컴포넌트 제작의 깊은 곳

(Advanced Components Writing Techniques)

그리드 컴포넌트의 제작

비교적 복잡하면서도 유용하게 사용할 수 있는 컴포넌트가 그리드 컴포넌트 들이다. 델파이는 그리드를 작성하기 쉽도록 TCustomGrid 라는 기초 컴포넌트를 제공하고 있다. 그러면, 이 컴포넌트를 바탕으로 해서 달력 컴포넌트를 하나 만들어 보도록 하자.
TCustomGrid 컴포넌트를 상속하도록 하고, 컴포넌트의 이름을 TSampleCalendar 라고 하자. 이 컴포넌트 역시 Samples 페이지에 등록하도록 한다.

- 상속된 프로퍼티 publish 와 초기값 변경

TCustomGrid 는 추상적인 그리드 컴포넌트이기 때문에, 많은 수의 protected 프로퍼티를 제공한다. 그러므로, 다음과 같이 필요한 프로퍼티를 published 섹션에 선언해 주도록 한다.

```
published
  property Align;
  property BorderStyle;
  property Color;
  property Ctl3D;
  property Font;
  property GridLineWidth;
  property ParentColor;
  property ParentFont;
  property OnClick;
  property OnDblClick;
  property OnDragDrop;
  property OnDragOver;
  property OnEndDrag;
  property OnKeyDown;
  property OnKeyPress;
  property OnKeyUp;
```

```
  end;
```

달력은 기본적으로 행과 열의 수가 고정되어 있기 때문에, ColCount 나 RowCount 같은 프로퍼티를 publish 할 필요가 없다. 그렇지만, 이런 초기 값을 constructor 에서 설정할 필요가 있다.

```
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  ColCount := 7;
  RowCount := 7;                                  //머릿글 포함
  FixedCols := 0;
  FixedRows := 1;                                 //요일 표시 하는 행
  ScrollBars := ssNone;
  Options := Options - [goRangeSelect] + [goDrawFocusSelected];  //범위 선택을 할 수 없다.
end;
```

- 셀 크기 조절과 내부 채우기

사용자나 어플리케이션이 컨트롤의 크기를 변경하면, 윈도우는 WM_SIZE 메시지를 받게 된다. 컴포넌트는 이 메시지에 맞추어 이미지를 다시 그릴 필요가 있다. 이를 위해서 WM_SIZE 메시지에 반응하는 메시지 처리 메소드를 추가할 필요가 있다.
먼저 protected 섹션에 다음과 같이 메시지 처리 메소드를 선언한다.

```
procedure WMSize(var Message: TWMSize); message WM_SIZE;
```

그리고, 다음과 같이 구현한다.

```
procedure TSampleCalendar.WMSize(var Message: TWMSize);
var
  GridLines: Integer;
begin
  GridLines := 6 * GridLineWidth;                                  //전체 줄의 크기
  DefaultColWidth := (Message.Width - GridLines) div 7;            //새로운 셀의 폭
  DefaultRowHeight := (Message.Height - GridLines) div 7;                  //새로운 셀의 높이
end;
```

그리드 컨트롤은 셀 단위로 내부를 채우게 된다. 그러므로, 달력 컴포넌트라면 각각의 셀마다 날짜를 계산해서 그려야 한다. 그리드 셀은 DrawCell 가상 메소드를 호출하여 그리게 되므로, 이 메소드를 오버라이드해야 한다.
먼저 첫번째 행의 요일을 해당되는 열에 그리도록 한다.
protected 섹션에 DrawCell 메소드를 선언하고, 이를 다음과 같이 구현한다.

```
procedure DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState); override;

procedure TSampleCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect;
  AState: TGridDrawState);
begin
  if ARow = 0 then
    Canvas.TextOut(ARect.Left, ARect.Top, ShortDayNames[ACol + 1]);
end;
```

- 날짜 계산

달력 컨트롤을 만들 때 중요한 것은 사용자나 어플리케이션이 연, 월, 일을 설정할 수 있는 기전을 제공해야 한다는 것이다. 델파이는 날짜와 시간은 TDateTime 데이터 형의 변수에 저장한다. 그러므로, 날짜 자체는 TDateTime 데이터 형 변수에 저장하되 Day, Month, Year 프로퍼티를 제공하여 쉽게 날짜를 조정할 수 있도록 해야 한다.

1. 날짜의 저장

날짜를 저장하기 위해서는 먼저 TDateTime 데이터 형의 필드 변수를 하나 선언하고, 이를 constructor 에서 현재의 날짜를 얻어서 저장한다.

```
  private
    FDate: TDateTime;

constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  ...
  FDate := Date;
```

```
end;
```

그리고, 런타임에서 접근할 수 있는 프로퍼티를 하나 선언한다.

```
  TSampleCalendar = class(TCustomGrid)
  private
    procedure SetCalendarDate(Value: TDateTime);
  public
    property CalendarDate: TDateTime read FDate write SetCalendarDate;
  ...

procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;
  Refresh;                    //화면을 업데이트
end;
```

2. Day, Month, Year 프로퍼티 선언

먼저 다음과 같이 프로퍼티를 선언한다. 특이하게 보일지도 모르겠지만 하나의 날짜에 의해서 이들 프로퍼티는 동시에 변경되고, 설정되므로 이들을 각각의 Get, Set 메소드로 구현할 필요가 없다. 이렇게 중복되는 부분을 같은 접근 메소드(access method)를 이용해서 사용할 때에는 index 를 사용하면 유용하다.

```
  public
    property Day: Integer index 3 read GetDateElement write SetDateElement;
    property Month: Integer index 2 read GetDateElement write SetDateElement;
    property Year: Integer index 1 read GetDateElement write SetDateElement;
  ...
```

그러면 GetDateElement, SetDateElement 메소드를 다음과 같이 선언하고 구현하면 된다.

