1. 부: C#

1 장: 들어가면서

[책 내용]

2 장: 개발환경 준비(닷넷 프레임워크)

우선 비주얼 스튜디오 2022를 이미 설치한 사용자라면 곧바로 3장으로 넘어가도 된다. 하지만 그 외의 환경이라면 이 책에 실린 예제 코드를 실습하기 위해 약간의 개발 환경 설정이 필요하다.

최신 버전인 C# 10을 완전하게 실습하려면 닷넷 6가 필요하므로 다음 URL에서 별도로 다운로드해 설치할 수 있다.

.NET 6 SDK 설치 https://dotnet.microsoft.com/download/dotnet/6.0

만약 닷넷 프레임워크 4.8을 사용한다면 C# 8.0의 기능 중 "기본 인터페이스 메서드"와 C# 9.0의 "공변 반환 형식"을 제외하고 모두 실습할 수 있다. 따라서 그 두 개의 기능을 제외한다면 ".NET Framework 4.8 개발자 팩"을 설치해 이 책의 예제를 따라할 수 있다.

.NET Framework 4.8 Developer Pack 설치 https://learn.microsoft.com/en-us/dotnet/framework/install/guide-for-developers

정상적으로 닷넷 프레임워크가 설치돼 있다면 32비트/64비트 운영체제에 따라 각각 다음 폴더가 생성된 것을 확인할 수 있다.

32비트: C:₩Windows₩Microsoft.NET₩Framework₩v4.0.30319 64비트: C:₩Windows₩Microsoft.NET₩Framework64₩v4.0.30319

개인적인 의견이지만 취미로 개발하는 것이 아니라면 개발 환경을 영문으로 구성할 것을 권장한다. 운영체제도 '영문' 버전으로 설치하고, 닷넷 프레임워크 및 개발 툴도 '영문' 버전으로 설치하는데, 이렇게 하는 이유는 오류가 발생한 경우 해결책을 찾기 위해 검색엔진을 이용했을 때 높은 확률로 해답을 찾을 수 있기 때문이다. 또한 개발자를 직업으로 삼고 싶다면 앞으로의 지식 습득을 위해 어차피 영어와 친숙해져야 하기 때문에 꼭 영작이나 영어로 말하는 수준은 아닐지라도 기본적인 프로그래밍 용어를 영문으로 익혀두는 것이 도움이 될 수 있다.

참고!	영문 버전을 권장한다고 썼지만 이 책의 모든 비주얼 스튜디오 화면은 한글 버전으로 진행한다. 대신 영문 사용자에게도 혼동이 안 되도록 메뉴와 기능 이름에 대해 한글/영문을 병기했다.

2.1 기본 예제 (닷넷 프레임워크)

프로그래밍 공부에는 반드시 '실습'이 뒤따른다. 하지만 어느 정도의 문법적 지식이 쌓이기 전까지는 초보 입장에서 완전히 실행 가능한 예제를 직접 만드는 것은 어려울 수 있으므로 간단한 예제 코드를 먼저 제시할 필요가 있다. 재미있는 것은, 그 기본적인 예제를 완전하게 이해하려면 다시 문법적인 설명이 뒤따라야 하기 때문에 결국 '닭이 먼저냐, 달걀이 먼저냐'라는 문제에 빠지게 된다는 점이다. 따라서 지금은 궁금하더라도 자세한 문법은 이 책을 읽어나가면서 자연스럽게 이해할 수 있으니 일단 무조건 다음의 예제 코드를 메모장(notepad.exe)에 입력하고 Program.cs 파일로 저장하자.

예제 1.1 기본 예제 파일[1] – Program.cs

```
// c:₩temp 폴더에 Program.cs 파일명으로 저장
using System;

namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            // 문자열 출력
            Console.WriteLine("Hello World");
        }
    }
}
```

예제 1.1에 나온 프로그램은 단순히 화면에 "Hello World"라는 문자열을 출력하는 일을 하는데, Console.WriteLine("……[출력하고 싶은 문자열]……") 문장이 그런 역할을 한다. 또한 대부분의 프로그래밍 언어는 주석(Comment)이라는 구문을 제공하는데 주석을 이용하면 프로그램의 실행과는 무관하게 개발자가 임의로 일련의 텍스트를 적어 넣을 수 있다. 예제 1.1에서는 "// 문자열 출력" 이 주석에 해당한다. 즉, "//" 문자 이후에 같은 줄에 속한 모든 단어는 프로그램 실행에서 완전히 배제된다. 그 밖의 코드가 하는 역할은 이 책을 읽어가면서 차차 이해하게 될 것이다.

이처럼 예제 1.1은 '사람'이 이해할 수 있는 형식을 갖추고 있고 이를 '소스코드(Source code)'라 한다. 당연히 기계는 소스코드를 직접 이해할 수 없기 때문에 이런 문자열을 컴퓨터가 이해할 수 있는 일련의 기계어로 바꿔야 하고, 이 과정을 "컴파일(Compile)"이라고 한다. 컴파일 명령을 내리

---

[1] C# 9.0부터 기본 예제 파일을 한 줄로 표현할 수 있다. (**Error! Reference source not found.**절 '**Error! Reference source not found.**' 참고)

는 방법은 다양하므로 이야기를 나눠서 설명할 필요가 있다.

2.2 편집기 + 명령행 컴파일러 사용 (닷넷 프레임워크)

컴파일을 해주는 프로그램을 컴파일러(Compiler)라고 한다. 닷넷 프레임워크 또는 닷넷 코어/5+를 설치하면 기본적으로 C# 문법을 따르는 소스코드를 컴파일할 수 있는 C# 컴파일러 프로그램이 함께 설치된다. 따라서 메모장(notepad.exe) 등의 편집기를 이용해 소스코드를 입력한 후 파일로 저장하고 명령 프롬프트(cmd.exe)에서 C# 컴파일러[2]를 직접 구동해 해당 파일을 컴파일할 수 있다.

참고로 명령행에서 컴파일러를 이용하는 방식으로는 이 책의 1부와 2부의 내용을 공부할 수 있지만 3부의 내용을 따라 하는 것은 쉽지 않다. 또한 대부분의 경험 있는 윈도우 개발자들에게도 편집기와 명령행만으로 프로그램을 만드는 경우는 흔치 않다. 따라서 여러분도 이런 방법이 있다는 사실만 알고 실제로 이 책의 내용을 학습할 때는 다음 절에서 설명하는 전용 개발 환경을 이용할 것을 권장한다.

2.2.1 닷넷 프레임워크 명령행 (닷넷 프레임워크)

닷넷 프레임워크의 경우 운영체제별로 설치된 컴파일러 경로가 달라질 수 있으니 자신의 컴퓨터 환경을 먼저 확인하고 그에 맞게 선택해야 한다.

- 32비트 운영체제의 csc.exe 위치

32비트: C:\Windows\Microsoft.NET**Framework**\v4.0.30319**csc.exe**

- 64비트 운영체제의 csc.exe 위치

32비트: C:\Windows\Microsoft.NET**Framework**\v4.0.30319**csc.exe**
64비트: C:\Windows\Microsoft.NET**Framework64**\v4.0.30319**csc.exe**

다행히 32비트와 64비트 모두 "C:\Windows\Microsoft.NET\Framework\v4.0.30319"에 csc.exe 파일이 들어 있기 때문에 별도로 언급하지 않는 한 편의상 32비트 csc.exe로 예제를 컴파일한다고 가정한다.

그럼, 이제 실제로 컴파일을 해보자. 예제 1.1을 저장한 c:\temp\Program.cs 파일을 명령 프롬프트를 이용해 컴파일하는 경우 다음과 같이 csc.exe 프로그램을 실행하면 된다.

---

[2] 닷넷 프레임워크에 포함된 C# 컴파일러는 C# 6.0 이상의 구문을 지원하지 않는다. 명령행에서 C# 6.0 이상의 소스코드를 컴파일하고 싶다면 http://www.sysnet.pe.kr/2/0/11266을 참고한다.

```
c:\temp>C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc Program.cs
Microsoft (R) Visual C# Compiler version 4.7.3190.0
for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.

This compiler is provided as part of the Microsoft (R) .NET Framework, but only supports language versions up to C# 5, which is no longer the latest version. For compilers that support newer versions of the C# programming language, see http://go.microsoft.com/fwlink/?LinkID=533240

c:\temp>dir
Volume in drive C has no label.
 Volume Serial Number is FCD6-4B58

 Directory of C:\temp

2013-02-15  오전 10:16    <DIR>          .
2013-02-15  오전 10:16    <DIR>          ..
2013-02-15  오전 10:15               221 Program.cs
2013-02-15  오전 10:16             3,584 Program.exe
               2 File(s)          3,805 bytes
               2 Dir(s)  24,232,890,368 bytes free

C:\temp>Program.exe
Hello World
```

정상적으로 컴파일되면 Program.exe 파일이 생성되고, 이를 실행하면 "Hello World"라는 문자열이 출력되는 것을 확인할 수 있다.

프로그램에 오타가 있거나 C# 컴파일러가 허용하지 않는 문법이 있다면 csc 프로그램은 그에 해당하는 오류 메시지를 화면에 출력한다. 예를 들어, 예제 소스코드의 Console.Write**Line**을 일부러 Console.Write**Lin**으로 오타를 낸다면 다음과 같은 컴파일 오류 메시지를 볼 수 있다.

```
c:\temp>C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc Program.cs
Microsoft (R) Visual C# Compiler version 4.7.3190.0
for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.

Program.cs(10,17): error CS0117: 'System.Console' does not contain a definition for
            'WriteLin'
c:\temp>
```

컴파일러는 오류 메시지와 함께 소스코드의 어느 부분이 잘못됐는지를 나타내는 행(Line)과 열(Column) 정보를 출력한다. "Program.cs(10,17)"을 보면 10번째 라인의 17번째 문자에 해당하는 구문을 C# 컴파일러가 이해할 수 없다는 것이므로 편집기에서 그곳을 찾아 오류를 수정하면 된다. 컴파일 오류가 발생한다고 해서 당황할 필요는 없다. 오류 메시지가 영어라고는 하지만 대부분 중학교 수준의 영어 실력만으로도 충분히 해석할 수 있으므로 침착하게 오류 메시지를 살펴보면 의미를 어렵지 않게 알 수 있고 왜 오류가 발생했는지 이해할 수 있다.

참고!	오류가 발생한 경우 직접 찾아서 해결하는 것도 결국 '경험'으로 남는다. 많은 오류를 경험하고 많은 오류를 수정할수록 배우는 속도는 더딜 수 있지만 그만큼 철저하게 몸으로 터득할 수 있다.

3 장: C# 기초

[책 내용]

4 장: C# 객체 지향 문법

[책 내용]

5 장: C# 1.0 완성하기

[책 내용]

5.1 문법 요소

5.1.1 구문

5.1.1.1 전처리기 지시문 (닷넷 프레임워크)

C#의 전처리기 지시문(preprocessor directive)은 특정 소스코드를 상황에 따라 컴파일 과정에서 추가/제거하고 싶을 때 사용한다. 예를 들어, 다음의 Program.cs 파일을 보자.

```
static void Main(string[] args)
{
    string txt = Console.ReadLine();

    if (string.IsNullOrEmpty(txt) == false)
    {
        Console.WriteLine("사용자 입력: " + txt);
    }
}
```

Console.ReadLine 메서드는 엔터(Enter) 키가 눌릴 때까지의 키보드 입력을 받는 역할을 한다. 즉, Console.ReadLine 메서드가 실행되면 콘솔 화면에는 입력을 기다리는 프롬프트가 깜빡이고 키보드로 입력된 내용을 내부적으로 저장해 뒀다가 엔터 키가 눌리면 반환값으로 돌려준다. 이어서 string.IsNullOrEmpty 메서드는 인자로 들어온 string 객체가 null이거나 빈 문자열("")을 담고 있으면 true를 반환하고 1개 이상의 문자를 담고 있다면 false를 반환한다. 따라서 위의 코드는 사용자에게서 입력받은 내용이 "test"라면 화면에 "사용자 입력: test"라는 문자열을 출력하고, 그냥 아무 내용 없이 엔터 키만 눌렀다면 아무것도 하지 않고 종료한다.

그런데 위의 프로그램을 사용하는 특정 고객으로부터 아무 입력도 받지 않은 경우에 "입력되지 않음"이라는 메시지를 출력해 달라는 요청을 받았다고 가정해 보자. 이 요청을 반영하려면 소스코드를 다음과 같이 고쳐야 한다.

```
if (string.IsNullOrEmpty(txt) == false)
{
    Console.WriteLine("사용자 입력: " + txt);
}
else
{
    Console.WriteLine("입력되지 않음");
}
```

문제는 모든 고객이 이런 변경을 찬성하지 않았을 때 발생한다. 그렇다면 원본 Program.cs를 복사해서 Program2.cs 파일을 만들어 위의 변경사항을 관리해야 한다. 그래서 Program.cs와 Program2.cs로부터 각각 program.exe를 만들어서 원하는 고객에게 전달하게 된다.

이렇게 소스코드 파일이 나뉘면 여러 가지 문제가 발생한다. 하나의 소스코드에서 변경된 사항을 다른 소스코드에 반영해야 하는 관리상의 부담이 생긴다. 메서드를 통해 중복 코드를 방지한 것처럼 소스코드 파일 역시 중복되는 것은 지양해야 한다.

바로 이럴 때 사용할 수 있는 기법이 #if/#endif 전처리기 지시문이다.

```
static void Main(string[] args)
{
    string txt = Console.ReadLine();
    if (string.IsNullOrEmpty(txt) == false)
    {
        Console.WriteLine("사용자 입력: " + txt);
    }
#if OUTPUT_LOG
    else
    {
        Console.WriteLine("입력되지 않음");
    }
#endif
}
```

위 예제에서는 OUTPUT_LOG 전처리 상수(preprocessor constant)가 정의돼 있으면 #if / #endif 사이의 소스코드를 포함해서 컴파일하게 만들고, 정의돼 있지 않으면 컴파일 과정에서 제거한다. 개발자는 컴파일러에게 /define 옵션을 통해 전처리 상수를 설정할 수 있다.

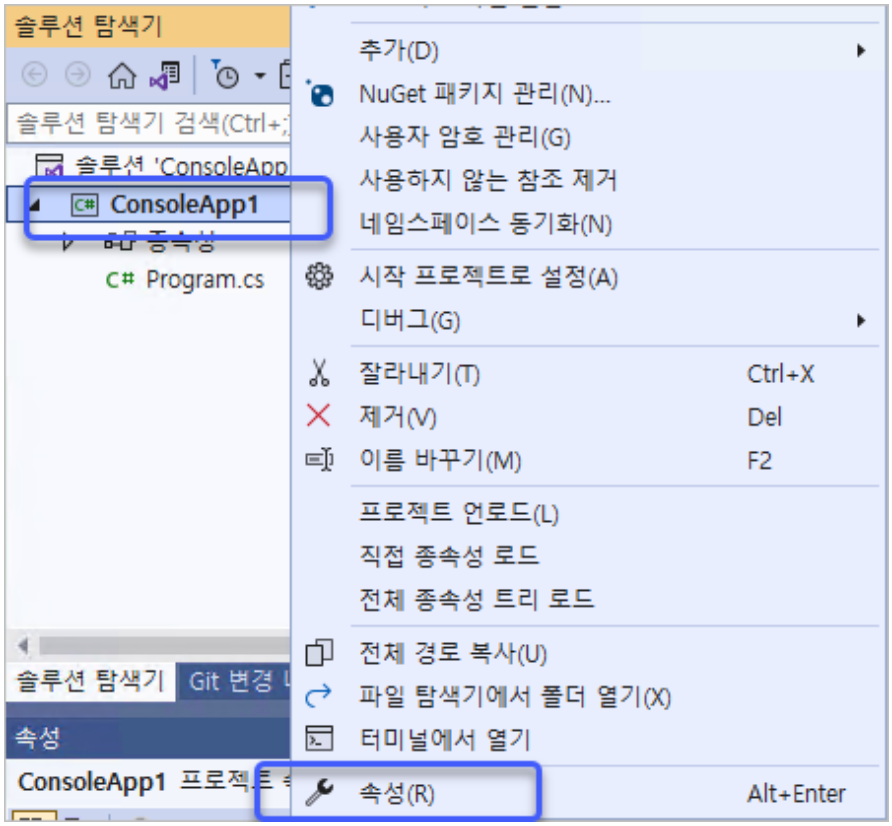
[OUTPUT_LOG가 정의되지 않은 컴파일]
csc program.cs

[OUTPUT_LOG가 정의된 컴파일]

```
csc /define:OUTPUT_LOG program.cs
```

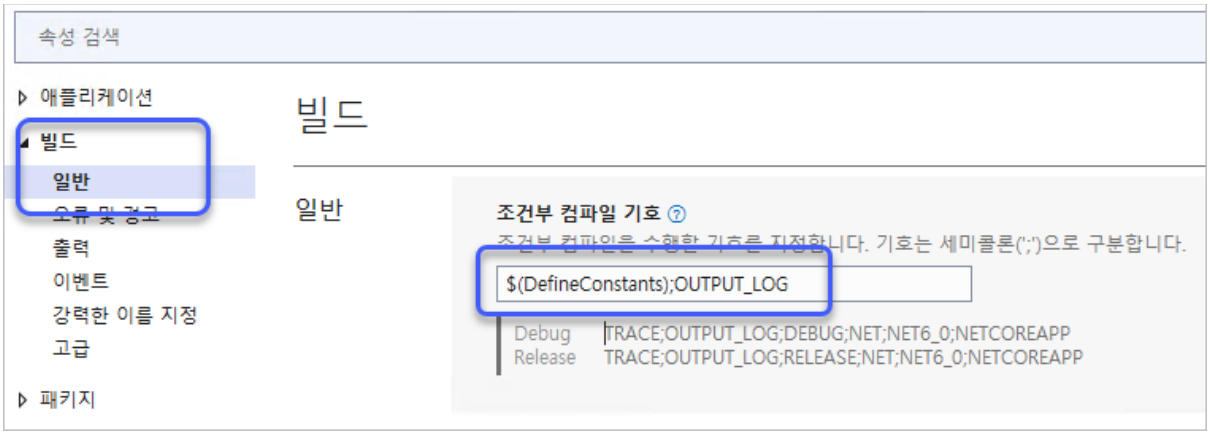
비주얼 스튜디오 환경에서는 솔루션 탐색기에서 프로젝트 항목을 선택한 후 마우스 오른쪽 버튼을 누르면 나오는 메뉴에서 "속성(Properties)"을 선택한다.

그림 1.1 프로젝트 속성 창을 띄우는 방법



잠시 후 "프로젝트 속성" 창이 나타나면 "빌드(Build)" / "일반(General)" 항목을 선택하고 "조건부 컴파일 기호(Conditional compilation symbols)" 입력 상자에 기존 값 " $(DefineConstants)"에 이어 세미콜론을 구분자로 추가한 다음 "OUTPUT_LOG를 입력한다.

그림 1.2 프로젝트 속성 창의 빌드 / 조건부 컴파일 기호 설정



참고!	프로젝트 속성 창에는 이 밖에도 여러 부가 옵션이 있는데, 지면 관계상 이 책에서 모든 옵션을 다루지는 않는다.

이렇게 전처리기를 이용하면 하나의 소스코드 파일로 여러 가지 상황을 만족하는 프로그램을 만들 수 있다.

#if, #endif가 조건문이기에 이를 보완하기 위한 #else, #elif 지시자도 있다. 그리고 전처리기 기호를 csc.exe의 옵션을 지정하지 않고 소스코드에서 직접 지정할 수 있도록 #define 문도 제공되며, 반대로 정의를 취소할 수 있는 #undef 문도 있다. 다음은 이를 사용한 간단한 예제 코드다.

```
#define __X86__
#undef OUTPUT_LOG

using System;

class Program
{
    static void Main(string[] args)
    {
#if OUTPUT_LOG
        Console.WriteLine("OUTPUT_LOG가 정의됨");
#else
        Console.WriteLine("OUTPUT_LOG가 정의 안 됨");
#endif

#if __X86__
        Console.WriteLine("__X86__ 정의됨");
#elif __X64__
        Console.WriteLine("__X64__ 정의됨");
#else
        Console.WriteLine("아무것도 정의 안 됨");
#endif
    }
}
```

참고!	#define / #undef 문은 반드시 소스코드보다 먼저 나타나야 한다. 일례로, 위의 소스코드에서 #define 구문을 using 문 다음으로 옮기면 컴파일할 때 "Cannot define/undefined preprocessor symbols after first token in file" 오류가 발생한다.

이 밖에도 #warning, #error, #line, #region, #endregion, #pragma 지시문이 있지만 지면 관계상 생략한다.

5.2 프로젝트 구성 (닷넷 프레임워크)

프로젝트(project)는 비주얼 스튜디오의 소스코드 관리를 위해 도입된 개념이다. 한 프로젝트는 여러 개의 소스코드를 담을 수 있고, 해당 프로젝트를 빌드하면 하나의 EXE 또는 DLL 파일이 만들어진다.

프로젝트를 생성하면 그 프로젝트에서 관리하는 모든 정보를 담는 "프로젝트 파일"이 만들어진다. 솔루션 탐색기에서 "프로젝트" 항목을 대상으로 마우스 오른쪽 버튼을 누르면 "파일 탐색기에서 폴더 열기(Open Folder in File Explorer)" 메뉴가 나온다. 이를 선택하면 윈도우 탐색기가 실행되고 곧바로 프로젝트 파일을 담은 폴더를 보여준다.

프로젝트 파일은 언어마다 확장자가 다르다. 이 책에서 배우는 C# 언어의 경우 비주얼 스튜디오가 생성하는 프로젝트 파일의 확장자는 "csproj"다. 프로젝트 이름이 "ConsoleApp1"이라면 프로젝트 파일은 "ConsoleApp1.csproj"가 되고 탐색기에서 이 파일을 찾을 수 있다. 윈도우의 "메모장"으로 프로젝트 파일을 열어 보자.

```
<?xml version="1.0" encoding="utf-8"?>
<Project ……[생략]……>
  <Import ……[생략]…… />
  <PropertyGroup ……[생략]……>
……[생략]……
  </PropertyGroup>
  <ItemGroup>
……[생략]……
  </ItemGroup>
……[생략]……
</Project>
```

참고!	프로젝트(csproj) 파일의 내용은 XML(eXtensible Markup Language) 형식을 따른다. 닷넷 전반적으로 XML이 많이 사용되므로 이쯤에서 간략하게 XML을 설명하고 넘어가

	겠다. XML에서 꺾쇠 괄호(<, >)를 사용해 의미를 나타내는 문자열을 둘러싼 것을 "태그(tag)"라고 한다. 즉 "<Project>"와 "</Project>"는 태그인데, 전자는 열림 태그, 후자는 닫힘 태그라고 하며, 반드시 쌍으로 존재해야 한다. 태그 사이에는 또 다른 태그가 올 수 있는데, 이를 자식 태그라고 한다. 위에서 Import, PropertyGroup, ItemGroup은 모두 Project 태그의 자식이다. 또한 태그 사이에 값을 넣는 것도 가능하다. 예를 들어, <WarningLevel>4</WarningLevel>은 WarningLevel 태그에 4라는 값을 넣은 것이다. 값을 담고 있지 않는 태그는 열림 태그와 닫힘 태그를 합쳐서 <WarningLevel />과 같이 표현하는 것도 가능하다. 태그 사이가 아닌, 태그 안에 값을 넣는 것도 가능하다. 예를 들어, <Compile Include="Program.cs" />에서 Compile 태그는 Include 속성을 포함한다고 말한다. Include 속성의 값은 Program.cs다.

프로젝트에 소스코드 파일을 추가하거나 프로젝트의 속성 창을 통해 설정을 바꾸는 경우 변경 사항은 모두 프로젝트 파일에 보관된다. 지금까지 이 책에서 실습한 비주얼 스튜디오의 프로젝트 유형은 2.3절 '비주얼 스튜디오 개발 환경'에서 생성한 "콘솔 앱(Console App)"에 해당한다. 그 밖의 프로젝트 유형들도 기본 구조는 콘솔 앱과 비슷하며, 단지 상황에 따라 csproj 파일 내부의 옵션이 변경되는 정도에 불과하다.

프로젝트보다 큰 단위가 솔루션(Solution)이다. 일반적으로 프로그램을 한 개의 프로젝트로 만드는 경우는 거의 없다. 여러 개의 프로젝트가 모여 하나의 솔루션을 구성한다. 비주얼 스튜디오에서 "솔루션 탐색기"를 통해 보여주는 내용이 바로 "솔루션"으로서 이 역시 파일로 저장된다. 솔루션 탐색기에서 최상단의 솔루션 항목을 대상으로 마우스 오른쪽 버튼을 누르면 "파일 탐색기에서 폴더 열기" 메뉴가 나오고 이를 선택하면 확장자가 sln인 파일이 들어 있는 폴더를 보여준다. 메모장으로 솔루션 파일을 열면 솔루션에 등록된 프로젝트의 경로가 포함된 것을 확인할 수 있다.

예를 들어, 솔루션과 프로젝트의 관계를 마이크로소프트 오피스 제품으로 비교해 보면 다음과 같다.

표 1.1 솔루션과 프로젝트의 관계

솔루션	프로젝트
오피스(Office)	워드(Word)
	엑셀(Excel)
	파워포인트(PowerPoint)

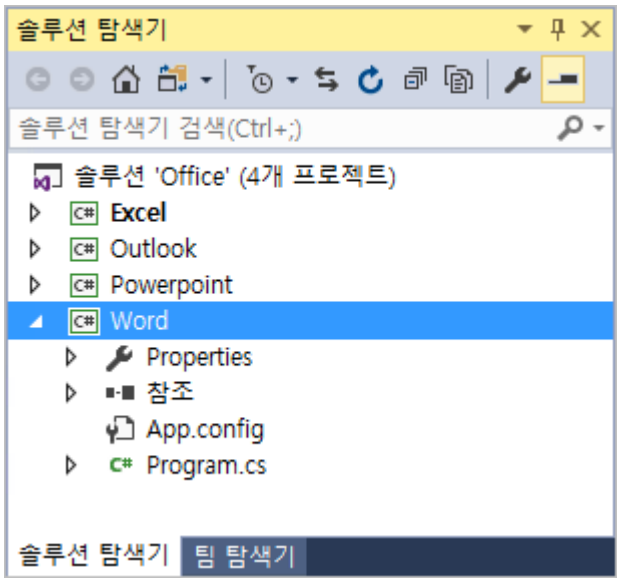
	아웃룩(Outlook)

위의 제품 구조를 비주얼 스튜디오로 표현해 보자. 우선 "파일(File)" → "새로 만들기(New)" → "프로젝트(Project...)" 메뉴를 선택하면 나타나는 대화상자에서 "프로젝트 형식"의 필터를 "기타(Other)"로 설정하면 "빈 솔루션(Blank Solution)" 유형이 나타나는데 이를 선택하고 다음(Next) 버튼을 누른다. 입력 화면이 나오면 이름으로 Office를, 폴더 위치는 원하는 경로로 적당하게 입력한 다음 "만들기(Create)" 버튼을 누른다.

그림 1.3 솔루션 생성

그다음 "파일(File)" → "새로 만들기(New)" → "프로젝트(Project...)" 메뉴를 선택해 차례대로 4개의 콘솔 앱 형식의 프로젝트를 솔루션 아래에 추가한다. 최종적으로 다음과 같은 구조의 솔루션이 만들어진다.

그림 1.4 4개의 프로젝트를 소유한 솔루션



이처럼 비주얼 스튜디오를 이용해 프로그램을 만든다면 맨 먼저 하는 작업이 솔루션을 만드는

것이다. 그리고 그 아래에 제품을 구성하기 위한 프로젝트를 만든다. 따라서 비주얼 스튜디오로 만들어진 프로그램의 소스코드를 보관하려면 프로젝트 파일과 솔루션 파일까지 함께 저장해야 한다.

이제 비주얼 스튜디오를 종료하고, 탐색기에서 솔루션 파일(Office.sln)을 마우스로 두 번 눌러보자. 그럼 비주얼 스튜디오가 다시 실행되면서 솔루션 탐색기가 재구성되는 것을 볼 수 있다. 여러 개의 프로그램을 동시에 개발하고 있다면 그 수만큼의 비주얼 스튜디오를 띄우고 개별 솔루션을 불러와야 한다.

이어지는 절에서는 프로젝트와 관련된 사항을 좀 더 자세하게 설명한다.

5.2.3 응용 프로그램 구성 파일: app.config (닷넷 프레임워크)

참고!	이번 절의 내용은 닷넷 프레임워크용 프로젝트에서만 실습할 수 있다. 닷넷 코어/5+ 환경의 경우 기존 닷넷 프레임워크와의 호환을 위해서만 지원하기 때문에 이번 절의 내용 중 "appSettings"만 실습할 수 있다. 또한 마이크로소프트는 처음 닷넷 코어를 만들 때 기존 닷넷 프레임워크의 BCL을 1:1로 가져오지 않았다. 그 한 예가 바로 이번 절에서 실습할 Configuration 관련 타입들인데, 닷넷 코어부터는 이것을 별도의 패키지로 분리시켜 배포하고 있다. 따라서 이번 실습을 닷넷 코어/5+ 프로젝트에서 실습하려면 System.Configuration.ConfigurationManager 패키지를 수동으로 프로젝트에 추가해야 한다. 이를 위해 "도구" → "NuGet 패키지 관리자" → "패키지 관리자 콘솔" 메뉴를 선택한 후 뜨는 "패키지 관리자 콘솔" 창에서 다음과 같은 명령어를 입력해 해당 패키지를 설치한다. `Install-Package System.Configuration.ConfigurationManager` 필자는 이번 절의 실습을 가능한 닷넷 프레임워크용 프로젝트를 만들어 실습하길 권장한다.

닷넷 응용 프로그램을 실행하면 맨 처음 CLR 환경이 초기화된 후 개발자가 작성한 코드가 실행되는 순서로 진행된다. 그런데 가끔은 CLR 초기화 과정에 어떤 값을 전달해야 할 때가 있다. 아쉽게도 이를 위해 C# 코드를 작성할 수는 없다. 왜냐하면 C# 코드는 CLR이 초기화된 후에야 실행되기 때문이다.

이런 경우를 위해 닷넷은 app.config라는 설정 파일을 제공한다. 이 파일을 추가하는 가장 쉬운 방법은 솔루션 탐색기의 프로젝트 항목을 대상으로 마우스 오른쪽 버튼을 눌러 "추가(Add)" → "새 항목(New Item)..."을 선택해 "애플리케이션 구성 파일(Application Configuration File)" 항목을 선택하는 것이다. 그러면 프로젝트에 다음과 같은 내용이 담긴 app.config 파일이 추가된다.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
</configuration>
```

참고!	비주얼 스튜디오에서 닷넷 프레임워크용 실행 파일 유형의 프로젝트(예: 콘솔 앱)를 생성한 경우 기본적으로 app.config 파일이 포함돼 있으며 이후에 설명할

	supportedRuntime 값이 채워져 있다.

app.config 파일은 XML 형식을 따른다. 현재는 configuration 값이 비어 있지만 그 아래에는 다양한 설정 태그가 들어갈 수 있다. 일단 비어있는 상태에서 프로젝트를 빌드해 보자. 그런 다음 솔루션 탐색기에서 "파일 탐색기에서 폴더 열기" 메뉴를 선택해 프로젝트가 위치한 폴더를 탐색기로 연다. 그리고 그 아래의 /bin/Debug 폴더 또는 /bin/Debug/net6.0 폴더로 이동하면 다음과 같은 파일을 확인할 수 있다.

- 닷넷 프레임워크

ConsoleApp1.exe
ConsoleApp1.exe.config

- 닷넷 코어/5+

ConsoleApp1.dll
ConsoleApp1.dll.config
ConsoleApp2.exe

app.config 파일명은 프로젝트에 포함돼 있을 때의 이름일 뿐, 빌드하고 나면 [응용 프로그램 이름].[exe 또는 dll].config 파일명으로 자동으로 변경된다는 점을 기억해 두자.

아쉽게도 app.config에서 허용되는 모든 설정값을 나열하는 것은 이 책의 범위를 넘어서므로 생략하고, 여기서는 특히 자주 쓰이는 두 가지 설정에 대해 알아보겠다.

5.2.3.1 supportedRuntime (닷넷 프레임워크)

참고!	이 옵션을 실습하려면 컴퓨터에 닷넷 프레임워크 3.5를 설치해야 한다. 아직 설치돼 있지 않다면 아래의 경로에서 내려받아 설치한다. Microsoft .NET Framework 3.5 http://www.microsoft.com/en-us/download/details.aspx?id=21

닷넷 프레임워크는 그동안 1.0, 1.1, 2.0, 3.0, 3.5, 4.0, 4.5, 4.6, 4.7, 4.8의 순서로 제품이 출시됐다. 1.0과 1.1은 현재 거의 사용되지 않으므로 2.0 이상으로 범위를 좁혀 보자. 그런데 CLR 입장에서 보면 2.0, 3.0, 3.5는 동일한 버전이다. 왜냐하면 닷넷 2.0이 출시된 후 몇몇 라이브러리를 추가해서 3.0이 됐고, 다시 또 다른 라이브러리가 추가 및 개선되어 3.5가 됐기 때문에 근간이 되는 CLR 자체는 변화가 없었다. 닷넷 4.0은 3.5까지 그대로였던 CLR 2.0을 개선해서 CLR 4.0이 적용됐다. 만약 시스템에 닷넷 2.0, 3.0, 3.5, 4.0이 설치돼 있다면 "C:\Windows\Microsoft.NET\Framework" 폴더

에 각각 "v2.0.50727", "v3.0", "v3.5", "v4.0.30319"라는 이름의 하위 폴더가 있을 것이다.

참고!	64비트 운영체제라면 "C:\Windows\Microsoft.NET" 폴더 아래에 32비트용 "Framework" 폴더와 64비트용 "Framework64" 폴더가 함께 있다.

4.0까지의 닷넷 버전들은 설치할 때마다 추가 폴더가 생성된 것에 반해 닷넷 4.5 ~ 4.8에서는 새로운 폴더를 생성하는 정책을 유지하지 않고 닷넷 4.0을 완전히 교체하는 방식으로 설치된다. 따라서 닷넷 4.5, 4.6, 4.7, 4.8을 설치했다고 해서 그에 대응되는 별도의 폴더가 생성되지는 않고 여전히 폴더의 이름이 "v4.0.30319"로 나타난다.

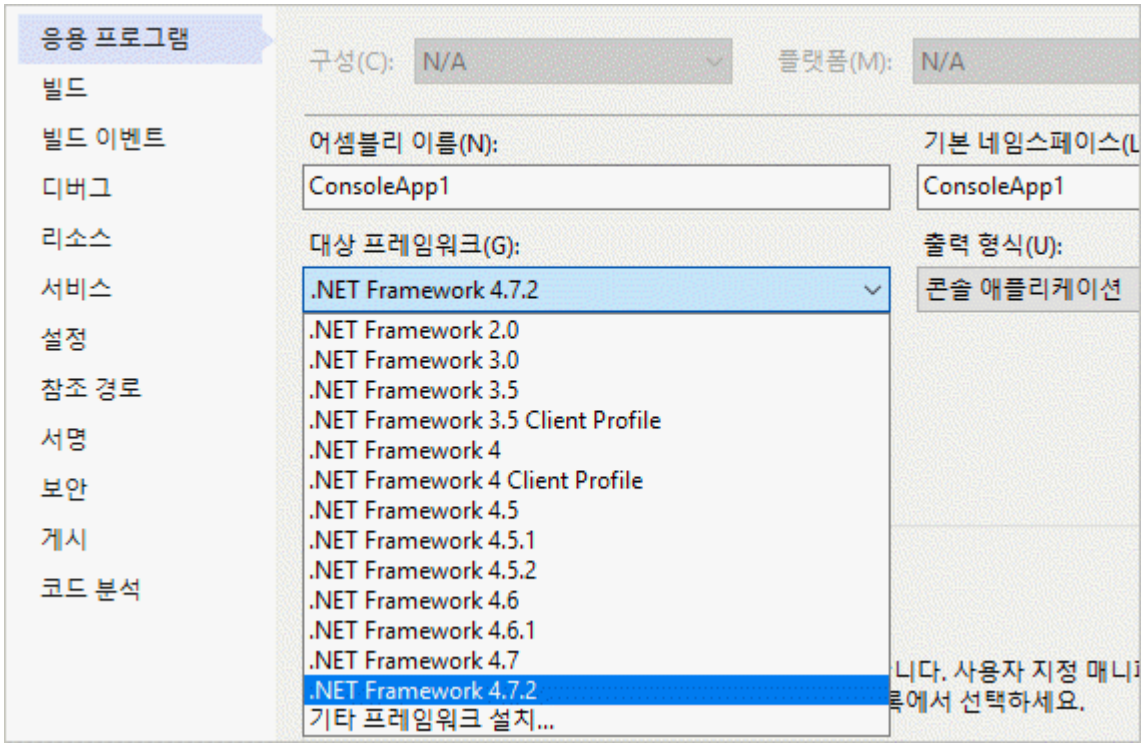
표 1.2 닷넷 버전 이력

<table>
<tr><th>닷넷 버전</th><th>CLR 버전</th><th>설치 폴더명</th><th>C# 릴리스</th><th>설명</th></tr>
<tr><td>.NET 2.0</td><td rowspan="3">CLR 2.0</td><td>/v2.0.50727</td><td rowspan="2">C# 2.0</td><td>닷넷 프레임워크 2.0 릴리스</td></tr>
<tr><td>.NET 3.0</td><td>/v3.0</td><td>닷넷 2.0 + 4가지 주요 라이브러리</td></tr>
<tr><td>.NET 3.5</td><td>/v3.5</td><td>C# 3.0</td><td>닷넷 3.0 + 라이브러리 보강</td></tr>
<tr><td>.NET 4.0</td><td rowspan="5">CLR 4.0</td><td rowspan="5">/v4.0.30319</td><td>C# 4.0</td><td>닷넷 프레임워크 4.0 릴리스</td></tr>
<tr><td>.NET 4.5</td><td>C# 5.0</td><td rowspan="4">닷넷 프레임워크 4.0 교체</td></tr>
<tr><td>.NET 4.6</td><td rowspan="3">로슬린(Roslyn) 컴파일러로 바뀌면서 독자적으로 버전 업그레이드</td></tr>
<tr><td>.NET 4.7</td></tr>
<tr><td>.NET 4.8</td></tr>
</table>

닷넷의 이력이 중요한 이유는 닷넷 응용 프로그램을 배포할 때 CLR 버전이 맞지 않으면 실행이 안 되기 때문이다. CLR 4.0에 해당하는 닷넷 응용 프로그램은 CLR 2.0에 해당하는 닷넷 프레임워크만 설치된 윈도우에서는 실행할 수 없다. 그 반대의 경우도 마찬가지다. 물론, CLR 버전이 같아도 닷넷 버전에 따라 기능상의 차이가 있으므로 만약 닷넷 3.0에서만 제공되는 BCL 클래스를 사용하는 응용 프로그램을 만들었다면 닷넷 2.0에서 실행하면 정상적으로 동작하지 않는다.

비주얼 스튜디오 커뮤니티 2019를 이용해 프로젝트를 만들면 기본적으로 닷넷 4.8 이상의 응용 프로그램이 만들어진다. 물론 원한다면 대상이 되는 닷넷 버전을 바꿀 수 있다. 테스트를 위해 비주얼 스튜디오에서 콘솔 앱을 하나 만들자. 프로젝트 속성 창에서 "애플리케이션(Application)" 탭을 누르고 "대상 프레임워크(Target framework)" 영역을 펼치면 현재 윈도우에 설치된 닷넷 프레임워크 목록이 나타나고 이 중에서 선택할 수 있다.

그림 1.5 대상 프레임워크 변경



참고!	그림 1.5의 대상 프레임워크 목록을 보면 닷넷 버전 외에 "Client Profile"이 붙은 것을 볼 수 있다. ".NET Framework 4"의 경우 설치 파일 크기는 48MB인 반면, ".NET Framework 4 Client Profile"은 41MB로 다소 작다. 마이크로소프트에서는 조금이라도 설치 파일의 크기를 줄이기 위해 데스크톱용 응용 프로그램을 만들 수 있는 BCL만을 담은 "Client Profile" 버전을 내놓은 것이다. 간단하게 정리하면 ".NET Framework 4 = .NET Framework 4 Client Profile + 서버 응용 프로그램을 위한 BCL" 관계다.

예를 들어, ".NET Framework 3.5"로 변경하고 app.config 파일의 내용을 보면 다음과 같이 supportedRuntime 설정이 추가돼 있다.

```
<?xml version="1.0"?>
<configuration>
    <startup>
        <supportedRuntime version="v2.0.50727"/>
    </startup>
</configuration>
```

다음은 프로젝트 속성 창의 대상 프레임워크에 따른 supportedRuntime 값 설정을 보여준다.

표 1.3 supportedRuntime 값

대상 닷넷 프레임워크 버전	supportedRuntime

	version	sku
.NET Framework 2.0	v2.0.50727	
.NET Framework 3.0		
.NET Framework 3.5		
.NET Framework 3.5 Client Profile		Client
.NET Framework 4	v4.0	.NETFramework,Version=v4.0
.NET Framework 4 Client Profile		.NETFramework,Version=v4.0,Profile=Client
.NET Framework 4.5		.NETFramework,Version=v4.5
.NET Framework 4.5.1		.NETFramework,Version=v4.5.1
.NET Framework 4.5.2		.NETFramework,Version=v4.5.2
.NET Framework 4.6		.NETFramework,Version=v4.6
.NET Framework 4.6.1		.NETFramework,Version=v4.6.1
.NET Framework 4.6.2		.NETFramework,Version=v4.6.2
.NET Framework 4.7		.NETFramework,Version=v4.7
.NET Framework 4.7.1		.NETFramework,Version=v4.7.1
.NET Framework 4.7.2		.NETFramework,Version=v4.7.2
.NET Framework 4.8		.NETFramework,Version=v4.8

일반적으로 supportedRuntime을 이렇게 명시적으로 지정하면 반드시 해당 프레임워크 버전이 설치돼 있어야 응용 프로그램이 동작한다. "4 Client Profile" 버전의 경우 4.0에 포함돼 있기 때문에 닷넷 4.0 또는 4.5 이상만 설치돼 있다면 문제없이 실행된다.

그런데 모든 닷넷 프레임워크 환경에서 실행되는 응용 프로그램을 만들 수는 없을까? 예를 들어 그동안 실습했던 프로그램은 모두 닷넷 2.0에서도 실행될 수 있는 기능만 사용하고 있다. 따라서 2.0의 기능을 포함하는 3.0, 3.5, 4.0, 4.5 이상에서도 정상적으로 실행될 법도 하다. 물론 방법은 있다.[3]

우선 응용 프로그램의 프로젝트 설정에서 닷넷 프레임워크 대상을 2.0으로 맞춘다. 닷넷 프레임워크는 하위 호환성을 보장하므로 상위 버전에서 하위 버전의 어셈블리를 해석할 수 있기 때문에 2.0으로 맞추면 2.0 ~ 4.8에서 실행할 수 있다. 그러고 나서 해당 응용 프로그램이 CLR 2.0과 4.0에서 실행할 수 있다는 점을 supportedRuntime을 이용해 명시한다.

표 1.4 다중 supportedRuntime

```
<?xml version="1.0"?>
<configuration>
```

[3] C++ 개발자라면 CLR 버전을 직접 선택할 수 있다(http://www.sysnet.pe.kr/2/0/1746).

```
    <startup>
        <supportedRuntime version="v4.0"/>
        <supportedRuntime version="v2.0.50727"/>
    </startup>
</configuration>
```

이 방법이 정말 통하는지 확인하는 방법이 있다. 닷넷 BCL에는 현재 응용 프로그램이 어떤 버전의 닷넷에서 실행되고 있는지 Environment 타입의 Version 속성을 통해 알 수 있다.

```
static void Main(string[] args)
{
    Console.WriteLine(Environment.Version);
}
```

위의 프로그램을 닷넷 4.0 또는 4.5 이상이 설치된 컴퓨터에서 실행해 보자. 그러면 런타임 출력값이 "4.0.30319.42000"으로 출력되고, 닷넷 2.0, 3.0, 3.5 중 하나가 설치된 컴퓨터에서 실행하면 "2.0.50727.8784"로 출력되는 것을 확인할 수 있다. 그런데 닷넷 2.0과 4.0이 모두 설치된 컴퓨터에서는 어떤 버전으로 실행될까? 이럴 때는 app.config에 지정된 supportedRuntime 순서가 중요하다. 위에 명시한 버전이 먼저 평가되므로 표 1.4대로라면 "4.0.30319.42000"이 출력된다.

참고!	4.0.30319.42000 버전의 마지막 자리의 숫자 42000은 윈도우 업데이트 등의 이유로 바뀔 수 있다. 따라서 여러분의 컴퓨터에서는 42000이 아닌 다른 숫자가 출력될 수 있다.

정리하면, 처음에 닷넷 응용 프로그램을 실행하면 CLR을 로드하고 초기화하는 코드가 가장 먼저 실행된다. 초기화 코드에서는 [……].exe.config 파일이 있다면 이 파일을 읽어 들여 supportedRuntime 설정이 있는지 확인한다. supportedRuntime이 설정돼 있다면 현재 시스템에 설치돼 있는 CLR 버전과 supportedRuntime이 요구하는 CLR 버전에 일치하는 것을 로드한다. 그런 후에야 비로소 닷넷 어셈블리에 포함된 중간 언어가 특정 버전의 CLR에 의해 실행된다.

일반적으로 데스크톱 사용자를 목표로 응용 프로그램을 만든다면 하위 버전 사용자의 불편을 줄이기 위해 다중 supportedRuntime을 지정하는 것이 좋다. 반면 제한된 환경의 서버 측 응용 프로그램을 만든다면 닷넷 프레임워크를 설치하는 데 문제가 되지 않으므로 최신 버전의 닷넷을 대상으로 응용 프로그램을 만드는 것을 권장한다.

5.2.4. 디버그 빌드와 릴리스 빌드 (닷넷 프레임워크)

프로그램을 만들다 보면 크게 두 가지 오류를 접할 수 있다.

- **컴파일 시 오류(Compile-time error)**: 문법 오류(syntax error)이며, 컴파일러의 오류 메시지 내용에 따라 올바른 문법으로 변경하면 해결된다.
- **실행 시 오류(Run-time error)**: 정상적으로 컴파일된 프로그램이지만 실행되는 시점에 오류가 발생하는 것으로 논리 오류(logical error) 등의 원인으로 발생한다.

프로그램을 컴파일하는 시점에 발생하는 오류의 경우 근래에는 통합 개발 환경(예: 비주얼 스튜디오)의 도움으로 컴파일하기 전에 이미 편집 창에서 오류를 파악할 수 있다. 그뿐만 아니라 설령 컴파일 오류가 발생하더라도 문법상으로 오류가 발생한 위치를 쉽게 알 수 있어 원인을 수정하기가 굉장히 쉽다. 문제는 실행 시 오류다. 실행 시에 발생하는 오류는 대개 예상치 못한 상황이며, 자칫 프로그램의 실행에 심각한 영향을 끼칠 수 있다. 예를 들어 다음과 같은 코드를 보자.

예제 1.2 실행 시 오류가 발생하는 유형

```
using System;

class Program
{
    static void Main(string[] args)
    {
        int [] nArray = new int [] { 0, 1, 2, 3, 4 };
        nArray[5] = 0; // 예외 발생
    }
}
```

예제 1.2의 코드는 5개의 요소가 담긴 배열의 6번째 요소에 접근하려고 했기 때문에 System.IndexOutOfRangeException 오류가 발생한다.

이런 오류를 버그(bug)라고 부르며, 해당 버그를 수정하는 작업을 디버깅(debugging)이라 한다. 디버깅의 핵심은 버그의 원인을 파악하는 것인데, 이때 원인 파악을 도와주는 용도의 전문적인 프로그램을 디버거(debugger)라고 한다. 일반적으로 비주얼 스튜디오와 이클립스 등의 통합 개발 환경에는 디버거 기능까지 포함돼 있다. 예제 1.2처럼 소스코드의 양이 적은 경우에는 이러한 오류를 찾기가 매우 쉽지만 수천~수만 줄의 코드에서 디버거의 도움 없이 정확한 오류의 원인을 찾기란 쉽지 않다.

그런데 여기서 한 가지 알아둬야 할 점은 여러분이 작성한 코드가 그대로 기계어로 생성되지는

않는다는 점이다. 왜냐하면 컴파일러 제작자들은 개발자가 작성한 코드를 가능한 한 가장 빠른 속도 또는 가장 작은 용량의 프로그램으로 번역하는 최적화(optimization) 처리를 하기 때문이다. 최적화를 직접 확인하기 위해 예제 1.2을 다음과 같이 변경해 보자.

예제 1.3 최적화 예제

```
 1: using System;
 2:
 3: class Program
 4: {
 5:     static void AccessArray(int [] array)
 6:     {
 7:         array[5] = 0;
 8:     }
 9:
10:     static void Main(string[] args)
11:     {
12:         int [] nArray = new int [] { 0, 1, 2, 3, 4 };
13:         AccessArray(nArray);
14:     }
15: }
```

이 프로그램을 지금까지 실습해 온 것처럼 명령 프롬프트에서 다음과 같이 빌드하고 실행해 본다.

```
D:\temp\ConsoleApp1>csc program.cs
Microsoft (R) Visual C# Compiler version 2.2.0.61624
Copyright (C) Microsoft Corporation. All rights reserved.

D:\temp\ConsoleApp1>program.exe
Unhandled Exception: System.IndexOutOfRangeException: Index was outside the bounds of the
array.
   at Program.Main(String[] args)
```

소스코드상으로 보면 분명히 AccessArray 메서드에서 오류가 발생했음에도 실제 응용 프로그램을 실행했을 때는 Main 메서드에서 발생했다고 메시지가 출력되고 있다. C# 컴파일러에 의해 생성된 중간 언어를 실행하는 것은 CLR이다. 즉, CLR은 중간 언어를 기계어로 번역할 때 AccessArray 메서드의 중간 언어가 단지 array[5] = 0에 불과하므로 아예 이 코드를 Main 메서드에 포함시켜 최종 기계어를 생성하고 실행한다. 결국 최적화된 프로그램은 다음과 같이 작성한 것과 동일하게 수행되는 것이다.

```
static void Main(string[] args)
{
    int [] nArray = new int [] { 0, 1, 2, 3, 4 };
    array[5] = 0;
}
```

이렇게 호출하는 측(Caller)의 코드에 대상 메서드(Callee)의 코드가 포함되는 것을 인라인(inline)됐다고 표현한다. 메서드의 코드를 인라인시키는 이유는 속도를 빠르게 하기 위해서다. 왜냐하면 메서드로 호출하려면 매개변수를 전달하는 등의 작업을 해야 하므로 상대적으로 속도 저하가 발생하기 때문이다.

최적화에는 인라인 외에도 여러 가지 세세한 작업들이 포함된다. 이런 식으로 최적화를 허용하는 빌드를 릴리스(release) 빌드라 한다. 앞에서 설명한 바와 같이 코드 최적화는 문제의 원인 파악을 매우 어렵게 만든다. 위와 같이 오류 메시지가 발생하면 개발자는 그 원인을 찾기 위해 Main 메서드를 집중적으로 살펴볼 뿐 AccessArray 메서드의 검증은 등한시할 수 있다. 오류 파악 관점에서 볼 때 위의 릴리스 빌드에는 두 가지 주요 문제점이 있다. 하나는 문제가 발생한 소스코드의 라인 정보가 없다는 것이고, 다른 하나는 최적화를 통해 메서드가 인라인됐다는 점이다.

우선 문제가 발생한 소스코드 라인을 파악하기 위해서는 반드시 PDB(program database) 파일이 있어야 한다. 이 파일은 C# 컴파일러의 /debug 옵션으로 생성할 수 있다.

```
D:\temp\ConsoleApp1>csc program.cs /debug:pdbonly
Microsoft (R) Visual C# Compiler version 2.2.0.61624
Copyright (C) Microsoft Corporation. All rights reserved.

D:\temp\ConsoleApp1>program.exe
Unhandled Exception: System.IndexOutOfRangeException: Index was outside the bounds of the array.
   at Program.Main(String[] args) in D:\temp\ConsoleApp1\Program.cs:line 12
```

이렇게 빌드하면 program.exe 파일과 program.pdb 파일이 함께 생성되고, 오류가 발생했을 때 CLR은 program.pdb 파일을 함께 로드해서 그 파일로부터 문제가 발생한 영역의 소스코드 라인 정보를 찾아 함께 출력해 준다. 라인 정보가 출력되더라도 예제 1.3의 라인과 비교하면 역시 100% 일치하지는 않는다. 이 부분은 최적화로 인한 어쩔 수 없는 문제다. 이러한 불일치를 완전히 없애려면 빌드 자체를 다음과 같이 해야 한다.

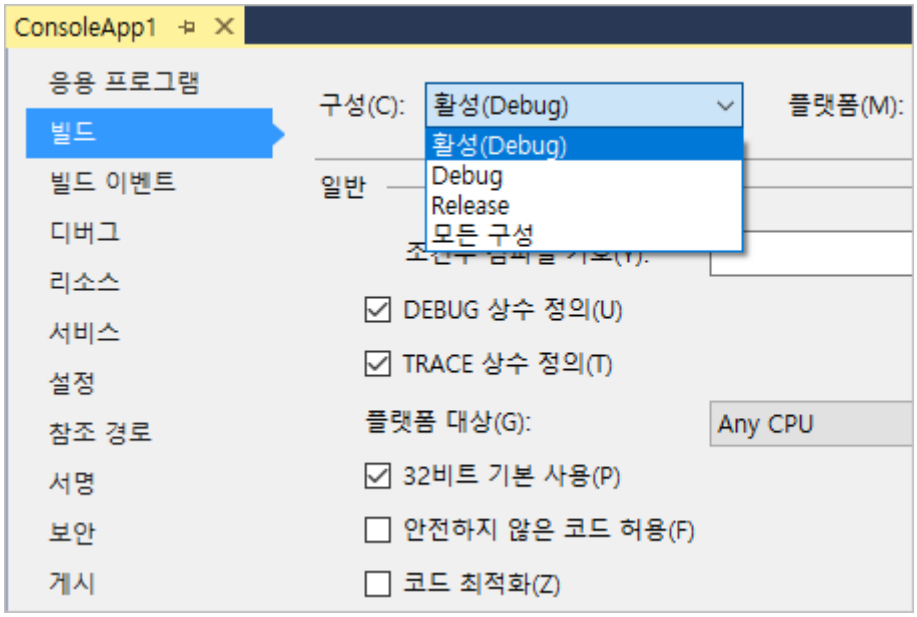
```
D:\temp\ConsoleApp1>csc program.cs /debug
Microsoft (R) Visual C# Compiler version 2.2.0.61624
Copyright (C) Microsoft Corporation. All rights reserved.

D:\temp\ConsoleApp1>program.exe

Unhandled Exception: System.IndexOutOfRangeException: Index was outside the bounds of the array.
 at Program.AccessArray(Int32[] array) in D:\temp\ConsoleApp1\Program.cs:line 7
```

```
at Program.Main(String[] args) in D:\temp\ConsoleApp1\Program.cs:line 13
```

여기서 /debug 옵션은 기본값으로 "/debug:full"과 동일하게 동작한다. 이 옵션을 지정해서 빌드된 EXE/DLL은 실행할 때 CLR에 의해 최적화 과정이 생략되고 개발자가 생성한 코드를 그대로 기계어로 번역한다. 따라서 오류가 발생한 위치를 정확하게 파악하고 문제의 원인을 좀 더 빠르게 찾을 수 있다. 이렇게 최적화를 하지 않는 빌드를 디버그 빌드라고 한다.

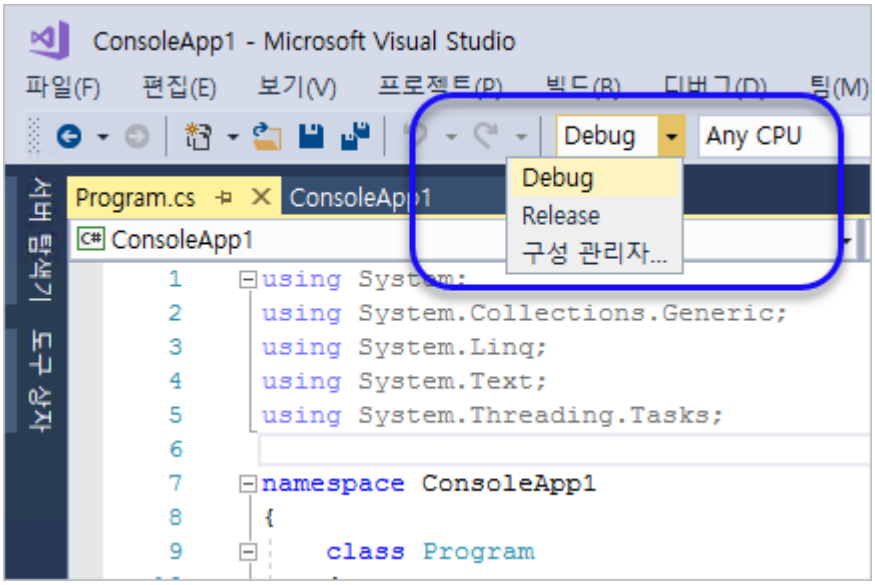
일반적으로 프로그램을 개발할 때는 디버그 모드로 빌드(/debug)하고 개발된 프로그램을 사용자에게 배포할 때는 릴리스 모드로 빌드한다. 하지만 릴리스 빌드의 경우 소스코드 라인 정보를 얻기 위해 /debug:pdbonly 옵션을 함께 적용한다. 이 원칙은 비주얼 스튜디오에서도 그대로 따르고 있다. 비주얼 스튜디오에서는 새 프로젝트를 만들면 Debug와 Release에 대한 옵션을 따로 갖고 이 값은 프로젝트 속성 창을 통해 확인할 수 있다.

그림 1.6 프로젝트의 구성(Configuration) 설정: Debug, Release



비주얼 스튜디오에서 구성(Configuration) 옵션은 Debug와 Release로 나뉘고 각각 목적에 맞게 미리 옵션 설정이 돼 있기 때문에 특별히 조정할 사항은 없다. 프로젝트에는 이처럼 두 가지 빌드 옵션이 미리 정의돼 있는데, 어떤 빌드 옵션으로 EXE/DLL 파일을 생성할지는 개발자가 선택해야 한다. 이 경우 비주얼 스튜디오 툴바의 "시작(Start) 버튼" 옆에 있는 콤보 박스를 이용해 설정한다.

그림 1.7 비주얼 스튜디오에서 Configuration 설정



이 값을 Debug로 놓고 빌드하면 현재 솔루션에 포함된 모든 프로젝트를 프로젝트 각각에 포함된 Debug 옵션으로 빌드하고, Release로 놓으면 전체 프로젝트를 최적화 빌드로 진행한다. 원칙은 간단하다. 프로그램을 개발할 때는 Debug로 놓고, 최종 사용자에게 배포할 제품으로 만들 때는 Release 빌드를 하면 된다.

5.2.5 플랫폼(x86, x64, AnyCPU) 선택 (닷넷 프레임워크)

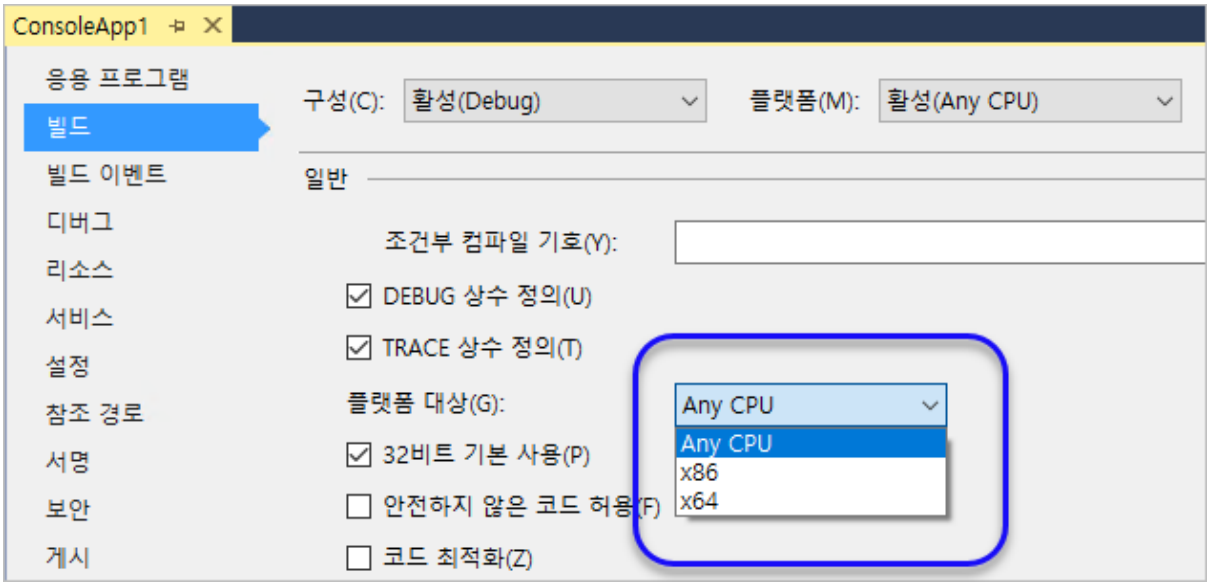
프로세스가 32비트로 실행되는지 64비트로 실행되는지 파악하려면 Environment 타입의 Is64BitProcess 속성을 이용하면 된다.

```
Console.WriteLine("64 bit process: " + Environment.Is64BitProcess);
```

// 실행된 EXE가 32비트 프로세스이면 False 반환, 64비트 프로세스이면 True 반환

32비트와 64비트 대신 비주얼 스튜디오에서는 그에 대응되는 단어로 각각 x86, x64를 사용하며 플랫폼에 상관없는 AnyCPU를 추가로 두고 있다. 프로젝트 속성 창에서 "빌드(Build)" 탭의 "플랫폼 대상(Platform target)"에서 EXE/DLL이 실행될 대상 플랫폼을 선택할 수 있다.

그림 1.8 프로젝트의 플랫폼 지정



이 옵션을 x86으로 설정하고 프로젝트를 빌드하면 32비트/64비트 윈도우에서 32비트 프로세스로 실행된다. 반면 x64로 설정하고 빌드하면 32비트 윈도우에서는 실행이 불가능하고 오직 64비트 윈도우에서만 64비트 프로세스로 실행된다. AnyCPU의 경우 32비트 윈도우에서는 32비트로, 64비트 윈도우에서는 64비트 프로세스로 실행된다. 비주얼 스튜디오에서 프로젝트를 생성하면 기본값은 AnyCPU다.

표 1.5 플랫폼 설정에 따른 프로세스 유형

플랫폼	윈도우 운영체제	
	32비트 윈도우	64비트 윈도우
x86	32비트 EXE로 실행	32비트 EXE로 실행
x64	실행 불가	64비트 EXE로 실행
AnyCPU	32비트 EXE로 실행	64비트 EXE로 실행

그런데 AnyCPU로 지정하고 빌드해도 여전히 64비트 윈도우에서 32비트로 실행되는 것을 볼 수 있다. 그 이유는 그림 1.8의 하단에 있는 "32비트 기본 사용(Prefer 32-bit)" 옵션이 설정돼 있기 때문이다. 이 옵션은 64비트에서도 32비트로 실행될 수 있게 한다. 그렇다면 애당초 x86으로 지정하면 되는데 왜 굳이 AnyCPU에 "32비트 기본 사용"옵션을 별도로 두는 것일까? 그 이유는 윈도우 8 운영체제부터 ARM 32비트 CPU에 대한 지원이 추가됐기 때문이다. 따라서 단순히 x86으로만 설정하면 인텔/AMD CPU 기반의 32비트 윈도우 운영체제에서는 프로그램이 32비트로 실행되지만 ARM CPU에서 동작하는 32비트 운영체제에서는 실행되지 않는다. 참고로 "32비트 기본 사용"은 AnyCPU 상태에서만 선택할 수 있다.

표 1.6 "32비트 기본 사용" 옵션에 따른 프로세스 유형

플랫폼	윈도우 운영체제		
	Intel/AMD 32비트	ARM 32비트	Intel/AMD 64비트
x86	32비트 EXE로 실행	실행 불가	32비트 EXE로 실행
AnyCPU + 32비트 기본 사용	32비트 EXE로 실행	32비트 EXE로 실행	32비트 EXE로 실행

여기서 한가지 더 의문이 들 수 있다. 가장 이상적인 AnyCPU만으로 구성하는 것이 좋을 텐데 왜 혼란스럽게 32비트를 고려해야 하는 걸까? 그 이유는 아직도 많은 수의 공개된 DLL이 32비트로만 만들어져 있어 그 DLL을 사용하려면 32비트 프로세스로 실행돼야 하기 때문이다.

참고!	32비트용 DLL은 64비트 프로세스(EXE)에 사용될 수 없다. 예전에 웹 브라우저의 확장 기능을 ActiveX로 구현했을 때 거의 대부분의 ActiveX가 32비트용으로 빌드된 DLL이었다. 그래서 64비트 운영체제에서도 여전히 웹 브라우저만큼은 32비트를 써야 하는 한계가 발생했다.

지금도 여전히 32비트용 DLL이 많기 때문에 AnyCPU의 장점은 살리면서 ARM CPU를 고려한 "32비트 기본 사용" 옵션을 비주얼 스튜디오 2015부터 기본 빌드 옵션으로 선택한 것이다.

정리하면, 32비트 네이티브 DLL(예: ActiveX)을 사용하는 경우가 있다면 AnyCPU + "32비트 기본 사용" 옵션을 사용하고, 그렇지 않은 대부분의 경우에는 AnyCPU로 만드는 것을 권장한다.

5.2.6 버전 관리 (닷넷 프레임워크)

DLL 파일은 서로 다른 프로세스(EXE)에서 동시에 사용할 수 있다. 예를 들어, A.DLL이 있다면 B.EXE와 C.EXE에서 해당 DLL을 파일 참조를 통해 함께 사용할 수 있다. 이처럼 서로 다른 프로그램에서 같은 DLL을 공유할 때 고려해야 할 것이 바로 버전(version)이다.

C/C++로 윈도우 프로그램을 만드는 경우 공유되는 DLL을 c:\windows\system32 폴더에 넣어두곤 했다. C/C++ 윈도우 프로그램은 DLL을 EXE와 같은 폴더에서 발견하지 못하면 system32 폴더도 함께 검색했기 때문에 여러 프로그램에서 공유하는 DLL을 넣어 두는 장소로 사용됐다.

참고!	윈도우에서 네이티브 DLL 파일이 검색되는 경로는 그렇게 간단하지 않다. 자세한 검색 순서는 다음 글을 참고한다. Dynamic-Link Library Search Order (Windows) https://learn.microsoft.com/en-us/windows/win32/dlls/dynamic-link-library-search-order

하지만 system32 폴더를 통해 공유하는 방법은 이후 "DLL 지옥(DLL Hell)"이라는 심각한 문제를 낳았다. 이 현상을 설명하면 다음과 같이 정리할 수 있다.

1. A.DLL을 사용하는 D 제품을 설치한다. A.DLL은 C:\Windows\system32 폴더에 설치되고, D 제품은 C:\Program Files\DProduct 폴더에 설치된다. 현재 상태에서 D 제품은 정상적으로 실행된다.
2. 시간이 흘러 A.DLL은 버전 2로 업데이트되어 구조가 바뀐다.
3. 버전 2의 새롭게 바뀐 A.DLL을 사용한 E 제품이 개발된다.
4. D 제품을 설치했던 사용자가 E 제품을 함께 설치한다. 이 과정에서 system32 폴더에 있던 기존 A.DLL 파일이 E 제품에 포함된 신규 A.DLL 파일로 바뀐다. E 제품은 정상적으로 실행된다.
5. 기존의 D 제품을 실행하면 구조가 바뀐 버전 2의 A.DLL을 사용하려고 시도한다. 하지만 구조가 바뀌었으므로 예측할 수 없는 상태로 빠진다. 대개 응용 프로그램이 비정상적으로 종료된다.

닷넷은 이 문제를 "전역 어셈블리 캐시(global assembly cache)"와 "강력한 이름의 어셈블리(strong-named assembly)"라는 것을 도입해서 해결한다.

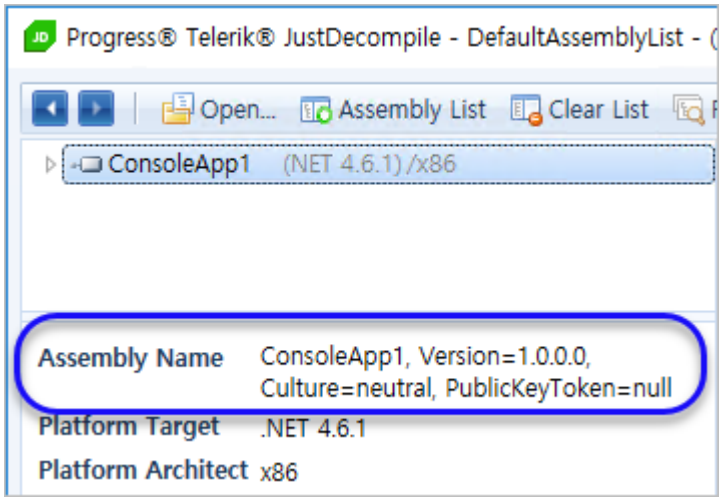
5.2.6.1 어셈블리의 버전과 이름 (닷넷 프레임워크)

비주얼 스튜디오에서 생성한 프로젝트의 경우 자동으로 포함되는 AssemblyInfo.cs 파일에 버전 정보를 위한 AssemblyVersion 특성이 제공된다.

```
[assembly: AssemblyVersion("1.0.0.0")]
```

기본값은 1.0.0.0이지만 개발자가 임의로 바꿀 수 있고, 각 숫자는 0 ~ 65534 범위에서 지정할 수 있다. 이 특성이 적용되어 빌드된 어셈블리(DLL/EXE)로부터 버전 정보를 확인하려면 JustDecompile 같은 역컴파일러를 이용해야 한다(윈도우 탐색기에서는 확인할 수 없다). 그림 1.9은 ConsoleApp1.exe 파일을 JustDecompile 도구로 열어 버전 정보를 확인하는 모습이다.

그림 1.9 JustDecompile로 확인한 어셈블리 버전 정보



그런데 여기서 눈여겨봐야 할 것은 Version 정보가 포함된 문자열의 제목이 "어셈블리 이름(Assembly Name)"으로 표기돼 있다는 점이다. 즉, ConsoleApp1.exe는 파일 이름일 뿐 어셈블리의 진짜 이름은 "ConsoleApp1, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"이 되는 것이다. AssemblyVersion 특성으로 지정된 버전 정보는 어셈블리 이름의 한 구성 요소가 된다.

5.2.6.2 공개키 토큰과 어셈블리 서명 (닷넷 프레임워크)

어셈블리 이름은 특정 어셈블리를 식별하기 위한 고윳값이다. "DLL 지옥" 현상은 파일명 하나만 가지고 DLL을 식별하기 때문에 나타나는 현상이고, 이를 해결하려면 파일명에 몇 가지 고윳값을 더 추가해야만 한다. 앞서 마이크로소프트에서는 고윳값으로 버전 번호를 함께 추가했다. 하지만 여전히 어셈블리 이름이 중복될 가능성이 있다. 예를 들어, A라는 업체가 shared.dll, 1.0.0.0이라는 이름의 어셈블리를 가질 수 있고, 우연히도 B라는 업체도 동일한 이름과 버전의 어셈블리를 가질 확률이 있다. 이를 해결하기 위해 마이크로소프트에서는 어셈블리 저작자의 공개키 토큰값(public key token)을 어셈블리 이름에 포함하기로 결정한다.

이름에서 짐작할 수 있듯이 공개키 기반 구조(PKI: Public Key Infrastructure)가 여기에 사용된다. 마이크로소프트에서는 어셈블리 저작자가 개인키를 쉽게 생성할 수 있도록 비주얼 스튜디오나 윈도우 SDK에 sn.exe라는 도구를 포함시켜 배포한다. 비주얼 스튜디오가 설치돼 있다면 시작 메뉴를 통해 "Developer Command Prompt for VS 2022"를 실행하고 다음과 같은 명령을 내리면 개인키가 담긴 my.snk라는 파일을 생성할 수 있다.

```
D:\temp>sn -k my.snk

Microsoft (R) .NET Framework Strong Name Utility  Version 4.0.30319.0
Copyright (c) Microsoft Corporation.  All rights reserved.

키 쌍을 my.snk에 썼습니다.
```

이렇게 생성된 my.snk 파일을 사용하면 공개키 토큰 값을 어셈블리 파일명에 부여할 수 있다. 이 경우 C# 컴파일러에 my.snk 파일을 인자로 전달하면 된다.

