oldest* (ie least recently used) form in the cache, which is in turn freed. If the cache contains a full quota of forms which are currently in use, then none of the cache entries are candidates for removal, the new form instance cannot be cached and is simply destroyed when the form is released. Alternative strategies are possible in this situation: for example, it would be possible for our new form to replace the oldest cache entry, which, being still in use, cannot be freed at this time. This approach would work only if the release mechanism for the form was able to be aware of such changes and could free itself if the original entry could not be found in the cache.

Although the above caching requirements sound complex, in actual fact a very simple implementation presents itself. The cache itself can be simply handled as a list of forms, with very simple `request` and `release` methods, to return a form instance of the correct class and to release the same. Some consideration must be given to the syntactical usage of the cache. Ideally, it would be possible to use the cache in a transparent manner, so that the code would be the same as would normally be used (see Listing 2). Unfortunately, this is not possible as to do so would mean that the standard `Create` and `Free` methods would be redirected to our caching routines. Unfortunately, the instance is already partially constructed or destroyed by the time our `Create` or `Free` is called, even if the `Create` is defined as a class function that returns an object, rather than a true constructor, as in Listing 3.

Therefore, it is impossible to reuse the `Create` and `Free` methods names, as they expect to deal with the real construction and destruction of objects, rather than the handling of pre-constructed instances. Seasoned Delphi developers may be familiar with the `NewInstance` and `FreeInstance` methods on `TObject`, which are called by the standard constructors for memory allocation purposes, and are declared `virtual` so that the construction behaviour of the object can be customised. This approach does not prevent the `Destroy` destructor from being called, however, and so therefore cannot be used in this situation where the object instance must be retained.

Instead, we must introduce some new syntax that must be used throughout the application to replace standard calls to `Create` and `Free`. To keep the syntax as close as possible to the original, I have chosen to call these `CreateForm` and `FreeForm`, and they can be used as drop-in replacements for the standard constructors and destructors. Listing 3 shows the interface declaration for a new form, `TCachedForm`.

One caveat to the above syntax is that the class function for `CreateForm` returns a `TCachedForm`. This means that the result of this function cannot be directly assigned to a variable declared as a descendant of `TCachedForm`, you will get a compiler error. There are two approaches to overcoming this: either typecast the result of the function to the required type whenever you use the `CreateForm` method, or statically override the `CreateForm` method in descendant forms which typecast their result appropriately.

Both techniques are demonstrated in Listing 4 and whichever one is chosen is down to personal preference. Of the two, I prefer the second option as it eliminates typecasts and keeps the syntax the same as the standard `Create` constructor. An alternative is not to descend from `TCachedForm` for your new class and simply implement a `CreateForm` and `FreeForm` method, as shown by `TForm3` in Listing 4.

Now that we have some way of centralising form instance handling, we must give some consideration to the routines which will actually create and destroy forms for us. These will be implemented as methods on a new class, `TFormCache`, which will maintain a list of cached forms up to some defined limit. When a call is made to `CreateForm`, it will scan through the list of forms attempting to find an unused instance of the same type. If one is located then this is immediately returned, otherwise one is constructed and added to the cache, provided that it has not grown beyond some definable limit (initially set at 10). The `TFormCache` interface is shown in Listing 5.

➤ *Listing 4: Two ways of handling the creation of cached forms.*

```
type
  TForm1 = class (TCachedForm)
    end;
  TForm2 = class (TCachedForm)
    public
      class function CreateForm: TForm2;
    end;
  TForm3 = class
    public
      class function CreateForm: TForm3;
      procedure FreeForm;
    end;
class function TForm2.CreateForm: TForm2;
begin
  // note static override of CreateForm method
  Result := TForm2 (inherited CreateForm);
end;
class function TForm3.CreateForm: TForm3;
begin
  Result := TForm3 (FormCache.CreateForm (Self));
end;
procedure TForm3.FreeForm;
begin
  FormCache.FreeForm (Self);
end;
procedure TestForms;
var
  Form1: TForm1;
  Form2: TForm2;
begin
  // create Form1 - need to typecast
  Form1 := TForm1 (TForm1.CreateForm);
  try
    Form1.ShowModal;
  finally
    Form1.FreeForm;
  end;
  { create Form2 - no need to typecast but extra method
    required. Creating Form3 would be the same
    syntactically as for Form2 }
  Form2 := TForm.CreateForm;
  try
    Form2.ShowModal;
  finally
    Form2.FreeForm;
  end;
end;
```

Note that the class is very minimalist: only two methods are provided and three properties. Of these, MaximumCacheSize is the most important as it can be written to at runtime to adjust the size of the cache. Note that reducing the maximum cache size does not always take immediate effect, forms that are in use cannot be freed until they are released (any forms in the cache that are not in use can be, however). Whenever forms are created or destroyed, the cache attempts to ensure that the MaximumCacheSize limit is not exceeded. Of course, if the MaximumCacheSize is increased, the cache grows to fill the limit as forms are requested. Applications can monitor memory usage and possibly performance and adjust the cache size dynamically, the Count and ActiveCount properties of the FormCache object indicate the number of current cache entries and the number that are currently in use, respectively.

A FormCaching unit is provided on the disk which you are free to use in your own projects. Note that a FormCache object is automatically constructed and destroyed in the initialisation and finalisation stages, as there should only ever be one instance of the class in an application. Note that when a form is returned from the cache, it will be in the same state as when it was last released, so you will need to re-initialise the form controls appropriately. Generally speaking this is something you will do anyway (for example, if you are displaying a dialog for editing the details of something) and so to facilitate this the OnCreate event is called when a form is returned from the cache. If you always want a form to be in its design-time state when it is displayed, it is probably easiest to use the standard Create and Free methods, this is probably not the type of form that needs to be cached. However, for complex forms it can still be significantly quicker to re-initialise a cached form to design-time values in code than to recreate it from scratch.

In this article the rationale for, and an approach to centralising and optimising form handling has been introduced. For anything more than a trivial application, resource management becomes an important part of the system and using a standard technique throughout the source code increases reliability and offers greater scope for future maintenance.

```
type
  TFormCache = class
  public
    function CreateForm (FormClassToCreate: TFormClass): TForm;
    procedure FreeForm (FormToFree: TForm);
    property MaximumCacheSize: Integer;
    property Count: Integer;
    property ActiveCount: Integer;
  end;
```

➤ *Listing 5: The FormCache class public interface.*

Philip Brown is a senior consultant with Informatica Consultancy & Development, specialising in OO systems design and training. When not orienting objects he enjoys sampling fine wine. Contact him at phil@informatica.uk.com