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    function GetDateElement(Index: Integer): Integer
```

```
    procedure SetDateElement(Index: Integer; Value: Integer);
  ...

function TSampleCalendar.GetDateElement(Index: Integer): Integer;
var
  AYear, AMonth, ADay: Word;
begin
  DecodeDate(FDate, AYear, AMonth, ADay);
  case Index of
    1: Result := AYear;
    2: Result := AMonth;
    3: Result := ADay;
    else Result := -1;
  end;
end;

procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
var
  AYear, AMonth, ADay: Word;
begin
  if Value > 0 then
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);
    case Index of
      1: AYear := Value;
      2: AMonth := Value;
      3: ADay := Value;
      else Exit;
    end;
    FDate := EncodeDate(AYear, AMonth, ADay);
    Refresh;
  end;
end;
```

그렇게 어려운 코드는 아니다. EncodeDate 와 DecodeDate 를 이용하면 TDateTime 데이터 형과 Year, Month, Day 의 정수형 사이를 자유롭게 변환할 수 있다는 것을 알아두면 된

다.

3. 날짜 표시하기

날짜를 달력에 그릴 때에는 달마다 일수가 다르고, 주어진 연도가 윤년인지 여부를 고려해야 한다. IsLeapYear 함수를 이용하면 해당 연도가 윤년인지 알아볼 수 있으며, SysUtils.pas 유닛의 MonthDays 배열을 이용하면 해당 달의 날짜 수를 알아낼 수 있다.
일단 윤년 정보와 그 달의 날짜 수를 알게 되면, 그리드에서 각 날짜의 위치를 계산할 수 있게 된다.
먼저 그 달에서 첫번째 날짜가 적당한 요일에 위치하는지를 계산할 때 사용할 오프셋을 저장할 필드와 필드 값을 업데이트할 메소드를 선언하고 이를 다음과 같이 구현한다.

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FMonthOffset: Integer;
  …
  protected
    procedure UpdateCalendar; virtual;
  end;
…

procedure TSampleCalendar.UpdateCalendar;
var
  AYear, AMonth, ADay: Word;
  FirstDate: TDateTime;                          //그 달의 첫번째 날짜
begin
  if FDate <> 0 then
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);
    FirstDate := EncodeDate(AYear, AMonth, 1);
    FMonthOffset := 2 - DayOfWeek(FirstDate);    //오프셋을 초기화한다.
  end;
  Refresh;                                       //달력을 다시 그린다.
end;
```

그리고, constructor 와 SetCalendarDate, SetDateElement 메소드에 UpdateCalendar 메소드를 호출하도록 수정한다.

```
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  ...
  UpdateCalendar;
end;

procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;
  UpdateCalendar;
end;

procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
begin
  ...
    FDate := EncodeDate(AYear, AMonth, ADay);
    UpdateCalendar;
  end;
end;
```

이제는 셀의 행과 열의 번호를 넘겨 주면, 이 위치가 그 달의 몇 번째 날인지 계산하는 메소드를 다음과 같이 구현한다.

```
function TSampleCalendar.DayNum(ACol, ARow: Integer): Integer;
begin
  Result := FMonthOffset + ACol + (ARow - 1) * 7;
  if (Result < 1) or (Result > MonthDays[IsLeapYear(Year), Month]) then
    Result := -1;                          //셀의 위치의 날짜가 유효하지 않다.
end;
```

이들 메소드를 이용하면, 날짜의 위치를 알 수 있다. 그러면, DrawCell 메소드를 다음과 같이 구현하여 날짜를 표시하도록 한다.

```
procedure TSampleCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState);
var
  TheText: string;
  TempDay: Integer;
begin
  if ARow = 0 then                                          //헤더이면 ...
    TheText := ShortDayNames[ACol + 1]                      //요일을 표시한다.
  else
  begin
    TheText := '';                                          //일단 셀을 비운다.
    TempDay := DayNum(ACol, ARow);                          //셀의 위치에 따른 날짜를 구한다.
    if TempDay <> -1 then TheText := IntToStr(TempDay);  //결과가 유효할 때만 사용 !
  end;
  with ARect, Canvas do
    TextRect(ARect, Left + (Right - Left - TextWidth(TheText)) div 2,
      Top + (Bottom - Top - TextHeight(TheText)) div 2, TheText);    //셀의 중앙에 날짜 표시
end;
```

4. 현재 날짜의 선택

이제 선택된 날짜를 표시하도록 해보자. 이를 위해서는 UpdateCalendar 메소드를 Refresh 메소드를 호출하기 전에 Row, Column 프로퍼티를 설정하도록 수정해야 한다.

```
procedure TSampleCalendar.UpdateCalendar;
begin
  if FDate <> 0 then
  begin
    ...
    Row := (ADay - FMonthOffset) div 7 + 1;
    Col := (ADay - FMonthOffset) mod 7;
  end;
  Refresh;
end;
```