```
// 파일명: Program.cs

using System;
using System.Reflection;

[assembly: AssemblyVersion("1.0.0.0")]

class Program
{
    static void Main(string[] args)
    {
```

```
    }
}
```

```
D:\temp\ConsoleApp1>csc /keyfile:d:₩temp₩my.snk program.cs
```

최종 생성된 Program.exe 파일을 JustDecompile로 보면 어셈블리 이름이 다음과 같이 표시된다.

Program, Version=1.0.0.0, Culture=neutral, **PublicKeyToken=0365f04bf6feccfd**

** PublicKeyToken 값은 필자가 생성한 my.snk에 담긴 공개키 토큰 값이므로 여러분이 테스트한 경우와는 값이 다르게 보일 것이다.*

이로써 좀 더 높은 유일성을 보장받는 어셈블리 이름을 만들 수 있다. 어셈블리 저작자만이 가지고 있는 키 파일의 고유한 8바이트 공개키 토큰 값은 쉽게 겹치지 않을 것이기 때문이다. 이러한 이유로 공개키 토큰이 부여된 어셈블리를 "강력한 이름의 어셈블리(strong-named assembly)"라고 하며, 이러한 고유성을 지키기 위해 키 파일(예: my.snk)은 외부에 유출되지 않도록 안전한 곳에 보관해야 한다.

C# 컴파일러로 /keyfile 옵션을 지정하는 경우 어셈블리 이름에만 영향을 주는 것은 아니다. 부가적으로 *C#* 컴파일러는 다음과 같이 어셈블리를 서명(sign)하는 작업까지 함께 수행한다.

1. *C#* 컴파일러는 어셈블리 파일의 내용에 대해 해시(hash) 함수를 통해 해시값을 구한다.
2. 주어진 키 파일의 개인키를 이용해 1번 단계에서 구한 해시값을 서명(암호화)한다.
3. 빌드된 어셈블리에 서명된 데이터와 키 파일의 공개키를 함께 보관한다.

이 과정을 가리켜 "어셈블리를 서명한다"라고 표현한다. 서명된 어셈블리를 확인하는 작업은 실행 시에 CLR에 의해 다음과 같이 이뤄진다.

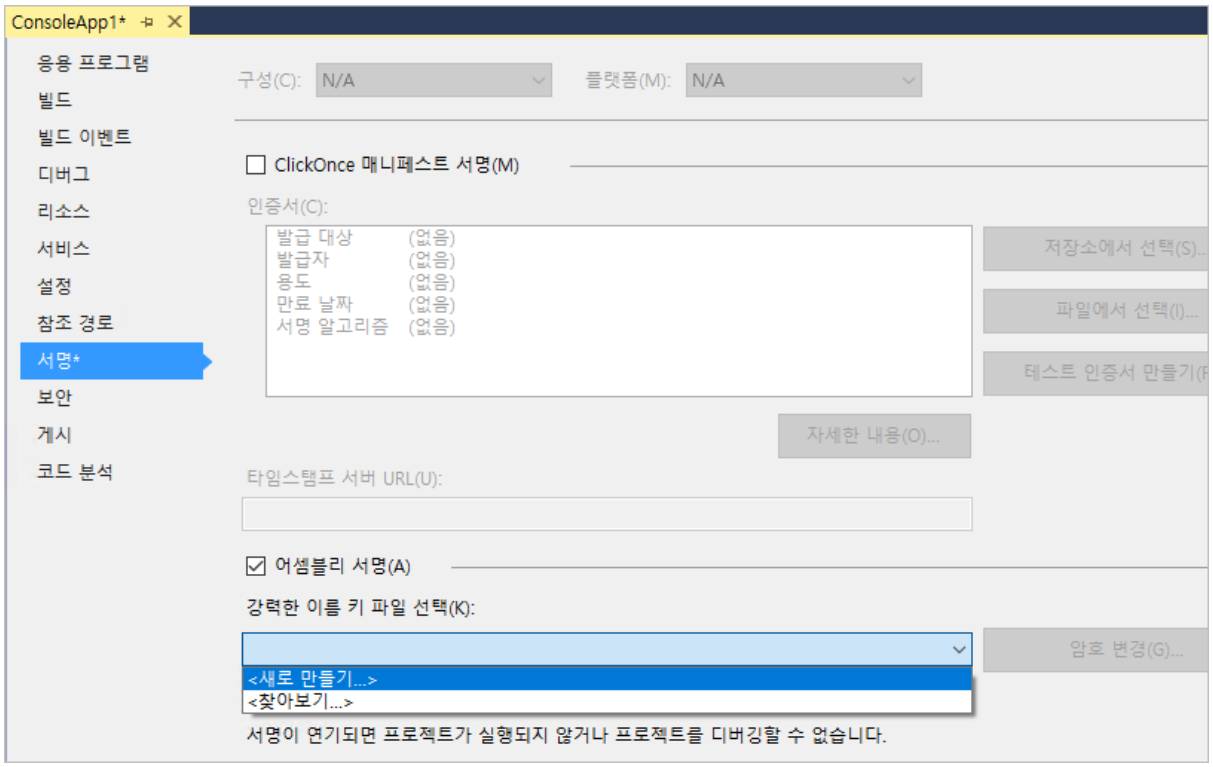
1. CLR은 어셈블리 파일 안에 서명된 데이터가 포함된 경우 어셈블리 파일의 내용을 다시 한번 해시 함수를 통해 해시값을 구한다.
2. 어셈블리 내에 함께 저장돼 있던 서명된 데이터와 공개키를 추출한다.
3. 서명된 데이터를 공개키로 복호화한다.
4. 복호화된 데이터와 1번 단계에서 구한 해시값을 비교한다. 이 값이 일치하지 않으면 어셈블리에 포함된 중간 언어가 빌드된 시점의 것과 다르다는 것을 의미한다. 즉, 어셈블리 파일의 내용이 위변조됐다고 판단하고 실행을 중단한다.

정리하면, "강력한 이름의 어셈블리"는 해당 어셈블리를 고유하게 식별할 수 있는 이름을 갖게 됐을 뿐만 아니라 어셈블리의 내용이 누군가에 의해 위변조될 수 없음을 보증한다.

비주얼 스튜디오에서 프로젝트를 생성했다면 어셈블리 서명을 좀 더 쉽게 할 수 있다. 프로젝트

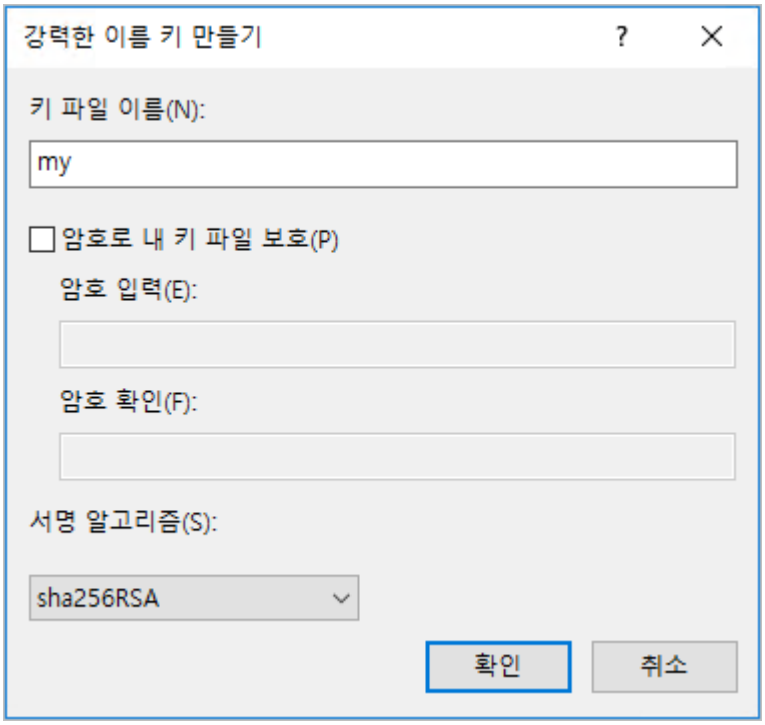
의 속성 창에서 "서명(Signing)" 탭을 선택하고, "어셈블리 서명(Sign the assembly)" 체크상자를 선택한 후 "강력한 이름 키 파일 선택(Choose a strong name key file)" 콤보 상자에서 "새로 만들기(New)" 항목을 선택한다. (이미 다른 프로젝트에서 키 파일을 생성했으면 "찾아보기(Browse)" 항목을 선택해 기존의 키 파일을 선택하면 된다.)

그림 1.10 프로젝트 속성 창의 서명 옵션



그러면 다음과 같은 키 파일 생성을 위한 대화상자가 나타난다. 키 파일 이름을 적절히 입력하고 원한다면 "암호로 내 키 파일 보호(Protect my key file with a password)" 항목을 선택해 암호를 지정할 수 있다.

그림 1.11 새로운 키 파일 지정



확인(OK) 버튼을 누르면 프로젝트에 my.snk 파일이 포함된다. 이제부터는 빌드하면 서명된 어셈블리(강력한 이름의 어셈블리)가 자동으로 생성된다.

5.2.6.3 전용 어셈블리, 전역 어셈블리 (닷넷 프레임워크)

라이브러리(DLL) 파일이 서명됐다고 해서 사용법이 바뀌는 것은 아니다. 여전히 DLL 파일을 EXE와 같은 폴더에 놓으면 EXE를 실행하는 데 지장은 없다. 이렇게 배포되는 DLL을 "전용 어셈블리(private assembly)"라고 한다.

참고!	전용 어셈블리를 개인(private) 어셈블리라고 표현하기도 한다.

비주얼 스튜디오에서 프로젝트 참조나 파일 참조를 하는 경우, 대상 DLL은 EXE가 빌드된 폴더에 함께 놓인다. 따라서 전용 어셈블리만 포함된 프로그램이라면 다른 컴퓨터에 배포할 때 단순히 가지고 있는 파일을 모두 복사하기만 하면 된다. 이 때문에 이런 식의 배포 방식을 "XCopy 배포"라고 한다.

참고!	XCopy는 윈도우의 셸 명령어 중 하나다. 이 명령어를 이용하면 폴더 단위의 복사를 할 수 있고, 폴더의 하위 폴더까지 모두 복사하는 것이 가능하다. 따라서 "XCopy 배포"란 단순히 응용 프로그램과 관련된 파일 및 폴더만을 복사하는 것으로 다른 컴퓨터에서 실행할 수 있음을 의미한다.

기본적으로 전용 어셈블리는 프로그램 간의 공유를 가정하지 않는다. 따라서 버전 관리도 스스로 하면 되므로 "DLL 지옥" 같은 현상은 나타나지 않는다.

그렇다면 닷넷에서는 어떻게 DLL을 공유할까? 네이티브 DLL은 C:\Windows\system32 폴더를 통해 공유됐지만 이 때문에 "DLL 지옥"이라는 단점이 나타났다. 1.0 버전의 DLL과 1.1 버전의 DLL을 구분하는 기준은 결국 파일명에 불과했기 때문에 이런 현상이 나타난 것이다. 닷넷에서는 이를 보완하기 위해 "파일명 + 버전 + 컬처(culture) + 공개키 토큰"을 조합한 강력한 이름의 어셈블리를 만들 수 있게 했고, 이런 어셈블리만이 배포될 수 있는 컴퓨터의 저장소를 별도로 마련해 뒀다. 그것이 바로 "전역 어셈블리 캐시(GAC: Global Assembly Cache)"다. 닷넷 프레임워크가 설치되면 C:\Windows\assembly라는 폴더가 생성되고 오직 강력한 이름을 가진 어셈블리만 GAC 폴더에 배포될 수 있는 자격을 가진다.

참고!	GAC는 닷넷 프레임워크 버전에서만 사용하고 닷넷 코어/5+부터는 더 이상 GAC를 지원하지 않는다.

그림 1.12 닷넷 프레임워크 2.0 ~ 3.5를 위한 GAC 저장소

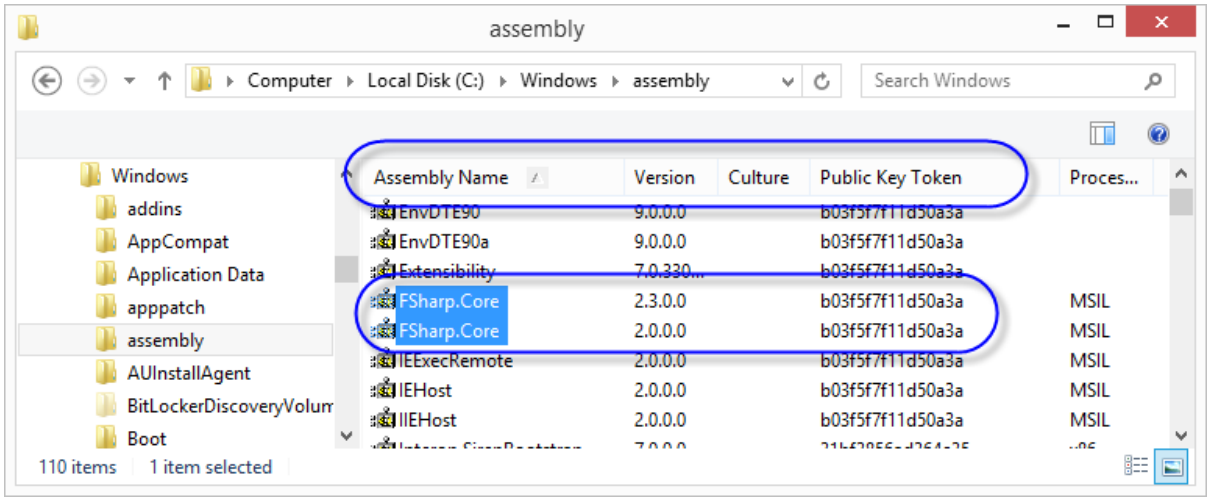


그림 1.12의 FSharp.Core 어셈블리를 보자. 파일명이 동일한 두 개의 어셈블리가 버전이 2.0.0.0, 2.3.0.0으로 함께 GAC에 등록된 것을 볼 수 있다. 이처럼 닷넷은 같은 파일명이라고 해도 버전이나 공개키 토큰이 다르다면 구분되어 파일을 보관할 수 있는 GAC를 도입함으로써 기존의 네이티브 응용 프로그램이 안고 있던 DLL 지옥 현상을 극복했다.

참고!	닷넷 프레임워크 4.0 이상에서는 GAC 저장소 폴더가 C:\Windows\Microsoft.NET\assembly로 분리된다.

여러분이 만든 서명된 DLL을 GAC에 등록하려면 gacutil.exe 도구를 사용하면 된다. 이 프로그램은 sn.exe와 같이 비주얼 스튜디오나 윈도우 SDK에 포함돼 있다. 따라서 시작 메뉴를 통해 "Developer Command Prompt for VS 2022"를 "관리자 권한"으로 실행하면 gacutil.exe를 사용할 수 있다. 다음은 강력한 이름을 가진 ClassLibrary1.dll 파일을 GAC에 등록/해제하는 예다(등록은 /i 옵션, 해제는 /u 옵션).

```
C:\temp>gacutil /i ClassLibrary1.dll
Microsoft (R) .NET Global Assembly Cache Utility.  Version 4.0.30319.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Assembly successfully added to the cache

C:\temp>gacutil /u ClassLibrary1
Microsoft (R) .NET Global Assembly Cache Utility.  Version 4.0.30319.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Assembly: ClassLibrary1, Version=1.0.0.0, Culture=neutral, PublicKeyToken=ef46b7f6beada04e,
processorArchitecture=MSIL
Uninstalled: ClassLibrary1, Version=1.0.0.0, Culture=neutral, PublicKeyToken=ef46b7f6beada04e,
processorArchitecture=MSIL
Number of assemblies uninstalled = 1
Number of failures = 0
```

한 가지 주의할 점은 등록할 때는 파일명과 확장자(.dll)를 함께 명시하지만 해제할 때는 파일명만 지정한다는 점이다.

GAC 등록을 다음과 같은 순으로 직접 테스트해 보자.

1. 비주얼 스튜디오에서 클래스 라이브러리 유형의 프로젝트를 만든다(프로젝트 이름을 ClassLibrary1이라고 가정한다).
2. 프로젝트 속성 창에 서명 옵션을 켜고 키 파일을 생성한 후 빌드한다.
3. gacutil.exe를 이용해 2번 단계에서 빌드한 DLL을 GAC에 등록한다.
4. AssemblyInfo.cs 파일을 열고 AssemblyVersion의 값을 1.1.0.0으로 증가시킨다.
5. 다시 빌드한다.
6. gacutil.exe를 이용해 5번 단계에서 빌드한 DLL을 GAC에 등록한다.

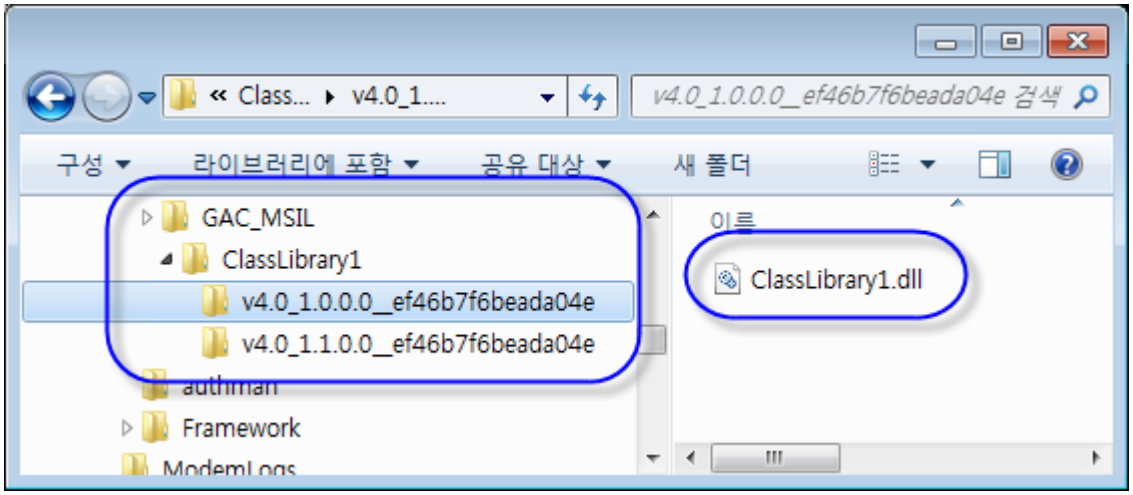
정상적으로 GAC에 등록됐는지 확인하려면 윈도우 탐색기를 실행하고 assembly 폴더를 보면 된다. 비주얼 스튜디오 2022의 경우 기본값이 닷넷 4.8용 DLL을 생성하므로 C:\Windows\Microsoft.NET\assembly 폴더를 확인해야 한다. 4.0 ~ 4.8용 GAC는 그림 1.12처럼 보이지 않고 다음과 같은 식의 하위 폴더가 선택적으로 보인다.

표 1.7 닷넷 프레임워크 4.0/4.5/4.6/4.7/4.8의 GAC 하위 폴더

GAC 하위 폴더	의미
GAC_32	x86용으로 빌드된 어셈블리가 보관되는 폴더
GAC_64	x64용으로 빌드된 어셈블리가 보관되는 폴더
GAC_MSIL	AnyCPU용으로 빌드된 어셈블리가 보관되는 폴더

우리가 실습한 어셈블리는 AnyCPU용으로 빌드됐을 것이므로 GAC_MSIL 폴더로 가면 1개의 ClassLibrary1 폴더가 보인다. 다시 그 아래의 폴더로 들어가면 AssemblyVersion 특성에 기록한 버전과 공개키 토큰이 합쳐져 두 개의 폴더로 나뉘고 각 폴더에 ClassLibrary1.dll 파일이 존재한다.

그림 1.13 GAC에 등록된 1.0.0.0, 1.1.0.0 어셈블리



참고!	닷넷 프레임워크 2.0 ~ 3.5의 GAC 폴더도 4.0 ~ 4.8과 같은 식으로 볼 수 있다. 명령 프롬프트를 열고 다음과 같이 SUBST 셸 명령어를 실행한다. SUBST L: C:\Windows\Assembly 그럼 윈도우 탐색기를 실행해 새롭게 생긴 L 드라이브를 확인할 수 있는데, 이는 GAC의 실제 물리 폴더 구조를 보여준다.

이것이 GAC의 실체다. 윈도우에서 특정 파일을 구분하는 것은 파일 시스템의 제약을 받는다. 따라서 닷넷 프레임워크에서 아무리 강력한 이름의 어셈블리로 여러 가지 식별자를 뒀다고 해도 같은 폴더에 버전이 다른 DLL을 함께 보관하는 것은 불가능하다. 따라서 닷넷은 GAC라는 특별한 폴더에 강력한 이름의 어셈블리를 파일명, 버전, 공개키 토큰 값을 이용해 폴더별로 구분해 보관한다. 이 때문에 C:\Windows\system32에 단순히 파일 이름으로만 복사되던 네이티브 응용 프로

그램과는 달리 DLL 지옥 현상이 닷넷에서는 발생하지 않는 것이다.

이처럼 GAC에 등록된 어셈블리를 전용 어셈블리와 비교해 전역 어셈블리(global assembly)라고 한다. 전역 어셈블리를 사용하는 방법은 기존의 전용 어셈블리를 사용하는 것과 완전히 동일하다. 단지, 전용 어셈블리의 경우 해당 DLL이 실행 파일(EXE)과 동일한 폴더에 있어야 했지만 전역 어셈블리는 GAC에 등록돼 있기 때문에 실행 파일과 동일한 폴더에 없어도 된다. 설령 같은 DLL이 실행 파일과 동일한 폴더에 들어 있어도 CLR은 GAC에 등록된 강력한 이름의 어셈블리를 우선적으로 로드해서 사용한다.

그런데 전역 어셈블리를 사용하는 응용 프로그램은 어떻게 배포해야 할까? 크게 두 가지 방법이 있을 수 있다.

1. 개발자의 컴퓨터에서는 GAC에 등록돼 있더라도 이를 무시하고 EXE와 같은 폴더에 모두 함께 복사해서 배포한다.
2. 전역 어셈블리를 원칙대로 GAC에 등록하고 프로그램은 다른 폴더에 나누어 배포한다.

문제는 후자의 경우 GAC에 등록하는 gacutil.exe 파일이 닷넷 프레임워크만 설치된 컴퓨터에는 존재하지 않는다는 것이다. 현재 사용자 컴퓨터의 GAC에 DLL을 등록하는 공식적인 방법은 MSI(Microsoft Installer) 설치 파일로 만드는 것이 유일하다. 즉, 응용 프로그램을 배포할 설치 프로그램을 만들어야만 하는 번거로움을 수반한다. 이런저런 이유로 인해 지금은 마이크로소프트나 일부 라이브러리 제작 업체를 제외하고는 GAC를 통한 응용 프로그램 배포 방법은 잘 사용하지 않는 편이다. 사실 DLL 공유로 인한 하드디스크의 공간 절약이나 메모리 절약 효과는 대용량화되는 요즘의 컴퓨터 시대에 의미가 많이 줄어들었다. 그보다는 관리의 편의성을 위해 XCopy 배포가 선호되어 전용 어셈블리를 더 자주 사용하는 편이다.

6 장: BCL (Base Class Library)

[책 내용]

6.3 직렬화/역직렬화

6.3.5 System.Runtime.Serialization.Formatters.Binary.BinaryFormatter (닷넷 프레임워크)

지금까지 C#의 기본 타입을 직렬화/역직렬화하는 주제를 다뤘다. 그렇다면 다음과 같은 사용자 정의 클래스는 어떻게 직렬화할 수 있을까?

예제 1.4 직렬화 예제 클래스 – Person

```
class Person
{
    public int Age;
    public string Name;

    public Person(int age, string name)
    {
        this.Age = age;
        this.Name = name;
    }

    public override string ToString()
    {
        return string.Format("{0} {1}", this.Age, this.Name);
    }
}
```

물론 Age 값은 BitConverter 타입으로, Name 값은 Encoding 타입을 이용해 각각 바이트 배열로 바꾼 다음 Stream에 쓰면 된다. 복원은 Person 객체를 만든 다음 마찬가지 방법으로 Age, Name 속성에 값을 대입해 주면 된다.

하지만 당연히 그런 절차는 번거로울 수밖에 없다. 마이크로소프트에서는 이런 불편함을 없애기 위해 별도의 직렬화 클래스를 제공하는데, 그것이 바로 BinaryFormatter 타입이다. 하지만 BinaryFormatter를 이용해 직렬화 혜택을 받으려면 Person 클래스 측에 요구되는 선행 조건을 만족해야 한다. 즉, 개발자가 직접 Person 클래스는 직렬화가 가능하다고 표시해줘야 한다는 것이다. 표시 방법은 간단하게 해당 클래스에 [Serializable] 특성을 지정하기만 하면 된다.

```
[Serializable]
class Person
{
    // ……[생략]……
}
```

이렇게 준비한 직렬화 가능 클래스를 BinaryFormatter 타입을 이용해 직렬화/역직렬화하는 코드는 다음과 같이 간단하게 작성할 수 있다.

예제 1.5 BinaryFormatter 사용 예

```
Person person = new Person(36, "Anderson");

BinaryFormatter bf = new BinaryFormatter();

// MemoryStream에 person 객체를 직렬화
MemoryStream ms = new MemoryStream();
bf.Serialize(ms, person);

ms.Position = 0;

// MemoryStream으로부터 역직렬화해서 복원
Person clone = bf.Deserialize(ms) as Person;

Console.WriteLine(clone); // 출력 결과: 36 Anderson
```

Serialize 메서드를 호출한 후 MemoryStream의 ToArray 메서드를 호출하면 직렬화된 데이터 내용을 바이트 배열로 얻을 수 있다. 이 바이트 배열을 네트워크를 넘어 다른 컴퓨터에 실행 중인 프로그램에 넘겨주면 어떨까? 그 프로그램에서는 바이트 배열로부터 다시 역직렬화를 수행해 원래의 Person 데이터를 복원할 수 있다.

[Serializable] 특성은 기본적으로 클래스 내의 모든 필드를 대상으로 직렬화를 수행한다. 하지만 경우에 따라 특정 필드는 제외하고 싶을 수도 있는데, 이때는 해당 필드에 [NonSerialized] 특성을 부여하면 된다. 예를 들어, Person 타입의 Age 필드의 값을 직렬화에서 제외하고 싶다면 다음과 같이 특성을 지정한다.

```
[Serializable]
class Person
{
```

```
    [NonSerialized]
    public int Age;

    public string Name;
// ..... [생략] .....
```

그런데 BinaryFormatter에는 한 가지 단점이 있다. 직렬화 방식이 닷넷 내부에서 고유하게 정의돼 있기 때문에 자바와 같은 상이한 플랫폼에서는 그 바이트 배열을 어떻게 역직렬화해야 원본 데이터를 복원할 수 있는지 알 수가 없다. 이를 두고 "상호운영성(interoperability)이 없다"고 표현한다. 물론 단점을 만회할 수 있는 장점도 있다. BinaryFormatter는 2진 데이터로 직렬화하기 때문에 기타 다른 직렬화 방법에 비해 속도가 빠르고 용량도 작다. 이 때문에 닷넷 응용 프로그램끼리 데이터를 교환해야 한다면 BinaryFormatter를 사용하는 것을 선호한다.

6.3.7 System.Runtime.Serialization.Json.DataContractJsonSerializer (닷넷 프레임워크)

이전에 배운 BinaryFormatter와 XmlSerializer는 각각 장/단점이 있었다. DataContractJsonSerializer는 두 타입의 장점만을 취한 효과를 낸다. 타입의 이름에 포함된 Json은 JavaScript Object Notation의 약어로 그것의 탄생 배경에는 웹에서 널리 사용되고 있는 자바스크립트 언어가 있다. 즉, DataContractJsonSerializer는 자바스크립트의 객체 직렬화 방식을 닷넷에서 동일하게 구현한다.

DataContractJsonSerializer는 System.Runtime.Serialization.dll에 포함돼 있기 때문에 사용하기 전에 **Error! Reference source not found.**의 참조 관리자를 이용해 "System.Runtime.Serialization"을 추가해야 한다. 사용법은 다른 직렬화 방식과 거의 유사하다. 다음은 예제 1.5에서 직렬화를 DataContractJsonSerializer로 변경한 예제다.

```
DataContractJsonSerializer dcjs =
                        new DataContractJsonSerializer(typeof(Person));

MemoryStream ms = new MemoryStream();

Person person = new Person(36, "Anderson");

// MemoryStream에 문자열로 person 객체를 직렬화
dcjs.WriteObject(ms, person);
```

```
ms.Position = 0;

// MemoryStream으로부터 객체를 역직렬화해서 복원
Person clone = dcjs.ReadObject(ms) as Person;

Console.WriteLine(clone); // 출력 결과: 36 Anderson
```

직렬화된 MemoryStream의 내용을 살펴보면 DataContractJsonSerializer의 장점을 이해할 수 있다.

```
byte [] buf = ms.ToArray();
Console.WriteLine(Encoding.UTF8.GetString(buf));
```

```
// 출력 결과
{"Age":36,"Name":"Anderson"}
```

같은 기능을 하는 XmlSerializer는 문자열 크기가 176글자였지만 DataContractJsonSerializer는 28자에 불과하다. 게다가 문자열로 돼 있어서 가독성이 높아 닷넷 이외의 플랫폼에서도 쉽게 데이터를 주고받아 해석할 수 있다. 이런 장점으로 인해 최근에는 객체 직렬화를 위한 방법으로 DataContractJsonSerializer를 선호하는 추세다.

6.7 네트워크 통신

6.7.6 System.Net.HttpWebRequest (닷넷 프레임워크)

BCL에는 HTTP 통신을 좀 더 쉽게 할 수 있는 HttpWebRequest 타입이 제공된다. 이 타입을 이용하면 **Error! Reference source not found.**의 작업을 아래와 같이 간단하게 처리할 수 있다.

```
// HttpWebRequest 타입은 내부적으로 TCP 소켓을 생성하고,
HttpWebRequest req =
                  WebRequest.Create("http://www.naver.com") as HttpWebRequest;
// GetResponse 호출 단계에서 지정된 웹 서버로 HTTP 요청을 보내고, 응답을 받는다.
HttpWebResponse resp = req.GetResponse() as HttpWebResponse;

// 응답 내용을 담고 있는 Stream으로부터 문자열을 반환해서 출력한다.
using (StreamReader sr = new StreamReader(resp.GetResponseStream()))
{
    string responseText = sr.ReadToEnd();
    Console.WriteLine(responseText);
    File.WriteAllText("naverpage.html", responseText);
}
```

저장된 "naverpage.html" 파일을 확인해 보면 **Error! Reference source not found.**과는 달리 HTTP 헤더 영역이 제거된 순수하게 HTTP 본문만 포함돼 있다. 이렇게 HttpWebRequest 객체는 HTTP 통신과 관련된 요청/응답 데이터를 적절하게 해석하는 역할까지 대행하므로 TCP 소켓을 직접 사용해서 통신해야 하는 불편함이 줄어든다.

6.7.7 System.Net.WebClient (닷넷 프레임워크)

HttpWebRequest도 복잡하게 느껴진다면 더 기능을 추상화한 WebClient 타입을 고려해 보자. 이 타입을 이용하면 **Error! Reference source not found.**을 단 두 줄로 줄일 수 있다.

```
// WebClient 타입은 내부적으로 HttpWebRequest 객체를 사용해 통신
WebClient wc = new WebClient();
string responseText = wc.DownloadString("http://www.naver.com");
```

WebClient 타입에는 이 밖에도 HTTP 통신을 이용해 파일을 업로드할 수 있는 UploadFile, 반대로 파일을 다운로드하는 DownloadFile과 같은 유용한 메서드가 제공되므로 마이크로소프트의 docs 도움말을 통해 기능을 좀 더 알아보는 것도 좋다.

참고!	비주얼 스튜디오 사용자라면 코드 에디터 창에서 WebClient에 커서를 두고 F1 키를 누르면 곧바로 웹 브라우저가 실행되면서 docs 도움말로 이동해 관련된 문서를 보여준다.

6.8 데이터베이스

6.8.2 ADO.NET 데이터 제공자 (닷넷 프레임워크)

참고!	이번 절의 내용은 닷넷 프레임워크용 프로젝트에서 실습할 수 있다. 만약 닷넷 코어/5+ 환경이라면 별도의 선행 작업이 필요한데, 이번 절에서 실습할 System.Data.SqlClient.dll 라이브러리를 닷넷 코어부터는 별도의 패키지로 분리시켜 배포하기 때문이다. 따라서 이번 실습을 닷넷 코어/5+ 프로젝트에서 실습하려면 System.Data.SqlClient 패키지를 수동으로 프로젝트에 추가해야 한다. 이를 위해 "도구" → "NuGet 패키지 관리자" → "패키지 관리자 콘솔" 메뉴를 선택한 후 뜨는 "패키지 관리자 콘솔" 창에서 다음과 같은 명령어를 입력해 해당 패키지를 설치한다. `Install-Package System.Data.SqlClient` 참고로, 마이크로소프트는 System.Data.SqlClient를 닷넷 코어/5+ 환경에 맞게 보다 개선한 Microsoft.Data.SqlClient 패키지를 제공하며, 제공하는 클래스와 그것의 메서드 이름들이 모두 이 글에서 설명하는 내용과 일치하므로 원한다면 그 패키지를 이용해 실습할 수 있다.

데이터베이스 프로그램은 대부분 TCP 서버로 동작한다. 따라서 TCP 클라이언트 프로그램에서 데이터베이스를 사용하려면 서버의 IP 주소 또는 컴퓨터 이름과 함께 포트 번호가 필요하다. SQL 서버의 경우 기본적으로 1433 포트 번호를 사용하므로 만약 SQL 서버가 192.168.0.10 컴퓨터에서 실행 중이라면 그 접점 정보는 192.168.0.10:1433이 된다. SQL 서버와 통신하기 위해 그다음으로 알아야 할 것은 데이터를 주고받는 프로토콜 형식이다. 다행히 이 프로토콜 형식을 개발자가 알 필요는 없다. 왜냐하면 데이터베이스를 만든 업체에서 프로토콜을 가장 잘 알고 있기 때문에 데이터베이스 통신을 위한 전용 라이브러리를 제작해서 배포하기 때문이다. 결국 개발자가 알아야 할 것은 DB 서버와의 통신 프로토콜이 아니라 해당 라이브러리를 어떻게 사용하느냐다.

이러한 전용 라이브러리를 일컬어 "ADO.NET 데이터 제공자(data provider)"라고 하며, 모든 ADO.NET 데이터 제공자는 마이크로소프트에서 미리 정의해 둔 다음의 공통 인터페이스를 상속받아 구현한다.

- System.Data.IDbConnection
  데이터베이스 서버와의 연결을 담당하는 클래스가 구현해야 할 인터페이스 정의
- System.Data.IDbCommand
  데이터베이스 서버 측으로 실행될 SQL 문을 전달하는 클래스가 구현해야 할 인터페이스 정의
- System.Data.IDataReader
  실행된 SQL 문으로부터 반환받은 데이터를 열람하는 클래스가 구현해야 할 인터페이스
- System.Data.IDbDataParameter
  IDbCommand에 전달되는 SQL 문의 인자(Parameter) 값을 보관하는 클래스가 구현해야 할 인터페

이스 정의

- System.Data.IDbDataAdapter
  System.Data.DataTable 개체와 상호작용하는 Data Adapter 클래스가 구현해야 할 인터페이스 정의

ADO.NET 데이터 제공자는 보통 데이터베이스 서버를 만든 업체에서 만들어서 배포한다. 마이크로소프트 SQL 서버의 경우 닷넷을 만든 마이크로소프트에서 만들었기 때문에 그에 대한 ADO.NET 데이터 제공자를 BCL에 포함시켜서 배포하고 있으므로 별도로 내려받을 필요는 없다. 하지만 여러분이 실습하는 데이터베이스 서버가 MySQL이라면 오라클 웹 사이트를 방문해 MySQL용 ADO.NET 데이터 제공자를 내려받아 사용해야 한다.

마이크로소프트 SQL 서버용 ADO.NET 데이터 제공자는 BCL에서 System.Data.SqlClient 네임스페이스 아래에 각각 다음과 같이 구현돼 있다.

표 1.8 Microsoft SQL 서버 ADO.NET 데이터 제공자

인터페이스	SQL 서버용 ADO.NET 구현 클래스
System.Data.IDbConnection	System.Data.SqlClient.SqlConnection
System.Data.IDbCommand	System.Data.SqlClient.SqlCommand
System.Data.IDataReader	System.Data.SqlClient.SqlDataReader
System.Data.IDbDataParameter	System.Data.SqlClient.SqlParameter
System.Data.IDbDataAdapter	System.Data.SqlClient.SqlDataAdapter

이론상 적어도 데이터베이스 수만큼 ADO.NET 데이터 제공자가 있을 테지만 모두 System.Data에서 제공되는 인터페이스를 상속받아 구현하고 있으므로 사용법을 한번 익혀두면 다른 데이터 제공자도 어렵지 않게 사용할 수 있다.

이 책에서는 마이크로소프트 SQL 서버를 대상으로 하므로 System.Data.SqlClient 네임스페이스에서 제공되는 ADO.NET 데이터 제공자의 사용법을 설명한다.

6.8.3 데이터 컨테이너 (닷넷 프레임워크)

6.8.3.3 Typed DataSet (닷넷 프레임워크)

"Typed DataSet"은 형식 안전성이 부여된 DataSet으로 비주얼 스튜디오 내에서 특정한 메뉴를

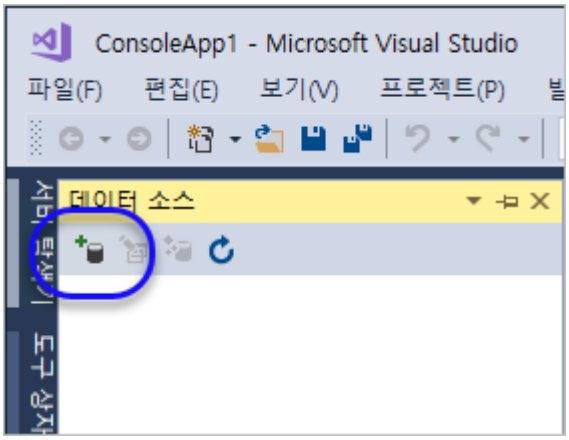
통해 자동 생성된다. '형식 안전성(type safety)'이라는 단어가 지금은 낯설겠지만 Typed DataSet을 직접 만들고 사용해 보면 자연스럽게 이해할 수 있을 것이다.

참고!	Typed DataSet은 비주얼 스튜디오에서 제공하는 마법사 기능을 사용하므로 이번 절의 실습은 Visual Studio Code 등의 환경에서는 진행할 수 없다. 또한 닷넷 프레임워크용 프로젝트에서만 동작하므로 닷넷 코어/5+ 프로젝트에서는 실습할 수 없다.

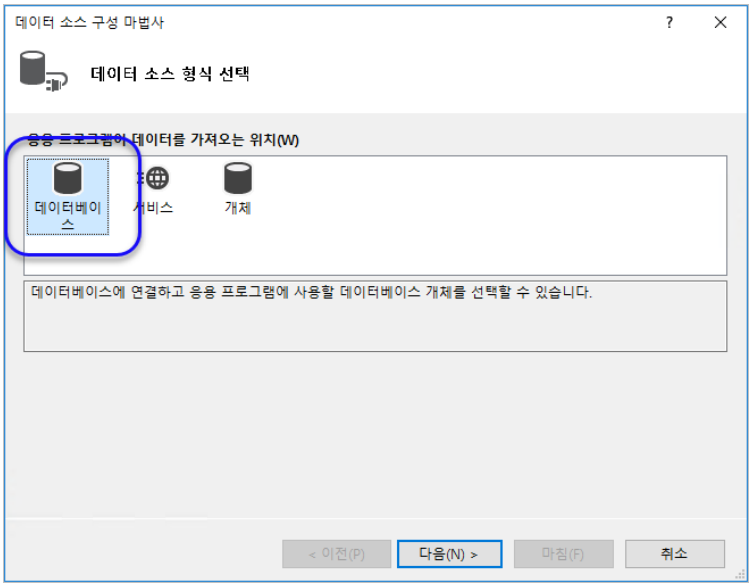
비주얼 스튜디오에서 "Typed DataSet"을 만드는 과정은 다소 복잡하다. 하지만 일정한 패턴이 있기 때문에 일단 한번 알아두면 쉽게 재사용할 수 있고 데이터베이스 한 개당 한 번만 생성하는 것이 보통이므로 크게 문제가 되진 않는다. 그럼 이제 Typed DataSet을 프로젝트에 추가해 보자.

1. 비주얼 스튜디오에서 "Typed DataSet"을 만들 프로젝트를 로드한 다음, "보기(View)" → "다른 창(Other Windows)" → "데이터 소스(Data Sources)" 메뉴를 차례로 선택한다(단축키: Shift + Alt + D).

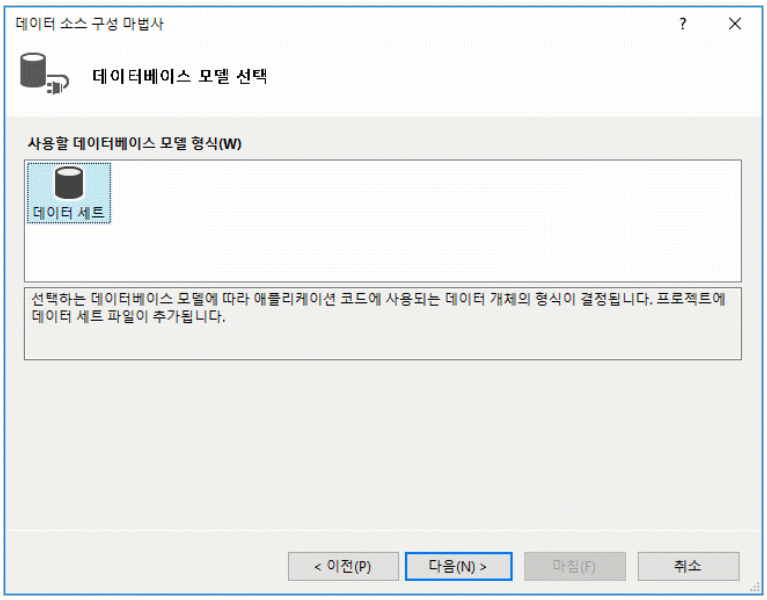
   그러고 나면 왼쪽에 "Data Sources" 창이 나타나는데, 툴바의 "새 데이터 소스 추가(Add New Data Source)" 버튼을 누른다.



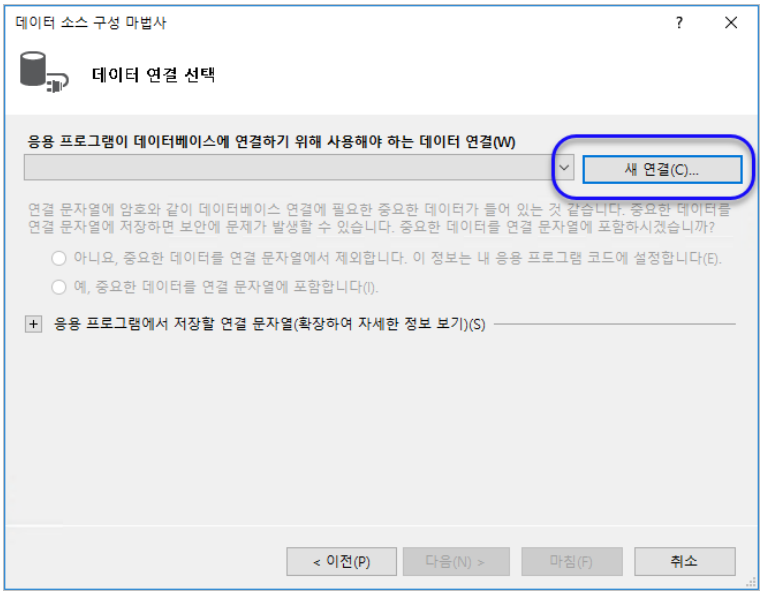
2. "데이터 소스 구성 마법사(Data Source Configuration Wizard)"가 실행되면 "데이터베이스(Database)"를 선택하고 진행한다.



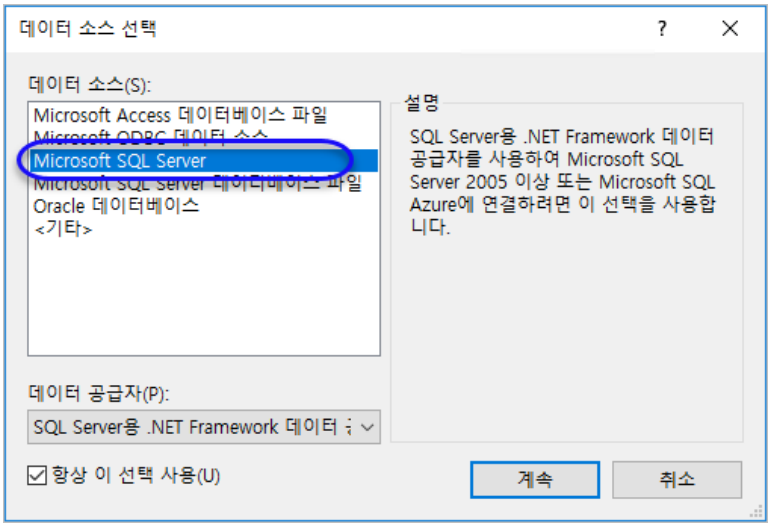
3. "데이터베이스 모델(Database Model)"을 선택하는 화면에서는 "데이터 세트(Dataset)"를 선택한다.



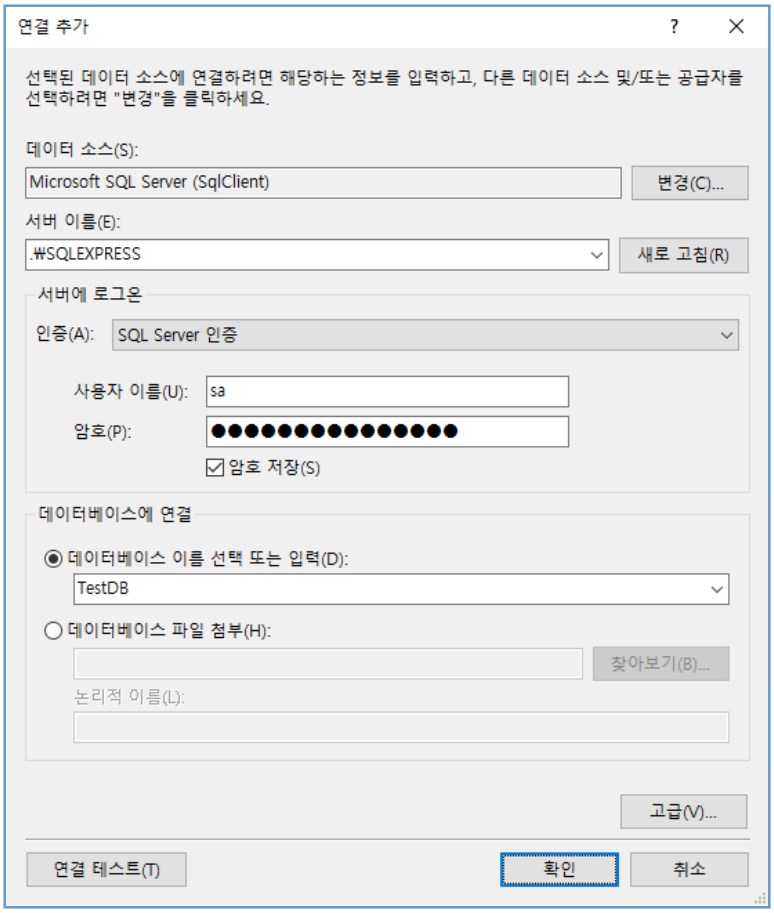
4. "데이터 연결(Data Connection)" 단계가 나오면 "새 연결(New Connection)..." 버튼을 누른다.



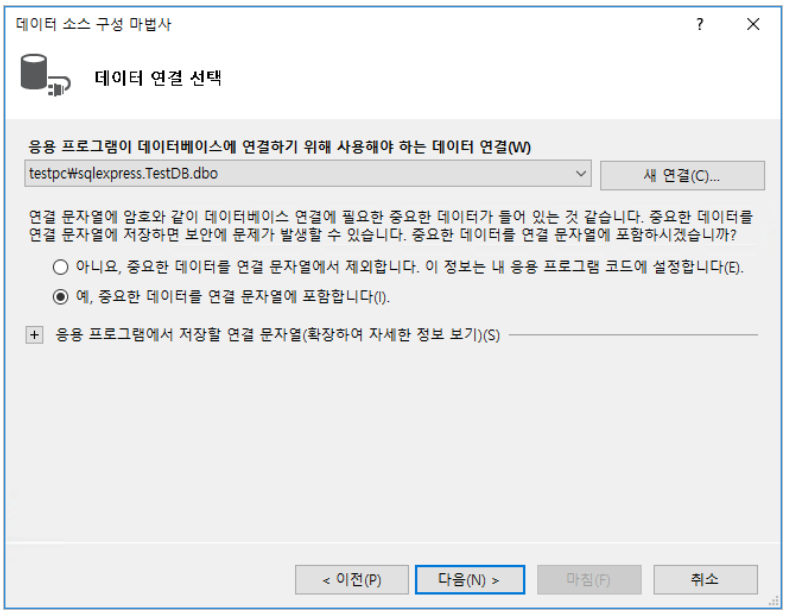
5. "데이터 소스(Data source)" 목록에서 "Microsoft SQL Server"를 선택하고, 계속(Continue) 버튼을 누른다.



6. "연결 추가(Add Connection)" 단계에서는 자신의 실습 환경에 맞는 SQL 서버 연결 정보를 입력하고 확인 버튼을 누른다.

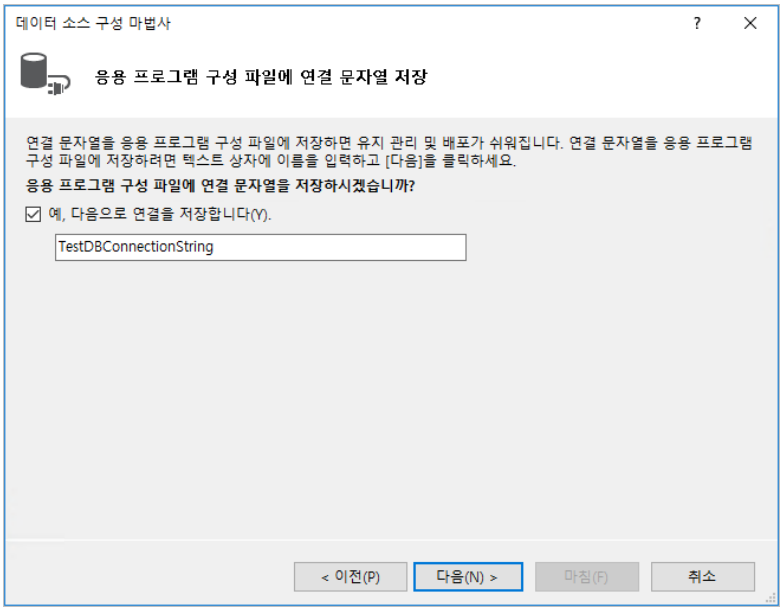


그럼 "데이터 연결" 선택 화면으로 돌아가서 방금 구성한 SQL 서버 연결 정보가 목록에 추가된다. "예, 중요한 데이터를 연결 문자열에 포함합니다(Yes, include sensitive data in the connection string)." 옵션을 선택하고 다음 단계로 진행한다. 이 옵션이 선택 불가능한 상태로 있어도 상관없이 그냥 다음 단계로 넘어가면 된다.

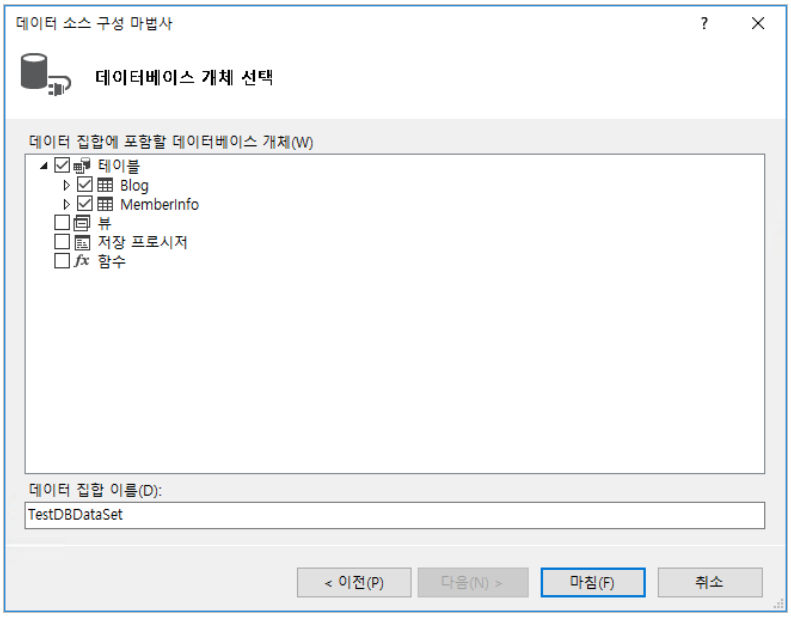


7. "응용 프로그램 구성 파일에 연결 문자열 저장(Save the Connection String to the Application

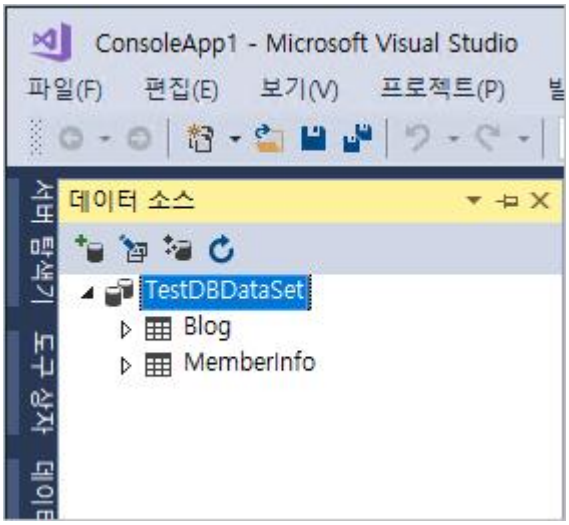
Configuration File)"은 기본적으로 선택된 값 그대로 진행한다.



8. 마지막으로 "데이터베이스 개체 선택(Choose Your Database Objects)" 화면에서는 실습 목적으로 만든 테이블 2개를 선택해서 작업을 완료한다.

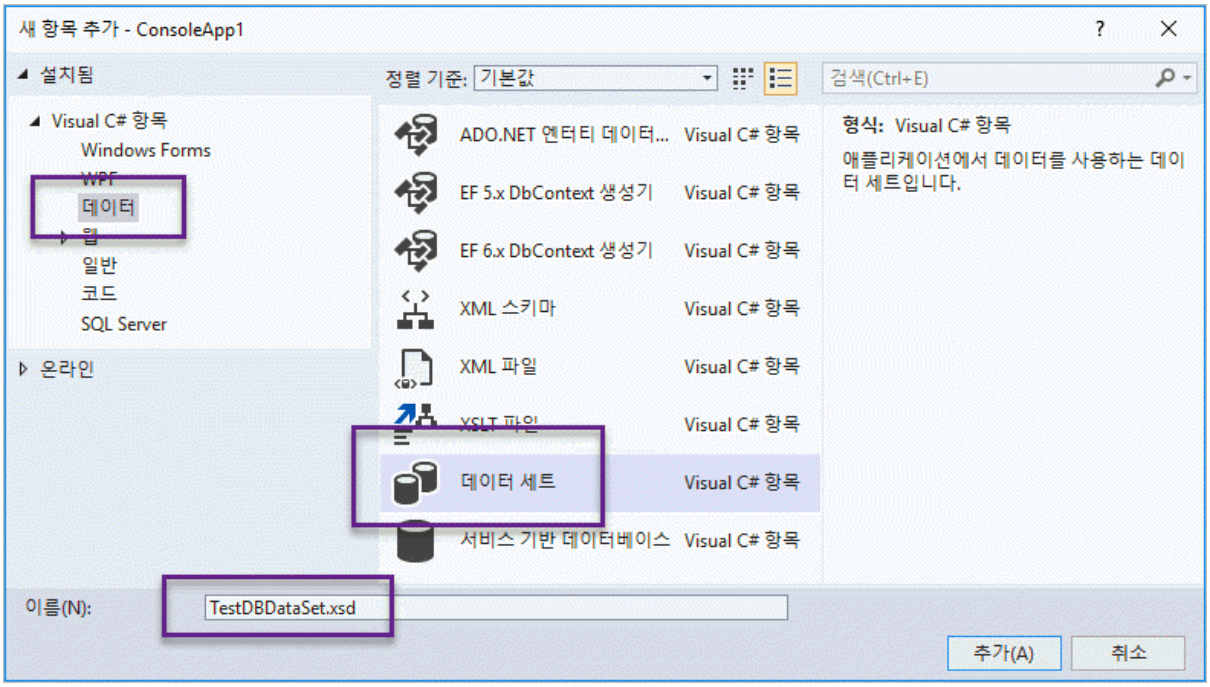


9. 정상적으로 완료되면 "데이터 소스" 패널에 다음과 같이 Typed DataSet이 포함된 것을 확인할 수 있다.



지금까지 설명한 방법 말고도 "Typed DataSet"을 생성하는 방법이 하나 더 있다. 솔루션 탐색기에서 프로젝트 항목을 대상으로 마우스 오른쪽 버튼을 누른 다음 "추가(Add)" → "새 항목(New Item)..."을 선택한 후(단축키: Ctrl + Shift + A) 다음과 같이 "데이터(Data)" 범주의 "데이터 세트(DataSet)"를 선택하면 된다. 이후의 과정은 데이터 소스를 통해 추가했던 것과 비슷하게 진행한다.

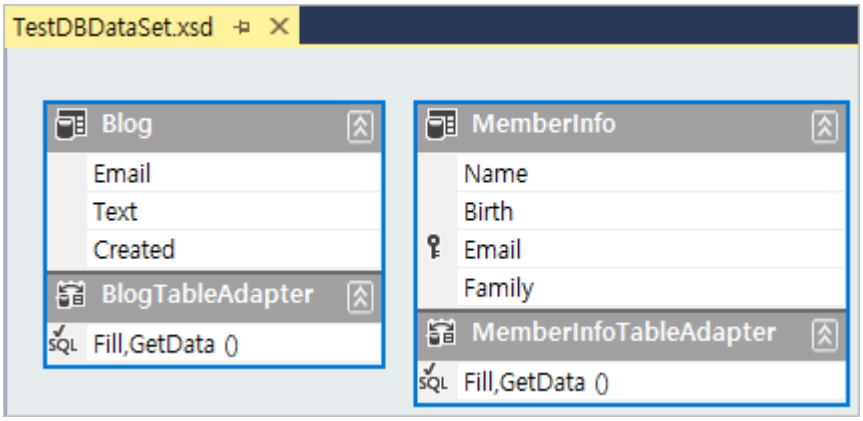
그림 1.14 "새 항목" 추가 메뉴를 이용한 Typed DataSet 추가



이렇게 Typed DataSet이 추가되면 프로젝트에 크게 두 가지 변화가 생긴다.

- App.config에 연결 문자열 추가
  생성된 TestDBDataSet은 자동으로 App.config 파일에 데이터베이스 연결 문자열을 저장한다.

- TestDBDataSet.xsd 파일 추가
  TestDBDataSet.xsd 파일을 편집 화면으로 열면 다음과 같이 2개의 테이블 정보가 담긴 화면이 나타난다.



  보다시피 우리가 실습용 테이블로 만들어 둔 모든 정보가 들어 있다. 솔루션 탐색기에서 TestDBDataSet.xsd 하위에 있는 TestDBDataSet.Designer.cs 파일도 편집 창으로 열어보자. 그럼 수많은 코드가 들어 있는 것을 확인할 수 있다. 즉, 비주얼 스튜디오에서는 데이터 소스로 지정된 데이터베이스로부터 테이블에 관한 정보를 조회해서 이렇게 미리 관련 소스코드까지 모두 생성해 둔다.

자, 그럼 DAC 클래스를 만들지 않고도 기본 생성된 Typed DataSet만으로도 **Error! Reference source not found.**와 **Error! Reference source not found.**에서 했던 것과 동일한 코드를 만들 수 있다.

예제 1.6 Typed DataSet을 이용한 데이터베이스 연동

```
using System;
using ConsoleApp1;
using ConsoleApp1.TestDBDataSetTableAdapters;

class Program
{
    static void Main(string[] args)
    {
        TestDBDataSet ds = new TestDBDataSet();
        MemberInfoTableAdapter da = new MemberInfoTableAdapter();

        // 새로운 레코드를 삽입: INSERT
        da.Insert("Julie", new DateTime(1985, 5, 6), "julie@naver.com", 1);

        // 테이블의 모든 레코드를 조회: SELECT
        da.Fill(ds.MemberInfo);

        foreach (TestDBDataSet.MemberInfoRow row in ds.MemberInfo)
        {
            Console.WriteLine(string.Format("{0}, {1}, {2}, {3}",
                row.Name, row.Birth, row.Email, row.Family));
        }
```

```
        // 테이블의 특정 레코드의 값을 변경: UPDATE
        TestDBDataSet.MemberInfoRow[] rows = ds.MemberInfo.Select("Name = 'Julie'")
            as TestDBDataSet.MemberInfoRow[];

        rows[0].Name = "July";
        da.Update(rows[0]);

        // 테이블의 특정 레코드를 삭제: DELETE
        da.Delete(rows[0].Name, rows[0].Birth, rows[0].Email, rows[0].Family);
    }
}
```

이렇게 코드가 간단하게 바뀔 수 있는 것은 TestDBDataSet.Designer.cs 파일 덕분인데, 크게 다음과 같이 4개의 자동 생성된 코드가 들어 있기 때문이다.

1. Blog 테이블과 MemberInfo 테이블에 대해 각각 DataTable을 상속받은 BlogTable, MemberInfoTable 타입이 추가돼 있다. 물론, 각 테이블에는 **Error! Reference source not found.**와 **Error! Reference source not found.**에서 수작업으로 구성했던 DataColumn 정의가 포함돼 있다.
2. 칼럼별로 공용 속성이 추가돼 있다. 그 덕분에 **Error! Reference source not found.**처럼 DataRow 객체에 문자열을 지정해 속성을 접근(예: `row["Email"]`)하는 방식보다 안전하게 `row.Email`과 같은 속성으로 접근할 수 있다.
3. Blog 테이블과 MemberInfo 테이블에 대해 각각 DataAdapter 역할을 하는 BlogTableAdapter, MemberInfoTableAdapter 타입이 추가돼 있다. 각 Adapter에는 **Error! Reference source not found.**에서 수작업으로 구성했던 SQL 문과 함께 INSERT, UPDATE, DELETE에 대해 기본적인 쿼리 문이 포함돼 있다.
4. 각 DataAdapter에는 쿼리 문에 필요한 SqlParameter 코드도 생성돼 있다. 따라서 **Error! Reference source not found.**에서 했던 SqlParameter 목록에 대한 수작업이 필요 없다.

정리하자면 데이터 컨테이너는 결국 데이터베이스에 정의된 "관계형 테이블 구조"를 프로그래밍 언어에 정의된 타입에 대응시킨 것을 의미한다. 이를 가리켜 "ORM(Object-Relational Mapping)", 흔히 OR 매핑이라고 하는데 OR 매핑과 관련된 기술 개선은 지금도 계속되고 있다. 예를 들어, 닷넷 프레임워크에 기본 내장된 "엔티티 프레임워크(Entity Framework)"나 오픈소스로 알려진 NHibernate 프레임워크는 모두 OR 매핑이 가능한 라이브러리다. 두 프레임워크 모두 별도의 책으로 다뤄야 할 만큼 긴 설명이 필요하므로 여기서는 다루지 않는다.

6.9 리플렉션 (닷넷 프레임워크)

앞에서 닷넷 응용 프로그램의 어셈블리 파일 안에는 메타데이터(metadata)가 있다고 배웠다. 하지만 그 말의 의미를 실감할 수는 없었을 것이다. BCL에서 제공되는 리플렉션(reflection) 관련 클래스를 이용하면 메타데이터 정보를 얻는 것이 가능하므로 이제 그 유용성을 직접 코드로 확인해 보자.

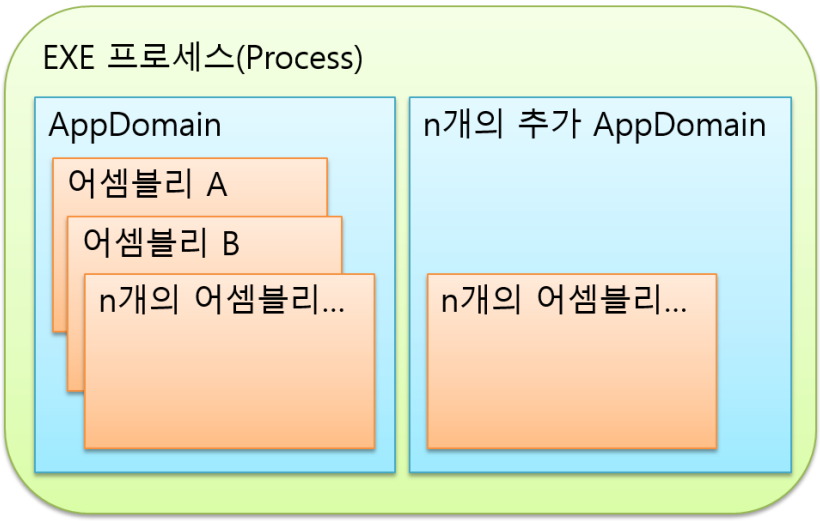
리플렉션을 본격적으로 알아보기에 앞서 우선 닷넷 응용 프로그램의 프로세스 구조를 먼저 살펴볼 필요가 있다. 닷넷 프로세스는 운영체제에서 EXE 프로세스로 실행되고 그 내부에 CLR에 의해 "응용 프로그램 도메인(AppDomain: Application Domain)"이라는 구획으로 나뉜다. AppDomain은 CLR이 구현한 내부적인 격리 공간이다. 따라서 AppDomain 간에는 별도의 통신 방법을 설정하지 않는 한 서로의 영역을 침범할 수 없다. 심지어 하나의 AppDomain이 불안정하게 종료된다고 해도 다른 AppDomain이 동작하는 데는 아무런 영향도 미치지 않는다. 일반적으로 여러분들이 만들게 되는 콘솔 응용 프로그램은 1개의 공유 AppDomain[4]과 1개의 기본 AppDomain으로 시작하지만 원한다면 임의로 만드는 것도 가능하다.

참고!	2개 이상의 AppDomain은 닷넷 프레임워크 버전에서 지원되고, 닷넷 코어/5+에서는 오직 한 개의 AppDomain만 허용한다.

AppDomain이 만들어지면 그 내부에 어셈블리들이 로드된다. 그림 1.15의 다이어그램은 이러한 관계를 잘 보여준다.

그림 1.15 프로세스 – AppDomain – 어셈블리의 관계

[4] 엄밀히 공유 AppDomain은 이름만 Domain이 붙을 뿐 다소 특별한 유형에 속한다. (참고: http://www.sysnet.pe.kr/2/0/10948)



즉, 지금까지 실습한 ConsoleApp1.exe 어셈블리는 닷넷 EXE 프로세스가 시작하면서 생성된 기본 AppDomain에 로드되어 실행된 것이다. 리플렉션을 이용하면 현재 AppDomain의 이름과 그 안에 로드된 어셈블리 목록을 구할 수 있다.

예제 1.7 현재 AppDomain에 로드된 어셈블리 목록

```
using System;
using System.Reflection;

class Program
{
    static void Main(string[] args)
    {
        AppDomain currentDomain = AppDomain.CurrentDomain;
        Console.WriteLine("Current Domain Name: " + currentDomain.FriendlyName);
        foreach (Assembly asm in currentDomain.GetAssemblies())
        {
            Console.WriteLine("  " + asm.FullName);
        }
    }
}
```

```
// 출력 결과
Current Domain Name: ConsoleApp1.exe
  System.Private.CoreLib,            Version=6.0.0.0,            Culture=neutral,
PublicKeyToken=7cec85d7bea7798e
  ConsoleApp1, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
…[ 이하 생략]…
```

6.9.2 AppDomain과 Assembly (닷넷 프레임워크)

참고!	이번 절의 실습은 닷넷 프레임워크용 프로젝트에서만 가능하다. 닷넷 코어/5+의 경우 다중 도메인을 지원하지 않으므로 AppDomain.CreateDomain 메서드를 호출하면 지원하지 못한다는 예외가 발생한다.

AppDomain은 EXE 프로세스 내에서 CLR에 의해 구현된 격리 공간이라고 설명했다. 원한다면 누구나 AppDomain을 별도로 생성하는 것이 가능하다.

```
AppDomain newAppDomain = AppDomain.CreateDomain("MyAppDomain");
```

현재 스레드가 실행 중인 어셈블리가 속한 AppDomain 인스턴스는 예제 1.7에서 설명한 대로 CurrentDomain 정적 속성을 이용해 접근할 수 있다.

```
AppDomain currentDomain = AppDomain.CurrentDomain;
```

닷넷 프로그램이 실행되면 기본적으로 1개의 AppDomain이 있어야 하는데, 이를 가리켜 "기본 응용 프로그램 도메인(default AppDomain)"이라 한다.

AppDomain 내에 어셈블리를 로드하는 간단한 방법은 CreateInstanceFrom 메서드를 이용해 어셈블리 파일의 경로와 최초 생성될 객체의 타입명을 지정하는 것이다. 이를 실습하기 위해 콘솔 응용 프로그램 외에 별도로 DLL 라이브러리 프로젝트를 만들고 다음과 같은 클래스를 정의해 두자.

```
using System;

namespace ClassLibrary1
{
    public class Class1
    {
        public Class1()
        {
            Console.WriteLine(typeof(Class1).FullName + ": Created");
        }
    }
}
```

DLL을 빌드하고 해당 DLL 파일의 경로명과 클래스의 완전한 이름(FQDN: 네임스페이스 경로까지 포함한 이름)을 알아야 한다. 실습하고 있는 코드는 다음과 같다고 가정하고 진행한다.

DLL 경로: `D:\Test\ClassLibrary1\bin\Debug\ClassLibrary1.dll`
Class1의 완전한 이름: `ClassLibrary1.Class1`

이제 콘솔 EXE 프로젝트 내의 코드에서 CreateInstanceFrom 메서드를 사용해 이 값들을 전달하면 해당 AppDomain에 어셈블리가 로드됨과 함께 클래스의 인스턴스가 하나 생성된다.

```
using System;
using System.Reflection;
using System.Runtime.Remoting;

class Program
{
    static void Main(string[] args)
    {
        AppDomain newAppDomain = AppDomain.CreateDomain("MyAppDomain");

        string dllPath = @"D:\Test\ClassLibrary1\bin\Debug\ClassLibrary1.dll";

        ObjectHandle objHandle =
              newAppDomain.CreateInstanceFrom(dllPath, "ClassLibrary1.Class1");
    }
}
```

예제 코드를 직접 실행해 보면 화면에는 "ClassLibrary1.Class1: Created"라는 문자열이 출력되는 것을 확인할 수 있다. 즉, 해당 클래스의 생성자가 실행된 것이다.

AppDomain에 어셈블리 파일이 로드되면 AppDomain이 내려가지 않고서는 해당 어셈블리를 해제할 방법이 없다. C/C++로 윈도우 프로그램을 만들어 본 경험이 있는 개발자는 DLL 파일을 LoadLibrary API를 이용해 프로세스에 로드했다가 FreeLibrary API를 이용해 메모리로부터 해제할 수 있다는 사실을 알고 있을 것이다. 닷넷 응용 프로그램의 경우, 어셈블리 자체를 해제하는 방법은 제공되지 않고 반드시 그것이 속한 AppDomain을 해제하는 경우에만 해당 AppDomain에 속한 어셈블리가 모두 해제된다. 다음 코드를 통해 이 같은 동작 방식을 확인할 수 있다.

예제 1.8 별도의 AppDomain에 로드되는 어셈블리

```
AppDomain newAppDomain = AppDomain.CreateDomain("MyAppDomain");

string dllPath = @"D:\Test\ClassLibrary1\bin\Debug\ClassLibrary1.dll";
ObjectHandle objHandle =
       newAppDomain.CreateInstanceFrom(dllPath, "ClassLibrary1.Class1");

Console.WriteLine("엔터키를 치기 전까지 ClassLibrary1.dll 파일을 지울 수 없습니다.");
Console.ReadLine();

AppDomain.Unload(newAppDomain); // AppDomain을 해제시킨다.

Console.WriteLine("이젠 ClassLibrary1.dll 파일을 지울 수 있습니다.");
Console.ReadLine();
```

그렇다면 Default AppDomain에 로드된 어셈블리는 어떻게 해제할 수 있을까? Default AppDomain

은 닷넷 프로세스가 시작하면서 생성된 기본 AppDomain이기 때문에 프로세스가 종료될 때까지는 해제할 수 없다. 따라서 Default AppDomain에 한번 로드된 어셈블리 파일은 해제할 수 없다.

마지막으로, 각 AppDomain이 격리된 상태라는 점을 잊어서는 안 된다. 예제 1.8에서 "MyAppDomain"에 로드된 ClassLibrary1.dll은 Default AppDomain에 보이지 않는다. 따라서 Default AppDomain에서 ClassLibary1.Class1 타입을 직접 사용할 수는 없다. 게다가 코드뿐 아니라 데이터까지도 격리됐다는 사실을 알아두자. C/C++ 개발자들이 C#을 사용하면서 흔히 실수하는 부분이 바로 클래스의 정적 필드가 프로세스(EXE)마다 고유하다고 생각한다는 점이다. 클래스에 정의된 static 필드는 AppDomain에 의해 격리되므로 EXE 프로세스에 여러 개의 AppDomain이 있고 그러한 AppDomain에서 동일한 클래스를 로드했다면 static 필드는 AppDomain마다 하나씩 존재하게 된다.

2. 부: C# 고급 문법

[책 내용]

7 장: C# 2.0

[책 내용]

8 장: C# 3.0

[책 내용]

9 장: C# 4.0

[책 내용]

10장: C# 5.0

[책 내용]

11장: C# 6.0

11. 10 기타 개선 사항

catch/finally 블록 내에서 await 사용 가능

C# 5.0의 경우 catch와 finally의 예외 처리 블록 내에서 비동기 호출(await)을 할 수 없다는 제약이 있었다. C# 개발팀 스스로도 이것이 가능할 거라고 생각지 않았다는데, 다행히 방법을 찾았고 C# 6.0부터 이를 지원해 다음과 같이 try/catch/finally 절에서의 비동기 호출을 깔끔하게 처리할 수 있게 됐다.

예제 2.1 catch/finally 절에서의 await 호출

```
using System;
using System.IO;
using System.Threading.Tasks;

class Program
{
    static void Main(string[] args)
    {
        ProcessFileAsync();
        Console.ReadLine();
    }

    private static async void ProcessFileAsync()
    {
        FileStream fs = null;

        try
        {
            fs = new FileStream("test.txt", FileMode.Open, FileAccess.Read);
            byte[] contents = new byte[1024];
            await fs.ReadAsync(contents, 0, contents.Length);
            Console.WriteLine("ReadAsync Called!");
        }
        catch (Exception e)
        {
            await LogAsync(e); // C# 5.0에서는 불가능했던 호출
        }
        finally
        {
```

```
            await CloseAsync(fs); // C# 5.0에서는 불가능했던 호출
        }
    }

    static Task LogAsync(Exception e)
    {
        return Task.Factory.StartNew(
            () =>
            {
                Console.WriteLine("Log Async Called!");
            });
    }

    static Task CloseAsync(FileStream fs)
    {
        return Task.Factory.StartNew(
            () =>
            {
                Console.WriteLine("Close Async Called!");

                if (fs != null)
                {
                    fs.Close();
                }
            });
    }
}

// 1. 출력 결과(예외가 발생하지 않은 경우)
ReadAsync Called!
Close Async Called!

// 2. 출력 결과(예외가 발생한 경우)
Log Async Called!
Close Async Called!
```

#pragma의 "CS" 접두사 지원

#pragma 지시자의 변화는 엄밀히 C# 언어의 문법과는 무관하다. 예를 들어, 다음의 코드를 컴파일하면 "CS0168" 경고가 발생하는데

```
class Program
{
    static void Main(string[] args)
    {
        int i; // 경고 CS0168: 'i' 변수가 선언되었지만 사용되지 않았습니다.
    }
}
```

기존에는 이 경고를 끄려면 #pragma에 "CS"를 뺀 0168 숫자만 넣어야 했지만

```
#pragma warning disable 0168
class Program
```

```
{
    static void Main(string[] args)
    {
        int i; // 경고 없이 컴파일
    }
}
```

C# 6.0부터는 컴파일러 메시지 그대로 "CS0168"을 사용할 수 있도록 바뀌었다.

```
#pragma warning disable CS0168

#pragma warning disable 0168

// C# 6.0부터 위의 2가지 구문 모두 지원
```

참고로 비주얼 스튜디오의 경우, 프로젝트 속성 창의 "빌드" 탭에 제공되는 "오류 및 경고" / "경고 표시 안 함(Suppress warnings)" 값에도 동일하게 이 규칙이 적용된다.

재정의된 메서드의 선택 정확도를 향상

엄밀히 말하면 이번 절의 내용은 문법적인 추가 부분이 아닌 컴파일러 자체의 기능 향상에 따른 변화에 해당한다. 가령 nullable 타입을 인자로 받는 메서드들이 중복 정의(overload)된 경우 기존 C# 5.0 컴파일러는 다음의 상황에서 어떤 메서드를 선택해야 할지 몰라 컴파일 오류를 발생시킨다.

```
using System;

class Program
{
    static void WriteLine(uint? arg)
    {
        Console.WriteLine("uint? == " + arg);
    }

    static void WriteLine(int? arg)
    {
        Console.WriteLine("int? == " + arg);
    }

    static void Main(string[] args)
    {
        WriteLine(null); // C# 5.0 이전에는 컴파일 오류 발생
    }
}
```

하지만 C# 6.0 컴파일러에서는 "int? arg"를 정의한 메서드를 선택해 컴파일이 이뤄진다. 이는 일관성 측면에서 볼 때 의미 있는 변화다. 일례로 다음 코드를 실행해 보면

```
using System;

class Program
{
    static void WriteLine(uint arg)
    {
        Console.WriteLine("uint == " + arg);
    }

    static void WriteLine(int arg)
    {
        Console.WriteLine("int == " + arg);
    }

    static void Main(string[] args)
    {
        WriteLine(5); // C# 5.0/6.0 모두 int 인자를 받는 메서드 선택
    }
}
```

C# 5.0 컴파일러에서도 int 형 메서드가 선택되기 때문에 C# 6.0에서는 기본 타입과 nullable 타입 간의 선택 기준에 차이가 없어진 것이다.

마이크로소프트의 문서에 의하면 이 밖에도 delegate를 인자로 받는 중복 정의 메서드에 대한 선택에 대해서도 정확도가 높아졌다고 언급하고 있다. 다음은 이를 확인할 수 있는 예제 코드다.

```
using System;
using System.Threading.Tasks;

class Program
{
    static Task NullTask()
    {
        Console.WriteLine("NullTask");
        return null;
    }

    static void Main()
    {
        Task.Run(NullTask); // C# 5.0 이전에는 컴파일 오류
    }
}
```

Task.Run 메서드는 인자가 다른 8개의 중복 정의 메서드가 제공되는데 그중에서 다음의 2가지 메서드 중에서

```
public static Task Run(Func<Task> function);
public static Task Run(Action action);
```

NullTask 인자에 대해 어떤 것을 선택해야 하는지 판단할 수 없어 컴파일 오류가 발생한다. 하지만 C# 6.0부터는 첫 번째 유형인 Run(Func<Task>) 메서드를 선택해 컴파일을 성공시킨다.

12장: C# 7.0

13장: C# 7.1

13. 4 기타 개선 사항

패턴 매칭 – 제네릭 추가

C# 7.0 컴파일러에 추가된 12.10절의 패턴 매칭은 제네릭 인자에 대한 패턴 매칭 구문을 허용하지 않는다. 예를 들어, 다음 코드는 C# 7.0에서 컴파일 오류가 발생한다.

```
static void Main(string[] args)
{
    WriteLog(DateTime.Now);
    WriteLog(DateTime.UtcNow);
}

// 제네릭 인자의 객체는 is 연산자와 switch/case 패턴 매칭 구문에서 허용되지 않으므로
// 컴파일 오류 발생(C# 7.0)
public static void WriteLog<T>(T item)
{
    if (item is DateTime time)
    {
        Console.WriteLine(time.ToString());
    }

    switch (item)
    {
        case DateTime dt when dt.Kind == DateTimeKind.Utc:
            Console.WriteLine(dt.ToLocalTime());
            break;

        case DateTime dt when dt.Kind == DateTimeKind.Unspecified:
            Console.WriteLine("Invalid DateTime Kind");
            break;

        case DateTime dt:
            Console.WriteLine(dt);
            break;
    }
```

```
}
```

하지만 C# 7.1부터 제네릭에 대한 패턴 매칭이 가능하기 때문에 정상적으로 컴파일된다.

참조 전용 어셈블리(Ref Assemblies)

마이크로소프트의 C# 7.1 기능 추가 문서[5]에 보면 "Ref Assemblies" 항목이 있긴 하지만 엄밀히 이것은 C# 언어의 문법과 관련한 기능이 아니다. 단지, C# 컴파일러 옵션에 추가된 것으로써 컴파일 결과물로 "참조 전용의 어셈블리"를 생성할 수 있는 기능을 제공한다. 참조 어셈블리가 어떤 것인지 직접 생성해 보면 더 이해가 빠를 것이므로 다음의 소스코드를 준비하고,

```
using System;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Main1");
        Console.WriteLine("Main2");
    }
}
```

명령행에서 C# 7.1 이상의 컴파일러로 아래와 같이 /refout 옵션을 넣어 빌드한다.

