객체지향 언어로서의 오브젝트 파스칼

(Object Pascal As A OOP)

오브젝트 파스칼의 객체지향성을 몰라도 델파이 어플리케이션을 쉽게 만들 수 있다. 단순히 폼을 하나 만들고, 거기에 여러가지 컴포넌트 들을 추가하고, 이벤트 핸들러에 적당한 내용의 코드 들을 추가하면 그걸로 충분한 것이다. 그렇지만, 이것을 이해하면 델파이가 어떤 방법으로 작업을 처리하는지 이해할 수 있고, 자신만의 컴포넌트를 만들어 내거나, 비교적 커다란 프로젝트를 진행할 때에 커다란 도움을 받게 될 것이다.

OOP 의 기본 개념

프로그램은 데이터와 이를 처리하는 알고리즘으로 구성되어 있다고 하는 유명한 말이 있다. 그렇기 때문에, 프로그래밍 언어의 구조를 가만히 살펴보면 대부분의 경우 데이터를 다루는데 필요한 각종 정의 들과, 이들을 제어하기 위한 여러가지 문법으로 구성되어 있다는 것을 알 수 있다. 기본적으로 데이터의 표현은 '데이터 형 (data type)'이라는 개념으로 표현되는데, 이 데이터 형의 개념이 발전하면서 언어들이 다른 모습을 띠게 된다.
초기의 프로그래밍 언어는 정해진 데이터 형을 이용하였으며, 그 이후의 프로그래밍 언어(파스칼 등)에서는 레코드나 배열 등의 개발자가 정의해서 사용할 수 있는 데이터 형을 도입하였다. 그러다가, 이러한 데이터 형에 그 데이터를 다룰 때 쓰이는 작업과 결부시키게 되었는데, 이와 같이 기본적인 데이터와 이를 다루기 위한 메쏘드로 이루어진 데이터 형인 클래스가 OOP 의 기초가 되는 개념이다.
데이터 형과 함께 가장 중요한 프로그래밍 언어의 구성요소인 제어 구조를 보면, 초기의 프로그래밍 언어는 점프(goto)와 분기문(if-then, case 등), 순환문(for, while 등)으로 이루어진 제어문을 가지고 있었다. 여기에 서브루틴의 개념이 추가되어 일반적인 프로그래밍 언어의 전반적인 구조를 가지게 되었다. OOP 에서는 이러한 서브루틴을 데이터와 마찬가지로 추상화 하여, 이들을 클래스로 관리하게 된다.
OOP 에 있어서 가장 중요한 3 가지 개념은 캡슐화, 상속, 다형성으로 아래에 간단하게 설명하였다.

- 캡슐화 (Encapsulation)

OOP 의 가장 핵심적인 요소라고 할 수 있는 것이 캡슐화 개념으로, 데이터와 서브루틴을 추상화 한 것이다. 이 개념은 클래스로 구현되는데, 클래스란 특정 객체 그룹을 추상적으로 정의해 놓은 것이라고 생각하면 된다. 클래스는 일종의 형(type)의 정의로서 필드(클래스의

객체 상태를 나타내는 데이터)가 있고, 이에 대한 작업을 정의하는 메소드가 있다.
여기서 혼동하지 말아야 할 것은 클래스란 일종의 데이터 형이고, 객체는 클래스라는 데이터 형인 하나의 구체적인 예(인스턴스)라는 것이다. 예를 들어 설명하면 클래스란 객체를 만들기 위한 주형이며, 객체란 클래스라는 주형에서 만들어진 병과 같은 것이다.

- 상속 (Inheritance)

상속은 클래스를 처음부터 만들지 않고, 기존의 클래스를 기반으로 해서 새로운 클래스를 정의하는 것이다. 이때 기반이 되는 클래스를 부모 클래스, 만들어진 클래스를 서브 클래스라고 하며 서브 클래스는 부모 클래스로부터 필드와 메소드를 상속한다.
상속 개념이 실제로 쓰일 때에는 다음과 같은 특징을 가지고 있다.

1. 상속은 일반적인 클래스의 특수한 경우를 나타낼 때 쓰인다. 즉, 자동차라는 부모 클래스에서 기본적인 자동차의 필드와 메소드를 표현했다고 하면, 여기에서 상속받은 티코라는 서브 클래스는 기본적인 자동차 클래스의 필드와 메소드 외에 자신만의 필드와 메소드를 정의할 수 있다.
2. 반대로 여러가지로 특화된 클래스 들의 공통점을 가진 부모 클래스를 만들 수도 있다. 에를 들어, 기존에 학생이라는 클래스와 선생이라는 클래스를 가지고 있는 경우 이들의 공통적인 필드와 메소드를 정의하는 사람이라는 부모 클래스를 만들고, 공통의 요소를 공유하게 할 수 있다.
3. 실제로 상속 개념이 쓰일 때에는 중복되는 코드를 막을 수 있으며 (공통부분을 가지는 부모 클래스가 사용되므로), 복잡한 데이터의 개념을 쉽게 이해할 수 있게 된다.

- 다형성 (Polymorphism)

다형성이란 하나의 프로그램 변수를 가지고 서로 다른 클래스 객체를 참조하는 것을 말한다. 즉, 어떤 객체에 대해 작업을 할 때 객체의 type 에 적절하게 반응할 수 있다는 의미이다.