- 날짜 변경

날짜 사이를 이동할 때에는 화살표 키를 이용하여 이동할 수도 있고, 마우스를 클릭할 수도 있다. 이들을 모두 처리해 주어야 하는 것이 중요하다.
선택된 날짜를 변경해 주어야 할 것이다. 기본적으로 그리드는 화살표 키를 누르거나, 마우스를 클릭할 때 선택된 셀을 옮기는 것을 처리하게 되어 있다. 그렇지만, 달력의 경우 약간의 수정이 필요하다. 이를 위해, 그리드의 Click 메소드를 오버라이드해야 한다.

```
procedure TSampleCalendar.Click;
var
  TempDay: Integer;
begin
  inherited Click;
  TempDay := DayNum(Col, Row);
  if TempDay <> -1 then Day := TempDay;
end;
```

날짜가 변경되었을 때 이벤트를 발생시킬 수 있다면 좋을 것이다. 그러면, OnChange 이벤트를 추가하도록 하자. 먼저 다음과 같이 프로시저 형을 선언하고 이벤트를 추가하자.

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FOnChange: TNotifyEvent;
  protected
    procedure Change; dynamic;
  ...
  published
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
  ...
```

그리고, Change 메소드를 다음과 같이 구현한다.

```
procedure TSampleCalendar.Change;
begin
  if Assigned(FOnChange) then FOnChange(Self);
end;
```

그리고, SetCalendarDate 와 SetDateElement 메소드에 Change 메소드를 호출하는 부분을 추가한다.

```
procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;
  UpdateCalendar;
  Change;
end;

procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);

begin
    ...
    FDate := EncodeDate(AYear, AMonth, ADay);
    UpdateCalendar;
    Change;
  end;
end;
```

이렇게 만든 달력 컴포넌트는 비어 있는 셀이 존재하게 되는데, 날짜가 찍혀 있지 않은 셀을 선택할 경우 선택이 되지 않도록 해야 할 것이다. 이를 위해서는 SelectCell 메소드를 오버라이드하여 구현해야 한다.

```
function TSampleCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if DayNum(ACol, ARow) = -1 then Result := False
  else Result := inherited SelectCell(ACol, ARow);
end;
```

이것으로 간단한 달력 컴포넌트를 그리드를 이용해서 작성해 보았다. 비록 볼품도 없고, 단순한 컴포넌트 이지만, 그리드를 이용하여 여러가지 컴포넌트를 개발하는 데에는 참고가 될만할 것이다.

데이터 인식 컨트롤의 제작

델파이는 데이터 소스와 연결해서 데이터를 보여주고, 편집할 수 있는 여러 가지 컨트롤 들을 제공한다. 이런 데이터 인식 컨트롤을 제작하기 위해서는 몇 가지 고려해야 할 공통적인 것들이 있다. 단순한 단일 필드와 연결된 데이터 인식 컨트롤을 만들 수도 있고, 여러 레코드를 관리하는 컨트롤을 만들 수도 있다.
컨트롤과 데이터베이스를 연결하는 것은 데이터 링크 클래스를 통해서 다루어진다. 데이터 링크 클래스 중에서 단일 필드와 연결할 때에는 보통 TFieldDataLink 클래스를 사용한다. 물론 전체 테이블과 연결할 수 있는 데이터 링크 클래스도 있다. 여기에서는 필자가 작성한 TDBTree 컴포넌트를 바탕으로 설명하겠지만, 지면 관계상 570 라인 가까이에 이르는 전체 소스를 설명할 수는 없고 데이터 인식 컨트롤을 구현하는 방법을 중심으로 설명하도록 한다. 그 밖에도 소스를 분석하면 객체를 저장하고, 이를 활용하는 등의 여러가지 테크닉을 배울 수 있을 것이다.

- TFieldDataLink 클래스

TFieldDataLink 클래스는 컨트롤과 데이터 소스 사이의 연결을 구성한다. 그러므로, 데이터 인식 컨트롤을 만들 때에는 이 클래스에 대한 이해가 필수적이라고 할 수 있다.
주요 메소드와 프로퍼티를 소개하면 다음과 같다.

메소드/프로퍼티	내 용
Edit	편집 가능한 레코드를 편집 모드로 만든다. 데이터 소스가 읽기 전용이면 False를 반환한다.
Modified	컨트롤이 데이터를 변경할 때 호출해야 한다. 데이터 링크는 레코드가 바뀌지 않았던 데이터 소스를 제공하며, 변경된 데이터를 요구할 때 OnUpdateData 이벤트 핸들러를 호출한다.
Reset	데이터 소스의 데이터를 변경하지 않거나, 새로운 데이터를 검색할 때 호출한다.
CanModify	읽기 전용으로 필드의 값을 수정할 수 있는지 여부를 결정한다.
Control	읽기 전용으로 연결된 데이터 컨트롤을 가리킨다. 데이터 링크가 입력 포커스를 받는 컨트롤을 결정하기 위해 Control 속성을 사용한다.
Editing	읽기 전용으로 데이터 소스의 State 속성이 dsEdit, dsInsert, dsSetKey 중 하나인지 나타낸다.
Field	읽기 전용으로 연결된 필드 클래스를, 연결된 것이 없으면 nil 을 반환한다.
FieldName	데이터베이스 필드 이름을 지정한다.
OnActiveChange	데이터 소스의 Active 프로퍼티가 변경될 때 호출되는 이벤트이다.
OnDataChange	데이터 레코드가 변한 후에 호출되는 이벤트이다. 모든 데이터 인식 컨트롤은

	여기에 대한 핸들러를 구현해야 한다. 이 핸들러가 없으면 컨트롤이 필드로부터 데이터를 검색할 때 알 수 있는 방법이 없다.
OnEditingChange	데이터 소스가 편집모드로 들어가거나 나올 때 호출되는 이벤트이다.
OnUpdateData	컨트롤로부터 변경된 데이터를 소스가 요구할 때 호출되는 이벤트이다. 여기에 대한 핸들러는 연결된 필드 구성요소의 데이터를 저장한다.
UpdateRecord	데이터 소스가 데이터 컨트롤에 의해 수정된 데이터를 가진 현재 레코드의 복사본을 수정하려 할 때 호출된다. 데이터 컨트롤에서 포커스를 잃을 때 호출된다.
BufferCount	사용가능한 레코드 버퍼의 개수를 반환한다. 그리드와 같은 다중 레코드 컨트롤을 구현할 때에 이를 이용하여 사용자가 한 번에 많은 수의 값을 편집할 때 성능을 향상시킬 수 있다. 디폴트 값은 1 이다.
DataSet	읽기 전용으로 데이터 소스로 연결된 데이터 세트를 되돌려 준다.
DataSource	연결된 데이터 소스를 지정한다.
RecordCout	현재 데이터 링크에 의해 버퍼되어진 레코드 수를 반환한다.

데이터 인식 컨트롤은 자체적인 데이터 링크 클래스를 가지고 있으며, 컨트롤이 데이터 링크 객체를 생성하고 파괴하는 책임을 가지고 있다.