```
csc Program.cs /refout:Program.ref.exe
```

출력 결과로 Program.exe 파일과 함께 Program.ref.exe 파일이 생성되는데 후자의 파일을 닷넷 역컴파일러로 열어 보면 본문의 코드가 모두 없어지고 예외를 던지는 코드 하나로 바뀐 것을 볼 수 있다.

```
internal class Program
{
    // Methods
    public Program()
    {
        throw null;
    }
}
```

결국 "참조 어셈블리"라는 것은 해당 어셈블리가 "메타 데이터"만을 온전하게 포함해 다른 어셈블리들이 참조만 가능케 하는 목적에 한해 쓸 수 있다. 따라서 현실적으로 이 기능은 닷넷 클래스

[5] 이 기능은 비주얼 스튜디오의 프로젝트 속성 창에서 설정할 수 있다.

라이브러리를 대량으로 다루는 마이크로소프트 정도의 규모를 가진 업체에게만 필요하므로 이런 것이 있다는 사실만 알고 넘어가자.

14장: C# 7.2

15장: C# 7.3

15.9 개선된 메서드 선택 규칙 3가지

지난 C# 6.0의 0절 '재정의된 메서드의 선택 정확도를 향상'을 통해 한 차례 개선이 있었지만 이번에도 3가지 범위에 대해 좀 더 정확한 메서드 식별 작업이 추가됐다.

정적/인스턴스 멤버의 호출 문맥 구분

우선, 정적/인스턴스 멤버의 구분이 향상됐다. 예를 들어 다음과 같이 정의했을 때,

```
class StaticButInstanceMatchFirst
{
    public static void Do(object obj)
    {
    }

    public void Do(string txt)
    {
    }
}

static void Main(string[] args)
{
    // C# 7.2까지 컴파일 에러: Error CS0120 An object reference is required for the non-static field, method, or property 'StaticButInstanceMatchFirst.Do(string)'
    StaticButInstanceMatchFirst.Do("TEST"); // 우회적으로 (object)"TEST"로 인자를 전달하면 컴파일 가능
}
```

C# 7.2까지는 Do 메서드 호출에 오류가 발생했다. 왜냐하면 Do 메서드 호출 시 string 타입과 정확히 일치하는 시그니처를 갖는 메서드를 찾으면 더 이상 검색하지 않았기 때문이다. 따라서 "public void Do(string txt)" 메서드를 일치하는 메서드로 판단했고 그에 대해 인스턴스 호출이 아니므로 컴파일 오류가 발생한 것이다.

마찬가지로 이 규칙은 반대의 상황에서도 적용된다.

```
class InstanceButStaticMatchFirst
{
    public static void Do(string txt)
    {
    }

    public void Do(object obj)
    {
    }
}

static void Main(string[] args)
{
    InstanceButStaticMatchFirst t = new InstanceButStaticMatchFirst();

    // C# 7.2까지 컴파일 에러: Error CS0176 Member 'InstanceButStaticMatchFirst.Do(string)' cannot be accessed with an instance reference; qualify it with a type name instead
    t.Do("TEST"); // 우회적으로 (object)"TEST"로 인자를 전달하면 컴파일 가능
}
```

이런 문제점을 해결하기 위해 C# 7.3부터 다음의 규칙이 도입된다.

1) 인스턴스 호출자나 그에 따른 문맥 없이 호출되면 인스턴스 멤버를 제외
2) 인스턴스 호출자로 호출되면 정적 멤버를 제외
3) 호출자가 없는 경우
   A. static 문맥인 경우 static 멤버만 포함
   B. 그 외의 경우 변함없이 모든 멤버 포함
4) 호출자가 모호한 경우 변함없이 모든 멤버 포함

1번 규칙으로 인해 StaticButInstanceMatchFirst 예제의 경우, 호출자가 인스턴스가 아니므로 모든 인스턴스 멤버가 후보에서 제외되므로 정적 Do 메서드 호출이 선택되어 정상적으로 빌드된다. 마찬가지로 InstanceButStaticMatchFirst 예제의 경우, 인스턴스 호출자가 명시됐으므로 정적 멤버가 제외되어 "public void Do(object obj)" 메서드의 호출로 연결된다.

3번 규칙은 다음의 메서드 호출에서 확인할 수 있다.

```
class StaticButInstanceMatchFirst
{
    public static void Do(object obj)
    {
        System.Console.WriteLine("static Do(object)");
    }

    public void Do(string txt)
    {
        System.Console.WriteLine("Do(string)");
```

```
    }

    public static void StaticDone()
    {
        // C# 7.2까지 컴파일 오류 - Error CS0120 An object reference is required for
the non-static field, method, or property 'StaticButInstanceMatchFirst.Do(string)'
        Do("test"); // 호출자 명시가 없고, static 메서드 내에서 호출됨
    }
}
```

호출자 명시가 없지만 "static" 메서드 호출 안에서 Do 메서드가 호출됐으므로 3.A 규칙에 따라 static 멤버만 포함해 결국 "public static void Do(object obj)" 메서드와 연결된다.

반면 다음의 예제는,

```
class InstanceButStaticMatchFirst
{
    public static void Do(string txt)
    {
        System.Console.WriteLine("static Do(string)");
    }

    public void Do(object obj)
    {
        System.Console.WriteLine("Do(object)");
    }

    public void InstanceDone()
    {
        Do("test"); // 호출자 명시가 없고, instance 메서드 내에서 호출됨
    }
}
```

3.B의 규칙이 적용되어 모든 멤버를 대상으로 메서드 시그니처가 가장 일치하는 "public static void Do(string txt)"를 선택하게 된다.

마지막으로 4번 규칙은 1 ~ 3번의 규칙을 세우기 이전에 이미 C# 컴파일러에서 적용하던 규칙[6]이다. 그리고 이 규칙은 C# 7.3에서도 변함이 없고 1 ~ 3번의 규칙에 영향을 받지 않아 기존 소스 코드를 빌드하는 데 문제가 없다.

제네릭의 형식 매개변수 타입 구분

제네릭 메서드의 선택에도 모호한 경우가 있는데 설명을 위해 다음의 코드를 보면,

```
using System;
using System.Linq;
```

[6] C# 3.0 스펙 문서의 7.5.4.1절을 참고

```
using System.Threading.Tasks;

public class AmbiguousMethods
{
    public static GenericResult<T> Do<T>(Func<T> func)
    {
        Console.WriteLine("GenericResult<T> Do");
        T inst = func();
        return new GenericResult<T>(inst);
    }

    public static int Do(Func<int> func)
    {
        Console.WriteLine("int Do");
        return func();
    }
}

public class GenericResult<T>
{
    T _inst;
    public GenericResult(T inst)
    {
        _inst = inst;
    }
}

class Program
{
    static void Main(string[] args)
    {
        AmbiguousMethods.Do(GetInteger); // 출력 결과: int Do
    }

    private static int GetInteger()
    {
        return 5;
    }
}
```

AmbiguousMethods.Do 메서드 호출에서 int를 반환하는 메서드가 찾아지므로 당연히 "public static int Do(Func<int> func)"가 호출된다. 반면 다음과 같은 호출은 어떨까?

```
class Program
{
    static void Main(string[] args)
    {
        // C# 7.2까지 컴파일 오류 - Error CS0121 The call is ambiguous between the following methods or properties: 'AmbiguousMethods.Do<T>(Func<T>)' and 'AmbiguousMethods.Do(Func<GenericResult>)'
        AmbiguousMethods.Do(GetString);
    }

    private static string GetString()
    {
        return "test";
```

```
    }
}
```

C# 7.2까지는 Func<T>와 Func<int>에 대한 형식 매개변수까지 구분하지 않아 2개의 메서드 모두 후보군에 오르게 되어 이런 경우 모호함 오류가 발생한다. 하지만 C# 7.3에서는 이를 개선하기 위해 다음과 같은 규칙을 추가했다.

1) 형식 매개변수를 만족하지 못하는 경우, 후보군에서 제외한다.

이 규칙의 도입으로 위의 소스코드가 정상적으로 컴파일된다. 또한 이로 인해 기존에 문제가 됐던 Task.Run 관련 모호함 문제가 해결된다.

```
class Program
{
    static void Main(string[] args)
    {
        // C# 7.2까지 컴파일 오류 - Error CS0121 The call is ambiguous between the following methods or properties: 'Task.Run<TResult>(Func<TResult>)' and 'Task.Run(Func<Task>)'
        Task.Run(SumOfIntegers);
    }

    private static int SumOfIntegers()
    {
        return Enumerable.Range(1, 1000).Sum();
    }
}
```

delegate 반환 타입 구분

C# 7.2까지 다음과 같은 코드는 모호함 오류가 발생한다.

```
class Program
{
    public delegate int IntStringDelegate(string txt);
    public delegate string StringObjectDelegate(object obj);

    static void Main(string[] args)
    {
        // C# 7.2까지 컴파일 오류 - Error CS0121 The call is ambiguous between the following methods or properties: 'Program.Call(Program.IntStringDelegate)' and 'Program.Call(Program.StringObjectDelegate)'
        Call(RetIntArgObject);

        // C# 7.2까지 컴파일 오류 - Error CS0121 The call is ambiguous between the following methods or properties: 'Program.Call(Program.IntStringDelegate)' and 'Program.Call(Program.StringObjectDelegate)'
        Call(RetStringArgObject);
    }
```

```
    public static void Call(IntStringDelegate func)
    {
        Console.WriteLine("IntStringDelegate");
    }

    public static void Call(StringObjectDelegate func)
    {
        Console.WriteLine("StringObjectDelegate");
    }

    private static int RetIntArgObject(object txt)
    {
        return 5;
    }

    private static string RetStringArgObject(object txt)
    {
        return txt.ToString();
    }
}
```

그 이유는 Call 메서드의 인자로 전달한 메서드의 반환 타입을 구분하지 않아 2개의 후보가 선정됐고, 이로 인해 모호함 오류가 발생하기 때문이다. C# 7.3부터는 다음의 규칙이 추가됐고,

1) delegate의 반환 타입이 일치하지 않으면 후보군에서 제외한다.

따라서 예제 코드의 Call 메서드 모두 성공적으로 단일 메서드를 선택하게 되어 정상적으로 컴파일된다.

16장: C# 8.0

17장: C# 9.0

18장: C# 10

18.8 기타 개선 사항

18.8.1 한정된 할당 분석 개선(Improved Definite Assignment Analysis)

C# 컴파일러는 초기화를 하지 않은 변수에 대해서는,

```
void DoWork()
{
    int i;

    // 컴파일 오류 발생
    // error CS0165: Use of unassigned local variable 'i'
    Console.WriteLine(i);
}
```

일부러 컴파일 오류를 발생시킨다. 사실 이런 경우는 개발자가 실수했을 가능성이 높기 때문에 오류 처리를 통해 인지시키는 것은 컴파일러의 순기능에 해당한다.

하지만 이렇게 좋은 기능이 복잡한 C# 코드를 직면하면 정상적으로 판정하지 못하는 경우가 있다. C# 10부터 이런 문제를 개선했다고 공식 문서에서 밝히고 있지만 현재 비주얼 스튜디오 2022에서 관련 사례에 대한 예제 코드[7]를 빌드해 보면 C# 9 컴파일러에서도 잘 컴파일이 된다. 어차피 이번 개선은 C# 10을 처음부터 사용하게 될 이 책의 독자들에게는 한 번도 겪어본 적이 없을 것이므로 중요치 않다. 단지 이렇게 언급해 두는 이유는 여러분이 C# 공식 문서[8]를 봤을 때 "한정된 할당 분석 개선"이라는 다소 어렵게 느껴지는 제목에 당황하지 않도록 하려는 단순한 의도에서다.

18.8.4 AsyncMethodBuilder 재정의

C# 7.0부터 비동기 메서드의 반환 타입으로 사용자 정의 Task 타입을 사용할 수 있도록 허용했다. 이때 해당 Task 타입을 이용한 비동기 코드를 어떻게 다뤄야 할지 C# 컴파일러에게 알려야 하는데 바로 그 수단으로 AsyncMethodBuilder 특성이 사용된다. 다음 코드는 ValueTask에 정해진 특성을 보여준다.

---

[7] https://www.sysnet.pe.kr/2/0/12793에 예제 코드를 실었다.

[8] https://learn.microsoft.com/ko-kr/dotnet/csharp/language-reference/proposals/csharp-10.0/improved-definite-assignment

```
[AsyncMethodBuilder(typeof(AsyncValueTaskMethodBuilder))]
public readonly struct ValueTask : IEquatable<ValueTask>
{
    // ...[생략]...
}
```

마이크로소프트 측에서 BCL을 통해 제공하는 비동기 메서드의 반환 타입은 크게 Task와 ValueTask 타입이 있고 이에 대해 각각 지정된 비동기 메서드 빌더 타입으로 AsyncTaskMethodBuilder와 AsyncValueTaskMethodBuilder가 있다.

이것은 곧 비동기 메서드의 처리 방법은 비동기 메서드의 반환 타입에 종속돼 있다는 것을 의미한다. 따라서 만약 새로운 비동기 처리 방법을 사용하고 싶다면 그 타입을 지정한 사용자 정의 Task까지 만들어서 제공해야만 한다.

바로 이런 불편함을 C# 10부터는 비동기 메서드의 반환 타입이 아닌, 비동기 메서드에 직접 특성을 지정할 수 있는 방식으로 개선했다. 일례로 마이크로소프트는 .NET 6부터 PoolingAsyncValueTaskMethodBuilder를 제공하고 있는데 사용자 정의 Task 타입을 경유하지 않고 곧바로 다음과 같이 메서드의 특성에 지정해 사용할 수 있다.

```
// C# 9 / .NET 5 이하 컴파일 오류
[AsyncMethodBuilder(typeof(PoolingAsyncValueTaskMethodBuilder<>))]
public static async ValueTask<int> GetSalaryAsync()
{
    await Task.Delay(0);
    return 60;
}
```

아마도 여러분이 새로운 AsyncMethodBuilder에 해당하는 타입을 만들어 사용할 일은 거의 없을 것이다. 단지 그런 타입을 마이크로소프트는 .NET BCL에 추가할 것이고 그것을 AsyncMethodBuilder 특성을 통해 사용할 수 있다는 정도만 이해하고 넘어가자.

18장: C# 11

19장: C# 12

3.부: 닷넷 응용 프로그램

1부와 2부를 통해 C# 언어를 익혔다면 이제 분명 뭔가를 만들고 싶을 것이다. 그런데 과연 어떤 프로그램을 만들 수 있을까? 프로그램의 종류에 따라 만드는 방법이 다르기도 하지만 처음에는 어디서부터 어떻게 시작해야 할지 막막할 수 있다. 그래서 3부에서는 C# 언어로 만들 수 있는 다양한 응용 프로그램 사례를 설명한다. 이를 통해 여러분이 만들고 싶은 프로그램을 어떤 종류의 프로젝트로 시작해야 할지 익힐 수 있을 것이다.

20 장: 프로젝트 유형

C#으로 어떤 종류의 프로그램을 만들 수 있는지 확인하고 싶다면 비주얼 스튜디오에서 "파일" / "새 프로젝트" 메뉴를 선택했을 때 나타나는 내용을 보면 된다. Visual Studio Express 2012를 설치했다면 그림 20.1처럼 나오고, Ultimate 버전 등을 설치했다면 좀 더 다양한 템플릿이 나온다.

그림 20.1 C# 새 프로젝트

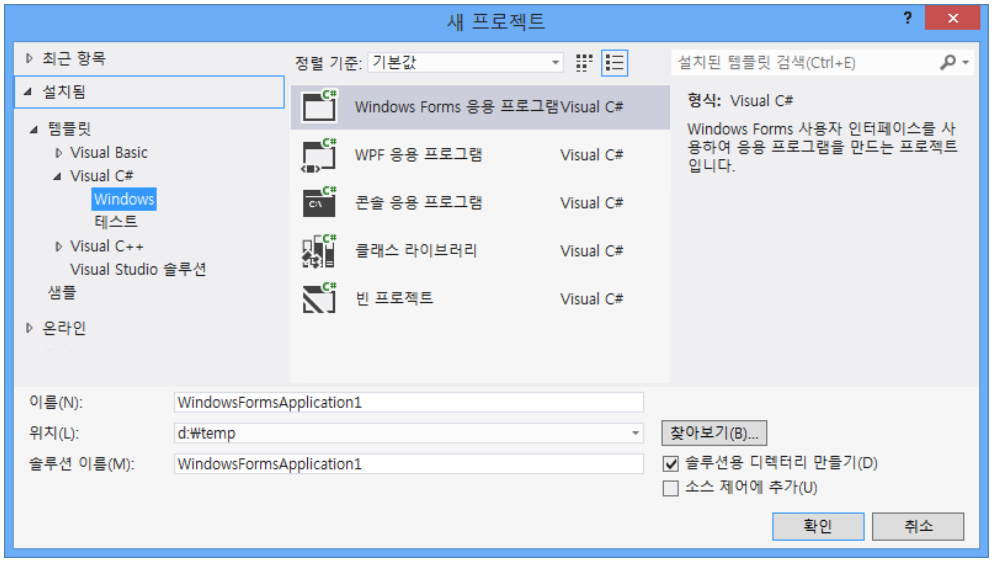


그림 20.1의 프로젝트 템플릿 종류를 정리하면 다음과 같다.

1. Windows Forms 응용 프로그램
2. WPF 응용 프로그램
3. 콘솔 응용 프로그램
4. 클래스 라이브러리
5. 빈 프로젝트

이 가운데 3번의 콘솔 응용 프로그램은 이미 1부와 2부의 예제에서 사용했던 유형이다. 콘솔 프로그램은 대부분 컴퓨터를 잘 다루는 사용자 층을 대상으로 제작되며, 배치 작업(batch process)으로 자동화할 수 있기 때문에 다중 작업을 손쉽게 처리할 수 있다는 장점이 있다. 따라서 서버 측 관리 프로그램을 만들 때 자주 콘솔 응용 프로그램 형식이 사용되지만, 아쉽게도 윈도우 운영체제를 구매한 거의 모든 일반 사용자 층에서는 선호하지 않는 유형이다.

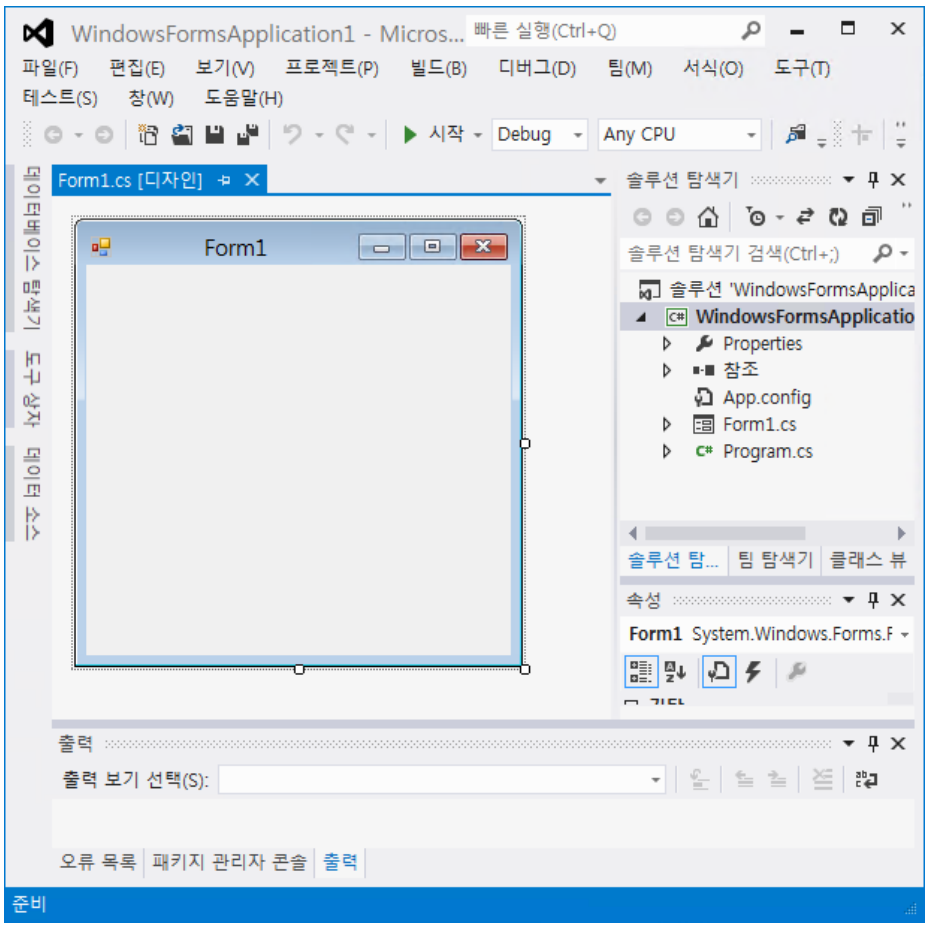
4번의 클래스 라이브러리는 이미 '5.2 프로젝트 구성' 절에서 자세히 다뤘다. 이 장에서는 아직 설명한 적이 없었던 1번 Windows Forms 및 2번의 WPF 응용 프로그램의 개발과 함께 서비스, 웹, 모바일 폰을 위한 프로젝트 유형까지 살펴본다.

20.1 Windows Forms 응용 프로그램

"윈도우 폼 응용 프로그램"은 윈도우 운영체제에서 일상적으로 실행하는 모든 응용 프로그램이 속하는 프로젝트 유형이다. 예를 들어, "윈도우 탐색기"는 콘솔 프로그램이 아니고 "윈도우 프로그램"인데, 이렇게 윈도우(Window)를 가진 응용 프로그램을 만들고 싶다면 C#에서 "윈도우 폼 응용 프로그램" 템플릿을 선택해서 시작하면 된다.

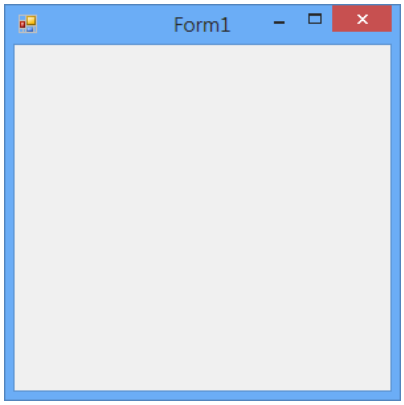
"새 프로젝트" 메뉴를 이용해 윈도우 폼 프로젝트를 만들어 보자. 그럼 비주얼 스튜디오는 C# 프로젝트에 일련의 파일을 추가하고 중앙에는 디자인 화면을 보여준다.

그림 20.2 비주얼 스튜디오의 윈도우 폼 프로젝트



이 상태에서 Ctrl + F5를 눌러 실행해 보면 다음과 같이 윈도우 하나가 실행되는 것을 확인할 수 있다.

그림 20.3 윈도우 폼 프로그램



보다시피 비주얼 스튜디오의 디자인 화면에서 본 윈도우와 동일한 모습으로 실행된다. 어떻게 이것이 가능할 수 있는지 이제부터 그 비밀을 파헤쳐 보자.

우선 그림 20.2의 우측 솔루션 탐색기 영역을 보면 "WindowsFormsApplicaiton1" C# 프로젝트 아래에 콘솔 프로젝트와는 다르게 Form1.cs 파일이 있음을 확인할 수 있다. 그 파일을 마우스로 2번 누르면 나오는 창이 바로 그림 20.2 중앙의 "Form1.cs [디자인]" 창이다. 디자인 창이 아닌 코드 파일을 보고 싶다면 "Form1.cs" 파일을 마우스 우측 버튼으로 누른 다음 "코드 보기(View code)" 메뉴를 선택하거나 단축키 F7을 누르면 된다.

예제 20.1 Form1.cs 기본 코드

```
// ……[using 문 생략]……
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

이것으로 비밀의 절반을 알게 됐다. 윈도우 폼 프로젝트를 실행했을 때 "윈도우"가 나타난 것은 바로 "Form" 클래스를 상속받은 Form1 덕분이다. Form 인스턴스 1개는 윈도우 한 개에 대응한다. 그런데 Form1 클래스를 언제 생성한 것일까? 이를 확인하려면 "Program.cs" 파일을 열어봐야 한다.

```
// ……[using 문 생략]……
using System;
using System.Windows.Forms;
```

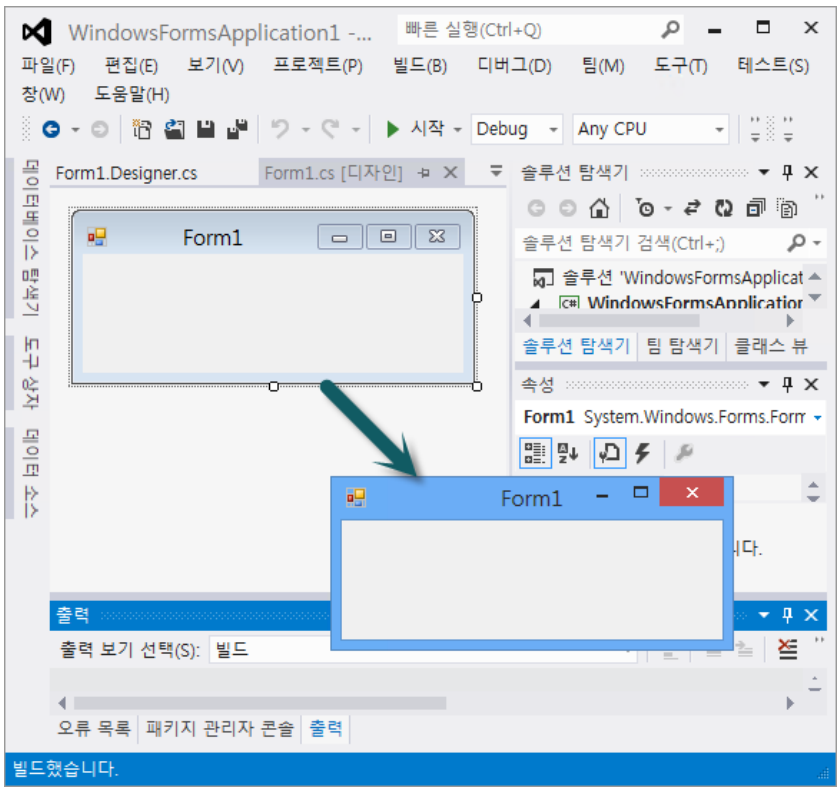
```
namespace WindowsFormsApplication1
{
    static class Program
    {
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

보다시피 Main 메서드가 실행되는 시점에 "new Form1" 코드를 통해 윈도우를 생성하고 있다. Application 타입은 윈도우 폼 프로그램의 기본적인 처리를 담당하는 정적 메서드를 제공하는데, 이 가운데 Run 정적 메서드는 윈도우가 닫히기 전까지 응용 프로그램을 계속 실행해 두는 역할을 한다. 즉, 스레드는 Main 메서드의 Application.Run 내부의 코드를 윈도우가 닫힐 때까지 실행하느라 Main 메서드를 벗어나지 않는다.

Program.cs 파일의 내용은 거의 바뀌지 않기 때문에 Form1 객체가 Run 메서드의 인자로 전달되는 정도만 알아두고 넘어가자. 여기서 새롭게 나온 Application 타입과 Form 타입은 모두 System.Windows.Forms 네임스페이스에 속해 있고, System.Windows.Forms.dll 어셈블리에 구현돼 있다. 여러분이 윈도우 폼 프로그램을 만들기 위해 별도로 어셈블리 참조를 하지 않아도 됐던 것은 "Windows Forms 응용 프로그램" 템플릿에 이미 그런 작업이 돼 있었기 때문이다. 만약 "빈 프로젝트" 템플릿으로 프로젝트를 시작했다면 System.Windows.Forms.dll 어셈블리를 명시적으로 참조해야만 한다.

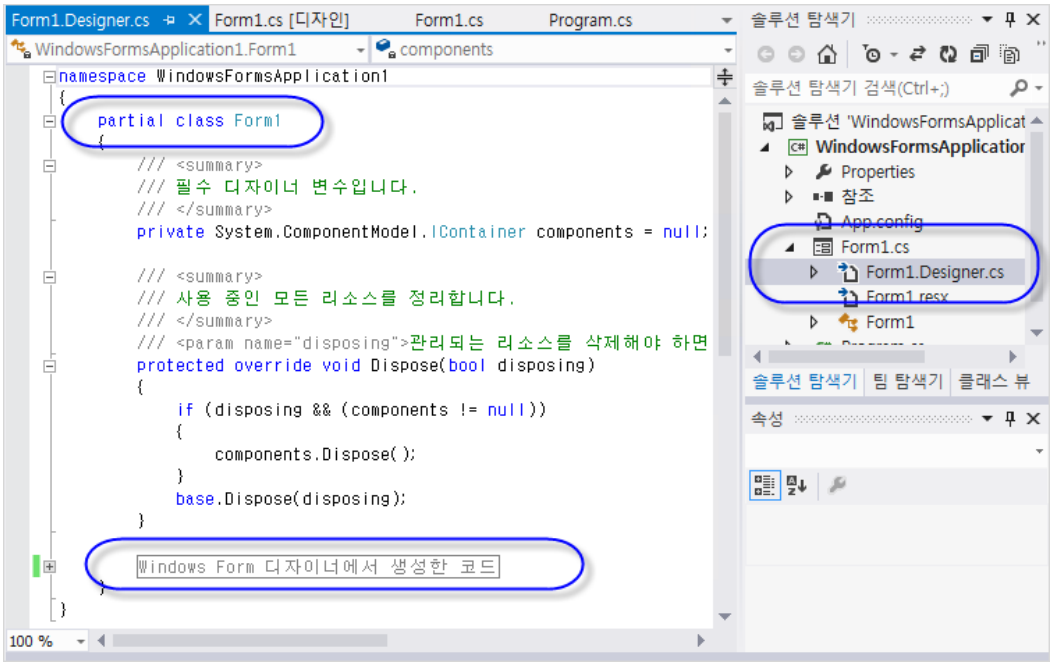
이제 디자인 창에 대해 좀 더 알아보자. Form1 디자인 창에서 마우스를 이용해 윈도우의 크기를 변경하고 다시 Ctrl+F5를 눌러 실행해 본다.

그림 20.4 디자인 창에서 변경한 크기가 반영됨



디자인에서 변경된 창의 크기(Width, Height) 값이 어딘가에 코드로 반영됐을 것이고, 그래서 프로그램을 실행하면 설정된 크기대로 윈도우가 나타나는 것을 짐작할 수 있다. 그런데 프로그램을 종료하고, Form1.cs 코드 파일을 열면(F7) 예제 20.1의 내용과 전혀 다르지 않다. 그럼, 도대체 바뀐 크기 정보는 어디에 저장된 것일까? 이를 알려면 솔루션 탐색기에서 "Form1.cs" 노드를 펼쳐 Form1.Designer.cs 파일을 열어보면 된다.

그림 20.5 Form1.Designer.cs



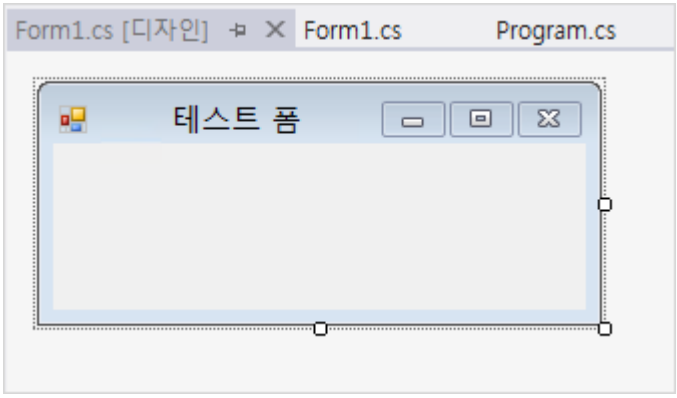
파일명은 Form1.Designer.cs이지만 내부에 정의된 타입명은 Form1으로 예제 20.1의 Form1과 같다. 동일한 이름의 타입이 가능한 이유는 C# 2.0에 소개된 부분 클래스(partial class)가 적용됐기 때문이다. 따라서 C# 컴파일러는 Form1.cs와 Form1.Designer.cs 파일을 컴파일 시에 결합해 단 하나의 Form1 타입을 만든다. 비주얼 스튜디오는 디자인 창에서 변경되는 모든 내용을 그림 20.5의 하단에 보이는 "Windows Form 디자이너에서 생성한 코드" 영역에 반영한다. 그 부분의 코드를 펼쳐 보면 다음과 같이 InitializeComponent 메서드 내에 변경된 창의 크기 정보가 포함된 것을 확인할 수 있다.

```
private void InitializeComponent()
{
    this.SuspendLayout();
    //
    // Form1
    //
    this.AutoScaleDimensions = new System.Drawing.SizeF(7F, 12F);
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.ClientSize = new System.Drawing.Size(266, 83);
    this.Name = "Form1";
    this.Text = "Form1";
    this.ResumeLayout(false);
}
```

비주얼 스튜디오는 디자인 창과 InitializeComponent 메서드의 내용을 동기화한다. 디자인 창에서 내용을 변경하면 InitializeComponent에 반영되고, InitializeComponent의 내용을 변경해도 디자인 창에 반영된다. 예를 들어, this.Text 속성의 값을 "테스트 폼"이라고 변경해서 저장한 후 디자인 창을 보면 그림 20.6과 같이 윈도우 캡션 영역의 문자열이 변경돼 있는 것을

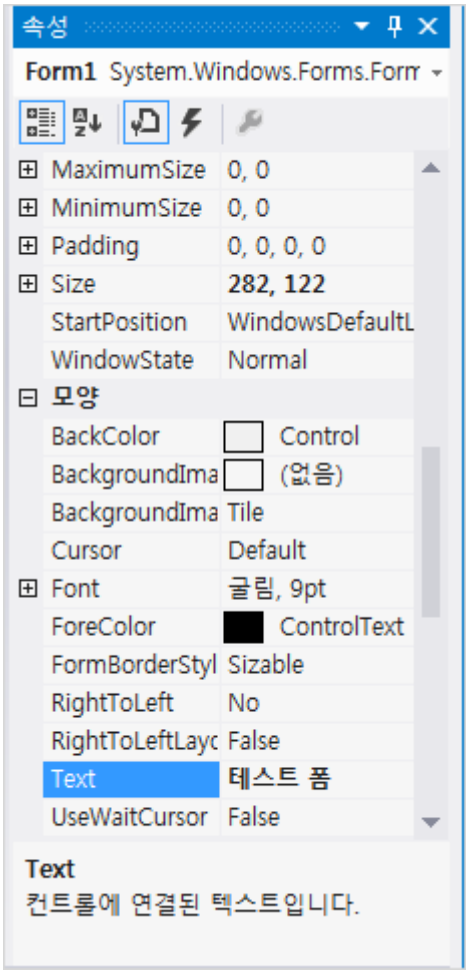
볼 수 있다.

그림 20.6 변경된 윈도우 타이틀



디자인 창에서 직접적으로 변경할 수 있는 것은 윈도우의 크기뿐이다. 그렇다면 그 외의 윈도우 속성값을 변경하려면 Form1.Designer.cs 파일을 열어서 편집해야 할까? 이런 불편함을 덜기 위해 비주얼 스튜디오는 그림 20.2의 우측 하단에 보이는 "속성(Properties)" 창을 별도로 제공한다. Form1 디자인 창에서 윈도우 영역을 마우스로 선택하면 그림 20.7처럼 자동으로 Form 타입의 속성을 속성창에 나열하고 직접 변경할 수 있다.

그림 20.7 디자인 창에 선택된 객체의 속성을 보여주는 속성창



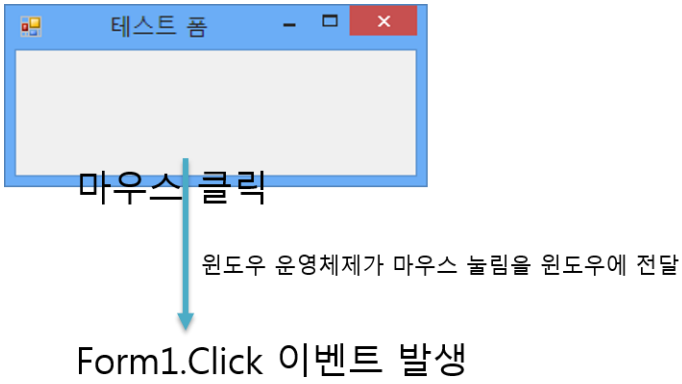
속성창을 통해 Form의 속성값을 자유롭게 바꿀 수 있고 변경사항을 다시 디자인 창을 통해 실시간으로 확인할 수 있다.

최종적으로 윈도우 폼 프로젝트의 동작 과정을 정리해 보자. 프로그램을 실행하면 맨 먼저 Program.cs 파일의 Main 메서드가 실행되고, new Form1 코드를 통해 Form1 타입의 인스턴스가 생성된다. 예제 20.1에 따라 Form1 생성자는 InitializeComponent 메서드를 호출하는데, 이 코드는 Form1.Designer.cs 파일에 비주얼 스튜디오의 디자인 창과 속성창의 내용이 코드로 반영돼 있다. 결국 개발자가 지정한 크기, 타이틀, 색상 등이 설정돼 그림 20.6의 윈도우가 실행된다.

19.1.1 메시지와 이벤트

윈도우 폼 프로그램의 가장 큰 특징은 메시지(message)를 기반으로 동작한다는 점이다. 이는 닷넷이 아닌 윈도우 운영체제 자체의 특징에서 기인한다. 예를 들어, 그림 20.8처럼 실행된 윈도우 폼 위에서 마우스 버튼을 한번 눌러보자. 그럼 내부적으로 윈도우 운영체제는 "마우스 버튼이 눌렸다."라는 메시지를 해당 윈도우 폼에 전달한다. Form 타입은 이러한 메시지가 발생했을 때 C# 클래스의 이벤트(event)를 발생시킨다.

그림 20.8 윈도우 폼 이벤트 발생



따라서 마우스가 눌릴 때 개발자가 원하는 코드를 실행하고 싶다면 Form1 타입의 Click 이벤트에 대응하는 코드를 작성하면 된다.

예제 20.2 마우스 버튼이 눌릴 때 실행되는 코드

```
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();

            this.Click += Form1_Click;
        }

        void Form1_Click(object sender, System.EventArgs e)
        {
            MessageBox.Show("마우스 눌림");
        }
    }
}
```

MessageBox 타입은 System.Windows.Forms 네임스페이스에 정의된 것으로, 사용자에게 간단한 메시지 창을 하나 보여주고 싶을 때 사용한다. 따라서 예제 20.2는 윈도우 위에서 사용자가 마우스 버튼을 누를 때마다 "마우스 눌림"이라는 대화상자가 나타난다.

Form 타입은 Click 이벤트를 제공하고, 개발자는 Click 이벤트가 발생했을 때 처리하는 메서드를 만든다. 이런 메서드의 코드를 이벤트 처리기(event handler)라고 한다. 예전에는 이벤트에 반응하기 위해 매번 메서드를 작성해야 했지만 익명 메서드/람다 식이 지원된 후부터는 다음과 같이 간단한 유형의 이벤트 처리기도 만들 수 있게 됐다.

```
public Form1()
{
    InitializeComponent();

    this.Click += (s, e) =>
        {
            MessageBox.Show("마우스 눌림");
        };
}
```

이벤트에 대응하는 또 다른 방법이 있다. Form 타입은 윈도우 메시지에 대응되는 방법을 이벤트뿐 아니라 메서드 재정의로도 제공한다. 대부분의 유명한 이벤트는 그 동작에 대응되는 가상 메서드(virtual method)를 정의해 뒀기 때문에 Click 이벤트를 사용하지 않고 다음과 같이 처리하는 방법도 가능하다.

```
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        // 일반적으로 이벤트 이름에 "On" 접두사가 붙어 정의된다.
        protected override void OnClick(System.EventArgs e)
        {
            base.OnClick(e);

            MessageBox.Show("마우스 눌림");
        }
    }
}
```

어떤 방식을 사용하느냐는 개발자의 취향 문제일 뿐 크게 중요하지는 않지만 가독성을 위해 한 가지 방법으로 통일하는 것이 바람직하다.

이벤트 처리기는 비주얼 스튜디오의 디자인 창에서도 생성할 수 있다. 그림 20.2에서 디자인 영역에 보이는 윈도우 내부를 마우스로 두 번 연속해서 눌러보자. 그러면 자동으로 비주얼 스튜디오는 Form1_Load 이벤트 처리기를 작성하고 Form1.cs 코드 파일을 보여준다. 다음은 그렇게 생성된 Load 이벤트 처리기에 일부러 MessageBox 코드를 넣어봤다.

```
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
```

```
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, System.EventArgs e)
        {
            MessageBox.Show("폼이 보이기 전입니다.");
        }
    }
}
```

Load 이벤트는 디자인 창에서 생성한 것이므로 "this.Load += Form1_Load" 코드는 InitializeComponent 내부에 생성돼 있다. 여기서 한가지 알 수 있는 사실은 이벤트의 종류가 많다는 점이다. 윈도우의 상태 변화, 마우스 이동/눌림, 키보드 눌림 등의 모든 상황에 대해 윈도우는 메시지를 통해 Form에게 알려주고 이를 이벤트로 노출시킨다. 표 20.1에 Form 타입에 정의된 25개의 이벤트 중 일부가 정리돼 있다.

표 20.1 Form 타입의 이벤트

이벤트	설명
FormClosed	폼이 닫히고 나서 발생하는 이벤트
FormClosing	폼이 닫히기 바로 전 발생하는 이벤트
Load	폼이 보이기 바로 전 발생하는 이벤트, 일반적으로 Form에서 사용될 객체들의 초기화를 Load 이벤트 처리기에 작성함.
ResizeBegin	폼의 크기가 변경되기 시작했음을 알리는 이벤트
ResizeEnd	폼의 크기가 변경됐음을 알리는 이벤트
Shown	폼이 화면에 처음 보일 때 발생하는 이벤트
SizeChanged	폼의 크기가 변경되는 동안 매번 발생하고 최소화/최대화시킬 때도 발생

물론 Form 타입의 상속 관계를 고려했을 때 더 많은 이벤트가 있다.

Form ➔ ContainerControl ➔ ScrollableControl ➔ Control 타입 ➔ Component 타입

ContainerControl에 정의된 이벤트 1개
ScrollableControl에 정의된 이벤트 1개
Control에 정의된 이벤트 68개
Component에 정의된 이벤트 1개

이 중에서 Control에 정의된 이벤트 중 자주 사용되는 것을 표 20.2에 정리했다.

표 20.2 Control 타입의 이벤트

이벤트	설명
BackColorChanged	컨트롤의 배경색이 바뀔 때 발생하는 이벤트
Click	컨트롤 위에서 클릭 동작이 발생했을 때
DoubleClick	컨트롤 위에서 더블 클릭 동작이 발생했을 때
DragEnter	마우스로 끌기 시작한 객체가 컨트롤에 진입했을 때
DragLeave	마우스로 끌기 시작한 객체가 컨트롤의 경계를 벗어났을 때
DragDrop	마우스의 끌어서 놓기 작업이 완료됐을 때
DragOver	마우스로 끌기 시작한 객체가 컨트롤 위에서 머무르는 동안 발생
KeyDown	키보드의 키가 눌렸을 때
KeyUp	키보드의 키가 눌렸다가 떼어졌을 때
KeyPress	키보드의 키 입력이 발생한 경우로서, 한번의 KeyDown + KeyUp 조합이 있을 때마다 함께 발생하는 이벤트
LocationChanged	컨트롤의 위치가 변경됐을 때
MouseClick	컨트롤 위에서 마우스 클릭이 발생했을 때, Click 이벤트보다 더 빨리 발생하며 오직 마우스 호환 장치에 의해서만 클릭이 됐을 때 발생
MouseDoubleClick	컨트롤 위에서 마우스 호환 장치에 의해 더블 클릭이 발생했을 때
MouseDown	마우스의 버튼이 눌렸을 때
MouseUp	마우스의 버튼이 눌렸다 떼어졌을 때
MouseEnter	컨트롤 위에 마우스가 진입했을 때
MouseHover	컨트롤 위에 마우스가 머무르는 동안 발생
MouseLeave	컨트롤의 경계를 마우스가 벗어났을 때
MouseMove	컨트롤 위에서 마우스를 이동하는 동안 발생
MouseWheel	마우스의 휠(Wheel)을 조작했을 때
Move	컨트롤의 위치가 이동하는 동안 발생
Paint	컨트롤의 내부를 새로 그릴 때 발생
Resize	컨트롤의 크기가 변경되는 동안 발생
SizeChanged	컨트롤의 크기가 변경이 완료됐을 때 발생
TextChanged	컨트롤의 타이틀 내용이 변경됐을 때 발생

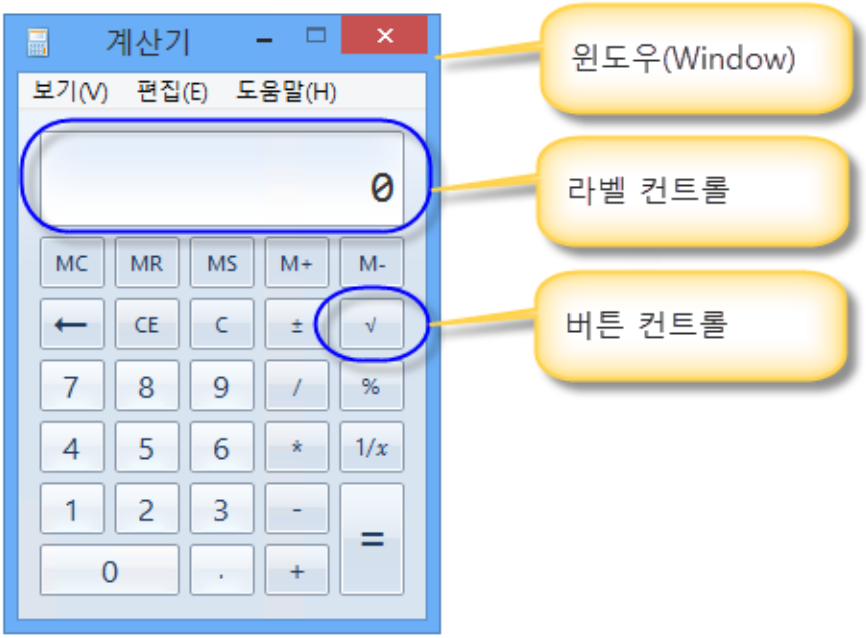
윈도우 폼 응용 프로그램을 잘 만들고 싶다면 어떤 종류의 이벤트가 지원되는지 정도는 틈틈이 익혀두는 것이 좋다.

19.1.2 컨트롤

윈도우 프로그램의 대표적인 구성 요소는 "폼(Form)"과 "컨트롤(Control)"이 있다. 폼은 윈도우에 대응되고, 컨트롤은 폼 내부에 포함된 "자식 윈도우"에 해당한다. 예를 들어, 그림 20.9의 계산기 프로그램을 보면 단일 윈도우 하나가 폼이고, 그 내부에 포함된 버튼 등의 구성 요소가 컨트

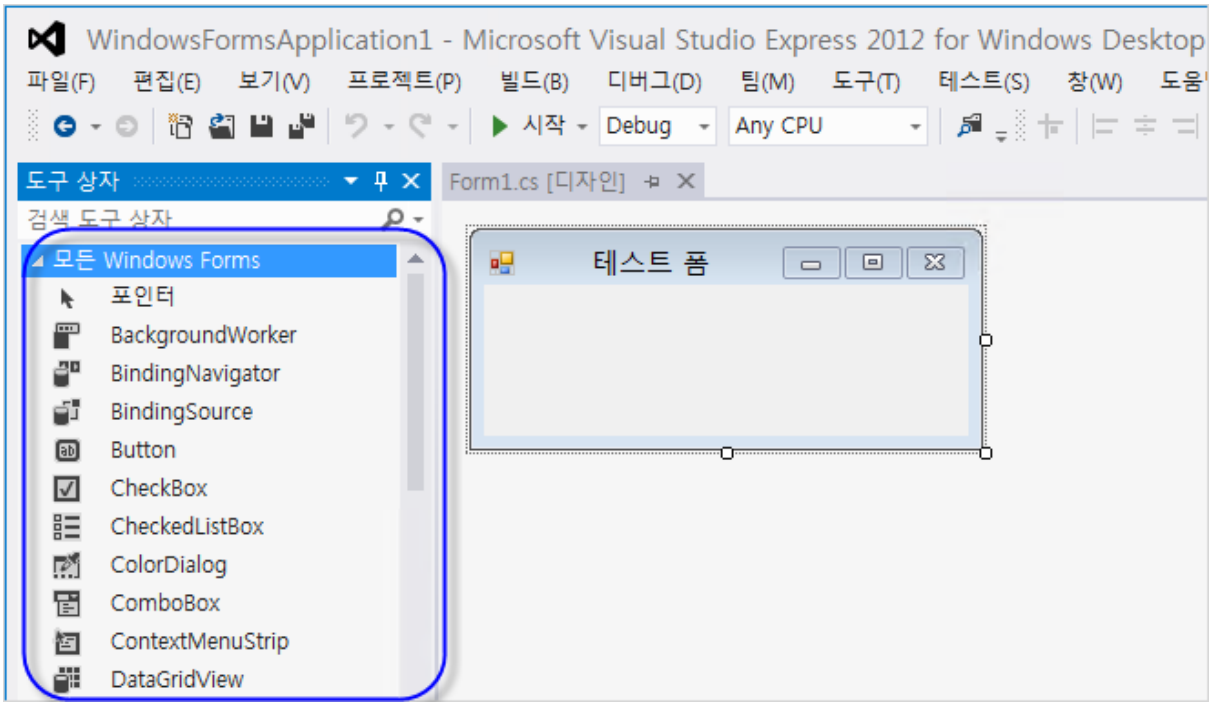
롤이다.

그림 20.9 윈도우와 컨트롤



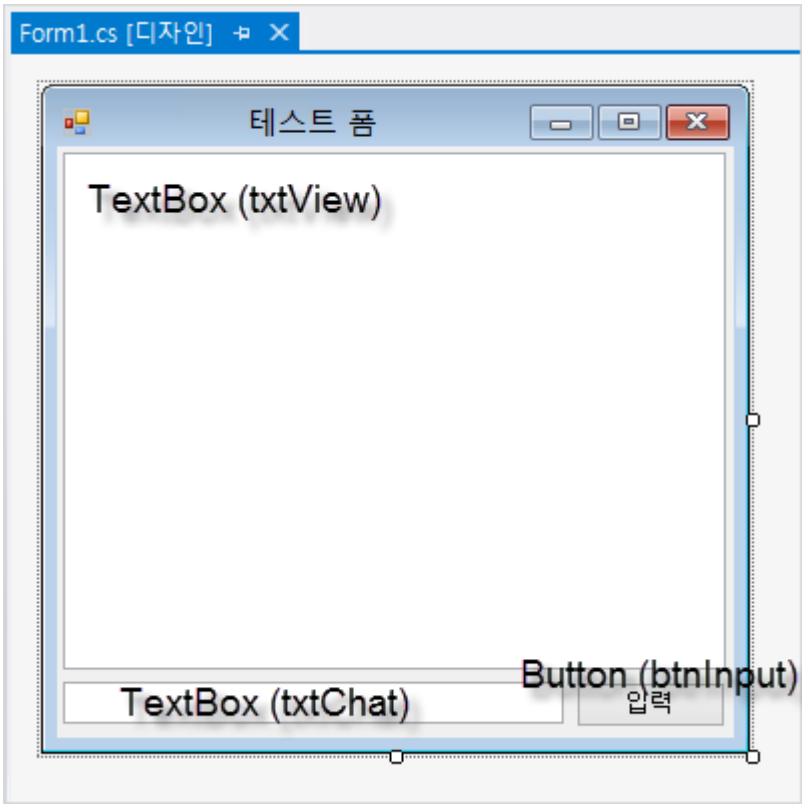
기본적인 용도의 컨트롤은 마이크로소프트가 이미 만들어 뒀으며, 비주얼 스튜디오의 디자인 창에서 "보기(VIEW)" / "도구 상자(Toolbox)" (단축키: Ctrl+ W, X)를 통해 확인할 수 있다. (참조: 그림 20.10)

그림 20.10 윈도우 폼에서 사용 가능한 컨트롤



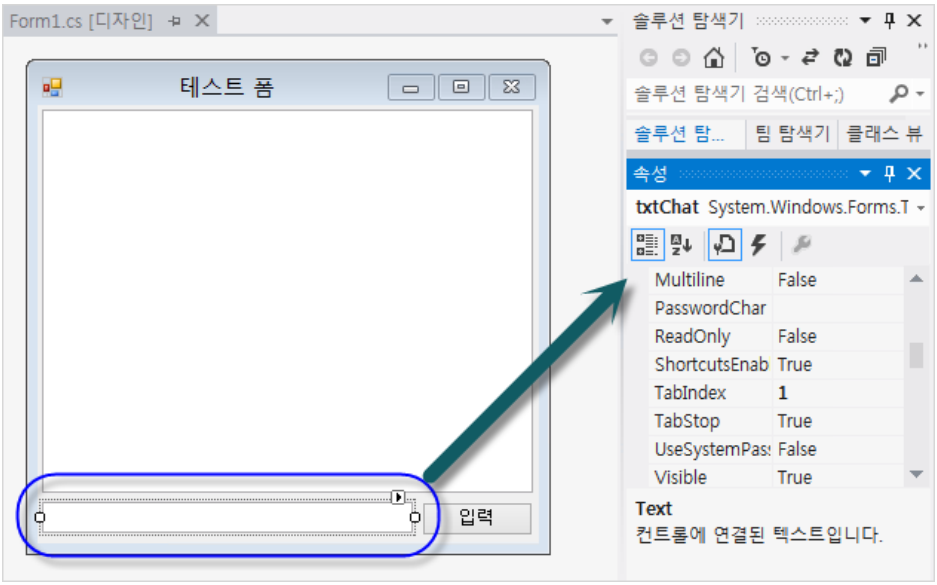
사용법은 간단하다. 도구 상자에 있는 특정 항목을 마우스로 끌어다 디자인 창의 윈도우 폼 위에 놓으면 된다. 예를 들기 위해 통신 기능은 없는 채팅 프로그램의 외관을 만들어 보자. 필자의 경우 다음과 같이 3개의 컨트롤을 장식해 놓았다.

그림 20.11 채팅 프로그램의 디자인



디자인 화면에서 각 컨트롤을 선택하면 그에 따른 설정을 속성 창에서 할 수 있다. 그림 20.12에서는 디자인 창에서 TextBox 컨트롤을 선택했을 때 그것의 속성을 나열하는 "속성 창"을 보여준다.

그림 20.12 선택된 컨트롤의 속성 창 정보



속성 창을 이용해 각 컨트롤에서 필자가 변경한 속성을 표 20.3에 정리했다.

표 20.3 채팅 윈도우의 컨트롤 속성

컨트롤	변경된 속성	값
TextBox	(Name)	txtView
	Multiline	True
TextBox	(Name)	txtChat
Button	(Name)	btnInput
	Text	입력

표 20.3의 (Name) 항목은 InitializeComponent 메서드 내에 다음과 같은 식으로 반영된다.

```
private System.Windows.Forms.TextBox txtView;
private System.Windows.Forms.TextBox txtChat;
private System.Windows.Forms.Button btnInput;
```

즉, Form1.cs 코드 파일에서 (Name)으로 변경된 값을 변수로 해서 해당 컨트롤을 접근할 수 있게 된다. 비록 통신이 안 되는 채팅 프로그램에 불과하지만 실행했을 때 txtChat 영역에 글을 입력하고 "입력" 버튼을 눌렀을 때 txtView 영역에 글이 출력되게 할 수는 있다.

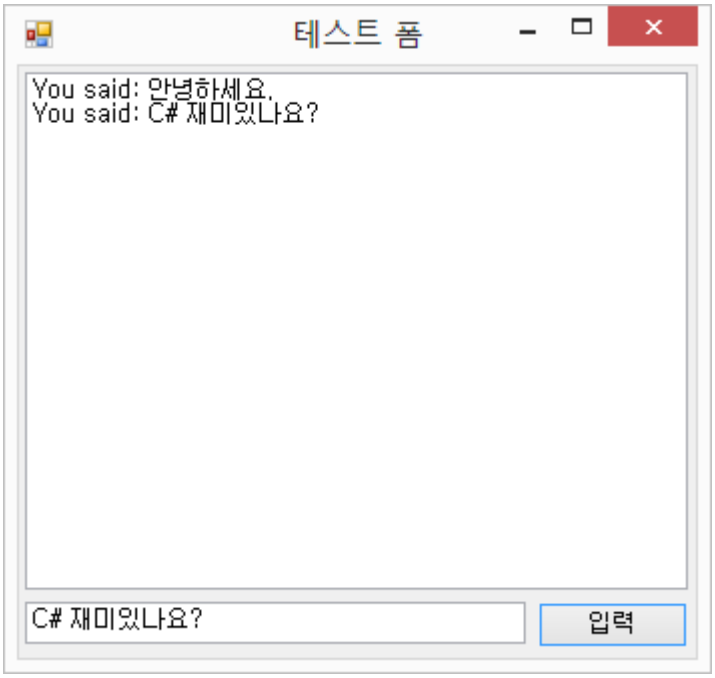
이를 위해 반응해야 하는 이벤트는 "입력" 버튼의 Click 이벤트다. 따라서 디자인 창에서 "입력" 버튼을 마우스로 두 번 클릭하면 자동으로 btnInput_Click 이벤트 처리기가 생성되고, 내부에 예제 20.3처럼 작성할 수 있다.

예제 20.3 btnInput 버튼의 Click 이벤트 처리기

```
// txtView로 txtChat에 입력된 문자열을 추가
private void btnInput_Click(object sender, EventArgs e)
{
    string oldText = txtView.Text;

    string newText = string.Format("You said: {0}{1}", txtChat.Text,
        Environment.NewLine);
    txtView.Text = oldText + newText;
}
```

TextBox 컨트롤은 문자열을 설정하거나 가져올 수 있는 Text 공용 속성을 제공한다. 이를 이용하면 사용자가 입력한 txtChat 컨트롤의 입력 내용을 btnInput 버튼이 눌린 시점에 가져와서 txtView 컨트롤에 이어 붙일 수 있다. 이제 실행해 보고 결과를 확인하자.

그림 20.13 실행 중인 예제 프로그램

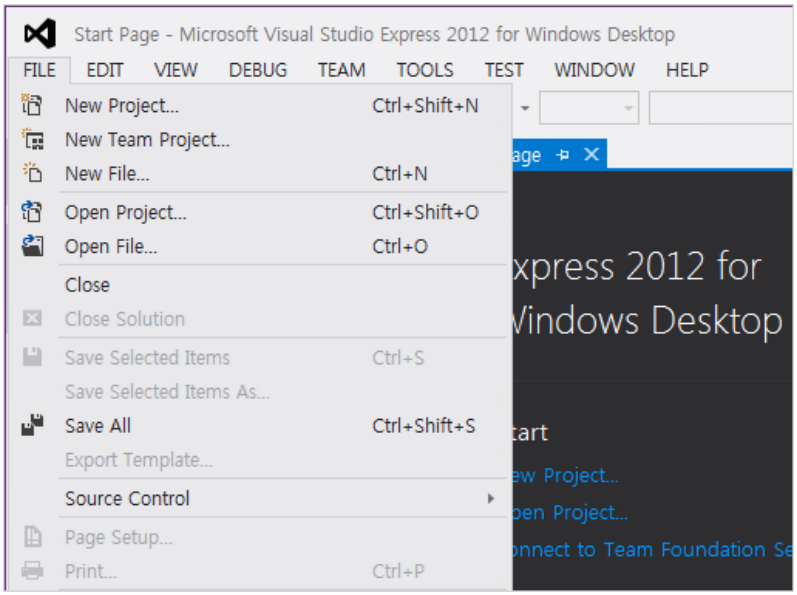


이처럼, 윈도우 폼 프로그램의 기본은 다양한 컨트롤의 사용법을 익히는 것과 관련된다. 지면 관계상 다른 컨트롤의 사용법을 모두 설명할 수는 없지만 시간 날 때마다 비주얼 스튜디오의 도구 상자에 있는 컨트롤을 하나씩 익히기를 권장한다.

19.1.3 메뉴

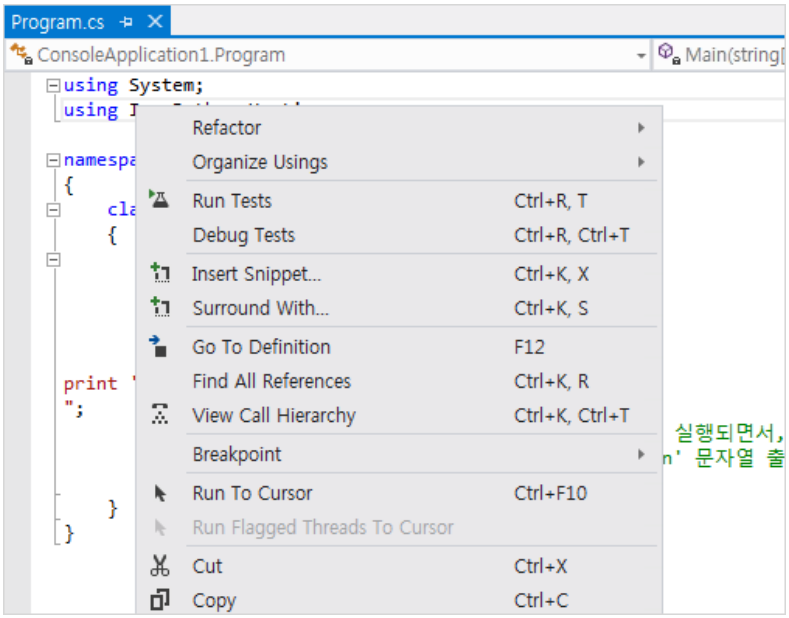
거의 모든 윈도우 폼 프로그램은 기본적으로 메뉴(Menu)를 가지고 있다. 여러분이 이 책의 실습을 위해 실행한 비주얼 스튜디오도 그림 20.14처럼 "FILE", "EDIT", ... 등의 메뉴를 제공하고, 마우스로 메뉴를 선택하면 아래로 펼쳐져 서브 메뉴(Sub Menu)가 나타난다. 이렇게 아래로 펼쳐진다고 해서 흔히 풀-다운 메뉴(pull-down menu)라고도 한다.

그림 20.14 비주얼 스튜디오의 펼쳐진 파일(FILE) 메뉴



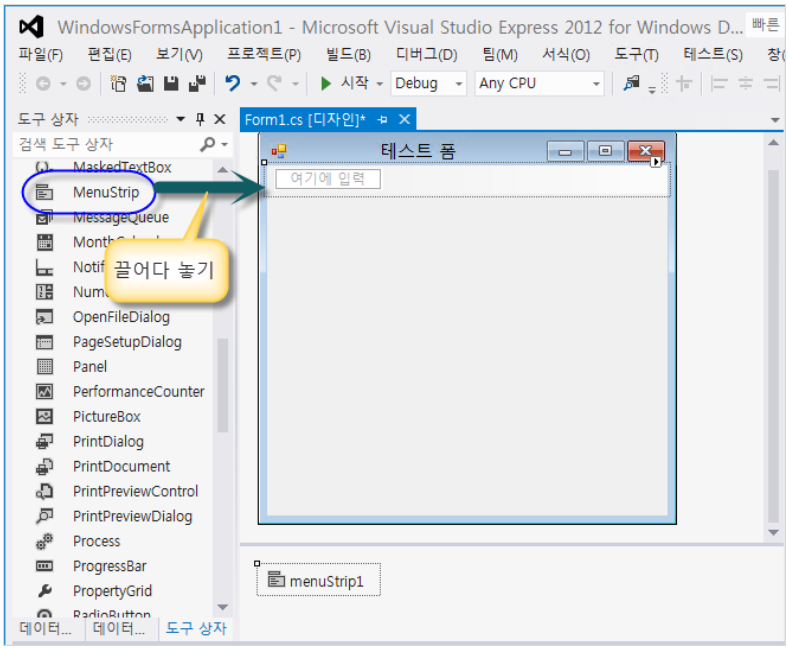
이 외에도 마우스 오른쪽 버튼을 이용해 메뉴를 나타나게 하는 것도 있다. 이런 방식의 메뉴는 원하는 어떤 위치에서든 서브 메뉴를 펼칠 수가 있는데, 일반적으로는 상황(context)에 따라 보여준다고 해서 컨텍스트 메뉴(context menu)라고 한다. 비주얼 스튜디오의 경우 그림 20.15처럼 코드 편집 창에서 마우스 오른쪽 버튼을 눌렀을 때 나오는 메뉴가 바로 이런 유형에 속한다.

그림 20.15 비주얼 스튜디오 편집 창의 컨텍스트 메뉴



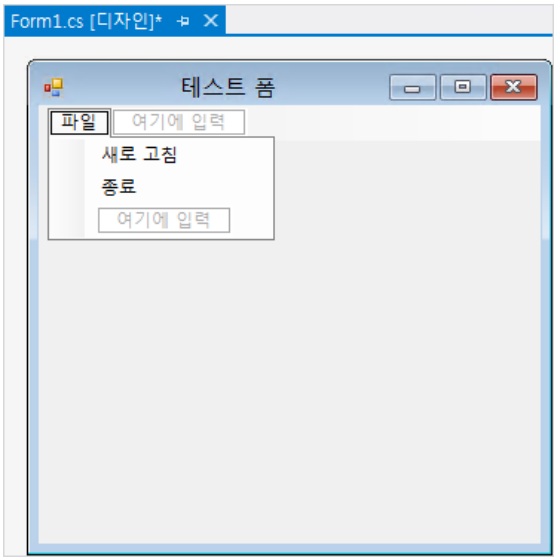
이 중에서 우선 가장 쉬운 풀다운 메뉴를 먼저 구현해 보자. 비주얼 스튜디오의 윈폼 디자인 창을 열고 도구 상자(Toolbox)의 "MenuStrip"을 끌어다 놓으면 그림 20.16처럼 "여기에 입력(Type Here)"이라는 바(Bar) 영역이 생긴다.

그림 20.16 윈폼 디자인 화면에 MenuStrip 배치



마우스로 "여기에 입력" 영역을 누르면 텍스트를 입력할 수 있게 바뀌는데, 간단한 실습을 위해 그림 20.17과 같이 구성한다(물론 원한다면 메뉴를 자유롭게 더 추가해도 된다.)

그림 20.17 "파일" 메뉴와 서브 메뉴 2개 구성

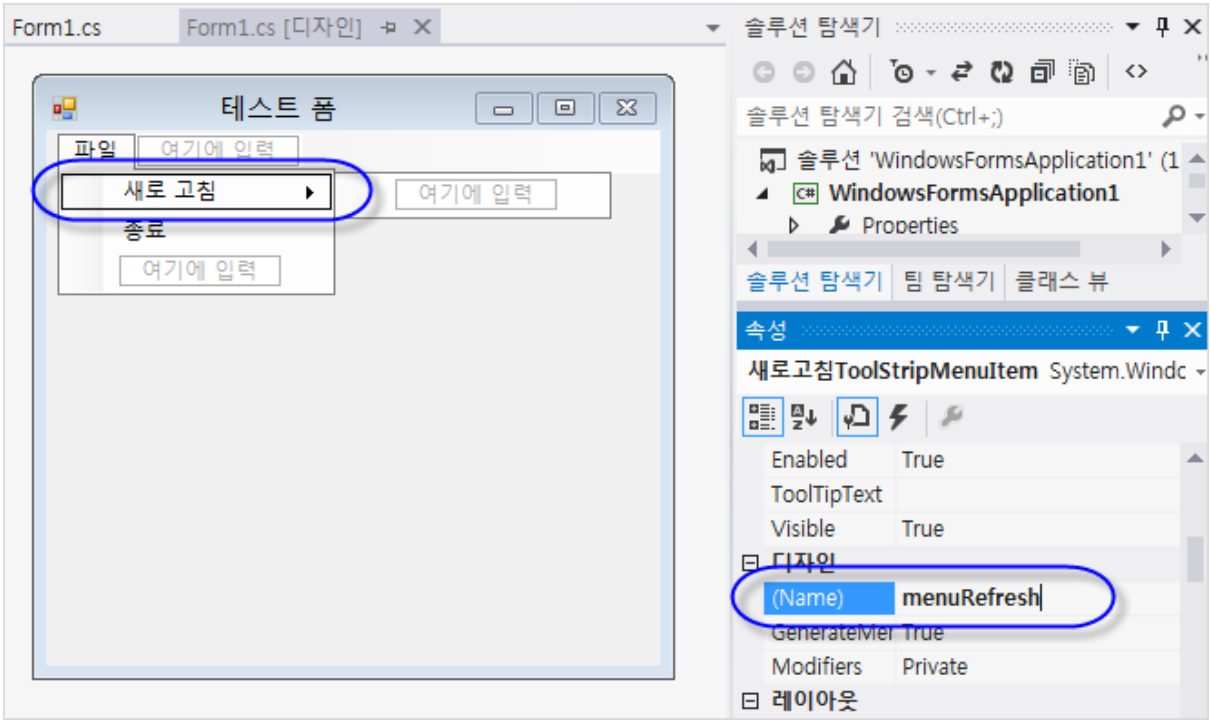


이 상태에서 Ctrl+F5를 눌러 프로젝트를 실행해 보면 그림 20.17에서 구성한 메뉴가 보이는 것을 확인할 수 있다. 하지만 "새로 고침", "종료" 메뉴를 선택해도 현재는 아무런 반응도 하지 않는다. 왜냐하면 해당 메뉴를 선택했을 때 실행돼야 할 코드가 지정되지 않았기 때문이다.

실행을 종료하고, 그림 20.18의 화면과 같이 비주얼 스튜디오의 디자인 창으로 돌아가서 "새로 고침" 메뉴를 선택하면 속성 창에서 해당 메뉴의 (Name)을 바꿀 수 있다. 그림 20.18의 경우

"menuRefresh"로 설정했다.

그림 20.18 "새로 고침" 메뉴의 (Name) 항목 편집



마찬가지 방법으로 "종료" 메뉴에 대해서도 (Name) 값을 "menuExit"로 설정한 후 "F7" 키를 눌러 코드 편집 창으로 넘어가서 각각의 메뉴에 대해 예제 20.4와 같이 이벤트 처리기를 연결해 준다.

예제 20.4:이벤트 처리기가 연결된 메뉴

```
using System;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();

            this.menuRefresh.Click += menuRefresh_Click;
            this.menuExit.Click += menuExit_Click;
        }

        void menuRefresh_Click(object sender, EventArgs e)
        {
            MessageBox.Show("새로 고침 버튼 눌림");
        }

        void menuExit_Click(object sender, EventArgs e)
```

```
        {
            Application.Exit(); // 윈폼 프로그램을 종료하는 메서드
        }
    }
}
```

이제 프로그램을 실행하고 "새로 고침", "종료" 버튼을 눌러보면 각각 menuRefresh_Click, menuExit_Click 이벤트 처리기의 코드가 실행되는 것을 확인할 수 있다. 이것이 풀다운 메뉴의 기본적인 사용법이다.

다음으로, 컨텍스트 메뉴의 사용법을 알아보자. 컨텍스트 메뉴는 그 특성상 마우스 오른쪽 버튼이 눌린 위치에서 펼쳐지기 때문에 기본적으로 마우스 이벤트 처리기를 하나 작성해야 한다. 그리고 내부에는 ContextMenu 타입을 이용해 각 메뉴 항목을 MenuItem 타입으로 생성해 컨텍스트 메뉴로 펼쳐질 내용을 구성하면 된다.

```
using System;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        // 마우스 우측 버튼이 눌린 경우 실행되는 Form 재정의 메서드
        protected override void OnMouseClick(MouseEventArgs e)
        {
            base.OnMouseClick(e);

            // 마우스 우측 버튼이 눌린 경우
            if (e.Button == System.Windows.Forms.MouseButtons.Right)
            {
                // 컨텍스트 메뉴 객체를 만들고
                ContextMenu ctxMenu = new ContextMenu();

                // 컨텍스트 메뉴에 들어갈 2개의 MenuItem을 생성해서 추가
                MenuItem menuItem = new MenuItem("새로 고침");
                menuItem.Click += menuRefresh_Click;
                ctxMenu.MenuItems.Add(menuItem);

                menuItem = new MenuItem("종료");
                menuItem.Click += menuExit_Click;
                ctxMenu.MenuItems.Add(menuItem);

                // 최종 구성된 컨텍스트 메뉴를 마우스가 눌린 위치에서 팝업
                ctxMenu.Show(this, e.Location);
            }
```

```
        }

        void menuRefresh_Click(object sender, EventArgs e)
        {
            MessageBox.Show("새로 고침 버튼 눌림");
        }

        void menuExit_Click(object sender, EventArgs e)
        {
            Application.Exit();
        }
    }
}
```

이 코드를 실행하면 그림 20.19와 같이 윈도우 폼 내부의 아무 위치에서나 마우스 오른쪽 버튼을 누르면 컨텍스트 메뉴가 나타나고 각 메뉴를 선택하면 연결된 이벤트 처리기의 코드가 실행되는 것을 확인할 수 있다.

그림 20.19 컨텍스트 메뉴 예제 실행

19.1.4 Graphics

윈도우는 내부에 그래픽을 출력할 수 있는데, 이에 대한 명령어가 Graphics 타입에 제공된다. 예제 20.5는 간단하게 윈도우 내부에 선을 그리는 코드를 보여준다.

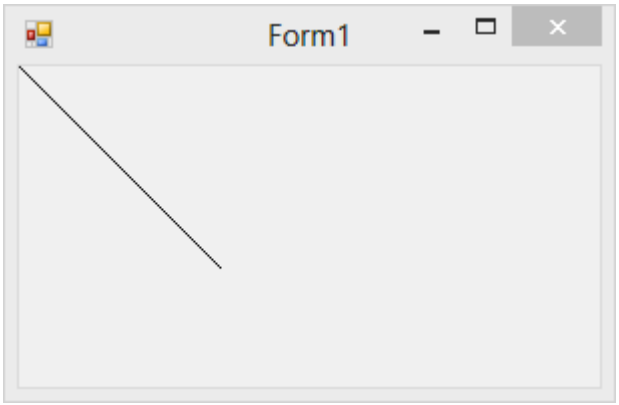
예제 20.5 폼에 (0,0) – (100,100) 좌표로 선을 그리는 방법

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace WindowsFormsApplication1
```

```
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        protected override void OnShown(EventArgs e)
        {
            base.OnShown(e);

            using (Graphics g = this.CreateGraphics())
            {
                g.DrawLine(Pens.Black, 0, 0, 100, 100);
            }
        }
    }
}
```

프로그램을 실행하면 그림 20.20과 같은 결과를 확인할 수 있다.

그림 20.20 검은 색 선이 그려진 윈도우



DrawLine 메서드의 첫 번째 인자로 Pen 타입의 인스턴스가 전달되는데, Pens 타입에는 대표적으로 141개에 해당하는 색상을 가진 Pen 인스턴스가 미리 만들어져 있으므로 원하는 색상이 존재한다면 그대로 가져다 써도 된다. 물론 직접 생성하는 것도 가능하다.

```
using (Graphics g = this.CreateGraphics())
{
    using (Pen bluePen = new Pen(Brushes.Blue))
    {
        g.DrawLine(bluePen, 0, 0, 100, 100); // 파란 색 선이 그어진다.
    }
}
```

Pen 생성자의 첫 번째 인자는 색상을 나타내는 Brush 타입의 인스턴스를 받는 것 말고도 펜의

굵기도 지정할 수 있다.

```
using (Pen bluePen = new Pen(Brushes.Blue, 10))
{
    g.DrawLine(bluePen, 0, 0, 100, 100); // 10픽셀의 굵기로 파란 색 선이 그어진다.
}
```

그림 20.21 굵기가 지정된 선

new Pen(……) 생성자의 첫 번째 인자인 Brush 타입을 다시 확인해 보면, 이것 역시 141개의 브러시가 Brushes 타입에 미리 만들어져 있기 때문에 가져다 재사용할 수 있고, 원한다면 객체를 직접 생성하는 것도 가능하다.

```
using (Graphics g = this.CreateGraphics())
{
    using (Brush brush = new SolidBrush(Color.Brown))
    using (Pen pen = new Pen(brush))
    {
        g.DrawLine(pen, 0, 0, 100, 100);
    }
}
```

SolidBrush 타입은 단일 색상을 갖는 브러시를 의미하고, 생성자에서는 Color 타입의 인스턴스를 받는데, 대표적인 141개의 색상을 정적 속성으로 제공하므로 대부분의 경우 그대로 재사용하면 된다. 원하는 색상이 없는 경우 당연히 직접 만들어 쓸 수 있다.

```
using (Graphics g = this.CreateGraphics())
{
    Color color = Color.FromArgb(0x0F, 0xF0, 0x10);
    using (Brush brush = new SolidBrush(color))
    using (Pen pen = new Pen(brush))
    {
        g.DrawLine(pen, 0, 0, 100, 100);
    }
}
```

Color 타입의 FromArgb 정적 메서드는 빛의 3원색(빨강, 초록, 파랑)에 해당하는 값을 차례대로 0 ~ 255 범위로 받아 색상을 표현한다. 빛의 3원색이므로 (Red, Green, Blue) 값이 각각 (0,0,0)이면 검은색을, (255, 255, 255)이면 흰색을 나타낸다. 또한 Argb의 "A"는 Alpha 값으로 투명도를 의미하는데, 기본값인 255는 불투명을 나타내고, 값이 작아지면서 점차 투명해지다가 0을 지정하면 완전히 투명해져서 아예 화면에 나타나지 않는다. 따라서 반투명 효과를 내고 싶다면 알파 값을 절반으로 주면 된다.

```
Color color = Color.FromArgb(0xFF / 2, 0x0F, 0xF0, 0x10);
```

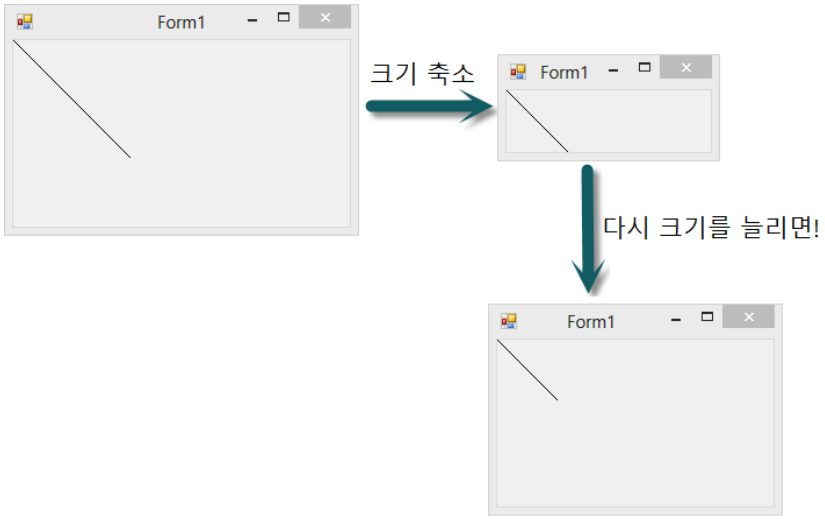
Pen, Brush, Color 타입은 Graphics 타입에서 제공되는 그 외의 모든 그리기 메서드에 동일한 규칙으로 적용된다. 참고로 표 20.4에 대표적인 그리기 메서드를 정리했으니 참고하자.

