How to call Delphi code from scripts running in a TWebBrowser (part 2 of 6)

Implementing the external object

Implementing the external object

As already noted we extend the *external* object by creating a COM automation object – i.e. one that implements an interface that derives from *IDispatch*.

An easy way to do this is to derive the new class from *TAutoIntfObject*. This class implements the methods of *IDispatch* and so saves us from having to do this ourselves. However, *TAutoIntfObject* needs a type library to work with. Consequently we will use Delphi's Type Library editor to create a suitable type library that defines our new interface.

Type Library Editor

To create a new type library using the Type Library Editor in Delphi 7, display the *New Items* dialog by selecting the *File* | *New* | *Other* menu option. Then choose *Type Library* from the dialog's *ActiveX* tab.

Once we have defined the required interface in the Type Library Editor we create the type library by pressing Ctrl+S. This will do four things:

1. Create the type library (`.tlb`) file.
2. Include the type library in the program's resources by inserting a suitable `$R` compiler directive in the project's `.dpr` file.
3. Create a Pascal unit containing, amongst other things, the interface definition. The unit will have the same name as the `.tlb` file but will end in `_TLB.pas`.
4. Add a reference to the `_TLB.pas` unit to the project file.

When creating the interface in the Type Library Editor we must abide by the following rules:

1. Ensure the new interface is derived from *IDispatch*.
2. Ensure all methods have a return value of *HRESULT*.
3. Use only automation compatible parameter types.
4. Ensure all *[out]* parameters are pointer types, i.e. they end with * (for example, *BSTR **).
5. Return any values from methods using a parameter that has the *[out,retval]* modifier.

Once we have our new type library and interface we create a new class that descends from *TAutoIntfObject*. Then we implement the methods of our interface in the class. This can be done by copying the method prototypes from the interface declaration in the `*_TLB.pas` file and pasting them into the class's declaration.

Note that Delphi creates the method prototypes using the *safecall* calling convention which means that any *[out,retval]* parameters become function results. For example, suppose we use the Type Library Editor to create an interface called *IMyIntf* that has two methods, *Foo* and *Bar*. Assume the method parameters are defined as in *Table 1*.

Example methods

Method	Parameters	Type	Modifiers
Foo	Param1	long	[in]
	Result	BSTR *	[out,retval]
Bar	Param1	BSTR	[in]

Table 1

The `*_TLB.pas` file created by Delphi would contain the following interface definition shown in *Listing 1*.

```
type
  IMyIntf = interface(IDispatch)
    ['{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}']
    function Foo(Param1: Integer): WideString; safecall;
    procedure Bar(const Param1: WideString); safecall;
  end;
```

Listing 1

We would therefore include the following methods in our class declaration:

```
type
  TMyClass = class(TAutoIntfObject,
    IMyIntf, IDispatch
  )
  private
    ...
  protected
    { IMyIntf methods }
    function Foo(Param1: Integer): WideString; safecall;
    procedure Bar(const Param1: WideString); safecall;
    ...
  end;
```

Listing 2

These methods would then be implemented as required. Remember that we don't declare or implement any methods of *IDispatch* since they are already implemented by *TAutoIntfObject*.

Now *TAutoIntfObject*'s implementation of the *IDispatch* methods depends on having access to the type library that describes the methods of the interfaces implemented by descendent classes. This is achieved by passing an object that implements *ITypeLib* as a parameter to *TAutoIntfObject*'s constructor. It is our job to create such an *ITypeLib* object that "knows about" our type library.

We do this by declaring a parameter-less constructor for the derived class. In the constructor we call the *LoadTypeLib* API function, passing the name of our application as a parameter. *LoadTypeLib* accesses the type library information that is embedded in the application's resources and creates the required *ITypeLib* object based on this information. The object is then passed to the inherited constructor. Assuming our derived class is named *TMyExternal*, *Listing 3* shows the constructor's implementation.

Type Library Resource

Remember the `{$R *.tlb}` directive mentioned earlier ensures that the `.tlb` file generated by the type library editor is included in our program's resources.

```
constructor TMyExternal.Create;
var
  TypeLib: ITypeLib;    // type library information
  ExeName: WideString;  // name of our program's exe file
begin
  // Get name of application
  ExeName := ParamStr(0);
  // Load type library from application's resources
  OleCheck(LoadTypeLib(PWideChar(ExeName), TypeLib));
  // Call inherited constructor
  inherited Create(TypeLib, IMyExternal);
  // ...
  // Do any other initialisation here
  // ...
end;
```

Listing 3

We have now seen how to implement the *external* object. In the *next section* we will examine how to register the object with the web browser control.