```
type
  TDBTree = class(TCustomTreeView)
  private
    FDataTitleLink: TFieldDataLink;
    ...
  end;
```

모든 데이터 인식 컨트롤은 데이터를 컨트롤에 제공하는 데이터 소스를 지정하는 DataSource 프로퍼티를 가지고 있다. 또한, 연결할 데이터 소스의 필드를 지정하는 DataField 프로퍼티를 가지고 있다. 물론, 보통의 컨트롤은 하나의 필드와 데이터 컨트롤을 연결하게 되므로 이렇게 DataField 프로퍼티로 연결하면 되지만, 컨트롤에 따라서는 여러 개의 필드를 설정해야 하는 경우도 있을 것이다. 보통의 경우에는 DataField 프로퍼티를 하나만 가지면 되지만, 트리 구조를 데이터베이스에 저장하기 위해서는 자신의 ID 와 Parent, Index 를 저장할 3 개의 필드와 타이틀의 내용을 저장할 필드가 필요하다. 물론 대표적인 필드는 Title 을 저장할 필드를 하나만 지정하도록 published 프로퍼티로 지정하고 이를 DataField 프로퍼티로 사용하고, 나머지는 런타임에서 접근할 수 있도록 public 섹션에 선언해서 사용하도록 한다. 이들 프로퍼티를 접근하고, 설정할 접근 메소드를 설정하고 구현할 때에는 데이터 링크 클래스를 이용한다.

```
  TDBTree = class(TCustomTreeView)
  private
    FIDFieldName: string;
    FParentFieldName: string;
    FIndexFieldName: string;
    function GetDataTitleField: string;
    function GetDataSource: TDataSource;
    function GetTitleField: TField;
    function GetIDField: TField;
    function GetParentField: TField;
    function GetIndexField: TField;
    function GetDataSet: TDataSet;
    procedure SetDataTitleField(const Value: string);
    procedure SetDataSource(Value: TDataSource);
    ...
  public
    property TreeDataSet: TDataSet read GetDataSet;
    property IDField: TField read GetIDField;
    property ParentField: TField read GetParentField;
    property IndexField: TField read GetIndexField;
    property TitleField: TField read GetTitleField;
    ...
  published
    property DataSource: TDataSource read GetDataSource write SetDataSource;
    property DataField: string read GetDataTitleField write SetDataTitleField;
    ...
end;

function TDBTree.GetDataSource: TDataSource;
begin
  Result := FDataTitleLink.DataSource;
end;

procedure TDBTree.SetDataSource(Value: TDataSource);
begin
  FDataTitleLink.DataSource := Value;
```

```
  if Value <> nil then Value.FreeNotification(Self);
end;

function TDBTree.GetDataSet: TDataSet;
begin
  Result := FDataTitleLink.DataSet;
end;

function TDBTree.GetDataTitleField: string;
begin
  Result := FDataTitleLink.FieldName;
end;

procedure TDBTree.SetDataTitleField(const Value: string);
begin
  FDataTitleLink.FieldName := Value;
end;

procedure TDBTree.SetIDField(FieldName: string);
begin
  if TreeDataSet.FindField(FieldName) <> nil then
    FIDFieldName := FieldName;
end;

procedure TDBTree.SetParentField(FieldName: string);
begin
  if TreeDataSet.FindField(FieldName) <> nil then
    FParentFieldName := FieldName;
end;

procedure TDBTree.SetIndexField(FieldName: string);
begin
  if TreeDataSet.FindField(FieldName) <> nil then
    FIndexFieldName := FieldName;
end;
```

```
function TDBTree.GetTitleField: TField;
begin
  Result := FDataTitleLink.Field;
end;

function TDBTree.GetIDField: TField;
begin
  if TreeDataSet.FindField(FIDFieldName) <> nil then
    Result := TreeDataSet.FieldByName(FIDFieldName)
  else MessageDlg('해당되는 ID 필드가 없습니다 !', mtError, [mbOK], 0);
end;

function TDBTree.GetParentField: TField;
begin
  if TreeDataSet.FindField(FParentFieldName) <> nil then
    Result := TreeDataSet.FieldByName(FParentFieldName)
  else MessageDlg('해당되는 Parent 필드가 없습니다 !', mtError, [mbOK], 0);
end;

function TDBTree.GetIndexField: TField;
begin
  if TreeDataSet.FindField(FIndexFieldName) <> nil then
    Result := TreeDataSet.FieldByName(FIndexFieldName)
  else MessageDlg('해당되는 Index 필드가 없습니다 !', mtError, [mbOK], 0);
end;
```

그리고, constructor 와 destructor 에서 데이터 링크 객체를 생성하고, 파괴하는 것이 중요하다.