표 20.4 Graphics 타입에 제공되는 그리기 메서드

메서드	설명
DrawArc	원 호를 그린다.
DrawBezier	베지어(Bezier) 곡선을 그린다.
DrawCurve	카디널 스플라인(Cardinal Spline)을 그린다.
DrawEllipse	타원을 그린다.
DrawImage	Image 객체에 담긴 그림을 출력한다.
DrawLine	선을 그린다.
DrawPie	부채꼴 모양의 원호를 그린다.
DrawRectangle	사각형을 그린다.
DrawString	인자로 전달된 문자열을 그린다.

윈도우에 그리는 방법까지 알았으니 이제는 그려진 그림을 유지하는 방법을 알아볼 차례다. 이해를 돕기 위해 예제 20.5를 실행하고 그림 20.22와 같이 윈도우의 크기를 줄였다가 다시 늘여보자.

그림 20.22 줄어든 영역의 라인이 사라지는 현상



최초 윈도우에서는 (0,0) – (100,100)까지 그려졌던 선이 크기가 축소된 영역만큼 없어진 채로 남아 있는 것을 확인할 수 있다.

이것은 윈도우 운영체제가 실행돼 있는 프로그램들의 윈도우 화면을 관리하는 방식에 따라 나타나는 현상이다. 윈도우 운영체제는 윈도우 영역에 그려진 정보를 보관하지 않으며, 이에 대한 책임을 각 윈도우에 맡긴다. 그렇다면 개발자는 사라진 그림 영역을 언제 다시 그려야 할지를 알 수 있을까? 이를 위해 윈도우 운영체제는 윈도우 화면이 무효화(Invalidate)됐을 때 WM_PAINT라는 윈도우 메시지를 이용해 각 윈도우에 그 사실을 통보한다. 닷넷의 윈도우 폼 프로그램에서는 WM_PAINT 메시지가 발생할 때마다 OnPaint 메서드를 실행하는 것으로 이를 처리한다. 따라서 예제 20.5를 윈도우 크기가 변하는 것에 상관없이 라인을 그대로 유지하고 싶다면 코드를 OnPaint 메서드 내부로 옮겨야 한다.

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        protected override void OnPaint(PaintEventArgs e)
        {
            base.OnPaint(e);

            using (Pen pen = new Pen(Brushes.Black))
```

```
            {
                e.Graphics.DrawLine(pen, 0, 0, 100, 100);
            }
        }
    }
}
```

정리하자면, OnPaint 메서드는 내부를 다시 그려야 할 필요가 있을 때마다 자동으로 다시 호출된다. 또한 그 인자로 PaintEventArgs 타입을 받으며, 여기에는 Graphics 속성이 제공되기 때문에 별도로 CreateGraphics 메서드를 이용해 생성하지 않아도 된다.

19.1.5 폼과 대화상자

엄밀히 말해서 폼(Form)과 대화상자(Dialog)는 윈도우 폼 응용 프로그램에서 Form 타입으로 동일하게 구현된다. 단지 Form을 어떻게 사용하느냐에 따라 폼(Form)이 될 수도 있고, 대화상자(Dialog)가 될 수도 있다.

기본 윈도우 폼 프로젝트를 생성한 상태에서 솔루션 탐색기의 프로젝트 항목을 마우스 오른쪽 버튼으로 누른 다음 "추가(Add)" / "새 항목(New Item)"을 차례로 선택한 후 "Windows Forms" 범주에서 "Windows Form" 항목으로 "SubForm.cs" 파일을 지정해서 추가한다. 간단하게 사용자에게서 이름을 입력받을 수 있는 Label 및 TextBox와 두 개의 Button을 폼 위에 올려 놓고 각 속성을 설정한다(그림 20.23과 표 20.5 참고).

그림 20.23 SubForm 디자인

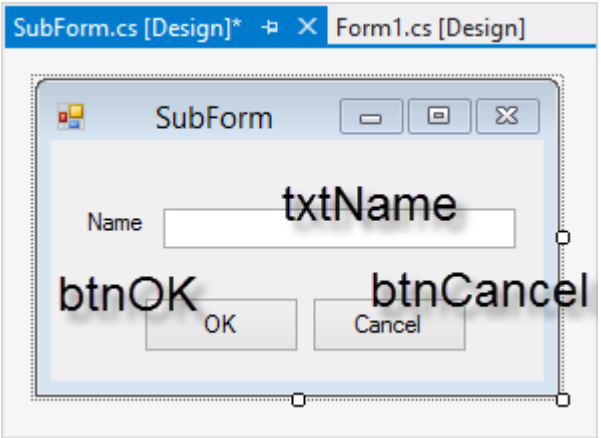


표 20.5 SubForm의 컨트롤 속성

컨트롤	변경된 속성	값
TextBox	(Name)	txtName
Button	(Name)	btnOK
	Text	OK
Button	(Name)	btnCancel

	Text	Cancel
Form	(Name)	SubForm
	AcceptButton	btnOK
	CancelButton	btnCancel

OK 버튼에 대해서는 마우스 클릭을 두 번해서 이벤트 처리기를 생성하고 예제 20.6처럼 코드를 작성한다.

예제 20.6 SubForm.cs의 btnOK 이벤트 처리기

```
using System;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class SubForm : Form
    {
        public string Result { get; private set; }

        public SubForm()
        {
            InitializeComponent();
        }

        private void btnOK_Click(object sender, EventArgs e)
        {
            this.Result = this.txtName.Text;
            this.DialogResult = System.Windows.Forms.DialogResult.OK;
        }
    }
}
```

보다시피 SubForm 타입이 하는 일이라고는 OK 버튼이 눌린 경우 사용자가 입력한 텍스트 박스(txtName)의 내용을 Result 속성에 저장해 두고 DialogResult 속성에 사용자가 한 행위의 의도를 설정하는 것이 전부다. 여기서는 정상적으로 입력했으므로 OK를 지정했고 DialogResult에 값이 대입되는 코드가 실행되면 자동적으로 대화상자가 닫힌다는 점을 알아두자. 반면 btnCancel의 경우에는 Close 메서드를 호출하는 이벤트 처리기를 굳이 만들지 않아도 된다. 왜냐하면 표 20.5에서 이미 버튼을 소유하고 있는 폼(SubForm)의 CancelButton 속성에 btnCancel을 지정했으므로 윈도우는 해당 버튼이 눌리면 자동으로 DialogResult의 값을 Cancel로 설정하기 때문이다. 물론 Cancel로 설정되는 것도 DialogResult에는 값이 대입되는 것이므로 대화상자는 마찬가지로 닫히게 된다.

SubForm을 모두 만들었으니 이제 이를 활용하는 코드를 Form1에 넣어보자. 우선 Button과 Label을 각각 그림 20.24, 표 20.6과 같이 구성하고

그림 20.24 Form1 디자인

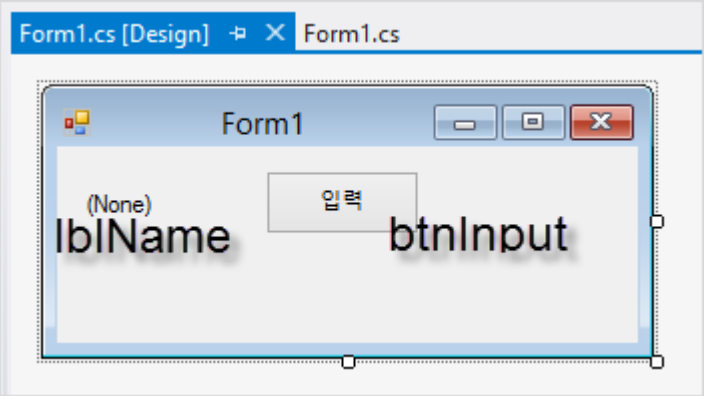


표 20.6 Form1의 컨트롤 속성

컨트롤	변경된 속성	값
Label	(Name)	lblName
	Text	(None)
Button	(Name)	btnInput
	Text	입력

디자인 창에서 "입력" 버튼을 마우스로 두 번 눌러 btnInput의 이벤트 처리기를 생성한 후 다음과 같이 SubForm을 띄우는 코드를 작성한다.

```
using System;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void btnInput_Click(object sender, EventArgs e)
        {
            SubForm subForm = new SubForm();
            if (subForm.ShowDialog() == System.Windows.Forms.DialogResult.OK)
            {
                this.lblName.Text = subForm.Result;
            }
        }
    }
}
```

btnInput 버튼이 눌리면 SubForm 인스턴스를 하나 생성하고 ShowDialog 메서드를 불러 폼을 화면에 보이게 만들었다. ShowDialog 메서드는 호출된 폼이 닫힐 때까지 제어를 반환하지 않기 때

문에 사용자는 계속 진행하려면 반드시 폼을 닫아야 한다. 물론 SubForm에는 OK, Cancel이라는 두 가지 버튼이 있기 때문에 이름을 입력하는 것과 관계없이 아무 버튼이나 눌리면 ShowDialog는 그 결과를 반환한다. 결과적으로 사용자가 OK 버튼을 누른 경우에 한해서 lblName 라벨에는 사용자가 입력한 이름 값이 들어간다.

이렇게 폼이 동작하는 것을 대화상자(Dialog)라고 한다. 대화상자는 일단 표시되면 사용자로 하여금 부모(Parent) 창(예제에서는 Form1)을 사용할 수 없게 만든다. 사용자는 반드시 대화상자가 의도한 동작을 수행하고 OK/Cancel과 같은 버튼을 눌러야만 부모 창으로 되돌아가 다음 작업을 수행할 수 있다.

참고!	SubForm이 보이는 상태에서 부모 폼(Form1)을 마우스를 이용해 선택해 보자. 아무리 해도 선택되지 않는다는 것을 알게 될 것이다. 이때 스피커가 켜져 있다면 효과음이 함께 재생되면서 요청한 작업이 불가능하다는 것을 알린다.

물론 폼을 대화상자로 사용하지 않게 만들 수도 있다. 그 차이를 테스트해보기 위해 btnInput_Click의 코드를 다음과 같이 변경해 보자.

```
using System;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void btnInput_Click(object sender, EventArgs e)
        {
            SubForm subForm = new SubForm();
            subForm.Show();
        }
    }
}
```

Show 메서드는 ShowDialog와는 다른 동작을 수행한다. 이 메서드는 SubForm을 화면에 보여주고 금방 제어를 반환한다. 따라서 DialogResult 값도 구할 수 없다. 왜냐하면 사용자가 어떤 동작을 취하기도 전에 Show 메서드의 실행은 끝나버린 상태이고, 실제로 OK/Cancel 버튼이 눌린 시점에는 btnInput_Click의 메서드에 있는 코드에서 스레드가 머물러 있지 않기 때문이다. 위의 코드를 실행해 보면 2개의 폼(Form1, SubForm)이 화면에 떠 있고 사용자는 마우스를 이용해 원하는 폼을 활성화해서 사용할 수 있다.

Show 또는 ShowDialog를 언제 사용해야 하는지에 대해서는 폼의 목적에 따라 다르다. 사용자에게서 반드시 입력을 받고 진행해야 할 필요성이 있다면 ShowDialog를 호출하면 되고, 부모 폼과 함께 연동되면서 떠 있을 필요가 있다면 Show를 호출하면 된다.

참고!	일례로, 비주얼 스튜디오에서 "새 프로젝트"를 선택하는 폼은 ShowDialog로 호출된 경우다. 반면 "Ctrl + F"키를 눌러 뜨는 "찾기" 창은 Show 메서드로 호출될 필요가 있는 전형적인 사례다.

사용자에게 메시지를 보여주는 목적으로 사용된 MessageBox.Show(.....)도 따지고 보면 전형적인 대화상자의 하나에 해당한다.

엄격히 말하면 대화 상자는 Form으로 만들어지기 때문에 이를 좀더 구분하기 위해 Show로 뜨는 경우 "모드리스(Modeless) 대화창"이라 하고, ShowDialog로 뜨는 경우를 "모달(Modal) 대화창"이라고 한다.

19.1.6 공통 대화상자

System.Windows.Forms.dll 어셈블리에는 미리 만들어진 유용한 대화상자들이 제공되는데 이를 일반적으로 "공통 대화상자(common dialog)"라고 한다. 여기서는 대표적으로 5개의 대화상자 사용법을 알아보겠다.

우선 폴더를 열람하면서 이미 생성돼 있는 파일을 선택하는 OpenFileDialog의 사용법을 보자.

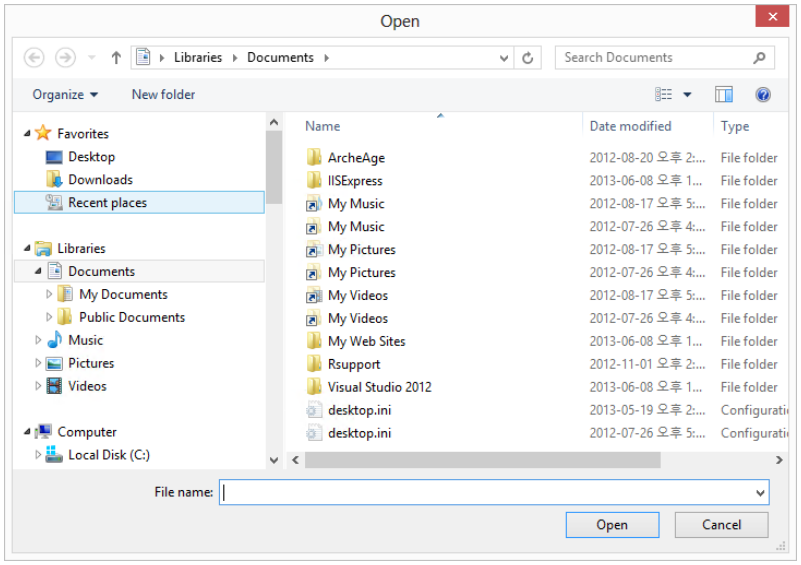
예제 20.7 OpenFileDialog 사용법

```
using System;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            using (OpenFileDialog openDlg = new OpenFileDialog())
            {
                if (openDlg.ShowDialog() == System.Windows.Forms.DialogResult.OK)
                {
                    MessageBox.Show("선택한 파일: " + openDlg.FileName);
                }
            }
        }
    }
```

```
}
```

이 대화상자의 사용법은 앞에서 배운 것에서 크게 벗어나지 않는다. ShowDialog를 호출한 후 사용자가 확인(OK) 버튼을 눌렀는지 여부를 검사하고, 만약 파일이 선택됐다면 해당 파일의 전체 경로를 FileName 속성으로 제공한다. 그림 20.25는 ShowDialog를 호출했을 때 나타나는 "파일 열기 대화상자"다.

그림 20.25 파일 열기 대화상자



기존 파일을 선택하는 경우 "파일 열기 대화상자"를 사용할 수 있겠지만, "새 파일"을 생성했다가 저장하거나 "다른 이름으로 저장"과 같은 기능을 구현하려면 별도로 "파일 저장 대화상자"도 필요하다. 이는 SaveFileDialog 타입으로 구현되고 다음과 같이 사용할 수 있다.

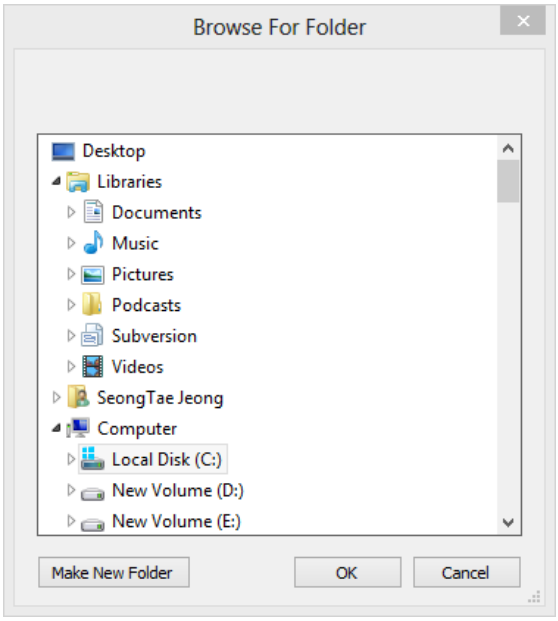
```
using (SaveFileDialog saveDlg = new SaveFileDialog())
{
    if (saveDlg.ShowDialog() == System.Windows.Forms.DialogResult.OK)
    {
        MessageBox.Show("새 파일 이름: " + saveDlg.FileName);
    }
}
```

"저장 대화상자"는 "열기 대화상자"와 외관이 거의 비슷한데 단지 저장할 파일을 선택하는 것이므로 사용자가 입력한 파일이 존재하지 않아도 정상적으로 반환받을 수 있게 해준다는 차이점이 있다.

파일에 대한 열기/저장 대화상자가 있다면 당연히 폴더를 선택하는 대화상자도 있을 것이다. 이는 FolderBrowserDialog 타입으로 제공되고 이 대화상자를 사용하는 코드와 대화상자의 외양은 각각 다음과 같다.

```
using (FolderBrowserDialog folderDlg = new FolderBrowserDialog())
{
    if (folderDlg.ShowDialog() == System.Windows.Forms.DialogResult.OK)
    {
        MessageBox.Show("선택한 폴더: " + folderDlg.SelectedPath);
    }
}
```

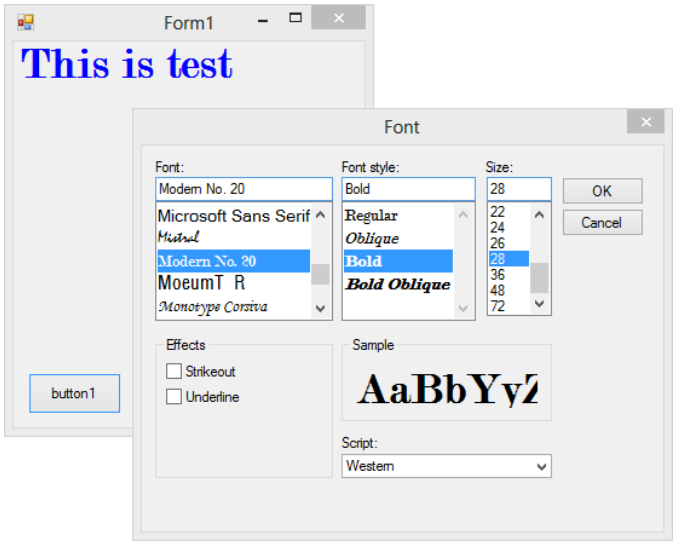
그림 20.26 폴더 선택 대화상자



파일/폴더와 상관없는 다른 공통 대화상자로는 폰트(FontDialog), 색상(ColorDialog)을 선택하는 대화상자가 있다. 폰트의 경우 대화상자 안에서 폰트의 종류와 크기, 효과, 색을 모두 선택할 수 있고 반환받은 Font 속성을 곧바로 코드에서 사용할 수 있다.

```
using (FontDialog fontDlg = new FontDialog())
{
    if (fontDlg.ShowDialog() == System.Windows.Forms.DialogResult.OK)
    {
        using (Graphics g = this.CreateGraphics())
        {
            g.DrawString("This is test", fontDlg.Font, Brushes.Blue, 0, 0);
        }
    }
}
```

그림 20.27 폰트 대화상자

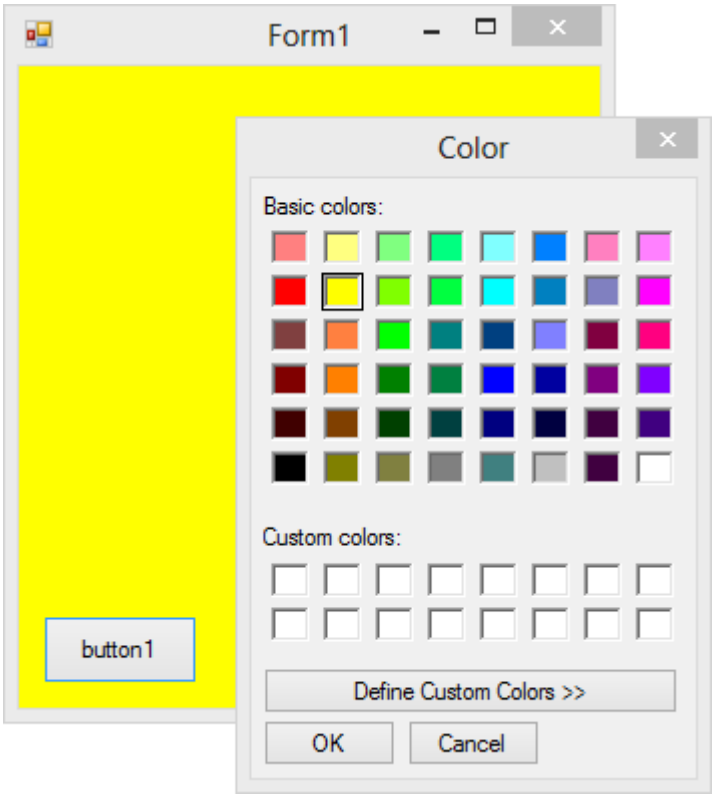


마지막으로 아래는 컬러 대화상자의 사용법을 보여준다.

```
using (ColorDialog colorDlg = new ColorDialog())
{
    colorDlg.Color = this.BackColor;

    if (colorDlg.ShowDialog() == System.Windows.Forms.DialogResult.OK)
    {
        this.BackColor = colorDlg.Color;
    }
}
```

ShowDialog를 호출하기 전에 윈도우 폼의 배경색(BackColor) 값을 미리 대화상자에 설정했다. 이렇게 하면 대화상자가 나타났을 때 미리 해당 색상을 선택된 채로 사용자에게 보여줄 수 있다.

그림 20.28 색상 선택 대화상자

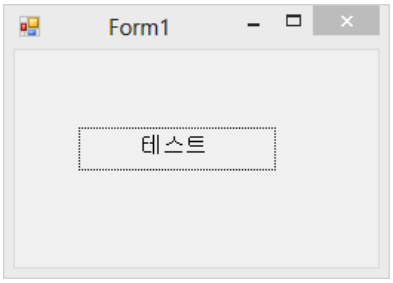


지금까지 소개한 5개의 대화상자는 윈도우 운영체제에서 제공하는 대화상자다. 물론 프로그램을 만들다 보면 이 밖에도 많은 대화상자가 필요해지는데, 그런 경우에는 폼(Form)을 하나 만들고 내부의 구성 요소를 적절하게 배치한 다음 ShowDialog 메서드를 호출하면 된다. 직접 만든 대화상자 코드의 재사용성을 높인다면 다른 프로젝트를 할 때도 가져다 쓸 수 있다.

19.1.7 사용자 지정 컨트롤

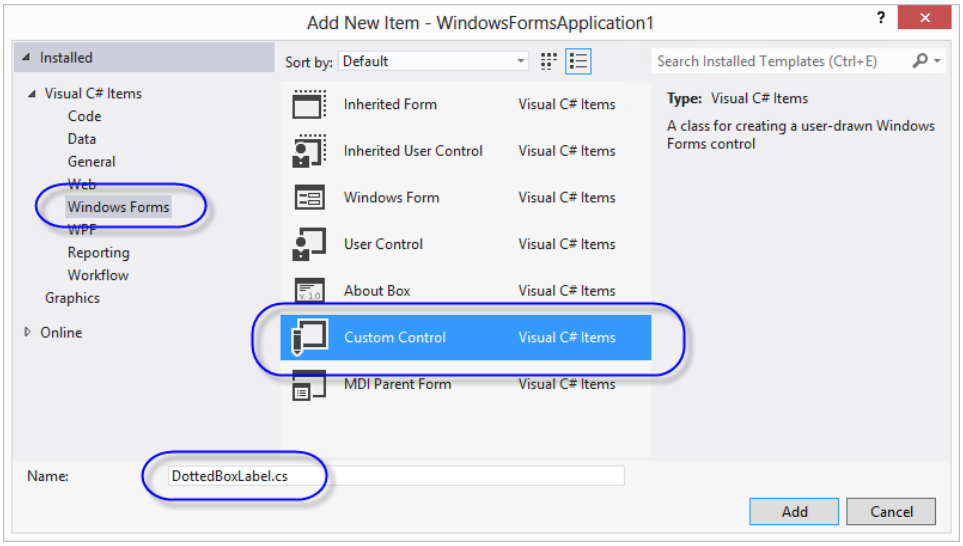
폼과 대화상자를 사용자가 새롭게 만들었던 것과 마찬가지로, Form 내부에 배치되는 컨트롤도 직접 만들 수 있다. 컨트롤은 윈도우(Window) 자원의 일종이므로 지금까지 배운 메시지, 메뉴, Graphics를 모두 사용할 수 있고, 이를 이용해 원하는 동작을 수행하는 컨트롤을 만들어 재사용할 수 있다. 간단한 예를 들어보기 위해 이번 절에서는 점선으로 둘러싸인 사각형에 문자열을 출력하는 DottedBoxLabel 컨트롤을 만들어 볼 텐데, 그림 20.29처럼 보이게 만들 것이다.

그림 20.29 DottedBoxLabel의 모습



솔루션 탐색기의 프로젝트 항목을 마우스 오른쪽 버튼으로 누른 다음 "추가(Add)" / "새 항목(New Item)"을 선택한 후 나오는 대화상자의 "Windows Forms" 범주에서 "사용자 지정 컨트롤(Custom Control)" 항목으로 "DottedBoxLabel.cs" 파일을 추가한다.

그림 20.30 DottedBoxLabel.cs 사용자 지정 컨트롤 추가



생성된 DottedBoxLabel.cs 파일의 코드를 보면 다음과 같은 기본 구성을 볼 수 있다.

예제 20.8 사용자 지정 컨트롤의 기본 코드

```
using System.Drawing;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class DottedBoxLabel : Control
    {
        public DottedBoxLabel()
        {
            InitializeComponent();
        }

        protected override void OnPaint(PaintEventArgs pe)
        {
            base.OnPaint(pe);
```

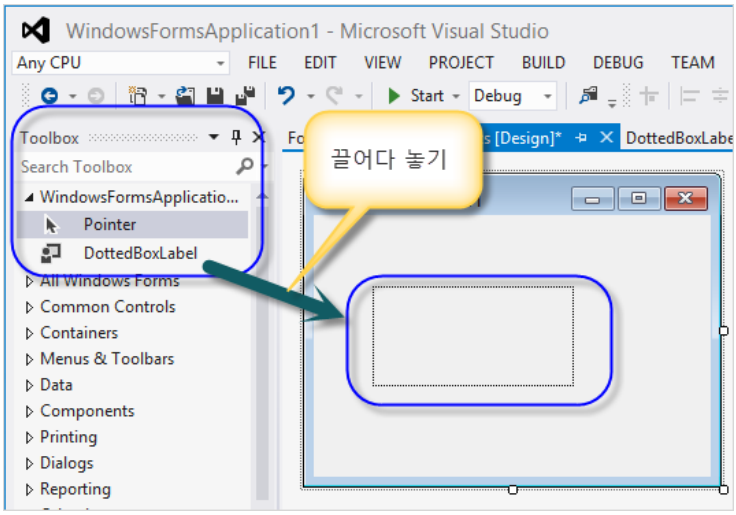
```
        }
    }
}
```

컨트롤로서 동작하기 위한 모든 코드를 직접 작성할 수도 있겠지만, 다행히 System.Windows.Forms.dll 어셈블리에는 사용자 지정 컨트롤의 기본 동작을 구현한 Control 클래스가 제공되므로 대부분 이를 상속받아 구현하면 된다. 따라서 기본적인 윈도우 메시지 처리 작업은 신경 쓸 필요 없이 원하는 동작에 집중할 수 있다. 우리가 만드는 컨트롤은 사각형 박스가 점선으로 이뤄져 있으므로 OnPaint에 다음과 같은 작업을 추가한다.

```
protected override void OnPaint(PaintEventArgs pe)
{
    base.OnPaint(pe);

    // 컨트롤의 크기를 ClientRectangle 속성으로 구하고,
    Rectangle rectArea = this.ClientRectangle;

    // Inflate 메서드는 Rectangle 크기를 조정한다.
    // width와 height 인자에 모두 -1을 지정했기 때문에 사각형 크기가 1씩 줄어든다.
    // 이렇게 하는 이유는 선을 그리기 위한 영역으로 1 픽셀씩 차지하기 때문이다.
    rectArea.Inflate(-1, -1);

    using (Pen dottedPen = new Pen(Brushes.Black, 1))
    {
        // Pen의 선 스타일을 Dot로 변경
        dottedPen.DashStyle = System.Drawing.Drawing2D.DashStyle.Dot;
        pe.Graphics.DrawRectangle(dottedPen, rectArea);
    }
}
```

일단 이렇게만 코드를 작성하고 프로젝트를 빌드한다. 그러면 도구 상자(Toolbox) 영역에 DottedBoxLabel 컨트롤이 생성되는 것을 볼 수 있다. 이를 Form 디자인 창에 끌어다 놓으면 그림 20.31처럼 점선으로 된 사각형을 그리는 컨트롤이 나타난다.

그림 20.31 도구 상자에 추가된 DottedBoxLabel 컨트롤

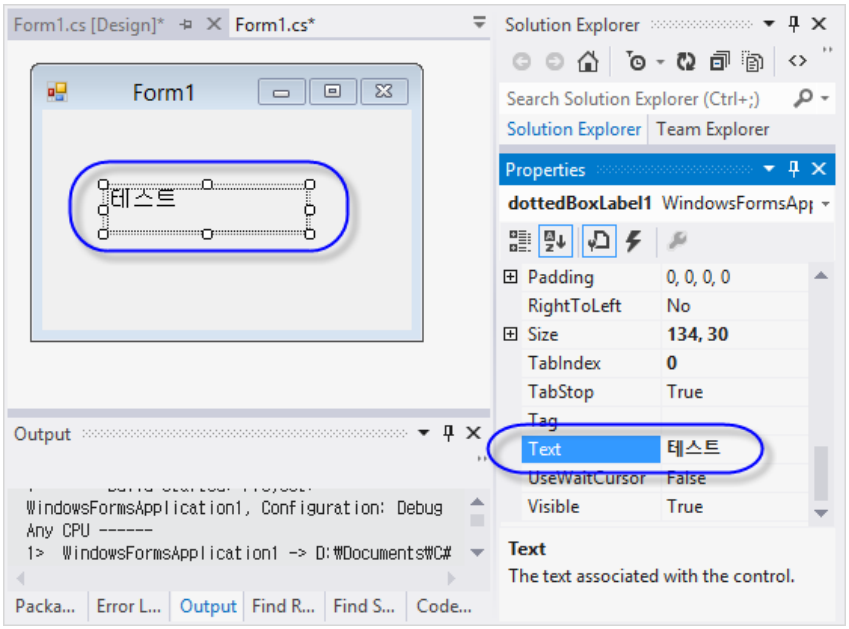


라벨 컨트롤이라서 당연히 출력해야 할 텍스트가 필요하다. DottedBoxLabel 타입에 Text 속성을 정의해야 하지만 부모 클래스인 Control 타입은 이 속성을 기본적으로 제공하므로 단순히 OnPaint에서 이를 출력하면 된다.

```
protected override void OnPaint(PaintEventArgs pe)
{
    base.OnPaint(pe);

    // ..... [생략] .....

    // 문자열이 그려질 때 사용할 폰트를 생성
    using (Font normalFont = new Font(FontFamily.GenericSansSerif, 10))
    {
        // 문자열을 점선으로 그려진 사각형 내부에 출력한다.
        pe.Graphics.DrawString(this.Text, normalFont, Brushes.Black, rectArea);
    }
}
```

프로젝트를 다시 빌드하면 그림 20.32처럼 Form 디자인 창에서 컨트롤을 마우스로 선택한 후 "속성(Properties)" 창에서 Text 속성을 바꿀 수도 있고, Form1.cs 코드 창에서 해당 컨트롤의 변수를 이용해 직접 Text 속성을 바꿀 수도 있다.

그림 20.32 속성창을 통해 컨트롤의 Text 상태값 변경



텍스트의 출력 위치가 아직 정교하지 않은데, 원한다면 다음과 같이 중앙으로 올 수 있게 변경할 수도 있다.

```
protected override void OnPaint(PaintEventArgs pe)
{
    base.OnPaint(pe);

    // ..... [생략] .....

    using (Font normalFont = new Font(FontFamily.GenericSansSerif, 10))
    {
        // 지정된 폰트로 출력될 텍스트의 크기를 먼저 계산하고,
        SizeF outputSize = pe.Graphics.MeasureString(this.Text, normalFont);

        // 사각형의 중간에 출력되는 위치를 구한다.
        PointF pt = new PointF();
        pt.X = (rectArea.Width - outputSize.Width) / 2;
        pt.Y = (rectArea.Height - outputSize.Height) / 2;

        pe.Graphics.DrawString(this.Text, normalFont, Brushes.Black, pt);
    }
}
```

위와 같이 코드를 작성하고 실행하면 최종적으로 그림 20.29처럼 컨트롤의 외양이 출력되는 모습을 볼 수 있다.

이렇게 해서 간단하게 재사용할 수 있는 사용자 지정 컨트롤을 만들어봤는데, 실제로 현업에서 사용할 만한 컨트롤을 만들려면 더 많은 윈도우 폼 관련 프로그래밍 지식이 필요하다. 지면 관계상 모든 내용을 설명할 수는 없고, 더 자세한 사항들은 시간을 두고 꾸준히 학습하길 바란다.

19.1.8 사용자 정의 컨트롤

윈도우 폼 프로그램에서 재사용 가능한 컨트롤을 만드는 방법은 두 가지가 있다.

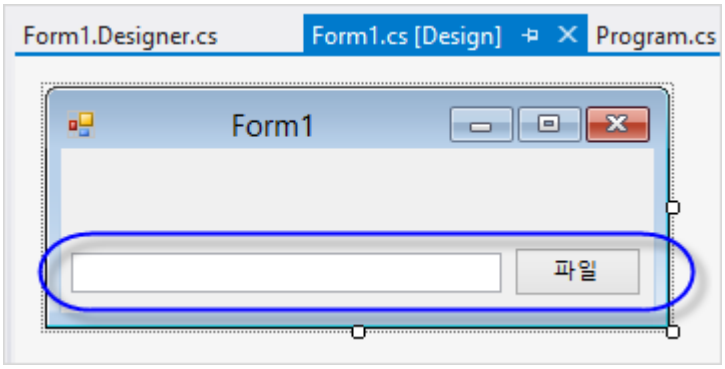
- 사용자 지정 컨트롤(Custom Control)
  Control 타입을 상속받아 단일 컨트롤로서의 동작을 정의

- 사용자 정의 컨트롤(User Control)
  UserControl 타입을 상속받고, 다양한 컨트롤을 조합해서 재사용 가능한 컨트롤을 정의

이름이 다소 비슷해서 혼동이 올 수 있지만, 이번에 배우는 것은 기존 컨트롤을 조합해서 새로운 복합 컨트롤을 정의하는 "사용자 정의 컨트롤"을 만드는 방법이다.

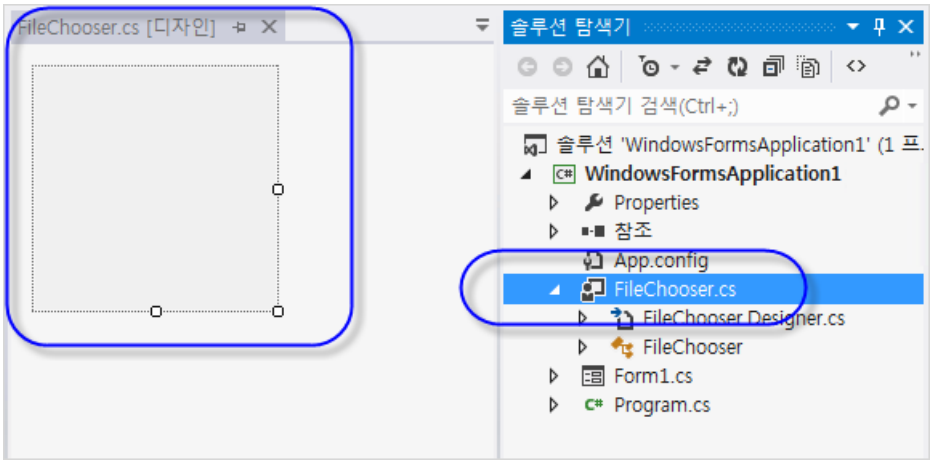
여기서 실습할 사용자 정의 컨트롤은 파일을 선택하면 좌측의 텍스트 박스에 그 경로를 보여주는 FileChooser 타입인데, 그림 20.33에서 보다시피 두 가지 컨트롤(TextBox, Button)이 하나의 사용자 정의 컨트롤로 동작하게 된다.

그림 20.33 FileChooser 사용자 정의 컨트롤



본격적인 실습을 위해 새롭게 윈도우 폼 프로젝트를 하나 만들고 솔루션 탐색기의 프로젝트 항목을 마우스 오른쪽 버튼으로 누른 다음 "추가(Add)" / "새 항목(New Item)"을 선택한 후 나오는 대화상자의 "Windows Forms" 범주에서 "사용자 정의 컨트롤(UserControl)" 항목으로 "FileChooser.cs" 파일명을 지정해서 추가한다. 그러면 그림 20.34처럼 Form 디자인 화면과 유사한 화면이 제공되는데, 이것이 바로 "사용자 지정 컨트롤"과 비교했을 때 가장 큰 차이점이다.

그림 20.34 디자인 화면이 제공되는 사용자 정의 컨트롤



디자인 영역의 크기를 적당히 조정하고 그림 20.35, 표 20.7에 따라 TextBox, Button 컨트롤 두 개를 추가한 후 적절하게 속성값을 입력한다.

그림 20.35 FileChooser 사용자 정의 컨트롤 디자인

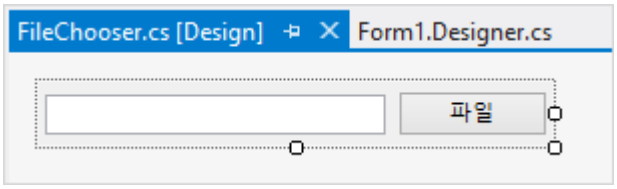


표 20.7 FileChooser의 내부 컨트롤 속성

컨트롤	변경된 속성	값
TextBox	(Name)	txtFileName
Button	(Name)	btnFile

이제 "파일" 버튼이 눌렸을 때 파일 선택 대화상자를 띄우고, 사용자가 파일을 선택한 경우 텍스트 박스에 파일 경로를 출력하는 코드를 작성한다.

예제 20.9 FileChooser.cs 코드 파일

```
using System;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class FileChooser : UserControl
    {
        public event EventHandler FileSelected; // 파일이 선택됐을 때 이벤트 발생

        // 외부에서 선택된 파일 경로를 가져갈 수 있게 읽기 속성을 제공
        string _selectedFilePath;
```

```
        public string SelectedFilePath
        {
            get { return _selectedFilePath; }
        }

        public FileChooser()
        {
            InitializeComponent();
        }

        private void btnFile_Click(object sender, EventArgs e)
        {
            using (OpenFileDialog openDlg = new OpenFileDialog())
            {
                if (openDlg.ShowDialog() == DialogResult.OK)
                {
                    this.txtFileName.Text = openDlg.FileName;
                    this._selectedFilePath = openDlg.FileName;

                    if (FileSelected != null)
                    {
                        // 외부에서 이벤트를 걸어뒀다면 파일이 선택됐음을 통보한다.
                        FileSelected(this, EventArgs.Empty);
                    }
                }
            }
        }
    }
}
```

프로젝트를 빌드하면 도구 상자에 새롭게 "FileChooser" 항목이 나타나는데, Form1.cs 디자인 창을 열고 FileChooser를 끌어서 폼 위에 추가한다(그림 20.36과 표 20.8 참고).

그림 20.36 Form1 디자인 창에 FileChooser 사용자 정의 컨트롤 추가

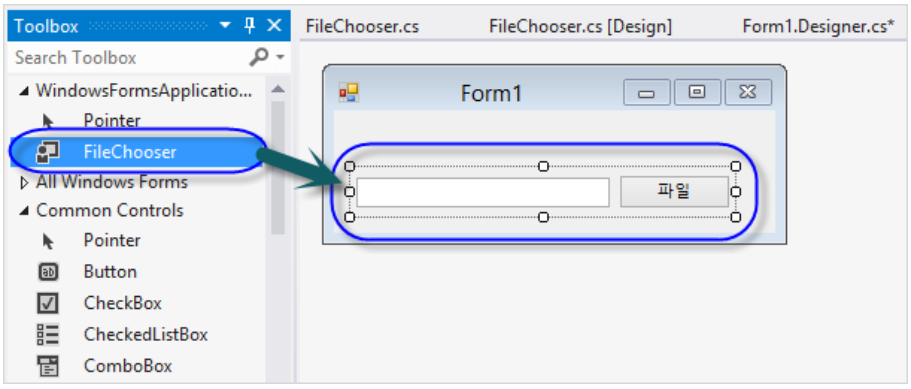


표 20.8 FileChooser 속성

컨트롤	변경된 속성	값
FileChooser	(Name)	fileChooser

그림 20.36에서 볼 수 있듯이 두 개의 컨트롤(TextBox, Button)이 Form 디자인에 추가됐지만 Form1.cs 코드 파일에서는 단일 컨트롤처럼 다룰 수 있다. 따라서 Form1.cs 코드에서 FileSelected 이벤트를 추가하고 선택된 파일의 경로를 구하는 것까지 할 수 있다.

```
using System;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            // 이벤트 처리기를 추가하고,
            this.fileChooser.FileSelected += fileChooser_FileSelected;
        }

        void fileChooser_FileSelected(object sender, EventArgs e)
        {
            // 파일이 선택됐음을 통지받아 처리한다.
            MessageBox.Show(this.fileChooser.SelectedFilePath);
        }
    }
}
```

정리하자면 사용자 정의 컨트롤을 반드시 알아야 할 필요는 없다. 왜냐하면 굳이 사용자 정의 컨트롤로 만들지 않아도 직접 폼 위에 표 20.7에 나열된 컨트롤을 올려 놓고 사용해도 되기 때문이다. 그럼에도 일반적으로 사용자 정의 컨트롤을 만드는 데는 두 가지 이유가 있다.

- 재사용 가능한 컨트롤 묶음 정의
  서로 연관이 있는 컨트롤을 하나의 "사용자 정의 컨트롤"로 만들어 재사용성을 높인다. 수많은 상업용 컨트롤 라이브러리가 이처럼 재사용을 염두에 두고 만들어져 배포되고 있다.

- Form 내부의 컨트롤에 대한 구획화
  Form 안에 너무 많은 컨트롤이 내장되는 경우, 이러한 컨트롤에 연계되는 코드까지 더해지면 Form 타입의 코드 복잡도가 증가한다. 따라서 적절하게 사용자 정의 컨트롤로 분리해 Form 내부의 코드를 분산하면 유지보수하는 데도 도움이 된다.

정리

윈도우가 포함된 응용 프로그램은 광범위하게 사용된다. 가장 흔한 예로 "일반 사용자를 대상으

로 한 응용 프로그램"은 거의 대부분 윈도우 프로그램이다. 또한 윈도우 안에서 다양한 그래픽 처리를 할 수 있기 때문에 시각화 처리가 필요한 응용 프로그램에도 필수적으로 쓰인다. 심지어 UI가 필요 없는 서버 측 프로그램을 주로 만드는 개발자에게도 해당 프로그램을 모니터링하는 용도의 응용 프로그램으로는 UI를 제공하는 윈도우 프로그램으로 만드는 것이 일반적이다.

이처럼 다양한 활용 사례를 기대할 수 있기 때문에 어떤 식으로든 "윈도우 프로그램"을 만들 줄 아는 것은 개발자 입장에서 놓칠 수 없는 중요한 능력 중의 하나다. 그리고 필자가 아는 한, "윈도우 프로그램"을 만들 수 있는 가장 쉬운 방법은 닷넷의 "윈도우 폼 응용 프로그램"이다. 특히 기존의 Visual C++에서 MFC 라이브러리를 이용해 윈도우 프로그램을 만들던 개발자라면 얼마나 편리해졌는지 충분히 느꼈을 것이다. 이 때문에 닷넷 프레임워크는 Visual C++ 개발자에게 날개를 달아주는 것과 같다. 번잡한 UI 구성은 닷넷 언어를 이용해 쉽게 작성하고 보호가 필요한 핵심 코드는 Visual C++에 맡기는 식으로 프로그램을 만들면 극적인 생산성 향상을 기대할 수 있다.

20.2 WPF 응용 프로그램

마이크로소프트는 닷넷 프레임워크 3.0에서 XAML(Extensible Application Markup Language)의 한 사례로 WPF(Windows Presentations Framework)를 소개했다. 여기서 XAML이란 간단하게 말해 닷넷 객체를 XML로 표현하는 기술이라고 보면 된다. 이해를 돕기 위해 6.3 '직렬화/역직렬화' 절에서 배운 내용을 다시 살펴보자. 예를 들어, 다음 코드를 실행하면

```
using System;
using System.IO;
using System.Text;
using System.Xml.Serialization;

public class Person
{
    public string Name;
    public int Age;
}

class Program
{
    static void Main(string[] args)
    {
        Person p1 = new Person { Age = 37, Name = "Anderson" };

        MemoryStream ms = new MemoryStream();
        XmlSerializer xs = new XmlSerializer(typeof(Person));
        xs.Serialize(ms, p1);

        ms.Position = 0;
        Console.WriteLine(Encoding.UTF8.GetString(ms.GetBuffer()));
    }
}
```

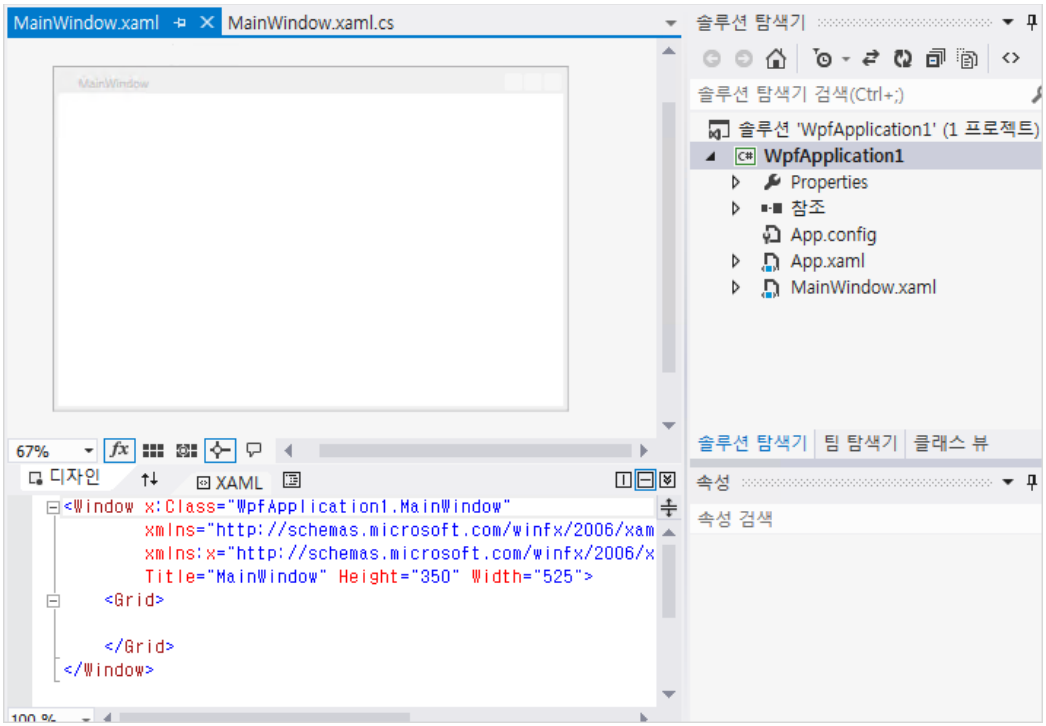
화면에는 Person 객체의 내용이 XML 형식으로 출력된다.

```
<?xml version="1.0"?>
<Person xmlns:xsi="....." xmlns:xsd=".....">
  <Name>Anderson</Name>
  <Age>37</Age>
</Person>
```

결국 위의 XML과 "Person p1 = new Person { Age = 37, Name = "Anderson" };"이라는 C# 코드는 완전히 등가적인 역할을 하는 셈이다. 마이크로소프트는 이런 관계를 사용자 UI 구성 요소를 표현하는 데 적용하기 시작했고 그 결과물로 나온 것이 바로 WPF다. 이 때문에 WPF 응용 프로그램에서는 UI 구성 요소를 코드뿐 아니라 XML 표기로도 생성하는 방식을 지원한다.

이쯤에서 WPF 프로젝트를 하나 만들어 보면 더욱 쉽게 이해할 수 있다. 비주얼 스튜디오에서 "파일" / "새 프로젝트" 메뉴를 선택했을 때 나온 그림 20.1의 "WPF 응용 프로그램(Application)"을 지정해 프로젝트를 생성하면 그림 20.37과 같은 초기 화면을 볼 수 있다.

그림 20.37 WPF 프로젝트



기본적으로 MainWindow.xaml 파일이 열려 있는데, 중앙에는 디자인 화면이 보이고 하단에는 xaml 파일의 원본 내용이 보인다. 하단의 xaml 내용을 자세히 보면 다음과 같은 XML 형식인데,

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
```

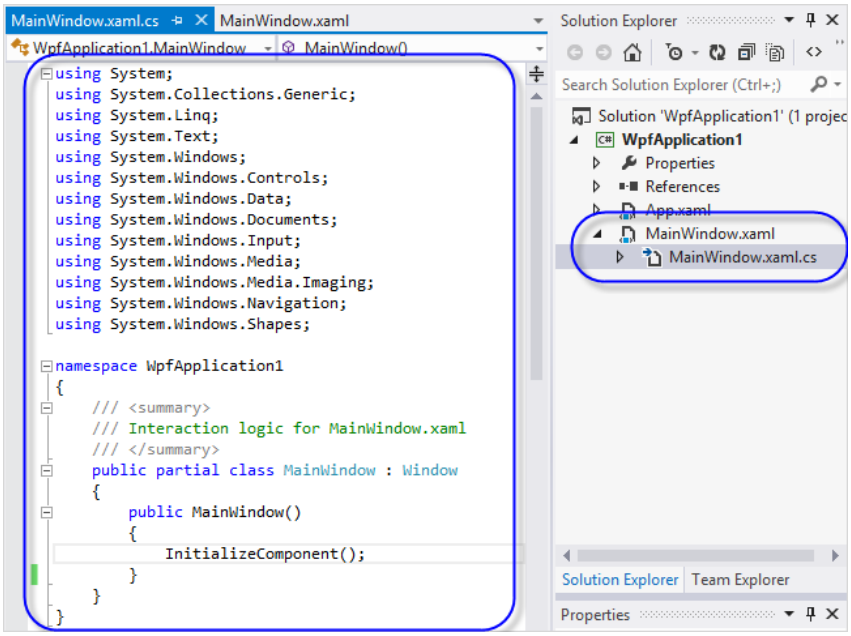
```
    </Grid>
</Window>
```

이는 아래의 코드와 동일한 역할을 수행한다.

```
WpfApplication1.MainWindow window = new WpfApplication1.MainWindow();
window.Title = "MainWindow";
window.Height = 350;
window.Width = 525;

Grid grid = new Grid();
window.Content = grid;
```

솔루션 탐색기에서 MainWindow.xaml을 펼쳐 보면 MainWindow.xaml.cs 파일을 선택할 수 있고 기본 내용은 그림 20.38과 같다.

그림 20.38 MainWindow.xaml.cs 파일



코드 파일에는 WpfApplication1 네임스페이스 아래에 MainWindow 타입이 정의돼 있고 생성자에는 InitializeComponent 메서드가 호출되고 있다. 이 관계를 예제 20.1의 윈도우 폼 응용 프로그램과 비교해 보면 표 20.9처럼 정리할 수 있다.

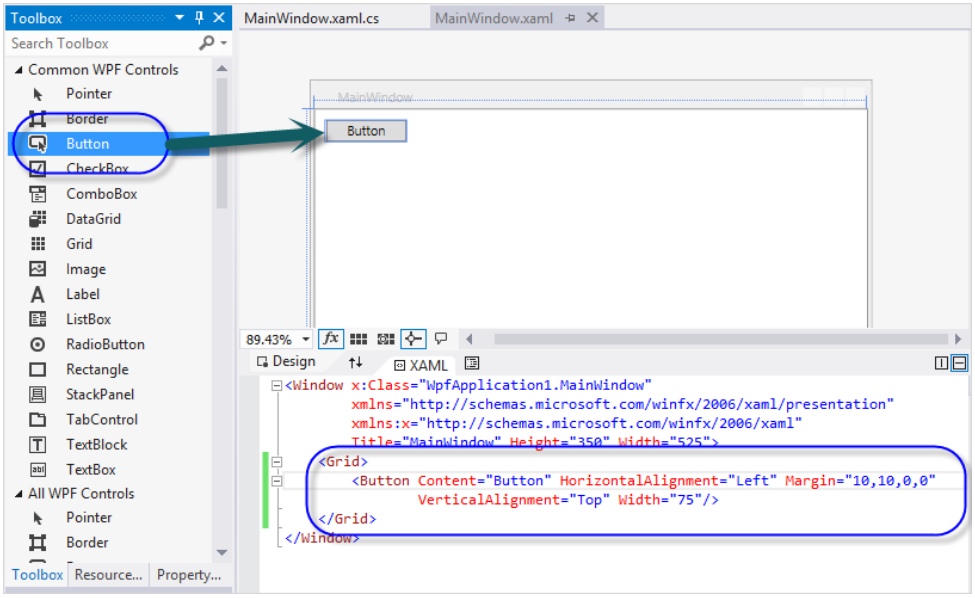
표 20.9 WPF와 윈도우 폼의 대응 관계

	윈도우 폼	WPF
코드 파일	Form1.cs	MainWindow.xaml.cs
디자인 파일	Form1.Designer.cs	MainWindow.xaml

즉, 디자인을 위한 코드 파일이 XML 형식의 xaml 파일로 아예 대체돼 버린 것이다.

기본적인 디자인 방법은 윈도우 폼과 비교해서 크게 다르지 않다. 도구 상자의 "Common WPF 컨트롤" 또는 "All WPF 컨트롤" 영역에 있는 항목을 xaml 디자인 화면에 끌어다 놓으면 되는데, 그림 20.39에서는 버튼을 하나 추가한 모습을 볼 수 있다.

그림 20.39 Button 컨트롤을 추가



xaml 디자인 화면에 방금 추가한 버튼이 하나 보이고, 하단의 xaml 파일 내용에는 새롭게 <Button /> 태그가 추가돼 있다. 윈도우 폼과 마찬가지로 디자인 화면에서 버튼 컨트롤을 마우스로 두 번 클릭하면 하단의 XAML 파일에는 Click 속성과 함께 이벤트 처리기의 메서드 이름이 추가되고 동시에 MainWindow.xaml.cs 파일이 열리면서 Click 이벤트 처리기가 추가된다.

```
<Window ……[생략]……
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button Content="Button" HorizontalAlignment="Left" Margin="10,10,0,0"
                VerticalAlignment="Top" Width="75" Click="Button_Click"/>
    </Grid>
</Window>
```

테스트를 위해 간단하게 예제 20.10과 같이 MessageBox를 추가하고 실행해 보면 버튼이 눌릴 때마다 메시지 창이 나타나는 것을 알 수 있다.

예제 20.10 WPF의 버튼 이벤트 처리기

```
using System.Windows;

namespace WpfApplication1
{
```

```
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show("Clicked!");
        }
    }
}
```

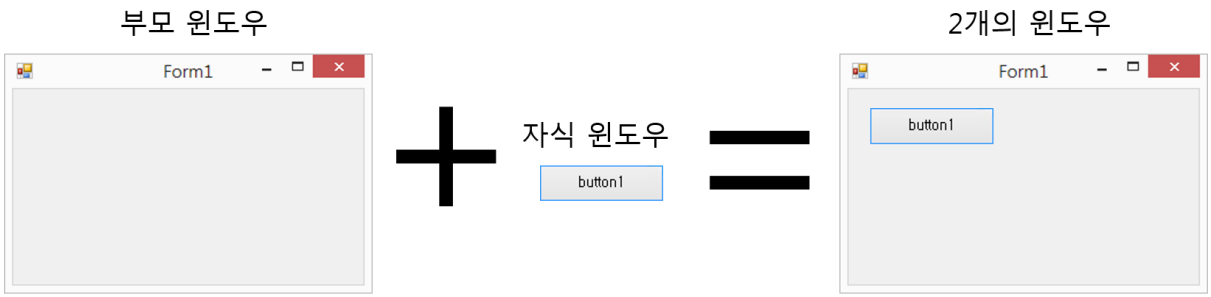
물론 디자인 화면에서 버튼을 두 번 누르는 방법 대신 XAML 파일에서 직접 <Button /> 태그 속성에 Click 텍스트를 입력하고 이벤트 처리기로 정의될 메서드 이름을 임의로 지정한 후, MainWindow.xaml.cs 파일을 열어 그 이름과 일치하는 메서드를 직접 정의해도 무방하다. 어떤 방식을 이용하느냐는 개인적인 취향의 문제다.

이렇게 디자인 요소를 기존의 코드 파일에서 XML 형식의 XAML로 바뀌어서 좋은 점이 있다면 디자인을 개발자가 아닌 디자이너에게 맡길 수 있다는 점이다. 실제로 마이크로소프트에서는 WPF 응용 프로그램의 XAML 디자인을 위해 디자이너에게 익숙한 포토샵(Photoshop) 프로그램과 유사한 "익스프레션 블렌드(Expression Blend)"라는 응용 프로그램을 개발했다. 이 프로그램은 WPF 프로젝트 파일을 읽어들일 수 있으며, 디자이너가 XAML 파일을 대상으로 코드에 영향을 주지 않고도 프로그램의 외양을 바꿀 수 있게 해준다. 게다가 TFS(Team Foundation Server) 형상 관리 시스템까지 지원하므로 디자이너와 개발자가 자연스럽게 소스 컨트롤 하에서 협업하는 것도 가능하다.

19.2.1 WPF 렌더링 방식

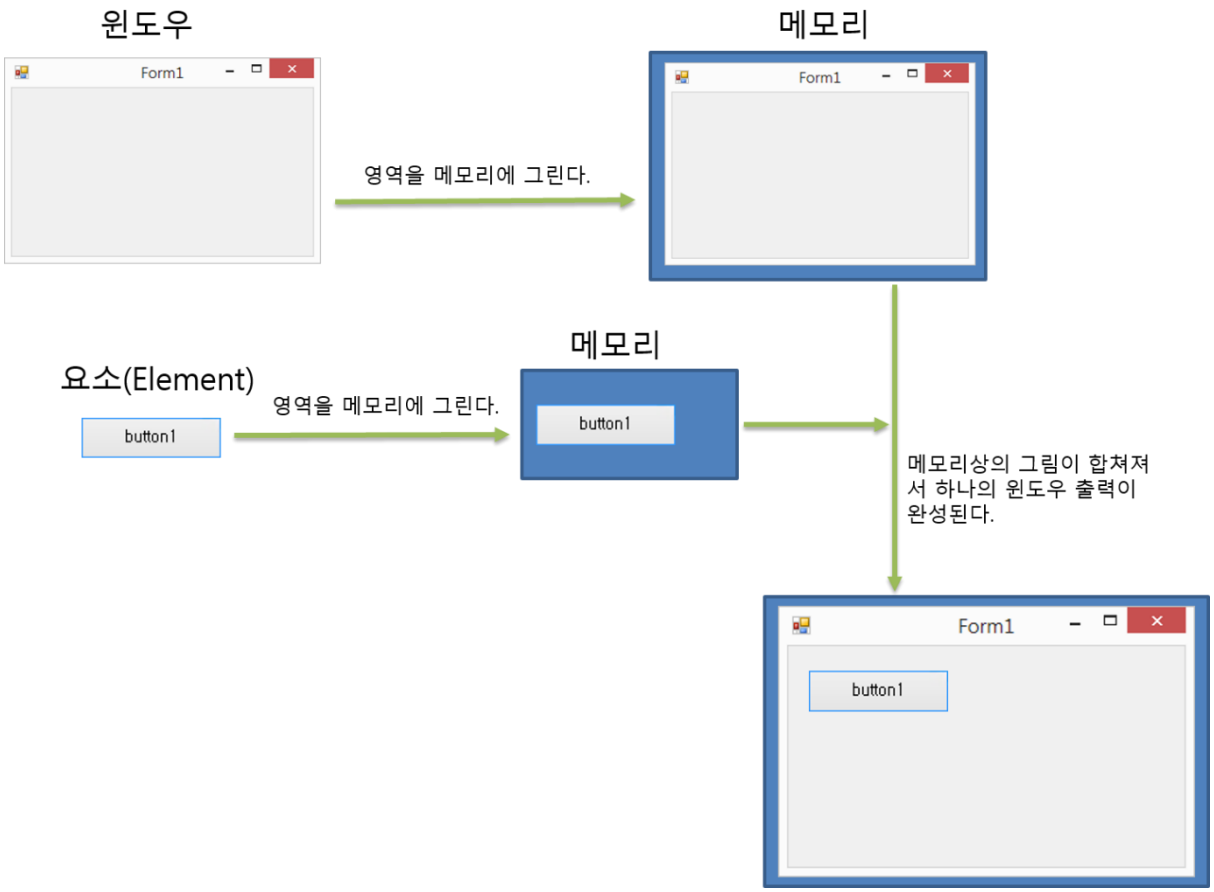
WPF 응용 프로그램에서 XAML이 디자인을 담당한다는 것과 함께 또 한 가지 크게 달라진 점이 있다면 내부 컨트롤이 더는 윈도우를 기반으로 동작하지 않는다는 점이다. 예를 들어, Button 컨트롤이 포함된 윈도우 폼 응용 프로그램의 동작은 Form과 Button이 각각 운영체제에서 제공되는 윈도우(Window) 자원으로 구현된다. 즉, 화면에는 두 개의 윈도우가 부모/자식 관계로 구성되어 각 윈도우는 독립적으로 메시지에 반응하도록 구현된다(그림 20.40 참고).

그림 20.40 윈도우 폼의 구성



하지만 WPF에서는 컨트롤이 더 이상 윈도우가 아니다. 오직 Window 태그에 해당하는 것만 운영체제의 윈도우로 구현되고, 내부의 모든 자식 컨트롤은 윈도우의 일정 영역을 할당받아 그리는 역할만 한다. 이 때문에 컨트롤이라는 이름 외에도 요소(element)라는 이름으로 더 자주 불린다. 화면에 보이게 되는 출력 방식도 독특하다. 요소들은 WPF 내부에 할당된 메모리 영역에 자신의 영역을 그리고, 그려진 모든 자식 요소들의 이미지는 차례대로 합쳐져서 윈도우 영역의 이미지와 합쳐져서 그려진다(그림 20.41 참고).

그림 20.41 WPF의 출력 구성



렌더링 방식을 새롭게 바꾼 데는 그만한 이유가 있다. 왜냐하면 다음과 같은 이점이 있기 때문이다.

- 그래픽 가속 가능
  GPU의 그래픽 가속 기능을 사용함으로써 렌더링으로 인한 CPU 부하를 줄인다.

- 다중 윈도우를 사용하지 않기 때문에 부하 감소
  윈도우 자원을 유지하는 것은 시스템에 부하를 준다. 간단한 비교를 위해 윈도우 폼 응용 프로그램에서 1,000개의 버튼 컨트롤을 자식으로 갖게 만들고, 동일하게 1,000개의 버튼 요소가 있는 WPF 응용 프로그램을 만들어 각각 실행해 보자. 후자의 WPF 응용 프로그램이 좀 더 빨리 실행되는 것을 확인할 수 있다. 왜냐하면 WPF는 1,000개의 버튼 모양을 그리는 것에 불과한 반면, 윈도우 폼 응용 프로그램은 운영체제로부터 1,000개의 윈도우 자원을 생성하도록 요청을 한 후에 그 안에 다시 그리는 작업을 하기 때문이다.

- 자유로운 UI 표현 가능
  메모리에 그려진 이미지가 조합돼 표현되기 때문에 요소 간의 중첩된 이미지 처리가 가능해져 자유로운 렌더링 효과를 얻는다.

한 가지 분명한 점은 WPF 응용 프로그램에서는 기존의 윈도우 폼과 비교했을 때 컨트롤의 외양을 바꾸는 것이 매우 자유롭다는 점이다. 이로 인해 대체로 WPF용 컨트롤이 윈도우 폼용 컨트롤에 비해 더욱 화려하다는 특징이 있다.

19.2.2 데이터 바인딩

바인딩(Binding)이란 일반적으로 어떤 대상과 묶이는 것을 의미한다. WPF에서의 데이터 바인딩은 데이터를 담고 있는 코드와 UI 요소가 연결되는 것을 의미한다. 사실 데이터 바인딩이 없었어도 우리는 그동안 정상적으로 데이터를 화면에 표시해 왔다.

예를 들어, 화면에 시간을 보여주는 WPF 응용 프로그램을 만들어 보자.

예제 20.11 WPF로 만드는 시계 프로그램

```
// ================== MainWindow.xaml ============================
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="200" Width="300">
    <Grid>
        <Label x:Name="lblTime" VerticalAlignment="Center"
               HorizontalAlignment="Center"></Label>
    </Grid>
</Window>
```

```
// ================== MainWindow.xaml.cs ============================
using System;
```

```
using System.Windows;
using System.Windows.Threading;

namespace WpfApplication1
{
    public partial class MainWindow : Window
    {
        DispatcherTimer _timer;

        public MainWindow()
        {
            InitializeComponent();

            _timer = new DispatcherTimer();
            _timer.Tick += _timer_Tick;
            _timer.Interval = new TimeSpan(0, 0, 1); // 1초 마다 Tick 이벤트 발생
            _timer.Start();
        }

        void _timer_Tick(object sender, EventArgs e)
        {
            this.lblTime.Content = DateTime.Now.ToLongTimeString();
        }
    }
}
```

윈도우 폼과 마찬가지로 Label 컨트롤을 사용했고, x:Name 속성에 이름을 주면, xaml.cs 코드 파일에서 그 이름으로 해당 컨트롤에 접근할 수 있다. MainWindow.xaml.cs 파일에는 DispatcherTimer를 이용해 지정된 Interval 시간마다 Tick 이벤트를 발생시키게 했고, 따라서 _timer_Tick 이벤트 처리기는 1초마다 실행되어 lblTime.Content 속성에 현재 시간을 설정한다. 한 가지 특이한 면이 있다면 윈도우 폼에서는 Label에 보여줄 텍스트를 Text 속성을 이용해 지정했던 반면, WPF의 Label에서는 Content 속성으로 바뀌었다는 정도의 차이가 있다.

이것이 바로 데이터 바인딩을 사용하지 않고 UI 요소와 코드의 실행을 연결한 경우다. 이제 데이터 바인딩을 이용하는 예제로 코드를 바꿔보자. 우선 xaml 파일은 다음과 같이 바뀐다.

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Name="thisWindow"
        Title="MainWindow" Height="200" Width="300">
    <Grid>
        <Label DataContext="{Binding ElementName=thisWindow}"
               Content="{Binding Path=Time}"
               VerticalAlignment="Center"
               HorizontalAlignment="Center"></Label>
    </Grid>
</Window>
```

Window 태그에 x:Name을 지정하고, Label 컨트롤에서는 DataContext 속성을 이용해 "{Binding}" 설정을 했는데, 연결될 ElementName의 대상으로 Window의 x:Name에 지정된 값을 전달했다. 그

리고는 Content 속성에도 역시 Binding 구문을 이용해 Path로 Time이라는 문자열을 전달한다.

궁금한 것이 많겠지만 일단 xaml.cs 파일까지 변경해야만 설명을 계속할 수 있다.

```
using System;
using System.ComponentModel;
using System.Windows;
using System.Windows.Threading;

namespace WpfApplication1
{
    public partial class MainWindow : Window, INotifyPropertyChanged
    {
        DispatcherTimer _timer;

        string _time;
        public string Time
        {
            get { return _time; }
            set
            {
                _time = value;
                PropertyChanged(this, new PropertyChangedEventArgs("Time"));
            }
        }

        public MainWindow()
        {
            InitializeComponent();

            _timer = new DispatcherTimer();
            _timer.Tick += _timer_Tick;
            _timer.Interval = new TimeSpan(0, 0, 1); // 1초 마다 발생
            _timer.Start();
        }

        void _timer_Tick(object sender, EventArgs e)
        {
            this.Time = DateTime.Now.ToLongTimeString();
        }

        public event PropertyChangedEventHandler PropertyChanged;
    }
}
```

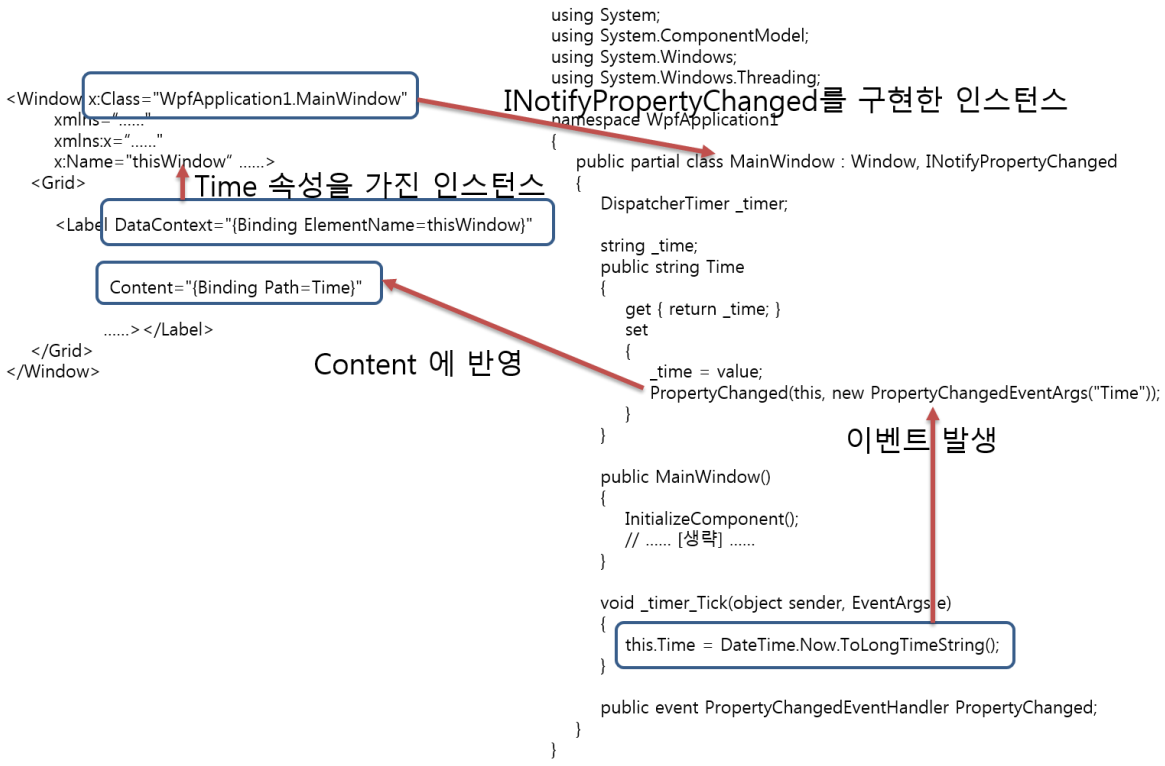
우선 _timer_Tick 메서드를 보면 단순히 값을 Time 속성 값에 넣기만 한다. 그리고 Time 속성의 set 구문을 보면 특이하게 PropertyChanged라는 이벤트를 "Time" 문자열로 초기화한 PropertyChangedEventArgs 인스턴스와 함께 발생시킨다.

이 예제를 실행해 보면 예제 20.11과 동일하게 수행되는 것을 확인할 수 있다. 어째서 그런 것일까? 그 이면에는 바로 WPF의 데이터 바인딩이 있기 때문이다. 보다시피 xaml.cs 파일에서는 Time 속성 값이 변경될 때마다 INotifyPropertyChanged 인터페이스의 PropertyChanged 이벤트를

발생시킨다. 한편, xaml에 정의된 Label 요소에서는 이를 Content="{Binding Path=Time}"이라는 구문을 통해 PropertyChanged 이벤트를 기다리다가 그 인자에 Path로 지정된 "Time"이라는 문자열과 일치하는 경우 그 값을 가져다가 Content 속성에 대입하는 역할을 한다.

그런데 Label은 Time이라는 속성값을 어디서 가져와야 하는지 알 수 있을까? 이를 위해 필요한 설정 값이 바로 DataContext다. Label의 DataContext에는 {Binding} 구문으로 ElementName의 대상이 thisWindow로 지정됐고, xaml 내에서는 thisWindow로 식별되는 요소가 <Window x:Name="thisWindow" ..... />이기 때문에 MainWindow 인스턴스가 선택되는 것이다. 따라서 코드상에서 직접 Label.Content 속성에 값을 넣지 않아도 데이터 바인딩 구문을 통해 코드의 속성 값을 변경하는 것만으로도 그림 20.42처럼 UI 요소까지 값이 전달될 수 있는 것이다.

그림 20.42 WPF 데이터 바인딩



DataContext 속성 값은 상위 요소에서 하위 요소로 전파되기 때문에 Label에 직접 지정하지 않고 Window 요소나 Grid에 지정해도 이 예제에서는 동일한 실행 결과를 얻을 수 있다.

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Name="thisWindow"
        DataContext="{Binding ElementName=thisWindow}"
        Title="MainWindow" Height="200" Width="300">
    <Grid>
        <Label Content="{Binding Path=Time}"
               VerticalAlignment="Center"
```

```
            HorizontalAlignment="Center"></Label>
    </Grid>
</Window>
```

위와 같이 변경하는 경우 Window 요소는 DataContext의 Binding 대상으로 자기 자신을 가리키게 되는데, WPF에서는 이런 상황에서 사용할 수 있는 RelativeSource 구문을 지원한다. 최종적으로 다음과 같은 관용적인 표현을 통해 xaml.cs에 정의된 클래스의 인스턴스를 가리키도록 DataContext를 지정할 수 있다.

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        DataContext="{Binding RelativeSource={RelativeSource Self}}"
        Title="MainWindow" Height="200" Width="300">
    <Grid>
        <Label
              Content="{Binding Path=Time}"
              VerticalAlignment="Center"
              HorizontalAlignment="Center"></Label>
    </Grid>
</Window>
```

아직까지는 이렇게 복잡한 데이터 바인딩을 사용하기보다는 예제 20.11처럼 직접 UI 요소의 x:Name을 통해 UI 요소와 상호 연동하는 방식이 더 편하다고 느껴질 것이다. 하지만 데이터 바인딩에 대해 조금만 더 알아보자. 사실, 데이터 바인딩의 진짜 매력을 알려면 Label과 같은 요소보다는 TextBox처럼 편집이 가능한 예제를 작성해 봐야 한다. 예를 들어, xaml에 Button과 TextBox를 넣고 Button이 눌린 경우 TextBox의 내용을 출력하는 예제를 다음과 같이 작성할 수 있다.

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        DataContext="{Binding RelativeSource={RelativeSource Self}}"
        Title="MainWindow" Height="200" Width="300">
    <Grid>
        <Button Content="Button" Click="Button_Click"
               HorizontalAlignment="Left" Margin="207,136,0,0"
               VerticalAlignment="Top" Width="75" Height="23"/>
        <TextBox Text="{Binding Path=InputText}"
                HorizontalAlignment="Left" Height="23"
                Margin="10,136,0,0" TextWrapping="Wrap"
                VerticalAlignment="Top" Width="192"/>
    </Grid>
</Window>
```

```
using System;
using System.ComponentModel;
using System.Windows;
using System.Windows.Threading;
```

```
namespace WpfApplication1
{
    public partial class MainWindow : Window, INotifyPropertyChanged
    {
        string _inputText;
        public string InputText
        {
            get { return _inputText; }
            set
            {
                _inputText = value;
                PropertyChanged(this, new PropertyChangedEventArgs("InputText"));
            }
        }

        public MainWindow()
        {
            InitializeComponent();
        }

        public event PropertyChangedEventHandler PropertyChanged;

        private void Button_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show(this.InputText);
        }
    }
}
```

TextBox의 Text 속성을 MainWindow 타입의 InputText 속성과 데이터 바인딩시키고 있는데, 이를 통해 값의 전파가 쌍방향으로 이뤄진다. 즉, TextBox에서 값을 입력하면 MainWindow의 InputText 변수에 그 결과가 반영된다. 따라서 Button_Click 이벤트 처리기에서는 TextBox로부터 Text 값을 가져올 필요 없이 곧바로 InputText 속성을 사용함으로써 사용자가 입력한 텍스트에 접근할 수 있다.

