

OLE 도큐먼트와 OLE 자동화 컨트롤러

(OLE Document and OLE Automation Controller)

마이크로소프트에서 처음 OLE 를 발표했을 때 OLE 는 DDE(Dynamic Data Exchange) 모델의 확장판이었다. 즉, 클립보드를 사용하여 데이터를 복사하고 DDE 를 이용하여 두 문서를 연결할 수 있었던 것처럼, OLE 를 이용하여 데이터를 서버 어플리케이션에서 클라이언트 어플리케이션 쪽으로 서버 관련 정보 또는 윈도우 레지스트리에 저장된 정보를 레퍼런스와 함께 복사할 수 있었다.

그러나, 그 이후 발표된 OLE2 는 과거의 OLE 와 같이 복합 문서를 작성하기 위한 도구로서의 역할을 하는 것이 아니라, COM 이라는 기반 기술을 가리키는 단어가 되었다.

여기에서 과거의 OLE 에 해당되는 내용을 OLE 도큐먼트 또는 액티브 도큐먼트라고 하며, OLE2 를 필두로한 최근의 핵심기술은 DCOM/ActiveX 라는 단어로 표현하고 있다.

이번 장에서는 OLE 도큐먼트를 다루기 위한 TOleContainer 클래스의 사용법과 OLE 자동화 컨트롤러를 작성하는 방법에 대해서 알아볼 것이다.

OLE 도큐먼트와 TOleContainer 컴포넌트

OLE 도큐먼트는 임베딩(embedding)과 연결(linkning)의 2 가지 기능을 가지고 있다.

OLE 도큐먼트에서 객체를 삽입하는 임베딩은 클립보드를 이용해서 객체를 삽입하는 것과 큰 차이는 없으나, 서버 어플리케이션에서 OLE 객체를 복사해서 이를 컨테이너 어플리케이션에 붙일 때 그 데이터와 서버에 대한 정보(GUID)를 같이 복사하게 된다는 것이다. 여기에 비해 객체를 연결하는 것은 서버에 대한 데이터와 정보의 레퍼런스만을 복사하는 것이다. 그러므로, 객체를 임베딩한 후 이 객체를 수정하면 이것은 독자적인 내용을 수정하는 것이 되지만, 연결된 객체를 수정하면 실제로는 별도의 파일에 있는 원래의 데이터를 수정하는 것이 된다. 또한, 임베딩된 객체의 데이터는 컨테이너 어플리케이션에 의해 저장되고 관리된다. 반면에 연결된 객체는 서버에 의해 독립적으로 다루어지는 물리적으로 별도의 파일 안에 저장됩니다.

어플리케이션에 OLE 2.0 서버로부터 객체를 삽입한 경우에는 컨테이너는 비주얼(in-place) 편집을 지원하는데, 이것은 컨테이너의 메인 윈도우 안에서 그 객체를 수정할 수 있게 해준다. 서버와 컨테이너의 어플리케이션 윈도우, 메뉴, 툴바가 자동으로 병합되고 이를 이용하여 사용자는 하나의 컨테이너 어플리케이션에서 OLE 서버를 지원하는 여러 종류의 객체를 작업할 수 있게 된다.

- TOleContainer 컴포넌트 사용하기

TOleContainer 컴포넌트는 델파이 어플리케이션에 OLE 도큐먼트 컨테이너를 쉽게 구현하도록 지원하는 컴포넌트이다. 사용자가 어플리케이션에 객체를 삽입할 수 있도록 지원하려면 InsertObjectDialog 메소드를 호출할 수도 있고, OLE 객체를 TOleContainer 에 연결시킬 때 CreateObject, CreateLinkToFile 메소드를 이용해서 객체를 생성하거나 연결할 수 있다.

TOleContainer 는 자동으로 메뉴 병합을 지원하기 때문에, 데이터를 in-place 활성화 시키면 컨테이너 폼의 메뉴가 해당 OLE 객체 서버 어플리케이션의 메뉴와 합쳐진다. 마찬가지로, in-place 활성화되는 OLE 서버의 툴바 역시 컨테이너 어플리케이션 윈도우에 나타난다. 이때 툴바로 사용되는 패널에 나타나는데, 이를 막기 위해서는 패널의 Locked 프로퍼티를 True 로 설정해주면 된다.

TOleContainer 클래스의 주요한 메소드와 프로퍼티를 소개하면 다음과 같은 것들이 있다.

이름	설명
AllowInPlace	In-place 활성화를 지원하는지 여부
AllowActiveDoc	이 값이 True 이면 OLE 컨테이너의 팝업 메뉴에 OLE 객체의 Verbs 를 포함하게 된다.
AutoActivate	컨테이너 안의 객체를 활성화하는 방법을 지정한다. aaManual, aaGetFocus, aaDoubleClick 등의 값이 있다.
Align	In-place 활성화를 제대로 지원하려면 alClient 로 지정하는 것이 좋다.
CanPaste	클립보드의 데이터를 임베딩 객체로 붙여넣을 수 있는지 지정한다.
CopyOnSave	이 값이 True 이면 OLE 객체를 기록할 때 파일을 이용하며, 중복된 데이터를 압축하므로 공간을 절약할 수 있다.
CreateLinkToFile	OLE 객체의 연결을 물리적 파일로 생성한다.
CreateObject	OLE 컨테이너에 새로운 임베딩 객체를 생성한다.
CreateObjectFromFile	임베딩 객체를 지정된 파일에서 생성한다.
CreateObjectFromInfo	TCreateInfo 레코드의 스펙에 기초하여 객체를 생성한다.
DestroyObject	객체와 변경된 사항을 제거한다.
DoVerb	OLE 객체가 특정 행동을 수행할 것을 요구한다.
Iconic	이 값이 True 이면 아이콘이 컨테이너에 표시되며, False 이면 객체의 데이터가 표시된다.
InsertObjectDialog	운영체제에서 제공하는 OLE 삽입을 지원하는 대화상자를 실행한다.
Linked	이 값이 True 이면 객체가 연결된 것이다.
LoadFromFile	지정된 파일에서 OLE 객체를 읽어온다.
LoadFromStream	스트림에서 OLE 객체를 읽어온다.

Modified	이 값이 True 이면 OLE 객체가 변경된 것이다.
ObjectVerbs	OLE 객체가 지원하는 모든 verbs 의 이름을 스트링 리스트로 반환한다.
OleClassName	OLE 객체의 클래스 이름을 지정한다.
OleObjectInterface	OLE 객체에 대한 IOleObject 인터페이스를 반환한다.
SaveToFile	OLE 객체를 지정된 파일에 저장한다.
SaveToStream	OLE 객체를 스트림에 저장한다.
Copy	클립보드로 컨테이너의 객체를 복사한다.
Paste	클립보드에서 컨테이너로 붙여넣는다.
Run	OLE 객체를 ovRunning 상태로 전환한다.
UpdateObject	OLE 객체의 현재 데이터를 반영하기 위해 소스를 다시 읽어들인다.
UpdateVerbs	OLE 객체의 verbs 리스트를 refresh 한다.
SourceDoc	연결된 OLE 객체에 대한 소스 문서의 이름
State	OLE 객체의 state (osEmpty, osLoaded, osRunning, osOpen, osInPlaceActive, osUIActive)
StorageInterace	OLE 객체의 IStorage 인터페이스

ToleContainer 컴포넌트는 4 가지 이벤트를 지원한다. OnActivate 이벤트는 OLE 객체가 활성화될 때 발생하며, OnDeactivate 이벤트는 OLE 객체가 비활성화될 때 발생한다.

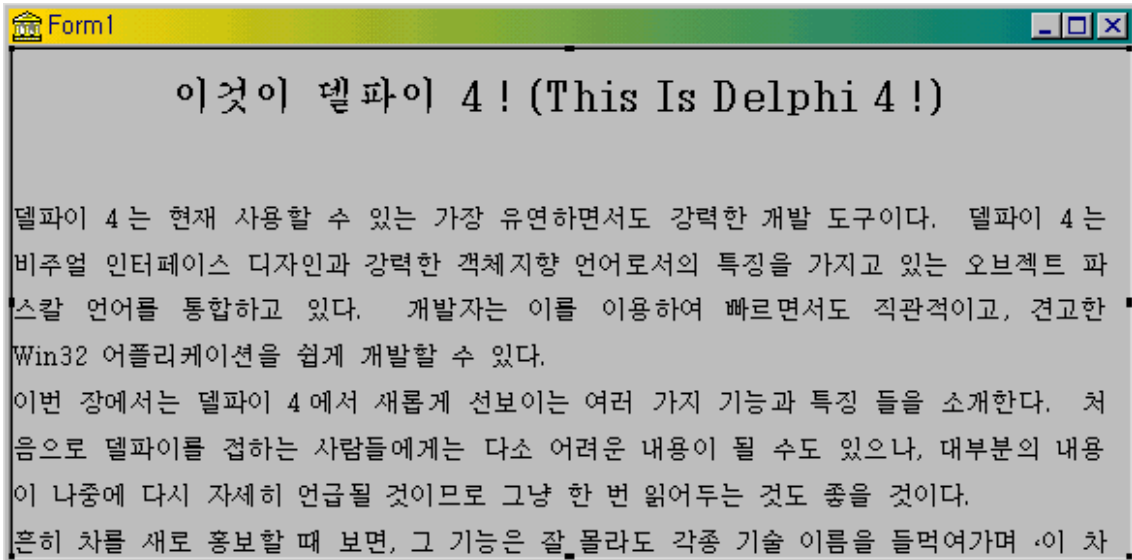
또한, OnObjectMove 이벤트는 OLE 객체의 크기나 위치가 바뀔 때 발생하는 것으로 이벤트의 Bounds 파라미터는 in-place 객체의 사각형 좌표를 지정한다. OnResize 이벤트는 OLE 컨테이너 윈도우의 크기가 변경될 때 발생하는 것으로 SizeMode 프로퍼티의 값이 smAutoSize 일 경우에 컨테이너 윈도우의 크기가 자동으로 변경된다.

