구조화 저장소 기법

(Structured Storage Technique)

만약 정해진 포맷의 파일 형식을 써야 하는 것이 아니라, 데이터를 저장할 때 대단히 유연하고도 강력한 방법이 존재한다면 얼마나 편리할까 ?
구조화 저장(structured storage)이라는 새로운 방식으로 이러한 문제를 해결할 수 있다. 구조화 저장은 DocFIle 이나 OLE 복합 파일 (OLE compound file)이라는 이름으로도 불리고 있는 새로운 저장방식이다.
이 방식을 이용하면 Load 와 Save 를 할 때 파일의 일부분만을 활용할 수 있다. 약간의 데이터 블록 만이 필요할 때 전체 데이터를 모두 불러오거나, 저장해야 한다면 이는 상당히 비효율적이라고 말할 수 있다. 이를 이용하면 순차적이면서, 점진적인 데이터의 접근이 가능하다. 기본적으로 윈도우에 의해서 지원되는 방식이므로, 쉽게 정보를 얻을 수 있다.
이미 마이크로소프트에서는 이러한 구조화 저장 방식이 차세대 윈도우 제품에서는 디폴트 파일 포맷으로 사용할 것임을 공언하고 있다. 또한, 현재 MS 오피스 제품군에서는 이 기술을 적용하고 있기도 하다.
이 정도의 소개만으로 구조화 저장의 중요성은 충분히 알고도 남음이 있을 것이다. 그러면, 이제 실제로 이 기술에 대한 설명과 델파이에서 이를 어떻게 구현할 것인지에 대해서 알아보도록 하자.

기본적인 이해

구조화 저장(structured storage)이라는 용어는 DocFile 이라는 용어와 혼용되고 있다. 사용의 편이성을 위해 지금 부터는 간단히 DocFile 이라고 지칭하도록 하겠다. DocFile 을 이해하는 가장 좋은 방법은 파일 내에 파일 시스템을 가지고 있는 것이라고 생각하면 된다. 즉, DocFile 에는 디렉토리와 파일 들을 가지고 있는 것이다.
예를 들어, Example.ole 라는 DocFile 이 있을 때 여기에 Version, Files 라는 디렉토리가 있으며 Files 라는 디렉토리 아래에 File1, File2, File3 와 같은 파일 들을 내부적으로 포함한다고 하자. 이때 File1 과 같은 DocFile 내부의 데이터 블록에 다른 데이터 블록에 전혀 영향을 주지 않고 접근하는 것이 가능하다.
참고로 앞으로 사용하는 용어 중에서 Storage 라는 용어는 DocFile 에서의 디렉토리와 동격으로 생각하면 되고, Stream 은 DocFile 에서의 파일로 생각하면 된다.

DocFile 생성 함수

DocFile 을 생성하는 함수는 StgCreateDocFile 이다. 이 함수는 델파이 3 의 activex.pas 유닛에 선언되어 있으며 선언부분은 다음과 같다.

```
function StgCreateDocfile(pwcsName: POleStr; grfMode: Longint;
  reserved: Longint; out stgOpen: IStorage): HResult; stdcall;
```

첫 번째 파라미터인 pwcsName 은 파일이름을 유니코드로(OLE 에서는 유니코드가 표준으로 사용된다.) 설정하면 되고, grfMode 에는 플래그를 설정하게 된다. 세 번째 파라미터는 현재는 사용되지 않기 때문에 보통 0 으로 설정하게 되며, 실제 사용하게될 IStorage 인터페이스가 네 번째 파라미터에서 넘어오게 된다.
하나의 DocFile 은 그 자체가 Storage 이다. 그렇기 때문에, 이 함수에 의해서 넘어오는 Storage 는 파일의 루트 저장소가 된다. 이 함수가 성공적으로 수행되었는지 여부를 검사할 때에는 SUCCEEDED() 함수를 사용한다. 그 Pseudo Code 를 아래에 들어 보았다.

```
Hr := StgCreateDocFile(...);
if (SUCCEEDED(Hr)) then ...;
```

보통 OLE 를 사용할 때 HResult 를 S_OK 와 비교하는 경우가 많은데, 이 방법은 그다지 좋은 방법이 못된다. 그 이유는 OLE 가 함수가 성공적으로 수행되더라도 미묘하게 차이가 나는 다른 여러가지 반환값을 가질 수 있기 때문이다. 그러므로, SUCCEEDED() 함수를 사용하는 것이 보다 효율적인 방안이 된다.