클래스와 객체 (Classes and Objects)

- 델파이 OOP 에 대한 문법적인 고찰

구체적인 예제에 들어가기 전에, 델파이의 OOP 에서 나오게 되는 중요한 개념들과 문법적인 특징에 대해서 알아보도록 하자.

1. 필드 (Fields)

필드는 객체에 속해있는 일종의 변수라고 이해하면 된다. 필드는 클래스 형을 포함해서 어떤 데이터 형으로도 선언해 사용할 수 있다. 필드를 선언하려면 단순히 변수를 선언하듯이 하면 된다. 예를 들어, 다음의 선언부는 TNumber 라는 컴포넌트를 선언하는데, 여기에는 Int 라는 정수 필드만을 가지고 있다.

```
type
TNumber = class
  Int: Integer;
end;
```

2. 메소드 (Methods)

메소드는 클래스와 연관되어 있는 프로시저나 함수를 말한다. 메소드를 호출할 때에는 반드시 객체를 지정해야 하며, 메소드는 그 객체 위에서 동작한다는 점이 일반적인 프로시저나 함수와의 차이점이다. 예를 들어 다음의 코드를 살펴보자.

```
SomeObject.Free;
```

이 코드는 SomeObject 라는 객체의 Free 메소드를 호출한다.

3. Inherited 키워드

메소드를 구현할 때 inherited 라는 키워드를 사용하면 클래스의 조상이 가지고 있던 메소드를 호출하게 된다. 보통 메소드를 오버라이드할 때, 그 메소드의 기능을 추가하기 위해 일단은 조상 클래스의 메소드를 호출하고 나서 부가적인 기능을 수행하거나, 사전 작업을 지정한 후 마지막에 조상 클래스의 메소드를 inherited 키워드를 이용하여 호출하는 것이 전형적인 방법이다.

4. Self 지시어

메소드의 구현부분에 Self 라는 지시어를 사용하면, 이 지시어는 메소드가 실행되는 객체를 지칭하는 것이다. 예를 들어, 다음의 코드는 TCollection 클래스의 Add 메소드를 구현한 부분이다.

```
function TCollection.Add: TCollectionItem;
```

```
begin
  Result := FItemClass.Create(Self);
end;
```

여기에서 Add 메소드는 FItemClass 필드에 의해 참조되는 Create 메소드를 호출하는데, 이 필드는 항상 TCollectionItem 의 자손이다. TCollectionItem.Create 메소드는 TCollection 형의 파라미터를 하나 가지기 때문에, Add 가 호출될 때 TCollection 인스턴스 객체를 넘겨 준다.

5. 프로퍼티 (Properties)

프로퍼티는 필드와 마찬가지로, 객체의 속성을 정의하는 것이다. 그렇지만 필드가 단지 내용이 변경되거나, 검사할 때 사용되는 단순한 저장소의 역할을 하는데 비해 프로퍼티는 데이터를 읽고, 쓰는 특별한 동작과 연관되어 있다. 프로퍼티는 객체의 속성에 접근할 수 있는 제어권을 제공한다. 프로퍼티에 대한 보다 자세한 내용은 제 4 부의 내용을 참고하기 바란다.

- 클래스의 선언, 생성, 파괴

클래스는 사용자가 정의한 데이터 형이다. 클래스는 내부적인 데이터와 메소드를 가지고 있으며, 유사한 많은 객체들의 전반적인 특징과 행동들을 정의한다. 이에 비해 객체는 클래스의 구체적인 변수로, 실제로 메모리를 차지하게 되는 인스턴스이다.
오브젝트 파스칼에서 새로운 클래스 데이터 형을 선언하는 구문은 다음과 같다.

```
type
  TPerson = class
    Name, ID: string;
end;
```

위의 코드는 TPerson 이라는 클래스를 선언하고, 이 클래스의 필드인 Name, ID 를 정의했다. 델파이에서는 일반적으로 모든 클래스의 이름 앞에는 T 자를 붙이도록 되어 있다. 이것은 강제 사항이 아니지만 관습을 따르는 것이 아무래도 좋을 것이다. 위의 선언문을 보면 마치 레코드 선언과 비슷하다는 것을 알 수 있을 것이다. 클래스의 상속을 하는 구문은 class 키워드 옆에 괄호를 치고 이 안에 상속 받을 클래스의 이름을 적어넣으면 된다.

```
TPerson = class(TObject)
```

라는 구문은 TObject 라는 클래스에서 상속받는 TPerson 클래스를 선언하는 구문이다.

이들에 접근하기 위해서는 다음과 같은 방법을 이용하면 된다.

```
var
  Person: TPerson;
begin
  Person.Name := '정지훈';
  Person.ID := 'ttolttol'
end;
```

그러나, 위의 코드는 실제로 실행되지 않는다. 이는 TPerson 이라는 클래스의 변수인 Person 을 선언했지만, 실제 이 클래스의 인스턴스가 생성되지 않았기 때문이다.

cf. 오브젝트 파스칼에서는 클래스 형으로 선언된 각 변수 들이 각각의 객체의 값을 가지고 있는 것이 아니라, 객체에 대한 참조값(reference), 즉 그 객체가 저장된 메모리 위치의 포인터를 가지고 있는 것이다. 이러한 패러다임을 '객체참조 모델 (object reference model)' 이라고 한다. 그러므로 아래와 같이 변수를 선언하면,

```
var
  Person: TPerson;
```

메모리에 객체가 만들어지는 것이 아니라 그저 객체에 대한 메모리의 위치가 정해지고, 여기에 대한 참조값이 Person 변수에 저장되는 것이다. 그러므로, 실제로 객체의 인스턴스를 사용하려면 인스턴스를 직접 만들어야 한다. 폼에 추가하는 컴포넌트의 인스턴스는 델파이에 의해 자동으로 만들어진다.