간단한 OLE 컨테이너 어플리케이션을 만들기 위해서, 먼저 OleContainer 컴포넌트를 폼에 올려 놓는다. 그리고, 이 컴포넌트를 선택하고 오른쪽 마우스 버튼을 클릭해서 팝업 메뉴를 띄운 후 여기에서 Insert Object 명령을 선택하면 표준 OLE Insert Object 대화 상자가 나타난다. 이 대화 상자에는 레지스트리에 저장된 OLE 서버 어플리케이션이 나열되는데, 여기서 적당한 객체를 선택해서 삽입하면 컨테이너 컴포넌트에 데이터가 표시된다. 이렇게 객체가 컨테이너 안에 삽입되면 새로운 메뉴 항목으로 OLE 객체의 프로퍼티를 변경하거나, 다른 OLE 객체를 삽입하는 메뉴, 객체의 복사 삭제, 해당 객체의 verb(Edit, Open, Play 등)가 메뉴로 추가된다. Verb 란 간단하게 설명하면 OLE 도큐먼트 객체의 특정 기능을 수행하는 명령이라고 생각하면 된다.

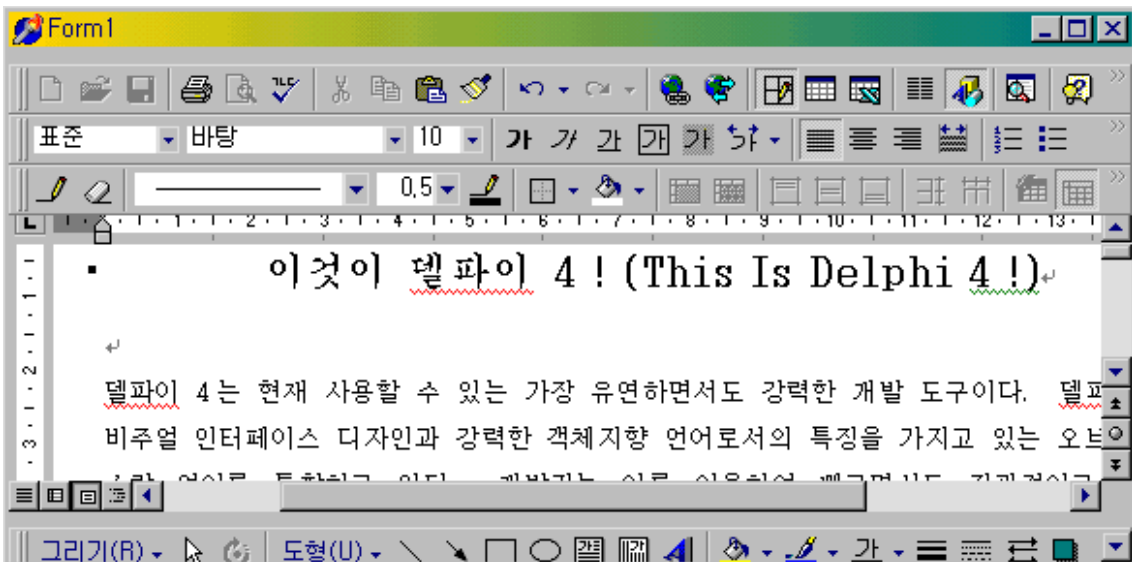
일단 이렇게 OLE 객체를 컨테이너에 추가하면, OLE 서버가 활성화되어 새로운 객체를 편집할 수 있게 된다. 서버 어플리케이션을 종료하면, 델파이가 컨테이너의 객체를 업데이트 해서 표시하게 된다.

간단한 OLE 컨테이너 객체를 제작하는 방법은 매우 간단하다. 그냥 ToleContainer 컴포넌트를 다음과 같이 올려 놓고, Align 프로퍼티를 alClient 로 설정하기만 하면 된다. 그리

고, 초기에 보여줄 객체를 오른쪽 버튼을 눌러서 Insert Object 메뉴를 선택한 후 적당한 객체를 삽입한다. 여기서는 이 책의 1 장을 선택하였다. MS 워드 문서로 작성되었기 때문에 다음과 같이 삽입된다.



이제 이 예제를 실행하고, 컨테이너 컴포넌트를 더블 클릭하면 워드가 작동하면서 다음과 같이 서버 윈도우가 어플리케이션 내부에서 실행되는 것을 볼 수 있을 것이다.

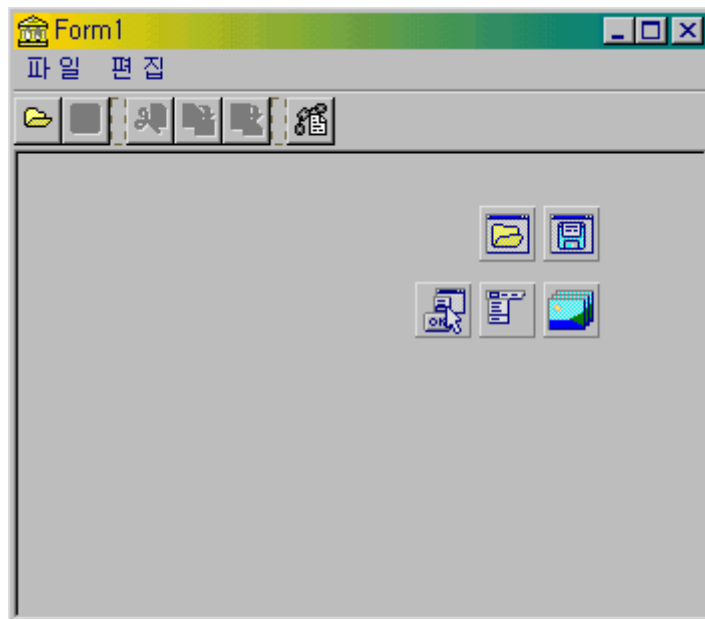


언뜻 보아서는 마치 워드를 실행해서 문서를 불러온 것 같이 보일 것이다. 하지만 잘보면 폼의 캡션이 'Form1'으로 이는 델파이 어플리케이션의 컨테이너 내에서 툴바와 함께 in-place 활성화가 된 것이다. 만약 어플리케이션에 메뉴 컴포넌트를 추가했으면, 메뉴도 병합되어 보일 것이다.

이때 AutoActivate 프로퍼티가 aaDoubleClick 으로 설정되어 있기 때문에, 더블 클릭할 경우 객체 편집 모드로 들어가는 것이다. 만약 이 값이 aaManual 이면 Active 속성을 이용해서 코드로 서버를 활성화 시킬 수 있게 되고, aaGetFocus 이면 입력 포커스가 컨테이너에 오기만 하면 서버가 활성화된다.

- OLE 컨테이너 어플리케이션 제작

그러면, 여러가지 OLE 서버 객체를 담을 수 있는 OLE 컨테이너 어플리케이션을 제작해보자. 먼저 폼을 다음과 같이 디자인한다.



폼에 TOleContainer, TImageList, TMainMenu, TToolBar, TActionList, TOpenDialog, TSaveDialog 컴포넌트를 각각 추가한 것이다. 액션 리스트의 사용법에 대해서는 이미 자세히 설명한 바 있으므로, 다시 다루지는 않겠다. ImageList 에 적절한 비트맵을 추가하여 툴바와 메뉴의 비트맵으로 사용할 수 있도록 한다. 즉, MainMenu 와 Toolbar, ActionList 컴포넌트의 Images 프로퍼티를 ImageList 로 선택하고 메인 메뉴와 툴바의 Action 을 적절하게 설정하면 된다.

이 어플리케이션에서 사용되는 액션에는 다음과 같은 것들이 있다.

액션	역할	액션	역할
actNew	새로운 객체 생성	actOpen	파일에서 객체 읽기
actSave, actSaveAs	객체를 파일로 저장	actExit	종료
actCut	객체 오려두기	actCopy	객체 복사하기

actPaste	객체 붙이기	actInsert	객체 삽입
----------	--------	-----------	-------

먼저, 파일의 내용을 저장할 전역 변수를 선언한다.

```
var
```

```
  Form1: TForm1;
```

```
  ObjectFileName: TFileName;
```

폼의 OnCreate 이벤트 핸들러에서 ObjectFileName 변수를 초기화 한다.

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
```

```
  ObjectFileName := '';
```

```
end;
```

그리고, 메뉴를 클릭할 때 파일 저장과 복사, 잘라내기, 붙여넣기 등의 액션과 메뉴의 Enabled 프로퍼티를 조절하도록 한다.

```
procedure TForm1.mnFileClick(Sender: TObject);
```

```
begin
```

```
  with OleContainer1 do
```

```
    begin
```

```
      actSave.Enabled := Modified;
```

```
      mnSaveAs.Enabled := Modified;
```

```
    end;
```

```
end;
```

```
procedure TForm1.mnEditClick(Sender: TObject);
```

```
begin
```

```
  with OleContainer1 do
```

```
    begin
```

```
      actCut.Enabled := State <> osEmpty;
```

```
      actCopy.Enabled := State <> osEmpty;
```

```
      actPaste.Enabled := CanPaste;
```

```
    end;
```

```
end;
```

이제, 파일을 읽어오거나 새로운 객체를 생성하는 부분을 구현할 차례이다.