게다가 데이터 바인딩은 xaml 요소 간에도 적용할 수 있다. 예를 들어, TextBox에서 입력하는 내용을 Label에 곧바로 출력하고 싶다면 부가적인 코드 없이 단순히 다음과 같은 xaml만으로도 표현할 수 있다.

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="200" Width="300">
    <Grid>
        <TextBox x:Name="txtBox"
                 HorizontalAlignment="Left" Height="23"
                 Margin="10,136,0,0" TextWrapping="Wrap"
                 VerticalAlignment="Top" Width="272"/>
        <Label Content="{Binding ElementName=txtBox, Path=Text}"
               HorizontalAlignment="Left" Margin="10,105,0,0"
               VerticalAlignment="Top" Width="272"/>
    </Grid>
</Window>
```

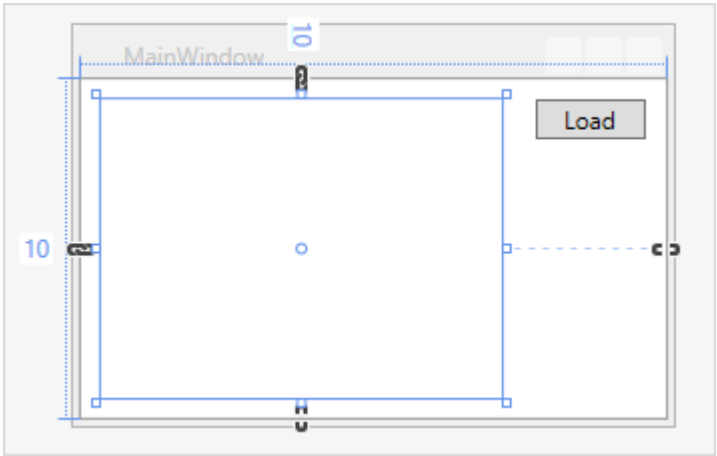
위의 xaml을 보면 Label.Content 속성에는 그 위의 TextBox의 x:Name 값으로 Binding 대상이 지정돼 있으며, Path 값으로는 TextBox 타입이 가진 Text 속성을 지정했다. 따라서 TextBox에 사용자가 내용을 입력할 때마다 그 결과가 곧바로 Label의 Content에 연결되어 화면에 출력된다. 여기서 한 가지 중요한 점은 이 기능을 코드 한 줄 없이 구현했다는 것이다.

지금까지 데이터 바인딩을 사용해 봤는데 쓸만하다고 느껴졌을는지 모르겠다. 처음에는 INotifyPropertyChanged 인터페이스를 구현하는 것이 번거로울 수 있지만 일단 속성에 PropertyChanged 이벤트가 적용만 되면 데이터 바인딩을 이용해 xaml 디자인 요소와의 상호 연동이 물 흐르듯이 자연스럽게 이뤄진다는 매력이 있다.

19.2.3 레이아웃을 위한 패널

여기서 레이아웃(layout)은 컨트롤의 배치를 의미한다. 고정된 윈도우 화면 크기에서의 컨트롤 배치는 단순히 x, y 위치 값을 고정시켜 지정할 수 있지만, 문제는 "유동적인 윈도우 화면 크기"에서 발생한다. 이번에는 예제로 이미지 뷰어 프로그램을 WPF로 만들어 보자. 그림 20.43처럼 디자인은 간단하게 파일을 선택할 수 있는 Button과 이미지를 출력하는 Image 컨트롤로 구성하고

그림 20.43 Image 컨트롤과 Button 컨트롤로 구성된 이미지 뷰어



예제 20.12 컨트롤의 위치를 고정

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        DataContext="{Binding RelativeSource={RelativeSource Self}}"
        Title="MainWindow" Height="200" Width="300">
    <Grid>
        <Image Source="{Binding Path=BitmapImage}"
               HorizontalAlignment="Left" Height="149"
               Margin="10,10,0,0" VerticalAlignment="Top"
```

```
            Width="200"/>
        <Button Content="Load" Click="Button_Click"
                HorizontalAlignment="Left" Margin="227,10,0,0"
                VerticalAlignment="Top" Width="55"/>
    </Grid>
</Window>
```

예제 20.12와 같이 Button 컨트롤에 Click 이벤트를 연결하는 XAML 코드를 완성한다. 그런 다음, 예제 20.13과 같이 Button_Click 이벤트 처리기에 OpenFileDialog를 띄워 그림 파일을 선택할 수 있게 한 후, 이 파일을 Image 컨트롤의 Source 속성에 데이터 바인딩시킨 BitmapImage에 지정한다.

예제 20.13 데이터 바인딩으로 연결된 BitmapImage 속성에 그림 파일을 지정

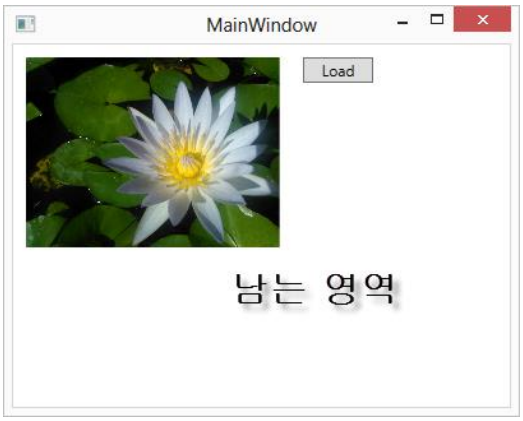
```
using System;
using System.ComponentModel;
using System.IO;
using System.Windows;
using System.Windows.Media.Imaging;
using System.Windows.Threading;
using Microsoft.Win32;

namespace WpfApplication1
{
    public partial class MainWindow : Window, INotifyPropertyChanged
    {
        BitmapImage _bitmapImage;
        public BitmapImage BitmapImage
        {
            get { return _bitmapImage; }
            set
            {
                _bitmapImage = value;
                OnPropertyChanged("BitmapImage");
            }
        }

        public MainWindow()
        {
            InitializeComponent();
        }

        public event PropertyChangedEventHandler PropertyChanged;

        protected virtual void OnPropertyChanged(string name)
        {
            if (string.IsNullOrEmpty(name) == true)
            {
                return;
            }

            if (PropertyChanged != null)
            {
                PropertyChangedEventArgs arg = new PropertyChangedEventArgs(name);
                PropertyChanged(this, arg);
```

```
            }
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
            OpenFileDialog ofd = new OpenFileDialog();
            if (ofd.ShowDialog() == true)
            {
                string txt = ofd.FileName;
                this.BitmapImage = new BitmapImage(new Uri(txt));
            }
        }
    }
}
```

우선, OpenFileDialog가 '20.1 Windows Forms 응용 프로그램' 절에서 배운 것과는 이름만 같을 뿐 다른 타입이라는 사실을 알 필요가 있다. 윈도우 폼 응용 프로그램이 사용하는 System.Windows.Forms 네임스페이스에 포함된 OpenFileDialog 타입을 이용하려면 별도로 System.Windows.Forms.dll 어셈블리를 포함시켜야 하는데 WPF 응용 프로그램에 또 다시 윈도우 폼 응용 프로그램을 위한 어셈블리를 내장하는 것은 성능면에서 그다지 바람직하지 않다. 이러한 이유로 마이크로소프트에서는 WPF 응용 프로그램의 기본 참조 어셈블리인 PresentationFramework.dll에 Microsoft.Win32 네임스페이스로 OpenFileDialog를 추가했다. WPF와 윈도우 폼 응용 프로그램은 닷넷으로 GUI 응용 프로그램을 만드는 서로 다른 방법을 제공하기 때문에 이처럼 동일한 기능이 양측에 제공되는 경우가 종종 있다는 점을 알아 두자.

이제 프로그램을 실행하고 "Load" 버튼을 눌러 여러분의 하드디스크에 저장된 그림을 선택하면 Image 컨트롤의 영역에 맞게 이미지가 축소/확대되어 나타나는 것을 확인할 수 있다. 그런데 이미지를 크게 보고 싶어서 윈도우를 확장하면 어떻게 될까? XAML 내부에 지정된 컨트롤의 위치가 고정돼 있으므로 그림 20.44처럼 공간 낭비가 발생한다.

그림 20.44 윈도우 크기 조정에 영향을 받지 않는 내부 컨트롤



이런 문제를 해결하기 위해 Grid 레이아웃 컨트롤을 활용할 수 있다. 방법은 윈도우에 상관없이 좌측의 "Load" 버튼에 대해서만 고정된 크기를 할당하고 이미지 영역을 자유롭게 축소/확장되게

끔 지정하면 된다. 다음은 이를 반영한 XAML 코드다.

예제 20.14 컨트롤의 위치를 Grid 레이아웃에 맡긴 XAML

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        DataContext="{Binding RelativeSource={RelativeSource Self}}"
        Title="MainWindow" Height="200" Width="300">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="70" />
        </Grid.ColumnDefinitions>
        <Image Grid.Column="0"
               Source="{Binding Path=BitmapImage}"
               HorizontalAlignment="Left"
               VerticalAlignment="Top"/>
        <Button Grid.Column="1"
                Content="Load" Click="Button_Click"
                HorizontalAlignment="Left"
                VerticalAlignment="Top" Width="55"/>
    </Grid>
</Window>
```

예제 20.14에서는 수직으로 영역을 분할하는 ColumnDefinition만 사용하는데, Grid는 수평 분할에 대한 RowDefinition도 지원하므로 상황에 맞게 사용하면 된다. 사실, 예제 20.14는 1개의 Row를 가지고 있는 것이나 마찬가지이기 때문에 다음과 같이 정의해도 무방하다.

```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="70" />
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
    <RowDefinition Height="*" />
</Grid.RowDefinitions>
```

0번째 칼럼(ColumnDefinition)의 Width 값이 별표(*)로 지정된 것에 유의하자. Width 값에는 double 형의 숫자값 외에 '*', 'Auto' 값이 예외적으로 허용된다. '*'로 설정된 경우 나머지 전체 크기를 의미하며, 'Auto'로 설정된 경우 해당 칼럼에 위치하는 자식 컨트롤들의 크기로 자동 지정된다.

이렇게 해서 2개의 수직 영역이 정의됐으므로 개별 컨트롤이 어느 영역에 들어가야 할지를 지정할 필요가 있다. 이런 목적으로 Grid.Column 속성이 Image 컨트롤에는 0을 지정해 0번째 칼럼에 들어가게 하고, Button 컨트롤에는 1을 지정해 1번째 칼럼에서 보이게 만들었다. 따라서 예제 20.14는 Button 컨트롤이 위치한 1번째 칼럼으로 70이라는 크기가 할당되고 그 밖의 나머지 모든 크기 값이 0번째 칼럼에 할당된다. 고정 크기를 지정한 예제 20.12와 함께 Grid 레이아웃 컨트롤을 이용해 가변 크기를 지원하는 예제 20.14를 각각 실행한 후 윈도우 크기를 마우스로 변경해

보면 지금까지 설명한 내용을 좀 더 쉽게 이해할 수 있다.

WPF에는 Grid 외에 Canvas, UniformGrid, DockPanel, StackPanel, WrapPanel 같은 기본 레이아웃 컨트롤이 제공되며, 이것들은 모두 공통적으로 System.Windows.Controls.Panel 타입을 상속받는다. 일반적으로 Panel 상속 컨트롤들은 또 다른 컨트롤을 담고 있는 역할을 하기 때문에 "컨테이너 컨트롤"이라고 한다.

참고!	이름은 다르지만 레이아웃 컨트롤은 윈도우 폼 응용 프로그램에서도 유사한 방식으로 제공된다.

19.2.4 Content 속성

레이아웃에 사용되는 "컨테이너 컨트롤"은 내부에 2개 이상의 컨트롤을 포함시키는 것이 가능하다. 하지만 일반적인 다른 컨트롤은 그것의 부모인 ContentControl 타입에서 제공하는 Content 속성에 단일 컨트롤을 담는 것만 허용된다.

Content 속성의 타입은 object인데, 여기에 WPF의 UI 요소가 아닌 다른 타입에 해당하는 인스턴스를 지정하면 자동으로 해당 객체의 ToString 메서드를 호출한 문자열을 보여준다. 다음 예제를 보자.

```
<Window
        ……[생략]……
        DataContext="{Binding RelativeSource={RelativeSource Self}}"
        Title="MainWindow" Height="200" Width="300">
        <Grid>
            <Button Content="{Binding SampleText}"
                HorizontalAlignment="Left" Margin="10,10,0,0"
                VerticalAlignment="Top" Width="156" Height="23"/>
            <Button Content="{Binding SampleDate}"
                HorizontalAlignment="Left" Margin="10,38,0,0"
                 VerticalAlignment="Top" Width="156" Height="23"/>
            <Button Content="{Binding SampleControl}"
                HorizontalAlignment="Left" Margin="10,66,0,0"
                VerticalAlignment="Top" Width="156" Height="23"/>
    </Grid>
</Window>
```

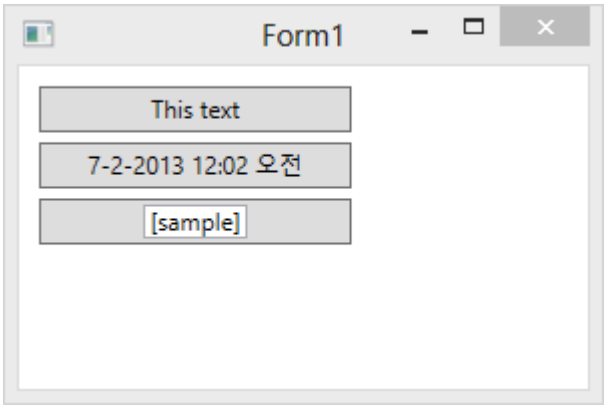
예제 20.15 Content 속성 사용 예

```
using System;
using System.ComponentModel;
using System.Windows;
using System.Windows.Controls;

namespace WpfApplication1
{
```

```
    public partial class MainWindow : Window, INotifyPropertyChanged
    {
        string _sampleText;
        public string SampleText
        {
            get { return _sampleText; }
            set
            {
                _sampleText = value;
                OnPropertyChanged("SampleText");
            }
        }

        DateTime _sampleDate;
        public DateTime SampleDate
        {
            get { return _sampleDate; }
            set
            {
                _sampleDate = value;
                OnPropertyChanged("SampleDate");
            }
        }

        UIElement _sampleControl;
        public UIElement SampleControl
        {
            get { return _sampleControl; }
            set
            {
                _sampleControl = value;
                OnPropertyChanged("SampleControl");
            }
        }

        public MainWindow()
        {
            InitializeComponent();

            this.SampleText = "This text";
            this.SampleDate = DateTime.Now;

            TextBox txtBox = new TextBox();
            txtBox.Text = "[sample]";
            this.SampleControl = txtBox;
        }

        // PropertyChanged / OnPropertyChanged 구현 생략: 예제 20.13 참조
    }
}
```

예제 20.15를 실행하면 그림 20.45와 같은 결과를 확인할 수 있다.

그림 20.45 Button의 Content에 지정된 string, DateTime, UIElement 인스턴스



첫 번째 버튼은 단순히 string 타입인 SampleText의 내용을 그대로 출력하고, 두 번째 버튼은 DateTime 인스턴스이므로 ToString 메서드를 호출한 결과를 Content에 보여준다. 특이한 것은 세 번째 버튼이다. UIElement를 상속받은 TextBox 컨트롤을 Content 속성에 지정함으로써 마치 자식 컨트롤을 포함하고 있는 것과 같은 효과를 낸다.

Content 속성은 XAML에서도 지정할 수 있는데, 문법은 <*컨트롤타입이름*.Content>로 설정한다. 예제 20.15의 세 번째 버튼을 XAML에서 다음과 같이 동일하게 표현할 수 있다.

```
<Button
    HorizontalAlignment="Left" Margin="10,94,0,0"
    VerticalAlignment="Top" Width="156" Height="31">
    <Button.Content>
        <TextBox Text="[sample]"></TextBox>
    </Button.Content>
</Button>
```

또는 좀 더 간단하게, Content가 기본 속성이므로 아예 생략하는 것도 가능하다.

```
<Button
    HorizontalAlignment="Left" Margin="10,94,0,0"
    VerticalAlignment="Top" Width="156" Height="31">
    <TextBox Text="[sample]"></TextBox>
</Button>
```

위의 구문을 이해한다면 WPF 프로젝트를 처음 생성했을 때 기본 생성되는 MainWindow.xaml의 구조도 이제 파악할 수 있을 것이다.

```
<Window x:Class="WpfApplication2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

Window 타입에는 Content 속성이 제공되며, 위의 코드는 <Window.Content>가 생략된 채로 하위에 Grid 패널을 포함하고 있는 것이다.

Content 속성은 1개의 자식 요소만 포함할 수 있지만, Panel 타입이 UIElement를 상속받은 것임을 감안한다면 이를 활용해 다중 컨트롤을 Content에 지정하는 것도 가능하다. 다음은 Button 내부에 CheckBox와 Label 컨트롤 두 개를 Panel을 이용해 포함하는 방법을 보여준다.

```
<Button
    HorizontalAlignment="Left" Margin="10,94,0,0"
    VerticalAlignment="Top" Width="156" Height="31">
    <Button.Content>
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="20"></ColumnDefinition>
                <ColumnDefinition Width="*"></ColumnDefinition>
            </Grid.ColumnDefinitions>
            <CheckBox VerticalAlignment="Center"></CheckBox>
            <Label Grid.Column="1">Button with CheckBox</Label>
        </Grid>
    </Button.Content>
</Button>
```

컨트롤 내부에 포함하는 내용으로 Content 속성이 항상 사용되는 것은 아니다. 가령 TextBox의 경우는 명시적으로 Text 속성을 통해 내부의 "편집 문자열"을 지정한다. 즉, Content 속성은 UIElement까지 자식으로 담을 필요가 있을 때 사용하는 반면, 그 밖의 특정 타입으로 값이 고정되는 경우에는 그에 맞는 이름의 속성을 정의해 두는 것이 일반적인 관례다.

19.2.5 Padding과 Margin

컨트롤의 위치를 세밀하게 제어하려면 Padding과 Margin 속성의 차이점을 알아둘 필요가 있다. 이 용어들은 윈도우 폼 응용 프로그램 및 HTML 웹 페이지의 CSS에도 동일하게 적용되므로 이번 기회에 알고 넘어가면 도움될 것이다.

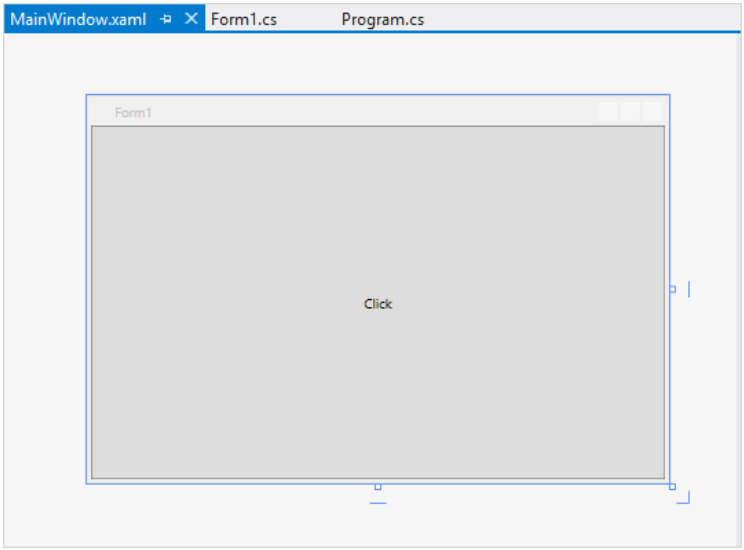
이해를 돕기 위해 비주얼 스튜디오의 XAML 디자인 화면을 이용해 보자. 우선, MainWindow.xaml 디자인 화면에 Button 하나를 추가한다.

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Test" Height="350" Width="525">
    <Grid>
        <Button>Click</Button>
    </Grid>
</Window>
```

그럼 화면에는 Window 요소 하위에 Grid 요소를 채우고, 이어서 Button 컨트롤을 Grid에 채워넣

는다(그림 20.46 참고).

그림 20.46 Window, Grid, Button 컨트롤 배치

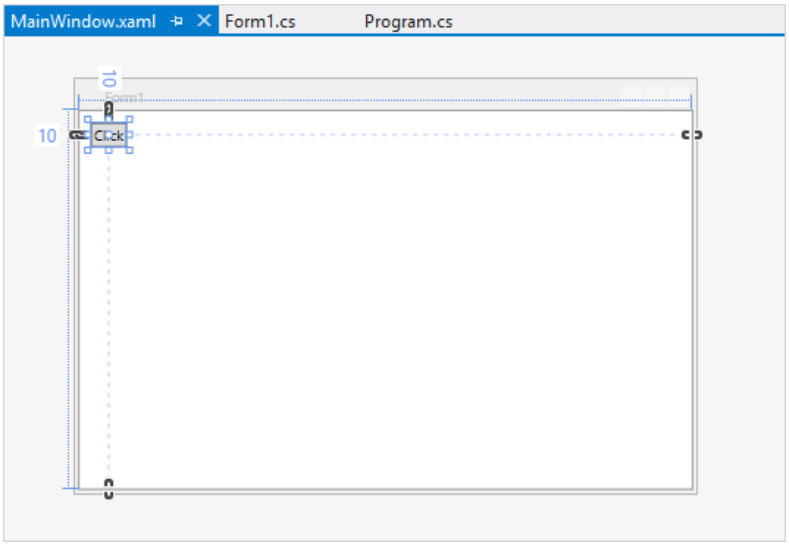


이 상태에서 Button 요소에 Margin과 HorizontalAlignment, VerticalAlignment를 설정해 보자.

```
<Button
    HorizontalAlignment="Left"
    VerticalAlignment="Top"
    Margin="10, 10, 0, 0">Click</Button>
```

Margin의 4개 숫자는 해당 컨트롤의 부모 요소로부터 각 4면에서 얼마나 떨어져 배치할 것인지를 지정한다. 각각 "[좌] [위] [우] [아래]"를 의미하며, 따라서 예제에서는 "10, 10, 0, 0"으로 지정했으므로 그림 20.47처럼 Grid의 좌로부터 10, 위로부터 10 단위만큼 Button 컨트롤을 떨어뜨리게 된다.

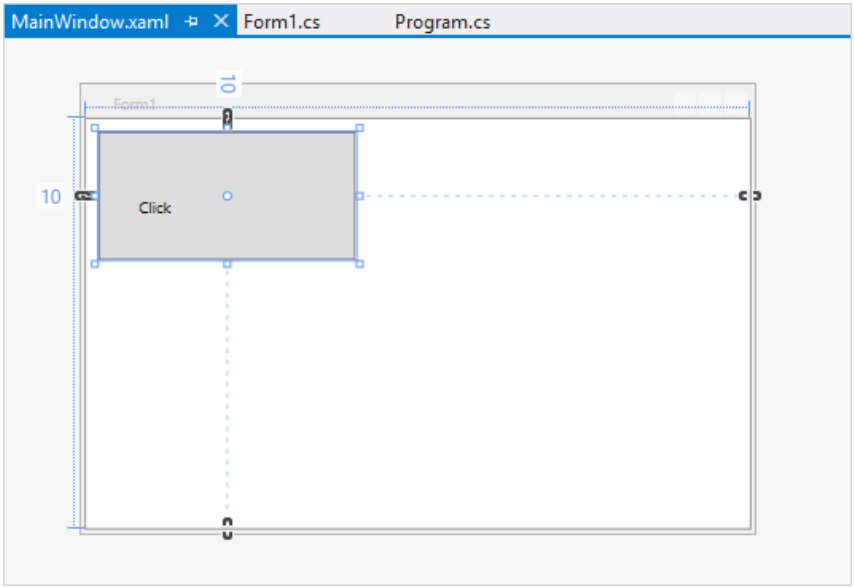
그림 20.47 Margin="10, 10, 0, 0"을 지정한 컨트롤 위치



HorizontalAlignment와 VerticalAlignment를 지정하지 않으면 기본적으로 중앙(Center) 정렬이 되기 때문에 10 정도의 Margin 값은 거의 차이가 없다.

이제 Padding 값을 지정할 차례다. 이 역시 4개의 값을 줄 수 있는데, Margin과 마찬가지로 "[좌] [위] [우] [아래]"를 의미하며, 이를 지정해 해당 컨트롤과 그것이 소유하는 Content 요소 간의 간격을 제어할 수 있다. 예를 들어, 그림 20.47의 Button에 Width=200, Height=100를 지정해 크기를 좀 더 키운 후, HorizontalContentAlignment="Left", VerticalContentAlignment="Top"과 함께 Padding 값을 주면 Content에 해당하는 "Click" 텍스트가 버튼 컨트롤 내에서 그 만큼의 간격을 유지한 상태로 출력되는 모습을 확인할 수 있다(그림 20.48 참고).

그림 20.48 Padding="30, 50, 0, 0"을 지정한 컨트롤



```
<Button
    HorizontalAlignment="Left"
    VerticalAlignment="Top"
    Margin="10, 10, 0, 0"
    HorizontalContentAlignment="Left"
    VerticalContentAlignment="Top"
    Width="200" Height="100"
    Padding="30, 50, 0, 0">Click</Button>
```

HorizontalContentAlignment와 VerticalContentAlignment의 효과는 Content에 영향을 미친다는 점을 제외하면 각각 HorizontalAlignment와 VerticalAlignment의 역할과 동일하다. 마찬가지로 기본값 역시 중앙 정렬이기 때문에 Left, Top으로 지정하지 않으면 Padding의 효과가 의도한대로 나타나지 않는다.

간단히 정리하면 Margin은 컨트롤과 그 컨트롤의 부모와의 간격을 제어하는 데 사용하고, Padding은 그 컨트롤이 소유한 Content와의 간격을 제어하는 용도로 쓰인다.

19.2.6 ItemsControl 타입

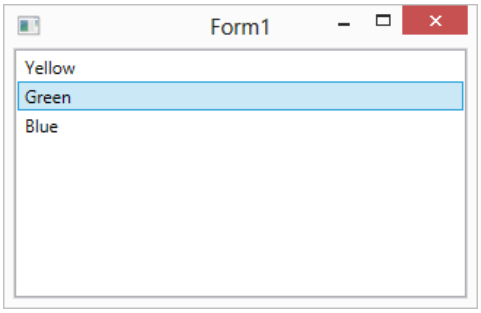
WPF의 진수를 맛보려면 ItemsControl을 살펴봐야 한다. 일반적으로 ItemsControl은 컬렉션에 담긴 데이터를 보여주는 용도로 사용한다. 즉, "목록"을 보여주는 역할을 하는 가장 기본적인 코드를 WPF에서는 ItemsControl 타입에 정의해 두고, 그것을 상속받은 여러 컨트롤을 이용해 쉽게 목록을 화면에 보여줄 수 있다.

ItemsControl을 상속받은 컨트롤들은 많지만 여기서는 그 중에서도 자주 사용되는 ListBox 컨트롤의 사용법을 예로 든다. 우선, 기본적으로 3개의 항목을 보여주는 ListBox는 예제 20.16의 XAML 코드로 구성되고, 이를 실행한 모습은 그림 20.49와 같다.

예제 20.16 ListBox 기본 사용 예

```
<ListBox>
    <ListBoxItem>Yellow</ListBoxItem>
    <ListBoxItem>Green</ListBoxItem>
    <ListBoxItem>Blue</ListBoxItem>
</ListBox>
```

그림 20.49 기본 ListBox를 실행한 화면



여기서 중요한 것은 ListBoxItem의 내부에 지정된 값이 다름 아닌 Content라는 점이다.

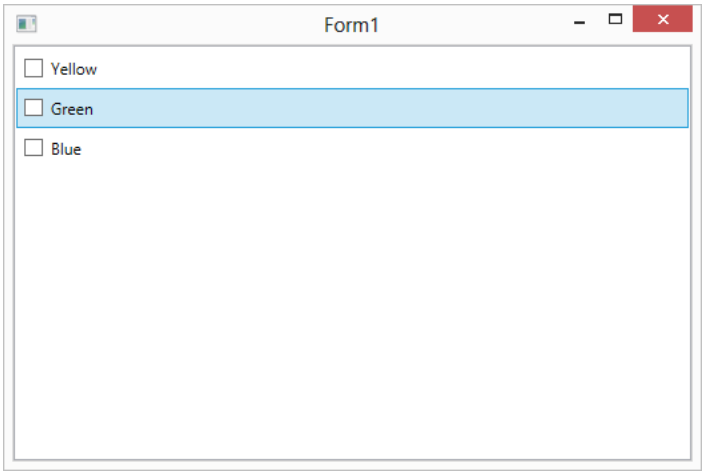
```
<ListBoxItem>
    <ListBoxItem.Content>Yellow</ListBoxItem.Content>
</ListBoxItem>
```

Content의 사용법을 잘 이해하고 있다면 여기서 더 재미있는 표현도 가능하다는 것을 짐작할 수 있다. 예를 들어, 예제 20.17처럼 각 항목의 좌측에 CheckBox를 넣는 것도 가능하다.

예제 20.17 ListBoxItem의 Content에 다중 컨트롤 설정

```
<ListBox>
    <ListBoxItem>
        <ListBoxItem.Content>
            <WrapPanel>
                <CheckBox VerticalAlignment="Center"></CheckBox>
                <Label>Yellow</Label>
            </WrapPanel>
        </ListBoxItem.Content>
    </ListBoxItem>
    <ListBoxItem>
        <ListBoxItem.Content>
            <WrapPanel>
                <CheckBox VerticalAlignment="Center"></CheckBox>
                <Label>Green</Label>
            </WrapPanel>
        </ListBoxItem.Content>
    </ListBoxItem>
    <ListBoxItem>
        <ListBoxItem.Content>
            <WrapPanel>
                <CheckBox VerticalAlignment="Center"></CheckBox>
                <Label>Blue</Label>
            </WrapPanel>
        </ListBoxItem.Content>
    </ListBoxItem>
</ListBox>
```

그림 20.50 ListBoxItem 내부에 CheckBox와 Label을 포함



ListBoxItem을 XAML에 직접 사용하는 대신, 코드에서 정의한 컬렉션 변수를 ListBox XAML에 데이터 바인딩시켜 적용하는 것도 가능하다. ListBox는 목록을 보여주는 컨트롤이기 때문에 컬렉션 타입을 받는 ItemsSource 속성을 제공하며, 이를 이용해 바인딩을 설정한다.

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Form1" Height="350" Width="525"
        DataContext="{Binding RelativeSource={RelativeSource Self}}">
    <Grid>
        <ListBox ItemsSource="{Binding Path=ColorList}">
        </ListBox>
    </Grid>
</Window>
```

그럼 당연히 xaml.cs 코드 파일에는 ColorList 속성을 제공해야 한다.

```
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Windows;

namespace WpfApplication1
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            _colorList.Add("Yellow");
            _colorList.Add("Green");
            _colorList.Add("Blue");
        }

        ObservableCollection<string> _colorList =
                                        new ObservableCollection<string>();
```

```
        public ObservableCollection<string> ColorList
        {
            get { return _colorList; }
        }
    }
}
```

그런데 이번에는 INotifyPropertyChanged 인터페이스가 MainWindow에 구현되지 않았다. _colorList 값은 MainWindow 인스턴스가 생성되는 순간부터 변하지 않고 고정되기 때문에 굳이 INotifyPropertyChanged 인터페이스 처리를 할 필요가 없는 것이다(물론 구현해도 무방하다). 대신 ColorList의 타입을 일반적인 List<T> 자료형을 사용하지 않고 ObservableCollection<T> 타입으로 대체하고 있다. 이것은 ColorList 컬렉션의 항목이 추가/삭제될 때마다 바인딩된 ListView 컨트롤에 통지를 보내야 하는데, List<T> 타입은 이것이 구현돼 있지 않기 때문이다. 반면 ObservableCollection<T> 타입은 컬렉션에 항목이 추가/삭제될 때마다 그것에 대한 변경사항을 통지하는 INotifyCollectionChanged 인터페이스를 상속받아 구현한 것으로 WPF의 데이터 바인딩을 위해 닷넷 프레임워크 3.0 BCL에 함께 포함됐다. 이렇게 구현된 상태에서 프로그램을 실행하면 그림 20.49와 같은 모습을 볼 수 있다.

말로만 설명한 ObservableCollection<T>의 동작을 확인하기 위해 Button 컨트롤 2개를 추가하고 각 이벤트 처리기에 ColorList 컬렉션에 항목을 더하고 빼는 기능을 구현해 보자.

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Form1" Height="350" Width="525"
        DataContext="{Binding RelativeSource={RelativeSource Self}}">
    <Grid>
        <ListBox ItemsSource="{Binding Path=ColorList}" Margin="0,0,0,35">
        </ListBox>
        <Button Content="Add" HorizontalAlignment="Left" Margin="10,289,0,0"
                VerticalAlignment="Top" Width="75" Click="Add_Click"/>
        <Button Content="Remove" HorizontalAlignment="Left" Margin="90,289,0,0"
                VerticalAlignment="Top" Width="75" Click="Remove_Click"/>
    </Grid>
</Window>
```

예제 20.18 XAML에 바인딩된 컬렉션의 항목 추가/삭제

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Reflection;
using System.Windows;
using System.Windows.Media;

namespace WpfApplication1
{
```

```
    public partial class MainWindow : Window
    {
        List<string> _sampleColors = new List<string>();

        public MainWindow()
        {
            InitializeComponent();

            _colorList.Add("Yellow");
            _colorList.Add("Green");
            _colorList.Add("Blue");

            // Colors 타입에 static 속성으로 정의된 컬러 값을
            // 리플렉션을 이용해 가져온다.
            foreach (PropertyInfo propInfo in typeof(Colors).GetProperties(
                    BindingFlags.Static | BindingFlags.Public))
            {
                _sampleColors.Add(propInfo.Name);
            }
        }

        ObservableCollection<string> _colorList =
                                new ObservableCollection<string>();
        public ObservableCollection<string> ColorList
        {
            get { return _colorList; }
        }

        private void Add_Click(object sender, RoutedEventArgs e)
        {
            // _sampleColors에서 임의의 요소를 추가
            Random rand = new Random((int)DateTime.Now.Ticks);
            int colorIndex = rand.Next(_sampleColors.Count);

            _colorList.Add(_sampleColors[colorIndex]);
        }

        private void Remove_Click(object sender, RoutedEventArgs e)
        {
            _colorList.RemoveAt(_colorList.Count - 1); // 마지막 요소를 제거
        }
    }
}
```

코드를 살펴보면 Add/Remove 버튼이 눌렸을 때 단지 ColorList 컬렉션에 항목을 추가하고 삭제하는 것밖에 없다. 그런데 예제 코드를 실행해 보면 버튼을 누름에 따라 ListBox 화면에서 동시에 색상이 추가/삭제되는 것을 확인할 수 있다. 왜냐하면 ObservableCollection<T> 타입이 컬렉션에 항목이 추가/삭제될 때마다 이벤트를 발생시키는 것과 함께, ListBox의 ItemsSource 속성이 해당 이벤트를 받아 ListBoxItem을 그에 맞게 추가/삭제하는 동작을 수행하기 때문이다.

이번에는 ItemsSource 바인딩을 이용해 그림 20.50처럼 CheckBox도 함께 보이도록 실행하는 방법을 알아보자. 물론 기본적으로는 UIElement를 컬렉션으로 구성하면 그와 동일하게 실행시킬

수는 있다.

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Reflection;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media;

namespace WpfApplication1
{
    public partial class MainWindow : Window
    {
        List<string> _sampleColors = new List<string>();

        public MainWindow()
        {
            InitializeComponent();

            AddElement("Yellow");
            AddElement("Green");
            AddElement("Blue");
.
            foreach (PropertyInfo propInfo in typeof(Colors).GetProperties(
                    BindingFlags.Static | BindingFlags.Public))
            {
                _sampleColors.Add(propInfo.Name);
            }
        }

        // UIElement 컬렉션 구성
        ObservableCollection<UIElement> _colorList =
                                    new ObservableCollection<UIElement>();
        public ObservableCollection<UIElement> ColorList
        {
            get { return _colorList; }
        }

        private void Add_Click(object sender, RoutedEventArgs e)
        {
            Random rand = new Random((int)DateTime.Now.Ticks);
            int colorIndex = rand.Next(_sampleColors.Count);

            AddElement(_sampleColors[colorIndex]);
        }

        // '예제 20.17'의 ListBoxItem.Content XAML을 코드로 구성
        private void AddElement(string colorName)
        {
            WrapPanel panel = new WrapPanel();

            CheckBox box = new CheckBox();
            box.VerticalAlignment = System.Windows.VerticalAlignment.Center;
            Label lbl = new Label();
            lbl.Content = colorName;

            panel.Children.Add(box);
```

```
            panel.Children.Add(lbl);

            _colorList.Add(panel);
        }

        private void Remove_Click(object sender, RoutedEventArgs e)
        {
            _colorList.RemoveAt(_colorList.Count - 1);
        }
    }
}
```

그런데 이 방법은 코드에서 직접 UI 구성을 함으로써 서로 강하게 결합 관계를 맺는다는 문제를 낳는다. 그보다는 기존의 ObservableCollection<string> 데이터 구성은 순수하게 유지하고 UI를 담당하는 XAML에서 표현을 변경하는 방법이 권장된다. WPF의 모든 ItemsControl 및 그것을 상속한 타입은 ItemTemplate 속성을 제공하는데, 여기에 개별 항목의 UI 요소를 임의로 구성하는 것이 가능하다. 따라서 예제 20.18의 C# 코드를 그대로 재사용하면서 XAML만 예제 20.19와 같이 바꾸면 그림 20.50처럼 체크 박스가 함께 나오게 된다.

예제 20.19 ItemTemplate 사용자 정의

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Form1" Height="350" Width="525"
        DataContext="{Binding RelativeSource={RelativeSource Self}}">
    <Grid>
        <ListBox ItemsSource="{Binding Path=ColorList}" Margin="0,0,0,35">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <WrapPanel>
                        <CheckBox VerticalAlignment="Center"></CheckBox>
                        <Label Content="{Binding Path=.}"></Label>
                    </WrapPanel>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
        <Button Content="Add" HorizontalAlignment="Left" Margin="10,289,0,0"
                VerticalAlignment="Top" Width="75" Click="Add_Click"/>
        <Button Content="Remove" HorizontalAlignment="Left" Margin="90,289,0,0"
                VerticalAlignment="Top" Width="75" Click="Remove_Click"/>
    </Grid>
</Window>
```

ItemTemplate에 사용된 Label 컨트롤의 바인딩 구문에 Path 값이 점(.)으로 지정된 것은 해당 ItemsSource에 바인딩된 ObservableCollection<string> 컬렉션의 개별 항목을 의미한다.

방금 실습한 예제가 바로 WPF 응용 프로그램만이 가진 매력 포인트다. 코드 파일은 변경하지 않은 상태에서 XAML 디자인을 어떻게 하느냐에 따라 프로그램의 외관이 자유롭게 바뀔 수 있기 때문에 개발자는 업무 코드에 집중하고 디자인 요소는 분리해서 디자이너에게 맡김으로써 분업이

가능해진다.

계속해서 예제 20.18에서 ObservableCollection의 형식 매개변수를 일반적인 클래스로 대체해서 넣어 보자. 이를 위해 ColorData 클래스를 하나 정의하고

```
using System.ComponentModel;
using System.Windows.Media;

namespace WpfApplication1
{
    public class ColorData : INotifyPropertyChanged
    {
        Brush _color;
        public Brush Color
        {
            get { return _color; }
            set
            {
                _color = value;
                OnPropertyChanged("Color");
            }
        }

        string _name;
        public string Name
        {
            get { return _name; }
            set
            {
                _name = value;
                OnPropertyChanged("Name");
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;

        protected virtual void OnPropertyChanged(string name)
        {
            if (PropertyChanged == null)
            {
                return;
            }

            PropertyChanged(this, new PropertyChangedEventArgs(name));
        }
    }
}
```

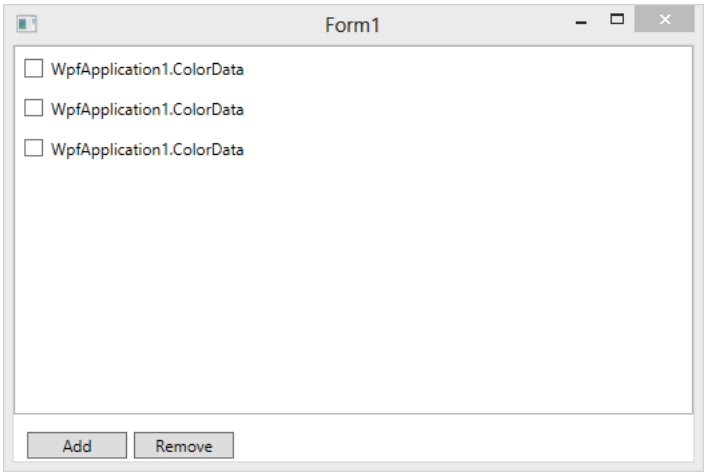
예제 20.18의 ObservableCollection의 형식 매개변수를 ColorData로 교체한다.

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Reflection;
using System.Windows;
using System.Windows.Controls;
```

```
using System.Windows.Media;

namespace WpfApplication1
{
    public partial class MainWindow : Window
    {
        List<string> _sampleColors = new List<string>();

        public MainWindow()
        {
            InitializeComponent();

            AddColor("Yellow");
            AddColor("Green");
            AddColor("Blue");

            foreach (PropertyInfo propInfo in typeof(Colors).GetProperties(
                    BindingFlags.Static | BindingFlags.Public))
            {
                _sampleColors.Add(propInfo.Name);
            }
        }

        private void AddColor(string colorName)
        {
            ColorData item = new ColorData();
            item.Name = colorName;
            Color color = (Color)ColorConverter.ConvertFromString(colorName);
            item.Color = new SolidColorBrush(color);

            _colorList.Add(item);
        }

        ObservableCollection<ColorData> _colorList =
                    new ObservableCollection<ColorData>();
        public ObservableCollection<ColorData> ColorList
        {
            get { return _colorList; }
        }

        private void Add_Click(object sender, RoutedEventArgs e)
        {
            Random rand = new Random((int)DateTime.Now.Ticks);
            int colorIndex = rand.Next(_sampleColors.Count);

            AddColor(_sampleColors[colorIndex]);
        }

        private void Remove_Click(object sender, RoutedEventArgs e)
        {
            _colorList.RemoveAt(_colorList.Count - 1);
        }
    }
}
```

그림 20.51은 이 코드를 실행한 결과를 보여준다.

그림 20.51 ColorData 타입으로 바꾼 후 실행한 모습



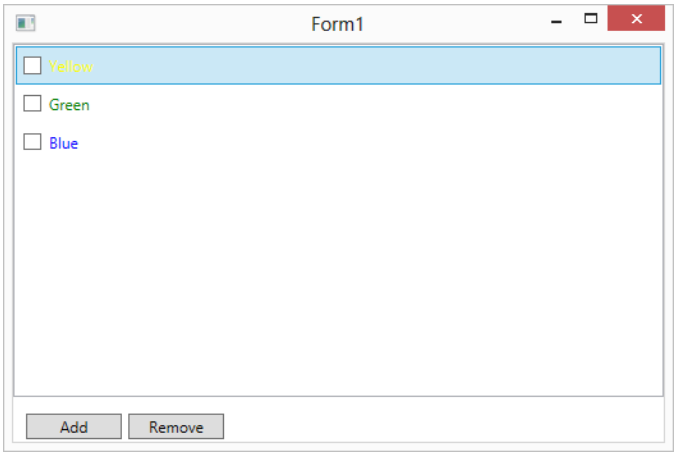
항목이 "WpfApplication1.ColorData"라는 문자열로 출력되는 것을 볼 수 있다. 왜냐하면 XAML의 <Label Content="{Binding Path=.}"></Label>에서 Binding Path를 점(.)으로 설정했기 때문이다. 이는 ItemsSource에 지정된 컬렉션의 항목 하나를 대표하므로 결국 ColorData 인스턴스와 대응되고 해당 인스턴스의 ToString 메서드가 호출된 결과를 보여주기 때문에 나타나는 현상이다. 따라서 원하는 대로 색상명을 나타내고 싶다면 ColorData 타입의 Name 속성을 Binding Path에 지정하면 된다. 즉, <Label Content="{Binding Path=Name}"></Label>로 변경하면 그림 20.50과 같이 원래 의도한 결과를 볼 수 있다.

여기서 조금 재미있는 효과를 더 내보자. ColorData 타입에는 Color 속성이 System.Windows.Media.Brush 타입으로 정의돼 있다. 이 타입은 WPF에서 정의된 각종 컨트롤의 색상을 나타내는 속성으로도 사용할 수 있기 때문에 원한다면 Binding을 이용해 XAML에서 활용할 수도 있다. 예를 들어, ItemTemplate을 다음과 같이 변경하면

```
<ListBox.ItemTemplate>
    <DataTemplate>
        <WrapPanel>
            <CheckBox VerticalAlignment="Center"></CheckBox>
            <Label Foreground="{Binding Path=Color}"
                   Content="{Binding Path=Name}"></Label>
        </WrapPanel>
    </DataTemplate>
</ListBox.ItemTemplate>
```

Label의 텍스트 색상이 바뀌어서 실행되는 것을 확인할 수 있다(그림 20.52 참고).

그림 20.52 Foreground 속성에 바인딩된 글자 색



Button 하나를 더 추가해서 이미 ColorList에 있는 항목의 속성을 바꿔보자.

```
<Button Content="Red" HorizontalAlignment="Left" Margin="170,289,0,0"
        VerticalAlignment="Top" Width="75" Click="Red_Click"/>
```

```
private void Red_Click(object sender, RoutedEventArgs e)
{
    foreach (var item in _colorList)
    {
        item.Color = new SolidColorBrush(Colors.Red);
        item.Name = "Red";
    }
}
```

프로그램을 실행한 다음 "Red" 버튼을 누르면 ListBox가 담고 있는 모든 항목이 빨간 색으로 바뀐다. 왜냐하면 ColorData 타입은 INotifyPropertyChanged 인터페이스를 구현하고 있으므로 속성이 바뀐 경우 이 사실을 데이터 바인딩된 요소에 통보하게 되고, 결과적으로 Label의 내용이 바뀌는 것이다.

이 절의 내용을 통해 알 수 있듯이 WPF의 가장 큰 매력은 개발자가 UI보다는 데이터 처리에 좀 더 집중할 수 있게 만들어 준다는 점이다. 데이터만 있다면 UI는 XAML 수준에서 자유롭게 표현할 수 있고 심지어 디자이너가 그 과정을 도와줄 수 있는 기반을 제공한다.

정리

한 가지 분명한 점은 WPF 응용 프로그램을 만드는 방법이 윈도우 폼 응용 프로그램을 만드는 방법과 비교해 더 어렵다는 점이다. 만약 "윈도우 운영체제"에서 실행되는 "윈도우 응용 프로그램"만 작성한다면 WPF 응용 프로그램을 굳이 배우지 않고 "윈도우 폼 응용 프로그램"으로 프로그램을 만들어도 좋다. 하지만 WPF는 배워 둘 만한 가치가 충분히 있다. 왜냐하면 WPF 응용 프로

그램을 만드는 XAML 방식은 "실버라이트 응용 프로그램", "윈도우 폰 응용 프로그램", "윈도우 8 스토어 앱"과 같은 프로그램에도 적용되기 때문이다. 이것이 WPF를 공부해야 하는 가장 확실한 이유다.

WPF 및 XAML과 같은 프로그램이 재미있는 또 한 가지 이유는 디자이너와의 협업이 가능하다는 점이다. 솔직히 말해서 WPF가 처음 나왔을 때 필자 역시 아무리 WPF라고 해도 응용 프로그램을 개발할 때 디자이너와 협업할 수 있다는 사실을 믿지 않았다. 하지만 그 생각이 잘못됐다는 것을 확인하는 데는 그리 오랜 시간이 걸리지 않았는데, 필자와 함께 일한 디자이너가 익스프레션 블렌드를 이용해 WPF 요소의 디자인을 자유자재로 다루는 모습을 봤기 때문이다. 개발팀에 포토샵을 다루는 디자이너가 있다면 지금 당장 익스프레션 블렌드 책을 한 권 사주고 WPF/실버라이트/윈도우 폰/스토어 앱을 협업해서 만들어 보라. 필자가 느꼈던 놀라움을 머지 않아 똑같이 느끼게 될 것이다.

20.3 서비스 응용 프로그램

컴퓨터가 켜지면 윈도우 운영체제가 초기화 과정을 마치고 로그인할 사용자 계정을 묻는 단계가 나온다. 로그인하기 전까지 여러분은 어떠한 프로그램도 실행할 수 없다. 그런데 이렇게 사용자가 로그인하지 않은 상태에서도 어떤 특정 용도의 프로그램은 실행돼야 할 때가 있다. 예를 들어, 네이버 홈페이지 서비스를 담당하는 HTTP 소켓 서버 프로그램은 사용자가 해당 컴퓨터에 로그인하지 않아도 자동으로 시스템이 부팅되면서 실행될 필요가 있다. 윈도우 운영체제에서는 전통적으로 "NT 서비스"라는 유형으로, 유닉스/리눅스 계열에서는 데몬(Daemon) 프로세스라고 알려져 있는데, 이런 프로그램의 특징은 사용자가 로그인하지 않은 상태에서도 실행될 수 있다는 것이다.

"제어판" / "관리 도구" / "서비스"를 실행하면 여러분의 컴퓨터에 설치된 각종 "NT 서비스 프로그램"을 볼 수 있다.

그림 20.53 서비스 목록

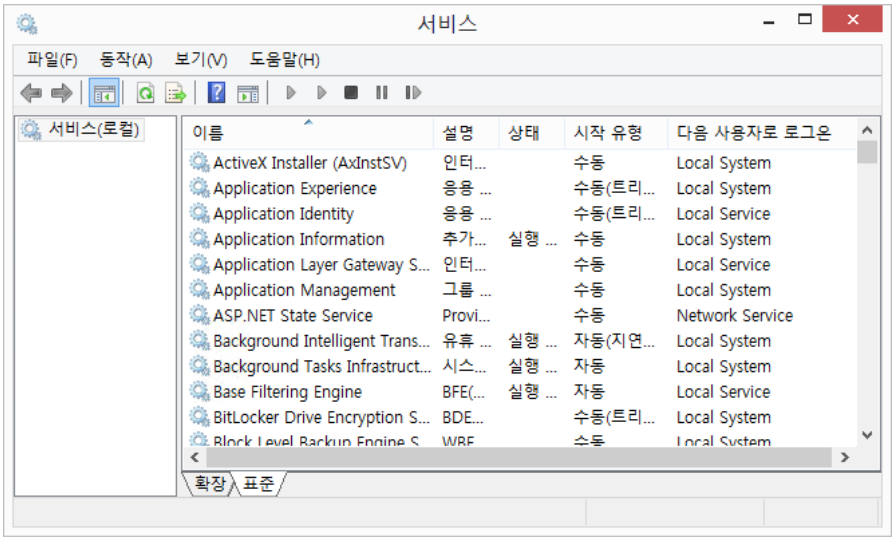


그림 20.53에서 "시작 유형" 칼럼의 값이 "자동"으로 설정된 서비스는 운영체제가 시작될 때 실행된다. 한 가지 눈여겨봐야 할 것이 있다면 "다음 사용자로 로그온" 칼럼에 보이는 "Local System", "Local Service", "Network Service"와 같은 특수한 계정들이다. 윈도우 운영체제에서 실행되는 모든 EXE 프로세스는 해당 프로세스가 가진 권한을 "사용자 계정"으로 묶기 때문에 반드시 해당 프로세스와 연관돼야 할 사용자 계정을 명시해야 한다. 사실 이 문제는 로그온한 후에 실행하는 프로그램(예: 웹 브라우저)에는 문제가 되지 않는다. 왜냐하면 그러한 프로그램은 여러분이 로그온한 계정에서 실행하는 것이므로 자연스럽게 해당 사용자 계정이 가진 권한을 사용하면 되기 때문이다. 하지만 서비스 프로그램은 사용자가 로그온하지 않은 상태에서 실행되므로 반드시 실행 권한을 확보하기 위해 사용자 계정을 명시해야 한다. 물론 일반적인 사용자 계정으로 지정하는 것도 가능하지만 윈도우 운영체제는 서비스를 위한 세 가지 특별한 사용자 계정을 제공한다.

- Local System
  관리자 그룹(Administrators)에 준하는 높은 수준의 권한을 지닌 서비스용 계정으로서 보안상 특별한 이유가 없다면 사용하지 않는 것을 권장한다.
- Local Service
  Local System 계정이 지닌 높은 수준의 권한을 제거한 계정으로, 일반적인 용도의 서비스라면 이 사용자 계정을 사용하는 것을 권장한다.
- Network Service
  Local Service 계정과 완전하게 동일한 수준의 권한을 가진다. 하지만 "Local Service" 계정은 액티브 디렉터리 내의 다른 컴퓨터로 접근하는 경우 "익명(Anonymous)" 권한을 갖지만, Network Service는 "컴퓨터 계정(Computer account)" 권한으로 접근한다.

참고!	액티브 디렉터리(AD: Active Directory)를 간단히 설명하면 윈도우 서버 운영체제에서 제공하는 통합 계정 관리 서비스를 의미한다. 액티브 디렉터리에 계정을 하나 등록하면 액티브 디렉터리에 참여하는 모든 컴퓨터에 해당 사용자 계정 권한으로 접근하는 것이 가능하다.

선택의 기준은 명확하다. 높은 권한이 필요하다면 Local System, 낮은 권한으로 충분하지만 액티브 디렉터리 내의 다른 컴퓨터로 접근해야 한다면 Network Service, 그렇지 않다면 Local Service를 지정하면 된다. 액티브 디렉터리가 구성되지 않은 환경이라면 Network Service와 Local Service의 권한은 사실상 같다.

서비스 프로그램으로 등록되려면 우선 그 프로그램을 레지스트리에 등록해야 한다. 그러면 윈도우 운영체제에 포함된 SCM(서비스 컨트롤 관리자: Service Control Manager)은 그 프로그램을 인식하게 되고, 그림 20.53의 목록에도 나타난다. 이후에 윈도우 시스템이 부팅될 때마다 SCM은 등록된 서비스 중에 "자동"으로 설정된 항목을 찾아서 레지스트리에 설정된 EXE의 경로에 해당하는 프로그램을 실행한다.

실습을 위해 TCP 서버 소켓을 이용한 서비스를 하나 만들어 보자.

1. "새 프로젝트" 창에서 "빈 프로젝트" 유형의 템플릿을 선택하고 이름을 "MyService"로 지정한다.
2. "System.ServiceProcess" 어셈블리와 "System.Configuration.Install" 어셈블리를 참조 추가한다.
3. 프로젝트 속성 창을 열고 "응용 프로그램(Application)" 탭에서 "출력 형식(Output type)"을 "Windows 응용 프로그램"으로 변경한다.
4. 프로젝트에 "클래스" 유형으로 "Program.cs" 파일과 "EchoServer.cs" 파일을 각각 하나씩 추가한다.

먼저 MyService 프로젝트가 "서비스용 프로그램"으로 구동하기 위한 최소한의 조건을 만족하는 코드를 추가하면 예제 20.20과 같다.

예제 20.20 서비스를 위한 기본 코드

```
// EchoServer.cs
using System.ServiceProcess;

namespace MyService
{
    // 서비스를 위한 기본 구현이 포함된 ServiceBase 타입을 상속받는다.
    public class EchoServer : ServiceBase
    {
        public EchoServer()
        {
            this.ServiceName = "MyEchoServer"; // 서비스 이름을 지정
        }

        // 서비스 시작 시 실행되는 메서드
        protected override void OnStart(string[] args)
        {
```

```
        }

        // 서비스 중지 시 실행되는 메서드
        protected override void OnStop()
        {
        }
    }
}
```

```
// Program.cs
using System.ServiceProcess;

namespace MyService
{
    static class Program
    {
        static void Main()
        {
            ServiceBase[] ServicesToRun;
            ServicesToRun = new ServiceBase[]
            {
                new EchoServer()
            };
            ServiceBase.Run(ServicesToRun);
        }
    }
}
```

EXE 프로세스가 구동될 때 가장 먼저 실행되는 Program.cs의 Main 메서드를 보면 ServiceBase 배열에 EchoServer 객체를 담아 ServiceBase.Run 메서드에 전달하는 것을 볼 수 있다. 따라서 하나의 EXE 프로세스 안에는 여러 개의 서비스를 담을 수 있는데, 실제로 그림 20.53 목록에서 보여지는 많은 서비스가 각각 하나의 EXE로 실행되는 것이 아니고, 어떤 서비스의 경우에는 하나의 EXE를 공유하기도 한다. EchoServer.cs 파일의 경우에는 아무 일도 하지 않는 기본 서비스 코드가 포함돼 있다. 이 서비스가 레지스트리에 등록되면 그림 20.53과 같은 화면이 나타나고 메뉴를 이용해 서비스를 시작하게 만들면 EchoServer 타입의 OnStart 메서드가 SCM에 의해 실행된다. 마찬가지로 서비스를 중지하면 OnStop 메서드가 실행된다.

위의 과정은 Express 버전의 비주얼 스튜디오를 사용하는 경우에만 해당한다. Professional 이상의 유료 버전을 사용하고 있다면 다음과 같이 간단하게 서비스 프로젝트를 생성할 수 있다.

1. "새 프로젝트" 창에서 "Windows" 범주의 "Windows Service" 유형의 템플릿을 선택하고 이름을 "MyService"로 지정한다.

2. 기본 생성된 "Service1.cs" 파일의 이름을 "EchoServer.cs"로 변경한다.

기본적인 동작만 수행하는 서비스 프로젝트를 생성했으니, 이제 레지스트리에 등록해서 서비스

관리자를 이용해 실제로 "시작" / "중지"시킬 수 있다. 이 작업을 레지스트리 편집기(regedit.exe)를 이용해 직접 할 수도 있지만 닷넷 프레임워크에서는 이 과정을 수월하게 만들어주는 "InstallUtil.exe"라는 프로그램이 제공되므로 이 프로그램을 사용하길 권장한다.

- 32비트 운영체제인 경우

32비트 서비스 등록: "C:₩Windows₩Microsoft.NET₩Framework₩v2.0.50727₩InstallUtil.exe"

- 64비트 운영체제인 경우

32비트 서비스 등록: "C:₩Windows₩Microsoft.NET₩Framework₩v2.0.50727₩InstallUtil.exe"
64비트 서비스 등록: "C:₩Windows₩Microsoft.NET₩Framework64₩v2.0.50727₩InstallUtil.exe"

InstallUtil.exe 프로그램을 이용하면 서비스 등록/해지를 다음과 같이 간단하게 할 수 있다.

- 서비스 등록

InstallUtil.exe MyService.exe

- 서비스 해지

InstallUtil.exe /u MyService.exe

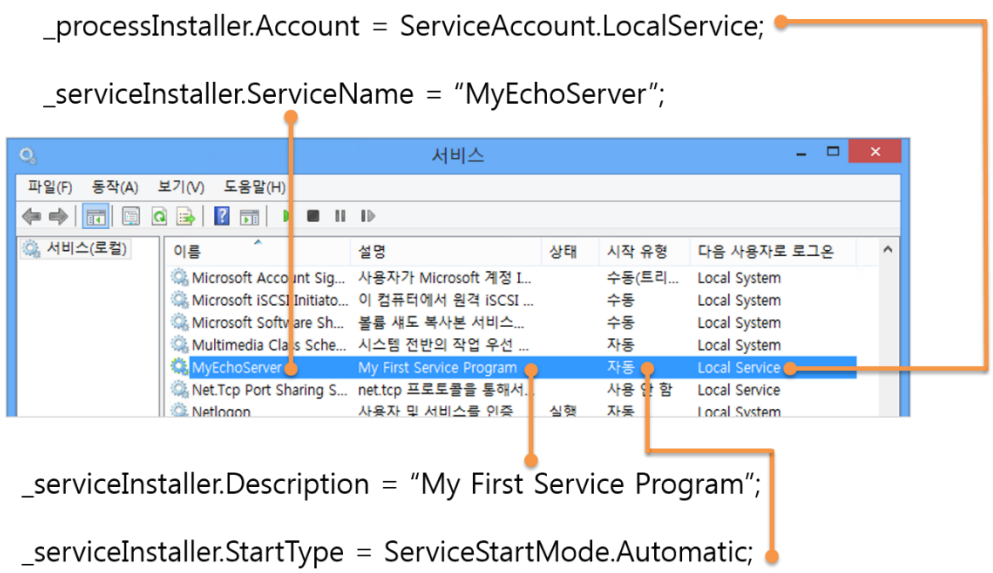
하지만 아직은 InstallUtil.exe로 등록할 수 없는 상태다. 왜냐하면 EXE 프로그램 제작자는 InstallUtil.exe에게 서비스 등록과 관계된 몇 가지 정보를 전달해야 하기 때문이다. 그리고 그 방법은 클래스를 통해 제공할 수 있다. 따라서 MyService 프로젝트에 새롭게 "MyServiceInstaller.cs" 코드 파일을 추가하고 다음과 같은 내용을 작성한다.

```
using System.ServiceProcess;
using System.Configuration.Install;

namespace MyService
{
    [RunInstaller(true)]
    public class MyServiceInstaller : Installer
    {
        private ServiceProcessInstaller _processInstaller;
        private ServiceInstaller _serviceInstaller;

        public MyServiceInstaller()
        {
            this._processInstaller = new ServiceProcessInstaller();
            this._serviceInstaller = new ServiceInstaller();

            this._processInstaller.Account = ServiceAccount.LocalService;

            this._serviceInstaller.ServiceName = "MyEchoServer";
            this._serviceInstaller.Description = "My First Service Program";

            this._serviceInstaller.StartType = ServiceStartMode.Automatic;
```

```
            this.Installers.AddRange(new Installer[] {
                this._processInstaller,
                this._serviceInstaller});
        }
    }
}
```

한 가지 중요한 것은 Installer 타입을 상속받고 반드시 RunInstaller 특성을 지정해야 한다는 점이다. 그렇게만 하면 InstallUtil.exe는 자동으로 이 클래스를 감지하고 서비스 설치와 관련된 정보를 알아낸다. 나머지 변수는 그림 20.54에서 볼 수 있듯이 개별적인 의미를 갖는다.

그림 20.54 서비스 관리자에 설정되는 변수의 값



드디어 등록 준비가 모두 끝났다. 예제 프로젝트를 빌드하고 생성된 EXE 폴더에서 "관리자 권한"으로 실행한 명령행 창을 이용해 InstallUtil.exe를 실행하면 다음과 같은 출력 결과를 확인할 수 있고, "서비스" 관리 도구를 실행하면 그림 20.54처럼 "MyEchoServer" 항목이 생성된 것을 볼 수 있다.

```
C:₩Windows₩system32>cd D:₩temp₩MyService₩MyService₩bin₩Debug

C:₩Windows₩system32>d:

D:₩temp₩MyService₩MyService₩bin₩Debug>installutil MyService.exe
Microsoft (R) .NET Framework Installation utility Version 4.0.30319.18010
Copyright (C) Microsoft Corporation. All rights reserved.


트랜잭트 설치를 실행하고 있습니다.
```

```
설치의 Install 단계를 시작하고 있습니다.
D:₩temp₩MyService₩MyService₩bin₩Debug₩MyService.exe 어셈블리의 진행 상황을 보려면 로그 파일 내용을 검토하십시오.
파일은 D:₩temp₩MyService₩MyService₩bin₩Debug₩MyService.InstallLog 위치에 있습니다.
어셈블리 'D:₩temp₩MyService₩MyService₩bin₩Debug₩MyService.exe'을(를) 설치하고 있습니다.
영향을 받는 매개변수:
   logtoconsole =
   logfile = D:₩temp₩MyService₩MyService₩bin₩Debug₩MyService.InstallLog
   assemblypath = D:₩temp₩MyService₩MyService₩bin₩Debug₩MyService.exe
MyEchoServer 서비스를 설치하고 있습니다...
MyEchoServer 서비스가 설치되었습니다.
EventLog 소스 MyEchoServer을(를) 로그 Application에 만들고 있습니다...

Install 단계는 완료되었으며 Commit 단계를 시작하고 있습니다.
D:₩temp₩MyService₩MyService₩bin₩Debug₩MyService.exe 어셈블리의 진행 상황을 보려면 로그 파일 내용을 검토하십시오.
파일은 D:₩temp₩MyService₩MyService₩bin₩Debug₩MyService.InstallLog 위치에 있습니다.
어셈블리 'D:₩temp₩MyService₩MyService₩bin₩Debug₩MyService.exe'을(를) 커밋하고 있습니다.
영향을 받는 매개변수:
   logtoconsole =
   logfile = D:₩temp₩MyService₩MyService₩bin₩Debug₩MyService.InstallLog
   assemblypath = D:₩temp₩MyService₩MyService₩bin₩Debug₩MyService.exe

Commit 단계가 완료되었습니다.

트랜잭트 설치가 완료되었습니다.

D:₩temp₩MyService₩MyService₩bin₩Debug>
```

하지만 이 작업으로는 서비스가 설치만 됐을 뿐 실행 중인 것은 아니다. 직접 "서비스" 관리 도구에서 "시작" 메뉴를 선택하거나 컴퓨터를 다시 부팅하면 서비스가 실행된다.

이제 본격적으로 서비스를 위한 코드를 추가해 보자. 서비스 응용 프로그램은 사용자가 실행하지 않고 SCM에 의해 실행되는데, 그 진입점은 예제 20.20에서 구현한 OnStart 메서드다. 따라서 서비스가 "시작"할 때 실행해야 할 코드는 OnStart에 넣어두고, 마찬가지로 서비스를 "중지"시킬 때 실행될 코드는 OnStop에 넣으면 된다. 한 가지 특이한 점이 있다면 SCM은 OnStart 메서드를

실행한 후 해당 코드가 모두 실행이 완료될 때까지 대기한다는 것이다. 기본적으로 30초를 기다리게 되며, 만약 그 시간 내에 OnStart 메서드가 실행을 반환하지 않으면 강제로 프로세스를 종료한다. 이런 이유로 일반적으로는 서비스를 위한 코드를 별도로 스레드 함수로 분리하고 OnStart에서는 스레드를 시작하는 식으로 처리한다. 따라서 우리가 구현하는 EchoServer의 코드 역시 그와 같은 구조를 따른다.

```
Thread _workerThread;
bool _exitThread = false;

protected override void OnStart(string[] args)
{
    _workerThread = new Thread(echoFunc);
    _workerThread.IsBackground = true;
    _workerThread.Start(); // 스레드를 시작하는 것으로 OnStart의 역할을 끝낸다.
}

protected override void OnStop()
{
    _exitThread = true;

    if (_workerThread != null)
    {
        if (_workerThread.Join(5000) == false) // 5초간 스레드 종료 대기
        {
            _workerThread.Abort();
            _workerThread = null;
        }
    }
}

private void echoFunc(object obj)
{
    // echo server 코드
}
```

echoFunc의 내부 코드에서는 '예제 6.38 TCP 서버 측 소켓을 구현'의 코드를 그대로 복사해 넣으면 된다. 단지, OnStop에서 서비스 중지가 처리될 수 있게 _exitThread 불린형 변수를 이용해 루프를 제어하는 코드를 추가하는 식으로 변경한다.

```
private void echoFunc(object obj)
{
    using (Socket srvSocket =
    new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp))
    {
        IPEndPoint endPoint = new IPEndPoint(IPAddress.Any, 11201);

        srvSocket.Bind(endPoint);
        srvSocket.Listen(10);

        while (_exitThread == false)
        {
            Socket clntSocket = srvSocket.Accept();
```

```
            byte[] recvBytes = new byte[1024];

            int nRecv = clntSocket.Receive(recvBytes);
            string txt = Encoding.UTF8.GetString(recvBytes, 0, nRecv);

            byte[] sendBytes = Encoding.UTF8.GetBytes("Hello: " + txt);
            clntSocket.Send(sendBytes);
            clntSocket.Close();
        }
    }
}
```

물론 Socket 타입의 Accept 메서드가 동기 형식의 블록킹 호출이기 때문에 클라이언트가 접속하지 않는 경우 Accept 단계에서 스레드가 계속 머물러 있게 된다. 그런 경우에는 OnStop 메서드에서 _exitThread 변수를 true로 변경해도 스레드가 종료되는 것은 아니므로 이럴 때는 강제로 Thread.Abort 메서드를 호출하는 방식으로 해결한다. 참고로 Thread.Abort 메서드의 사용은 권장되지 않으므로 좀 더 코드를 개선하고 싶다면 Accept를 비동기 방식으로 변경해서 처리하는 편이 좋다.

이렇게 완성된 Echo 서비스는 컴퓨터가 재부팅되고 나서 사용자가 로그인하지 않아도 "Local Service" 계정의 권한으로 자동 실행된다. 소위 "서버"라고 하는 제품은 모두 이 같은 방식으로 제작돼 있다는 점을 알아두자. 예를 들어, 6.8 '데이터베이스' 절에서 배운 SQL 서버를 비롯해 HTTP 웹 서버, FTP 서버, 메일 서버와 같은 프로그램들이 NT 서비스 유형의 대표적인 사례다.

20.4 웹 응용 프로그램

앞에서 예제 6.42 'TCP 소켓으로 구현한 HTTP 서버'를 통해 간단하게 HTTP 통신을 하는 서버를 만들어봤다. 사실 그 예제는 HTTP 프로토콜의 기능 중 극히 일부분만 구현한 것에 불과하고 그 밖의 모든 기능을 넣으려면 더욱 많은 코드를 추가해야만 한다. 다행인 점은 HTTP 프로토콜 명세를 충실하게 구현한 제품들이 이미 많이 공개돼 있기 때문에 대부분의 개발자는 그 제품을 기반으로 동작하는 "서비스"를 만드는 데 더욱 집중할 수 있다. 여기서 말하는 "서비스"가 바로 "웹 응용 프로그램"에 해당한다.

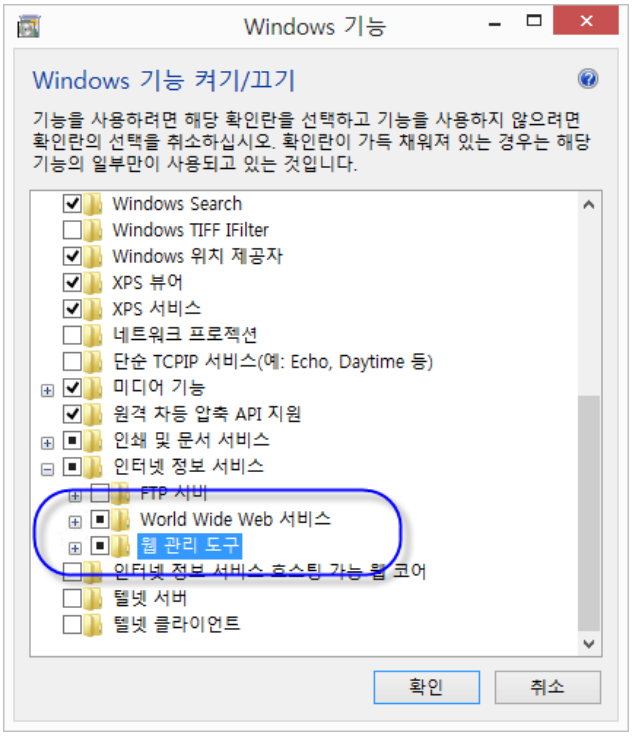
처음 HTTP(Hyper Text Transport Protocol) 서비스가 등장했던 시절에는 그 이름에 맞게 단순히 링크로 엮어진 문서(Hyper Text)를 웹 브라우저가 요청하면 서버 측에서 전송(Transport)해주는 규약(Protocol)에 불과했다. HTML 텍스트를 담은 문서와 각종 이미지 파일, CSS, 자바 스크립트로 구성된 파일들은 정적 콘텐츠(Static Content)라 하고 웹 서버는 요청된 콘텐츠를 가공 없이 그대로 전송한다.

지금까지의 설명을 직접 실습해보자. 윈도우 운영체제에서 쉽게 사용할 수 있는 무료 웹 서버는 대표적으로 "인터넷 정보 서비스(IIS: Internet Information Services)"가 있는데, 아래의 방법에

따라 간단하게 설치해서 사용할 수 있다.

1. "제어판"을 실행하고 "프로그램 및 기능(Programs and Features)" 프로젝트" 항목을 선택한 후 좌측 상단의 "Windows 기능 켜기/끄기(Turn Windows features on or off)" 링크를 누른다.

2. 선택 창에서 "인터넷 정보 서비스(Internet Information Services)" 항목을 펼치면 그 아래에 "World Wide Web 서비스"와 "웹 관리 도구"가 나타나는데, 모두 선택하고 "확인" 버튼을 누른다.



IIS 웹 서버 설치가 완료되면 "C:₩inetpub₩wwwroot" 폴더가 생성된 것을 확인할 수 있다. 메모장 등의 편집기를 이용해 다음과 같이 간단한 HTML 문서를 만들어 wwwroot 폴더에 "mywebpage.html"이라는 이름으로 복사해 넣고

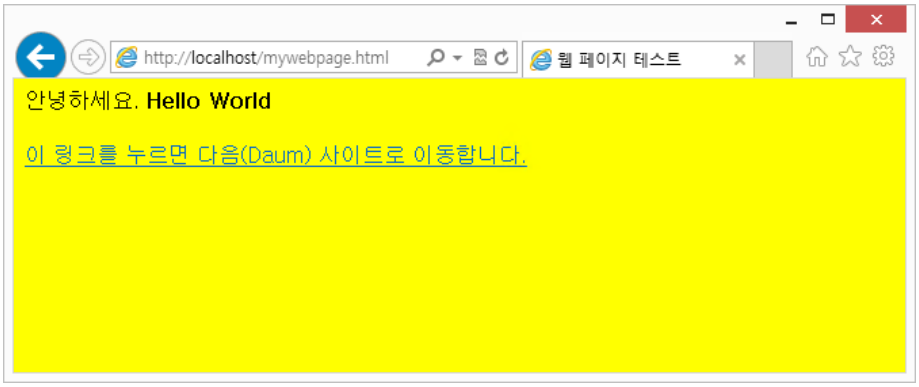
파일명: mywebpage.html
<!DOCTYPE html> <html> <head> <title>웹 페이지 테스트</title> </head> <body style="background-color: yellow;"> 안녕하세요. Hello World   이 링크를 누르면 다음(Daum) 사이트로 이동합니다.   </body> </html>

참고!	이 장에서는 독자가 기본적인 HTML 문법을 알고 있다고 가정한다. 하지만 처음 접한

	다고 해서 문제될 것은 없는데, WPF의 XAML 구문에서 다룬 XML 표기 방식이 그대로 사용되기 때문이다.

웹 브라우저의 주소 표시줄에 http://localhost/mywebpage.html이라고 입력하면 그림 20.55와 같은 화면이 나타난다.

그림 20.55 mywebpage.html을 웹브라우저에서 연 모습



이때 윈도우 운영체제의 작업 관리자를 통해 목록을 확인해 보면 w3wp.exe라는 이름의 프로세스가 실행된 것을 볼 수 있다. 예제 6.42를 빌드해서 만든 EXE를 실행해 HTTP 요청을 처리했던 것처럼, IIS 서비스는 w3wp.exe 프로세스를 실행해 웹 요청을 처리한다. 단지 차이점이 있다면 IIS는 '20.3 서비스 응용 프로그램' 절에서 배운 서비스 유형으로 실행되므로 컴퓨터에 사용자가 로그인하지 않아도 동작할 수 있다.

그림 20.55와 같은 결과가 나오기까지 어떤 처리가 진행되는지 짚어볼 필요가 있다. 우선 웹 브라우저는 사용자가 입력한 URL에서 표 20.10의 규칙에 따라 요청 정보를 추출한다.

표 20.10 URL 의미

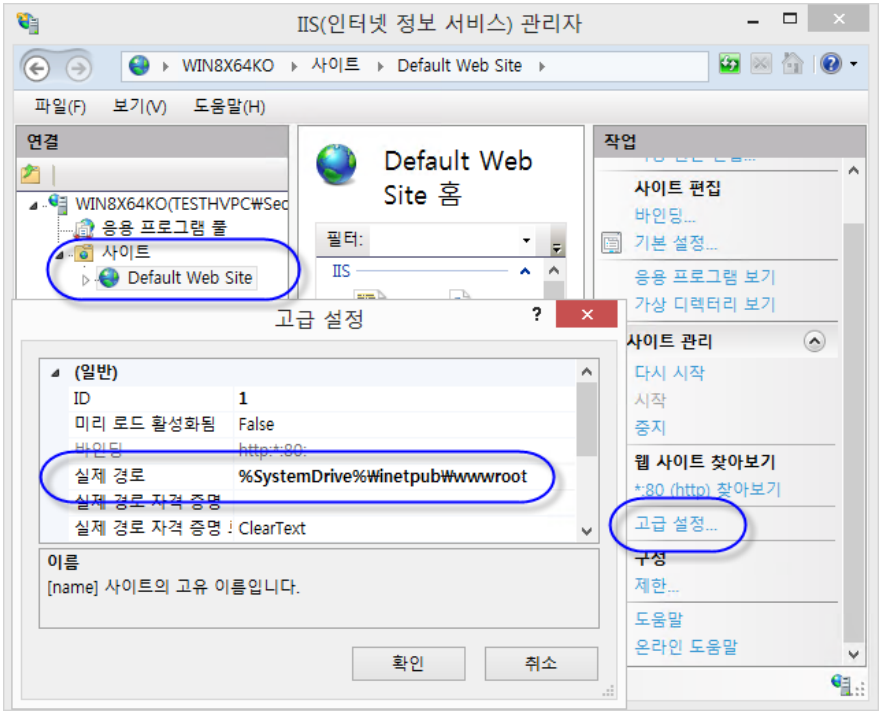
URL	의미
http://	HTTP 통신임을 나타내는 접두사
localhost	웹 서버의 도메인 네임 (localhost는 컴퓨터 자신이 대상임을 지칭하는 특별한 문자열 주소) 또는 IP 주소
/mywebpage.html	요청해야 할 웹 자원(Resource)의 경로

이 정보를 바탕으로 웹 브라우저는 다음과 같은 GET 요청을 TCP 소켓을 통해 웹 서버로 전송하고 HTTP 규약에 따라 웹 서버는 "/mywebpage.html"이라는 경로에 위치한 문서를 웹 브라우저로 TCP 소켓을 통해 응답하게 된다.