유니코드(UniCode)의 사용

앞의 함수 선언부분에서 언급했듯이 OLE 세계에서는 유니코드가 표준으로 사용된다. 그렇지만 지금까지의 프로그래밍 환경에서는 안시코드(AnsiCode)를 표준으로 사용해 왔기 때문에 다소간의 혼란이 있을 수 있다.
델파이 3 에서는 유니코드와 안시코드를 쉽게 변환할 수 있는 방법을 제공하기 때문에 이런 변화가 커다란 문제가 되지 않는다. WideString 문자열 데이터 형이 유니코드를 지원하게 된다. 그러면, 간단히 안시코드와 유니코드를 변환하는 몇 가지 방법에 대해 알아보도록 하자.

```
var
  s: string;
  ws: WideString;
begin
```

```
  s := ‘abc’;
  ws := s;
end;
```

어떤가 ? 너무나 단순하지 않은가 ? 그냥 일반 문자열을 위에서와 같이 WideString 형의 대입하는 것으로 모든 것이 끝난다. 마찬가지로 유니코드 문자열을 안시코드로 변환할 때에도 단순히 다음과 같이 하면 된다.

```
var
  s: string;
  ws: WideString;
begin
  ws := ‘abc’;
  s := ws;
end;
```

그렇지만, OLE 함수를 사용할 때에는 보통 WideString 데이터 형 보다는 PWideChar 데이터 형을 파라미터로 사용하기 때문에 이를 호출할 때에는 아래와 같은 방식으로 형변환시켜 사용하면 간단히 해결된다.

```
var
  ws: WideString;
begin
  ws := ‘abc’;
  SomeOLEFunction(PWideChar(ws));
end;
```

델파이 2 에서는 WideString 데이터 형을 지원하지 않고, PWideChar 데이터 형만을 지원하기 때문에 이를 안시코드 문자열과 호환시키기 위해서는 해당하는 문자열의 크기의 두배 만큼의 메모리를 할당받고, 실제로 문자를 유니코드로 바꾸기 위해서는 API 함수인 MultiByteToWideChar 함수를 호출해야 했다. 마찬가지로 유니코드 문자열을 안시코드로 변환시킬 때에도 메모리 할당과 WideCharToMultiByte API 함수를 사용해야 한다.
이 점이 델파이 4 가 얼마나 OLE/COM 환경에 적합한 형태로 바뀌었는지를 보여주는 단적이 예가 될 수 있다.

Stream, Storage 이름의 제한

DocFile 시스템에서도 약간의 이름에 대한 제한을 가지고 있다. 31 자가 넘는 이름을 가질 수는 없으며, 이름에 '!', ':', '/', '₩' 등의 문자는 사용할 수 없다. 그리고 첫번째 문자는 ordinal 값이 32 이하인 문자가 되면 안된다. 이러한 문자들은 특수한 목적에 사용되게 된다.

STGM 상수

STGM 상수는 storage, stream 인터페이스에서 객체에 대한 접근 모드나 객체를 실제로 생성, 삭제 등을 하게 되는 조건을 정의하고 있다. 이들 상수에 대해서 알아보도록 하자.

- STGM_READ, STGM_WRITE, STGM_READWRITE

Stream 객체에 대해서는 어떤 메소드를 허용할 것인지를 결정하는 상수이다. 예를 들어, STGM_READ 는 IStream 의 Read 메소드를 허용하게 된다. Storage 객체에 대해서는 가능한 요소를 나열하고, 이들을 open 한다. STGM_WRITE 는 객체를 저장할 수 있도록 하며, STGM_READWRITE 는 STGM_READ 와 STGM_WRITE 를 혼합한 것이다.

- STGM_SHARE_DENY_NONE, STGM_SHARE_DENY_READ,
  STGM_SHARE_DENY_WRITE, STGM_SHARE_EXCLUSIVE

STGM_SHARE_DENY_NONE 은 어떤 객체를 open 하더라도 이것이 그 객체에 대한 read, write 접근에 대한 제한을 가지지 않는 것을 의미한다.
STGM_SHARE_DENY_READ 는 open 한 객체에 대해서 STGM_READ 모드로 접근할 수 없도록 제한다. 주로 root storage 객체에 대해 사용된다. STGM_SHARE_DENY_WRITE 는 STGM_WRITE 모드로 접근할 수 없도록 제한하는데, 이것은 여러 명의 사용자가 객체에 접근했을 때 생길 수 있는 문제점을 해결할 수 있다.
STGM_SHARE_EXCLUSIVE 는 STGM_READ, STGM_WRITE 모드 둘 다 접근할 수 없도록 하는 값이다.