```
procedure TForm1.actOpenExecute(Sender: TObject);
```

```
begin
```

```
  with OpenFileDialog1 do
```

```
    if Execute then
```

```
      begin
```

```
        OleContainer1.CreateObjectFromFile(FileName, False);
```

```
        ObjectFileName := FileName;
```

```
        actCut.Enabled := True;
```

```
        actCopy.Enabled := True;
```

```
        actPaste.Enabled := True;
```

```
      end;
```

```
end;
```

```
procedure TForm1.actNewExecute(Sender: TObject);
```

```
begin
```

```
  if (OleContainer1.State = osEmpty) or
```

```
    (MessageDlg('현재 객체를 삭제할까요?', mtConfirmation, mbOkCancel, 0) = mrOk) then
```

```
  begin
```

```
    with OleContainer1 do
```

```
      begin
```

```
        DestroyObject;
```

```
        actInsertExecute(Sender);
```

```
        DoVerb(PrimaryVerb);
```

```
        ObjectFileName := '';
```

```
      end
```

```
    end;
```

```
end;
```

```
procedure TForm1.actInsertExecute(Sender: TObject);
```

```
begin
```

```
  if (OleContainer1.State = osEmpty) or
```

```
    (MessageDlg('현재 객체를 삭제할까요?', mtConfirmation, mbOkCancel, 0) = mrOk) then
```

```
    if OleContainer1.InsertObjectDialog then
```

```

begin
  actCut.Enabled := True;
  actCopy.Enabled := True;
  actPaste.Enabled := OleContainer1.CanPaste;
  with OleContainer1 do
    DoVerb(PrimaryVerb);
  end;
end;

```

3 가지 액션이 모두 비슷한 역할을 한다. ‘열기’ 메뉴를 선택한 경우에는 일단 OpenFileDialog 를 실행하여 파일 이름을 얻은 후 OleContainer 의 CreateObjectFromFile 메소드를 이용한다. 그에 비해 ‘새 문서’를 선택한 경우에는 먼저 컨테이너의 객체를 DestroyObject 메소드를 이용해서 삭제한 후 ‘객체 삽입’을 수행하게 된다.

‘객체 삽입’ 메뉴에서는 InsertObjectDialog 메뉴를 호출하여 표준 대화상자에서 삽입할 객체를 선택하도록 한다.

Cut, Copy, Paste 에 대한 액션은 다음과 같이 구현한다.

```

procedure TForm1.actCopyExecute(Sender: TObject);
begin
  OleContainer1.Copy;
  actPaste.Enabled := True;
end;

```

```

procedure TForm1.actCutExecute(Sender: TObject);
begin
  if OleContainer1.State <> osEmpty then
    with OleContainer1 do
      begin
        Copy;
        DestroyObject;
        actCopy.Enabled := False;
        actPaste.Enabled := OleContainer1.CanPaste;
        ObjectFilename := '';
      end;
    end;
end;

```



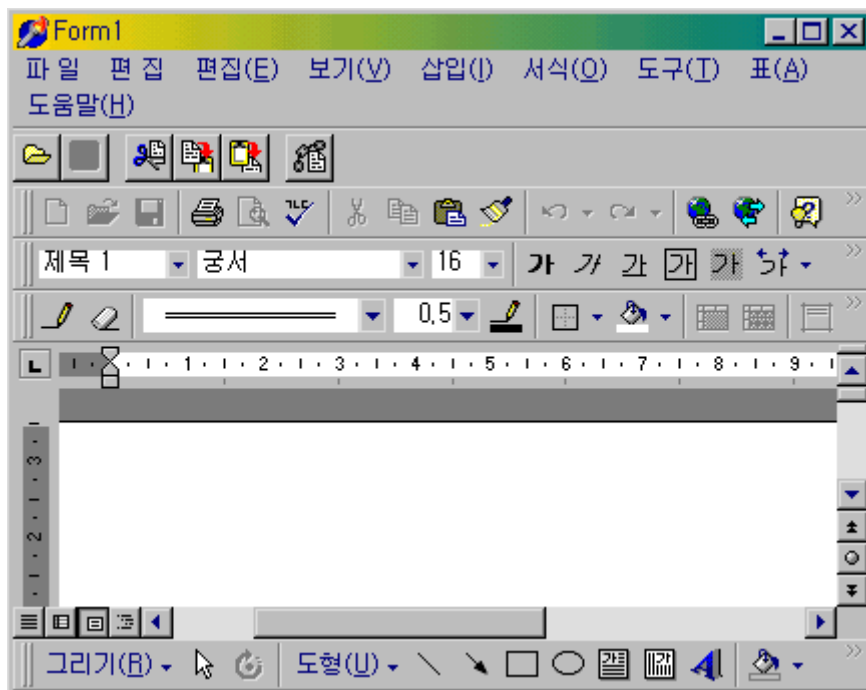
```

procedure TForm1.actPasteExecute(Sender: TObject);
begin
  if (OleContainer1.State = osEmpty) or
    (MessageDlg('현재 객체를 교체할까요?', mtConfirmation, mbOkCancel, 0) = mrOk) then
  begin
    OleContainer1.Paste;
    actCopy.Enabled := True;
    actCut.Enabled := True;
  end;
end;
end;

```

복사는 단순히 Copy 메소드를 호출하면 된다. 그에 비해 Cut 는 컨테이너에 객체가 들어 있다면 DestroyObject 메소드를 호출하여 이를 삭제하고, Copy 메소드를 수행하도록 한다. 또한, 이런 변경 사항에 따른 부대적인 설정이 더 필요하다. Paste 는 Paste 메소드를 이용하여 구현하게 된다.

이것으로 구현이 완료되었다. 이 예제는 Chap27 디렉토리의 Exam1.dpr 프로젝트로 저장되어 있으니 이를 참고하기 바란다. 컴파일하고 실행하면 쉽게 여러가지 OLE 서버 객체를 컨테이너에 담을 수 있다. 또한, 다음과 같이 in-place 활성화를 시켜서 비주얼 편집을 하게 되면 메뉴 병합 등이 실행되는 것을 볼 수 있을 것이다.



자동화 컨트롤러(Automation controller)의 제작

자동화 컨트롤러는 자동화 서버를 제어하는 어플리케이션을 말한다. 자동화 서버는 자동화 프로토콜을 통해 접근하여 기능을 할 수 있는 어플리케이션으로 대표적인 자동화 서버로는 마이크로소프트 워드, 엑셀, 인터넷 익스플로러 등을 들 수 있다.

자동화 서버의 타입 라이브러리를 import 하면 자동으로 자동화 서버를 제어할 수 있는 클래스가 만들어진다. 이런 클래스에는 dispatch 인터페이스 wrapper 와 경우에 따라서는 듀얼 인터페이스 wrapper 가 포함되어 있다. 타입 라이브러리를 import 하려면 다음과 같이 하면 된다.

1. Project|Import Type Library 메뉴를 선택한다.
2. 목록에서 타입 라이브러리를 선택한다.

만약 타입 라이브러리가 목록에 없다면, Add 버튼을 누르고 타입 라이브러리 파일(TLB)을 찾아서 이를 선택하고 OK 버튼을 누른다.

OLE 자동화 기법을 이용한 엑셀과 워드 제어

델파이를 이용해서 프로그래밍을 하다가 보면 간혹 오피스와의 연계를 통한 프로그래밍이 필요한 경우가 있다. 이럴 때에는 OLE 자동화 기법을 이용해서 오피스 객체를 직접 제어하면 생각보다 쉽게 문제를 해결할 수가 있다.

기본적으로 OLE 자동화를 이용해서 오피스 객체를 사용하는 것은 그다지 어렵지 않다. 실제로 공부를 해야 하는 것은 델파이의 문제라기 보다는, 다소 복잡한 엑셀과 워드 객체의 구조를 익히는 것이 문제라고 할 수 있다.

이렇게 엑셀과 워드를 OLE 자동화를 이용해서 다루는 방법에는 크게 2 가지가 있다.

첫 번째 방법은 워드와 엑셀 객체를 가변형 변수와 IDispatch 인터페이스를 이용해서 사용하는 것이며, 두 번째 방법은 표준 COM 인터페이스와 dispinterface 를 이용하는 방법이다. 이 2 가지 방법에는 중요한 차이가 있는데 가변형 변수를 사용하는 방법은 아주 쉽지만 속도가 다소 처지는 방법이며, COM 인터페이스를 사용하는 방법은 다소 어렵지만 속도가 빠르다.

OLE 자동화를 원활하게 사용하려면, 시스템의 RAM 이 많이 필요하다. OLE 자동화는 실제로 CPU 속도보다는 메모리 크기에 수행 속도가 좌우된다고 해도 과언이 아니다. 그러므로, 가능하면 많은 메모리를 확보한 시스템에서 돌릴 수 있도록 하는 것이 중요하다.

- OLE 자동화를 시작하자 !

앞에서도 언급했지만, 델파이에서 OLE 자동화를 사용하는 방법에는 크게 2 가지가 있다. 인터페이스를 직접 이용하거나 IDispatch 인터페이스와 가변형 변수를 사용하는 방법이 그것인데, 인터페이스를 직접 사용하면 클라이언트 측에서 개발자의 코드의 데이터 형 검사를 하기 때문에 비교적 빠른 속도를 낼 수 있다. 이런 방식을 ‘early binding’이라고 한다. 그렇지만, 처음 OLE 자동화를 익히는 사람에게는 이 방식보다는 IDispatch 인터페이스와 가변형 변수를 이용하는 방법이 훨씬 쉽기 때문에, IDispatch 인터페이스를 이용하는 방법에 대해서 알아보도록 하자.

다음의 코드는 엑셀을 시작하는 핵심 부분이다.

```
var
    V: Variant;
begin
    V := CreateOLEObject('Excel.Application');
    V.Visible := True;
end;
```

반대로, 엑셀을 마칠 때에는 다음과 같이 한다.