객체의 인스턴스를 만들기 위해서는 그 객체의 생성자(constructor)인 Create 메소드를 사용하면 된다. 생성자는 새로운 객체를 위해 메모리 배치를 하고 초기화 시키는 특수한 프로시저이다. 이러한 생성자는 델파이의 객체 모델의 가장 기본적인 클래스인 TObject 에서 상속받게 되므로, 개발자들은 아무 걱정없이 이를 사용할 수 있다. 아래의 코드를 보자.

```
var
  Person: TPerson;
begin
```

```
  Person := TPerson.Create;
  Person.Name := '정지훈';
  Person.ID := 'ttolttol';
end;
```

TPerson.Create 구문에 의해 실제 객체가 생성된다. 이 Create 메소드는 TObject 클래스의 constructor 로서, 모든 클래스가 이를 상속하게 된다.
그러므로, TPerson = class 와 TPerson = class(TObject) 는 같은 의미이다.
이렇게 일단 객체를 생성했으면 이를 결국에는 삭제해야 한다. 이때에는 Free 메소드를 호출하면 된다. Free 메소드 역시 TObject 클래스의 메소드이다. 이와 같이 필요할 때 객체를 만들어서 사용하고 일이 끝나면 없애 버리는 식으로 사용하면 된다.
그럼, 이를 이용해서 간단한 예제를 만들어 보도록 하겠다.
다음과 같이 에디트 박스 2 개와 버튼 4 개를 폼 위에 올려 놓고, 각 버튼의 Name 프로퍼티를 btnCreate, btnFree, btnAssign, btnShow 로 설정하자 (그림 5-1). 그리고, Caption 프로퍼티를 '생 성', '해 제', '대 입', '보 기'로 설정한다. 각 에디트 박스의 Text 프로퍼티는 지운다. 유닛의 type 문장에 TPerson 클래스를 선언하고, 전역변수 Person 을 선언한다. 그리고, 각 버튼의 OnClick 이벤트 핸들러를 아래와 같이 작성한다.

```
type
  ...(중략)
  TPerson = class                          //클래스 선언부
    Name, ID: string;
  end;

var
  Form1: TForm1;
  Person: TPerson;              //전역변수 선언

procedure TForm1.btnCreateClick(Sender: TObject);
begin
  Person := TPerson.Create;  //인스턴스 생성
end;

procedure TForm1.btnAssignClick(Sender: TObject);
begin
  Person.Name := Edit1.Text; //에디트 박스의 값을 대입
```

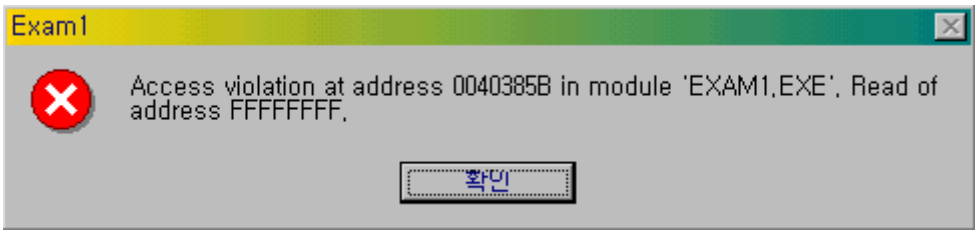
```
  Person.ID := Edit2.Text;
end;

procedure TForm1.btnShowClick(Sender: TObject);
begin
  ShowMessage ('안녕하세요 ? '+Person.Name+ '씨, 당신의 ID 는 '+Person.ID+'입니다.');
end;

procedure TForm1.btnFreeClick(Sender: TObject);
begin
  Person.Free;
end;
```

(그림 5-1) 예제 폼 그림

4 개의 버튼에 대한 이벤트 핸들러를 이와 같이 작성하면 '생성' 버튼을 클릭하면 TPerson 클래스의 인스턴스가 생성되어 Person 변수에 대입되고, '대입' 버튼을 클릭하면 Person 객체의 Name, ID 필드에 에디트 박스의 내용이 대입된다. '보기' 버튼을 클릭하면 Person 객체의 Name, ID 필드를 이용해 다음(그림 5-2)과 같은 메시지 박스가 뜬다. 이때 TPerson 클래스의 인스턴스가 생성되지 않았다면 'Access violation' 에러 메시지가 나타날 것이다 (그림 5-3). 마찬가지로 '생성' 버튼을 클릭한 후, '해제' 버튼을 클릭하면 생성되었던 TPerson 클래스의 인스턴스가 파괴되므로, 이 때 '대입', '보기' 버튼을 클릭하면 역시 'Access violation' 에러가 발생한다.

(그림 5-2) 에디트 박스 이름(정지훈)과 ID(ttolttol)를 입력한 후 '생성', '대입', '보기' 버튼을 클릭하

면 나타나는 메시지 박스



(그림 5-3) Create 하지 않았거나, Free 를 호출한 후 '대입', '보기' 버튼을 클릭해서 인스턴스에 접근하려 하면 이와 같은 에러 메시지가 나타난다.