```
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    ...

constructor TDBTree.Create(AOwner: TComponent);
begin
```

```
  inherited Create(AOwner);
  inherited ReadOnly := True;
  ControlStyle := ControlStyle + [csReplicatable];
  FDataTitleLink := TFieldDataLink.Create;
  FDataTitleLink.Control := Self;
  FIDFieldName := 'ID';
  FParentFieldName := 'Parent';
  FIndexFieldName := 'Index';
  FMemorySaving := True;
end;

destructor TDBTree.Destroy;
var
  i: integer;
begin
  i := 0;
  FDataTitleLink.Free;
  inherited Destroy;
end;
```

데이터 링크를 이용하여 필드와 데이터 소스를 설정하는 것까지 했으면, 절반은 끝난 셈이다. 이제는 데이터의 변화를 컨트롤에 반영하도록 구현하는 것이 중요하다. 보통의 경우 이 때에는 데이터 링크 클래스의 OnDataChange 이벤트를 이용한다. 데이터 소스에서 데이터의 변화를 감지하면, 데이터 링크 객체는 OnDataChange 이벤트 핸들러를 호출하게 된다. 그러므로, DataChange 메소드를 구현하고, 이 메소드를 OnDataChange 이벤트에 연결하는 것이 중요하다. 그러나, DBTree 에서는 이를 이용하지 않고 따로 public 메소드 들을 제공하여 구현하였다. 표준적인 방법이 아니기 때문에 따로 설명하지는 않는다. 대신 일반적인 데이터 인식 컨트롤을 구현할 때에는 앞에서 설명한 방법을 사용하는데, 여기에서는 TDBEdit 컨트롤이 어떻게 구현되었는지 소개한다.

```
  TDBEdit = class(TCustomMaskEdit)
  private
    ...
    procedure DataChange(Sender: TObject);

procedure TDBEdit.DataChange(Sender: TObject);
```

```
begin
  if FDataLink.Field <> nil then
  begin
    if FAlignment <> FDataLink.Field.Alignment then
    begin
      EditText := '';  {forces update}
      FAlignment := FDataLink.Field.Alignment;
    end;
    EditMask := FDataLink.Field.EditMask;
    if not (csDesigning in ComponentState) then
    begin
      if (FDataLink.Field.DataType = ftString) and (MaxLength = 0) then
        MaxLength := FDataLink.Field.Size;
    end;
    if FFocused and FDataLink.CanModify then
      Text := FDataLink.Field.Text
    else
    begin
      EditText := FDataLink.Field.DisplayText;
      if FDataLink.Editing and FDataLink.FModified then
        Modified := True;
    end;
  end else
  begin
    FAlignment := taLeftJustify;
    EditMask := '';
    if csDesigning in ComponentState then
      EditText := Name else
      EditText := '';
  end;
end;
```

데이터 인식 컨트롤의 내용을 편집했을 때에는 필드 데이터 링크 객체를 업데이트해야 데이터 세트에 변화를 반영할 수 있다. 데이터가 변했을 때에는 OnDataChange 이벤트에서 처리하듯이, 데이터 컨트롤이 변경 되었을 때에는 OnUpdateData 이벤트에서 처리해야 한다. 그러므로, UpdateData 메소드를 구현하고 OnUpdateData 이벤트에 UpdateData 메소

드를 연결한다. TDBEdit 컨트롤은 다음과 같이 구현되어 있다.

```
  TDBEdit = class(TCustomMaskEdit)
  private
    ...
    procedure UpdateData(Sender: TObject);

procedure TFieldDataLink.UpdateData;
begin
  if FModified then
  begin
    if (Field <> nil) and Assigned(FOnUpdateData) then FOnUpdateData(Self);
    FModified := False;
  end;
end;
```

DataChange, UpdateData 메소드에서 중요한 것은 이렇게 구현된 메소드를 constructor 에서 이벤트 핸들러를 대입하는 코드를 추가해야 한다는 것이다.

```
constructor TDBEdit.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  inherited ReadOnly := True;
  ControlStyle := ControlStyle + [csReplicatable];
  FDataLink := TFieldDataLink.Create;
  FDataLink.Control := Self;
  FDataLink.OnDataChange := DataChange;
  FDataLink.OnEditingChange := EditingChange;
  FDataLink.OnUpdateData := UpdateData;
end;
```

데이터 인식 컨트롤의 Change 메소드는 새로운 값이 설정될 때마다 호출된다. Change 메소드는 OnChage 이벤트 핸들러가 존재하면 이를 호출한다. 그러므로, 컴포넌트 사용자는 OnChage 이벤트 핸들러를 이용하여 데이터 변화에 대한 여러가지 처리를 하는 코드를 입력할 수 있다. 설정 값이 변경되면 연결된 데이터 세트는 반드시 변화에 대해 반응하도록 해야하는데, 이를 위해서는 Change 메소드를 오버라이드하여야 한다. DBTree 의 경우 다

음과 같이 구현한다.