```
if not VarIsEmpty(V) then V.quit;
```

앞의 코드를 사용하려면, 먼저 델파이 유닛의 uses 절에 ComObj 를 추가해야 한다. ComObj 유닛에는 OLE 자동화 객체를 불러오고, 이들을 호출할 수 있는 루틴들이 포함되어 있다. 보통 가장 많이 사용하는 함수가 앞에서 사용한 CreateOLEObject 함수이다. 그리고 그 밖에도 VarDispInvoke, DispatchInvoke, GetIDsOfNames 등의 함수를 사용할 수 있다.

처음에 사용한 CreateOLEObject('Excel.Application') 줄이 실행되면 엑셀이 백 그라운드에서 시작되기 때문에 사용자에게 보이지 않는다. 보통 엑셀 객체의 엔진만 사용하는 경우라면 이렇게 생성한 객체를 직접 사용해서 원하는 작업을 하면 된다. 그런데, 우리가 하는 작업이 엑셀에서 어떻게 이루어지는지를 보여주려면 Visible 프로퍼티를 True 로 설정해야 한다. 보통 코드가 어떻게 작동하는지 눈으로 볼 수 있기 때문에, 디버깅 과정에서는 이를 True 로 설정했다가 디버깅이 끝난 후에는 나타나지 않도록 해서 속도를 증진 시키는 것이 바람직하다.

여기에서 사용한 V 라는 변수는 가변형 변수이며, CreateOLEObject 함수는 내부적으로 여러가지 OLE 함수를 호출한다. 그 결과로 COM 객체를 생성하고 이 객체의 IDispatch 인터페이스를 돌려주는 역할을 한다.

가변형(variant) 변수는 기본적으로 어떠한 데이터 형도 모두 담을 수 있다. 그렇기 때문에,

여기에 COM 객체에 대한 IDispatch 인터페이스도 담을 수 있는 것이다.

그러니까, 'V := CreateOLEObject('Excel.Application')' 이라는 코드의 의미는 엑셀의 Application 이라는 COM 객체에 대한 인스턴스를 생성하고, 이 인스턴스를 가리키는 IDispatch 인터페이스를 V 라는 가변형 변수에 대입하는 것이다. 이렇게 되면, V 라는 가변형 변수는 이때부터 직접 COM 객체를 다루듯이 사용할 수 있게 된다. 그렇기 때문에, 'V.Visible := True' 라는 문장이 동작하게 되는 것이다.

보기에는 간단하지만 'V.Visible := True'라는 문장에 의해 내부적으로 많은 일이 일어나게 되는데, 이를 수행하기 위해 IDispatch 인터페이스의 GetIDsOfNames, Invoke 메소드가 호출된다.

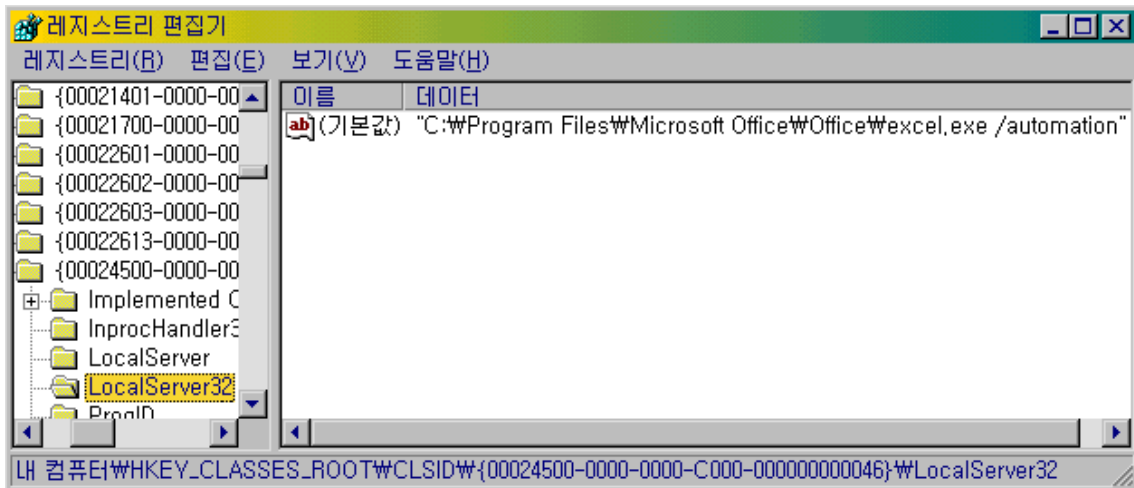
기본적으로 IDispatch 인터페이스는 가변형 변수를 사용해서 쉽게 사용할 수 있도록 디자인되었다. 이를 이용하면 디자인 시에 데이터 형 검사를 하지 않고, 런타임에서 데이터 형을 검사하게 되는데, 이를 'late binding'이라고 한다. 다른 말로 앞의 코드를 예로 들면, 델파이는 엑셀의 Application 객체가 Visible 이라는 프로퍼티가 있는지 전혀 알지 못한다. 그렇기 때문에, 만약에 COM 객체가 지원하지 않는 프로퍼티나 메소드를 사용한다고 해도 델파이의 컴파일러는 아무런 에러도 발생시키지 않는다. 다만, 실제로 이 메소드나 프로퍼티가 호출될 때 예외가 발생하게 될 것이다.

그러면 이제 마칠 때 사용하는 코드를 살펴 보자. VarIsEmpty 함수는 파라미터로 전해지는 가변형 변수가 어떤 객체나 실제 데이터를 가리키는지 알아보게 된다. 그러므로, 'if not VarIsEmpty(V) then ...' 의 의미는 가변형 변수 V 가 실제로 어떤 객체를 가리키고 있다면 then 절의 문장이 실행된다는 것이다.

이 문장에 전혀 문제가 없는 것은 아닌데, 예를 들어 V 에 엑셀의 객체가 아닌 다른 값이 들어 있더라도 then 절이 실행되게 된다.

● 엑셀 자동화 객체의 생성

CreateOLEObject 함수를 사용하면 COM 객체를 생성하고, 여기에 대한 IDispatch 인터페이스를 얻을 수 있다는 것을 앞에서 설명했다. 이 함수는 파라미터로 COM 객체를 지정하는 문자열을 받게 되어 있다. 앞에서 예를 든 코드에서는 'Excel.Application'이 사용되었다. 이런 문자열은 시스템의 레지스트리에서 발견할 수 있는데, 좀더 정확히 말하면 이 문자열에 해당되는 CLSID 값을 레지스트리에서 찾아서 레지스트리의 LocalServer 키의 정보를 가지고 COM 객체의 인스턴스를 생성하게 된다. 엑셀의 경우 Local Server 정보는 다음 그림과 같이 엑셀 파일의 패스 경로와 /automation 이라는 스위치로 이루어져 있다.



이렇게 프로그램을 나타내는 문자열을 프로그램 ID 라고 하며, 'ProgIDs'로 표시한다.

- 엑셀 자동화 객체의 이해

엑셀 자동화 객체의 구조에서 가장 상위의 객체가 Application 객체이다. 그 밑에 Workbooks, Worksheets, Charts 등의 객체가 존재한다. 그런데, 이러한 객체의 구조는 델파이의 VCL 객체구조와는 다소 차이가 있다. 델파이의 방식으로 이해를 하면 Workbooks 객체는 Application 객체의 자손이며, Worksheets 객체는 Workbooks 객체의 자손 객체로 생각할 수도 있다. 그러나, OLE 자동화에서 나타나는 객체의 구조는 OOP의 계층 구조와는 전혀 다른 형태를 가지고 있다. OLE 자동화에서의 계층 구조는 단지 어떤 객체에 접근할 때, 어떤 객체를 통해서 접근해야 하는지를 나타내는 것이다. 물론, OOP의 계층 구조로도 이를 나타낼 수는 있지만, 근본적으로는 다른 것임을 잊지 말도록 한다.

예를 들어, OOP의 계층 구조라면 A 밑에 B, B 밑에 C가 있다면, A.B, A.B.C의 형태로는 접근이 가능하지만, A.C의 형태의 접근은 불가능하다. 그러나, OLE 자동화의 계층 구조에서는 이것도 가능한 경우가 많다.

엑셀의 Application 객체는 사용이 가능한 엑셀 자동화 객체들의 세트를 나타내는 가장 일반적이고, 추상적인 객체라고 말할 수 있다. 즉, Application 객체는 엑셀에서 접근이 가능한 모든 기능을 담고있는 최상위 레벨의 컨테이너이다.

Application 객체의 바로 아래 객체인 Workbooks 객체는 Worksheets와 Charts 객체의 집합을 포함하고 있다. Worksheet 객체는 스프레드 시트의 표준 페이지를 나타내며, Chart 객체는 그래프를 나타낸다. 또한, Sheets 객체는 Worksheets와 Charts 객체를 모두 포함한다.

이러한 엑셀의 객체를 사용하는 순서와 방법은, 실제로 엑셀이라는 스프레드 시트 어플리케이션을 사용하는 순서에 입각해서 생각하면 비교적 쉽게 익힐 수 있다. 보통의 경우 엑셀을 사용하는 사용자는 일단 스프레드 시트 파일을 열고, 데이터를 입력한 후, 이를 이용한

계산식을 입력하고, 데이터에 대한 그래프를 그리게 된다. 엑셀 자동화 객체를 이용하는 방법도 여기에 입각해서 생각하면 비교적 간단하게 이해할 수 있다.

그럼 이제 실제 엑셀 객체를 이용하는 프로그램을 작성해 보도록 하자.

새로운 어플리케이션을 시작하고, 폼 위에 버튼 1 개와 리스트 박스를 1 개 없도록 하자.

그리고, OLE 자동화를 사용해야 하므로 폼 클래스 선언문의 private 섹션에 XL 이라는 가변형 변수를 선언하고, uses 절에 ComObj 를 추가한다.

버튼의 OnClick 이벤트 핸들러를 다음과 같이 작성하자.

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
const
```

```
  {Excel Sheet Type}
```

```
  xlChart = -4109;
```

```
  xlDialogSheet = -4116;
```

```
  xlExcel4IntlMacroSheet = 4;
```

```
  xlExcel4MacroSheet = 3;
```

```
  xlWorksheet = -4167;
```

```
  {Excel WBA Template}
```

```
  xlWBATChart = -4109;
```

```
  xlWBATExcel4IntlMacroSheet = 4;
```

```
  xlWBATExcel4MacroSheet = 3;
```

```
  xlWBATWorksheet = -4167;
```

```
var
```

```
  i, k: Integer;
```

```
  Sheets: Variant;
```

```
begin
```

```
  XL := CreateOLEObject('Excel.Application');
```

```
  XL.Visible := True;
```

```
  XL.Workbooks.Add;
```

```
  XL.Workbooks.Add(xlWBATChart);
```

```
  XL.Workbooks.Add(xlWBATWorksheet);
```

```
  XL.Workbooks[2].Sheets.Add(,1, xlChart);
```

```
  XL.Workbooks[3].Sheets.Add(,1, xlWorksheet);
```

```
  for i := 1 to XL.Workbooks.Count do
```

```
  begin
```

```

ListBox1.Items.Add('워크북 이름: '+XL.Workbooks[i].Name);
for j := 1 to XL.Workbooks[i].Sheets.Count do
    ListBox1.Items.Add(' 시트 이름: '+XL.Workbooks[i].Sheets[j].Name);
end;
end:
end:

```

Workbooks 는 일종의 컬렉션 객체로 Workbooks 객체의 컬렉션이다. 마찬가지로 Sheets, Charts 역시 각각 Sheet, Chart 의 컬렉션 객체이다. Workbooks 객체의 Add 메소드는 Workbooks 컬렉션에 새로운 Workbook 객체를 추가하는 메소드이다. COM 객체의 메소드를 사용할 때에는 가변 파라미터 리스트(variable parameter list)라는 기법을 사용할 수 있는데, 이 기법은 메소드의 파라미터를 원한다면 건너뛴 수 있는 것이다. 위에서 보면 Workbooks 의 Add 메소드가 처음에는 아무런 파라미터 없이도 쓰였다가, 그 뒤의 두 줄에서는 파라미터를 하나씩 가지고 있다. 이 파라미터로 쓰인 것이 어떤 형태의 Sheet 객체를 추가할 것인가를 결정한 것인데, xlWBATWorksheet 파라미터는 Worksheet 객체를 xlWBATChart 파라미터는 Chart 객체를 추가한다는 것을 의미한다.