- 메소드와 생성자(constructor), 파괴자(destructor)의 추가

이제 TPerson 클래스에 몇가지 메소드를 추가해서 조금은 쓸모가 있는 클래스로 만들어 보자. 현재 가지고 있는 Name 필드는 그대로 두고, ID 필드 대신, 주민등록번호를 저장할 수 있는 RegID 필드를 만들자. 이때 필드 임을 나타내기 위해 각 필드의 앞에 접두어로 'F'를 붙이도록 한다. 그리고, 주민등록번호 필드 값을 가지고 이 값이 유효한지 알아보는 IsValid 함수와 나이와 성별을 알 수 있는 GetAge, GetSex 라는 함수를 추가한다.
또한, 이 클래스의 객체가 생성될 때 값을 초기화 시키기 위해 생성자를 추가하자. 생성자(constructor)는 특별한 형태의 프로시저로, 이것을 클래스에 적용하면 자동적으로 그 클래스의 객체를 위해 메모리를 할당하게 되며, 초기화 작업을 지정할 수 있다. 단순히 constructor 라는 예약어로 선언하면 되며, 클래스 메소드로 사용할 때에는 객체에 대한 메모리 할당 작업을 하게 되지만, 이미 인스턴스화된 객체에서 사용될 때에는 메모리 할당 작업은 하지 않고, 정의된 초기화 작업만 하게 된다. 여기서 만드는 클래스에는 생성자로 Create 라는 프로시저를 정의하는데, 파라미터로 FName, FRegID 필드를 채울 수 있도록 선언한다.
마찬가지로 파괴자(destructor)도 정의할 수 있으며 destructor 라는 예약어로 선언하면 된다. 이 프로시저는 객체가 파괴되기 전에 시스템 자원을 release 하는 작업을 주로하게 되는데, 여기서는 특별히 추가하지 않는다. 따로 파괴자를 추가하지 않아도 모든 클래스는 TObject 에서 파생되기 때문에 기본적인 Free, Destroy 프로시저는 정의되어 있다. .
이렇게 확장한 클래스의 선언부는 다음과 같다.

```
type

  TSex = (sMale, sFemale);              //GetSex 함수에서 쓰임
  TPerson = class(TObject)
    FName: string;
```

```
    FRegID: string;
    constructor Create(Name, ID: string);
    function IsValid: Boolean;
    function GetSex: TSex;
    function GetAge: integer;
  end;
```

이제 각 함수를 구현해 보도록 하자.
먼저 생성자를 구현해 보자. 생성자에서는 Name, ID 를 파라미터로 받아서 이 값을 필드에 적용하여 초기화 한다.

```
constructor TPerson.Create(Name, ID: string);
begin
  FName := Name;
  FRegID := ID;
end;
```

다음으로 FRegID 필드에 저장된 주민등록번호가 잘못된 것이 아닌지 검사하는 IsValid 함수를 구현하도록 하자. 주민등록번호를 검증하는 방법은 주민등록번호의 13 자리 중 마지막 자리를 제외한 12 자리에 각각 2, 3, 4, 5, 6, 7, 8, 9, 2, 3, 4, 5 를 곱한 뒤 전체를 더한다. 예를 들어, 701203-1079727 이라는 주민등록번호가 있다고 하면, (7*2) + (0*3) + (1*4) + (2*5) + (0*6) + (3*7) + (1*8) + (0*9) + (7*2) + (9*3) + (7*4) + (2*5) = 136 이다. 이 값을 11 로 나눈 나머지 값을 11 에서 뺀다. 위의 경우에는 나머지가 4 이므로 11-4=7 이 된다. 이 값이 마지막 주민등록번호 값과 일치하면 일단 유효한 번호인 것이다. 일종의 해싱 함수(hashing function)이다.
이를 이용해서 IsValid 함수를 구현해 보자.

```
function TPerson.IsValid: Boolean;
begin
  Result := False;
  if ((Length(FRegID) = 13) and ('0000000000000' < FRegID) and ('9999999999999' > FRegID))
    then
    if (11 - ((StrToInt(FRegID[1]) * 2 + StrToInt(FRegID[2]) * 3 + StrToInt(FRegID[3]) * 4 +
      StrToInt(FRegID[4]) * 5 + StrToInt(FRegID[5]) * 6 + StrToInt(FRegID[6]) * 7 +
      StrToInt(FRegID[7]) * 8 + StrToInt(FRegID[8]) * 9 + StrToInt(FRegID[9]) * 2 +
      StrToInt(FRegID[10]) * 3 + StrToInt(FRegID[11]) * 4 + StrToInt(FRegID[12]) * 5) mod 11)) =
```

```
        StrToInt(FRegID[13]) then
        Result := True
      else
        Result := False;
end;
```

즉, 일단 FRegID 필드 값이 13 자리이고, 숫자로 이루어 졌는지 확인하고, 각 자리의 값을 정수형으로 바꿔서 위에서 설명한 공식에 따라 적절한 지 알아보고 적절하면 True, 그렇지 않으면 False 를 반환한다.

Cf.