```
  TDBTree = class(TCustomTreeView)
  ...
  protected
    procedure Change(Node: TTreeNode); override;
     ...

procedure TDBTree.Change(Node: TTreeNode);
begin
  inherited Change(Node);
  GotoID(Node);
end;

procedure TDBTree.GotoID(ANode: TTreeNode);
begin
  if ANode <> nil then
    TreeDataSet.Locate(FIDFieldName, PID(ANode.Data)^, []);
end;
```

여기서 Change 메소드는 사용자가 트리 노드를 선택하여 변경한 경우이므로 해당 트리 노드에 해당되는 레코드로 옮겨가야 한다. 이를 처리하는 메소드가 GotoID 이며, GotoID 메소드는 DataSet 의 Locate 메소드를 이용하여 구현한다.
이것으로 데이터 인식 컨트롤의 가장 핵심적인 부분에 대한 구현 방법에 대해서 소개하였다. 예제로 보여준 DBTree 의 경우 일반적인 데이터 인식 컨트롤과는 달리 여러 개의 레코드를 보여주고, 이들 사이를 옮겨 다니는 처리를 해야 하기 때문에 다소 복잡하다. 그래서, 사실 가장 전형적인 예제로서는 부적절한 면이 있지만, 어떤 식으로 구현하는지에 대해서는 감을 잡았을 것으로 믿는다. 델파이 소스 중에서 DBCtrls.pas 유닛에 보면 좋은 예제가 많으므로 이들을 참고로 할 것을 권하고 싶다.
이번 장에서 같이 제공하는 DBTree 컴포넌트는 인터넷에서 구할 수 있는 여러가지 DBTreeView 컴포넌트와 조금은 다른 방식으로 간단하게 구현한 컴포넌트이다. 물론 인터넷에서 구한 컴포넌트보다 미약한 부분도 많지만, 사용하기에는 더 쉽고 간단하다.
컴포넌트를 제작할 때 아쉬운 점은 필자가 DBTree 컴포넌트를 처음 만들 때에는 인터넷에서 없었기 때문에 96 년 가을에 사용을 위해서 제작한 것인데, 몇 달 지나지 않아서 비슷한 컴포넌트 들이 공개되었다. 이처럼 기다리면 비슷한 것들이 나오는 것을 보면 누구나 생각하고 필요로 하는 내용은 비슷한가 보다 (^^;). DBTree 컴포넌트의 소스와 함께 제공되는

Readme.txt 파일을 읽어 보면 DBTree 컴포넌트의 사용법에 대해서 적어 놓았으므로 활용해서 좋은 어플리케이션을 만들어보기 바란다.

컴포넌트 제작에 유용한 프로퍼티/메소드

지금까지 언급한 여러가지 컴포넌트 제작에 대한 내용 이외에도 조금은 고급스러운 컴포넌트로 만들기 위해서는 몇 가지 알아야 할 프로퍼티와 메소드가 있다. 여기서 이들에 대해 모두 다룰 수는 없지만 그래도 자주 쓰이는 것들을 중심으로 소개하고자 한다.

- 컴포넌트 State 의 검사

TComponent 컴포넌트의 ComponentState 프로퍼티는 컴포넌트의 현재 상태를 파악하는데 사용할 수 있다. 보통 컴포넌트를 쉐어웨어로 배포할 때, 디자인 타임과 런타임을 구별해서 제한을 두는 컴포넌트 등을 제작할 때 사용된다.
ComponentState 프로퍼티 값에는 다음과 같은 것들이 있다.

값	설 명
csAncestor	컴포넌트가 조상 폼에 나타난 경우 설정된다. csDesigning 이 같이 설정
csDesigning	컴포넌트가 디자인 모드에 있을 경우 설정된다.
csDestroying	컴포넌트가 파괴되려 할 때 설정된다.
csFixups	컴포넌트가 다른 폼의 컴포넌트와 연결되어 있으나, 연결된 컴포넌트가 아직 로드되지 않은 경우 설정된다.
csLoading	Filer 객체에서 컴포넌트를 로딩하는 중이면 설정된다.
csReading	스트림에서 프로퍼티를 읽어들이는 도중이면 설정된다.
csUpdating	컴포넌트의 변경 사항을 업데이트하는 도중일 때. csAncestor 가 설정된 경우에만 사용된다.
csWriting	프로퍼티 값을 스트림에 기록하는 도중이면 설정된다.

예를 들어, 디자인 타임에서 등록하라는 메시지를 다음과 같이 보여줄 수 있다.