이러한 컬렉션 객체는 뒤에 대괄호로 숫자를 넣어서 몇 번째 객체임을 표시할 수 있게 되어 있다. Workbooks[2]란 두 번째 Workbook 객체를 의미한다.

```
XL.Workbooks[2].Sheets.Add(,1, xlChart)
```

앞의 코드에서는 Sheets 객체의 Add 메소드가 사용되었다. Sheets 객체의 Add 메소드는 4 개의 파라미터를 가진다. 첫번째와 두번째 파라미터인 Before, After 는 각각 새로운 sheet 가 추가될 때 그 전후에 위치하는 sheet 객체를 담은 가변형 값을 지정하며, 세번째인 Count 파라미터에는 추가할 시트의 수를, 마지막으로 네번째인 Type 파라미터에는 추가할 시트의 종류를 지정하게 된다. 추가되는 Sheet 의 종류로는 xlWorksheet, xlChart, xlExcel4MacroSheet, xlExcel4IntlMacroSheet 등의 것들이 있는데, 디폴트 값으로는 xlWorksheet 가 지정된다.

앞의 코드의 경우에는 콤마를 처음에 2 개 찍음으로써 처음 2 개의 파라미터는 생략하고, xlChart 형의 Sheet 객체를 하나 추가하게 된다. 이런 식으로 파라미터를 생략하는 것은 IDispatch 를 이용한 'late binding'에서만 허용되며, 우리가 나중에 공부하게 될 COM 인터페이스를 직접 이용하는 방법에서는 사용할 수 없다.

그 뒤에 나오는 for..loop 에서는 Workbook 객체의 이름과 각 Workbook 객체의 Sheet 객체의 이름을 리스트 박스에 표시하게 된다.

그럼, 여기에서 선언해서 사용된 각종 상수 들의 구체적인 값을 알아내는 방법을 알아보자. 기본적으로 이런 상수는 엑셀의 타입 라이브러리(type library)의 값을 읽어서 알아낼 수 있다. 타입 라이브러리의 값을 읽는 방법은 마이크로소프트 SDK 에서 제공되는 OleView 어

플리케이션 등의 도구를 이용하거나, 델파이의 타입 라이브러리 에디터를 사용할 수 있다.

여기서는 델파이에 내장된 타입 라이브러리를 이용하는 방법을 사용하도록 하겠다.

델파이 메뉴에서 Project|Import Type Library 를 선택하고, 엑셀의 타입 라이브러리 파일인 Excel8.olb 파일을 선택한다. 그러면, 많은 수의 경고 메시지가 나오기는 하지만 예러는 없이 타입 라이브러리가 로드될 것이다. 이렇게 많은 경고 메시지가 나오는 이유는 엑셀의 타입 라이브러리에 정의된 많은 수의 이름과 델파이의 예약어가 같을 경우, 이 이름들을 자동으로 수정하면서 나오는 메시지이다. 보통의 경우 이름의 처음이나 끝에 밑줄(_) 기호가 추가된다. 이렇게 해서 타입 라이브러리를 읽어온 후에 원하는 상수의 실제 값을 찾아보면 된다. 그러나, 매번 엑셀이나 워드 등의 자동화 객체를 사용할 때마다 이들 상수를 선언해서 사용하는 것은 매우 비효율적이다. 그러므로, 엑셀과 워드에서 사용되는 상수들을 선언한 유닛을 따로 만들어 놓고 이를 이용하는 것이 좋을 것이다. 그래서, 앞으로의 설명에는 엑셀과 워드에 대한 상수를 담은 유닛인 XLConst.pas, WordConst.pas 유닛을 만들어 놓고, 이를 사용하도록 하겠다. 이들 유닛은 이달의 디스켓으로 제공되니 유용하게 사용하기 바란다.

- 엑셀 워크시트를 사용하자

이번에는 워크 시트를 이용해서 실제 데이터를 입력하고, 이를 이용해 계산을 하는 방법과 데이터를 저장하는 방법을 알아보도록 하자.

새로운 어플리케이션을 시작하고 폼에 버튼을 하나 없도록 하자. 그리고, 폼 클래스의 private 섹션에 XL 이라는 가변형 변수를 선언하고, uses 절에 ComObj.pas 와 XLConst.pas 유닛을 추가한다. 이제 Button1 에 대한 OnClick 이벤트 핸들러를 다음과 같이 작성하자.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
  Sheet, Range, Column: Variant;
begin
  XL := CreateOLEObject('Excel.Application');
  XL.Visible := True;
  XL.Workbooks.Add(xlWBATWorksheet);
  XL.Workbooks[1].Worksheets[1].Name := 'Example1';
  Sheet := XL.Workbooks[1].Worksheets['Example1'];
  for i := 1 to 10 do
    Sheet.Cells[i, 1] := i;
```



```

Range := XL.Workbooks(1).Worksheets('Example1').Range('B1:F10');
Range.Formula := '=RAND()';
Range.Columns.Interior.ColorIndex := 5;
Range.Borders.LineStyle := xlContinuous;
Column := XL.Workbooks[1].Worksheets(1).Columns;
Column.Columns[1].ColumnWidth := 5;
Column.Columns[1].Font.Color := clRed;
end;

```

일견 복잡해 보이지만, 그다지 어렵지는 않을 것이다. 위의 코드에서 사용하는 OLE 자동화 객체를 각각 Sheet, Range, Column 이라는 가변형 변수에 담아서 사용한다. 일단 XL 이라는 가변형 변수에 엑셀의 Application 자동화 객체를 저장하고, Workbooks 통합문서에 워크 시트를 하나 추가한다. 그리고, 추가한 워크 시트의 이름을 'Example1'이라고 명명했다.

그러면 이제 가변형 변수 Sheet 에 우리가 사용할 Worksheet 객체를 대입하도록 하자.

Sheet := XL.Workbooks[1].Worksheets['Example1'] 이라는 문장은 엑셀의 첫번째 통합 문서내의 'Example1' 이라는 이름의 워크시트 객체를 찾아서 대입한다. 이때 Workbooks[1] 통합문서의 첫번째 워크 시트의 이름을 'Example1'으로 지정했으므로, Worksheets['Example1']은 Worksheets[1]과 같은 의미를 가지게 된다.

Sheet 객체를 이용해서 실제 데이터를 입력하려면 Cells 프로퍼티나 Range 객체를 사용해야 한다. Cells 프로퍼티의 경우에는 행과 열을 각각 숫자로 기입해서 셀의 위치를 나타낸다. 그러므로, Cells[1, 1]은 엑셀의 'A1' 셀을 Cells[3, 2]는 'B3' 셀을 나타낸다. Range 객체는 한꺼번에 여러 개의 셀을 선택할 때 사용하는 것으로 앞의 예제의 Range('B1:F10')의 경우에는 'B1' 셀이 가장 좌측 위에 위치하고, 'F10' 셀이 가장 우측 아래라고 할 때 이들에 의해 둘러싸여지는 직사각형 범위의 모든 셀을 가리키게 된다.

이를 이해하면 앞의 코드 들을 쉽게 이해할 수 있을 것이다.

한 가지 재미있는 것은 대괄호와 보통 괄호를 마구 혼용해서 썼는데, 이렇게 사용하는 것이 모두 허용된다. 그리고, 직접 셀이나, Range 객체의 Formula 프로퍼티에 문자열로 엑셀에서 사용하던 함수식을 입력할 수 있다. 그러므로, Range.Formula := '=RAND()' 와 같은 문장을 사용할 수 있다. 이 문장으로 Range 변수에 지정된 모든 셀들에 0~1 사이의 난수가 대입된다.

그 다음 문장은 Range 에 해당되는 모든 셀들의 내부 색상을 5 번 색상으로 선택하였다. ColorIndex 프로퍼티에는 미리 정의되어 있는 색상들이 있는데 1 번부터 8 번까지가 각각 black, white, red, green, blue, yellow, purple, cyan 으로 지정되어 있다. 그러므로, 이 문장에서는 5 번인 blue 컬러가 선택된다.

이렇게 색상이 선택되면 엑셀에서는 셀간을 구분하는 구분선이 없어지는데, 이를 없애지

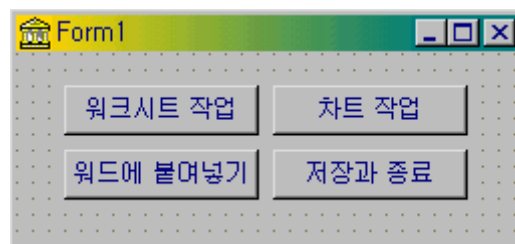
않게 하려면, 그 다음 문장에서처럼 Range 객체의 Borders.LineStyle 프로퍼티를 xlContinuous 로 지정하면 된다.

이런 방식으로 셀의 내부를 지정하는 많은 프로퍼티가 있는데, 예를 들어 Interior.Pattern 프로퍼티는 셀 내부를 그리는 패턴을 지정할 수 있다. 이들 프로퍼티에 대한 자세한 설명은 엑셀 객체에 대해 자세히 설명한 서적이나 엑셀의 도움말을 참고하기 바란다.

다음에 이어지는 문장들은 워크 시트의 Columns 객체를 Column 이라는 가변형 변수에 대입하고 이를 이용해서 Column 의 형태를 결정하는 문장 들이다. 여기에서는 첫번째 열의 폭을 5, 폰트의 색상을 clRed 로 설정하였다.

- 엑셀 차트 기능과 워드 객체의 사용

이제 데이터를 입력하고, 이를 이용해서 차트를 그리고 이를 MS 워드 문서로 복사까지 해 줄 수 있도록 확장해보자. 폼에 버튼을 3 개 더 얹고 (합이 4 개이다), 폼 클래스의 private 섹션에 Word 라는 가변형 변수를 하나 더 추가하고, uses 절에 WordConst, ActiveX 를 추가한다. 그리고 각 버튼의 캡션을 다음과 같이 ‘워크시트 작업’, ‘차트 작업’, ‘워드 붙여넣기’, ‘저장과 종료’ 라고 설정한다.



그리고, Button1 의 OnClick 이벤트를 핸들러를 다음과 같이 수정한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Sheet: Variant;
  i: Integer;
begin
  XL := CreateOLEObject('Excel.Application');
  XL.Visible := True;
  XL.Workbooks.Add(XLWBATWorksheet);
  XL.Workbooks[1].Worksheets[1].Name := 'Example1';
  Sheet := XL.Workbooks[1].Worksheets['Example1'];
  for i := 1 to 10 do
```

```
Sheet.Cells[i, 1] := i;
end;
```

