CHAPTER _2

프로그램과 유닛

델파이 프로그램은 기본적으로 '유닛' 들로 구성되고, 이들 유닛들을 묶는 프로젝트 소스가 있습니다. 이 장에서는 유닛 내의 구성 요소들과 여러 유닛들 사이의 의존 관계에 대해 알아봅니다.

■프로그램의 구조와 문법 ■유닛의 구조와 문법 ■유닛 참조와 uses 절

프로그램은 '유닛' 이라는 소스 코드 모듈들로 구성됩니다. 유닛들은 program, library 혹은 package 헤더로 시작되는 특수한 소스 코드 모듈에 의해 함께 묶이게 됩니다. 각 유닛은 각각의 파일로 저장되고 별도로 컴파일되며, 컴파일된 유닛이 링크되어 애플리케이션이 만들어집니다. 델파이는 계층적인 네임스페이스를 도입하여 유닛들을 구성하는 데에 더 큰 유연성을 제공합니다. 네임스페이스와 유닛을 사용함으로써 다음과 같은 일들이 가능해집니다.

- 큰 프로그램을 별도로 편집 가능한 모듈들로 나눌 수 있습니다.
- 프로그램들 사이에 공유할 수 있는 라이브러리를 만들 수 있습니다.
- 소스 코드 없이 다른 개발자에게 라이브러리를 배포할 수 있습니다.

이 장에서는 program 헤더, 유닛 선언 문법, uses 절 등 델파이 애플리케이션의 전반적인 구조를 다룹니다.

프로그램의 구조와 문법

완전한 델파이 애플리케이션은 여러 유닛 모듈들로 구성되며, 그 유닛들은 프로젝트 파일이라고 불리는 단일 소스 코드 모듈로 묶여집니다. 전통적인 파스칼 프로그래밍에서는 메인 프로그램을 포함한 모든 소스 코드는 .pas 파일들에 저장되었습니다. 코드기어 개발툴에서는 메인 프로그램 소스 모듈을 지정하는 .dpr 파일을 사용합니다. 대부분의 다른 소스 코드는 전통적인 .pas 확장자를 가진 유닛 파일들에 저장됩니다. 컴파일러가 프로젝트를 빌드하려면 프로젝트 소스 파일과 함께 각 유닛들의 소스 파일이나 컴파일된 유닛 파일이 필요합니다.
실행 가능한 델파이 애플리케이션의 소스 코드 파일에는 다음과 같은 내용이 들어 있습니다.

Note
엄밀히 말하면, 프로젝트에서 모든 유닛들을 명시적으로 uses 할 필요는 없으며, 모든 프로그램은 System 유닛과 SysInit 유닛을 자동으로 uses 합니다.

- program 헤더
- uses 절 (선택적)
- 선언과 문장 블럭

추가적으로, RAD 스튜디오 프로그램은 제네릭 유닛들을 검색하기 위한 추가적인 네임스페이스들을 지정하기 위해 네임스페이스 절을 포함할 수도 있습니다.
컴파일러와 IDE는 이들 세가지 파일들을 단일 프로젝트(.dpr) 파일에서 찾으려고 하게 됩니다.

program 헤더

program 헤더는 실행 가능한 프로그램의 이름을 지정합니다. 예약어 program에 유효한 식별자, 그리고 세미콜론(;) 순으로 구성됩니다. 코드기어 툴로 개발된 애플리케이션의 경우 이 식별자는 프로젝트 소스 파일 이름과 반드시 일치해야 합니다.
다음의 예제는 Editor라는 이름의 프로그램을 위한 프로젝트 소스를 보여줍니다. 이 프로그

```
program Editor;

uses Forms, REAbout,  // "어바웃" 박스
  REMain; // 메인 폼

{$R *.res}

begin
  Application.Title := 'Text Editor';
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
end.
```

램의 이름이 Editor이므로, 프로젝트 파일도 Editor.dpr이어야 합니다.
첫 라인은 program 헤더입니다. 이 예제의 uses 절에서는 Forms, ReAbout, ReMain의 세 유닛에 의존함을 알려줍니다. $R 컴파일러 지시자는 이 프로젝트의 리소스 파일을 프로그램에 링크시킵니다. 마지막으로, begin과 end 예약어 사이의 문장들의 블럭은 프로그램이 기동될 때 실행됩니다. 다른 모든 델파이 소스 파일들과 마찬가지로 프로젝트 파일도 (세미콜론이 아닌) 마침표(.)로 끝납니다.
프로그램의 로직 대부분은 유닛 파일에 있기 때문에, 델파이 프로젝트 파일은 대체로 길이가 짧습니다. 델파이 프로젝트 파일은 일반적으로 애플리케이션의 메인 윈도우를 띄우고 이벤트 처리 루프를 시작하는 정도의 간단한 코드만 가지고 있습니다. 프로젝트 파일은 IDE에서 자동으로 생성되고 유지 관리되므로 직접 편집할 필요가 있는 경우는 드뭅니다.
표준 파스칼에서는, 프로그램 헤더의 프로그램 이름 뒤에 다음과 같이 파라미터가 올 수 있습니다.

```
program Calc(input, output);
```