1. 파스칼의 문자열은 일종의 배열로 생각할 수 있기 때문에, 이런 형태의 코드가 가능한 것이다. 예를 들어 Name 이라는 문자열 변수의 값이 'Sample'이라고 할 때, Name[1], Name[2], Name[3], Name[4], Name[5]의 값은 각각 'S', 'a', 'm', 'p', 'l', 'e' 이다.
2. Length 함수
   문자열의 길이를 구하는 함수이다. 델파이 1.0 까지는 문자열의 0 번째 요소, 즉 위의 예를 들면 Name[0] 값에 문자열의 길이가 저장되어 있었으나, 2.0 부터는 Length 함수를 사용하여 길이를 구한다.
3. StrToInt, IntToStr 함수
   정수와 문자열의 형전환을 해주는 함수이다. 예를 들어 IntToStr(123)의 결과값은 '123'이고, StrToInt('123')의 결과값은 123 이다.

마지막으로 GetSex 와 GetAge 를 구현해 보자.

```
function TPerson.GetSex: TSex;
begin
  if IsValid then
    if FRegID[7] = '1' then Result := sMale else Result := sFeMale;
end;

function TPerson.GetAge: integer;
var
  Date: string;
begin
  if IsValid then
  begin
```

```
    Date := DateToStr(Now);
    Result := StrToInt(Copy(Date, 1, 2)) - StrToInt(Copy(FRegID, 1, 2));
    if (Copy(Date, 4, 2) + Copy(Date, 7, 2)) < Copy(FRegID, 3, 4) then
      Result := Result - 1;
  end;
end;
```

GetSex 함수는 주민등록번호의 7 번째 자리가 '1'이면 남자, '2'이면 여자이므로 쉽게 이해가 될 것이다. GetAge 함수는 일단 System 의 날짜를 얻을 수 있는 Now 함수를 사용한 후 이를 문자열로 변환한다. 그리고, 여기서 연도만을 뽑아 정수로 변환한 후, 주민등록번호의 첫 두자리가 태어난 해를 가리키게 되므로 이를 정수로 변환해서 뺀다. 여기에 생일이 지나지 않았으면 1 을 빼야 만으로 계산한 나이가 된다.
생일을 비교할 때, 입력 받은 주민등록번호 상의 생일은 '0301' 과 같이 연속된 4 개의 문자로 구성되지만, DateToStr 로 변환된 Date 변수의 값은 '98-06-21'과 같은 형식으로 저장되기 때문에 이를 동등하게 비교하기 위해 Copy 함수를 이용하여 주민등록번호 상의 생일처럼 연속된 4 개의 문자로 만드는 루틴이 추가되었다.
그럼, 이 클래스를 활용하는 프로그램을 만들어 보자.
두번째 예제에 TPerson 의 구현 부분을 추가하고, 폼을 다음(그림 5-4)과 같이 디자인 한다. 여기서도 첫번째 예제와 마찬가지로 에디트 박스를 2 개 추가한다. 여기에는 이름과 주민등록번호를 입력받을 것이다. 그리고, 버튼을 2 개 추가한 후 Name 프로퍼티를 btnAssign 과 btnExecute 로 설정하고, Caption 을 각각 '대입'과 '실행'으로 설정한다.

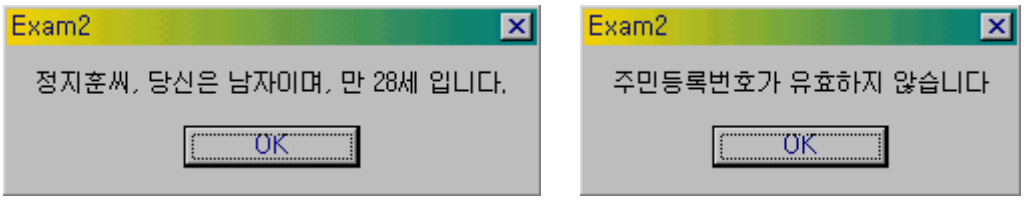
(그림 5-4) 두번째 예제의 폼 디자인

그리고, 각 버튼의 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.btnAssignClick(Sender: TObject);
begin
  if Assigned(Person) then Person.Destroy;
  Person := TPerson.Create(Edit1.Text, Edit2.Text);
end;
```

```
procedure TForm1.btnExecuteClick(Sender: TObject);
var
  Sex, Age: string;
begin
  if not Assigned(Person) then
  begin
    ShowMessage('클래스가 생성되지 않았습니다 !');
    Exit;
  end;
  if not Person.IsValid then
  begin
    ShowMessage('주민등록번호가 유효하지 않습니다');
    Exit;
  end
  else
  begin
    if Person.GetSex = sMale then Sex := '남자' else Sex := '여자';
    ShowMessage(Person.FName + '씨, 당신은 ' + Sex + '이며, 만 ' +
      IntToStr(Person.GetAge) + '세 입니다.');
  end;
end;
```

그렇게 어렵지 않은 코드이므로 자세한 설명은 생략하도록 하겠다.
참고로 제대로 된 주민등록번호를 입력했을 때와 그렇지 않았을 때의 메시지 박스는 다음과 같다.



(그림 5-5) 예제 실행 결과

이런 클래스의 사용 방법은 델파이에서 TComponent 를 클래스를 상속해서 더욱 재사용성이 뛰어난 컴포넌트로 개발할 수 있다. 이를 위해서는 프로퍼티에 대한 이해와 델파이가 제공하고 있는 컴포넌트 모델에 대한 이해가 필요한데, 여기에 대해서는 제 4 부에서 자세

하게 다룰 것이다.

상속성과 델파이 폼

새로운 프로젝트를 생성하면, 델파이는 새로운 폼을 보여준다. 코드 에디터의 새로운 프로젝트의 내용을 살펴 보면, 델파이가 폼에 대한 새로운 객체 클래스를 선언하고 새로운 폼 객체를 생성하는 코드를 만들어 낸다는 것을 알 수 있다. 그러면 초기에 생성되는 델파이의 코드를 살펴 보자.