Button1 의 이벤트 핸들러는 거의 앞에서 설명한 예제와 변함이 없다. 그러면, 이제 이를 이용해서 차트를 그릴 Button2 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Range, Sheets: Variant;
begin
  XL.Workbooks[1].Sheets.Add(,1, xlChart);
  XL.Workbooks[1].Sheets[1].Name := 'ExamChart1';
  Sheets := XLApp.Sheets;
  Range := Sheets.Item['Example1'].Range['A1:A10'];
  Sheets.Item['ExamChart1'].SeriesCollection.Item[1].Values := Range;
  Sheets.Item['ExamChart1'].ChartType := xl3DPie;
  Sheets.Item['ExamChart1'].SeriesCollection.Item[1].HasDataLabels := True;
end;
```

차트 객체를 처음 생성할 때에는 Sheets 객체의 Add 메소드를 사용해서 xlChart 형식의 Sheet 를 추가하면 된다. 앞의 코드에서 처음 2 개의 파라미터를 생략했는데, 이것은 앞뒤에 위치할 Sheet 를 지정하지 않은 것이며, 그 뒤의 파라미터는 xlChart 형식의 시트를 1 개 추가한다는 의미이다.

앞의 코드에서는 Sheets 객체를 계속 사용하는데, 앞에서도 설명했지만 Sheets 객체는 Worksheets 와 Charts 객체를 모두 가리킬 수 있기 때문에 꽤 편리하게 사용할 수 있다.

일단 이렇게 차트 객체가 생성되면 실제로 어떤 데이터를 가지고 차트를 그릴 것인지를 결정해야 한다. 이를 위해서 Button1Click 이벤트 핸들러에서 삽입한 셀들을 Range 로 설정해서 이 셀들을 SeriesCollection 의 Values 로 설정한다. 이렇게 Values 프로퍼티에 데이터를 설정하면 이를 이용해서 그래프를 그릴 수 있게 된다.

Series 란 그래프를 그리려고 하는 데이터의 range 를 의미한다. 그리고, SeriesCollection 이란 이러한 Series 의 Collection 이다.

그러므로, Sheets.Item['ExamChart1'].SeriesCollection.Item[1].Values := Range 란 코드의 의미는 'ExamChart1'이라는 이름을 가진 시트의 첫번째 Series 의 데이터 Range 값을 Range 라는 변수에 들어있는 값으로 설정한다는 의미이다.

그 다음 문장은 차트의 형을 3 차원 파이 그래프로 결정한 것이며, 데이터 라벨을 보이도록 하였다.

이제 이렇게 만들어진 차트를 워드에 추가해 보도록 하자. 일단은 데이터를 클립보드에 복사를 하고, 이 데이터를 다시 워드로 붙여넣기(paste) 하면 된다. Button3 의 OnClick 이벤트 핸들러를 다음과 같이 작성하자.

```
procedure TForm1.Button3Click(Sender: TObject);
var
  Sheets, Range: Variant;
  i, ParNo: Integer;
begin
  Sheets := XL.Sheets;
  Sheets.Item['ExamChart1'].Select;
  XL.Selection.Copy;
  Word := CreateOLEObject('Word.Application');
  Word.Visible := True;
  Word.Documents.Add;
  Word.Documents.Item(1).Paragraphs.Add;
  Word.Documents.Item(1).Paragraphs.Add;
  ParNo := Word.Documents.Item(1).Paragraphs.Count;
  Range := Word.Documents.Item(1).Range(
    Word.Documents.Item(1).Paragraphs.Item(ParNo).Range.Start,
    Word.Documents.Item(1).Paragraphs.Item(ParNo).Range.End);
  Range.Text := '이것은 그래프 입니다.';
  Word.Documents.Item(1).Paragraphs.Add;
  Range := Word.Documents.Item(1).Range(
    Word.Documents.Item(1).Paragraphs.Item(ParNo + 1).Range.Start,
    Word.Documents.Item(1).Paragraphs.Item(ParNo + 1).Range.End);
  Range.PasteSpecial(,,,wdPasteOleObject);
end;
```

먼저, 복사할 객체를 선택해야 한다. 객체의 Select 메소드를 사용해서 차트 객체를 선택하고, 선택된 객체를 클립보드로 복사하는 Copy 메소드를 호출한다.

그리고, 워드 객체를 사용하기 위해 Word := CreateOLEObject('Word.Application') 문장에서 워드 객체를 생성한다. 오피스 95 의 워드 까지는 워드 객체가 'Word.Basic'으로 표현되었으나, 오피스 97 의 워드 8.0 부터는 Word.Application 으로 바뀌었다. 엑셀에서의 통합 문서 객체인 Workbooks 에 해당하는 것이 워드에서는 Documents 객체이다. 그러므로,

Word.Documents.Add 메소드에 의해 새로운 워드 문서가 추가된다. 그리고, 그 다음 문장에서 사용한 Paragraphs 객체의 Add 메소드는 새로운 문단을 추가하는 역할을 하게 된다. 워드에서는 엔터 키를 누를 때마다 새로운 문단이 시작되므로 이것의 의미는 새로운 줄을 하나 추가하는 것과 같은 의미가 된다. 앞의 경우에는 처음에 두 줄의 빈문단을 삽입하게 된다. 그리고 나서, 범위를 설정하고, 그 범위에 적절한 문장을 삽입한다. Range 변수의 범위를 설정할 때 범위의 시작점과 끝점을 모두 지정하는데, 이들은 경우에 따라서는 생략이 가능하다. .

다음에 이어지는 문장에서 역시 엑셀의 차트를 워드에 삽입하는 루틴 들이 이어지고 있다. 이 문장들도 앞에서 설명한 기본적인 문장에 대한 설명과 동일하므로, 중복되는 설명은 따로 하지 않는다. 차트를 붙여넣을 때에는 Paste 메소드와 PasteSpecial 메소드를 사용할 수 있는데, 여기서는 OLE 객체로 간주하여 붙여넣게 되는 PasteSpecial 메소드를 사용한다. 이렇게 PasteSpecial 메소드를 사용해서 붙여넣은 OLE 객체는 나중에 워드 문서에서 이 객체를 더블 클릭할 때 이를 편집할 수도 있게 된다. 이 메소드는 상당히 많은 파라미터를 가지는데, 여기에 대한 설명은 오피스 97 의 OLE 자동화 객체의 메소드에 대한 책이나 도움말을 참고하기 바란다.

이제 워드에 삽입 되었을 것이다. Button4 를 누르면 워드 문서가 저장이 되고, 워드 객체와 엑셀이 모두 종료되도록 하자.