- STGM_DIRECT, STGM_TRANSACTED

Direct 모드에서는 storage 요소에 대해서 변화가 일어날 경우 이 값이 그대로 반영된다. 이 모드가 디폴트로 되어 있다. Transacted 모드에서는 변화가 일어날 경우 그 값이 버퍼에 저장되었다가 commit 이 호출될 때 객체에 반영된다. 만약 IStream, IStorage 인터페

이스에서 Revert 메소드가 호출되면 이러한 변화가 무시된다. 그러나, 이 모드는 현재 OLE 에서는 구현되지 않고 있다. 아마도 조만간에는 이것이 지원될 것으로 생각된다.

- STGM_CREATE, STGM_CONVERT, STGM_FAILIFTHERE

STGM_CREATE 는 현재 존재하는 storage, stream 객체가 새로운 객체가 생성될 때에는 반드시 제거되어야 한다는 것을 지정한다. 만약 현재의 객체가 성공적으로 제거되지 않으면 새로운 객체가 생성되지 않는다.
STGM_CONVERT 플래그는 현재 존재하는 stream 의 데이터를 보존하면서 새로운 객체를 생성하는데, 이때 이전 객체의 데이터는 CONTENTS 라는 객체에 보존된다. 이때 과거의 storage 객체에 있던 정보는 stream 의 형태로 변경되어 보존되므로, storage 의 계층 구조 정보는 망실된다.
STGM_CONVERT 플래그는 디스크에 storage 객체를 생성하려고 하는데, 이미 그런 파일 이름이 존재하거나, Storage 객체 내부에 새로운 stream 을 생성하려고 하는데 같은 이름의 stream 이 있을 경우 등에서 사용하게 된다.

- STGM_FAILIFTHERE

이 플래그는 만약 지정된 이름의 객체가 있을 경우에 생성 과정을 취소하는 역할을 해준다. 이 경우에 STG_E_FILEALREADYEXISTS 상수가 반환된다.

- STGM_PRIORITY

이 플래그가 지정되면 현재 우선권을 가진 사용자 만이 객체의 변화를 줄 수 있다. 이를 이용하면 다른 사용자들은 이 객체에 접근해도 이를 변화시킬 수 없게 된다. 이 경우에는 반드시 STGM_DIRECT, STGM_READ 가 설정되어 있어야 한다.

- STGM_DELETEONRELEASE

이 플래그는 임시 파일을 사용할 때 유용하게 쓰이는 것으로, 부모 storage 객체가 해제되면 자동으로 그 아래의 파일 들이 파괴되도록 지정하는 것이다.

DocFile 을 만들어 보자.

앞에서 설명한 StgCreateDocFile 함수를 사용해서 실제로 DocFile 을 만들어 보기로 하자. 이미 함수 선언과 파라미터에 대해서 간단한 설명을 했지만, 다시 한번 정리해 보자.

함수의 선언부는 다음과 같다.

```
function StgCreateDocfile(pwcsName: POleStr; grfMode: Longint;
  reserved: Longint; out stgOpen: IStorage): HResult; stdcall;
```

그리고, 각 파라미터에는 다음과 같은 내용들을 설정하게 된다.

1. pwcsName: 유니코드 형식의 실제 파일 명. (예) 'c:₩temp₩example.ole'
2. grfMode: STGM 플래그가 설정된다. (예) STGM_CREATE or STGM_READWRITE STGM_DIRECT or STGM_SHARE_EXCLUSIVE
3. reserved: 0
4. stgOpen: 실제로 storage 를 담게 될 레퍼런스 파라미터

그럼, 이제 실제 DocFile 을 만드는 프로시저를 하나 만들어 보자.

```
procedure Create;
var
  Hr: HResult;
  Root: IStorage;
begin
  Hr := StgCreateDocFile('c:₩Temp₩Example1.ole', STGM_CREATE or STGM_READWRITE or
        STGM_DIRECT or STGM_SHARE_EXCLUSIVE, 0, Root);
  if (SUCCEEDED(HR)) then begin end else begin end;
end;
```