```
unit Unit1;

interface

uses Windows, Classes, Graphics, Forms, Controls, ... ;

type

  TForm1 = class(TForm)      {폼에 대한 새로운 클래스 선언}
  private
    { Private declarations }
  public
    { Public declarations }
  end;                       {클래스 선언부는 여기서 종료된다.}

var

  Form1: TForm1;

implementation      {구현 부분의 시작}

{$R *.DFM}

end.                {구현 부분과 유닛의 끝}
```

새로운 객체 데이터 형인 TForm1 은 TForm 에서 상속받는다는 것을 알 수 있다. 객체에는 데이터 필드와 메소드를 가진다는 것은 이미 설명한 바 있다. 그런데, TForm1 에는 아

직 메소드나 데이터 필드를 지정하지 않았다. 다만 TForm 클래스에서 상속받은 메소드와 프로퍼티를 가지게 될 것이다. 개발자는 TForm1 의 선언 부분에 마음대로 메소드나 프로퍼티 등을 추가할 수 있다. 또한, 컴포넌트를 폼에 추가하는 동작에 의해 델파이가 메소드나 데이터 필드를 추가해 준다.
그렇지만, 이 어플리케이션을 실행시켜 보면 간단한 폼이 생성되는 것을 볼 수 있다. 이것이 의미하는 것은 무엇인가 ? 즉, 기본적으로 TForm 클래스의 모든 메소드와 기능을 상속받아서 사용할 수 있다는 것이다.
변수 선언부에서는 새로운 변수인 Form1 을 TForm1 으로 선언한다. 이렇게 선언함으로써 TForm1 클래스의 인스턴스가 될 수 있다. 클래스의 인스턴스는 여러 개 생성될 수 있다는 것은 이미 알고 있을 텐데, 만약 TForm1 클래스를 여러 인스턴스로 생성하면 그것이 바로 MDI(Multiple Document Interface) 어플리케이션이 되는 것이다.
그러면, 폼에 버튼을 하나 추가하고 그 버튼의 OnClick 이벤트 핸들러를 다음과 같이 작성하자.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.Color := clGreen;
end;
```

이렇게 이벤트 핸들러를 작성하고 나면, 폼의 유닛의 코드는 다음과 같이 바뀌어 있을 것이다.

```
unit Unit1;

interface

uses Windows, Classes, Graphics, Forms, Controls, ...;

type
  TForm1 = class(TForm)
    Button1: TButton;                          {새로운 데이터 필드}
    procedure Button1Click(Sender: TObject);   {새로운 메소드}
  private
    { Private declarations }
  public
    { Public declarations }
```

```
  end;

var

  Form1: TForm1;

... (후략)
```

TForm1 의 선언부에 새로운 Button1 이라는 필드가 추가 되었음을 알 수 있다. 이렇게 새로운 컴포넌트를 폼에 추가할 때마다 새로운 필드가 type 선언부에 추가된다. 또한, 델파이에서 작성하는 모든 이벤트 핸들러는 폼 객체의 메소드로 선언된다. TForm1 에는 이제 Button1Click 이라는 새로운 메소드 프로시저가 추가되었다.
여기서 이런 동작들을 OOP 의 개념으로 생각해보자. 델파이의 폼 디자이너는 과연 무엇인가 ? 결국 델파이의 폼 디자이너는 TForm 이라는 클래스를 상속받은 새로운 형태의 클래스를 쉽게 만들어주는 일종의 위저드인 셈이다.
이제 상속의 의미를 이해하기 쉬운 비유를 들어 생각해 보자.
앞에서 처음 어플리케이션이 생성되었을 때의 폼은 기본 옵션으로된 자동차를 한 대 구입한 것으로 가정하자. 기본 옵션의 자동차는 자동차 메이커에서 항상 정해진 방식대로 만들어지기 때문에 가장 기본적인 기능만을 가지고 달릴 수 있을 것이다. 그렇지만, 이 자동차를 구입한 사람이 에어컨도 달고, 파워 핸들과 에어백 등의 여러가지 옵션을 장착할 수 있다. 이를 위해서 자동차를 구입한 사람은 자동차 전체를 새로 만들 필요는 없는 것이다. 즉, 이를 상속을 통해 설명하자면 기본적인 자동차를 상속받은 사용자가 자식 클래스 자동차에 새로운 옵션들을 추가한 것이다.
마찬가지로 델파이의 폼은 상속성을 설명할 때 가장 알기 쉽고, 전형적인 방법을 보여준다고 말할 수 있다.

클래스의 범위 (scope)

클래스의 멤버들은 서로 다른 범위를 가질 수 있다. 이러한 범위를 나타내는 지시어로는 private, protected, public, published 의 4 가지가 있다. 참고로 오브젝트 파스칼에서는 유닛도 범위를 결정하는데 한 몫을 한다. 비록 private 로 선언되었더라도 같은 유닛에 있으면 모두 접근이 가능하다.