```
GET /mywebpage.html HTTP/1.1
Host: localhost
```

그런데 웹 서버는 mywebpage.html 파일을 어떤 근거로 c:₩inetpub₩wwwroot 폴더에서부터 찾는 걸까? 이를 확인하려면 "제어판" / "관리 도구"에서 "IIS(인터넷 정보 서비스) 관리자"를 실행해야 한다. 그림 20.56과 같은 화면이 나오면 좌측 상단의 "연결" 트리에서 "사이트" 하위의 "Default Web Site"를 마우스로 선택하고, 우측 하단의 "고급 설정..." 링크를 누르면 팝업 설정 창이 나타나는데, 여기서 "실제 경로"에 지정된 값을 확인할 수 있다.

그림 20.56 IIS 관리자에서 사이트 고급 설정을 통해 경로를 확인



일반적으로는 "실제 경로" 대신 "웹 루트 폴더" 또는 더 간단하게 "웹 루트"라고 부르기도 한다. 그림 20.56에서 볼 수 있듯이 IIS가 기본으로 지정한 "%SystemDrive%₩inetpub₩wwwroot" 경로는 결국 "C:₩inetpub₩wwwroot"가 되기 때문에 IIS 웹 서버는 "/mywebpage.html" 파일을 해당 폴더에서 찾아 웹 브라우저로 HTTP 응답 헤더와 함께 파일의 내용을 함께 실어 전송한다.

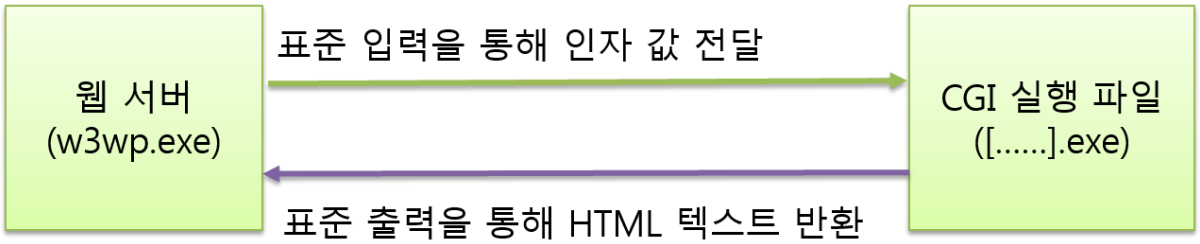
```
HTTP/1.1 200 OK
Content-Type: text/html
Server: Microsoft-IIS/8.0
Date: Wed, 17 Jul 2013 10:42:09 GMT
Content-Length: 263

<html>

<head>
    <title>웹 페이지 테스트</title>
</head>

.....[생략].....

</html>
```

응답을 받은 웹 브라우저는 HTTP 헤더 내용으로부터 "Content-Length"에 지정된 길이(예제에서는 263)를 통해 HTTP 본문(Body)의 크기를 알게 되고 그것을 해석해 화면에 보여준다.

물론 원한다면 웹 루트 폴더를 변경해도 된다. "C:₩temp"로 바꾸고 "mywebpage.html" 파일을 그곳으로 복사해서 서비스해도 무방하다. 만약 정적 콘텐츠만 담긴 웹 사이트를 만들고 싶다면 이 정도 구성만으로도 지금 당장 서비스를 시작할 수 있다. 공용 IP가 할당된 컴퓨터에 IIS 서비스를 설치하고 웹 루트 폴더에 파일을 넣어두면 전 세계 사용자들이 방문해 문서를 열람할 수 있다.

문제는 정적 콘텐츠로는 다양한 웹 서비스를 만드는 데 한계가 있다는 점이다. 즉, 프로그램에 의해 자유롭게 변경 가능한 동적 콘텐츠(Dynamic Content)를 생산할 수 있는 능력이 필요한데, 이러한 요구 사항을 만족시키는 과정에서 본격적인 "웹 응용 프로그램"의 시대가 열린다. 초기에는 동적 콘텐츠를 생산하기 위해 그림 20.57에 나온 "CGI(Common Gateway Interface)"라는 기술을 사용했다. 웹 서버는 동적 콘텐츠가 필요할 때마다 내부적으로 EXE 프로세스를 인자 값과 함께 실행하고 그 결과를 HTML 텍스트로 반환받는다.

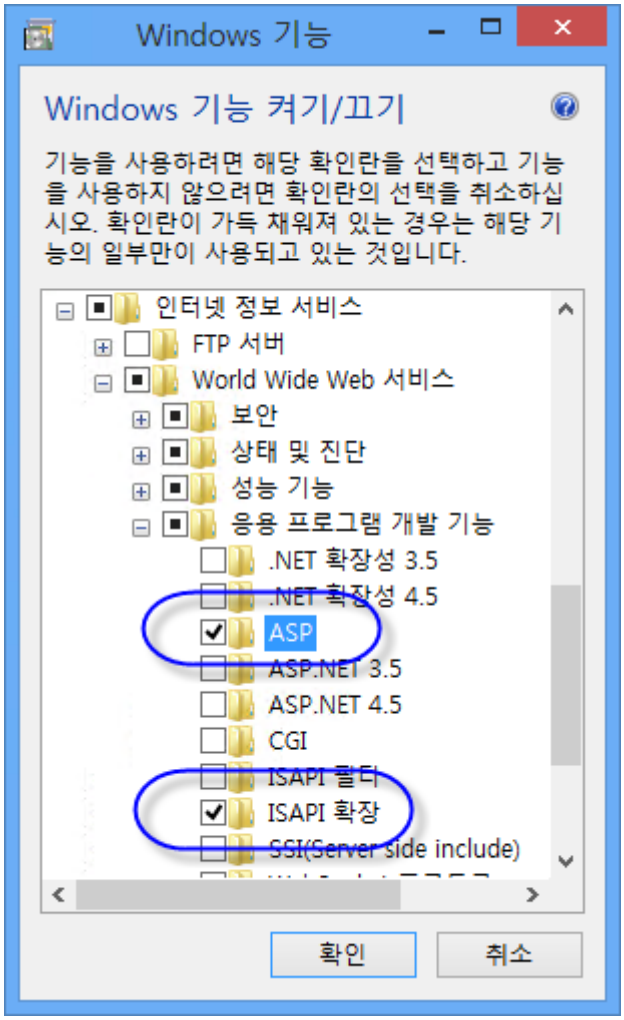
그림 20.57 CGI 동작 방식



웹 사이트의 특성상 1초에도 수십 건씩 요청이 들어올 수 있는데, 그때마다 EXE 파일을 실행하면 적지 않은 부하(오버헤드: overhead)가 발생한다. 부하가 발생하면 자연스럽게 응답 속도가 느려지고 사용자의 불편함이 가중된다. 이런 이유로 CGI는 현재 거의 사용되지 않고 있다. 하지만 코드에 문제가 있는 경우 웹 서버 프로세스(w3wp.exe)에 영향을 미치지 않고 CGI로 실행된 EXE 프로세스만 종료되기 때문에 안정성이 높다는 장점은 있다.

시간은 흘러 CGI는 마이크로소프트에서 개발한 ASP(Active Server Pages) 기술에 빠르게 자리를 내주게 된다. ASP를 이용하면 EXE 실행 파일을 만들 필요 없이 스크립트(Script) 언어를 이용해 동적 콘텐츠를 만들어낼 수 있으므로 웹 응용 프로그램의 제작 속도가 획기적으로 향상된다. ASP로 웹 응용 프로그램을 만드는 것이 얼마나 쉬운지 직접 시험해볼 텐데, 이를 위해서는 기본 설치된 IIS에 그림 20.58에 나와 있는 "ASP"와 "ISAPI 확장" 구성 요소를 추가로 설치해야 한다.

그림 20.58 ASP, ISAPI 확장 구성 요소 설치



이 두 가지 구성 요소가 설치된 후부터 w3wp.exe 프로세스는 확장자가 asp인 파일에 대해 해당 파일 내부에 포함된 스크립트 언어를 처리할 수 있다. 테스트를 위해 다음과 같은 내용으로 mycalc.asp 파일을 만들어 C:₩inetpub₩wwwroot에 저장한 후

파일명: mycalc.asp

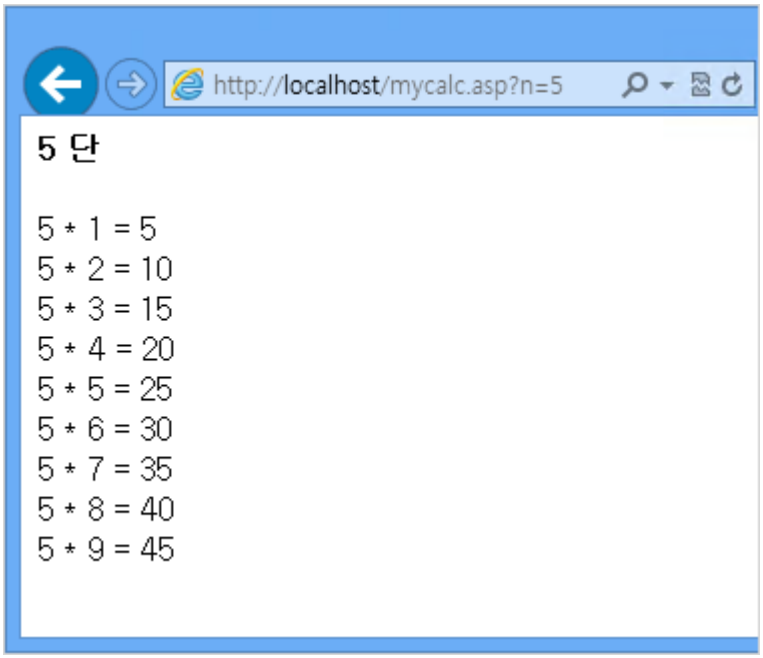
```
<!DOCTYPE html>
<html>

<head>
    <title>구구단</title>
</head>

<body>

<%
Dim number, i

number = Request.QueryString("n")
%>
```

```
<b><% Response.Write(number) %> 단</b><br />
<br />
<%
For i = 1 to 9
%>
    <%=number%> * <%=i%> = <%=number * i%><br />
<%
Next
%>

</body>

</html>
```

참고!	코드에 사용된 언어는 Visual Basic 스크립트다. 지면 관계상 이 언어를 자세하게 설명할 수는 없지만 ASP 예제는 더 이상 나오지 않으므로 C#의 For 문과 비교해서 흐름만 이해하고 넘어가자. 또한 이 파일을 비스타 운영체제 이상에서 "C:₩inetpub₩wwwroot" 폴더에 저장하려면 보안 문제로 인해 저장이 안 된다. 그런 경우에는 wwwroot 폴더에 현재 로그인 한 사용자 계정으로 쓰기 권한을 부여하거나 메모장 프로그램을 관리자 권한으로 실행시켜서 저장해야 한다.

웹 브라우저의 주소 표시줄에 "http://localhost/mycalc.asp?n=5"를 입력해서 방문하면 그림 20.59처럼 5단에 해당하는 구구단이 출력된다.

그림 20.59 asp로 구구단 출력



자세히 보면 일반적인 HTML 태그가 있는 파일인데, VBScript 코드를 넣는 부분은 <%, %>로 둘러

싸여 있는 것을 알 수 있다. 이제 웹 브라우저의 주소 표시줄에 입력된 URL에서 물음표 다음에 나오는 n의 값을 다른 숫자로 대체해보자. 그럼 mycalc.asp 파일 안에서 VBScript로 작성한 Request.QueryString 메서드를 이용해 구한 n의 값에 그 결과가 반영되고 For 루프를 돌아 구구단 결과가 그에 맞게 바뀐다. EXE를 만들어야 하는 CGI보다는 확장자만 asp로 바뀐 텍스트 파일에 스크립트 언어를 추가해서 동적 콘텐츠를 구성하는 편이 확실히 쉽다는 것을 알 수 있다.

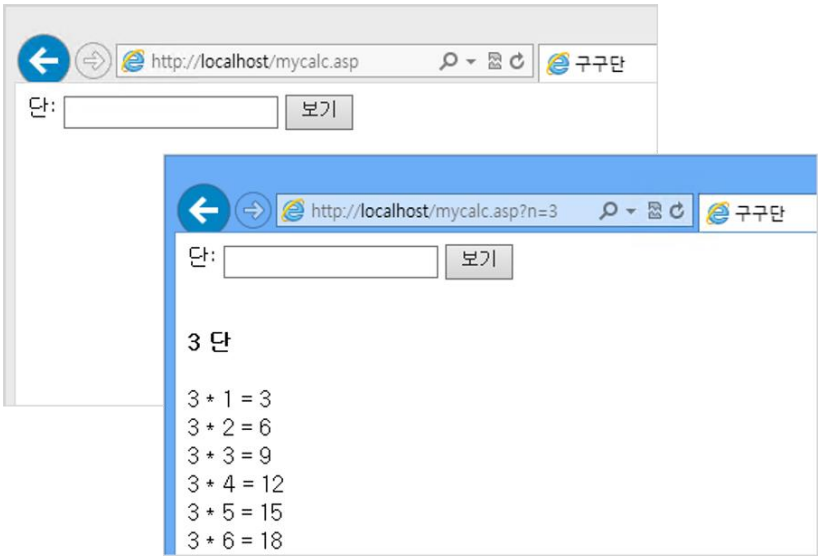
조금 더 편의 사항을 추가해 보자. 이 예제에 <FORM /> 태그를 추가하면 사용자에게서 입력값을 받게 할 수도 있다.

파일명: MyCalc.asp

```
<!DOCTYPE html>
<html>

<head>
    <title>구구단</title>
</head>

<body>

<form action="./mycalc.asp">
단: <input type="text" name="n" /> <input type="submit" value="보기" /><br />
</form><br />

<%
Dim number, i

number = Request.QueryString("n")

if number <> 0 then
%>

<b><% Response.Write(number) %>  단</b><br />
<br />
    <%
    For i = 1 to 9
    %>
    <%=number%> * <%=i%> = <%=number * i%><br />
    <%
    Next
End if
%>

</body>

</html>
```

변경 사항을 저장하고 다시 웹 브라우저로 방문하면 그림 20.60처럼 사용자 입력을 스크립트에서 받아 처리한다.

그림 20.60 <Form /> 입력 값을 asp 스크립트로 처리



이 동작이 바로 여러분이 사용하는 웹 사이트의 핵심 기능에 해당한다. 이 코드와 함께 6.8 '데이터베이스' 절에서 배운 데이터베이스 처리를 곁들이면 사용자 로그인, 게시판에 글쓰기, 덧글 달기 등의 기능을 제공하는 "웹 응용 프로그램"이 만들어지는 것이다.

ASP는 내부 언어로 VBScript 또는 자바스크립트를 사용하는데, 닷넷 프레임워크가 나오면서 마이크로소프트는 기존의 ASP 처리 방식을 근간으로 닷넷 언어를 사용할 수 있는 ASP.NET을 내놓는다. 기본 설치된 IIS에 ASP.NET을 활성화하려면 그림 20.58에서 "ASP.NET"으로 시작하는 항목을 선택해서 설치해야 한다. 화면에서는 "ASP.NET 3.5", "ASP.NET 4.5"가 있는데, IIS 관리자 사용법에 능숙하다면 개별적으로 선택해서 설치해도 되지만 여기서는 편의성을 위해 모두 선택해서 설치한다. 설치가 완료되면 mycalc.asp 파일을 ASP.NET의 확장자인 aspx에 맞춰 mycalc.aspx로 복사해서 테스트해보자. 웹 브라우저로 방문해 보면 여전히 기존의 ASP 스크립트가 동작하는 것을 확인할 수 있는데, ASP.NET이 기존의 ASP에 대한 하위 호환성을 제공하기 때문이다. 물론 언어를 닷넷으로 변경하는 것도 가능하다. 즉, aspx 파일에서 VBScript를 걷어내고 그 자리에 C# 언어를 사용할 수 있다.

파일명: MyCalc.aspx

```
<%@ Page language="c#" %>
<!DOCTYPE html>
<html>

<head>
    <title>구구단</title>
</head>

<body>
```

```
<form action="./mycalc.aspx">
단: <input type="text" name="n" /> <input type="submit" value="보기" /><br />
</form><br />

<%
int number, i;

string txt = Request.QueryString["n"];
if (string.IsNullOrEmpty(txt) == true)
{
    return;
}

number = Int32.Parse(txt);

if (number != 0)
{
%>

<b><% Response.Write(number); %> 단</b><br />
<br />
<%
    for (i = 1; i <= 9; i ++)
    {
%>
    <%=number%> * <%=i%> = <%=number * i%><br />
<%
    }
}
%>

</body>

</html>
```

닷넷 언어는 여러 가지가 있으므로 ASP.NET 웹 페이지에서 C# 언어를 사용하려면 파일의 첫 부분에 Page 지시자(Directive)를 통해 Language 속성으로 C#을 명시해야 한다. 그 이후의 코드는 전부 기존의 VBScript 문법에서 C# 문법으로 변경한 것에 불과하다. 이제 다시 웹 브라우저를 통해 "mycalc.aspx" 웹 페이지를 방문하면 mycalc.asp와 완전히 동일하게 동작하는 모습을 확인할 수 있다.

ASP와 ASP.NET에 대한 IIS 웹 서버 측의 처리 과정이 다르다는 점을 알아둘 필요가 있다. ASP의 경우 요청이 오면 스크립트 엔진이 매번 asp 파일을 해석(Interpret)해서 결과를 출력하는 반면, ASP.NET은 aspx 파일을 요청한 그 순간 하나의 클래스 파일로 변환한 후 컴파일까지 한 다음에야 처리하는 구조다. 그런데 어느 것이 더 처리 속도가 빠를까? ASP는 asp에 대한 요청이 들어오면 매번 스크립트 언어를 해석한다. 반면 ASP.NET은 개별 aspx 파일의 최초 요청에 대해서만 'aspx ➔ 클래스 ➔ 컴파일'로 이어지는 작업이 필요하기 때문에 느릴 수밖에 없다. 하지만 두 번째 요청부터는 상황이 역전된다. 여전히 ASP는 스크립트 언어의 한계상 같은 요청이 와도 asp 파일을 해석해야 하지만, ASP.NET은 최초 컴파일된 결과를 캐시로 저장해 두기 때문에 이후의 요청에 대해서는 곧바로 실행하는 단계로 넘어간다. 따라서 전체적인 웹 응용 프로그램의 처리 속도

는 ASP.NET 측이 월등하게 빠르다.

19.4.1 Form을 이용한 사용자 입력 처리: GET/POST

웹 브라우저에서 웹 서버로 사용자 입력을 전달하는 유형은 크게 GET과 POST로 나뉜다. 우선 GET 방식은 URL 자체에 Key=Value의 쌍으로 값을 전달하는 형식으로서, 여러 개의 값을 전달할 때는 '&' 기호를 이용해 붙인다.

```
1) http://localhost/test.aspx?name=anders
2) http://localhost/test.aspx?name=taylor&age=35
```

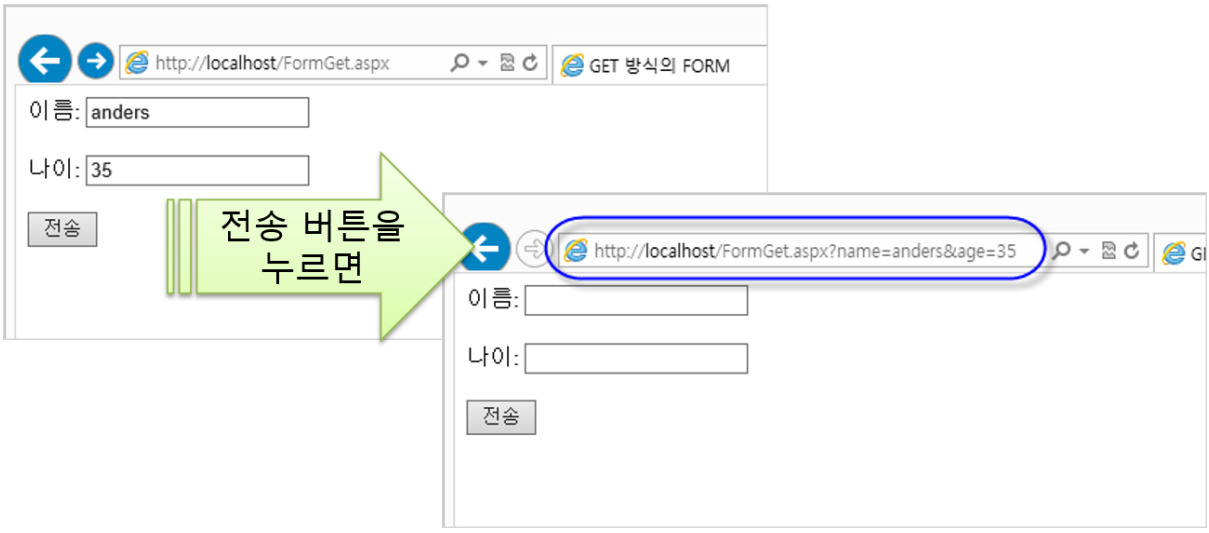
URL에 값을 함께 전달하기 때문에 웹 브라우저의 주소 표시줄에 직접 입력하는 경우도 있지만, <FORM /> 태그로도 GET 방식의 데이터를 전송할 수 있다. 가령 위의 2번 사례처럼 name과 age 값을 <FORM />을 이용해 전달하고 싶다면 다음과 같이 HTML 태그를 구성하면 된다.

파일명: FormGet.aspx

```
<!DOCTYPE html>
<html>

<head>
    <title>GET 방식의 FORM</title>
</head>

<body>

<form method="get" action="FormGet.aspx">
이름: <input type="text" name="name" /><br /><br />
나이: <input type="text" name="age" /><br /><br />
<input type="submit" value="전송" /><br />
</form><br />

</body>
</html>
```

역시 이 파일을 C:₩inetpub₩wwwroot에 저장하고 웹 브라우저를 이용해 http://localhost/FormGet.aspx로 방문한 후 이름과 나이에 임의의 값을 입력하고 "전송" 버튼을 눌러 보자. 그럼 그림 20.61에서 볼 수 있듯이 <INPUT name="name" />, <INPUT name="age" />의 값이 [name]=[value]의 쌍으로 웹 브라우저의 주소 표시줄에 나타나는 것을 볼 수 있다.

그림 20.61 GET 방식을 이용한 FORM 데이터 전송



이 상황을 직접 TCP 소켓을 열어 통신한다고 가정하면 다음과 같은 패킷들이 서로 오가는 것을 확인할 수 있다.

1) 최초 사용자가 FormGet.aspx 파일을 웹 서버로 호출했을 때		
웹 브라우저	데이터 전달 방향	웹 서버
GET /FormGet.aspx HTTP/1.1 Host: localhost	→	

2) 웹 서버의 FormGet.aspx 파일 응답		
웹 브라우저	데이터 전달 방향	웹 서버
	←	HTTP/1.1 200 OK Content-Type: text/html Server: Microsoft-IIS/8.0 Date: Wed, 17 Jul 2013 10:42:09 GMT Content-Length: 298 <html> <head> <title>GET 방식의 FORM</title> </head> .....[생략]..... </html>

3) 사용자가 웹 브라우저에서 FORM 값을 채우고 "전송" 버튼을 눌렀을 때

웹 브라우저	데이터 전달 방향	웹 서버
GET /FormGet.aspx?name=anders&age=35 HTTP/1.1 Host: localhost		

<FORM /> 내부의 <INPUT /> 데이터를 이렇게 URL에 붙여서 전송하려면 FORM 태그의 method 속성에 "GET"을 설정해야 하지만 이는 FORM 태그의 기본값이므로 생략해도 된다. 또한 action 속성을 이용하면 다른 URL로 값을 전달할 수 있다. 예를 들어, action="Process.aspx"로 설정하면 웹 브라우저는 다음과 같은 GET 요청으로 폼 데이터를 전송한다.

```
http://localhost/Process.aspx?name=.....&age=.....
```

물론 action 값도 생략할 수 있는데, 그러한 경우 기본적으로 현재 웹 페이지 URL로 다시 전송하게 된다. 따라서 다음의 세 가지 폼 태그는 완전히 같은 역할을 한다.

```
<form></form>
➔ <form method="get"></form>
➔ <form method="get" action="FormGet.aspx"></form>
```

FORM 데이터를 이처럼 GET 방식으로 웹 서버에 전송했으면 ASP.NET 측에서는 사용자가 입력한 이 변수들의 값을 가져올 수 있어야 한다. ASP.NET 웹 페이지는 기본적으로 System.Web.HttpRequest 타입의 Request 변수를 포함하고 있으며, 이것의 QueryString 컬렉션을 이용해 GET 요청 시 전달된 값을 가져올 수 있다.

```
string userName = Request.QueryString["name"];
string userAge = Request.QueryString["age"];
```

해당 변수의 값이 QueryString 컬렉션에 없으면 null이 반환된다는 사실도 알아두자. 이를 종합해서 사용자가 입력한 이름과 나이를 화면에 출력하는 FormGet.aspx 웹 페이지를 다음과 같이 만들 수 있다.

파일명: FormGet.aspx

```
<%@ Page Language="C#" %>
<!DOCTYPE html>
<html>

<%
string userName = Request.QueryString["name"];
string userAge = Request.QueryString["age"];
%>
```

```
<head>
    <title>GET 방식의 FORM</title>
</head>

<body>

<form method="get" action="FormGet.aspx">
이름: <input type="text" name="name" /><br /><br />
나이: <input type="text" name="age" /><br /><br />
<input type="submit" value="전송" /><br />
</form><br />

<%
if (string.IsNullOrEmpty(userName) == false &&
    string.IsNullOrEmpty(userAge) == false)
{
%>
사용자가 입력한 정보: <%=userName%>, <%=userAge %>
<%
}
%>

</body>

</html>
```

HTTP 통신은 웹 브라우저 측에서 데이터를 전송하기 위해 GET 말고도 POST라는 방식을 하나 더 추가해 뒀다. GET 방식은 사용자가 직접 웹 브라우저 주소 표시줄에 입력할 수 있다는 장점이 있지만, 달리 말하면 보안에 취약하다는 의미이기도 하다. 가령 웹 사이트에서 로그인을 시도하는 웹 페이지가 다음과 같이 GET 방식으로 만들어졌다고 가정해 보자.

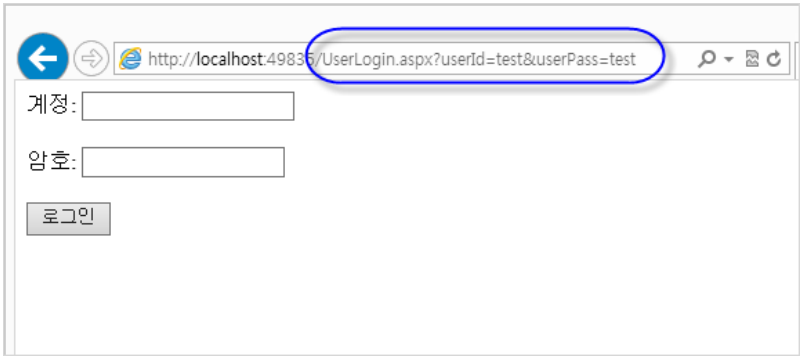
파일명: LoginUser.aspx

```
<%@ Page Language="C#" %>
<!DOCTYPE html>
<html>

<head>
    <title>웹 사이트 로그인 예제</title>
</head>

<body>

<form method="get" action="UserLogin.aspx">
계정: <input type="text" name="userId" /><br /><br />
암호: <input type="password" name="userPass" /><br /><br />
<input type="submit" value="로그인" /><br />
</form><br />

</body>

</html>
```

웹 브라우저를 통해 LoginUser.aspx에 방문하고 계정, 암호를 입력한 후 "로그인" 버튼을 누르면 그림 20.62처럼 사용자가 입력한 값이 그대로 보이는 문제점이 발생한다(이 순간 옆에 친구가 있

다면 여러분의 계정 정보를 알 수 있다).

그림 20.62 주소 표시줄에 보이는 FORM 전송 값



이런 문제를 해결하려면 <FORM />의 method 속성 값을 "post"로 바꾸면 된다.

```
<form method="post" ……>
</form>
```

웹 브라우저는 post로 설정된 Form 데이터를 GET 요청이 아닌 POST 요청으로 바꾸고 HTTP 헤더 다음의 본문 영역에 <INPUT /> 값을 Key=Value 쌍으로 넣어서 웹 서버로 보낸다.

```
POST /UserLogin.aspx HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 25
Host: 192.168.0.85

userId=test&userPass=test
```

GET 요청에서는 주소 표시줄을 통해 전송됐던 "userid=...&userPass=..." 값이 HTTP 본문에 담기도록 바뀌었고, 그 길이가 Content-Length에 지정돼 있다. 이처럼 웹 브라우저가 내부적으로 폼 데이터를 전송하기 때문에 GET 요청 시 주소 표시줄을 통해 민감한 데이터가 노출되는 문제가 사라진다. 아울러 ASP.NET 서버 측에서는 GET 요청에 대해 QueryString 컬렉션으로 값을 가져올 수 있었지만, POST로 전송된 데이터에 대해서는 Form 컬렉션으로 바꿔서 구해야 한다.

```
<%
string userName = Request.Form["userId"];
string userAge = Request.Form["userPass"];
%>
```

GET과 POST 가운데 어느 것이 좋다고 단정할 수는 없다. GET은 웹 브라우저의 주소 표시줄에 그 결과가 반영되므로 해당 주소만 보관해 둔다면 이후에 언제든지 같은 결과를 얻기 위해 그 페이지를 방문할 수 있다는 유연함이 있다. 따라서 각 웹 페이지에서 요구되는 특성에 맞게 적절한 요청 방식을 사용하는 것이 중요하다.

19.4.2 파일 업로드와 보안

POST 요청에서 또 한 가지 주목할 것은 Content-Type 헤더다. 이는 Form 태그가 가진 또 다른 속성인 enctype으로 설정되는데, 기본값은 "application/x-www-form-urlencoded"지만 다음과 같이 명시해도 무방하다.

```
<form enctype="application/x-www-form-urlencoded" method="post" ……>
</form>
```

대부분의 경우 "application/x-www-form-urlencoded" 기본값을 사용하지만, 딱 한 가지 예외적으로 웹 브라우저에서 "파일"을 전송하는 경우 enctype을 "multipart/form-data"로 바꿔야 한다. 즉, 웹 브라우저를 통해 파일을 전송하는 모든 웹 페이지는 POST 요청의 "multipart/form-data" 방식으로 전송하는 것이다.

다음은 POST 요청을 이용한 파일 전송 예제다.

파일명: FileSend.aspx

```
<%@ Page Language="C#" %>
<!DOCTYPE html>
<html>

<%

string tempPath = @"C:₩inetpub₩temp";

if (this.Request.Files.Count == 1)
{
    HttpPostedFile aFile = this.Request.Files[0];
    string fileName = System.IO.Path.Combine(tempPath, aFile.FileName);
    aFile.SaveAs(fileName);
}

%>

<head>
    <title>파일 업로드 예제</title>
</head>

<body>

<form method="post" action="FileSend.aspx" enctype="multipart/form-data">
첨부할 파일: <input type="file" name="myfile" /><br /><br />
<input type="submit" value="파일 업로드" /><br />
</form><br />
```

```
</body>
</html>
```

사용자가 첨부한 파일을 <INPUT type="file" ..... />을 통해 웹 서버로 전송하면 ASP.NET에서는 Request 속성의 Files 컬렉션을 이용해 HttpPostedFile 타입의 인스턴스로 다룰 수 있다. FileSend.aspx 예제에서는 전송된 파일 데이터를 C:₩inetpub₩temp 폴더에 저장하는데, 직접 실습해 보면 다음과 같은 예외를 만나는 경우도 있을 것이다.

Access to the path 'C:₩inetpub₩temp₩test.txt' is denied.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.UnauthorizedAccessException: Access to the path 'C:₩inetpub₩temp₩test.txt' is denied.

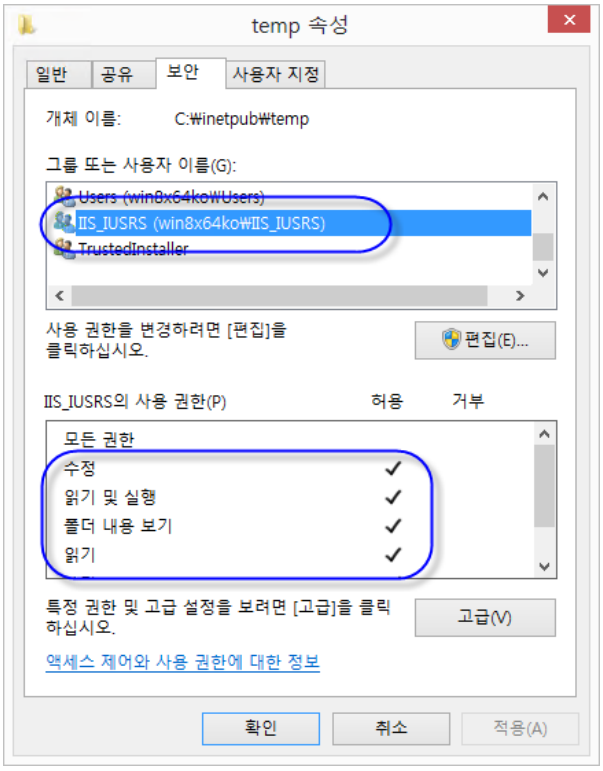
바로 이전 절에서 언급한 '20.3 서비스 응용 프로그램' 절의 내용을 알고 있어야만 이 오류의 내용을 이해할 수 있다. 원격지에서 들어오는 HTTP 요청을 처리하는 웹 서버는 "서비스 응용 프로그램"의 한 사례다. 즉, 로그인한 사용자가 직접 실행한 프로그램이 아니기 때문에 미리 웹 서버 프로세스(w3wp.exe)에 대한 사용자 계정이 할당된다. 문제는 윈도우 서버는 보안상의 이유로 기본적으로 권한이 축소된 특별한 계정으로 w3wp.exe 프로세스를 실행한다는 점이다. 바로 그 계정이 "C:₩inetpub₩temp" 폴더에 대해 쓰기 권한이 없기 때문에 "FileSend.aspx"에서 "Access to the path ... is denied"라는 오류가 발생하는 것이다.

이 오류를 해결하는 데는 두 가지 방법이 있다.

1. C:₩inetpub₩temp 폴더에 w3wp.exe 프로세스를 실행한 계정에 대해 "쓰기" 권한을 부여
2. w3wp.exe 프로세스의 실행 계정을 C:₩inetpub₩temp 폴더에 쓰기 권한을 가진 것으로 교체

보안상의 이유로 인해 보통 첫 번째 방법이 선호된다. 따라서 업로드되는 파일을 저장하기 위한 용도로 사용되는 폴더는 그림 20.63에서 보는 바와 같이 별도로 윈도우 탐색기를 이용해 w3wp.exe의 기본 사용자 계정을 대표하는 "IIS_IUSRS"를 추가해 "수정(Modify)" 권한을 허용해야 한다.

그림 20.63 IIS_IUSRS 계정으로 쓰기 권한 설정



이렇게 변경하고 나면 FileSend.aspx의 파일 업로드가 정상적으로 수행되고, c:₩inetpub₩temp 폴더에 사용자가 선택한 파일이 저장되는 것을 확인할 수 있다.

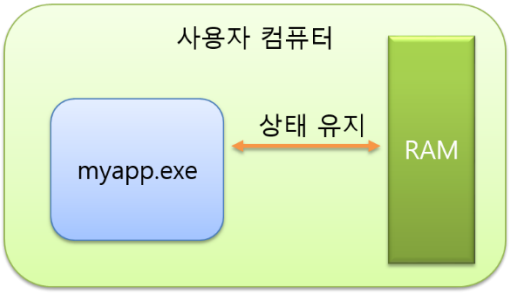
파일 업로드를 구현할 때 한 가지 주의할 점이 있다면 업로드되는 파일의 저장 경로가 웹 응용 프로그램의 루트 및 그 아래가 돼서는 안 된다는 것이다. 예를 들어, 파일의 저장 경로를 "C:₩inetpub₩wwwroot"로 설정한다면 악의적인 목적을 가진 사용자가 임의의 코드를 aspx 웹 페이지에 담아 웹 서버 측에 전송할 수 있고, 따라서 언제든 웹 브라우저로 실행할 수 있는 상태가 된다. 설상가상으로 이렇게 만들어진 웹 응용 프로그램의 w3wp.exe 실행 권한이 "Local SYSTEM"으로 할당돼 있다면 해당 웹 서버가 크래커에 의해 완전히 장악되는 것은 시간 문제에 불과하다.

참고!	사실 웹 서버에서 "Access ..... denied"가 발생하는 거의 모든 원인은 w3wp.exe를 실행하는 계정의 약한 권한 때문이다. 이로 인해 발생하는 설정이 귀찮을 수도 있지만 그렇다고 해서 w3wp.exe를 "Local SYSTEM"과 같은 막강한 권한을 가진 계정으로 실행하는 것은 최대한 심사숙고해야 한다. 본문에서 예를 든 것처럼 파일 업로드 폴더를 웹 루트 폴더로 지정하는 것과 같은 실수를 개발자는 언제든 할 수 있고, 이런 실수로 인한 피해를 최소화할 수 있는 마지막 보루가 바로 w3wp.exe의 실행 권한이기 때문이다.

19.4.3 상태 관리

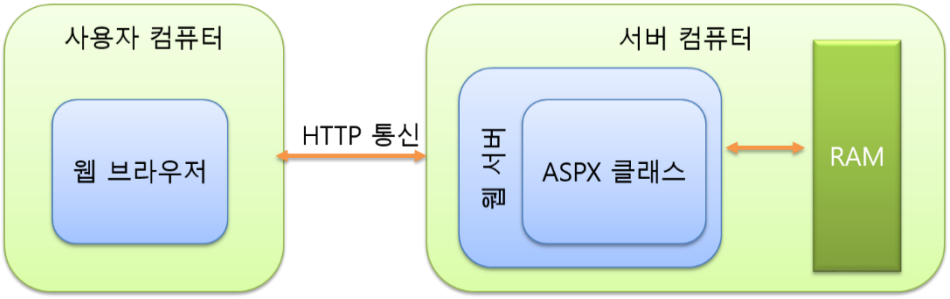
콘솔, 윈폼, WPF와 같은 데스크톱용 응용 프로그램으로 프로그래밍 공부를 처음 시작하는 경우 일부 프로그래머들이 ASP.NET 웹 폼 응용 프로그램을 만들면서 겪게 되는 공통적인 문제가 하나 있다. 이는 대부분 웹 서버와 웹 브라우저가 HTTP를 이용해 통신한다는 근본적인 환경상의 차이를 이해하지 못해서 발생하는 문제점이다. 데스크톱 응용 프로그램들은 그림 20.64처럼 한 대의 PC에서 EXE가 실행되는 동안 사용되는 모든 변수의 상태가 메모리에 저장돼 유지된다.

그림 20.64 데스크톱 응용 프로그램의 실행 구조



반면 웹 응용 프로그램은 그림 20.65에서 보는 바와 같이 클라이언트 측의 웹 브라우저로부터 전달된 요청이 처리되는 순간 클래스가 서버 측 메모리에 올라오고 HTML 콘텐츠를 생성해서 반환한 다음 메모리에서 삭제된다.

그림 20.65 웹 응용 프로그램의 실행 구조



이 차이점을 이해하기 위해 제곱근을 구하는 sqrt.aspx 웹 페이지를 다음과 같이 만들어 보자.

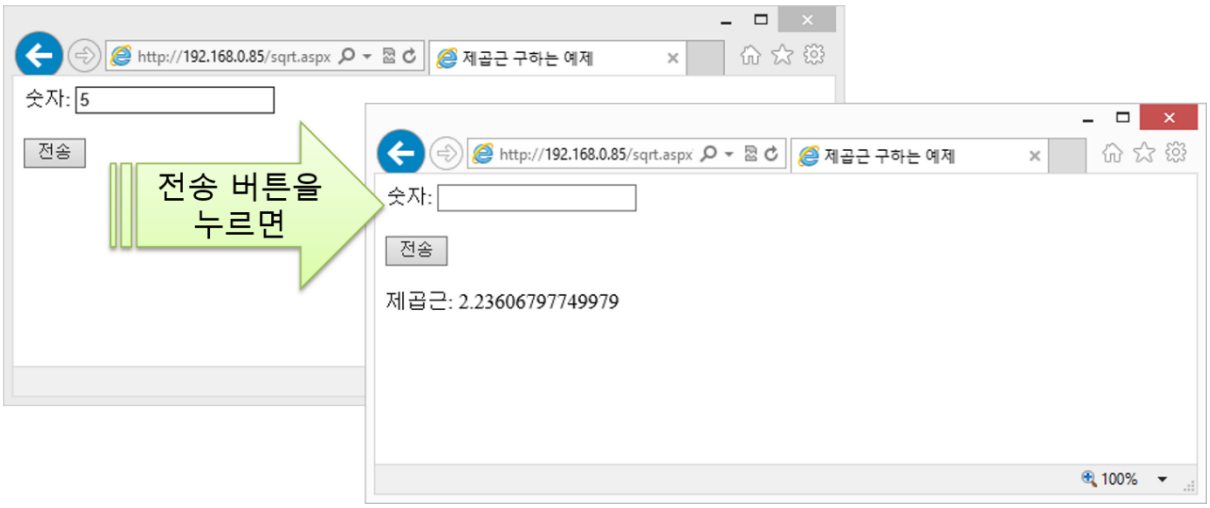
파일명: sqrt.aspx

<%@ Page Language="C#" %>
<!DOCTYPE html>
<html>
 	
<%
string text = Request.QueryString["inputValue"];
double result = 0;

```
if (string.IsNullOrEmpty(text) == false)
{
      int number = Int32.Parse(text);
      result = Math.Sqrt(number);
}
%>

<head>
      <title>제곱근 구하는 예제</title>
</head>

<body>

<form method="get" action="sqrt.aspx">
숫자: <input type="text" name="inputValue" /><br /><br />
<input type="submit" value="전송" /><br />
</form><br />

<%
if (result != 0)
{
%>
제곱근: <%=result%>
<%
}
%>

</body>
</html>
```

작성된 예제 페이지를 웹 브라우저로 방문하면 그림 20.66처럼 동작하는 것을 확인할 수 있다.

그림 20.66 폼을 submit 한 경우



그런데 그림 20.66에서 뭔가 아쉬운 부분이 있지 않을까? 전통적인 윈도우 폼 응용 프로그램을 다뤄 본 개발자라면 폼이 전송된 후에 보여지는 화면의 "숫자: "란에도 당연히 "5"가 출력되리라 생각할 수 있다. 실제로 이 프로그램이 윈도우 폼 응용 프로그램이었다면 그렇게 동작했을 테지만, 웹 응용 프로그램에서는 이것이 불가능하다. 그 이유를 알고 싶다면 웹 브라우저 동작을 자세하게 살펴볼 필요가 있다. 우선 "전송" 버튼이 눌린 경우 웹 브라우저는 웹 서버로 TCP 연결을

맺고 GET 요청을 전송한다. 웹 서버는 해당 GET 요청에 따라 aspx 웹 페이지를 처리하고 그 결과를 HTML 태그가 담긴 문자열로 반환한다. 응답을 받은 웹 브라우저는 방금 전까지 화면에 보인 내용을 모두 지우고, 새롭게 받은 HTML 태그 내용을 기반으로 화면을 다시 구성한다. 여기서 중요한 것은 웹 브라우저 입장에서 보면 폼을 전송하기 전에 보여준 HTML 내용과 폼을 전송한 후에 새롭게 보이게 될 HTML 내용과는 어떠한 연관성도 없다는 점이다.

이처럼 HTTP 통신은 기본적으로 상태 관리가 되지 않는(Stateless) 방식이다. 만약 상태 관리를 하고 싶다면(Stateful) HTTP 통신 규약 범위 내에서 제공되는 수단을 최대한 활용해 개발자가 직접 구현해야 한다. 예를 들어, sqrt.aspx의 경우 FORM 처리가 된 후 숫자 값에 사용자가 입력한 값을 보여주려면 다음과 같은 코드를 추가해야 한다.

파일명: sqrt.aspx

```
.....[생략].....

<%
string text = Request.QueryString["inputValue"];
.....[생략].....
%>

.....[생략].....

<form method="get" action="sqrt.aspx">
숫자: <input type="text" name="inputValue" value="<%=text%>" /><br /><br />

.....[생략].....

</html>
```

변경된 sqrt.aspx 페이지를 웹 브라우저로 방문하면 사용자가 입력했던 숫자 값이 그대로 제곱근이 구해진 화면에서도 입력 상자에 보이는 것을 확인할 수 있다.

웹 응용 프로그램을 만드는 일이 어렵거나 자잘한 반복 작업이 많이 필요한 주요 요인이 바로 "상태 관리"에 있다. HTTP 통신 자체가 상태를 관리하지 않는 프로토콜인데, 개발자가 상태 관리를 하는 코드를 일일이 만들어야 하는 시점부터 귀찮은 작업이 시작된다.

19.4.4 ASP.NET 웹 폼 응용 프로그램

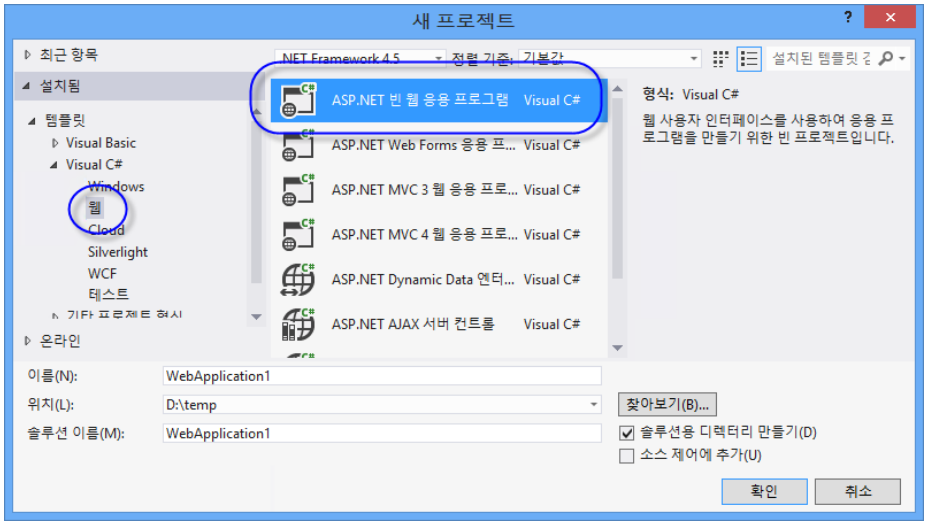
지금까지 설명한 aspx 웹 페이지는 ASP.NET을 ASP처럼 사용한 방식이었다. 이것의 장점은 기존 ASP 웹 페이지를 ASP.NET으로 빠르게 이전할 수 있다는 것인데, 엄밀히 말해서 일반적인 닷넷 프로그래머는 잘 사용하지 않는 방식이다. 왜냐하면 ASP.NET은 마치 "윈도우 폼" 응용 프로그램을 다루듯이 웹 응용 프로그램을 만들 수 있는 "웹 폼(web form)"을 지원하기 때문인데, 이 방식을 활용하면 좀 더 빠르게 웹 사이트를 제작할 수 있다.

웹 폼 개발의 생산성 향상이 높은 데에는 비주얼 스튜디오 개발 도구와 잘 통합돼 있다는 이유

도 한몫한다. 따라서 이번 절을 실습하려면 반드시 비주얼 스튜디오가 설치돼 있어야 한다.

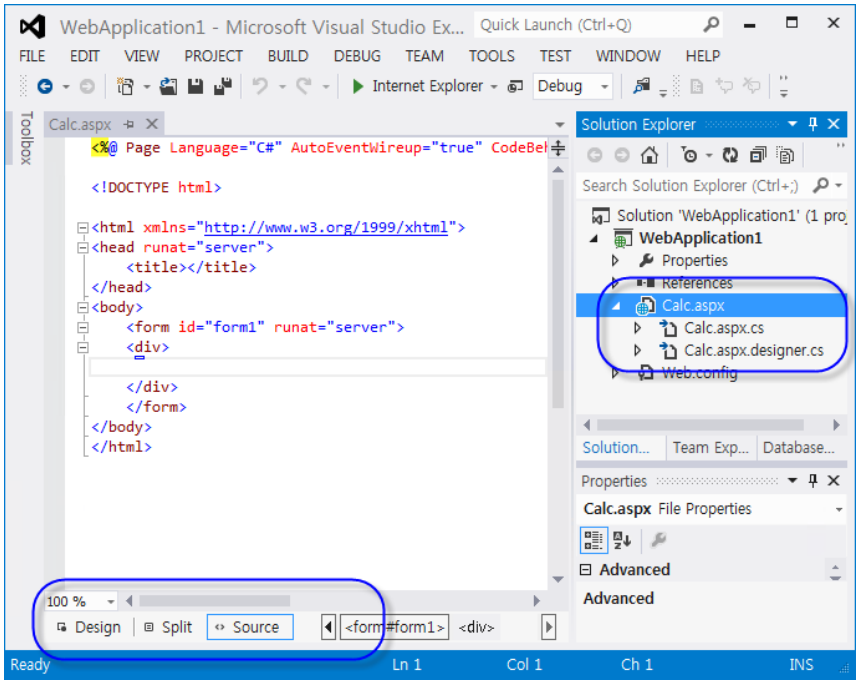
그럼 첫 번째 ASP.NET 웹 폼 응용 프로그램을 만들어 보자. 비주얼 스튜디오를 실행하고 "파일(FILE)" / "새 프로젝트(New Project)"를 선택하면 그림 20.67과 같은 화면이 나타나고 좌측의 "설치됨(Installed)" / "템플릿(Templates)" / "Visual C#" / "웹(Web)" 범주에 들어가면 보이는 우측 목록의 "ASP.NET 빈 웹 응용 프로그램(Empty Web Application)"을 선택한다.

그림 20.67 ASP.NET 웹 폼 프로젝트 선택



기본 생성된 프로젝트를 보면 웹 프로그램을 위한 어셈블리가 참조됐고 Web.config 파일이 추가된 정도다. web.config은 5.2.3 '응용 프로그램 구성 파일: app.config' 절에서 설명한 app.config과 이름만 다를 뿐 같은 역할을 한다. 이제 웹 폼을 하나 추가해 보자. 솔루션 탐색기에서 프로젝트를 마우스 오른쪽 버튼으로 누르고 "추가(Add)" / "새 항목(New Item)"을 선택하면 "웹 폼(Web Form)" 항목이 보이는데, 파일명을 "Calc.aspx"로 변경해서 추가한다. 웹 폼 파일이 생성되면 비주얼 스튜디오는 그림 20.68처럼 보일 것이다.

그림 20.68 웹 폼 추가

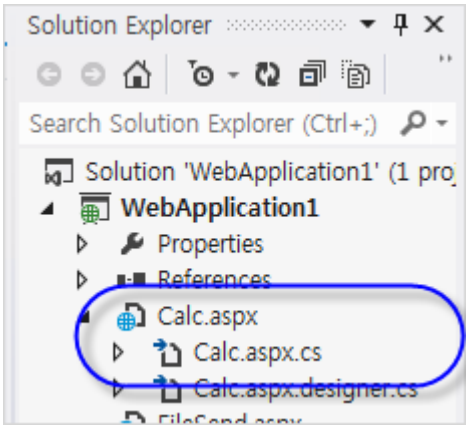


설명할 내용이 많지만 우선 솔루션 탐색기에 새롭게 포함된 세 파일을 먼저 보자. "Calc.aspx" 파일에는 HTML 소스코드가 포함돼 있다. 이전에 실습한 MyCalc.aspx 파일과 Calc.aspx의 역할은 완전히 동일하다. 실제로 calc.aspx 상단에 보면 Page 지시자가 다음과 같이 설정된 것을 볼 수 있다.

```
<%@ Page
Language="C#"
AutoEventWireup="true"
CodeBehind="Calc.aspx.cs"
Inherits="WebApplication1.Calc" %>
```

Language 값이 C#으로 기본 설정돼 있기 때문에 Calc.aspx에 포함된 HTML 소스에도 <%, %> 기호를 이용해 직접 C# 소스코드를 추가할 수 있다. 생소한 속성인 CodeBehind 값이 Calc.aspx.cs라고 설정돼 있고, 이는 그림 20.69와 같이 솔루션 탐색기에 포함된 "Calc.aspx"의 하위 항목에 묶여 있는 Calc.aspx.cs를 가리킨다.

그림 20.69 솔루션 탐색기에도 나오는 CodeBehind 파일



이 파일을 편집 창에서 확인하면 예제 20.21의 내용을 볼 수 있다.

예제 20.21 Calc.aspx.cs 파일의 내용

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace WebApplication1
{
    public partial class Calc : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
        }
    }
}
```

현재 만들어진 기본 상태 그대로 프로젝트를 빌드하면 /bin 폴더 아래에 WebApplication1.dll 어셈블리가 만들어진다. 해당 DLL 파일에는 웹 프로젝트에 포함된 각종 *.aspx.cs 파일들이 컴파일되어 들어간다. 즉, Calc.aspx.cs 파일은 컴파일되면 WebApplication1.dll로 통합된다.

Calc.aspx의 Page 지시자에 Inherits 값이 "WebApplication1.Calc"를 가리키고 있다는 점을 주목하자. ASP.NET은 이처럼 CodeBehind와 Inherits가 지정된 aspx 웹 페이지의 요청이 들어오면 다음과 같은 순서로 작업을 처리한다.

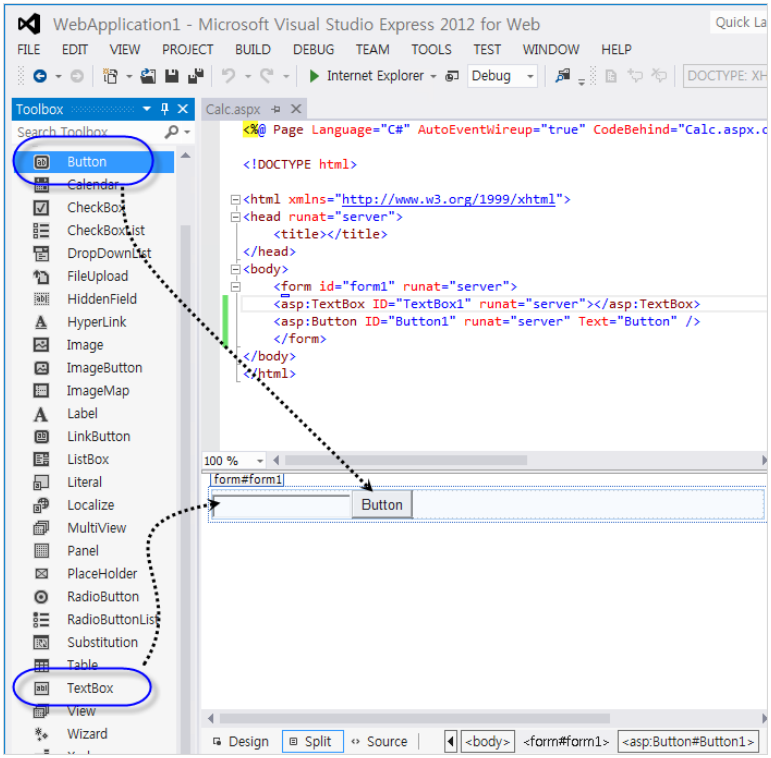
1. Inherits로 지정된 타입을 상속받아 aspx 자체를 클래스로 만들어 생성한다. 즉, Calc.aspx.cs에 정의됐던 Calc 타입을 상속받는 Calc.aspx 클래스가 별도로 만들어진다.
2. aspx 클래스를 컴파일한다.
3. 컴파일된 aspx 타입을 생성하고 웹 폼의 요청을 전달해 실행한다.
4. 웹 폼이 반환한 HTML 응답을 웹 브라우저로 전송한다.

Calc.aspx.cs에 정의된 Calc 타입이 System.Web.UI.Page를 상속받았다는 사실도 알아두자. Page 타입은 웹 페이지 처리와 관련된 모든 기반 코드를 담고 있다. 보통 웹 폼 개발자는 Page_Load 이벤트 처리기에서 각 웹 페이지 요청 시 처음 실행될 코드를 넣어둔다. 이는 마치 윈도우 폼에서 Form 타입의 Load 이벤트를 정의한 것과 유사한 역할을 한다.

이제 다시 그림 20.68로 돌아가 Calc.aspx 편집 창의 하단 영역에 있는 세 개의 탭을 보자. 그림 20.68에서는 기본적으로 "소스" 탭이 선택돼 있는데, 이는 편집 창 전체 영역을 HTML 텍스트로 보여주는 역할을 한다. "나누기(Split)", "디자인(Design)" 탭을 각각 눌러 보면 어떤 역할을 하는지 금방 알 수 있다. 말 그대로 나누기 탭은 편집 영역을 반으로 나눠 상단은 HTML 소스, 하단은 해당 HTML 소스가 웹 브라우저에 표시되는 모습의 디자인 창을 보여준다. 그리고 "디자인" 탭은 편집 영역 전체를 디자인 모드로 바꾼다.

간단하게 웹 폼을 어떻게 이용할 수 있는지 실습해 보자. 이전에 만든 MyCalc.aspx 구구단과 동일하게 Calc.aspx 웹 폼을 작성해 볼 텐데, 우선 비주얼 스튜디오의 도구 상자(Toolbox)로부터 TextBox와 Button 하나를 추가하는 것으로 시작할 수 있다. 그림 20.70과 같이 디자인 창으로 끌어다 놓거나, 아니면 HTML 소스 창의 <form ...>과 </form> 사이에 놓아도 된다.

그림 20.70 웹 폼에 Button, TextBox를 추가



어느 쪽에 추가했든지 변경된 calc.aspx 파일을 저장하고 나면 HTML 소스와 디자인 창의 내용이 일치하는 것을 확인할 수 있다. 결과적으로 Calc.aspx의 HTML 소스에 추가된 내용은 다음과

같다.

```
<%@ Page Language="C#" AutoEventWireup="true"
 CodeBehind="Calc.aspx.cs" Inherits="WebApplication1.Calc" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
        <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
        <asp:Button ID="Button1" runat="server" Text="Button" />
    </form>
</body>
</html>
```

표준 HTML 태그 외에 asp 접두사와 함께 runat="server" 옵션이 붙은 확장 태그는 오직 ASP.NET 에서만 인식되는데, 이를 가리켜 "ASP.NET 웹 폼 컨트롤"이라 한다. 여기서 사용된 <asp:TextBox />, <asp:Button />의 경우에는 System.Web.UI.WebControls 네임스페이스에 정의된 TextBox, Button 컨트롤 타입에 각각 대응된다. 아울러 태그에 포함된 ID의 값이 TextBox1과 Button1로 설정된 것을 보자. runat="server" 옵션과 함께 설정되는 ID들은 Calc.aspx.designer.cs 파일에 반영되므로 해당 파일을 열어 보면 각 ID에 대응하는 변수가 선언돼 있음을 확인할 수 있다.

```
namespace WebApplication1 {

    public partial class Calc {
        protected global::System.Web.UI.HtmlControls.HtmlForm form1;
        protected global::System.Web.UI.WebControls.TextBox TextBox1;
        protected global::System.Web.UI.WebControls.Button Button1;
    }
}
```

TextBox1, Button1 외에 form1도 포함돼 있는데, 마찬가지로 aspx 파일에는 <form id="form1" ..... runat="server" />로 정의돼 있기 때문에 designer.cs 파일에도 함께 존재한다. 기본적으로 Calc.aspx에 추가된 모든 <asp:..... /> 확장 태그는 Calc.aspx.designer.cs 파일에 반영된다고 보면 된다.

designer.cs 파일 내에 정의된 Calc 타입에 partial 예약어가 적용돼 있다는 것도 눈여겨볼 필요가 있다. 따라서 designer.cs의 내용은 컴파일 시에 Calc.aspx.cs에 포함된 Calc 타입과 합쳐진다. 이러한 구조는 윈도우 폼 응용 프로그램에서 다룬 그림 20.5 Form1.Designer.cs의 폼 디자인과 매우 유사하다. 정리하자면 aspx 파일에 추가된 <asp:..... /> 웹 폼 컨트롤은 designer.cs 파일에 반영되고, 이 파일의 클래스는 aspx.cs에 정의된 클래스와 부분(partial) 클래스의 관계에 놓여 있기 때문에 Calc.aspx.cs의 코드에서 aspx에 정의된 웹 폼 컨트롤의 ID를 직접 접근하는 것이 허용된다. 게다가 당연히 HTML의 ID 값을 바꾸면 Calc.aspx.designer.cs의 소스코드에 있는 변수명도 함

께 바뀐다.

이뿐만 아니라 심지어 이벤트 처리기 구조도 윈도우 폼과 유사하게 구현돼 있다. 웹 폼 디자인 화면에서 <asp:Button />에 해당하는 버튼 컨트롤을 마우스로 두 번 누르면 Calc.aspx.cs에는 예제 20.22와 같이 Click 이벤트 처리기 코드가 추가된다. 따라서 사용자가 웹 브라우저에서 버튼을 누르면 웹 서버 측으로 요청이 전달되고, Calc 타입이 생성된 후 Button1_Click 이벤트 처리기가 실행된다. 물론 사용자가 입력한 TextBox1의 내용은 Calc.aspx.designer.cs에 정의됐던 그 변수명(TextBox1)의 Text 속성 값을 이용해 가져올 수 있다.

예제 20.22 웹 폼 버튼의 이벤트 처리기

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace WebApplication1
{
    public partial class Calc : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
        }

        protected void Button1_Click(object sender, EventArgs e)
        {
            int number = Int32.Parse(TextBox1.Text);
        }
    }
}
```

계속해서 구구단의 내용을 HTML로 출력하기 위해 도구 상자에서 Literal 항목을 웹 폼에 추가해 보자.

```
<%@ Page Language="C#" AutoEventWireup="true"
……[생략]……
<body>
    <form id="form1" runat="server">
        <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
        <asp:Button ID="Button1" runat="server" Text="Button" />
        <br />
        <asp:Literal ID="Literal1" runat="server"></asp:Literal>
    </form>
</body>
</html>
```

마지막으로 Button이 눌렸을 때 Literal1 영역에 구구단의 내용이 출력되는 코드를 다음과 같이 작성할 수 있다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace WebApplication1
{
    public partial class Calc : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
        }

        protected void Button1_Click(object sender, EventArgs e)
        {
            int number = Int32.Parse(TextBox1.Text);

            StringBuilder sb = new StringBuilder();

            for (int i = 1; i < 10; i++)
            {
                sb.AppendFormat("{0} * {1} = {2}<br />", number, i, number * i);
            }

            Literal1.Text = sb.ToString(); // HTML 텍스트를 Literal1 영역에 출력
        }
    }
}
```

확인을 위해 편집 창에 Calc.aspx를 띄운 상태에서 F5 키를 눌러 실행해 보자. 그럼 웹 브라우저가 실행되면서 자동으로 "http://localhost:[임의의 포트번호]/Calc.aspx" 주소로 이동하게 되고 곧바로 동작을 테스트해 볼 수 있다.

이번에는 ASP 방식으로 만들었던 또 다른 예제인 sqrt.aspx도 ASP.NET 웹 폼으로 만들어 보자. 그림 20.68에서 설명한 것처럼 sqrtform.aspx 웹 폼을 추가하고, 아래의 컨트롤을 추가한다.

표 20.11 제곱근을 구하는 웹 폼의 컨트롤 속성

웹 폼 컨트롤	변경된 속성	값
TextBox	ID	txtNumber
Button	ID	btnCalc
Label	ID	lblResult
	Text	(빈 문자열)

이를 반영한 aspx 파일은 다음과 같다.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="sqrtform.aspx.cs"
```

```
Inherits="WebApplication1.sqrtform" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>제곱근 구하는 웹 폼 예제</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        숫자: <asp:TextBox ID="txtNumber" runat="server"></asp:TextBox><br /><br />
        <asp:Button ID="btnCalc" runat="server" Text="전송" /><br /><br />
        <asp:Label ID="lblResult" runat="server" Text=""></asp:Label>
    </div>
    </form>
</body>
</html>
```

비주얼 스튜디오 디자인 창에서 <asp:Button /> 영역을 마우스로 두 번 클릭한다. 그러면 sqrtform.aspx.cs 파일이 열리면서 btnCalc_Click 이벤트 처리기가 추가되고, 여기에 제곱근을 구하는 코드를 작성한다.

```
using System;

namespace WebApplication1
{
    public partial class sqrtform : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {

        }

        protected void btnCalc_Click(object sender, EventArgs e)
        {
            int number = Int32.Parse(txtNumber.Text);
            lblResult.Text = "제곱근: " + Math.Sqrt(number);
        }
    }
}
```

코드를 실행해서 결과를 확인해 보자. 이전에 "상태 관리"까지 구현한 sqrt.aspx의 마지막 코드 예제와 완전히 동일하게 실행되는 것을 볼 수 있다.

이쯤에서 기존의 ASP 방식으로 만든 MyCalc.aspx / sqrt.aspx 웹 페이지와 Calc.aspx / sqrtform.aspx 웹 폼 페이지의 차이점을 비교해 보자. 어떤 형식이든 상관없이 여러분은 원하는 동작이 구현된 웹 페이지를 만들 수 있었지만 ASP 방식의 구현에는 HTML과 C# 소스코드가 뒤섞여 있는 반면 ASP.NET 웹 폼으로 구현할 때는 HTML과 C# 소스코드가 분리됐다는 특징이 있다. 전자의 경우를 가리켜 스파게티 코드라고 하는데, 임의의 변경을 손쉽게 할 수 있다는 장점은 있지만 웹 응용 프로그램의 규모가 커질수록 프로그램을 유지보수하기가 어려워진다는 것과 디자이

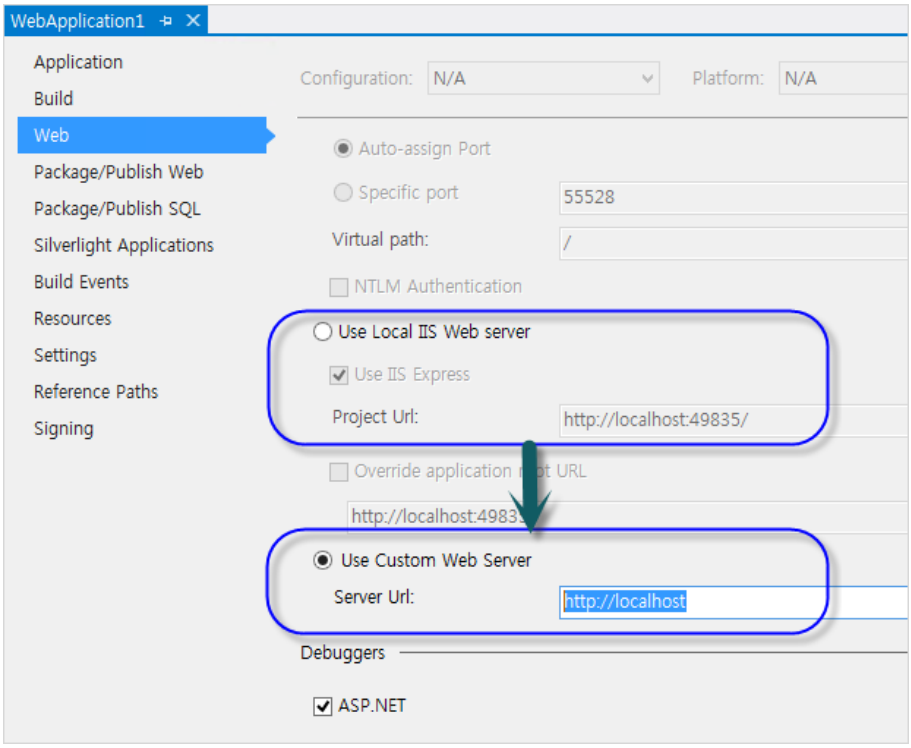
너와 개발자의 협업을 거의 불가능하게 만들어 버린다는 단점이 있다. 일반적으로 ASP.NET 응용 프로그램이라고 하면 "웹 폼"을 이용한 프로그램을 의미하며, 실제로 대부분의 ASP.NET 프로그래머들이 "웹 폼"을 이용해 웹 사이트를 만든다.

19.4.5 배포 및 서비스

비주얼 스튜디오에서 F5를 눌러 실행한 인터넷 익스플로러의 주소 표시줄을 보면 "http://localhost:[포트번호]/...aspx"와 같은 형식으로 돼 있다. 비주얼 스튜디오는 편의상 현재 컴퓨터에 IIS가 설치돼 있느냐와 상관없이 웹 응용 프로그램을 실행할 수 있게 IIS의 간편 버전인 IISExpress.exe 프로세스를 통해 ASP.NET 응용 프로그램을 호스팅한다. ASP.NET 측면에서 봤을 때 IIS와 IISExpress 간의 차이점은 거의 없지만 한 가지 주의해야 할 점이 있다면 "프로세스 실행 권한"이 다르다는 것이다. IIS는 기본적으로 축소된 권한으로 실행되지만, IISExpress는 현재 로그인한 사용자 계정의 권한을 따르므로 보안과 관련된 작업을 했을 때 IISExpress 상에서는 정상적으로 동작하던 것이 IIS에서 동작시킬 때는 오류가 발생할 수 있다.

이런 차이를 미리 방지하려면 비주얼 스튜디오에서의 웹 프로젝트 실행을 IISExpress가 아닌 IIS 상에서 동작하도록 바꿔야 한다. 그러자면 당연히 해당 컴퓨터에는 IIS 서비스가 미리 설치돼 있어야 하고, 여러분이 만든 ASP.NET 웹 프로젝트가 생성된 폴더를 그림 20.56에서 설명한 IIS 웹 사이트의 "실제 경로"로 지정해야 한다. 일반적으로 프로젝트 파일(.csproj)이 있는 폴더가 기준이 되는데, 이 경로가 "D:₩temp₩WebApplication1₩WebApplication1"이라면 그림 20.56의 "실제 경로"란에 이 값을 그대로 넣으면 된다. 그런 다음, 그림 20.71처럼 비주얼 스튜디오에서 웹 프로젝트의 속성 창을 열고 "웹(Web)" 범주에서 "사용자 지정 웹 서버 사용(Use Custom Web Server)" 옵션을 선택한 후 "서버(Server) Url"을 "http://localhost"로 지정해야 한다.

그림 20.71 IISExpress에서 IIS로 호스팅 환경을 전환



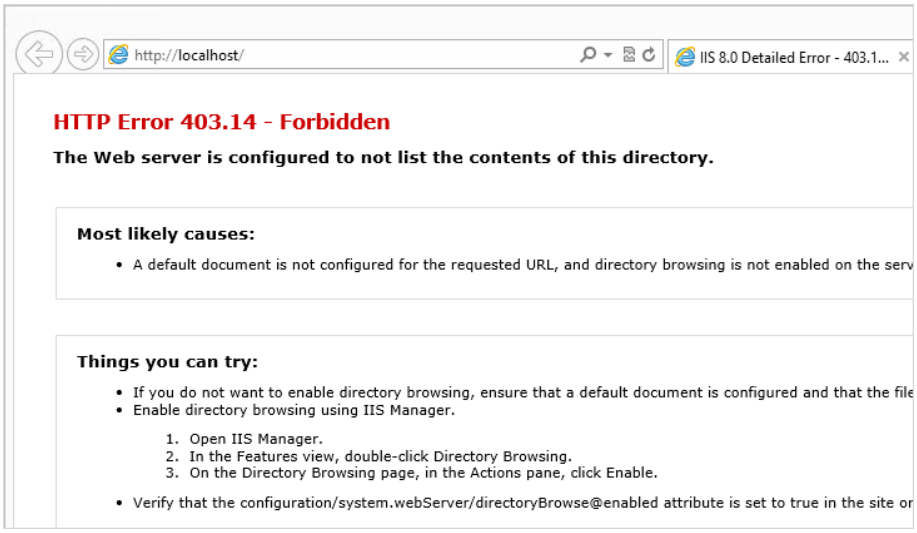
이렇게 변경하고 F5를 눌러 실행해 본다. 윈도우 비스타 이후의 운영체제를 사용하는 독자라면 로컬 관리자(Administrator) 계정으로 로그인하지 않은 경우 UAC(사용자 계정 제어: User Access Control)로 인해 다음과 같은 오류 메시지가 나타날 수 있다.

> 웹 서버에서 디버깅을 시작할 수 없습니다. 웹 서버 프로세스를 디버깅할 권한이 없습니다. 웹 서버와 동일한 사용자 계정으로 실행하거나 관리자 권한이 있어야 합니다.
>
> Unable to start debugging on the web server. You do not have permissions to debug the web server process. You need to either be running as the same user account as the web server, or have administrator privilege.

이는 웹 애플리케이션이 수행될 w3wp.exe는 비주얼 스튜디오의 실행 프로세스인 devenv.exe와는 다른 윈도우 세션(Session)에서 실행되기 때문에 일반 사용자 계정 권한으로는 F5를 눌러 디버깅 상태로 열 수 없기 때문이다. 이 문제를 해결하려면 비주얼 스튜디오를 "관리자 권한"으로 실행한다.

비주얼 스튜디오가 "관리자 권한"으로 실행됐다면 F5 키를 눌러 다시 실행해 보자. 그림 20.67과 같은 식으로 프로젝트를 생성했다면 그림 20.72와 같은 오류가 발생할 것이다.

그림 20.72 HTTP 403.14 오류



이런 오류가 발생하는 원인은 "http://localhost/"로 지정된 주소 때문이다. 명시적으로 "http://localhost/calc.aspx"처럼 aspx 파일 경로까지 넣어준다면 오류 없이 잘 실행된다. 그런데 여기서 한 가지 의문이 생길 수 있다. 예를 들어, 다음(Daum)같은 웹 사이트를 방문하게 되는 경우 보통 웹 브라우저에 "http://www.daum.net"이라고 입력할 뿐 그다음의 페이지 경로까지 지정하지는 않는다. 그 이유는 IIS에 설정된 "기본 문서(default document)"라는 옵션 때문이다. IIS 웹 서버는 페이지 경로가 지정되지 않은 경우 자동으로 루트 폴더에서 default.htm, default.asp, index.htm, index.html, iisstart.htm, default.aspx의 순서로 파일이 있는지 검색하고 그 중에 하나라도 있다면 먼저 검색된 순으로 웹 브라우저로 전송하는 기능이 있다. 따라서 일반적으로는 ASP.NET 웹 프로젝트에 "default.aspx" 웹 폼을 추가하고 사용자가 가장 먼저 봐야 할 내용을 그 안에 채워 넣는 것이 관례다. 여기서는 테스트를 위해 default.aspx 웹 폼을 새롭게 추가하고 HTML 내용을 이번 절에서 실습한 웹 페이지로 이동할 수 있게 다음과 같이 작성한다.

파일명: default.aspx

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="default.aspx.cs" Inherits="WebApplication1._default" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
    <a href="Calc.aspx">Calc.aspx</a><br />
    <a href="FileSend.aspx">FileSend.aspx</a><br />
    <a href="FormGet.aspx">FormGet.aspx</a><br />
    <a href="sqrt.aspx">sqrt.aspx</a><br />
    <a href="sqrtform.aspx">sqrtform.aspx</a><br />
```

```
    <a href="UserLogin.aspx">UserLogin.aspx</a><br />
    </div>
    </form>
</body>
</html>
```

default.aspx가 마련됐으면 다시 비주얼 스튜디오에서 F5 키를 눌러 실행해 보자. 웹 브라우저에서는 "http://localhost"만 주소 표시줄에 입력했지만 이번에는 아무런 오류 없이 default.aspx의 내용이 출력되는 것을 확인할 수 있다.

개발자 컴퓨터에서 정상적으로 개발을 마쳤으면 이제 실제로 서비스할 수 있는 웹 서버에 배포할 차례다. 일단, 여러분의 웹 프로젝트를 릴리즈(Release) 모드로 빌드한다.

참고!	디버그/릴리즈 빌드의 차이점은 5.2.4 '디버그 빌드와 릴리스 빌드' 절에서 설명한 바 있다.