델파이 컴파일러에서는 이 파라미터는 무시됩니다.
program 헤더는 자신만의 네임스페이스를 시작합니다. 이것을 프로젝트 기본 네임스페이스라고 합니다.

프로그램의 uses 절

uses 절은 프로그램으로 통합될 유닛들을 나열합니다. 이러한 유닛들도 자체 uses 절을 가질 수 있습니다. uses 절에 대한 자세한 내용은 2장의 "유닛 참조와 uses 절"을 참조하십시오.

블럭

블럭에는 프로그램을 실행할 때 실행되는 단순 문장 또는 구조 문장이 있습니다. 대부분의 프로그램에서 블럭은 예약어 begin과 end 쌍으로 묶인 복합문으로 구성되는데, 단순히 프로젝트의 Application 객체의 메소드 호출입니다. 모든 프로젝트는 TApplication, TWebApplication 혹은 TServiceApplication의 인스턴스를 가지는 Application 변수를 가지고 있습니다. 블럭은 상수, 타입, 변수, 프로시저 및 함수의 선언도 포함할 수 있습니다. 이러한 선언은 반드시 블럭의 문장 부분보다 앞에 있어야 합니다.

유닛의 구조와 문법

유닛은 타입(클래스 포함), 상수, 변수 및 루틴(함수와 프로시저)으로 구성됩니다. 각 유닛은 자체 유닛(.pas) 파일에서 정의합니다.
유닛 파일은 unit 헤더로 시작되고, 뒤이어 interface 예약어가 나옵니다. interface 예약어에 이어 uses 절에서 유닛 의존성을 기술합니다. 다음으로 implementation 섹션이 오고, 선택적으로 initialization 및 finalization 섹션이 올 수 있습니다. 유닛 파일 구조는 다음과 같습니다.

```
unit Unit1;

interface

uses  { 유닛 의존성 리스트… }

   { Interface 섹션의 내용 }

implementation

uses  { 유닛 의존성 리스트… }

   { 클래스 메소드, 프로시저, 함수들의 구현부가 여기에… }

initialization
   { 유닛 초기화 코드가 여기에… }

finalization
   { 유닛 최종 작업 코드가 여기에… }

end.
```

유닛은 반드시 end와 마침표(.)로 끝나야 합니다.

unit 헤더

unit 헤더는 유닛의 이름을 지정합니다. 유닛 헤더는 예약어 unit, 유효한 식별자, 세미콜론 순으로 구성됩니다. 코드기어 툴을 사용하여 개발된 애플리케이션의 경우 식별자는 반드시 유닛 파일 이름과 일치해야 합니다. 따라서 MainForm.pas이라는 이름의 소스 파일의 유닛 헤더는,

```
unit MainForm;
```

와 같을 것이며, 컴파일된 유닛 파일의 이름은 MainForm.dcu가 됩니다.
유닛 이름은 반드시 프로젝트 내에서 유일해야 합니다. 유닛 파일이 다른 디렉토리에 있더라도 같은 이름을 가진 두 개의 유닛을 한 프로그램에서 사용할 수 없습니다.

interface 섹션

유닛의 interface 섹션은 interface 예약어로 시작해서 implementation 섹션이 시작될 때까지입니다. 인터페이스 섹션에는 다른 유닛이나 프로그램에서 사용 가능한 상수, 타입, 변수, 프로시저 및 함수를 선언합니다. 다른 유닛의 코드가 마치 자신의 유닛 안에서 선언된 것처럼 호출할 수 있기 때문에 이런 항목들을 public이라고 합니다.
프로시저나 함수의 인터페이스 선언은 루틴의 시그니처(signature), 즉 해당 루틴의 이름, 파라미터들, 그리고 리턴 타입(함수의 경우)만을 포함합니다. 프로시저나 함수의 실행 코드 블럭은 임플먼테이션 섹션에 위치합니다. 그러므로 인터페이스 섹션의 프로시저와 함수 선언은 forward 선언과 비슷하다고 할 수 있습니다.
클래스의 인터페이스 선언은 필드, 속성, 프로시저, 함수 등 모든 클래스 멤버의 선언을 포함해야 합니다.
인터페이스 섹션은 자체의 uses 절을 포함할 수 있으며, 반드시 예약어 interface 바로 뒤에 나타나야 합니다. uses 절에 대한 자세한 내용은 2장의 "유닛 참조와 uses 절"을 참조하십시오.