```
procedure TForm1.Button4Click(Sender: TObject);
begin
  Word.Documents.Item(1).SaveAs('c:\WExam.doc');
  if not VarIsEmpty(XL) then
  begin
    XL.DisplayAlerts := False;
    XL.Quit;
  end;
  if not VarIsEmpty(Word) then
  begin
    Word.Documents.Item(1).Close(wdDoNotSaveChanges);
    Word.Quit;
  end;
end;
```

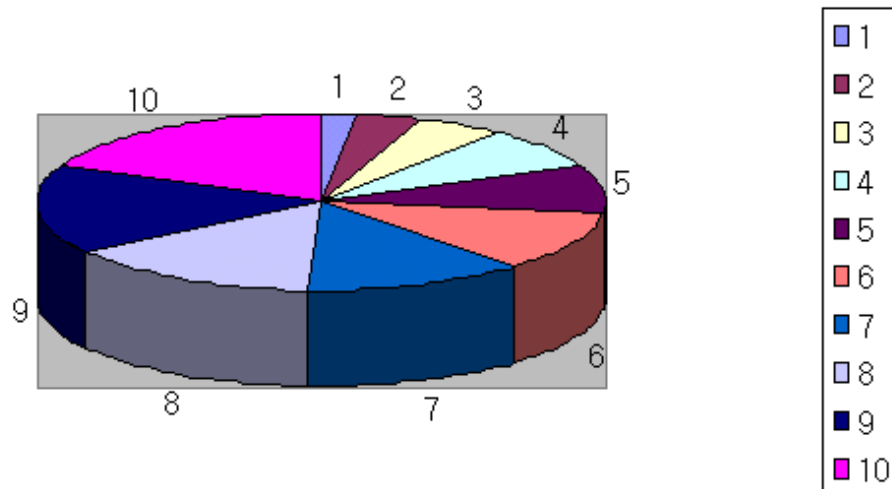
위의 코드에서 많은 부분은 앞에서 설명했다. 참고로 엑셀의 경우에는 빠져나갈 때 DisplayAlerts 프로퍼티를 False 로 설정하면 파일을 저장하지 않았는지 묻는 메시지가 나타나지 않는다. 워드의 경우에도 Close 메소드에 wdDoNotSaveChanges 를 파라미터로

넘겨주면 사용자에게 저장에 대한 메시지를 묻지 않고 종료하게 된다.

이제 완성된 예제를 실행하고, 4 개의 버튼을 차례로 누르면 c:\W 디렉토리에 Exam.doc 이라는 워드 파일이 만들어진다. 이 파일을 워드에서 읽어오면 다음 그림과 같은 문서일 것이다.

↵
↵

이것은 그래프입니다.↵



간단하게 엑셀과 워드를 제어하는 어플리케이션을 제작해 보았다. 실제로 동작하는 모습을 보여주기 위해서 Visible 프로퍼티를 True 로 설정하는 등의 수행 성능을 떨어뜨리는 코드가 많이 들어가 있으므로, 이점은 꼭 유념해서 살펴보기 바란다.

또한, 차트를 삽입할 때 워드에서 보여지는 크기 등을 지정하지 않았기 때문에, 실제 실행해보면 차트가 너무 커서 잘려서 워드에 삽입되는 등의 모습을 볼 수 있을 것이다. 이런 부분의 세세한 제어는 코드를 추가하면 쉽게 해결할 수 있다.

Early 바인딩과 Late 바인딩 (IDispatch vs VTable)

앞에서도 간단히 설명한 바 있지만, OLE 자동화를 사용할 때에는 가변형 변수를 사용하는 late 바인딩과 인터페이스를 직접 사용하는 early 바인딩이 있다.

이들은 파라미터를 넘기는 방식이나, 접근 방식에 따라 속도차이가 날 수 밖에 없다. 먼저 기본적인 메모리 사용 방법에 따른 속도를 생각해보면, 스택에 기초한 메모리를 할당하는 경우가 힙에 기초한 메모리 할당 기법에 비해 처리 속도가 빠르다. 그런데, 모든 가변형 파라미터는 기본적으로 가변형 배열로 형변환이 되는데, 이들은 모두 힙에 기초한 메모리를 사용한다.

물론 COM 인터페이스를 사용할 때 가장 빠른 방법은 VTable 의 메소드 주소를 알아내어 직접 호출하는 방법일 것이다. 그러나, 이 방법은 융통성이 적고 사용 방법이 까다로운 단점이 있다. 이를 보완하기 위해서 나온 것이 듀얼 인터페이스(dual interface)이다. 듀얼 인터페이스는 가변형으로 사용하는 IDispatch 의 유연한 동적 디스패치와 VTable 직접 접근 방법의 수행 성능을 모두 지원할 수 있는 인터페이스이다.

기본적으로 모든 COM 인터페이스는 IUnknown 에서 상속받게 되고, OLE 자동화를 지원하는 모든 인터페이스는 IDispatch 를 상속한다. 듀얼 인터페이스는 IDispatch 에서 상속받은 인터페이스로 듀얼 인터페이스는 파라미터를 스택에서 처리하며, 이렇게 하기 위해서 타입 라이브러리를 사용하므로 반드시 듀얼 인터페이스로 처리하기 위해서는 타입 라이브러리를 통해서 데이터 형을 파악할 수 있어야 한다.

- Late 바인딩과 IDispatch

Late 바인딩은 IDispatch 에 의해 구현된다. IDispatch 인터페이스는 IUnknown 인터페이스에 다음과 같은 메소드가 추가된 인터페이스이다.

```
function GetIDsOfNames(const IID: TGUID; Names: Pointer;
    NameCount, LocaleID: Integer; DisplIDs: Pointer): HRESULT; virtual; stdcall;
function GetTypeInfo(Index, LocaleID: Integer; out TypeInfo): HRESULT; virtual; stdcall;
function GetTypeInfoCount(out Count: Integer): HRESULT; virtual; stdcall;
function Invoke(DispID: Integer; const IID: TGUID; LocaleID: Integer; Flags: Word;
    var Params: VarResult, ExcepInfo, ArgErr: Pointer): HRESULT; virtual; stdcall;
```

GetIDsOfNames 메소드는 런타임에서 함수 이름을 가지고 적절한 메소드의 디스패치 ID 를 반환하는 역할을 한다. IDispatch 인터페이스는 Invoke 메소드에 디스패치 ID 를 이용하여 실제 메소드를 호출한다.

Invoke 메소드는 자동화의 핵심이라고 말할 수 있다. 디스패치 ID 를 이용해서 메소드를 호출하게 되는데, 파라미터로 가변형을 사용하게 된다. 함수 호출이 이루어지기 전에, 이러한 파라미터 들은 반드시 가변형 데이터 형으로 변환된다.

이런 방법의 장점은 호출자가 컴파일 시에 자동화 객체 메소드의 수를 알 필요가 없고, 각 함수에 대한 자세한 파라미터의 리스트를 몰라도 되며, 디폴트 파라미터나 옵션 파라미터를 설정하기가 매우 쉽다.

- 수행성능의 문제

자동화 객체를 사용할 때의 가장 근본적인 문제가 되는 것은 Win32 프로세스 공간을 넘어

다니는 것이다. Win32 프로세스 공간은 서로 독립되어 있기 때문에, 서로가 주소를 공유할 수 없는 구조로 되어 있다. 그러므로, 파라미터를 넘길 때에는 일종의 전역 메모리 버퍼나 메시징 시스템을 사용해야 하는데 이런 작업에서 시간을 많이 소요하게 된다.

그러므로, 자동화 객체 호출에서의 속도를 증진시키려면 이렇게 시간을 많이 소요하는 작업에서 조금이라도 효율적인 방법을 써야 한다. 그러므로, Invoke 메소드를 호출하기 전에 모든 파라미터를 가변형으로 변환해야 하고, 이를 함수를 호출한 쪽에서 다시 원래의 데이터 형으로 변경해야 한다면 성능에 막대한 지장을 초래할 것임은 불을 보듯이 뻔한 일이다. 이렇게 가변형으로의 파라미터 변경을 막고 직접 데이터 형을 사용할 수 있게 하려면, 메소드를 호출한 쪽에서 메소드의 이름과 파라미터의 데이터 형을 알 수 있다면 가능한데 이를 위해서 필요한 것이 바로 타입 라이브러리이다.

자동화에서 타입 라이브러리는 객체에 의해 지원되는 인터페이스, 각각의 인터페이스에 의해 지원되는 메소드의 이름과 파라미터, 적절한 도움말 파일 등의 정보를 제공하게 된다.

잘 만들어진 자동화 컨트롤러는 객체의 타입 라이브러리를 이용해서 early 바인딩을 지원하도록 제작해야 한다. 즉, 타입 라이브러리를 이용해서 지원되는 데이터 형을 가변형으로의 변환 과정이 없게 직접 사용할 수 있도록 하는 것이 핵심이다.

Late 바인딩과 early 바인딩을 모두 지원하게 하기 위해서는 듀얼 인터페이스를 사용해야 하는데 이때 파라미터로 사용되는 데이터 형은 반드시 자동화 호환(automation compatible)해야 한다. 이는 다른 말로 레코드와 같이 사용자가 정의한 데이터 형은 쓸 수 없다는 의미이다. 만약, 자동화 객체에 레코드를 파라미터로 사용하려면 데이터 멤버를 몇 개로 분리해야 한다.

- 기본적인 구현 방법

그러면, 지금까지의 대략적인 설명을 바탕으로 간단한 코드 예제로 late 바인딩과 early 바인딩을 설명하겠다. 여기서는 ISample 이라는 인터페이스가 있고, 이 인터페이스의 CoClass 가 CoSample, OLE 자동화 객체의 ProgID 를 'Sample.Application'이라고 가정하고 코드를 설명하도록 한다.

1. Late 바인딩

가장 느린 방법으로, 앞에서 예제를 통해서 설명했으므로 잘 이해했을 것으로 믿는다. 사용하기가 매우 쉽다는 장점이 있다. 앞의 가정에 의해서 late 바인딩 코드를 작성하면 다음과 같이 하면 된다.

```
var
```

```
MyServer: OleVariant;
```



```

begin
  MyServer := CreateOleObject('Sample.Application');
  MyServer.SampleMethod;
end;

```

이렇게 하면, 내부적으로는 IDispatch 인터페이스를 통해 호출을 하게 되는데 IUnknown 의 QueryInterface 로 위치를 찾기 위해 여러 인터페이스를 검색하게 되고, 인터페이스와 메소드의 위치를 찾게 되면 인덱스를 서버가 클라이언트로 넘겨 준다. 클라이언트는 내부적으로 MyServer.Invoke(인덱스)의 형태로 호출하여 동작한다.

2. Early 바인딩 – IDispatch 인터페이스

Early 바인딩에는 듀얼 인터페이스를 사용하는 방법과 IDispatch 인터페이스를 이용한 방법의 2 가지가 있다. 사용하는 방법은 간단하다. 기본적으로 인터페이스에 대한 선언부가 있는 유닛을 uses 절에 추가하고 다음과 같이 코딩하면 된다.

```

var
  MyServer: ISample;
begin
  MyServer := CreateOleObject('Sample.Application') as ISample;
  MyServer.SampleMethod;
end;

```

이렇게 하면, 컴파일할 때 메소드와 클라이언트에 메소드 수 등의 정보를 넘겨줄 수 있으므로 클라이언트는 컴파일 시에 주어진 메소드 인덱스를 이용해서 IDispatch 인터페이스의 Invoke 메소드를 호출하면 된다.

3. Early 바인딩 – VTable/듀얼 인터페이스

이 방법이 가장 빠른 방법으로, 델파이에서 타입 라이브러리를 이용해서 직접 지원하고 있는 방법이다. 사실 2 번의 방법과 속도 차이는 거의 없다. 사용 방법은 다음과 같다.