```
if (csDesigning in ComponentState) then ShowMessage('등록하세요 !');
```

- 컨트롤의 State 프로퍼티

TControl 클래스에는 런타임에서의 컨트롤의 상태를 반영하는 ControlState 라는 프로퍼티가 정의되어 있다. 이 프로퍼티는 세트로 정의되어 있으며, 다음과 같은 값을 가질 수 있

다.

값	의 미
csLButtonDown	왼쪽 마우스 버튼을 누르고 아직 놓지않은 상태
csClicked	클릭 이벤트가 발생할 때 설정된다.
csPalette	WM_PALETTECHANGED 메시지를 받았다.
csReadingState	컨트롤이 state 정보를 스트림에서 읽고 있다.
csAlignmentNeeded	컨트롤이 재정렬될 필요가 있을 때
csFocusing	어플리케이션이 컨트롤에 포커스를 주려고 한다.
csCreating	컨트롤이 생성되고 있다.
csPaintCopy	컨트롤이 복제되어 그려지고 있다.

- 컨트롤의 Style 설정

ControlStyle 프로퍼티는 컨트롤의 특징을 결정하는 세트 프로퍼티이다. 이 프로퍼티는 주로 컴포넌트의 constructor 에서 설정하게 된다. 다음과 같은 값들과 의미를 가질 수 있다.

값	의 미
csAcceptsControls	디자인 타임에서 컨트롤을 드롭하면 parent 가 될 수 있다.
csCaptureMouse	마우스를 클릭했을 때 이벤트를 캡쳐할 수 있다.
csDesignInteractive	디자인 타임에서 오른쪽 마우스 버튼을 클릭하면 왼쪽 버튼을 클릭하는 것으로 매핑하여 컨트롤을 다룬다.
csClickEvents	마우스의 클릭을 받아들인다.
csFramed	컨트롤이 3D 프레임을 가진다.
csSetCaption	Name 프로퍼티가 변경될 때, 자동으로 캡션이 변경된다.
csOpaque	컨트롤이 클라이언트 영역을 완전히 채운다.
csDoubleClicks	더블 클릭 메시지를 처리한다.
csFixedWidth	컨트롤의 폭이 변경되지 않는다.
csFixedHeight	컨트롤의 높이가 변경되지 않는다.
csNoDesignVisible	디자인 타임에 컨트롤이 보이지 않는다.
csReplicatable	컨트롤을 DBCtrlGrid 에 드롭할 수 있다.
csNoStdEvents	마우스, 키보드, 클릭과 같은 표준 이벤트를 무시한다.

- Loaded 메소드

Loaded 메소드는 컨트롤이 생성될 때 프로퍼티를 DFM 파일에서 읽어온 뒤, 컴포넌트가 보여지기 전에 프로퍼티 값에 대한 생성 프로세스를 처리할 기회를 얻을 수 있다. 보통, 여기에서 저장된 프로퍼티 값에 대해 private, public 데이터 필드의 값을 초기화하거나 제작된 컴포넌트에 대한 여러가지 설정을 할 기회가 된다.

- Notification 메소드

Notification 메소드는 TComponent 클래스에서 제공되는 가상 메소드로, 델파이 IDE 가 컴포넌트가 폼에 드롭되거나 제거될 때 호출하는 메소드이다. Notification 메소드는 다음과 같이 선언되어 있다.

```
procedure Notification(AComponent: TComponent; Operation: TOperation); virtual;
```

Notification 메소드의 첫번째 파라미터는 폼에 추가되거나 삭제되는 컴포넌트를 나타내며, 두번째 파라미터는 opInsert, opRemove 라는 값을 가질 수 있다. 이 메소드는 다른 컴포넌트가 폼에 추가되거나 삭제될 때 특정 작업을 해야할 때 사용된다. 예를 들어, TBatchMove 의 Notification 메소드는 다음과 같이 구현되어 있다.

```
procedure TBatchMove.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if Operation = opRemove then
  begin
    if Destination = AComponent then Destination := nil;
    if Source = AComponent then Source := nil;
  end;
end;
```

먼저 상속된 Notification 메소드를 호출하고, 작업이 opRemove 인 경우에 제거되려는 컴포넌트의 종류가 Source, Destination 프로퍼티에 지정된 컴포넌트이면 이들 프로퍼티의 값을 nil 로 설정한다. 이렇게, 다른 컴포넌트와의 관련이 있도록 만들어진 경우에는 컴포넌트의 추가와 삭제에 따른 처리를 해주어야 한다. DBTree 컴포넌트에도 Notification 메소드를 다음과 같이 구현하고 있다.

```
procedure TDBTree.Notification(AComponent: TComponent;
```

```
  Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Operation = opRemove) and (FDataTitleLink <> nil) and
    (AComponent = DataSource) then DataSource := nil;
end;
```

- FreeNotification 메소드

FreeNotification 메소드는 Notification 메소드와 관련되어 사용되는 TComponent 의 메소드이다. Notification 메소드가 폼에서 컴포넌트가 추가되고 삭제될 때 호출되지만, Owner 가 다른 컴포넌트의 추가, 삭제에는 반응하지 못한다. 즉, 데이터 인식 컨트롤의 경우 데이터 모듈에 데이터 소스가 있고 폼에 컨트롤이 있는 경우와 같이 Owner 가 다른 경우에는 데이터 소스가 추가, 삭제될 때 Notificaiton 메소드를 호출하지 못한다.
이런 경우에는 연결될 컴포넌트를 FreeNotification 메소드를 이용해서 컴포넌트가 삭제될 때 Notification 메소드를 호출하도록 지정할 수 있다. 사소한 듯하지만 데이터 인식 컨트롤 등과 같이 연결된 컴포넌트가 있어야 하는 컴포넌트를 제작할 때에는 놓치기 쉬운 부분이다. TDBTree 컨트롤의 데이터 소스 컨트롤을 지정하는 SetDataSource 메소드의 구현 부분을 살펴 보자.