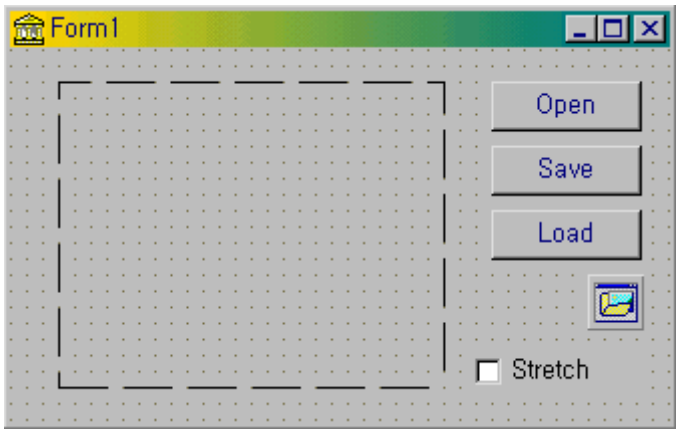
아무 것도 하지 않고, Root 라는 IStorage 인터페이스만 받아오는 프로시저가 완성되었다. 사용법이 그다지 어렵지는 않다는 것을 쉽게 알 수 있을 것이다.
마찬가지로 DocFile 을 여는 것도 그다지 어렵지 않다. 이때에는 StgOpenStorage 라는 API 함수를 사용하는데, 이는 DocFile 자체가 root storage 이기 때문이다. 이 API 의 사용법도 거의 유사하므로 여기에서 간단히 소개하겠다.

```
procedure OpenDocFile;
var
  Hr: HResult;
  Root: IStorage;
begin
```

```
  Hr := StgOpenStorage('c:₩Temp₩Example1.ole', nil, STGM_READWRITE or
        STGM_DIRECT or STGM_SHARE_EXCLUSIVE, nil, 0, Root);
  if (SUCCEEDED(HR)) then begin end else begin end;
end;
```

DocFile 기법을 사용한 최초의 어플리케이션

그럼 이제, DocFile 을 이용해서 그림을 저장하고 불러올 수 있는 어플리케이션을 하나 만들어 보기로 하자. 새로운 어플리케이션을 하나 시작하고, 다음 그림과 같이 폼위에 버튼 3 개와 이미지 컴포넌트, 체크 박스, TOpenPictureDialog 대화상자 컴포넌트를 하나씩 얹어서 디자인하도록 하자.



이때 각 버튼의 캡션을 Open, Save, Load 로 정하고, 체크 박스에는 Stretch 라고 캡션을 정하도록 한다. 이제 간단하게 이 어플리케이션에 대해서 설명하면, 구조화 저장 방법을 이용하는 방법을 익히기 위한 어플리케이션으로 Open 버튼을 누르면 대화상자를 띄워서, 아무 그림 파일이나 선택하게 하고, 이를 이미지 컴포넌트에 보여준다. 이때 'Stretch' 체크 박스가 체크되어 있을 경우에는 이미지를 Stretch 해서 보여준다. 그리고, 보여주는 이미지를 'c:₩temp₩Exam1.ole' 파일에 저장할 때에는 Save 버튼을 누르고, 저장된 이미지를 불러올 때에는 Load 버튼을 누르게 한다.

유닛의 implementation 섹션에 우리가 사용하게 될 activex.pas 와 AxCtrls.pas 유닛을 uses 문장에 추가한다.

먼저 체크 박스의 OnClick 이벤트 핸들러를 아래와 같이 작성해서, 이미지 컴포넌트의 Stretch 속성에 반영할 수 있도록 하자.

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
```

```
  Image1.Stretch := CheckBox1.Checked;
end;
```

그리고, Open 버튼의 OnClick 이벤트 핸들러를 작성하자.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenPictureDialog1.Execute then
    Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
end;
```

이제 실제로 Save 버튼을 클릭하면 그림이 DocFile 로 저장되도록 해야 한다. 이렇게 하려면 IStorage 에 IStream 스트림을 하나 생성하고 이 스트림에 내용을 기록해야 한다. 스트림을 생성하는 코드는 IStorage 인터페이스의 CreateStream 메소드를 사용한다. 이 메소드의 파라미터로 첫번째 파라미터에 스트림의 이름과 두번째에 STGM 플래그를 설정하고, 마지막 파라미터로 생성된 IStream 인터페이스가 저장될 변수를 지정한다. 즉, 다음과 같은 코드를 사용한다.

```
Hr := Root.CreateStream('ExamStream', STGM_CREATE or STGM_READWRITE or
  STGM_DIRECT or STGM_SHARE_EXCLUSIVE), 0, 0, Stream);
```