implementation 섹션

유닛의 implementation 섹션은 예약어 implementation로 시작하여 initialization 섹션이 시작할 때까지, initialization 섹션이 없는 경우에는 유닛의 끝까지입니다. 임플먼테이션 섹션은 인터페이스 섹션에 선언된 프로시저와 함수를 정의합니다. 임플먼테이션 섹션 내에서 이러한 프로시저와 함수는 순서에 상관없이 정의하고 호출할 수 있습니다. 임플먼테이션 섹션에 public 프로시저와 함수 헤더를 정의할 때 파라미터 리스트를 생략할 수도 있지만, 파라미터 리스트를 포함할 경우에는 인터페이스 섹션의 선언과 반드시 일치해야 합니다.
임플먼테이션 섹션은 public 프로시저 및 함수들의 정의 외에도, 해당 유닛 내에서만 사용 가능한(private) 상수, 타입(클래스 포함), 변수, 프로시저 및 함수를 선언할 수 있습니다. 다시 말해, 인터페이스 섹션과 달리 임플먼테이션 섹션에서 선언된 항목들은 다른 유닛들에서는 접근할 수 없다는 것입니다.
임플먼테이션 섹션은 자체의 uses 절을 포함할 수 있는데, 반드시 implementation 바로 뒤에 나타나야 합니다. 임플먼테이션 섹션에서 지정된 유닛들의 식별자들은 임플먼테이션 섹션에서만 사용 가능하며 인터페이스 섹션에서는 참조할 수 없습니다.

uses 절에 대한 자세한 내용은 2장의 "유닛 참조와 uses 절"을 참조하십시오.

initialization 섹션

initialization 섹션은 선택적으로 사용됩니다. 이니셜라이제이션 섹션은 initialization 예약어로 시작하여 finalization 섹션이 시작될 때까지, finalization 섹션이 없는 경우에는 유닛의 끝까지 계속됩니다. 이니셜라이제이션 섹션에는 프로그램이 시작될 때 실행될 문장들을 포함하며, 나타나는 순서대로 실행됩니다. 그러므로 예를 들면 초기화해야 할 데이터 구조를 정의했을 경우, 이니셜라이제이션 섹션에서 초기화를 할 수 있습니다.
인터페이스 uses 리스트의 유닛들의 경우, 외부로부터 uses된 유닛들의 이니셜라이제이션 섹션들은 uses 절에 나타난 순서대로 실행됩니다.

finalization 섹션

finalization 섹션은 선택적이며 initialization 섹션이 있는 유닛에서만 사용할 수 있습니다. 파이널라이제이션 섹션은 finalization 예약어로 시작하여 유닛의 끝까지 계속됩니다. 파이널라이제이션 섹션에는 메인 프로그램이 종료될 때 실행될 문장이 들어 있습니다(프로그램 종료를 위해 Halt 프로시저를 호출한 경우는 예외). 파이널라이제이션 섹션을 사용하면 이니셜라이제이션 섹션에 할당했던 리소스를 해제할 수 있습니다.
파이널라이제이션 섹션은 이니셜라이제이션 섹션과 반대로 실행됩니다. 예를 들어 애플리케이션에서 유닛을 A, B, C 순으로 초기화했다면 C, B, A 순으로 유닛을 해제합니다.
일단 어떤 유닛의 이니셜라이제이션 코드가 실행되기 시작하면, 애플리케이션 종료 시에는 해당 유닛의 파이널라이제이션 섹션의 실행이 보장됩니다. 런타임 에러가 발생하면 이니셜라이제이션 코드가 완전히 실행되지 않을 수도 있기 때문에, 파이널라이제이션 섹션은 불완전하게 초기화된 데이터도 처리할 수 있어야 합니다.

유닛 참조와 uses 절

uses 절은 자신이 포함된 프로그램, 라이브러리 또는 유닛에서 사용하는 유닛들을 나열합니다. (라이브러리에 대한 자세한 내용은 10장 "DLL과 패키지"를 참조하십시오.) uses 절이 나타날 수 있는 곳은 다음과 같습니다.