- private
  다른 유닛에 있는 경우 private 섹션에 선언된 멤버에는 접근할 수 없다. 다른 사용자가 접근할 필요가 없는 멤버들은 여기에 선언한다.

- protected
  이 클래스에서 상속받은 클래스에서만 접근할 수 있는 멤버들을 여기에 선언한다. 즉, 현재의 클래스를 상속받아서 새로운 클래스를 만들 때 수정이 필요하다면 개발자에게 접근이 가능해야 하지만, 일반적으로 사용할 때에는 접근할 수 없도록 할 때 사용된다.

- public, published
  다른 클래스에서 제한 없이 사용될 수 있는 멤버들을 여기에 선언한다. published 는 오브젝트 인스펙터에서도 볼 수 있는 멤버들을 선언할 때 사용한다. published 로 선언하면 멤버에 대한 RTTI(runtime type information)가 생성되며, 이를 이용해서 다른 프로그램들이 런타임에서 객체에 대한 필드, 메소드, 프로퍼티에 접근하게 된다. 델파이는 RTTI 를 이용하여 오브젝트 인스펙터에 프로퍼티를 보여주며, 폼 파일에 프로퍼티의 값을 저장하고 불러올 수 있다. Published 프로퍼티에 사용할 수 있는 데이터 형은 서수형과 문자열, 클래스와 메소드 포인터형, 세트형, 실수형 등을 사용할 수 있다. 배열은 사용할 수 없다. published 멤버가 있는 대부분의 클래스는 TPersistent 클래스에서 상속받는 것이 보통이다.

범위에 대한 지시어가 없을 때에는 일단 public 으로 간주한다. 그렇지만, 프로그래밍을 할 때에는 꼭 이러한 범위를 지정해주는 것이 좋은 버릇이다.

메소드 (Method)

객체의 동작은 메소드에 의해 정의된다. 메소드는 프로시저나 함수와 비슷하지만 지정한 클래스와 그의 파생 클래스 만의 객체를 위해 정의된다는 점이 다르다. 메소드에는 보이지 않는 파라미터로서 Self 라는 객체 자신의 참조자가 전달된다. 메소드는 또한 클래스 메소드로서 선언될 수 있는데, 이런 경우에는 클래스 참조로는 호출될 수 있으나 객체 참조로서는 호출되지 않는다. 객체가 없으므로 Self 파라미터도 없다.

- 메소드의 종류

메소드에도 그 동작방식과 용도에 따라 여러 가지 종류가 있다. 다음에 이들 각각의 특징에 대해 설명하였다.

1. 정적 메소드 (Static method): 지시어 static;
   아무런 지시어가 없을 때에는 디폴트로 정적 메소드로 간주된다. 컴파일 할 당시에 메소드가 위치한 메모리 주소가 확정되는 메소드이므로, 그만큼 실행속도가 빠르지만 상

속을 받은 클래스에서 메소드를 새롭게 정의하면, 그 메소드만(같은 이름의 경우) 사용이 가능하므로 융통성이 적다.

2. 가상 메소드 (Virtual mehtod): 지시어 virtual;
   가상 메소드는 실행 시에 late binding 이라는 과정을 통해 실제로 호출될 메모리 주소가 결정된다. 내부적으로 가상 메소드가 참조되면 변수에 의해 참조되는 객체의 실제 클래스 형이 사용되는데, 이 작업이 가상함수 테이블(virtual method table(VMT), vtable)에 기록되어 있는 주소를 참조하여 이루어진다. 그러므로, 실제 참조되는 클래스 형에 따라서 여러가지 메소드가 호출될 수 있는 '다형성'이 구현될 수 있다.

3. 동적 메소드 (Dynamic method): 지시어 dynamic;
   기본적인 사용법이나 목적은 가상 메소드와 동일하지만, 내부적인 처리 방법에 약간의 차이가 있다. 가상 메소드가 내부적으로 VMT 를 이용해서 메모리 주소를 참조하는데 비해 동적 메소드는 메소드를 지정하는 코드를 이용해서 메모리 주소를 찾게 된다. 그렇기 때문에 가상 메소드처럼 테이블을 직접 참조하는 방법보다 다소 느리게 동작하지만 메모리는 덜 사용하게 된다.

4. 추상 메소드 (Abstract method): 지시어 abstract;
   일반적으로 메소드를 클래스에 선언할 때, 컴파일러는 메소드 프로시저가 유닛의 어느 부분인가 구현되어 있을 것으로 간주하게 된다. 그런데, 추상 메소드로 선언하면 컴파일러는 구현 부분이 자손 클래스에 있을 것으로 생각하고 이를 검사하지 않는다. 그런데, 만약 자손 클래스에서 이를 구현하지 않아서 추상 메소드가 호출되면 치명적인 에러가 발생하게 되므로 주의하기 바란다.