그러면, 이렇게 Stream 이라는 변수에 IStream 인터페이스를 담아오게 되면 실제 데이터를 여기에 저장해야 한다. 저장하는 방법은 크게 두가지가 있는데, 첫번째 방법은 IStream 의 메소드를 직접 이용하는 것이고, 두번째 방법은 AxCtrls.pas 유닛에서 제공하는 TOleStream 클래스를 이용하는 것이다.
델파이 3 에서는 TOleStream 클래스를 이용해서 이 작업을 아주 쉽게 할 수가 있다. 사용법은 아래와 같이 아주 간단하다.

```
OleStream := TOleStream.Create(Stream);
Image1.Picture.SaveToStream(OleStream);
OleStream.Free;
```

이제 Save 버튼의 OnClick 이벤트 핸들러를 제작해 보자.

```
procedure TForm1.Button2Click(Sender: TObject);
var
```

```
    Hr: HResult;
    Stream: IStream;
    OleStream: TOleStream;
    Root: IStorage;
begin
  Hr := StgCreateDocFile(   'c:₩Temp₩Example1.ole', STGM_CREATE or STGM_READWRITE or
    STGM_DIRECT or STGM_SHARE_EXCLUSIVE, 0, Root);
  if (not SUCCEEDED(Hr)) then Exit;
  Hr := Root.CreateStream('ExampleStream', STGM_CREATE or STGM_READWRITE or
    STGM_DIRECT or STGM_SHARE_EXCLUSIVE, 0, 0, Stream);
  if (not SUCCEEDED(Hr)) then Exit;
  OleStream := TOleStream.Create(Stream);
  Image1.Picture.Graphic.SaveToStream(OleStream);
  OleStream.Free;
end;
```

이제는 저장된 이미지를 스트림을 통해서 읽어올 차례이다. 읽을 때에도 쓸 때와 마찬가지로 IStream 의 메소드를 직접 이용하는 방법과 TOleStream 클래스의 메소드를 사용하는 방법이 있다. 먼저 IStream 의 Read 메소드를 사용하는 방법에 대해 알아보자. 이 메소드는 파라미터를 3 개 사용한다. 첫번째 파라미터에는 데이터를 저장할 버퍼를, 두번째 파라미터에는 읽어올 데이터의 크기(바이트), 세번째 파라미터에는 실제로 읽어들인 데이터의 크기가 넘어온다.
그러므로, 데이터를 읽어 들이기에 앞서 읽어올 데이터의 크기를 알아야 한다. 보통 스트림에 지금과 같이 하나의 데이터를 저장한 경우에는 스트림의 크기가 읽어올 데이터의 크기가 된다. 그러면 스트림의 크기를 알아볼 수 있는 함수에 대해서 알아보자

```
function GetStreamSize(Stream: IStream): LongInt;
var
  Hr: HResult;
  StatStg: TStatStg;
begin
  Hr := Stream.Stat(StatStg, STATFLAG_NONAME);
  if (not SUCCEEDED(Hr)) then
  begin
    Result := -1;
    Exit;
```

```
  end;
  Result := Round(StatStg.cbSize);
end;
```

IStream 인터페이스의 Stat 메소드를 이용하면 현재의 스트림에 대한 정보를 TStatStg 클래스에 담아 준다. 위에서 Stat 메소드에 대한 파라미터로 사용한 STATFLAG_NONAME 플래그는 스트림의 이름은 담아오지 말라고 지정한 것인데, 사실 이 경우에는 이름을 사용할 이유가 없기 때문에 이 플래그를 지정함으로써 쓸데 없는 메모리의 낭비를 막을 수가 있다. TStatStg 클래스의 cbSize 멤버에 스트림의 크기가 저장되어 있으므로 이를 정수형으로 바꾸어 그 값을 반환한다.
그리고 나서 IStream 인터페이스를 담고 있는 Stream 이라는 변수가 있다고 하면 아래와 같이 사용하면 된다.

```
Stream.Read(pBuffer, GetStreamSize(Stream), ReadBytes);
```