```

var
  MyServer: ISample;
begin
  MyServer := CoSample.Create;

```

```
MyServer.SampleMethod:
end:
```

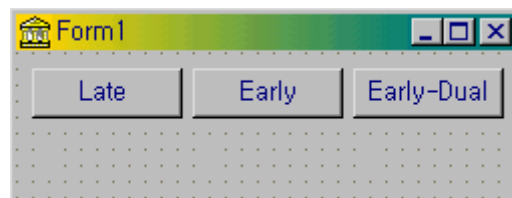
이때, CoSample 이 정의된 타입 라이브러리의 선언부 유닛이 uses 절에 추가되어 있어야 함은 물론이다. 듀얼 인터페이스는 이와 같이 타입 라이브러리 정보를 이용해서 클라이언트가 서버 객체의 주소를 직접 얻을 수 있도록 한다. 즉, 인터페이스를 질의할 필요도 없고, Invoke 메소드도 호출할 필요도 없으므로 효율적인 것이다. 듀얼 인터페이스는 이렇게 직접적인 접근 이외에도 1 번의 방법처럼 전통적인 가변형을 이용한 접근도 가능하다.

● 실전 예제

그렇다면, late 바인딩과 early 바인딩을 이용해서 같은 OLE 자동화 컨트롤러를 이용해서 반복적인 작업을 하면 얼마나 차이가 날지 무척 궁금할 것이다.

예제를 통해서 워드 객체에 작업을 앞에서 작성한 방식의 late 바인딩과 early 바인딩을 이용한 방법을 사용하여 이들을 처리하는데 걸리는 시간을 비교해 보도록 하자. 이 예제를 한번 작성하고 나면, 다른 OLE 자동화 컨트롤러 어플리케이션을 만들더라도 쉽게 early 바인딩을 구현할 수 있을 것이다.

먼저 간단하게 다음과 같이 버튼 3 개와 라벨 컴포넌트 3 개를 폼에 올려 놓자. 여기에서 라벨 컴포넌트의 캡션을 지우고, 각 버튼의 캡션을 다음과 같이 설정한다.



그러면, 이들 3 개 버튼의 OnClick 이벤트 핸들러를 작성하자. Late 버튼은 late 바인딩을 이용할 것이고, Early 버튼은 IDispatch 인터페이스의 early 바인딩을 이용할 것이다. 마지막으로 Early-Dual 버튼을 클릭하면 듀얼 인터페이스를 사용한다.

이를 위해서 먼저 타입 라이브러리를 import 할 필요가 있다. Project|Import Type Library 메뉴를 선택하면 나타나는 대화 상자에서 Microsoft Word 8.0 Object Library 를 선택하고 생성될 _TLB.pas 파일을 저장할 패스를 Unit dir name 에디트 박스에 지정하고 OK 버튼을 클릭하면 Word_TLB.pas, Office_TLB.pas, VBIDE_TLB.pas 파일이 디렉토리에 생성될 것이다.

프로젝트의 uses 절에 Word_TLB.pas, ComObj.pas 유닛을 추가하고 late 바인딩을 위해 Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```

procedure TForm1.Button1Click(Sender: TObject);
var
  MsWordOleVariant: OleVariant;
  i, StartTick, StopTick: Integer;
begin
  MSWordOleVariant := CreateOleObject('Word.Application');
  StartTick := GetTickCount;
  for i := 1 to 10 do
  begin
    MSWordOleVariant.UserName := 'Sample';
    MSWordOleVariant.UserName := '';
    MSWordOleVariant.StatusBar := 'Sample';
    MSWordOleVariant.StatusBar := '';
  end;
  StopTick := GetTickCount;
  Label1.Caption := IntToStr(StopTick - StartTick);
  MSWordOleVariant.Quit(wdDoNotSaveChanges);
  MSWordOleVariant := UnAssigned;
end;

```

Late 바인딩의 방법은 앞에서 소개한 워드, 엑셀 제어 방법과 기본적으로 동일하다. CreateOleObject 메소드를 이용하며, 생성된 객체를 OleVariant(또는 Variant) 형의 변수에 저장한다. 그리고, 이를 이용하여 접근이 가능하다. 시간은 GetTickCount 함수를 이용하여 백만분의 1 초 단위로 측정하며, UserName 과 StatusBar 프로퍼티의 내용을 10 차례에 걸쳐 변경하는 속도를 쥘다.

마지막으로 워드 객체를 닫는데, 원래의 Quit 프로시저에는 3 개의 파라미터를 가지게 되어 있지만 가변형을 사용하므로 이렇게 데이터 형에 연연하지 않고 생략된 파라미터를 이용하여 메소드 호출이 가능하다.

Early 바인딩을 위한 Button2 의 OnClick 이벤트 핸들러는 다음과 같이 작성한다.

```

procedure TForm1.Button2Click(Sender: TObject);
var
  MsWordInterface: _Application;
  i, StartTick, StopTick: Integer;
  wdSaveOptions, FormatOpt, Routing: OleVariant;
begin

```

```

MSWordInterface := CreateOleObject('Word.Application') as _Application;
wdSaveOptions := wdDoNotSaveChanges;
FormatOpt := 0;
Routing := 0;
StartTick := GetTickCount;
for i := 1 to 10 do
begin
    MSWordInterface.UserName := 'Sample';
    MSWordInterface.UserName := '';
    MSWordInterface.StatusBar := 'Sample';
    MSWordInterface.StatusBar := '';
end;
StopTick := GetTickCount;
Label2.Caption := IntToStr(StopTick - StartTick);
MSWordInterface.Quit(wdSaveOptions, FormatOpt, Routing);
end;

```


어플리케이션 인터페이스를 저장할 변수를 _Application 인터페이스로 선언한다. 그리고, 이를 CreateOleObject 함수를 이용하여 OLE 객체를 생성한 뒤 as 연산자를 이용하여 _Application 으로 형변환하여 변수에 저장하는 것이 중요한 키가 된다.

나머지 부분은 거의 동일한데, 아마도 코딩을 하다 보면 다음과 같이 코드 인사이트(Code Insight) 기능이 동작하는 것을 볼 수 있을 것이다. 이는 런타임이 아닌 컴파일 타임에서 데이터 형에 대한 검사를 하기 때문에 가능한 것이다.

```

    MSWordInterface.UserName := TempName;
    MSWordInterface.StatusBar := TempStatus;
    MSWordInterface.Q
end;
end.

```



또 한가지 눈여겨 보아야 할 것은, 마지막 부분에 Quit 메소드를 호출할 때 데이터 형에 맞도록 OleVariant 타입의 변수를 3 개 선언하고, 이들을 파라미터로 모두 사용해야만 컴파일 이 된다는 점이다. 즉, early 바인딩은 선언된 데이터 형을 그대로 사용해서 호출하지 않으면 안되는 것이다.

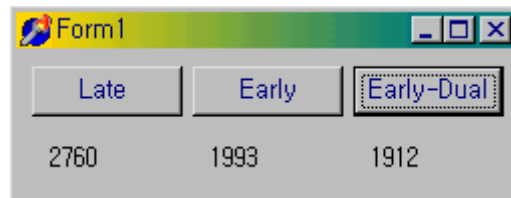
듀얼 인터페이스를 이용하는 경우도 거의 동일하다. 다만, 처음 선언부만 차이가 나는데

Button3 의 OnClick 이벤트 핸들러는 다음과 같이 작성한다.

```
procedure TForm1.Button3Click(Sender: TObject);  
var  
    MSWordInterface: _Application;  
    i, StartTick, StopTick: Integer;  
    wdSaveOptions, FormatOpt, Routing: OleVariant;  
begin  
    MSWordInterface := CoApplication_.Create;  
    ... (후략)
```

생략한 부분의 코드는 Button2 의 경우와 거의 동일하다. 참고로 인터페이스의 이름인 _Application 이나 CoClass 이름인 CoApplication_과 같이 밑줄이 있는 이름 들은 파스칼의 이름 규칙에 따라 다른 예약어와 겹칠 가능성이 있는 이름을 델파이가 변경한 것이라는 것을 알아두도록 하자.

자 그럼, 이제 결과를 보도록 하자. 이 프로그램을 컴파일하고 실행한 후, 버튼을 차례로 누르면 다음과 같은 결과를 볼 수 있을 것이다.



아마도 이 결과는 시스템에 따라서 조금씩은 차이가 날 것이다. 결과를 어떻게 생각하는가? 솔직히 필자는 이보다 훨씬 더 차이가 날 것으로 생각했는데 의외로 큰 차이가 나지 않은 결과이다. 어쨌든 early 바인딩이 더 빠르고, 듀얼 인터페이스가 미세하게나마 가장 빠르다는 것은 확실하다.

여기서 수행 성능에 큰 차이가 나지 않은 이유는 우리가 수행한 메소드가 단순한 문자열을 이용하여 값을 설정하는 정도의 간단한 것이기 때문이다. 또한, 워드는 기본적으로 OLE 자동화를 비주얼 베이직을 염두에 두고 제작되었기 때문에, 대부분의 파라미터가 OleVariant 데이터 형을 사용하도록 되어 있어 early 바인딩에 의한 성능 향상을 대폭 증가시키지는 못한다. 그렇지만, 만약에 필요한 OLE 자동화 서버를 작성할 때 직접 듀얼 인터페이스로 구현하여 이를 사용할 때 late 바인딩과 early 바인딩을 이용한 클라이언트 어플리케이션을 비교해보면 수행 성능에 엄청난 차이가 존재한다는 것을 쉽게 느낄 수 있을 것이다.

정 리 (Summary)

이번 장에서는 OLE 가 처음 나올 때 중시하던 복합 문서를 위한 OLE 도큐먼트(액티브 도큐먼트)를 OleContainer 컴포넌트를 이용해서 지원하는 방법과 OLE 자동화 컨트롤러 어플리케이션을 제작하여 여러가지 OLE 자동화 서버를 제어하는 방법에 대해서 알아보았다.

특히 델파이에서 OLE 자동화를 이용해서 오피스 객체를 사용하는 것은, 실제로 마이크로소프트에서 발표한 오피스 개발자 에디션(Microsoft Office Developer Edition)의 성능과 같거나, 델파이의 성능을 이용하면 더 나은 수행 성능을 보여주는 것으로 알려져 있다. 그러므로, 이를 잘 활용한다면 보다 통합된 어플리케이션을 개발하는데 커다란 도움이 될 것이다. 다음 장에서는 OLE 자동화 서버를 작성하는 방법과 타입 라이브러리와 듀얼 인터페이스에 대해서 보다 심도 있게 다룰 것이다.