```
procedure TDBTree.SetDataSource(Value: TDataSource);
begin
  FDataTitleLink.DataSource := Value;
  if Value <> nil then Value.FreeNotification(Self);
end;
```

이렇게 데이터 소스를 연결할 때 파라미터로 Notification 을 받을 컴포넌트를 지정하면 된다. 여기서 Value 는 데이터 소스를 가리키며, 데이터 소스의 FreeNotification 메소드에 Self(여기서는 TDBTree)를 파라미터로 하여 호출함으로써 데이터 소스가 추가, 삭제될 때 TDBTree 의 Notification 메소드를 호출하게 된다.

- CreateWnd, DestroyWnd 메소드

CreateWnd 메소드는 윈도우 컨트롤이 처음 생성되거나, 윈도우가 프로퍼티의 변경에 따라 파괴되었다가 재생성 되어야 할 필요가 있을 때 호출된다. 그러므로, CreateWnd 메소드는 윈도우가 처음 생성될 때 추가적인 초기화 메시지를 넘겨주기 위해 오버라이드하여 구현한

다. CreateWnd 메소드는 처음에 CreateParams 메소드를 호출하여 윈도우 생성 파라미터를 초기화하고, CreateWindowHandle 을 호출하여 컨트롤에 대한 윈도우 핸들을 생성한다. 그리고, 새로운 크기의 윈도우를 적용하게 되며 WM_SETFONT 메시지를 이용해 Perform 메소드를 호출하여 컨트롤의 폰트를 설정한다.
DestroyWnd 메소드는 TWinControl 에서 추가한 메소드로, 컴포넌트가 파괴될 때 컨트롤의 윈도우 핸들과 연관된 디바이스 컨텍스트를 해제하기 위한 메소드이다. 이 메소드는 컴포넌트의 Destroy 메소드 이전에 호출된다. 만약 Destroy 메소드에서 윈도우 핸들을 이용한 작업을 하려고 하면, 'Window has no parent' 에러 메시지를 보게 된다. 이는 윈도우 핸들이 없어졌기 때문이다. 그러므로, 컴포넌트가 파괴될 때 윈도우 핸들을 가지고 디바이스 컨텍스트를 이용한 여러가지 작업을 하고자 할 때에는 DestroyWnd 메소드를 오버라이드하여 구현해 주어야 한다. 여기서 꼭 생각해야 할 것은 상속된 DestroyWnd 메소드를 호출하면 핸들이 없어지므로, 상속된 DestoryWnd 메소드는 가장 나중에 호출하는 것이 좋다.
VCL 소스 코드 중에서 비교적 간단한 TCustomEdit 의 예를 들어 설명 하겠다.

```
procedure TCustomEdit.CreateWnd;
begin
  FCreating := True;
  try
    inherited CreateWnd;
  finally
    FCreating := False;
  end;
  DoSetMaxLength(FMaxLength);
  Modified := FModified;
  if FPasswordChar <> #0 then
    SendMessage(Handle, EM_SETPASSWORDCHAR, Ord(FPasswordChar), 0);
  UpdateHeight;
end;
```

먼저 상속된 CreateWnd 메소드를 호출하여 초기 작업을 한다. 그리고, FMaxLength 필드의 값을 이용하여 길이를 결정하고 Modified 프로퍼티를 설정한다. 또한, FPasswordChar 의 값이 존재하면 패스워드 문자를 보여주기 위해 메시지를 넘겨주고 컨트롤의 높이를 맞추게 된다. 이와 같이 CreateWnd 메소드에서는 윈도우가 생성될 때 고려해야 할 여러가지를 설정한다.

```
procedure TCustomEdit.DestroyWnd;
```

```
begin
  FModified := Modified;
  inherited DestroyWnd;
end;

function TCustomEdit.GetModified: Boolean;
begin
  Result := FModified;
  if HandleAllocated then Result := SendMessage(Handle, EM_GETMODIFY, 0, 0) <> 0;
end;
```

여기서는 FModified 필드의 값을 Modified 프로퍼티의 값과 일치시킨다. 자동적으로 접근 메소드인 GetModified 가 호출되는데, 여기에서 Handle 을 이용하여 변경된 내용을 반영하게 된다.

정 리 (Summary)

이것으로 컴포넌트 제작에 대한 여러가지 방법 들에 대한 설명을 마치고자 한다. 이번 장에서 제작한 달력과 DBTree 컴포넌트는 Chap23.dpk 패키지 파일에 저장하였으니 직접 설치하고 익혀보기 바란다. 컴포넌트 제작은 델파이로 프로그래밍을 할 때 가장 필수적인 부분이고 중요한 부분이라고 할 수 있다. 나름대로 여러가지 부분을 다루었으나, 컨테이너 컴포넌트를 제작하는 방법이나 프로퍼티 에디터를 제작하는 방법 등의 고급 컴포넌트 제작에 대한 내용을 지면 관계상 모두 다루지 못한 아쉬움이 남는다. 아마도 컴포넌트를 제대로 제작하는 내용을 모두 담으려고 하면 적어도 그것만 가지고 1000 페이지는 넘는 책을 써야 할 것이다. 델파이의 소스를 열심히 분석해보면 얻는 것이 많을 것이라는 정도로 밖에 여기서는 말할 수 없는 것이 무척 아쉽다.
제 4 부에서 다룬 내용은 델파이에서 사용하게 되는 델파이 컴포넌트에 대한 개발 방법에 대해서 알아 보았다. 다음 장에서부터 다루게 되는 제 5 부의 내용은 델파이에 한정되지 않고, 다른 개발 도구에서도 사용할 수 있는 표준적인 개발 방법론인 DLL/DCOM/CORBA 에 대해서 알아보도록 할 것이다.