그러나, 보통의 경우에는 TOleStream 을 이용하는 것이 훨씬 편리하다. 사용법도 데이터를 쓸데와 거의 유사하므로, 아래의 코드를 살펴보면 금방 이해할 수 있을 것이다.
그러면 Load 버튼의 OnClick 이벤트 핸들러를 다음과 같이 작성하자.

```
procedure TForm1.Button3Click(Sender: TObject);
var
  Hr: HResult;
  Stream: IStream;
  OleStream: TOleStream;
  Root: IStorage;
begin
  Hr := StgOpenStorage('c:₩Temp₩Example1.ole', nil, STGM_READWRITE or
    STGM_DIRECT or STGM_SHARE_EXCLUSIVE, nil, 0, Root);
  if (not SUCCEEDED(Hr)) then Exit;
  Hr := Root.OpenStream('ExampleStream', nil, STGM_READWRITE or
    STGM_DIRECT or STGM_SHARE_EXCLUSIVE, 0, Stream);
  if (not SUCCEEDED(Hr)) then Exit;
  OleStream := TOleStream.Create(Stream);
  if (OleStream.Size > 0) then Image1.Picture.Graphic.LoadFromStream(OleStream);
  OleStream.Free;
end;
```

자 이제 첫번째 구조화 저장기법을 이용한 어플리케이션이 완성되었다. 아직 기능이 많은 것은 아니지만 기본적인 테크닉을 익히는 데에는 유용했을 것으로 생각한다. 그럼 이제 더 기능이 많은 두번째 어플리케이션을 제작해 보도록 하자.

파일 뷰어의 제작

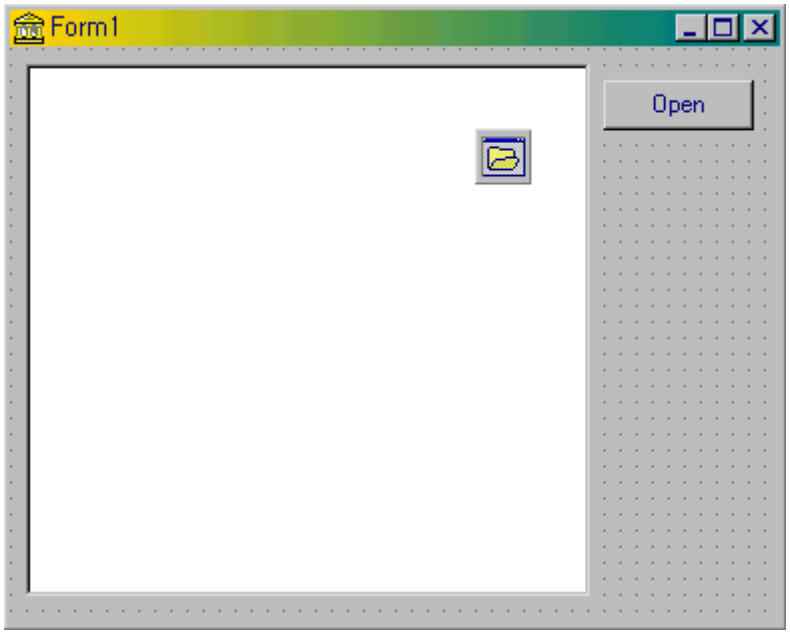
이번에는 DocFile 의 내부를 들여다 볼 수 있는 뷰어를 제작해 보자. 이를 구현하기 위해서 IStorage 인터페이스의 EnumElements 메소드를 사용하게 되는데 이 메소드의 속도가 다소 느린 것이 흠이다.
EnumElements 메소드의 선언부는 다음과 같다.

```
function EnumElements(reserved1: Longint; reserved2: Pointer; reserved3: Longint;
  out enm: IEnumStatStg): HResult; stdcall;
```

즉, 마지막 파라미터에 적절한 IEnumStatStg 인터페이스 형의 변수를 넣어주면 여기에 정보를 담아서 오게 된다. 이렇게 일단 IEnumStatStg 인터페이스를 받아오면 이 인터페이스의 Next 메소드를 사용해서 각각의 하부 요소 들을 얻을 수 있게 된다. Next 메소드는 아래와 같이 선언되어 있다.

```
function Next(celt: Longint; out elt; pceltFetched: PLongint): HResult; stdcall;
```

첫번째 파라미터에는 받아올 아이템의 수, 두번째 파라미터에는 실제로 받아오게 될 TStatStg 클래스 형의 변수를 지정하고, 세번째 파라미터에는 실제로 넘어온 아이템의 수가 반환된다.
이 함수들을 이용해서 워드나 엑셀 등의 내부적인 저장이 어떤 식으로 되어있는지 들여다 볼 수 있는 간단한 뷰어를 제작해보자.