- 메소드 오버라이드 (method override)

override 지시어는 가상 또는 동적 메소드를 재정의할 때 사용된다. 재정의하는 방법은 다음과 같다.

```
type

  TMyParent = class
    procedure AMethod; virtual;
  end;

  TMyClass = class(TMyParent)
```

```
    procedure AMethod; override;
  end;
```

- 클래스 메소드 (Class method)

보통 메소드는 클래스의 인스턴스에 대한 행동을 정의한다. 그런데, 어떤 때에는 클래스 자체에 대한 메소드가 있으면 할 경우가 있다. 이럴 때에는 클래스 메소드를 정의해서 사용한다. 클래스 메소드는 객체가 생성되지 않아도 사용할 수 있다.
클래스 메소드를 선언하려면 메소드 정의 부분에서 class 키워드를 앞에 붙여주면 된다.

- 메소드 오버로딩

델파이 4 에서는 객체 들이 같은 이름을 가진 여러 개의 메소드를 가질 수 있다. 이를 메소드 오버로딩이라고 하는데, 같은 이름을 가진 메소드 들은 argumets 의 type signature 를 가지고 서로를 구별한다. 오버로드된 메소드는 키워드 오버로드로 표시된다. 그렇기 때문에, 객체 들은 다음과 같이 다른 두 개의 생성자를 가질 수 있다.

```
constructor Create(AOwner: TComponent); overload; override;
constructor Create(AOwner: TComponent; Text: string); overload;
```

전역 함수와 프로시저역시 오버로드가 가능하다.
델파이 3 까지만 해도 메소드 오버로딩을 지원하지 않았기 때문에, 생성자의 이름을 다르게 했어야 했다. 예를 들어 모든 윈도우 컨트롤은 Create, CreateParented 의 2 개의 생성자를 공통적으로 가지고 있었다. 델파이 4 부터는 Create 라는 메소드 이름을 가진 여러 생성자를 가질 수 있다.
오브젝트 파스칼의 이전 버전에서는 같은 이름의 메소드를 선언할 경우 조상 클래스의 메소드는 사용되지 않았다. 예를 들어, Create 메소드를 Owner 파라미터를 넘겨주지 않고 호출할 경우 과거에는 TObject 에 선언된 생성자를 호출하지 않고 컴파일 에러를 발생시켰다. 메소드 오버로딩으로 이러한 문제들이 다소 변경되었는데, 다음의 코드를 살펴보자.

```
type
  A = class
    public
      procedure p(I: Integer); virtual;
  end;
  B = class(A)
```

```
    public
      procedure p(S: string);
  end;

var
  aB: B;
begin
  aB.p('one');      { works }
  aB.p(1);          { compile error! }
end;
```

여기에서 변경된 메소드의 파라미터를 대입할 경우에는 동작하지만, 원본 클래스의 메소드는 동작하지 않는다. 이를 해결하기 위해서는 overload 지시어를 사용하면 된다. 다음의 코드를 살펴보자.

```
type
  A = class
    public
      procedure p(I: Integer); overload; virtual;
  end;
  B = class(A)
    public
      procedure p(S: string); overload;
  end;

var
  aB: B;
begin
  aB.p('one');      { works }
  aB.p(1);          { now this one works too! }
end;
```

이 경우에는 동작하지만, 컴파일러가 경고를 한다. 이를 없애기 위해서는 상속받은 메소드에 다음과 같이 reintroduce 키워드를 지정하면 된다.

```
type
```

```
A = class
  public
    procedure p(I: Integer); overload; virtual;
end;
B = class(A)
  public
    procedure p(I: Integer; S: string); reintroduce; overload;
end;
```

동적 바인딩(Dynamic binding)과 다형성(Polymorphism)

파스칼의 함수와 프로시저는 기본적으로 정적 바인딩(static binding)을 이용하고 있다. 이것은 메소드 호출이 컴파일러나 링커에 의해 해석되며, 컴파일러나 링커는 이 호출문을 그 함수나 프로시저가 존재하는 특정 메모리 위치의 호출로 바꾸어 놓는다는 의미이다.
오브젝트 파스칼을 비롯한 객체지향 언어는 이와 다른 형태의 동적 바인딩을 지원한다. 이 경우에는 메소드의 실제 메모리 주소가 실행 중에 결정되는 것이다. 이런 특성을 이용해 다형성을 지원할 수 있게 되는데, 다형성이란 어떤 메소드의 호출문을 작성하고 그것을 변수에 대입해도, 어느 메소드가 호출될지는 그 변수에 관계된 객체의 데이터 형에 따라 달라지는 것이다. 다시 말해, 주어진 메소드에 대해 다수의 버전이 있을 수 있고, 그래서 하나의 메소드가 이러한 버전들 각각을 가리킬 수 있다.

정 리 (Summary)

이번 장에서는 오브젝트 파스칼의 OOP 적인 특성에 대해 간단하게나마 알아보았다. 사실 오브젝트 파스칼의 기능을 충분히 활용하려면 OOP 에 대한 전반적인 이해가 필수적이다. 이 책의 범위가 OOP 에 대해 심도 있게 논의할 수는 없지만, 이 책을 읽는 독자들은 반드시 따로 OOP 에 대해 공부해서 그 기법과 철학을 체득하길 바란다. 그렇게 익힌 OOP 에 대한 개념 들은 실제로 델파이 프로젝트를 진행하게 되면 소중한 지식으로 활용될 것이다.
다음 장에서는 OOP 의 대표적인 언어인 C++과 자바를 오브젝트 파스칼과 비교하는 시간을 가지도록 한다. 서로 다른 언어의 특징에 대해서 알아보는 것도 델파이를 잘 이해하는데 큰 도움이 될 것이다.