다음으로 개발자 컴퓨터에 있는 웹 프로젝트 폴더를 서비스가 운영될 웹 서버의 IIS 웹 사이트가 지정한 루트 폴더에 복사한다. 여기서 유의해야 할 점은 웹 프로젝트에 있는 모든 파일이 복사될 필요는 없다는 것이다. 예를 들어, aspx.cs 파일들은 모두 빌드되어 /bin 폴더에 DLL로 통합됐기 때문에 보안상 웹 서버 측에 복사하지 않을 것을 권장한다. 파일 복사를 할 때 이런 파일을 일일이 빼고 복사하는 것은 귀찮은 작업이기 때문에 보통 robocopy.exe와 같은 프로그램을 이용해 복사하는 배치(batch) 파일을 만들어 둔다. 예를 들어, 웹 서버의 IP가 192.168.0.85이고, 웹 사이트의 루트 폴더가 "wwwroot"라는 이름으로 공유된 경우 다음과 같은 배치 파일을 만들어 명령행에서 실행한다.

```
robocopy D:₩temp₩WebApplication1₩WebApplication1 ₩₩192.168.0.85₩wwwroot /S /XF *.cs *.csproj *.csproj.user
```

위의 명령을 실행하면 robocopy 프로그램은 첫 번째 인자(D:₩temp₩WebApplication1₩WebApplication1)로 전달한 경로의 내용을 두 번째 인자(₩₩192.168.0.85₩wwwroot)로 지정된 폴더로 복사한다. 이때 /S 옵션이 지정돼 있으므로 첫 번째 인자에 포함된 모든 하위 폴더도 함께 복사하지만, /XF 옵션 다음에 지정된 파일들은 복사에서 제외한다. 즉, 확장자가 .cs, .csproj, .csproj.user인 파일은 복사에서 제외되어 웹 애플리케이션 운영에 꼭 필요한 파일만 복사된다.

그런데 이 책을 보는 대부분의 독자가 웹 서비스를 운영할 수 있는 전용 서버를 사용하는 경우는 거의 없을 것이다. 그런 분들을 위해 집에서 사용하는 컴퓨터를 이용해 여러분이 만든 실습 사이트를 서비스하는 방법을 설명하겠다.

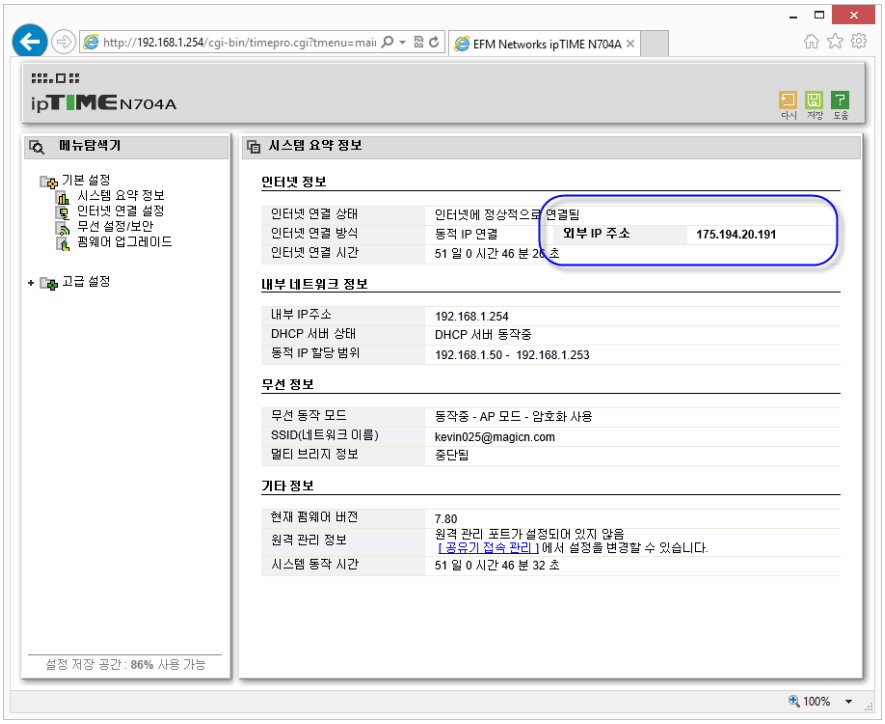
맨 먼저 할 일은 집에서 쓰는 컴퓨터에 IIS 웹 서버를 설치하고 여러분이 개발한 웹 사이트를

배포하는 것이다. 그런 다음 명령행에서 ipconfig을 실행하거나 '예제 6.32: 현재 컴퓨터에 할당된 IP 주소 출력' 코드를 실행하면 컴퓨터에 설정된 IP 주소를 구할 수 있다. 예를 들어, 집 컴퓨터의 IP 주소가 192.168.1.2라고 가정했을 때 웹 브라우저에서 "http://192.168.1.2"를 입력해서 정상적으로 웹 응용 프로그램이 동작하는지 확인한다.

이제부터 가장 중요한 설정 작업이 시작된다. 컴퓨터에서 사용 중인 IP 주소가 공용 IP인지, 사설 IP인지 확인해야 한다. 6.7.4 'Sysmte.Net.Dns' 절에서 이를 구분하는 방법을 설명했으니 다시 한번 확인한다. 컴퓨터가 공용 IP로 돼 있다면, 지금 당장 친구에게 전화를 걸어 웹 브라우저를 실행한 후 http://[공용IP]/를 입력해 보라고 한다. 그럼 여러분이 작성한 default.aspx의 내용이 친구의 웹 브라우저에 나타날 것이다. 하지만 대부분의 경우 192.x.x.x 영역에 해당하는 사설 IP가 할당돼 있을 것이므로 아직 친구에게 말해서는 안 된다. 그런대로 아쉽기는 하지만 집 안에 컴퓨터가 두 대 이상 있다면 다른 컴퓨터에서 "http://[사설IP]"로 웹 브라우저에 입력하면 default.aspx의 내용이 보일 것이다.

사설 IP가 사용된 주요 원인은 그림 6.25에서 설명한 것처럼 액세스 포인트 같은 공유기 장비가 연결돼 있기 때문이다. 이러한 경우 KT, LG 또는 SK텔레콤 등의 인터넷 서비스 제공자(ISP) 측에서 부여한 공용 IP를 공유기 장비에서 가져가 버리고, 여러분의 컴퓨터처럼 공유기에 연결된 모든 기기에는 사설 IP가 할당된다. 그래도 방법은 있으니 걱정하지 말자. 일반적인 공유기에는 관리자 홈페이지가 제공되고 그 화면을 통해 액세스 포인트에 할당된 공용 IP를 구할 수 있다. 예를 들어, 국내에 많이 보급된 ipTIME 네트워크 공유기는 그림 20.73처럼 http://192.168.1.254라는 주소를 통해 관리자 설정 홈페이지에 접근할 수 있고 첫 번째 화면에서 공용 IP 주소를 확인할 수 있다. (그림에서는 "외부 IP 주소"로 표현돼 있고 그 값은 175.194.20.191로 설정돼 있다.)

그림 20.73 공유기의 관리자 홈페이지를 이용한 공용 IP 주소 확인



참고!	다른 회사의 공유기를 사용 중이라면 공유기를 구매했을 때 보관해 둔 사용설명서를 펼쳐보면 각 공유기마다 제공되는 고유 관리자 페이지로 들어가는 방법이 설명돼 있을 것이다. 설령 사용설명서를 잊어버렸더라도 실망할 필요는 없다. 공유기를 만든 업체의 홈페이지에 들어가면 설명서가 제공될 것이므로 그곳에서 내려받으면 된다.

아직 친구에게 이 주소를 알려줘서는 안 된다. 왜냐하면 http://175.194.20.191로 접속하더라도 해당 IP 주소는 공유기에 할당돼 있기 때문에 집 컴퓨터까지 TCP 연결이 되지 않으므로 서비스할 수 없기 때문이다. 그럼 어떻게 해야 할까? 역시 이 문제를 공유기 측에 내장된 기능으로 해결할 수 있다. 공유기는 외부에서 들어오는 요청을 공유기에 연결된 내부 컴퓨터로 전달하는 NAT(Network Address Translation) 기능을 제공한다. ipTIME의 경우 그림 20.74처럼 좌측의 트리에서 "고급 설정" / "NAT/라우터 관리" / "포트포워드 설정" 메뉴에서 공용 IP에 대해 특정 포트(port)로 들어오는 요청을 내부 IP로 전달하도록 설정할 수 있다.

그림 20.74 내부 컴퓨터로 포트 포워드 설정

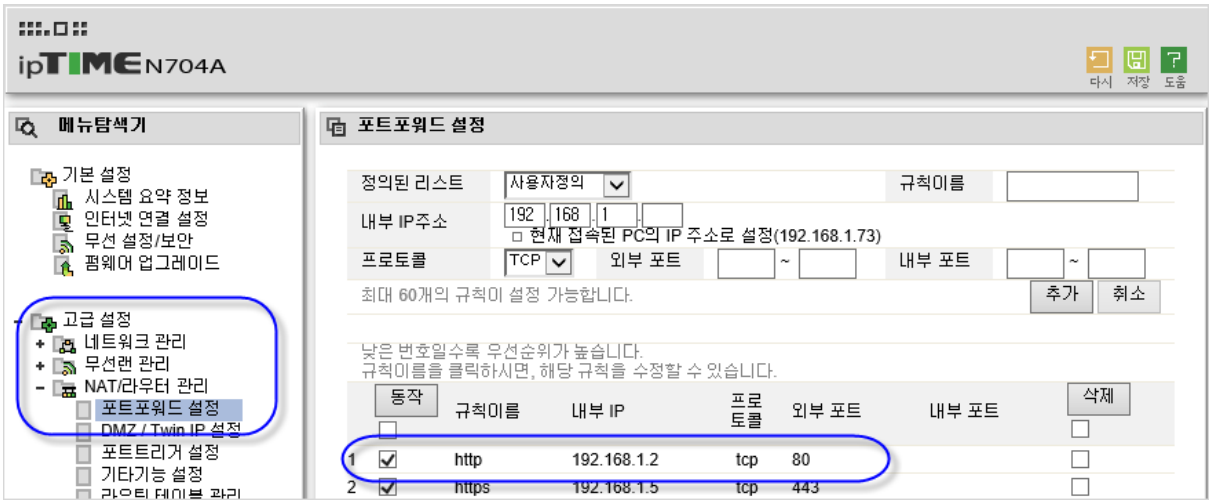


그림 20.74 화면에서는 공용 IP로 80포트를 통해 들어오는 요청을 공유기에 연결된 컴퓨터 중에서 192.168.1.2 내부 IP 주소를 가진 컴퓨터로 전달하도록 설정한 예를 보여준다. 물론 이 설정은 여러분의 상황에 맞게 해야 한다. 여기까지 설정했으면 이제 당당하게 친구에게 IP 주소를 알려주고 여러분이 만든 홈페이지를 자랑해 보자.

아쉬운 점이 있다면 여러분의 웹 응용 프로그램을 서비스하는 주소가 IP로 돼 있다는 것이다. 게다가 인터넷 서비스 제공자로부터 부여받은 공용 IP는 언제 회수되어 또 다른 공용 IP가 다시 부여될지 알 수 없는 "유동 IP"에 속한다. 공유기를 밤에 꺼두고 그다음 날 다시 켜면 거의 대부분은 새로운 공용 IP가 할당돼 있을 것이므로 여러분의 집 컴퓨터를 이용해 홈페이지를 개설하는 것은 아직은 무리다. 이러한 유동 IP 환경에서도 홈페이지를 변함없이 운영하고 싶다면 반드시 도메인 이름을 구매해야 한다. 국내에서는 가비아(https://www.gabia.com/) 등의 업체를 통해 도메인 명을 등록할 수 있으며, 종류에 따라 2년에 38,000원 ~ 82,000원 정도의 비용이 든다(이후로 갱신할 때마다 비용이 든다). 필자가 운영하는 도메인(sysnet.pe.kr)도 이런 식으로 구매해서 2년마다 연장하며 비용을 지출하고 있다.

도메인을 구매했으면 이제 공용 IP와 도메인을 연결해야 한다. 연결에 대한 설정은 도메인을 구매한 업체 측의 홈페이지에서 제공하므로 그 화면에 들어가 구매한 도메인과 여러분이 가진 공용 IP를 함께 설정하면 되지만, 여전히 같은 문제가 되풀이된다. 앞에서도 언급했듯이 인터넷 서비스 제공자로부터 받은 가정용 IP는 변경될 수 있으므로 도메인과 연결해봤자 지속적으로 서비스를 할 수는 없다. 바로 이때 필요한 것이 "Dynamic DNS" 서비스다. 국내에서 무료로 DDNS 서비스를 제공하는 곳은 ㈜디앤에스에버(https://kr.dnsever.com/) 같은 곳이 있다. 이곳에 간단하게 회원가입을 하고 아래 페이지에 설명된 대로 설정하고 나면 여러분이 구매한 DNS 명으로 언제든지 집에 있는 컴퓨터로 연결하는 것이 가능하다.

다이나믹 DNS 사용법

; http://kr.dnsever.com/index.html?selected_menu=aboutddns

다이나믹 DNS의 원리는 간단하다. 도메인 명과 연결될 IP 주소를 서비스하는 네임 서버를 dnsever 업체 측에서 관리하게 만든 다음, 집 컴퓨터에는 dnsever에서 제공하는 프로그램을 설치하면 된다. 그 프로그램은 컴퓨터의 유동 IP가 바뀔 때마다 dnsever의 네임서버에 통보하고 도메인 명을 최신 IP 주소로 매핑해서 서비스해준다.

정리

이 절에서 배운 내용이 웹 응용 프로그래밍의 전부는 아니다. 쿠키(Cookie), HTML5, 각종 자바스크립트 라이브러리, 웹 폼과 다른 구조로 웹 사이트를 만들 수 있는 ASP.NET MVC 방식 등을 비롯해 배워야 할 것은 무궁무진하게 많다.

그렇지만 그것들을 모두 배우고 나서 뭔가를 만들어 보려다가는 자칫 지루한 시간을 보내다가 지칠 수도 있다. 혹시 생각해본 웹 서비스가 있다면 지금 당장 시행착오를 겪어가면서 만들어 가는 것도 분명 의미가 있다. 가령, 블로그 정보를 수집해 양질의 정보를 제공하는 웹 사이트를 만든다고 가정해 보자. 전 세계의 블로그 주소를 수집해 그 내용을 가져오는 웹 로봇(Robot)을 '20.3 서비스 응용 프로그램' 절에서 배운 서비스 형식의 프로그램으로 만들 수 있다. 그 프로그램은 HttpWebRequest 타입을 이용해 끊임없이 블로그 주소(RSS)를 수집하고 그 안의 글을 내려받아 지정된 하드디스크 영역에 쌓게 만든다. 그다음 "웹 사이트"를 하나 만들고 로봇 프로그램이 쌓아 둔 데이터를 화면에 범주별로 보여주는 기능을 만들 수 있다. 핵심 기능이 완료되면 서서히 서비스 고도화 작업을 해나가면 된다. 예를 들어, 6.8 '데이터베이스' 절의 코드를 참고해 회원 정보를 데이터베이스와 연동할 수 있게 확장하는 것도 가능하다. 집에서 간단하게 만든 서비스가 인기를 끌어 서비스를 확장해야 할 시기가 되면 클라우드 서비스를 제공하는 업체 중에서 하나를 골라 서비스를 그곳으로 이전할 수도 있다. 물론 이쯤 되면 서비스를 통해 수익을 고민해야 하는 단계까지 온 것이다. 하지만 여러분이 만든 서비스가 성공하지 못했다고 해서 낙담할 필요는 없다. 그렇게 서비스를 운영하는 전체적인 실습 과정 자체가 여러분의 개발자 인생에 크나큰 힘을 실어줄 것이기 때문이다.

엄밀히 말해서 웹 응용 프로그래밍은 HTML 태그만을 주고받는 매우 간단한 유형의 개발 영역에 속한다. 하지만 쉽다고 해서 웹 응용 프로그래밍의 중요성을 평가절하해서는 안 된다. 인터넷의 발달과 함께 웹은 거대한 생태계를 구성하게 됐고, 그 규모는 지금도 폭발적으로 성장하고 있다. 기술 기반이 웹 개발과는 전혀 다른 임베디드(Embedded) 영역에서도 이제는 웹과 연동하려는 노력이 이뤄지고 있다. 말 그대로, 웹과 연결되지 않은 기술은 생존 자체가 위협받는 시기가 돼버린 것이다. 앞으로 어떤 영역의 개발자가 될지는 알 수 없으나, 한 가지 분명한 것은 웹을 항상 여러분의 곁에 두고 응용할 수 있는 여력은 남겨두는 것이 좋다는 것이다.

20.5 윈도우 폰 응용 프로그램

아이폰의 성공으로 모바일 폰은 또 하나의 개발 플랫폼으로 주목받고 있다. 윈도우 운영체제를 만든 마이크로소프트에서도 윈도우 운영체제의 모바일 버전을 개발했고, 이를 기반으로 노키아(Nokia)와 같은 제조업체에서 윈도우 폰을 출시하고 있다.

안드로이드/아이폰과 비교했을 때 윈도우 폰의 시장 점유율은 상대적으로 낮지만 윈도우 폰 응용 프로그램은 나름대로 장점을 가지고 있다.

- 쉬운 개발 환경
  비주얼 스튜디오와 잘 통합된 윈도우 폰 개발 환경은 모바일 응용 프로그램의 제작을 다른 모바일 폰에 비해 상대적으로 쉽게 만들어 준다.
- XAML + C#
  WPF 응용 프로그램을 만들면서 배웠던 XAML 기술로 윈도우 폰 응용 프로그램을 만들 수 있기 때문에 기존의 WPF 개발자에게는 진입 장벽이 낮다. 또한 C# 언어로 개발할 수 있으므로 기존의 닷넷 프레임워크에서 배운 기술의 연장선에서 응용 프로그램을 제작할 수 있다.

윈도우 폰 프로그래밍이 실제로 얼마나 쉬운지 한번 체험해보자. 우선 아래의 경로에서 현재 사용 중인 운영체제에 해당하는 윈도우 폰 SDK를 내려받아 설치한다.

- 윈도우 비스타/7 사용자:

Windows Phone SDK 7.1
; http://www.microsoft.com/en-us/download/details.aspx?id=27570

Windows Phone SDK 7.1.1 Update
; http://www.microsoft.com/en-us/download/details.aspx?id=29233

- 윈도우 8 사용자

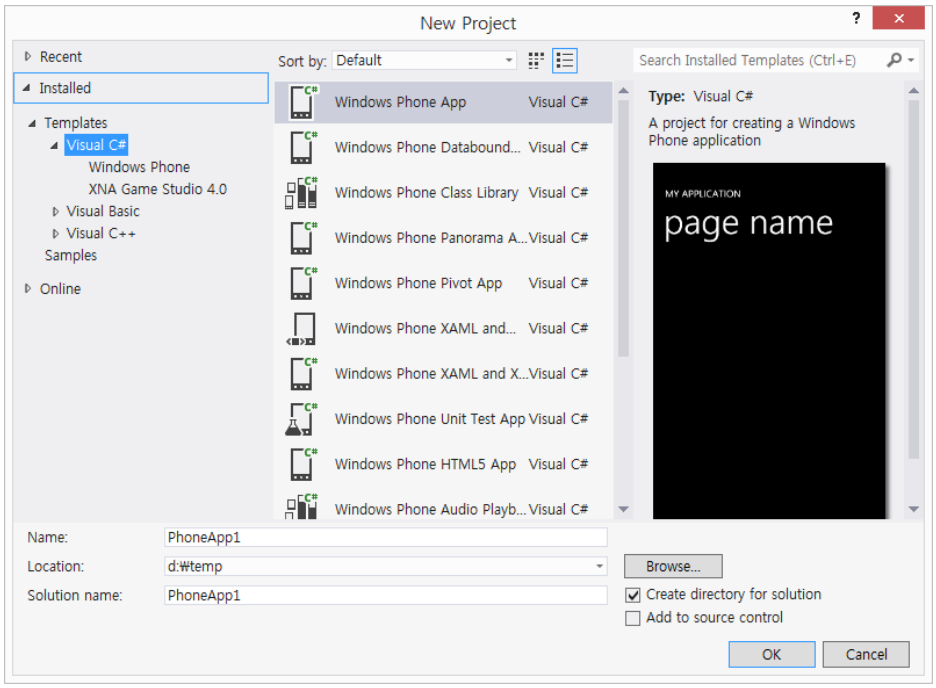
Windows Phone SDK 8.0
; http://www.microsoft.com/en-us/download/details.aspx?id=35471

참고!	현재 사용 중인 비주얼 스튜디오가 Express 버전이라면 "Windows Phone SDK 8.0"은 새롭게 "Visual Studio Express for Windows Phone" 개발 도구를 설치한다. 그 외에 Visual Studio Professional, Premium, Ultimate 버전이 설치돼 있는 경우에는 기존 비주얼 스튜디오에 플러그인 방식으로 확장되기 때문에 이전과 다름없이 비주얼 스튜디오를 이용해 개발할 수 있다.

설치 과정은 이전에 2.3 '비주얼 스튜디오 개발 환경' 절에서 설명한 방법과 동일하다. 설치를 마치면 새롭게 설치된 "Visual Studio Express for Windows Phone"을 실행하고, "파일(FILE)" / "새 프로

젝트(New Project)" 메뉴를 선택하면 그림 20.75와 같이 다양한 유형의 윈도우 폰을 위한 프로젝트 템플릿을 볼 수 있다.

그림 20.75 윈도우 폰용 프로젝트 템플릿



여기서는 가장 간단한 유형으로 실습할 것이므로 "Windows Phone App"을 선택하고 프로젝트 이름을 적절하게 입력한 후 확인(OK) 버튼을 누른다. 그러면 (8.0 버전의 SDK를 설치한 경우) 대상이 되는 윈도우 폰 OS 버전을 물어볼 수 있는데, 어차피 에뮬레이터에서만 구동할 것이므로 아무거나 선택해도 된다(여기서는 8.0을 선택하고 진행한다).

참고!	물론 윈도우 폰을 사용 중이라면 에뮬레이터에서 테스트한 후 윈도우 폰으로 직접 배포하는 것도 가능하다.

프로젝트가 생성되고 나면 그림 20.76과 같은 프로젝트 폴더 구조가 나타난다.

그림 20.76 솔루션 탐색기 - 윈도우 폰 프로젝트

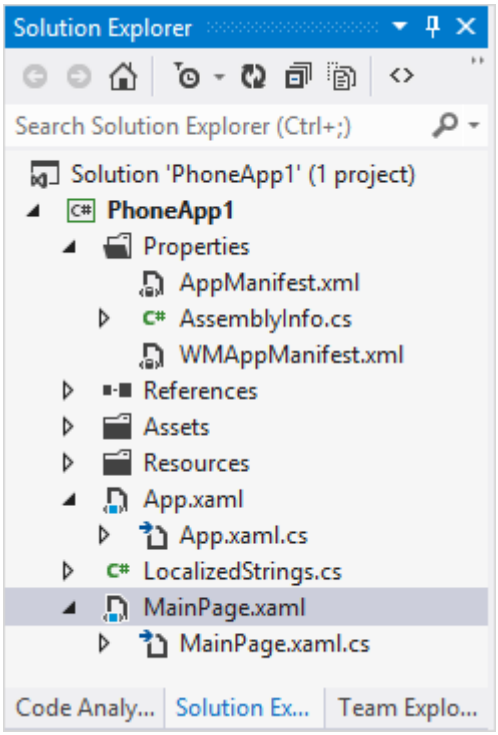
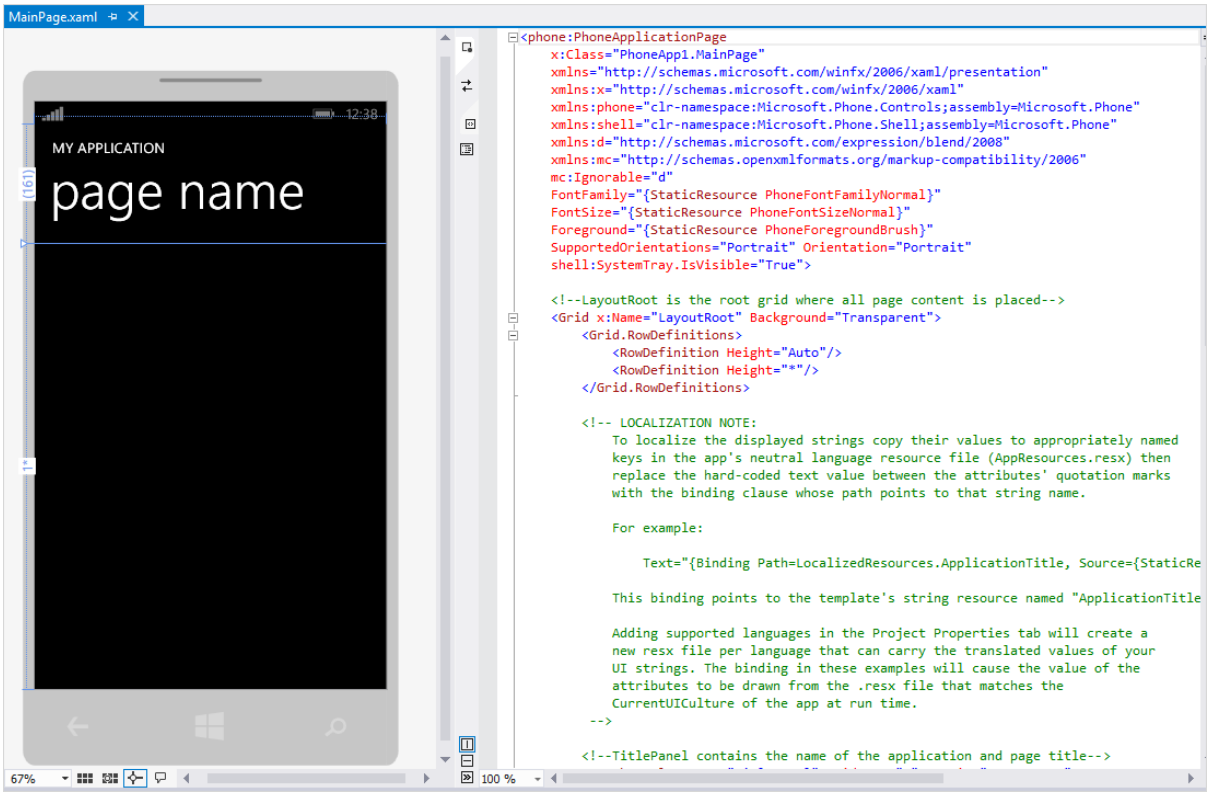


표 20.12 윈도우 폰 프로젝트의 기본 항목 설명

폴더	파일	설명
Properties	AppManifest.xml	비주얼 스튜디오에서 자동으로 관리하므로 거의 변경할 필요가 없다.
	WMAppManifest.xml	윈도우 폰 응용 프로그램의 전반적인 설정을 관리한다. 예를 들어, 아이콘을 변경하거나 지원하는 해상도를 설정할 수 있다.
Assets		프로그램에 사용되는 자원을 넣어둔다. 기본적으로는 프로그램의 아이콘 파일을 포함한다.
(프로젝트)	App.xaml	WPF의 App.xaml과 유사
	LocalizedStrings.cs	XAML에서 사용되는 문자열들의 다국어 지원을 위해 기본 포함된 코드 파일, 실제로 변경되는 파일은 /Resources/AppResources.resx이므로 이 파일 자체는 거의 변경할 필요가 없다.
	MainPage.xaml	WPF의 MainWindow.xaml과 유사

WPF 프로그램을 만드는 것과 비슷하기 때문에 MainPage.xaml 파일을 비주얼 스튜디오에서 여는 것으로 시작하면 된다. 그럼 비주얼 스튜디오는 그림 20.77과 같이 영역을 둘로 나눠 왼쪽에서는 XAML의 디자인을 보여주고, 오른쪽에서는 XAML 소스코드를 편집할 수 있다.

그림 20.77 MainPage.xaml 편집 및 디자인 화면



WPF 프로젝트에서 편집했던 것과 동일하게 이번에도 어느 쪽 영역을 편집하든 변경사항을 저장하면 다른 영역으로 반영된다.

MainPage.xaml의 내용은 주석을 제외하면 예제 20.23에서 볼 수 있는 것과 같이 간단하게 구성돼 있다.

예제 20.23 윈도우 폰 앱 기본 예제 – MainPage.xaml

```
// ================== MainPage.xaml ==============================
<phone:PhoneApplicationPage
    x:Class="PhoneApp1.MainPage"
    ......[생략]......
    FontFamily="{StaticResource PhoneFontFamilyNormal}"
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    SupportedOrientations="Portrait" Orientation="Portrait"
    shell:SystemTray.IsVisible="True">

    <Grid x:Name="LayoutRoot" Background="Transparent">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
```

```
            <TextBlock      Text="MY      APPLICATION"      Style="{StaticResource
PhoneTextNormalStyle}" Margin="12,0"/>
            <TextBlock  Text="page  name"  Margin="9,-7,0,0"  Style="{StaticResource
PhoneTextTitle1Style}"/>
        </StackPanel>

        <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
        </Grid>
    </Grid>

</phone:PhoneApplicationPage>
```

윈도우 폰에서의 XAML은 WPF에서 사용된 XAML과 구조는 같다. 예제 20.23의 경우에도 WPF에서 배운 각종 컨트롤을 유사하게 가져다 배치할 수 있다. WPF와 어느 정도 비슷한지 알 수 있게 예제 20.11의 시계 기능을 그대로 윈도우 폰 앱으로 구현해 보자.

우선 DataBinding을 위해 예제 20.24와 같이 XAML의 루트 객체에 DataContext를 설정하고, 시간을 출력할 TextBlock 컨트롤을 Grid 컨트롤 아래에 추가한다.

예제 20.24 윈도우 폰 앱 만들기 - 시계

```
// ================== MainPage.xaml ============================
<phone:PhoneApplicationPage
    x:Class="PhoneApp1.MainPage"
    ……[생략]……
    DataContext="{Binding RelativeSource={RelativeSource Self}}"
>

    <Grid x:Name="LayoutRoot" Background="Transparent">
        ……[생략]……

        <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
            <TextBlock  Text="Timer"  Style="{StaticResource  PhoneTextNormalStyle}"
Margin="12,0"/>
            <TextBlock Text="Current Time" Margin="9,-7,0,0" Style="{StaticResource
PhoneTextTitle1Style}"/>
        </StackPanel>

        <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
            <TextBlock Name="lblTime" Text="{Binding Path=Time}" />
        </Grid>
    </Grid>

</phone:PhoneApplicationPage>
```

기존의 예제 20.11과 비교해서 예제 20.24에서 달라진 것이 있다면 단지 시간 출력을 위한 Label 컨트롤을 TextBlock으로 바꾼 것밖에 없다. 아쉽게도 윈도우 폰의 제한된 자원으로 인해 WPF와 동등한 수준의 컨트롤이 모두 제공되지는 않기 때문에 이런 부분은 고려해야 한다.

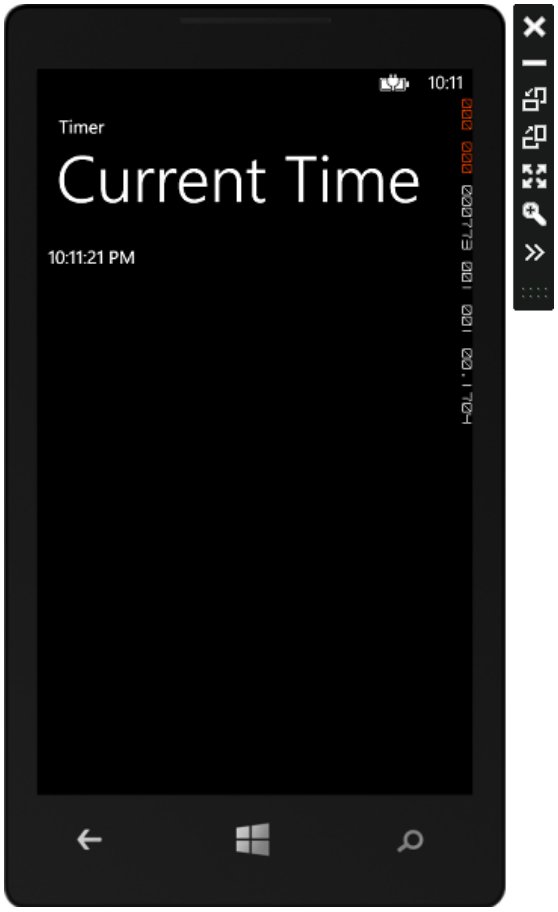
시계 기능을 구현하기 위한 나머지 작업으로 MainPage.xaml.cs에 예제 20.11과 동일한 변경 사

항을 적용하면 된다.

```
using System;
using System.ComponentModel;
using System.Windows.Threading;
using Microsoft.Phone.Controls;

namespace PhoneApp1
{
    public partial class MainPage : PhoneApplicationPage, INotifyPropertyChanged
    {
        DispatcherTimer _timer;

        string _time;
        public string Time
        {
            get { return _time; }
            set
            {
                _time = value;
                OnPropertyChanged("Time");
            }
        }

        public virtual void OnPropertyChanged(string propertyName)
        {
            if (PropertyChanged == null)
            {
                return;
            }

            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }

        public MainPage()
        {
            InitializeComponent();

            _timer = new DispatcherTimer();
            _timer.Tick += _timer_Tick;
            _timer.Interval = new TimeSpan(0, 0, 1); // 1초 마다 발생
            _timer.Start();
        }

        void _timer_Tick(object sender, EventArgs e)
        {
            this.Time = DateTime.Now.ToLongTimeString();
        }

        public event PropertyChangedEventHandler PropertyChanged;
    }
}
```

이제 F5 키를 눌러 완성된 윈도우 폰 시계 앱을 실행해 보자. 잠시 후 그림 20.78처럼 윈도우 폰 에뮬레이터가 실행되고 이어서 우리가 작성한 시계 앱이 구동된다.

그림 20.78 에뮬레이터에 실행된 윈도우 폰 시계 앱



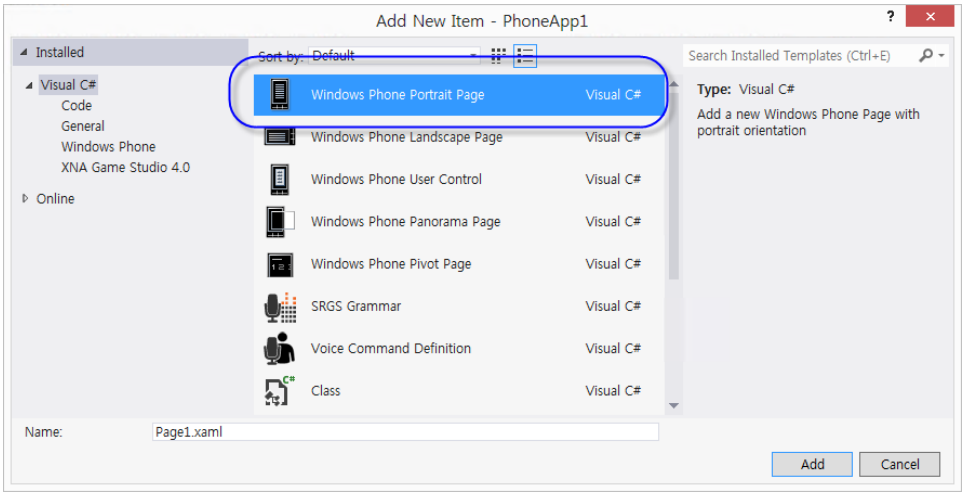
이처럼 WPF의 개발 경험을 윈도우 폰을 위한 앱에서도 그대로 활용할 수 있다. 단지 XAML의 구현 규모로 봤을 때 WPF가 전체 기능(Full-set)이면 윈도우 폰의 XAML은 WPF의 부분집합으로 보면 된다.

19.5.1 페이지 단위의 응용 프로그램 구현

WPF에서는 Window 타입을 상속받은 MainWindow가 진입점이지만, 윈도우 폰 프로젝트에서는 PhoneApplicationPage를 상속받은 MainPage 타입이 진입점이다. 폰의 특성상 화면의 크기와 충전식 배터리 문제로 인해 윈도우 폰에서는 "Window" 자원이 배제되고, 한 화면을 꽉 채우는 페이지(Page)의 개념이 나온 것이다.

윈도우 폰 앱 하나는 1개 이상의 페이지로 구성되는데, 각 페이지는 MainPage와 마찬가지로 화면 전체를 사용한다. 실습을 위해 새롭게 예제 프로젝트를 생성하고, 그림 20.79와 같이 새 항목 추가를 선택하고 "Windows Phone Portrait Page"를 "Second.xaml"이라는 이름으로 생성한다.

그림 20.79 새로운 페이지 추가



앱이 실행된 후 최초로 보일 페이지는 /Properties/WMAppManifest.xml의 "Navigation Page"에 의해 결정되는데, 기본값은 MainPage.xaml로 설정돼 있다. 따라서 현재 상태에서 앱을 실행하면 Second.xaml이 아닌 MainPage.xaml의 내용이 화면에 먼저 보인다. 페이지 간의 이동을 하고 싶다면 NavigationService 타입의 Navigate 정적 메서드를 이용한다. 예제 20.25에서는 MainPage.xaml과 Second.xaml의 화면에 버튼을 추가하고 그 버튼의 이벤트 처리기에서 서로의 페이지로 전환하는 방법을 보여준다.

예제 20.25 페이지 전환

```
// ================== MainPage.xaml ==============================
<phone:PhoneApplicationPage
    x:Class="PhoneApp1.MainPage"
    ......[생략]......
    DataContext="{Binding RelativeSource={RelativeSource Self}}"
>

    <Grid x:Name="LayoutRoot" Background="Transparent">
        ......[생략]......

        <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
            <TextBlock Text="Timer" Style="{StaticResource PhoneTextNormalStyle}"
Margin="12,0"/>
            <TextBlock Text="Current Time" Margin="9,-7,0,0" Style="{StaticResource
PhoneTextTitle1Style}"/>
        </StackPanel>

        <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
            <Button Content="Second.xaml로 이동" Click="Button_Click" />
        </Grid>
    </Grid>

</phone:PhoneApplicationPage>
```

```
// ================== MainPage.xaml.cs ============================
using System;
using System.Windows;
using System.Windows.Navigation;
using Microsoft.Phone.Controls;

namespace PhoneApp1
{
    public partial class MainPage : PhoneApplicationPage
    {
        public MainPage()
        {
            InitializeComponent();
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
            NavigationService.Navigate(new Uri("/Second.xaml", UriKind.Relative));
        }
    }
}
```

```
// ================== Second.xaml ============================
<phone:PhoneApplicationPage
    x:Class="PhoneApp1.Second"
    ……[생략]……
    SupportedOrientations="Portrait" Orientation="Portrait"
    shell:SystemTray.IsVisible="True">

    <Grid x:Name="LayoutRoot" Background="Transparent">
        ……[생략]……

        <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
            <Button Content="MainPage.xaml로 이동" Click="Button_Click" />
        </Grid>
    </Grid>

</phone:PhoneApplicationPage>
```

```
// ================== Second.xaml.cs ============================
using System;
using System.Windows;
using System.Windows.Navigation;
using Microsoft.Phone.Controls;

namespace PhoneApp1
{
    public partial class Second : PhoneApplicationPage
    {
        public Second()
        {
            InitializeComponent();
        }
```

```
        private void Button_Click(object sender, RoutedEventArgs e)
        {
          NavigationService.Navigate(new Uri("/MainPage.xaml", UriKind.Relative));
        }
    }
}
```

프로그램을 실행해서 확인해 보면 버튼을 누를 때마다 페이지가 전체 화면에 걸쳐 바뀌면서 교체되는 모습을 확인할 수 있다.

페이지가 모여서 하나의 모바일 폰 앱을 구성한다는 것은 어찌 보면 HTML 페이지로 구성된 웹 애플리케이션과 유사하다. 실제로 Navigate 메서드에 전달되는 문자열도 일반적인 URL과 유사한 형식을 띠고 있어 페이지 간에 정보를 전달하기 위한 목적으로 쿼리 문자열을 사용하는 것도 가능하다. 예제 20.26에서는 예제 20.25의 MainPage.xaml에 TextBox를 하나 더 추가하고 사용자가 입력한 문자열을 쿼리 문자열을 통해 Second.xaml에 전달한다.

예제 20.26 QueryString을 이용한 페이지 간의 정보 전달

```
// ================== MainPage.xaml ============================
<phone:PhoneApplicationPage
    x:Class="PhoneApp1.Second"
    ……[생략]……
    SupportedOrientations="Portrait" Orientation="Portrait"
    shell:SystemTray.IsVisible="True">

    <Grid x:Name="LayoutRoot" Background="Transparent">
        ……[생략]……

        <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
            <Grid.RowDefinitions>
                <RowDefinition Height="80" />
                <RowDefinition Height="*" />
            </Grid.RowDefinitions>
            <TextBox x:Name="txtInput" Grid.Row="0" />
            <Button Content="Second.xaml" Click="Button_Click" Grid.Row="1" />
        </Grid>
    </Grid>

</phone:PhoneApplicationPage>
```

```
// ================== MainPage.xaml.cs ============================
using System;
using System.Windows;
using System.Windows.Navigation;
using Microsoft.Phone.Controls;

namespace PhoneApp1
{
    public partial class MainPage : PhoneApplicationPage
```

```
    {
        public MainPage()
        {
            InitializeComponent();
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
            NavigationService.Navigate(new Uri("/Second.xaml?p1=" + txtInput.Text,
                                UriKind.Relative));
        }
    }
}
```

```
// ================== Second.xaml ============================
<phone:PhoneApplicationPage
    x:Class="PhoneApp1.Second"
    ……[생략]……
    SupportedOrientations="Portrait" Orientation="Portrait"
    shell:SystemTray.IsVisible="True">

    <Grid x:Name="LayoutRoot" Background="Transparent">
        ……[생략]……

        <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
            <Grid.RowDefinitions>
                <RowDefinition Height="80" />
                <RowDefinition Height="*" />
            </Grid.RowDefinitions>
            <TextBlock x:Name="lblText" Grid.Row="0" />
            <Button Content="MainPage.xaml로 이동"
                        Click="Button_Click" Grid.Row="1" />
        </Grid>
    </Grid>

</phone:PhoneApplicationPage>
```

```
// ================== Second.xaml.cs ============================
using System;
using System.Windows;
using System.Windows.Navigation;
using Microsoft.Phone.Controls;

namespace PhoneApp1
{
    public partial class Second : PhoneApplicationPage
    {
        public Second()
        {
            InitializeComponent();
        }

        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
```

```
            string inputValue;

            NavigationContext.QueryString.TryGetValue("p1", out inputValue);

            lblText.Text = inputValue;

            base.OnNavigatedTo(e);
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
          NavigationService.Navigate(new Uri("/MainPage.xaml", UriKind.Relative));
        }
    }
}
```

MainPage.xaml에 추가된 입력 상자의 값을 MainPage.xaml.cs의 버튼 클릭 이벤트 처리기에서 마치 웹 사이트를 방문하는 것처럼 쿼리 문자열을 전달하고 있다. Second.xaml.cs에서 쿼리 문자열로 전달된 값을 구하는 방법도 재미있다. 윈도우 폰 앱은 페이지가 로드될 때 OnLoad에 해당하는 OnNavigatedTo라는 특별한 메서드가 호출된다. 따라서 페이지 단위로 초기화해야 하는 코드가 있다면 OnNavigatedTo 메서드에서 할 수 있는데, 쿼리 문자열을 받아오는 것도 그에 해당하는 작업 중 하나가 될 수 있다. NavigationContext 타입의 QueryString 정적 속성을 이용해 전달된 값을 구하는데, 그 방법은 웹 브라우저의 QueryString 규칙과 동일하다. 부가적으로 OnNavigatedTo와 반대되는 OnNavigatedFrom 메서드도 있다는 점을 알아두자. 이 메서드는 페이지를 떠나는 순간에 발생하며, 일반적으로는 사용자가 해당 페이지 내에서 입력한 값을 저장하는 코드를 넣어둔다.

쿼리 문자열을 이용한 방법은 윈도우 폰 앱에서 제공되는 페이지 간의 정보 전달 방법 중 하나다. 사실 쿼리 문자열로 정보를 전달하는 것은 간단한 유형의 문자열에 한해서만 유용할 뿐 복잡한 데이터를 전달하기에는 적합하지 않다. 윈도우 폰 앱은 웹 사이트와는 달리 독자적인 자신만의 고유한 프로세스(EXE) 공간을 가지기 때문에 변수를 통해 정보를 전달할 수도 있다. 예를 들어, App.xaml.cs에 정의된 App 타입에 정적 변수를 두고 이를 공유해서 전달할 수 있다. 다음 예제를 보자.

예제 20.27 App 타입의 공유 변수를 이용한 페이지 간의 정보 전달

```
// ================== App.xaml.cs ============================
namespace PhoneApp1
{
    public partial class App : Application
    {
        // 데이터 공유를 위한 정적 변수 추가
        static Dictionary<string, object> _dataStore =
                                        new Dictionary<string, object>();
        public static Dictionary<string, object> DataStore
        {
```

```
            get { return _dataStore; }
        }
        // ……[생략]……
    }
}
```

```
// ================== MainPage.xaml.cs ============================
using System;
using System.Windows;
using System.Windows.Navigation;
using Microsoft.Phone.Controls;

namespace PhoneApp1
{
    public partial class MainPage : PhoneApplicationPage
    {
        public MainPage()
        {
            InitializeComponent();
        }

        // 이번 예제에서는 값 설정을 OnNavigatedFrom 메서드에서 설정
        // 이 메서드는 페이지를 벗어나기 바로 전에 호출된다.
        protected override void OnNavigatedFrom(NavigationEventArgs e)
        {
            App.DataStore["p1"] = txtInput.Text;

            base.OnNavigatedFrom(e);
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
            // 여기서 값을 설정하는 것도 가능
            // App.DataStore["p1"] = txtInput.Text;
            NavigationService.Navigate(new Uri("/Second.xaml", UriKind.Relative));
        }
    }
}
```

```
// ================== Second.xaml.cs ============================
using System;
using System.Windows;
using System.Windows.Navigation;
using Microsoft.Phone.Controls;

namespace PhoneApp1
{
    public partial class Second : PhoneApplicationPage
    {
        public Second()
        {
            InitializeComponent();
        }
```

```
        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
            lblText.Text = App.DataStore["p1"] as string;
            base.OnNavigatedTo(e);
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
          NavigationService.Navigate(new Uri("/MainPage.xaml", UriKind.Relative));
        }
    }
}
```

예제 20.27의 기능은 예제 20.26과 완전히 동일하다. 하지만 같은 기능을 QueryString이 아닌 App 타입의 정적 변수를 통해 구현하는 차이만 있을 뿐이다.

QueryString과 프로세스 메모리의 정적 변수를 이용한 방법 말고도 디스크를 이용하는 방법도 있다. 일반적인 윈도우 운영체제에서는 하드디스크가 있다면 드라이브 문자(예: C 드라이브)를 통해 전체 파일 시스템을 접근할 수 있게 허용하지만, 윈도우 폰의 경우 해당 앱만이 접근할 수 있는 제한적인 파일 시스템을 제공한다. 이름하여 "격리된 저장소(isolated storage)"라고 하며, 각 앱마다 저장소가 할당되어 서로 다른 앱끼리는 저장소 공유가 불가능하다.

격리된 저장소를 이용하는 방법 중 하나는 System.IO.IsolatedStorage 네임스페이스에서 제공되는 IsolatedStorageFileStream 타입을 이용하는 방법이다. 이를 이용해 예제 20.27을 바꾸면 예제 20.28과 같다.

예제 20.28 격리된 저장소를 이용한 페이지 간의 정보 전달

```
// ================== MainPage.xaml.cs ============================
using System;
using System.Windows;
using System.Windows.Navigation;
using Microsoft.Phone.Controls;
using System.IO.IsolatedStorage;
using System.Text;

namespace PhoneApp1
{
    public partial class MainPage : PhoneApplicationPage
    {
        public MainPage()
        {
            InitializeComponent();
        }

        protected override void OnNavigatedFrom(NavigationEventArgs e)
        {
            using (IsolatedStorageFileStream fs =
                new IsolatedStorageFileStream("p1.data", System.IO.FileMode.Create,
                IsolatedStorageFile.GetUserStoreForApplication()))
```

```
            {
                byte [] buffer = Encoding.UTF8.GetBytes(txtInput.Text);
                fs.Write(buffer, 0, buffer.Length);
                fs.Close();
            }

            base.OnNavigatedFrom(e);
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
            NavigationService.Navigate(new Uri("/Second.xaml?p1=" + txtInput.Text,
UriKind.Relative));
        }
    }
}
```

```
// ================== Second.xaml.cs ============================
using System;
using System.Windows;
using System.Windows.Navigation;
using Microsoft.Phone.Controls;

namespace PhoneApp1
{
    public partial class Second : PhoneApplicationPage
    {
        public Second()
        {
            InitializeComponent();
        }

        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
            using (IsolatedStorageFile fs =
                        IsolatedStorageFile.GetUserStoreForApplication())
            {
                bool exists = fs.FileExists("p1.data");
                if (exists == true)
                {
                    var file = fs.OpenFile("p1.data", System.IO.FileMode.Open);
                    byte [] buffer = new byte[file.Length];
                    file.Read(buffer, 0, buffer.Length);

                    lblText.Text =
                            Encoding.UTF8.GetString(buffer, 0, buffer.Length);
                }
            }

            base.OnNavigatedTo(e);
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
          NavigationService.Navigate(new Uri("/MainPage.xaml", UriKind.Relative));
        }
    }
```

```
}
```

이전에 다뤘던 6.5 '파일' 절의 System.IO.FileStream 사용법과 비교해서 생성자 부분을 제외하고는 격리된 저장소에 파일을 생성하고 읽는 작업은 별반 다르지 않다. 격리된 저장소를 이용하면 해당 데이터가 디스크에 영구적으로 저장되기 때문에 응용 프로그램이 종료된 이후에도 그 값을 재사용할 수 있다는 특징이 있다. 대신 디스크를 사용하기 때문에 많은 데이터를 다루는 경우 속도가 느려질 수 있다는 단점도 있다. 그렇게 되면 앱은 잠시 동안 반응하지 않게 되어 사용자 경험(UX: User Experience)이 떨어진다. 이러한 경우에는 부가적으로 비동기 메서드를 이용해 데이터를 저장/복구하는 것도 고려해 볼 수 있다.

19.5.2 제한적인 멀티 태스킹

윈도우 폰 환경의 특징 가운데 빠뜨릴 수 없는 것이 바로 "작은 배터리 용량"이다. 한 손에 들어가는 폰의 구조상 배터리 크기는 작아질 수밖에 없고, 결국 노트북과는 비교도 안 되는 저용량 배터리를 장착하게 된다. 게다가 항상 켜져 있는 상태라서 배터리 소모가 더 빠르다. 이로써 폰을 위한 운영체제는 최대한 배터리를 절약하는 방안을 강구해야 했고 결정적으로 PC용 운영체제의 특징인 "다중 작업(Multi-Tasking)"을 배제하게 된다. 즉, PC에서는 게임을 실행하다가도 웹 브라우저를 띄우면 게임이 멈추지 않은 상태로 웹 브라우저가 별도의 윈도우 영역을 갖고 실행됐지만 윈도우 폰에서는 전체 화면으로 게임을 실행하다가 웹 브라우저를 실행하면 이전에 실행했던 게임에 더는 CPU 자원을 할당하지 않고 오직 현재 순간에 전체 화면을 점유하고 있는 앱에만 CPU 자원을 할당한다.

예를 들어, 사용자가 윈도우 폰의 앱을 다음과 같은 순서로 실행한다고 가정해 보자.

1) "웹 브라우저 앱" 실행
2) "홈 버튼" 클릭
3) "홈 화면"
4) "음악 앱" 실행
5) "뒤로 가기 버튼" 클릭
6) "홈 화면"

이전에도 언급했듯이 윈도우 폰의 앱은 사용자에 의한 명시적인 "종료" 절차가 없다. 1번에서 "웹 브라우저 앱"을 실행한 사용자는 "홈 버튼"을 클릭해 "홈 화면"으로 이동한다. 이때 "웹 브라우저 앱"은 종료된 것일까? 윈도우 폰은 이런 경우 앱을 종료하지 않는다. 왜냐하면 사용자가 "뒤로 가기" 버튼을 이용해 곧바로 돌아올 수 있기 때문이다. 그렇다고 해서 실행 중인 것도 아니다. 왜냐하면 웹 브라우저 앱에 CPU가 더는 할당되지 않기 때문이다. 이런 앱의 상태를 "비활성화(deactivation)"됐다고 한다.

하지만 4, 5, 6번 단계에서 보다시피 앱을 실행하고 "뒤로 가기 버튼"을 이용해 홈 화면으로 이동한 경우는 다르다. 이런 경우는 다시 그 앱으로 들어가려면 "음악 앱"을 명시적으로 실행하는 동작이 수반되기 때문에 윈도우 폰은 "뒤로 가기 버튼"을 이용해 앱을 빠져나가는 경우는 프로세스를 종료해버린다.

그런데 여기서 한 가지 과정이 더 추가된다. "비활성화" 상태에서 여러 다른 앱을 실행해 메모리가 부족해지면 어떻게 될까? 윈도우 폰 운영체제는 이런 상황이 되면 어쩔 수 없이 비활성화 상태의 앱을 강제로 종료한다. 이렇게 된 앱을 "Tombstone(묘비가 세워진 상태)"됐다고 한다. 그렇다면 tombstone 상태와 "뒤로 가기 버튼"을 이용해 종료된 상태가 어떻게 다른지 의문이 남는다. 이 차이점이란 앱을 실행한 과정의 "이력"을 토대로 결정된다. 즉, 이력에 없으면 "종료" 상태이고, 이력에 있으면서 종료된 상태이면 tombstone 상태인 것이다. 위의 앱 실행 과정에서 1번의 웹 브라우저 앱은 아직 이력에 남아 있으므로 6번 과정에서 다시 한번 뒤로 가기를 실행하면 이전의 웹 브라우저 앱이 복원된다. 하지만 4번에서 실행한 음악 앱은 뒤로 가기를 통해 완전히 종료된 상태이므로 이력 목록에 남아 있지 않아 다시 음악 앱 아이콘을 눌러 실행하지 않는 한 해당 앱으로 진행할 수 있는 경로가 없다.

그런데 1번 과정에서 실행한 웹 브라우저 앱은 "홈 버튼"을 누름으로써 처음에는 비활성화 상태로 되지만, 정확히 어느 시점에 tombstone 상태로 빠질지는 알 수 없다. 그것은 운영체제에 의해 자원이 부족하다고 판단되는 임의의 시점이 될 수밖에 없다. 앱 개발자 입장에서 "비활성화" 상태의 앱이 다시 활성화되는 것은 아무런 문제가 없다. 하지만 tombstone 상태로 빠진 앱이 다시 활성화되는 것은 사정이 다르다. 왜냐하면 tombstone의 앱은 사실상 종료된 것이기 때문에 메모리에 보존했던 데이터가 모두 제거된 상태이므로 사용자가 "뒤로 가기 버튼"을 통해 tombstone 상태의 앱으로 진입하면 앱이 완전히 새롭게 실행되는 것과 같다.

tombstone을 제대로 이해해야만 이력에 남아 있는 앱이 다시 실행됐을 때 비정상적으로 종료되는 문제를 막을 수 있다. 이 문제가 실제로 어떻게 나타날 수 있는지 예제 20.27을 통해 직접 체험해 볼 수 있다. 예제 20.27을 다음과 같이 실행하는 경우를 예로 들어 보자.

1) 예제 20.27의 앱을 실행(MainPage.xaml)
2) Second.xaml로 페이지 전환
3) "홈 버튼"을 클릭해서 "홈 화면"으로 전환

이 상태에서 곧바로 "뒤로 가기 버튼"을 누르면 윈도우 폰은 Second.xaml이 실행됐던 상태의 앱 화면을 그대로 복구한다. 왜냐하면 비활성화 상태에 있었기 때문에 앱 프로세스가 살아 있었고, 따라서 메모리에 있던 변수가 그대로 보존돼 있었으므로 앱 전환이 정상적으로 이뤄진다.

하지만 tombstone 상태를 재현하기 위해 부하가 높은 게임 등을 몇 개 실행한 다음 "뒤로 가기 버튼"을 눌러 Second.xaml로 돌아오면 어떤 일이 발생할까? 그럼에도 메모리가 넉넉해서 비활

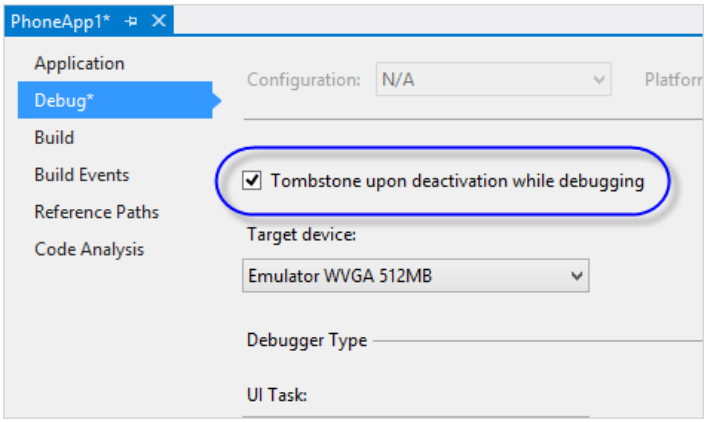
성화 상태에 머무르고 있었다면 아무런 오류도 발생하지 않겠지만 의도한 대로 tombstone 상태로 전환됐다면 Second.xaml.cs 파일의 OnNavigatedTo 코드에서 다음과 같은 오류가 발생한다.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    lblText.Text = App.DataStore["p1"] as string;   // ➔ 예외 발생
    base.OnNavigatedTo(e);
}

    ⇨ 발생한 예외
An exception of type 'System.Collections.Generic.KeyNotFoundException' occurred in mscorlib.ni.dll but was not handled in user code
```

이유는 간단하다. 프로세스가 종료되어 tombstone 상태이므로 이전에 앱이 가졌던 메모리 상의 모든 데이터가 제거된 것이나 다름없다. 그런데 사용자가 이력 목록에만 존재하는 앱을 "뒤로 가기 버튼"을 통해 진입했기 때문에 윈도우 폰 운영체제는 해당 앱을 새롭게 실행한 후 마지막으로 보였던 Second.xaml 페이지를 화면에 보여주게 된다. 결과적으로 OnNavigatedTo 메서드에서 실행되는 App.DataStore에는 더는 "p1" 키 값을 가진 데이터가 존재하지 않기 때문에 예외가 발생하는 것이다.

tombstone 상태를 재현하기가 쉽지 않기 때문에 비주얼 스튜디오에서는 디버깅을 쉽게 할 수 있도록 앱이 비활성화 상태로 빠지면 강제로 tombstone 상태로 전환하는 옵션을 가지고 있다. 윈도우 폰 프로젝트 속성 창을 띄우고 그림 20.80처럼 "디버그(Debug)" 탭의 "Tombstone upon deactivation while debugging" 옵션을 켜면 된다.

그림 20.80 tombstone 옵션



이렇게 설정하고, F5 키를 이용해 디버깅 상태로 진입하면 앱이 실행되고, Second.xaml 페이지로 이동한 다음 홈 버튼을 눌러 비활성화 상태로 만들어 보자. 옵션 설정 덕분에 해당 앱은 곧바

로 tombstone 상태로 진입하게 되고, 이 상태에서 "뒤로 가기" 버튼을 눌러 앱으로 진입하면 Second.xaml.cs의 OnNavigatedTo 메서드에서 예외가 발생하는 것을 확인할 수 있다. 경험상 안정화된 윈도우 폰 앱을 개발하려면 반드시 이 옵션을 켜고 추가된 페이지 하나당 모두 tombstone 테스트를 해야만 한다.

만약 tombstone 상태로 빠진 앱의 상태를 그대로 다시 진입하고 싶다면 어떻게 해야 할까? 답은 간단하다. 메모리를 사용해서는 안 되므로 페이지 상태를 복원하기 위해 필요한 데이터가 있다면 정적 변수에 보관하지 말아야 한다. 대신 격리된 저장소 영역과 같이 프로세스 종료 이후에도 보관/복원할 수 있는 방법을 택하면 된다.

19.5.3 Launcher / Chooser

페이지 간에 데이터를 연동하는 방법과 함께 윈도우 폰에서는 "앱 간에 데이터를 연동"하는 방법도 특별하다. 윈도우 폰 앱은 전체 화면을 차지하는 앱이 또 다른 앱에 의해 비활성화되면 CPU 할당을 전혀 받을 수 없기 때문에 상호 데이터를 연동하는 방법이 모호해질 수밖에 없는 것이다.

이런 문제를 해결하기 위해 윈도우 폰 앱은 특정 용도의 앱과 연동할 수 있는 Launcher와 Chooser라고 하는 라이브러리를 제공한다. 간단한 예를 들어, 여러분이 만든 앱에서 모바일 폰의 대표적인 "단문 메시지 서비스(SMS: Short Message Service)"를 이용해야 한다고 가정해 보자. 그러자면 단문 메시지를 발송하는 앱이 실행돼야 하는데, 윈도우 폰 앱에서는 이를 위해 SmsComposeTask 타입의 Launcher를 제공한다. 예제 20.29에서는 간단한 사용법을 보여준다.

예제 20.29 모바일 폰에서 단문 메시지를 전송하는 방법

```
// ================== MainPage.xaml.cs ============================
using System;
using System.Windows;
using System.Windows.Navigation;
using Microsoft.Phone.Controls;

namespace PhoneApp1
{
    public partial class MainPage : PhoneApplicationPage
    {
        public MainPage()
        {
            InitializeComponent();
        }

        private void btnSendSMS_Click(object sender, RoutedEventArgs e)
        {
            SmsComposeTask sms = new SmsComposeTask();
            sms.Body = "Hello World";
            sms.Show();
        }
    }
}
```

이 예제를 실행해 SmsComposeTask 타입의 Show 메서드가 호출되면 그림 20.81처럼 "Hello World" 문자열이 미리 입력돼 있는 SMS 앱이 실행되는 것을 볼 수 있다.

그림 20.81 SMS 앱이 Launcher를 통해 활성화된 모습

한 가지 주의할 점은 이 상태는 SMS 앱이 새롭게 실행된 것과 전혀 차이가 없다는 점이다. 따라서 SmsComposeTask를 실행했던 앱은 비활성화 상태에 빠지고 SMS를 발송하지 않고 "홈 버튼"을 눌러 연속적인 앱을 실행하다 보면 Tombstone 상태로 진입할 가능성도 있다는 점에 유의해야 한다. Launcher를 통해 원하는 외부 기능을 실행했으면 다시 원래의 앱으로 돌아오기 위해서는 동일하게 "뒤로 가기 버튼"을 이용하면 된다.

Launcher의 예를 알아봤으니 이제 Chooser를 다뤄 보자. Chooser의 한 예로 CameraCaptureTask 타입을 들 수 있는데, 이것을 이용하면 윈도우 폰에서 제공되는 카메라 앱을 실행할 수 있다.

예제 20.30 Camera를 통해 이미지를 받아 화면에 출력

```
// ================== MainPage.xaml ============================
<phone:PhoneApplicationPage
    x:Class="PhoneApp1.MainPage"
    …… [생략] ……
    SupportedOrientations="Portrait" Orientation="Portrait"
    shell:SystemTray.IsVisible="True">
```

```
    <Grid x:Name="LayoutRoot" Background="Transparent">
        …… [생략] ……
        <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
            <Grid.RowDefinitions>
                <RowDefinition></RowDefinition>
                <RowDefinition></RowDefinition>
            </Grid.RowDefinitions>
            <Button Content="Capture Photo" Click="btnCapturePhoto_Click" />
            <Image x:Name="imgPhoto" Grid.Row="1" />
        </Grid>
    </Grid>

</phone:PhoneApplicationPage>
```

```
// ================== MainPage.xaml.cs ============================
using System;
using System.Windows;
using System.Windows.Navigation;
using Microsoft.Phone.Controls;

namespace PhoneApp1
{
    public partial class MainPage : PhoneApplicationPage
    {
        public MainPage()
        {
            InitializeComponent();
        }

        private void btnCapturePhoto_Click(object sender, RoutedEventArgs e)
        {
            CameraCaptureTask camera = new CameraCaptureTask();
            camera.Completed += camera_Completed;
            camera.Show();
        }

        void camera_Completed(object sender, PhotoResult e)
        {
            if (e.TaskResult == TaskResult.OK)
            {
                BitmapImage photoBitmap = new BitmapImage();
                photoBitmap.SetSource(e.ChosenPhoto);