먼저 앞의 그림과 같이 새로운 어플리케이션을 시작하고 폼에 트리뷰 컨트롤 하나와 TOpenDialog 대화상자 하나, 그리고 버튼을 하나 올려 놓자. 버튼의 캡션을 'Open' 으로 설정한다.

그리고, uses 절에 Activex.pas, ComObj.pas 유닛을 추가하고, 버튼의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

일단 파일 이름을 유니 코드 형식으로 해야 하기 때문에, WideString 형식으로 선언한 변수에 파일 이름을 집어 넣고, 이를 PWideChar 로 형 변환해서 사용한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ws: WideString;
  Hr: HResult;
  Root: IStorage;
begin
  if not OpenDialog1.Execute then Exit;
  TreeView1.Items.Clear;
  ws := OpenDialog1.FileName;
```

이제는 이 파일에 실제로 DocFile 형식인지 검사해서 그렇다면 그 파일의 정보를 나타내야 한다. 이를 위해서 StgIsStorageFile 이라는 함수를 사용하는데, 이 함수의 파라미터로 유니코드 형식의 파일 이름을 넘겨 주게 되고 만약 DocFile 형식이라면 S_OK 가 반환된다.

```
if (StgIsStorageFile(PWideChar(ws)) <> S_OK) then
begin
  ShowMessage('DocFile 형식이 아닙니다.');
  Exit;
end;
```

지정된 파일이 DocFile 형식이므로 StgOpenStorage 함수를 사용해서 DocFile 을 열고 루트를 앞에서 선언한 Root 라는 IStorage 형의 변수에 담아온다.

```
Hr := StgOpenStorage(PWideChar(ws), nil, STGM_READWRITE or STGM_DIRECT or
        STGM_SHARE_EXCLUSIVE, nil, 0, Root);
if not SUCCEEDED(Hr) then
begin
  ShowMessage('파일을 열 수 없습니다 !');
  Exit;
end;
```

성공적으로 파일을 열고, Root 를 받아왔으면 파일의 이름을 트리뷰 컴포넌트의 루트로 추가한다.

```
TreeView1.Items.Add(nil, ws);
```

이제 Enumeration 을 이용해서 트리 뷰에 하부 요소의 이름 들을 추가해 나가자. DocFile 의 구조는 Storage 와 Stream 으로 이루어진 여러 단계의 디렉토리 형태를 가지고 있기 때문에 이를 모두 탐색할 수 있도록 하려면 재귀적 호출을 할 수 있어야 하므로 독자적인 프로시저를 하나 정의해야 한다.
이 프로시저를 Enumeration 이라고 정의하고, 파라미터로 IStorage 인터페이스와 트리 뷰의 해당 아이템을 지정하도록 하자. 그러면 아래와 같이 선언될 것이다.

```
procedure TForm1.Enumeration(Storage: IStorage; ANode: TTreeNode);
```

일단 Button1 의 OnClick 이벤트 핸들러에서는 Root 와 트리 뷰의 첫번 째 노드를 파라미터로 해서 이 프로시저를 호출하고, 트리 뷰를 펼쳐 보이면 모든 작업이 끝난다.

```
Enumeration(Root, TreeView1.Items[0]);
TreeView1.FullExpand;
```

```
end;
```

남은 작업은 Enumeration 프로시저를 구현하는 것이다. 일단 다음과 같이 프로시저와 사용할 변수를 선언하고, IEnumStatStg 인터페이스를 Enum 변수에 담아 온다.

```
procedure TForm1.Enumeration(Storage: IStorage; ANode: TTreeNode);
var
  Hr: HResult;
  Enum: IEnumStatStg;
  SubNode: TTreeNode;
  StatStg: TStatStg;
  SubStor: IStorage;
  HrSubStor: HResult;
  NumFetched: integer;
begin
  Hr := Storage.EnumElements(0, nil, 0, Enum);
  OleCheck(Hr);
```

OleCheck 프로시저는 ComObj.pas 유닛에 선언되어 있는데 결과 코드가 에러에 해당되면 EOleSysError 예외를 발생시킨다.
EnumElements 메소드를 이용해서 IEnumStatStg 인터페이스를 Enum 변수에 담아 왔으면 이 인터페이스의 Next 메소드를 사용해서 TStatStg 클래스 형으로 선언된 StatStg 변수에 정보를 담아 오도록 하자. 이때 Next 메소드의 마지막 메소드에는 실제로 넘어온 아이템의 수가 정수의 포인터 형으로 반환되므로 다음과 같이 사용한다.