- 프로그램이나 라이브러리의 프로젝트 파일

- 유닛의 인터페이스 섹션
- 유닛의 임플먼테이션 섹션

유닛들의 인터페이스 섹션처럼, 프로젝트 파일도 대부분 uses 절을 갖고 있습니다. 유닛의 임플먼테이션 섹션도 자신의 uses 절을 포함할 수 있습니다.
System 유닛과 SysInit 유닛은 모든 애플리케이션에서 자동으로 uses 되며, uses 절에 명시적으로 지정할 수 없습니다. (System은 파일 I/O, 문자열 처리, 부동 소수점 연산, 동적 메모리 할당 등에 대한 루틴을 포함합니다.) SysUtils 등의 다른 표준 라이브러리 유닛을 사용하려면 반드시 uses 절에 포함해야 합니다. 일반적으로는, 프로젝트에 유닛을 추가 혹은 제거하면 IDE가 그 유닛에 필요한 유닛들을 uses 절에 추가합니다.
유닛 선언과 uses 절에서, 유닛 이름은 파일 이름과 대소문자까지 일치해야 합니다. 다른 경우에는 유닛 이름은 대소문자를 구별하지 않습니다. 유닛 참조의 문제를 피하려면, 유닛 소스 파일을 명시적으로 참조하십시오.

```
uses MyUnit in "myunit.pas";
```

프로젝트 파일에서 이렇게 명시적으로 uses한 유닛은 프로젝트의 다른 소스 파일들에서는 대소문자가 다르더라도 간단한 uses 절로 참조할 수 있습니다.

```
uses Myunit;
```

uses 절의 위치와 내용에 대한 자세한 내용은 2장의 "복수의 유닛 참조와 간접 유닛 참조"와 "순환적인 유닛 참조"를 참조하십시오.

uses 절의 문법

uses 절은 예약어 uses와 하나 이상의 쉼표로 구분된 유닛 이름, 세미콜론 순으로 구성됩니

```
uses Forms, Main;

uses
  Forms,
  Main;

uses Windows, Messages, SysUtils, Strings, Classes, Unit2, MyUnit
```

다. 예를 들면, 다음과 같습니다.
program이나 library의 uses 절에서는, 유닛 이름 뒤에 예약어 in과 작은 따옴표로 싸인 소스 파일의 이름이 있을 수 있습니다. 이 소스 파일 이름은 디렉토리 패스(절대 패스나 상대 패스)를 포함할 수도 있습니다. 예를 들면 다음과 같습니다.

```
uses
   Windows, Messages, SysUtils,
   Strings in 'C:\Classes\Strings.pas', Classes;
```

유닛의 소스 파일을 지정할 필요가 있을 경우 유닛 이름 뒤에 예약어 in을 사용합니다. IDE에서는 유닛 이름이 유닛의 소스 파일 이름과 일치해야 하므로 보통은 이렇게 할 필요가 없습니다. in을 사용하는 것은 소스 파일의 위치가 명확하지 않을 경우에만 필요합니다. 예를 들면 다음과 같은 경우입니다.

- 프로젝트 파일과 다른 디렉토리에 있는 소스 파일을 uses 했는데, 그 디렉토리가 컴파일러의 서치 패스(Search Path)에 없는 경우.
- 컴파일러의 서치 패스에 같은 이름의 유닛이 있을 경우.
- 커맨드라인에서 콘솔 애플리케이션을 컴파일할 때, 유닛의 소스 파일 이름과 일치하지 않는 유닛 이름을 지정했을 경우.

또한 컴파일러는 어떤 유닛들이 프로젝트의 일부인지 결정할 때 in을 사용합니다. 프로젝트(.dpr) 파일의 uses 절에서 in과 파일 이름이 지정된 유닛들만 프로젝트의 일부로 간주되고, uses 절의 다른 유닛은 프로젝트에 속하지 않는 유닛으로 간주합니다. 이 차이는 컴파일에는 아무 영향도 없지만, Project Manager 같은 IDE 툴에 영향을 줍니다.
유닛의 uses 절에서는 소스 파일의 위치를 지정하기 위해 in을 사용할 수 없습니다. 모든 유닛은 반드시 컴파일러의 서치 패스에 있어야 합니다. 또 유닛 이름은 유닛의 소스 파일 이름과 일치해야 합니다.