```
repeat
  Hr := Enum.Next(1, StatStg, @NumFetched);
  if Hr <> S_OK then continue;
```

이제는 StatStg 변수의 dwType 정보에 따라서 다르게 대응해야 한다. 하부 요소가 Storage 라면 일단 Storage 의 이름이 담겨 있는 pwcsName 필드를 이용해서 트리뷰 컴포넌트에 노드를 하나 추가한다. 그리고 나서, Storage 를 다시 열어서 그 Storage 의 SubStorage 를 얻고, 이 SubStorage 에 해당되는 아이템과 추가할 트리 노드를 가지고 Enumeration 프로시저를 재귀 호출한다.

```
  case StatStg.dwType of
```

```
    STGTY_STORAGE:
    begin
      SubNode := TreeView1.Items.AddChild(ANode, StatStg.pwcsName);
      HrSubStor := Storage.OpenStorage(StatStg.pwcsName, nil, STGM_READWRITE
          or STGM_DIRECT or STGM_SHARE_EXCLUSIVE, nil, 0, SubStor);
      if SUCCEEDED(HrSubStor) then Enumeration(SubStor, SubNode);
    end;
```

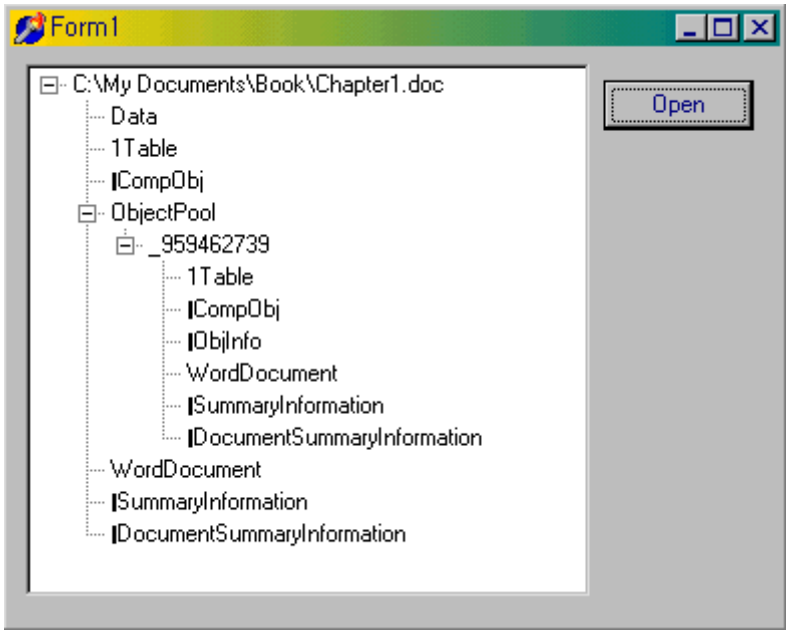
하부 요소가 Stream 이라면 스트림의 이름을 트리뷰에 추가하기만 하면 된다.

```
    STGTY_STREAM:
      TreeView1.Items.AddChild(ANode, StatStg.pwcsName);
  end;
```

이를 계속 반복하다가 마지막에 가면 S_OK 가 반환되지 않으므로, 이때 루프를 종료하면 된다.

```
  until (Hr <> S_OK);
end;
```

이것으로 두번째 어플리케이션이 완성되었다. 이제 실제로 이를 컴파일해서 실행하고, MS 오피스 파일 등을 열어서 하나의 파일에 어떤 형식으로 내용들이 구조적으로 저장되어 있는지 살펴보도록 하자. 아래의 그림은 오피스의 CommonDB.xls 파일을 열어본 것이다. 이를 보면 실제로 저장되는 내용이 하나의 파일에 여러 개의 Stream 과 Storage 로 이루어져 있다는 것을 알 수 있다.



정 리 (Summary)

이번 장에서는 새로운 파일 저장 방식으로 사용되고 있는 구조화 저장소 기법에 대해서 알아보았다. 이와 같은 구조화 저장소 기법을 이용해서 파일을 저장하면 파일 하나에 여러 내용을 분류하여 관리할 수 있다.

구조화 저장소 기법을 이용한 파일 저장 방식 역시 이제는 표준으로 정착되어 가고 있다. 그러므로 Stream 과 Storage 를 이용하는 방법에 대해서는 잘 익혀두는 것이 도움이 될 것이다.