복수의 유닛 참조와 간접 유닛 참조

uses 절에 나열되는 유닛들의 순서는 유닛의 초기화 순서를 결정하며("유닛의 구조와 문법"절의 "initialization 섹션" 참조), 컴파일러가 식별자를 찾는 방법에도 영향을 미칩니다. 두 유닛에서 같은 이름의 변수, 상수, 타입, 프로시저나 함수를 선언했을 경우, 컴파일러는

uses 절의 유닛 리스트에서 마지막 유닛의 식별자를 사용합니다. (다른 유닛의 식별자를 액세스하려면 UnitName.Identifier처럼 한정자(qualifier)를 추가해야 합니다)
uses 절에는 해당 프로그램이나 유닛이 직접 참조하는 유닛들만 포함시켜야 합니다. 다시 말해, 유닛 A가 유닛 B에 선언된 상수, 타입, 변수, 프로시저 또는 함수를 참조할 경우 A는 명시적으로 B를 uses 해야 합니다. B가 유닛 C의 식별자를 참조할 경우에 A는 간접적으로 C에 종속됩니다. 이 경우 C를 A의 uses 절에 포함할 필요는 없지만 컴파일러가 A를 처리하려면 여전히 B와 C를 모두 찾을 수 있어야 합니다.
아래 예제는 간접 종속 관계를 보여 줍니다.

```
program Prog;
uses Unit2;
const a = b;
// ...

unit Unit2;
interface
uses Unit1;
const b = c;
// ...

unit Unit1;
interface
const c = 1;
// ...
```

이 예제에서 Prog는 Unit2에 직접적으로 종속되고, Unit2는 Unit1에 직접적으로 종속됩니다. 그러므로 Prog는 Unit1에 간접적으로 종속됩니다. Unit1은 Prog의 uses 절에 없기 때문에 Prog에서는 Unit1에서 선언된 식별자를 사용할 수 없습니다.
컴파일러가 대상 모듈을 컴파일하기 위해서는 그 대상 모듈이 직접 또는 간접적으로 종속되어 있는 모든 유닛을 찾을 수 있어야 합니다. 그러나 이들 유닛의 소스 코드가 변경되지 않았다면 컴파일러는 소스(.pas) 파일이 아닌 .dcu 파일만 있으면 됩니다.
유닛의 인터페이스 섹션을 변경할 경우 이 유닛에 종속된 다른 유닛도 반드시 다시 컴파일해야 합니다. 하지만 유닛의 임플먼테이션 섹션이나 다른 섹션만 변경할 경우에는 종속된 유닛을 다시 컴파일할 필요가 없습니다. 컴파일러는 자동으로 이러한 종속 관계를 추적하고 필요할 때에만 유닛을 다시 컴파일합니다.

순환적인 유닛 참조

유닛이 직접 혹은 간접적으로 서로 참조할 경우, 이러한 유닛들을 상호 종속 관계에 있다고 합니다. 상호 종속 관계는 어떤 인터페이스 섹션의 uses 절과 또 다른 인터페이스 섹션의 uses 절을 연결하는 순환 패스가 없을 경우에만 가능합니다. 다시 말해, 어떤 유닛의 인터페이스 섹션에서 지정한다른 유닛의 인터페이스 섹션에서 참조를 따라가서 다시 원래 참조를 시작한 유닛으로 돌아오게 해서는 절대로 안됩니다. 상호 종속 패턴이 올바르려면 순환 참조에서 적어도 하나는 임플먼테이션 섹션의 uses 절을 사용해야 합니다.
가장 간단한 예로, 상호 종속하는 두 유닛의 인터페이스 섹션의 uses 절에서 서로를 지정할 수 없다는 것을 의미합니다. 따라서 다음 예제에서는 컴파일 에러가 발생합니다.

```
unit Unit1;
interface
uses Unit2;
// ...

unit Unit2;
interface
uses Unit1;
// ...
```

하지만 유닛 참조들 중 하나를 임플먼테이션 섹션으로 옮기면 두 유닛은 서로 참조할 수 있게 됩니다.

```
unit Unit1;
interface
uses Unit2;
// ...

unit Unit2;
interface
//...

implementation
uses Unit1;
// ...
```

순환 참조가 일어날 가능성을 줄이려면, 가능한 한 임플먼테이션 섹션의 uses 절에서 유닛을 나열하는 것이 좋습니다. 다른 유닛의 식별자가 인터페이스 섹션에서 사용될 경우에만 인터페이스 섹션의 uses 절에서 유닛을 사용해야 합니다.