                imgPhoto.Source = photoBitmap;
            }
        }

    }
}
```

CameraCaptureTask 타입은 Completed 이벤트를 제공함으로써 Show 메서드가 실행된 이후 카메라 앱으로 넘어갔을 때 찍힌 사진 데이터를 넘겨받는 작업을 할 수 있다. 따라서 Completed 이벤트 처리기에서는 PhotoResult 타입의 TaskResult 속성이 OK인 경우, 사용자가 카메라로 촬영한

사진 데이터를 ChosenPhoto 속성으로 넘겨받아 처리할 수 있다. 예제 20.30에서는 카메라 앱이 반환한 이미지 데이터를 XAML의 <Image /> 컨트롤을 통해 출력한다.

Launcher와 Chooser는 값의 반환 여부로 구분할 수 있다. 예제를 통해 알 수 있듯이 Launcher에 해당하는 타입은 다른 앱을 호출하는 것으로 끝나지만, Chooser 타입으로 활성화된 앱들은 동작을 완료한 후 호출자 측에 값을 반환한다는 뚜렷한 차이가 있다. 따라서 Chooser의 경우 Launcher와 비교해서 Completed 이벤트 처리기를 설정하는 작업이 하나 더 추가된다. 다음 표에서는 윈도우 폰에서 제공되는 Launcher와 Chooser의 일부가 정리돼 있다.

표 20.13 Launcher와 Chooser

종류	타입	설명
Launcher	ConnectionSettingsTask	네트워크 연결을 변경할 수 있게 설정 앱을 실행한다.
	EmailComposeTask	전자메일 앱을 실행한다.
	MediaPlayerLauncher	미디어 플레이어 앱을 실행한다.
	PhoneCallTask	전화를 걸 수 있는 앱을 실행한다.
	SearchTask	검색 창이 나오는 앱을 실행한다.
	WebBrowserTask	웹 브라우저 앱을 실행한다.
Chooser	AddressChooserTask	연락처 앱을 실행하고 선택된 주소를 반환한다.
	EmailAddressChooserTask	연락처 앱을 실행하고 선택된 전자메일 주소를 반환한다.
	PhoneNumberChooserTask	연락처 앱을 실행하고 선택된 전화번호를 반환한다.
	PhotoChooserTask	앨범 앱을 실행하고 선택된 이미지를 반환한다.
	SaveEmailAddressTask	연락처 앱을 실행하고 새로운 전자메일 주소 및 그 외 항목을 등록한다. 반환값을 통해 등록 여부만을 알 수 있다.
	SavePhoneNumberTask	연락처 앱을 실행하고 새로운 전화번호 및 그 외 항목을 등록한다. 반환값을 통해 등록 여부만을 알 수 있다.

정리

국내 시장에서 차지하는 윈도우 폰의 영향력은 사실상 미미한 수준에 불과하기 때문에 윈도우 폰용 앱을 제작하는 것은 아직까지는 그다지 매력적이지 않다. 그나마 앱을 제작하고 싶다면 반드시 세계 시장을 바라보고 시작하는 것이 좋다.

다만 한 가지 긍정적인 요소가 있다면 윈도우 폰 앱의 기반 기술이 XAML이라는 점이다. 데스크톱 응용 프로그램인 WPF와 윈도우 8의 스토어 앱(Store App)도 XAML을 바탕으로 하기 때문에

이 중에서 어느 하나라도 알고 있다면 다른 두 가지 프로그램을 만드는 것도 그리 어렵지 않다. 즉, 하나의 기술이 세 가지 서로 다른 응용 프로그램 분야(데스크톱, 모바일 폰, 태블릿)에 재사용 될 수 있는 것이다.

21 장: 실습 – 파워포인트 쇼 제어 프로그램

실습의 모델이 된 응용 프로그램은 윈도우 폰에서 마이크로소프트 오피스의 파워포인트 쇼를 제어할 수 있는 프로그램으로서 국내 개발사인 SmartShare에서 만들었다.

PRESENTER FOR WINDOWS PHONE ; http://www.ismartshare.net/portfolio/presenter/

이 프로그램을 선택한 이유는 단순히 학습을 위한 실습보다는 실제로 현실적인 수준의 응용 프로그램이 어떤 기술과 난이도로 만들어지는가에 대한 감을 잡는 데 도움될 것이라 생각해서였다. 윈도우 폰을 가지고 있는 독자가 많지 않을 것이므로 실습 주제로 적당하지 않을 수 있으나 우리가 만들 앱은 에뮬레이터에서도 충분히 실습할 수 있으므로 크게 걱정하지 않아도 된다. 게다가 실습을 마치고 나면 안드로이드나 아이폰용 앱을 개발해본 경험이 있는 독자라면 그리 어렵지 않게 해당 플랫폼을 지원하는 앱을 만들 수 있을 것이다.

이 장의 나머지 부분에서는 마치 여러분이 프로그램을 처음으로 만든다고 가정하고 설명을 진행한다. 이를 통해 프로그램의 개발 프로세스를 간접적으로 체험하는 데 도움이 되도록 구성했다.

21.1 앱 기획

맨 먼저 할 일은 어떤 앱을 만들어야 할지 결정하는 것이다. 간단한 질문 답변을 통해 이 과정을 완성할 수 있다.

1. **[질문]** 어떤 앱을 만들 것인가?
   [답변] PC를 통해 진행 중인 프레젠테이션 쇼를 모바일 폰으로 제어하는 앱을 만든다.

2. **[질문]** 어떤 프레젠테이션 응용 프로그램을 지원할 것인가?
   [답변] 마이크로소프트의 오피스 프로그램 중에 파워포인트를 지원한다. 또는 무료 프레젠테이션 프로그램 가운데 프레지(Prezi)가 있는데 이것도 지원하면 좋겠다.

이 이상으로 더 질문할 필요는 없다. 그 이유는 뭘까? 프레젠테이션 쇼를 제어하려면 대상이 되는 프레젠테이션 응용 프로그램을 제어할 수 있는 방법을 알아야 한다. 그렇게 하는 방법이 없다면 이 응용 프로그램은 애당초 기획될 의미가 없는 것이다.

따라서 어떤 앱이 만들어져야 할지 결정되면 다음 단계로는 정말 그것이 현실적으로 타당한지 신속히 기술 검토를 해야 한다.

21.2 기술 검토

응용 프로그램에 대한 기획이 나왔다고 해서 모든 앱이 만들어질 수 있는 것은 아니다. 기술 검토 단계를 거치지 않으면 나중에 해당 앱을 만들 수 없다는 판정이 나왔을 때 심각한 비용 손실을 입을 수밖에 없다. 따라서 만들고자 하는 앱이 기술적으로 개발 가능한지 여부를 앱 개발 초기에 반드시 확인해봐야 한다. 물론 응용 프로그램의 유형에 따라 난이도가 쉬운 경우라면 기술 검토 단계를 생략할 수 있다. 이 같은 결정은 프로그램을 만들 리더급 개발자가 참여한 가운데 진행해야 하며, 일단 기술 검토가 필요하다고 판단되면 팀 내 최고의 기술적인 실력을 갖춘 팀원을 뽑아서 빠르게 진행해야 한다.

프레젠테이션 쇼를 제어하는 앱은 반드시 기술 검토가 필요하다라고 판단됐으므로 이제 파워포인트와 프레지 프로그램을 대상으로 쇼를 제어하는 방법이 있는지 알아내야 한다. 우선 파워포인트에 대해 알아보니 외부에서 접근할 수 있게 확장 모델을 잘 구현하고 있음이 밝혀졌다.

PowerPoint Object Model ; http://msdn.microsoft.com/en-us/library/office/aa213568(v=office.11).aspx

게다가 웹 검색을 통해 예제도 찾을 수 있다.

How to use Automation to create and to show a PowerPoint 2002 presentation by using Visual C# .NET 2002 ; http://support.microsoft.com/kb/303718/en-us

자동화(Automation)라는 COM 기술을 통해 파워포인트를 제어하는 수단을 제공하고 있는데, 아직은 구체적으로 알 수 없지만 위의 두 문서를 보니 상당히 세부적인 부분까지 제어할 수 있을 것 같다.

반면 프레지는 Flash를 통해 웹 브라우저에서 실행되는 유형으로 보이는데, 딱히 외부에서 제어하는 방법을 찾을 수는 없었다. 기술 검토로 더 시간을 소비할 수는 없으므로 일단 프레지는 포기하고 오피스 파워포인트를 대상으로 앱을 만들기로 결정한다.

21.3 세부 요구 사항 정의

대략적인 기술 검토가 완료됐으면 이제 앞으로 만들 앱이 갖춰야 할 기능을 정의해야 한다. 앱이 도대체 어떤 기능을 제공해야 하는가에 대한 범위를 정하는 일은 대단히 중요하다. 최초로 공개될 앱이 너무 많은 기능을 제공한다면 기능을 구현하기 위한 개발 기간이 늘어날 것이고 자칫 경쟁사가 더 일찍 제품을 출시해서 타격을 받을 수 있다. 반대로 기능이 별로 없다면 사용자들은 해당 앱이 유용하지 않다는 선입견을 받을 수 있고 이후로 개선되더라도 첫인상으로 결정된 지배

적인 의견으로 더 이상의 시장 확대를 기대하지 못할 수도 있다. 따라서 버전 1에서 어떤 기능을 포함해야 할지는 신중하게 결정해야 한다.

쇼 제어 프로그램의 요구사항을 다음과 같이 간단하게 정리했다.

표 21.1 요구사항 정리

요구사항	우선순위
1. 폰을 이용해 PC에서 보여지는 프레젠테이션의 슬라이드를 함께 볼 수 있어야 한다.	**높음**
2. 터치를 이용해 왼쪽으로 밀면 다음 슬라이드로 쇼가 나타나고, 오른쪽으로 밀면 이전 슬라이드가 나타나야 한다.	**높음**
3. 슬라이드에 적어 놓은 메모를 함께 봤으면 좋겠다.	**낮음**

요구사항과 함께 우선순위를 결정하는 것도 좋다. 또한 필요하다면 개별 요구사항에 대한 기술 검토를 할 수도 있다.

21.4 응용 프로그램의 구조

요구사항이 정리됐으므로 이제 본격적으로 응용 프로그램을 구현할 차례다. 하지만 코드 작성에 앞서 응용 프로그램의 전체적인 구조(architecture)를 구상하는 것이 좋다. 이를 통해 응용 프로그램의 동작 구조를 한눈에 볼 수 있고 아직 개발되지 않은 상태에서도 프로그램이 어떤 식으로 동작하게 될지를 그려볼 수 있어 문제에 집중할 수 있다.

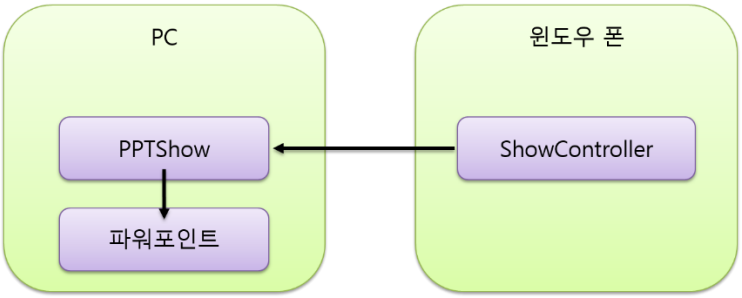
우선 쇼 제어를 하려면 두 개의 프로그램이 있어야 한다. 하나는 PC에서 실행 중인 파워포인트를 제어하는 윈도우 응용 프로그램이고, 또 다른 하나는 윈도우 폰에서 실행되어 사용자가 원격에서 쇼를 제어하는 역할을 한다. 이쯤에서 솔루션 이름과 함께 두 응용 프로그램의 이름을 정해보자.

표 21.2 솔루션 구조

솔루션	프로젝트	응용 프로그램 유형
OfficePresenter	ShowController	윈도우 폰 응용 프로그램
	PPTShow	WPF 응용 프로그램

파워포인트 프로그램을 같은 PC에서 실행 중인 PPTShow가 제어하고, PPTShow는 윈도우 폰으로부터 사용자 입력을 받는 ShowController의 제어를 받는다. 따라서 동작 구조는 다음과 같다.

그림 21.1 응용 프로그램 간 동작 구조



이제 다시 질문을 통해 몇 가지 사항을 결정해 보자.

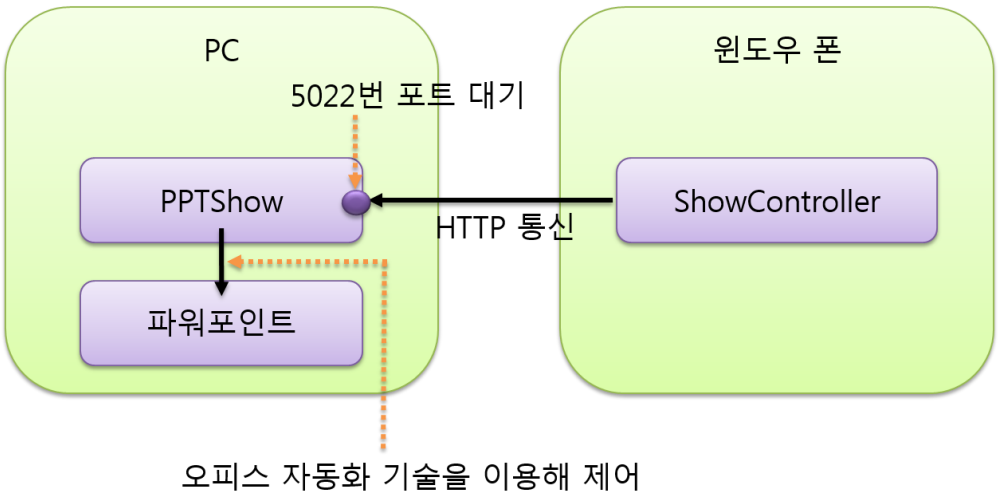
1. **[질문]** PC와 윈도우 폰 간의 통신은 어떻게 할 것인가?
   [답변] PC 측의 PPTShow에서 TCP 서버를 열고, 윈도우 폰 측의 ShowController에서 접속하는 방식이 좋겠다.

2. **[질문]** TCP 서버/클라이언트를 만들어 직접 소켓 통신을 하는 것이 정말 좋을까?
   [답변] 아무래도 소켓 수준에서 통신하는 것은 번거로우니 간편하게 HTTP 통신을 하는 것이 좋을 것 같다. 어차피 많은 데이터가 오가는 것은 아니므로 필요할 때마다 연결을 맺고 끊는 HTTP 통신이 적합할 듯하다. PPTShow 측에 HTTP 서버 기능을 넣고, ShowController 측에서는 WebClient 타입을 이용해 필요할 때마다 통신을 한다.

3. **[질문]** 대기 포트는 몇 번이 좋을까?
   [답변] 큰 의미는 없으니 그냥 5022번으로 하자.

여기까지 의견이 정해졌으면 구조도의 내용을 좀더 보강할 수 있다.

그림 21.2 세부 동작 구조



구조가 정해졌으면 모든 팀원들 간에 내용을 공유하는 것이 좋다. 전체적인 응용 프로그램의

구조를 개발자가 알고 작업하는 것과 모르고 작업하는 것에는 큰 차이가 있다. 모르고 작업한다는 것은 협업이 없다는 것이고 결국 프로젝트 후반에 가서 서로 간의 인터페이스가 맞지 않아 잦은 의견 불일치로 이어질 수 있다. 반면 전체적인 그림이 머릿속에 그려진 개발자들은 자신이 맡은 부분이 결국 어떤 식으로 상호 연동될지 판단할 수 있으므로 협업을 고려하면서 작업을 진행할 수 있다.

21.5 응용 프로그램의 동작 시나리오 및 화면 스케치

응용 프로그램의 동작 구조가 정해졌어도 여전히 코드를 작성할 수 있는 수준은 아니다. 왜냐하면 실제로 응용 프로그램이 어떻게 동작해야 하는지에 대해서는 정해지지 않았기 때문에 무턱대고 코드 먼저 작성하면 나중에 다시 작업해야 하는 부분들이 발생할 수 있다. 따라서 개략적인 화면 스케치와 함께 동작 시나리오를 작성해 봄으로써 응용 프로그램의 코드 수준에서 요구되는 사항들을 좀더 자세하게 이끌어 낼 수 있다. 앞에서 작성한 "세부 요구 사항 정의"를 토대로 실제로 응용 프로그램의 화면 동작까지 정의하는 과정을 거치면 된다.

이번에도 질문 답변을 통해 하나씩 정의해 보자.

1. **[질문]** PC 측의 PPTShow가 HTTP 서버로 동작하면 클라이언트 측에서 접속해야 할 텐데, 포트는 5022번으로 결정했지만 IP 주소는 사용자 상황에 따라 달라진다. 어떻게 이 주소를 쉽게 알 수 있을까?
   [답변] PPTShow가 실행되면 PC 측의 IP 정보를 프로그램에 출력하는 방법으로 해결하자.

2. **[질문]** PPTShow는 제어할 파워포인트 파일을 어떻게 선택할 것인가?
   [답변] PPTShow 측에 "Open" 버튼을 두고 제어할 PPT 파일을 직접 선택하게 하자.

3. **[질문]** 윈도우 폰 앱(ShowController)에서는 IP 주소를 어떻게 입력받는가?
   [답변] 그냥 간단하게 사용자가 텍스트 박스에 입력하게 하자.

4. **[질문]** 윈도우 폰 앱의 페이지 구성은 어떻게 할 것인가?
   [답변] 첫 화면에서는 PC측에 연결하기 위한 IP, 포트 정보와 "연결" 버튼을 둔다. PC 측에 정상적으로 연결되면 페이지가 바뀌고 쇼를 제어할 수 있는 화면 구성이 나타난다.

5. **[질문]** 쇼를 제어할 수 있는 화면 구성은 어떻게 할 것인가?
   [답변] 화면을 상하 절반으로 나누고 상단에는 현재 쇼에 나타난 슬라이드 화면이 나타나고, 하단에는 그 슬라이드에 포함된 메모가 보여지는 것이 좋겠다.

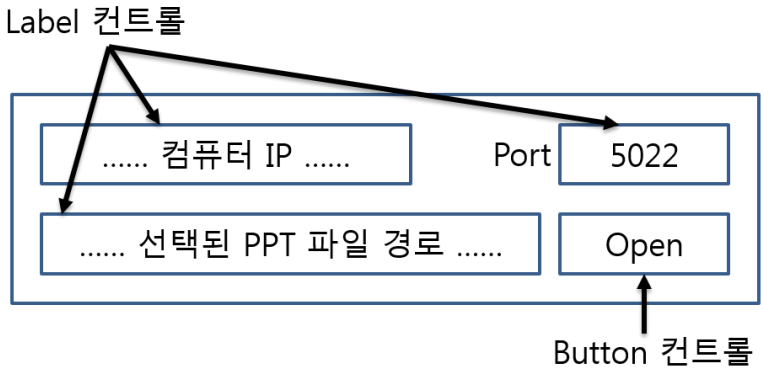
6. **[질문]** 쇼를 넘기는 터치 작업은 어떻게 처리할 것인가?
   [답변] 다행히 윈도우폰용 XAML에는 화면을 터치로 넘길 수 있게 Panorama 컨트롤이 기본적으로 제공되므로 이를 사용하면 되겠다.

회의 결과를 텍스트 형식으로 정의하는 것도 좋지만 이를 바탕으로 대략적인 화면을 그려볼 필요가 있다. 이를 통해 해당 앱을 기획한 사람의 의도와 잘 맞아떨어지는지 확인할 수 있고, 그래

야만 나중에 프로젝트가 전혀 다른 방향으로 흘러가는 것을 방지할 수 있다.

우선, PC 측에서 파워포인트 쇼를 제어하는 프로그램의 외양을 정의해 보자.

그림 21.3 PPTShow 응용 프로그램의 외양 스케치



이것은 회의 결과를 통해 개발자가 이해한 응용 프로그램의 외양이다. 이를 가지고 다시 앱 기획자와 이야기를 나눌 수 있다.

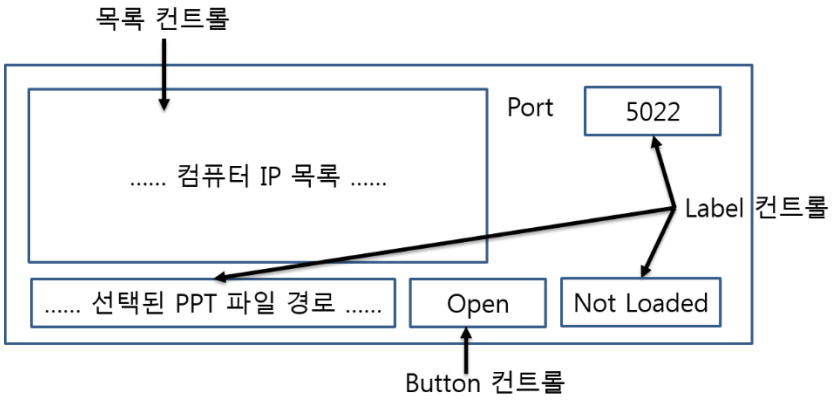
1. **[질문]** 컴퓨터에는 IP 주소가 한 개만 있는가?
   [답변] 노트북의 경우 유무선 어댑터가 함께 있으니 IP는 목록으로 보여줄 필요가 있다. 이 부분은 Label 컨트롤에서 리스트 컨트롤로 대체하겠다.

2. **[질문]** Open 버튼을 누르면 어떤 동작이 발생하는가?
   [답변] 파일 선택 창이 뜨고 쇼를 진행할 PPT 파일을 선택할 수 있다.

3. **[질문]** PPT 파일을 선택/로드했는지 알 수 있는 상태를 보여줄 수는 없을까?
   [답변] PPT 파일이 선택된 경우 "..... 선택된 PPT 파일 경로 ....." 라벨 컨트롤에 해당 파일의 경로가 출력되겠지만 명시적인 것을 원한다면 상태를 알리기 위한 라벨 컨트롤을 하나 더 추가하겠다.

회의를 마치고 그 결과를 기존 스케치에 반영하면 다음과 같다.

그림 21.4 PPTShow 2차 스케치



당연히 이 결과를 가지고 다시 기획팀과 협의해야 한다. 이번 실습에서는 여기서 협의가 완료된 것으로 가정하겠다.

다음으로 윈도우 폰에서 실행될 ShowController 앱을 스케치해보자. 회의에서 결정한 사항에 따라 이 프로그램은 2개의 페이지로 구현하기로 했다. 첫 페이지에는 PPTShow에 연결할 수 있는 정보와 "연결" 버튼을 두고, 두 번째 페이지는 실제로 쇼를 제어할 수 있는 페이지로 구성한다.

그림 21.5 ShowController 페이지 1 스케치

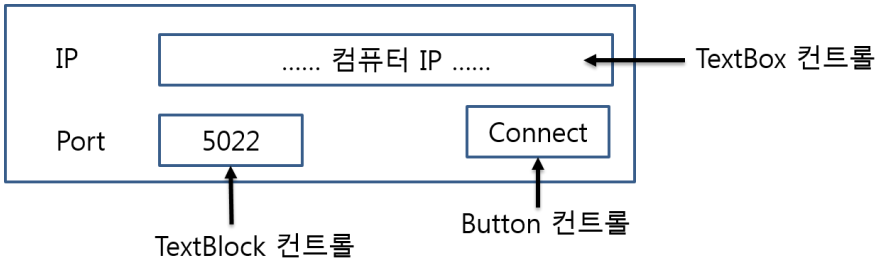
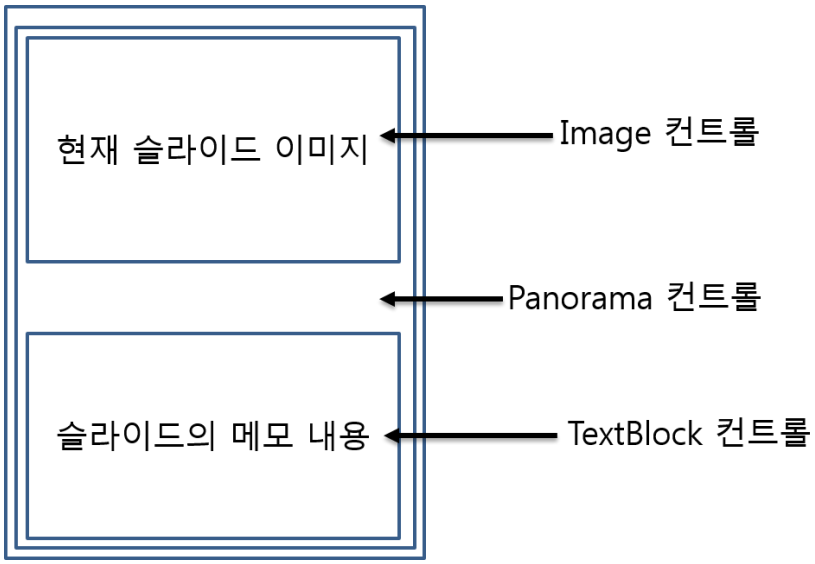


그림 21.6 ShowController 페이지 2 스케치



기획팀과의 회의에서 "페이지 1"은 통과했지만, "페이지 2"에 대해서는 의문이 제기됐다. 지난 번 회의에서 기획팀은 개발자로부터 "Panorama 컨트롤"이라고 하는 말을 듣긴 했지만 구체적으로 이것이 어떤 것인지 감이 오지 않아서 질문하게 된다(독자 여러분들도 마찬가지일 것이다).

Panorama 컨트롤은 윈도우 폰 응용 프로그램에서만 제공되며 터치 기반의 인터페이스를 제공한다. 그림 21.7에서 보면 Panorama 컨트롤은 윈도우 폰의 크기보다 넓은 콘텐츠를 담는다. 현재 보여지는 콘텐츠에서 왼쪽 내용을 보고 싶으면 터치를 이용해 화면을 오른쪽으로 밀어주면 된다. 마찬가지로 오른쪽 내용을 보고 싶으면 터치를 통해 화면을 왼쪽으로 밀어주는 식으로 보이는 영

역을 조절할 수 있다.

그림 21.7 Panorama 컨트롤

따라서 ShowController 앱에서는 로드된 PPT 파일의 전체 슬라이드를 Panorama 컨트롤에 일렬로 넣어뒀다가 터치를 이용해 화면을 좌/우로 밀면 쇼로 보여지는 슬라이드도 마찬가지로 이전/이후 슬라이드로 이동할 수 있게 제어할 수 있다. 설명을 다 듣고 난 기획팀에서 다음과 같은 질문이 이어진다.

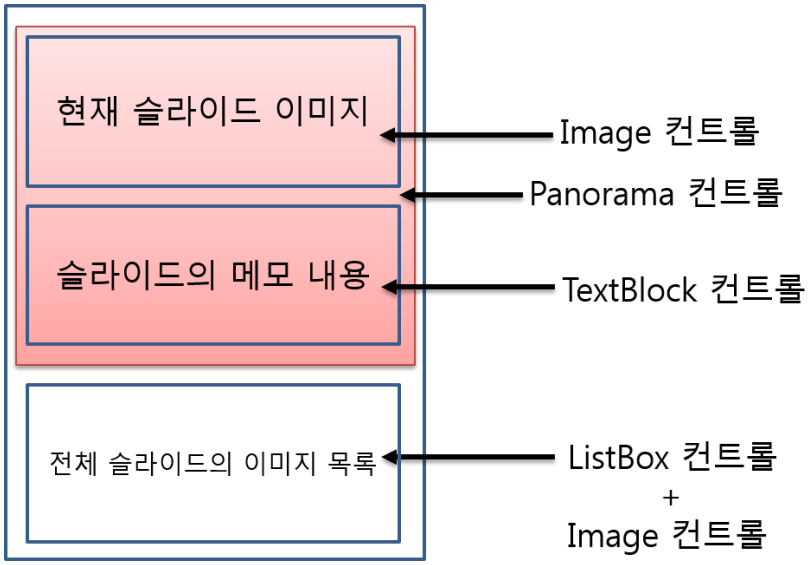
1. **[질문]** Panorama 컨트롤을 쓰는 경우, 터치 동작을 할 때마다 슬라이드를 앞/뒤로 하나씩만 이동할 수 있는 것 같다. 맞는가?
   [답변] 맞다.

2. **[질문]** 내 경험에 비춰볼 때 PPT 쇼를 하다 보면 한참 전에 있는 슬라이드로 되돌아갈 필요가 있었다. Panorama 컨트롤만 있는 경우 이 작업을 하려면 밀어내는 동작의 터치를 많이 해야 하는데 이 방식은 다소 불편한 것 같다. 개선할 수 없겠는가?
   [답변] 그렇다면 페이지 하단의 영역을 조금 할당해서 전체 슬라이드 목록을 보여주는 이미지 리스트를 제공하고 해당 이미지를 선택하면 곧바로 관련된 슬라이드로 이동하게끔 만들어 보겠다.

3. **[질문]** 전체 슬라이드 목록을 담고 있는 이미지 리스트도 터치를 이용해 좌/우로 화면을 밀어낼 수 있어야 하겠다. 가능한가?
   [답변] 다행히 윈도우 폰의 목록 컨트롤은 기본적으로 터치 인터페이스를 지원하므로 쉽게 구현할 수 있다.

다시 회의 결과를 반영해서 "페이지 2"의 스케치를 보강해 보자.

그림 21.8 ShowController 페이지 2 스케치의 최종 버전



이 화면 역시 최종적으로 기획팀과의 협의를 거쳐 확정해야 한다.

응용 프로그램을 개략적으로 스케치해보는 단계는 매우 중요하다. 스케치 단계를 통해 마치 실제로 서비스를 사용하는 상황을 묘사하면서 합의를 이끌어낼 수 있기 때문에 코드가 변경될 가능성을 그만큼 줄일 수 있다. 이 단계를 거치지 않으면 프로젝트를 진행할 때 최초의 요구사항과 맞지 않는 부분을 발견할 가능성이 높아지고 그로 인해 개발자가 가장 싫어하는 코드 재작업이 발생할 가능성이 커진다. 현업에서 프로젝트를 진행하다 보면 이렇게까지 협의를 했는데도 기본 전제였던 요구사항까지 바뀌는 사태가 발생해 코드를 대대적으로 변경해야 하는 경우도 있다.

21.6 응용 프로그램 개발

응용 프로그램을 스케치했으면 이제 프로그램을 본격적으로 개발하기 시작한다. 순서로 보면 서버 역할을 하는 PPTShow를 먼저 개발하는 것이 좋다. 따라서 PPTShow를 마무리하고 이어서 ShowController 앱 개발을 시작하겠다.

20.6.1 개발 환경 준비

OfficePresenter 응용 프로그램은 두 개의 응용 프로그램(WPF, 윈도우 폰 앱)으로 구성돼 있는데, 문제는 해당 응용 프로그램에 대한 프로젝트 지원이 비주얼 스튜디오 익스프레스 버전에서는 "for Windows Desktop"과 "for Windows Phone"용으로 나뉘어 있다는 점이다. 따라서 PPTShow WPF 응용 프로그램은 "for Windows Desktop" 버전의 비주얼 스튜디오 익스프레스로 개발하고 ShowController 윈도우 폰 앱은 "for Windows Phone" 버전으로 개발해야 한다.

표 21.3 비주얼 스튜디오 익스프레스의 개발 환경

솔루션	프로젝트	응용 프로그램 유형	비주얼 스튜디오 익스프레스
OfficePresenter	ShowController	윈도우 폰 응용 프로그램	for Windows Phone
	PPTShow	WPF 응용 프로그램	for Windows Desktop

반면 상용 버전의 비주얼 스튜디오는 모든 응용 프로그램 유형을 지원하므로 표 21.3처럼 개발 환경을 나누지 않아도 된다. 편리하긴 해도 이 책의 독자들이 익스프레스 버전을 주로 사용한다는 가정하에 개발 환경을 나눠서 진행하는 쪽으로 설명한다.

이와 함께 PPTShow를 개발하는 PC에는 오피스 프로그램도 설치해야 한다. 파워포인트 쇼를 제어하는 것이기에 당연한 준비 사항인데, 오피스 프로그램의 무료 버전은 없지만 60일 동안 한시적으로 사용할 수 있는 체험판을 아래 경로에서 내려받아 설치한다.

Download Microsoft Office Professional Plus 2013 (60일 체험판) ; http://technet.microsoft.com/en-us/evalcenter/jj192782.aspx

20.6.2 PPTShow 개발

첫 단계로 "PPTShow"라는 이름의 솔루션을 만들고 그 안에 WPF 응용 프로그램 유형으로 "PPTShow" 프로젝트를 추가한다.

PPTShow의 기능은 크게 세 가지로 나눌 수 있다.

- 기본적인 UI 구성
- ShowController와 통신하는 HTTP 서버 역할
- 파워포인트 쇼를 제어하는 역할

가장 쉬운 것부터 차례대로 구현을 시작해 보자.

20.6.2.1 UI 구성

그림 21.4의 스케치에 따라 XAML 내에 적절하게 컨트롤을 배치하자.

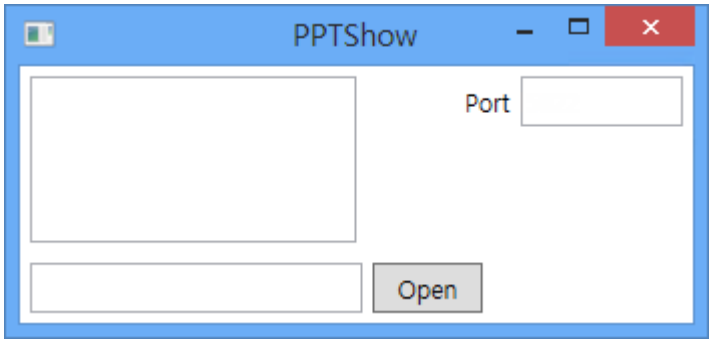
예제 21.1 PPTShow UI 디자인

```
// ======= MainWindow.xaml =======
<Window x:Class="PPTShow.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="PPTShow" Height="167" Width="352">
```

```
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
            <RowDefinition Height="35"></RowDefinition>
        </Grid.RowDefinitions>
        <Grid Margin="5">
            <Grid.ColumnDefinitions>
                <ColumnDefinition></ColumnDefinition>
                <ColumnDefinition></ColumnDefinition>
            </Grid.ColumnDefinitions>
            <ListBox>
            </ListBox>
            <Grid Grid.Column="1">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition></ColumnDefinition>
                    <ColumnDefinition></ColumnDefinition>
                </Grid.ColumnDefinitions>
                <Label HorizontalAlignment="Right">Port</Label>
                <TextBox VerticalAlignment="Top" Height="25"
                         VerticalContentAlignment="Center"
                         Grid.Column="1" IsReadOnly="True" />
            </Grid>
        </Grid>

        <Grid Margin="5" Grid.Row="1">
            <Grid.ColumnDefinitions>
                <ColumnDefinition></ColumnDefinition>
                <ColumnDefinition Width="60"></ColumnDefinition>
                <ColumnDefinition Width="100"></ColumnDefinition>
            </Grid.ColumnDefinitions>
            <TextBox IsReadOnly="True" Text="{Binding Path=FilePath}" />
            <Button Margin="5, 0, 0, 0" Grid.Column="1" Content="Open" />
            <Label  Margin="5, 0, 0, 0" Grid.Column="2"></Label>
        </Grid>
    </Grid>
</Window>
```

이렇게 디자인하고 프로젝트를 실행하면 다음과 같은 화면을 볼 수 있다.

그림 21.9 PPTShow 응용 프로그램의 XAML 디자인



자, 그럼 가장 쉬운 포트 값을 먼저 채워볼 텐데 단순히 XAML에 직접 입력하는 방법도 가능하지만

```
<TextBox VerticalAlignment="Top" Height="25" Text="5022"
                        VerticalContentAlignment="Center"
                        Grid.Column="1" IsReadOnly="True" />
```

여기서는 향후 사용자가 편집할 것에 대비해 데이터 바인딩을 이용해 처리한다.

```
<TextBox VerticalAlignment="Top" Height="25" Text="{Binding Path=Port}"
                        VerticalContentAlignment="Center"
                        Grid.Column="1" IsReadOnly="True" />
```

MainWindow의 생성자에서는 DataContext를 this로 초기화하고 데이터 바인딩을 위한 INotifyPropertyChanged 인터페이스와 Port 속성을 구현한다.

```
// ======= MainWindow.xaml.cs =======
using System.Windows;
using System.Windows.Documents;
using System.Net;
using System.IO;
using System.Threading;
using Microsoft.Win32;
using System.Windows.Threading;
using System.ComponentModel;

namespace PPTShow
{
    public partial class MainWindow : Window, INotifyPropertyChanged
    {
        short _port;
        public short Port
        {
            get { return this._port; }

            set
            {
                if (this._port == value)
                {
                    return;
                }

                this._port = value;
                OnPropertyChanged("Port");
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;
        protected virtual void OnPropertyChanged(string propertyName)
        {
            if (PropertyChanged == null)
            {
                return;
            }

            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
```

```
        public MainWindow()
        {
            InitializeComponent();
            this.DataContext = this;

            this.Port = 5022;
        }
    }
}
```

다음으로 IP 목록을 채워 넣는 작업을 처리한다. 물론 이것도 데이터바인딩을 이용해 처리한다.

```
<ListBox ItemsSource="{Binding Path=IPList}"></ListBox>
```

```
// ======= MainWindow.xaml.cs =======
using System.Windows;
//.....[생략].....
using System.Collections.ObjectModel;
using System.Net.Sockets;

namespace PPTShow
{
    public partial class MainWindow : Window, INotifyPropertyChanged
    {
        // ..... [생략] .....

        ObservableCollection<string> _iPList;
        public ObservableCollection<string> IPList
        {
            get { return this._iPList; }

            set
            {
                if (this._iPList == value)
                {
                    return;
                }

                this._iPList = value;
                OnPropertyChanged("IPList");
            }
        }

        public MainWindow()
        {
            InitializeComponent();
            this.DataContext = this;

            this.Port = 5022;

            this.IPList = new ObservableCollection<string>();
            PopulateIPList(this.IPList);
        }

        private void PopulateIPList(ObservableCollection<string> list)
```

```
        {
            IPAddress[] addrs = Dns.GetHostEntry(Dns.GetHostName()).AddressList;

            foreach (IPAddress ipAddr in addrs)
            {
                if (ipAddr.AddressFamily == AddressFamily.InterNetwork)
                {
                    list.Add(ipAddr.ToString());
                }
            }
        }
    }
}
```

IP 주소를 구하는 방법은 이전에 소켓을 다루면서 설명한 바 있다. 위의 코드는 IPList 컬렉션에 값을 채울 테고, XAML에 바인딩된 영향으로 자연스럽게 ListBox 요소에 IP 목록이 출력된다.

마지막으로 Open 버튼과 문서의 로딩 상태를 보여주는 Label 컨트롤을 처리해 보자.

```
<TextBox IsReadOnly="True" Text="{Binding Path=FilePath}" />
<Button Margin="5,0,0,0" Grid.Column="1" Click="btnOpenClicked" Content="Open" />
<Label Margin="5,0,0,0" Grid.Column="2" Content="{Binding Path=Ready}"></Label>
```

PPT 파일을 선택하기 전에는 Ready 변수에 "Not Loaded"를 출력하고 파일을 선택한 후에는 FilePath 바인딩 변수에 파일 경로를 넣고 Ready 변수의 상태는 "Loaded" 메시지로 바꾸는 처리를 해준다.

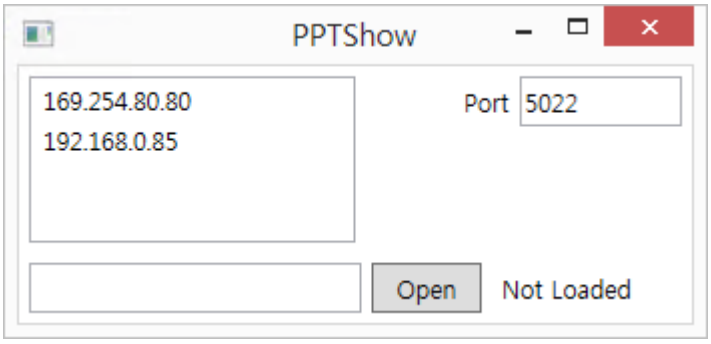
예제 21.2 PPT 파일 선택을 위한 코드 추가

```
// ======= MainWindow.xaml.cs =======
using System.Windows;
//.....[생략].....

namespace PPTShow
{
    public partial class MainWindow : Window, INotifyPropertyChanged
    {
        // ..... [생략] .....

        public MainWindow()
        {
            // ..... [생략] .....
            this.Ready = "Not Loaded";
        }

        // ..... [생략] .....

        void btnOpenClicked(object sender, RoutedEventArgs e)
        {
            OpenFileDialog openDlg = new OpenFileDialog();
            if (openDlg.ShowDialog() == true)
```

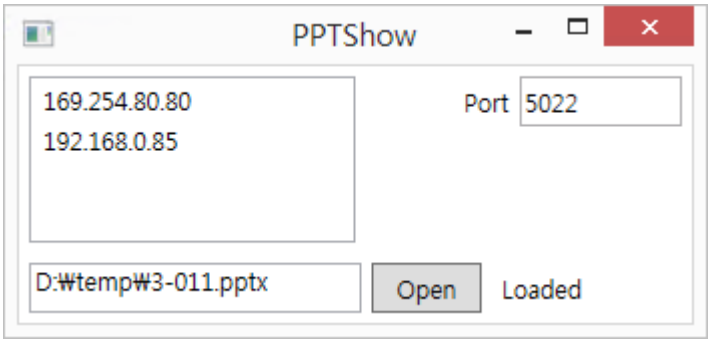
```
            {
                string path = openDlg.FileName;
                ProcessDocument(path);
            }
        }

        private void ProcessDocument(string path)
        {
            if (File.Exists(path) == false)
            {
                return;
            }

            // 현재 단계에서는 이 부분의 코드는 구현하지 않는다.

            this.FilePath = path;
            this.Ready = "Loaded";
        }

        string _filePath;
        public string FilePath
        {
            get { return this._filePath; }

            set
            {
                if (this._filePath == value)
                {
                    return;
                }

                this._filePath = value;
                OnPropertyChanged("FilePath");
            }
        }

        string _ready;
        public string Ready
        {
            get { return this._ready; }

            set
            {
                if (this._ready == value)
                {
                    return;
                }

                this._ready = value;
                OnPropertyChanged("Ready");
            }
        }
    }
}
```

여기까지 구현하고 실행하면 다음과 같은 화면을 볼 수 있다.

그림 21.10 PPTShow 실행 화면



"Open" 버튼을 누르고 원하는 PPT 파일을 선택하는 것까지 해보자. 그럼 PPT 파일 경로가 왼쪽 텍스트 상자에 출력되고, 상태 메시지는 "Loaded"로 바뀌는 것을 확인할 수 있다.



기본적인 UI 구성은 이 정도로 구현할 수 있다. 이어서 UI에 설정된 IP와 포트로 동작하는 HTTP 서버를 구현해 보자.

20.6.2.2 HTTP 서버 구현

HTTP 서버는 소켓을 이용해 구현할 수 있는데, 이미 6.7 '네트워크 통신'의 'HTTP 통신' 절에서 서버를 만드는 방법을 알아봤다. 'HTTP 통신' 절에서 작성한 소스코드를 보강해서 직접 HTTP 서버 기능을 구현해도 되지만 작업 시간을 줄이기 위해 C#으로 구현된 HTTP 서버를 웹으로 검색해 찾아봤다.

Simple HTTP Server in C# ; http://www.codeproject.com/Articles/137979/Simple-HTTP-Server-in-C

웹에는 이처럼 오픈소스로 공개돼 있는 다양한 소스코드를 구할 수 있으므로 이를 적절히 잘 활용하는 것도 개발자의 중요한 능력 중 하나다.

위의 소스코드를 내려받아 프로젝트에 추가한다. 이와 함께 HttpServer 클래스를 상속받아 다음과 같이 MyHttpServer 클래스를 정의한 MyHttpServer.cs 파일을 프로젝트에 추가한다.

```
// ======= MyHttpServer.cs =======
using System;
using Bend.Util;
using System.IO;

namespace PPTShow
{
    public class MyHttpServer : HttpServer
    {
        public MyHttpServer(int port)
            : base(port)
        {
        }

        // HttpServer 타입은 GET 요청이 올 때마다
        // handleGETRequest 메서드를 실행해 준다.
        // 따라서 이 메서드에서 특정 URL로 요청이 들어온 것을 감지할 수 있다.
        public override void handleGETRequest(HttpProcessor p)
        {
            string urlText = p.http_url;

            // URL에 '?' 문자 이후로 포함된 문자열을 제거한다.
            // 예를 들어, "http://localhost:5022/test?dummy=123" 이라고 요청이 오면
            // urlText 변수는 "http://localhost:5022/test" 문자열만 포함한다.
            int pos = urlText.IndexOf("?");
            if (pos != -1)
            {
                urlText = urlText.Substring(0, pos);
            }

            // 동작을 검증할 수 있는 테스트 요청 처리
            if (urlText.EndsWith("/test") == true)
            {
                p.writeSuccess("text/html");
                p.outputStream.Write(DateTime.Now + ": Hello World!");
            }
        }

        public override void handlePOSTRequest(HttpProcessor p, StreamReader data)
        {
        }
    }
}
```

HTTP 웹 서버는 프로그램이 시작될 때 대기를 시작하면 되므로 MyHttpServer 인스턴스를 MainWindow 생성자에서 만들어도 된다. 여기서 주의할 것은 소켓 대기는 스레드를 블로킹시키므로 별도의 스레드를 만들어 HTTP 요청을 대기하게 만들어야 한다.

```
// ======= MainWindow.xaml.cs =======
using System.Windows;
//.....[생략].....
```

```
namespace PPTShow
{
    public partial class MainWindow : Window, INotifyPropertyChanged
    {
        // …… [생략] ……

        MyHttpServer _listener;
        Thread _httpThread;

        public MainWindow()
        {
            // …… [생략] ……
            SetupHttpServer();
        }

        // …… [생략] ……

        private void SetupHttpServer()
        {
            _listener = new MyHttpServer(this.Port);
            _httpThread = new Thread(_listener.listen);
            _httpThread.IsBackground = true;
            _httpThread.Start();
        }
    }
}
```

자원 해제를 명시하기 위해 MainWindow의 Closed 이벤트 처리기를 작성하고 소켓을 닫는 것도 권장한다.

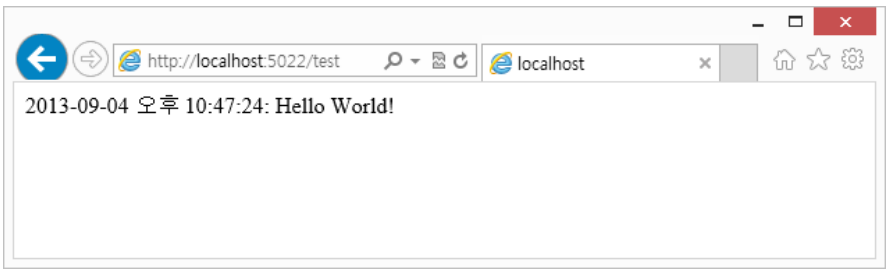
```
// ======= MainWindow.xaml.cs =======
using System.Windows;
//……[생략]……

namespace PPTShow
{
    public partial class MainWindow : Window, INotifyPropertyChanged
    {
        // …… [생략] ……

        MyHttpServer _listener;
        Thread _httpThread;

        public MainWindow()
        {
            // …… [생략] ……
            SetupHttpServer();
            this.Closed += new EventHandler(MainWindow_Closed);
        }

        // …… [생략] ……

        void MainWindow_Closed(object sender, EventArgs e)
```

```
        {
            ReleaseSocketResource();
        }

        private void ReleaseSocketResource()
        {
            try
            {
                if (_listener != null)
                {
                    _listener.Dispose();
                }
            }
            catch { }
            _listener = null;

            try
            {
                if (_httpThread != null)
                {
                    _httpThread.Abort();
                }
            }
            catch { }
            _httpThread = null;
        }
    }
}
```

엄밀히 말해서 여기서의 자원 해제 작업이 반드시 필요한 것은 아니다. 왜냐하면 MainWindow가 닫히는 경우 프로세스가 종료되므로 해당 프로세스에 속한 Background 스레드와 소켓 자원은 운영체제에 의해 모두 강제로 해제되기 때문이다.

PPTShow에 HTTP 서버 기능을 추가했으므로 이제 간단하게 PPTShow 프로그램을 실행하고 테스트해 보자. MyHttpServer.cs의 handleGETRequest 메서드에 추가한 /test 요청을 웹 브라우저를 이용해 테스트해보면 다음과 같은 결과를 볼 수 있다.

그림 21.11 http://localhost:5022/test 요청/응답 테스트

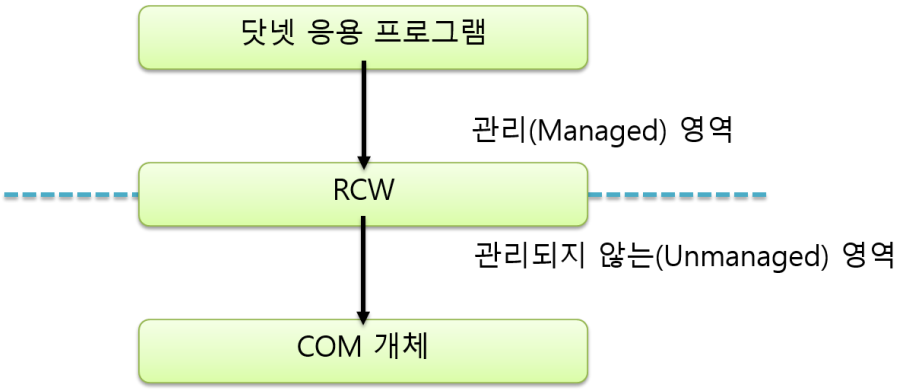


이런 식으로 HTTP 요청을 정의해 놓고 해당 요청에 대응시켜 파워포인트 쇼를 제어하는 코드가 실행되게 만들 것이다. 이 작업을 위해 우선 어떻게 파워포인트를 제어할 수 있는지 살펴보자.

20.6.2.3 파워포인트 제어 (1)

오피스의 파워포인트 프로그램은 그 하나가 거대한 구성요소(component)로서 외부에서 제어할 수 있게끔 구현돼 있다. 이렇게 제어가 가능할 수 있는 이유는 COM(Component Object Model)이라는 기술 덕분인데, 닷넷 프로그램에서는 이를 마치 클래스 다루듯이 호출할 수 있는 수단을 제공한다. 바로 RCW(Runtime Callable Wrapper)라는 기술이다.

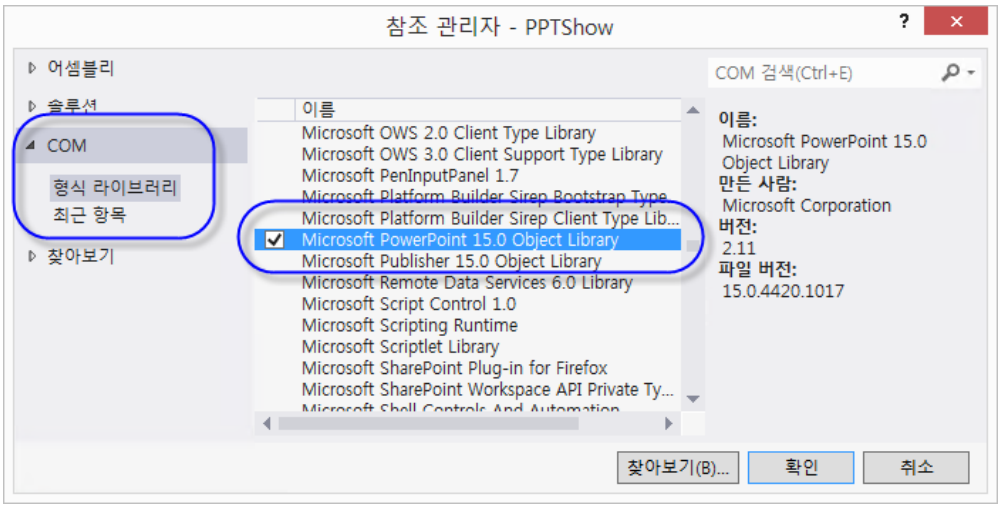
그림 21.12 RCW의 구조



닷넷 응용 프로그램과 COM 개체의 호출을 연결해 주는 RCW는 비주얼 스튜디오에서 생성해 준다. 생성되는 파일은 DLL 형식이며, 대개 "Interop"이라는 문자열이 포함된다. 실제로 비주얼 스튜디오에서 파워포인트에 대한 RCW를 생성해 보자.

솔루션 탐색기에서 "PPTShow" 프로젝트를 대상으로 마우스 오른쪽 버튼을 눌러 "참조 추가(Add Reference)" 메뉴를 선택하면 "참조 관리자" 대화상자가 나타난다. 그림 21.13에서 보는 것처럼 좌측의 "COM" / "형식 라이브러리" 범주를 선택한 후 나타나는 오른쪽 목록에서 "Microsoft PowerPoint 15.0 Object Library"를 선택한다.

그림 21.13 파워포인트 COM 개체의 RCW 생성을 위한 참조 추가



참고!	목록에 "Microsoft PowerPoint 15.0 Object Library"가 보이지 않는다면 해당 PC에 오피스 2013이 설치돼 있지 않기 때문이다.

"확인" 버튼을 누르면 비주얼 스튜디오는 Microsoft.Office.Interop.PowerPoint.dll 파일을 그 순간 생성해서 참조 목록에 추가한다. 그림 21.12에서 그려진 항목들을 현재 생성된 프로젝트에 대응시켜 보면 표 21.4와 같이 정리할 수 있다.

표 21.4 RCW 호출 관계

닷넷 응용 프로그램	RCW	COM
PPTShow	Microsoft.Office.Interop.PowerPoint.dll	파워포인트 프로그램

생성된 Interop을 이용해 파워포인트 프로그램을 제어하는 클래스를 정의할 PowerpointController.cs 파일을 추가하고 이제 코드를 조금씩 붙여보자.

앞에서 예제 21.2의 ProcessDocument 메서드가 비어 있었다는 것을 떠올려보자. 거기서 PPT 파일이 선택되면 우리는 파워포인트 프로그램을 실행하고 선택된 PPT 파일을 로드하는 것을 구현해야 한다. 따라서 PowerpointController 타입에는 Load 메서드를 구현하고 이를 ProcessDocument에서 호출하는 것부터 코드 작성을 시작한다.

```
// ======= PowerpointController.cs =======
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Office.Interop.PowerPoint;

namespace PPTShow
{
    public class PowerpointController
    {
        Application _app;
        Presentation _current;

        public bool Load(string documentPath)
        {
            try
            {
                if (_app == null)
                {
                    // 파워포인트 프로그램을 실행하고
                    _app = new Application();
                }

                // 실행된 파워포인트 프로그램을 보이게 한 후
                _app.Visible = Microsoft.Office.Core.MsoTriState.msoTrue;
```

```
                // 파워포인트 프로그램의 Presentations 객체를 구해
                var presentation = _app.Presentations;

                // Presentations 객체에 사용자가 선택한 PPT 파일을 로드한다.
                _current = presentation.Open(documentPath);

                return true;
            }
            catch
            {
            }

            return false;
        }
    }
}
```

RCW DLL에서 제공되는 Application 타입과 Presentation 타입을 사용하면 그 호출이 COM으로 전달된다. 결국 닷넷 프로그램에서 RCW 관련 클래스를 사용하는 것으로 파워포인트 프로그램을 제어할 수 있는 것이다. 이렇게 완성된 Load 메서드를 이제 MainWindow의 파일 선택 기능에 연결하면 된다.

```
// ======= MainWindow.xaml.cs =======
using System.Windows;
//……[생략]……

namespace PPTShow
{
    public partial class MainWindow : Window, INotifyPropertyChanged
    {
        // …… [생략] ……

        PowerpointController _pptController;

        public MainWindow()
        {
            // …… [생략] ……
            this._pptController = new PowerpointController();
            SetupHttpServer();
            // …… [생략] ……
        }

        // …… [생략] ……

        private void ProcessDocument(string path)
        {
            if (File.Exists(path) == false)
            {
                return;
            }

            if (_pptController.Load(path) == false)
```

```
            {
                return;
            }

            this.FilePath = path;
            this.Ready = "Loaded";
        }
    }
}
```

여기까지 코드를 완료하고 빌드한 후 테스트를 해보자. PPTShow에서 PPT 파일을 선택하면 이제 파워포인트 프로그램이 실행되면서 해당 PPT 파일이 자동적으로 로드되는 것을 확인할 수 있다.

PPTShow에서 RCW를 이용할 수 있게 됐으므로 이제 HTTP 서버를 통해 PPTShow를 제어할 수 있다. 간단한 유형으로 PPT 쇼를 시작하라는 명령을 내리는 것부터 구현해 보자. 이를 위해 우선 PowerpointController 타입에 StartShow 메서드를 구현한다.

```
// ======= PowerpointController.cs =======
using System;
// ..... [생략] .....

namespace PPTShow
{
    public class PowerpointController
    {
        Application _app;
        Presentation _current;

        // ..... [생략] .....

        public void StartShow(int startSlideNumber)
        {
            object inSlideShow = null;

            try
            {
                // 현재 파워포인트 프로그램이 Show 상태인지를 체크
                inSlideShow = _current.SlideShowWindow;
            }
            catch
            {
            }

            if (inSlideShow == null) // 쇼 상태가 아니라면
            {
                // 쇼를 시작한다.
                _current.SlideShowSettings.Run();

                // 지정된 슬라이드 번호가 파워포인트의 슬라이드 범위를 넘지 않게 만들고,
                int start = Math.Min(_current.Slides.Count, startSlideNumber);
```

```
                // 해당 슬라이드로 이동한다.
                // start 변수의 값이 1이라면 첫 번째 슬라이드가 보인다.
                _current.SlideShowWindow.View.GotoSlide(start,
                                        Microsoft.Office.Core.MsoTriState.msoTrue);
            }
        }
    }
}
```

이제 MyHttpServer 타입에서 PowerpointController의 StartShow 메서드를 호출해야 하는데, 현재는 그렇게 할 수 없다. 왜냐하면 MyHttpServer 타입은 PowerpointController에 대한 참조가 없기 때문이다. 따라서 MyHttpServer 생성자에서 PowerpointController를 전달받는 식으로 변경하고 그것을 이용해 /startShow 요청에서 StartShow 메서드를 호출하게 만든다.

```
// ======= MyHttpServer.cs =======
using System;
// ..... [생략] .....

namespace PPTShow
{
    public class MyHttpServer : HttpServer
    {
        PowerpointController _document;

        public MyHttpServer(PowerpointController document, int port)
            : base(port)
        {
            _document = document;
        }

        public override void handleGETRequest(HttpProcessor p)
        {
            string urlText = p.http_url;

            if (urlText.EndsWith("/test") == true)
            {
                // ..... [생략] .....
            }
            else if (urlText.Contains("/startShow") == true)
            {
                _document.StartShow(1);
                p.writeSuccess("text/html");
                p.outputStream.Write("OK");
            }
        }

        // ..... [생략] .....
    }
}
```

당연히 MainWindow의 생성자는 PowerpointController 인스턴스를 전달하도록 코드를 변경해야

한다.

```
// ======= MainWindow.xaml.cs =======
using System.Windows;
//.....[생략].....

namespace PPTShow
{
    public partial class MainWindow : Window, INotifyPropertyChanged
    {
        // ..... [생략] .....

        private void SetupHttpServer()
        {
            _listener = new MyHttpServer(this._pptController, this.Port);
            _httpThread = new Thread(_listener.listen);
            _httpThread.IsBackground = true;
            _httpThread.Start();
        }
    }
}
```

여기까지 코드를 작성하고 이제 다음과 같은 순서로 테스트를 진행해 보자.

1. PPTShow 프로그램을 실행한다.
2. "Open" 버튼을 눌러 PPT 파일을 선택한다. 그럼 파워포인트가 실행되고 선택된 문서가 로드된다.
3. 웹 브라우저를 실행하고 http://localhost:5022/startShow 요청을 보낸다. 그럼 파워포인트 프로그램은 현재 로드된 PPT 문서의 쇼를 시작한다.

지금 단계에서 그림 21.2의 구조를 다시 한번 들여다보자. 이쯤 되면 OfficePresenter 프로그램의 전체적인 동작 구조가 그려져야 한다. PPTShow는 파워포인트를 RCW 타입을 이용해 제어하고 윈도우 폰에서 실행되는 ShowController는 HTTP 요청을 통해 PPTShow를 제어하는 것이다. 아직은 제어 동작이 "startShow" 하나에 불과하지만 나머지 동작 과정은 startShow를 구현하는 방식과 유사하므로 그다지 어렵지 않게 구현해 나갈 수 있다.

20.6.2.4 파워포인트 제어 (2)

startShow 제어 외에 어떤 동작들이 필요할까? 이를 알아내기 위해서는 '21.5 응용 프로그램의 동작 시나리오 및 화면 스케치' 절에서 결정한 사항을 천천히 다시 읽어볼 필요가 있다.

윈도우 폰 앱에서 실행되는 ShowController 프로그램은 PPT 파일에 포함된 모든 슬라이드를 작은 그림 파일로 변환한 목록이 필요하다. 그리고 각 슬라이드에 포함된 메모 내용도 필요하다. 이 동작을 "/getSnapshot" 요청으로 정하자. 필자가 계획한 구현 방법은 다음과 같다.

1. PPTShow에서 PPT 파일을 선택해서 로드했을 때 슬라이드 하나당 이미지 하나로 대응시켜 파일로 저장한다.

2. PPT 파일에 포함된 모든 슬라이드를 열람하면서 메모 내용을 읽어들인다.
3. 추출된 모든 슬라이드의 이미지와 메모 목록을 JSON 형식으로 직렬화해서 단일 문자열로 변환한다.
4. 윈도우 폰 앱에서 /getSnapshot 요청이 오면 JSON으로 직렬화된 문자열을 그대로 반환한다.
5. (윈도우 폰 앱은 반환받은 JSON 문자열을 다시 역직렬화해서 이미지와 메모를 구한 다음 사용한다.)

물론 다른 방식으로 구현해도 된다. 단지 필자는 위의 방법으로 구현해 나가기로 결정했을 뿐이다. 여기서 한 가지 더 추가하면 JSON 직렬화/역직렬화를 쉽게 할 수 있게 PPT 파일과 그 안의 개별 슬라이드에 해당하는 클래스를 만들 것이다. 이를 위해 우선 개별 PPT 슬라이드 화면 정보를 담을 PPTPage 클래스를 추가한다.

```
// ======= PPTPage.cs =======
using System;

namespace PPTShow
{
    public class PPTPage
    {
        // 슬라이드를 이미지로 변경한 다음 문자열로 직렬화한 데이터
        string _imageAsText;
        public string ImageAsText
        {
            get { return _imageAsText; }
            set { _imageAsText = value; }
        }

        // 슬라이드에 포함된 메모 내용
        string _note;
        public string Note
        {
            get { return _note; }
            set { _note = value; }
        }
    }
}
```

그리고 PPTPage 페이지가 모여 결국 하나의 PPT 파일에 담긴 모든 슬라이드 정보를 담게 될 PPTDocument 클래스를 추가한다.

```
// ======= PPTDocument.cs =======
using System.Collections.Generic;

namespace PPTShow
{
    public class PPTDocument
    {
        public PPTDocument()
        {
            _list = new List<PPTPage>();
```

```
        }

        // PPT 슬라이드의 모든 내용을 보관하는 컬렉션
        List<PPTPage> _list;
        public List<PPTPage> List
        {
            get { return _list; }
            set { _list = value; }
        }

        // PPT 파일에 포함된 슬라이드의 수
        public int Count
        {
            get { return _list.Count; }
        }

        // PPTPage에 포함된 슬라이드 이미지의 가로 크기
        int _width;
        public int Width
        {
            get { return _width; }
            set { _width = value; }
        }

        // PPTPage에 포함된 슬라이드 이미지의 세로 크기
        int _height;
        public int Height
        {
            get { return _height; }
            set { _height = value; }
        }
    }
}
```

이제 본격적으로 PPT 파일을 로드하는 시점에 모든 슬라이드의 내용을 추출해 보자. PPT 파일을 로드하는 단계에서 처리할 것이므로 PowerpointController.cs 파일만 변경하면 된다. 이때 이미지 조작을 할 것이므로 System.Drawing 어셈블리 참조를 추가한다.

참고!	System.Drawing 어셈블리는 윈도우 폼 프로젝트인 경우 기본 참조가 되지만 WPF 프로젝트인 경우에는 그렇지 않다.

```
// ======= PowerpointController.cs =======
using System;
// ..... [생략] .....
using System.IO;
using System.Drawing;
using System.Drawing.Imaging;
using System.Web.Script.Serialization;
using Microsoft.Office.Core;

namespace PPTShow
```

```
{
    public class PowerpointController
    {
        Application _app;
        Presentation _current;

        const int _slideImageWideth = 480;
        const int _slideImageHeight = 358;

        // PPT 파일 하나의 모든 슬라이드 내용이 이 변수 하나에 문자열로 직렬화된다.
        string _documentAsText;
        public string DocumentAsText
        {
            get { return _documentAsText; }
        }

        public bool Load(string documentPath)
        {
            try
            {
                // ..... [생략] .....

                // 임시로 사용될 폴더를 하나 만든다.
                string tempPath = Path.Combine(Path.GetTempPath(),"ShowPresenter");
                tempPath = Path.Combine(tempPath, Guid.NewGuid().ToString());
                Directory.CreateDirectory(tempPath);

                if (Directory.Exists(tempPath) == true)
                {
                    // Presentation 객체의 Export 메서드는 PPT 파일에 포함된
                    // 슬라이드 하나당 이미지 파일 하나로 만들어 저장한다.
                    // PPT에 3개의 슬라이드가 있는 경우 "슬라이드1.JPG", "슬라이드2.JPG",
                    // "슬라이드3.JPG" 파일 3개가 tempPath로 지정된 폴더에 생성된다.
                    _current.Export(tempPath, "JPG");

                    // tempPath 폴더에 저장된 모든 이미지를 읽고,
                    // Presentation으로부터 해당 슬라이드의 메모를 찾아내서
                    // PPTDocument에 보관한다.
                    PPTDocument doc = ReadAll(tempPath);

                    // PPT 파일 하나에 대응되는 PPTDocument의 내용을
                    // JSON 객체로 직렬화한다. 따라서 문자열 하나로 저장된다.
                    JavaScriptSerializer json = new JavaScriptSerializer();
                    json.MaxJsonLength = Int32.MaxValue;
                    _documentAsText = json.Serialize(doc);
                }

                // 임시로 사용된 폴더는 더는 필요하지 않으므로 삭제한다.
                Directory.Delete(tempPath, true);

                return true;
            }
            catch
```

```
    {
    }

    return false;
}

PPTDocument ReadAll(string tempPath)
{
    int slideNumber = 0;

    // 슬라이드 하나당 이미지 파일로 저장된 파일을 열람한다.
    // 이때 정렬 함수를 CompareOnlyNumbers로 지정한 것에 유의하자.
    List<string> files = new List<string>();
    files.AddRange(Directory.EnumerateFiles(tempPath));
    files.Sort(CompareOnlyNumbers);

    PPTDocument document = new PPTDocument();

    // Presentation 객체의 Slides 속성을 통해 PPT 파일에 포함된 모든 슬라이드를
    // 열람할 수 있다. 열람하면서 해당 슬라이드에 포함된 메모를 구한다.
    foreach (Slide slide in _current.Slides)
    {
        string note = string.Empty;

        SlideRange nodePath = slide.NotesPage;

        foreach (Microsoft.Office.Interop.PowerPoint.Shape shape in
                                                        nodePath.Shapes)
        {
            PpPlaceholderType type = PpPlaceholderType.ppPlaceholderObject;

            try
            {
                type = shape.PlaceholderFormat.Type;
            } catch { }

            if (type == PpPlaceholderType.ppPlaceholderBody)
            {
                if (shape.HasTextFrame == MsoTriState.msoTrue)
                {
                    if (shape.TextFrame.HasText == MsoTriState.msoTrue)
                    {
                        note += shape.TextFrame.TextRange.Text;
                    }
                }
            }
        }

        // 메모와 함께 슬라이드 이미지 파일의 내용을
        // PPTPage 하나의 객체에 담는다.
        PPTPage page = new PPTPage();
        page.Note = note;
        page.ImageAsText = ConvertToImage(files[slideNumber],
                                    _slideImageWideth, _slideImageHeight);

        slideNumber++;
```

```
            document.List.Add(page);
        }

        document.Width = _slideImageWideth;
        document.Height = _slideImageHeight;
        return document;
    }

    private string ConvertToImage(string path, int width, int height)
    {
        try
        {
            // 이미지 파일을 Image 객체로 로드한다.
            using (Image img = Image.FromFile(path))
            {
                // 이미지 파일을 원본 크기 그대로 윈도우 폰에 전송할 필요는 없다.
                // 폰 화면은 작기 때문에 그에 맞는 크기로 변환해야
                // 전송되는 데이터의 양을 줄일 수 있다.
                // 따라서 이미지 크기를 축소하는 GetThumbnailImage 메서드를 이용한다.
                using (Image thumbnail = img.GetThumbnailImage(width, height,
                                         delegate { return false; }, IntPtr.Zero))
                {
                    MemoryStream ms = new MemoryStream();
                    thumbnail.Save(ms, ImageFormat.Jpeg);
                    ms.Position = 0;

                    // 바이트 배열이 아닌 텍스트로 반환한다.
                    // Base64 인코딩은 바이트를 텍스트로 바꿔준다.
                    return Convert.ToBase64String(ms.GetBuffer(), 0
,                                                 (int)ms.Length);
                }
            }
        }
        catch
        {
        }

        return string.Empty;
    }

    // 파일명 중에서 숫자만 취해 크기를 비교한다.
    // 따라서 파일명에 포함된 숫자로 파일 목록을 정렬한다.
    private static int CompareOnlyNumbers(string x, string y)
    {
        string fileNameX = System.IO.Path.GetFileNameWithoutExtension(x);
        string fileNameY = System.IO.Path.GetFileNameWithoutExtension(y);

        int xn = EraseChars(fileNameX);
        int yn = EraseChars(fileNameY);

        return xn.CompareTo(yn);
    }
```

```
        // 문자열에서 숫자만 취해서 정수로 반환한다.
        private static int EraseChars(string x)
        {
            StringBuilder sb = new StringBuilder();
            foreach (char ch in x)
            {
                if (Char.IsNumber(ch) == true)
                {
                    sb.Append(ch);
                }
            }

            return Int32.Parse(sb.ToString());
        }
    }
}
```

위 코드의 핵심은 _current.Export 메서드다. 이 메서드를 호출하면 파워포인트 프로그램은 현재 로드된 PPT 파일에 담긴 모든 슬라이드를 각각 대응되는 이미지 파일로 저장한다. 이 메서드가 있다는 사실을 모르면 PPT의 슬라이드를 그림으로 가져올 수 없기 때문에 ShowController에서 보일 2번째 페이지인 그림 21.6의 동작 화면을 구현할 수 없다. 코드가 길지만 나머지 부분은 _current.Export로 저장된 이미지 파일을 슬라이드 순서에 맞게 다시 읽어 PPTDocument에 넣는 것이 전부다.

일단 이런 식으로 _documentAsText 문자열 변수에 PPT 파일의 모든 내용을 담았다면 다음으로 HTTP 서버의 코드에 해당 변수의 값을 가져갈 수 있는 요청을 추가하면 된다.

```
// ======= MyHttpServer.cs =======
using System;
// ..... [생략] .....

namespace PPTShow
{
    public class MyHttpServer : HttpServer
    {
        // ..... [생략] .....

        public override void handleGETRequest(HttpProcessor p)
        {
            string urlText = p.http_url;

            // ..... [if 생략] .....
            else if (urlText.EndsWith("/getSnapshot") == true)
            {
                p.writeSuccess("application/json");
                p.outputStream.Write(_document.DocumentAsText);
            }
        }

        // ..... [생략] .....
```

```
    }
}
```

writeSuccess의 값을 "application/json"으로 바꾼 것은 HTTP 통신의 관례를 따른 것이다. DocumentAsText 변수에 담긴 내용이 JSON 직렬화를 한 내용이기 때문에 응답을 받는 측에 이 사실을 확실히 알리는 역할을 한다(/startShow의 응답에 사용된 것과 같이 "text/html"로 반환해도 무방하다).

이번에도 테스트를 위해 웹 브라우저를 이용해 방문해보자.

1. PPTShow 프로그램을 실행한다.
2. "Open" 버튼을 눌러 PPT 파일을 선택한다. 그럼 파워포인트가 실행되고 선택된 문서가 로드된다.
3. 웹 브라우저를 실행하고 http://localhost:5022/getSnapshot 요청을 보낸다. 그럼 JSON 형식으로 직렬화된 문자열을 반환받을 수 있다.

반환받은 문자열을 보면 다음과 같은 식이다.

{"List":[{"ImageAsText":"**/9j/4AAQSkZJRgABAQEAYABgAAD/2wBDAAgGBgcGBQgHBwcJCQgKDBQNDAsLDBkSEw8UHRofHh0aHBwgJC4nICIsIxwcKDcpLDAxNDQ0Hyc5PTgyPC4zNDL/2wBDAQ**
.....[생략].....
KACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKAP/9k=","Note":""}],"Count":17,"Width":480,"Height":358}

중간에 바이트가 Base64 인코딩 방식으로 변환되어 문자열로 변환된 결과를 확인할 수 있다. 물론 이 문자열을 다시 Base64 형식으로 디코딩하면 원래의 바이트 배열로 변환되고 그것은 곧 "이미지 파일의 내용"이 된다.

"getSnapshot" 요청을 통해 이제 윈도우 폰 앱에서 PPT 파일에 포함된 슬라이드를 그림 목록으로 보여줄 수 있는 기반이 마련됐다. 하지만 아직 끝난 것이 아니다. 앱 사용자가 슬라이드 이미지를 선택하면 PC에서 실행 중인 PPTShow 프로그램에서는 해당 슬라이드를 쇼에 나타나게 해야 한다. 이 기능을 구현하기란 매우 쉽다. 다음과 같이 PowerpointController 타입에 쇼에 나타날 슬라이드 번호를 지정할 수 있는 메서드를 만들고

```
// ======= PowerpointController.cs =======
using System;
// ..... [생략] .....

namespace PPTShow
{
    public class PowerpointController
    {
        Application _app;
```

```
        Presentation _current;

        // ..... [생략] .....

        public void SetCurrentSlide(int slideNumber)
        {
            try
            {
                _current.SlideShowWindow.View.GotoSlide(slideNumber,
                                                        MsoTriState.msoTrue);
            }
            catch { }
        }
    }
}
```

HTTP 통신으로 이 메서드를 호출할 수 있는 수단을 만들어준다.

```
// ======= MyHttpServer.cs =======
using System;
// ..... [생략] .....

namespace PPTShow
{
    public class MyHttpServer : HttpServer
    {
        // ..... [생략] .....

        public override void handleGETRequest(HttpProcessor p)
        {
            string urlText = p.http_url;

            // ..... [if 생략] .....
            else if (urlText.Contains("/setSlide/") == true)
            {
                string txt = urlText.Substring(urlText.LastIndexOf('/') + 1);

                int slide;
                if (Int32.TryParse(txt, out slide) == true)
                {
                    _document.SetCurrentSlide(slide);
                }

                p.writeSuccess("text/html");
                p.outputStream.Write("OK");
            }
        }

        // ..... [생략] .....
    }
}
```

이 동작을 테스트해보자.

1. PPTShow 프로그램을 실행한다.
2. "Open" 버튼을 눌러 PPT 파일을 선택한다. 그럼 파워포인트가 실행되고 선택된 문서가 로드된다.
3. 웹 브라우저를 실행하고 http://localhost:5022/startShow 요청을 보낸다. 그럼 PPT 쇼가 시작하고, 전체 화면을 차지한다(웹 브라우저가 가려지므로 "Alt + Tab" 키를 눌러 가려진 웹 브라우저를 나타나게 만든다).
4. 화면에 다시 나타난 웹 브라우저를 이용해 http://localhost:5022/setSlide/2라고 요청을 보낸다. 그럼 배경에 진행 중인 PPT 쇼는 두 번째 슬라이드로 바뀐다. 같은 방식으로 숫자만 바꿔서 setSlide 요청을 다양하게 테스트한다.

여기까지 PPTShow의 모든 코드가 완성됐다. 다음 단계로 넘어가기 전에 PPTShow 프로그램에서 정의한 HTTP 요청을 정리해 보자.

표 21.5 PPTShow에 정의된 HTTP 요청

요청	응답 포맷	설명

/getSnapshot	{
	"List":[
	{"ImageAsText":"...[base64]...",
	"Note":"...[memo]..."},
	{"ImageAsText":"...[base64]...",
	"Note":"...[memo]..."},
	{ ...[PPT 슬라이드 수만큼 반복]... }
	],
	"Width":480," Height":358
	}
		현재 PPT의 모든 슬라이드에 대한 스냅샷 이미지와 메모를 반환

/setSlide/*[number]*	OK	Show 중인 PPT에서 보일 슬라이드를 지정한다.
/startShow	OK	PPT 쇼를 시작한다.

이제 정해진 HTTP 통신을 이용해 쇼를 제어할 수 있는 윈도우 폰 앱을 만들 차례다.

20.6.3 ShowController 개발

앞에서도 설명했듯이 비주얼 스튜디오를 유료 버전이 아닌 익스프레스 버전을 사용하고 있다면 별도로 솔루션을 만들어 윈도우 폰 앱 프로젝트를 진행해야 한다.

첫 단계로 "ShowController"라는 이름의 솔루션을 만들고 다시 윈도우 폰 앱 유형으로 "ShowController" 프로젝트를 추가한다.

ShowController의 기능은 크게 두 가지로 나눌 수 있다.

- PPTShow에 연결해 PPT 파일 정보를 가져오는 MainPage
- Panorama 컨트롤을 이용해 쇼를 제어하는 PPTController 페이지

각 화면은 그림 21.5, 그림 21.6에서 이미 정의했으므로 그에 따라 XAML을 구성해 나가면서 구현한다.

20.6.3.1 연결 페이지 – MainPage.xaml

연결 페이지의 UI 구성은 간단하다. IP 주소와 포트 번호만 입력받고 연결 버튼을 누르기만 하면 된다.

```
// ======= MainPage.xaml =======
<phone:PhoneApplicationPage
    x:Class="ShowController.MainPage"
    ……[생략]……
    shell:SystemTray.IsVisible="True">

    <Grid x:Name="LayoutRoot" Background="Transparent">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
            <TextBlock x:Name="ApplicationTitle" Text="Show Controller"
                       Style="{StaticResource PhoneTextNormalStyle}"/>
        </StackPanel>

        <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
            <Grid.RowDefinitions>
                <RowDefinition Height="80"></RowDefinition>
                <RowDefinition Height="80"></RowDefinition>
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition></ColumnDefinition>
                <ColumnDefinition></ColumnDefinition>
                <ColumnDefinition></ColumnDefinition>
            </Grid.ColumnDefinitions>

            <TextBlock VerticalAlignment="Center">IP</TextBlock>
            <TextBox Grid.Column="1" Grid.ColumnSpan="2"></TextBox>

            <TextBlock VerticalAlignment="Center" Grid.Row="1">Port</TextBlock>
            <TextBox Grid.Row="1" Grid.Column="1"></TextBox>
            <Button Padding="0" Grid.Row="1" Grid.Column="2">Connect</Button>
        </Grid>
    </Grid>

</phone:PhoneApplicationPage>
```

PPTShow에서처럼 IP 주소와 포트 번호는 데이터 바인딩으로 해결한다.

```
// ======= MainPage.xaml =======
……[생략]……
<TextBlock VerticalAlignment="Center">IP</TextBlock>
<TextBox Text="{Binding Path=IPSelected, Mode=TwoWay}"
                                    Grid.Column="1" Grid.ColumnSpan="2"></TextBox>

<TextBlock VerticalAlignment="Center" Grid.Row="1">Port</TextBlock>
<TextBox Text="{Binding Path=Port, Mode=TwoWay}"
                                            Grid.Row="1" Grid.Column="1"></TextBox>

……[생략]……
```

바인딩에 사용된 구문 가운데 Mode=TwoWay는 전에 설명한 적이 없는데, 표 21.6에 가능한 값을 정리해 놨으니 참고하자.

표 21.6 데이터 바인딩의 Mode 값 정리

Mode	설명
OneWay	코드의 변수 값이 바뀌면 XAML 요소에 해당 값이 반영된다. 하지만 XAML 요소에서 값이 바뀌어도 코드의 변수에는 반영되지 않는다. 즉, 단방향으로만 값이 동기화된다.
TwoWay	코드의 변수 값이 바뀌면 XAML 요소에도 값이 반영되고, XAML 요소에서도 값이 바뀌면 코드의 변수로 반영된다. 즉, 양방향으로 값이 동기화된다.
OneTime	OneWay와 성격은 같지만 단 한 번만 동기화가 발생한다. 한번 코드의 변수 값이 바뀌면 XAML 요소로 동기화가 되지만 그 이후로 코드의 변수 값이 바뀌는 것은 XAML 요소로 반영되지 않는다.

WPF 응용 프로그램에서 굳이 바인딩의 Mode 값을 지정하지 않았던 것은 기본값이 TwoWay였기 때문이다. 하지만 윈도우 폰에서는 기본값이 OneWay이기 때문에 양방향 동기화를 위해서는 명시적으로 TwoWay로 지정해야 한다.

이제 XAML 요소에 바인딩을 한 변수를 MainPage 타입에 정의해 준다.

```
// ======= MainPage.xaml.cs =======
using System.ComponentModel;
using System.Windows;
using Microsoft.Phone.Controls;

namespace ShowController
{
    public partial class MainPage : PhoneApplicationPage, INotifyPropertyChanged
    {
        string _iPSelected;
        public string IPSelected
        {
            get { return this._iPSelected; }
```

```
            set
            {
                if (this._iPSelected == value)
                {
                    return;
                }

                this._iPSelected = value;
                OnPropertyChanged("IPSelected");
            }
        }

        int _port;
        public int Port
        {
            get { return this._port; }

            set
            {
                if (this._port == value)
                {
                    return;
                }

                this._port = value;
                OnPropertyChanged("Port");
            }
        }

        public MainPage()
        {
            InitializeComponent();

            this.DataContext = this;
            this.Port = 5022; // 초기값으로 5022 포트 번호를 미리 지정
        }

        public event PropertyChangedEventHandler PropertyChanged;
        protected virtual void OnPropertyChanged(string propertyName)
        {
            if (PropertyChanged == null)
            {
                return;
            }

            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

여기까지 입력을 마쳤으면 실행해 보자. 그럼 에뮬레이터에서 그림 21.14와 같이 실행되는 것을 볼 수 있다.

그림 21.14 MainPage 연결 화면

기획에 의하면 이 화면을 통해 PPTShow 프로그램에서 대기 중인 HTTP 서버에 getSnapshot 요청을 보내 PPT 파일의 내용을 가져오는 기능을 구현해야 한다. 따라서 Connect 버튼에 이벤트 처리기를 연결하고,

```
// ======= MainPage.xaml =======
……[생략]……
<Button Padding="0" Grid.Row="1" Grid.Column="2"
                                    Click="btnConnectClicked">Connect</Button>
……[생략]……
```

btnConnectClicked 메서드에서는 PPTShow에서 대기 중인 HTTP 서버에 /getSnapshot 요청을 WebClient 객체를 이용해 보내고 결과를 받아 처리한다.

```
// ======= MainPage.xaml.cs =======
using System.ComponentModel;
// ……[생략]……
using System;
using System.Net;
using Microsoft.Phone.Shell;

namespace ShowController
{
    public partial class MainPage : PhoneApplicationPage, INotifyPropertyChanged
```

```
    {
        // ……[생략]……

        private void btnConnectClicked(object sender, RoutedEventArgs e)
        {
            string url = string.Format("http://{0}:{1}/getSnapshot",
                                                    this.IPSelected, this.Port);
            Uri uri = new Uri(url);

            // 연결 버튼이 눌리면 지정된 IP/Port로 WebClient 객체를 이용해
            // HTTP 요청을 보낸다.
            WebClient wc = new WebClient();

            wc.Headers[HttpRequestHeader.IfModifiedSince] =
                                                    DateTime.UtcNow.ToString();
            wc.DownloadStringCompleted +=
                        new DownloadStringCompletedEventHandler(SnapshotCompleted);
            wc.DownloadStringAsync(uri);
        }

        void SnapshotCompleted(object sender, DownloadStringCompletedEventArgs e)
        {
            if (e.Error != null)
            {
                MessageBox.Show(e.Error.ToString());
            }
            else
            {
                try
                {
                    if (string.IsNullOrEmpty(e.Result) == false)
                    {
                        // 정상적으로 getSnapshot 결과를 받은 경우
                        string txt = e.Result;
                        // ……[생략: txt 내용으로 PPT 슬라이드 목록을 구한다.]……
                    }
                }
                catch (Exception ex)
                {
                    throw ex;
                }
            }
        }

        // ……[생략]……
    }
}
```

윈도우 폰에서의 WebClient는 동기 방식의 DownloadString 메서드를 제공하지 않는다. 이것은 마이크로소프트의 정책적인 결정에 따른 것인데, 폰의 사용자 반응성을 떨어뜨리는 동기화 메서드의 구현을 일부러 제공하지 않기 때문이다. 따라서 DownloadStringAsync 비동기 메서드만 사용할 수 있고 통신이 완료됐을 때 DownloadStringCompleted 이벤트 처리기를 구현함으로써 결과를

받아 처리할 수 있다.

성공적으로 결과를 반환받았다면 txt 문자열 변수에는 PPTShow에서 JSON 포맷으로 직렬화한 PPTDocument 객체의 내용이 들어 있다. 따라서 txt의 내용을 다시 JSON으로 역직렬화하면 PPTDocument의 내용을 그대로 복원할 수 있다.

```
// ======= MainPage.xaml.cs =======
using System.ComponentModel;
// ……[생략]……
using System.Text;
using System.Runtime.Serialization.Json;
using System.IO;

namespace ShowController
{
    public partial class MainPage : PhoneApplicationPage, INotifyPropertyChanged
    {
        // ……[생략]……

        void SnapshotCompleted(object sender, DownloadStringCompletedEventArgs e)
        {
            // ……[생략]……
            string txt = e.Result;

            MemoryStream ms = new MemoryStream(Encoding.UTF8.GetBytes(txt));
            DataContractJsonSerializer serializer =
                            new DataContractJsonSerializer(typeof(PPTDocument));
            PPTDocument document = serializer.ReadObject(ms) as PPTDocument;
        // ……[생략]……
    }
}
```

그런데 위의 코드를 컴파일하면 PPTDocument 타입이 ShowController 윈도우 폰 프로젝트에는 정의돼 있지 않으므로 오류가 발생한다. 이를 해결하려면 PPTShow에 포함된 PPTPage.cs 파일과 PPTDocument.cs 파일을 ShowController 프로젝트가 있는 폴더로 복사한 후 추가하면 된다. 추가된 PPTDocument/PPTPage 타입은 네임스페이스가 PPTShow로 돼 있으므로 using 문으로 추가하는 것도 잊지 말자.

```
// ======= MainPage.xaml.cs =======
using System.ComponentModel;
// ……[생략]……
using PPTShow;

namespace ShowController
{
    public partial class MainPage : PhoneApplicationPage, INotifyPropertyChanged
    {
        // ……[생략]……
    }
```

```
}
```

아직 처리해야 할 단계가 하나 남아 있는데, "Connect" 버튼이 눌린 경우 기획 단계에서 구상했던 그림 21.6의 UI가 탑재된 두 번째 페이지로 넘어가는 기능을 구현해야 한다. 추가돼야 할 페이지에 PPTController.xaml이라는 이름을 부여하고 최종적으로 SnapshotCompleted에서는 PPTController.xaml로 이동하는 것으로 마무리하자.

그런데 여기서 getSnapshot으로 구한 document 변수의 내용을 PPTController.xaml로 넘겨야 하는데 어떻게 하는 것이 좋을까? 앞의 '20.5 윈도우 폰 응용 프로그램' / '19.5.1 페이지 단위의 응용 프로그램 구현' 절에서 페이지 간 데이터 전송에 대해 다음과 같은 세 가지 방법이 있음을 설명했다.

1. QueryString을 이용한 키/값 전달
2. 전역 정적 변수를 이용한 상태 공유
3. 격리된 저장소를 이용해 상태 공유

이 가운데 PPTDocument 타입을 쿼리 문자열로 전달하는 방법은 바람직하지 않다. 그리고 전역 정적 변수를 이용하는 경우 tombstone 문제가 발생할 수 있기 때문에 남은 선택으로는 3번 방법이 최선일 것 같다(물론, tombstone을 무시한다면 2번 방법도 좋은 선택이 될 수 있다).

아시다시피 3번 방법은 코드가 다소 길어진다는 단점이 있다. 그래서 이번에는 또 다른 4번째 방법을 하나 더 소개하려고 한다. 바로 PhoneApplicationService에서 제공되는 State 컬렉션을 이용하는 것이다.

```
// ======= MainPage.xaml.cs =======
using System.ComponentModel;
// ……[생략]……
using PPTShow;

namespace ShowController
{
    public partial class MainPage : PhoneApplicationPage, INotifyPropertyChanged
    {
        // ……[생략]……
        void SnapshotCompleted(object sender, DownloadStringCompletedEventArgs e)
        {
            // ……[생략]……
            PPTDocument document = serializer.ReadObject(ms) as PPTDocument;

            PhoneApplicationService.Current.State["Document"] = document;

             Uri uri = new Uri(string.Format("/PPTController.xaml?ip={0}&port={1}",
                                    this.IPSelected, this.Port), UriKind.Relative);
             this.NavigationService.Navigate(uri);
             // ……[생략]……
```

```
        }
    }
}
```

PhoneApplicationService.Current의 State 컬렉션은 윈도우 폰 운영체제가 자동으로 관리해 주는 저장소다. 여기에 저장된 값은 앱이 비활성화될 때 자동으로 디스크에 저장되기 때문에 '격리 저장소'와 동일한 영속성(persistence)을 보장받을 수 있다.

이것으로 MainPage가 하는 역할은 모두 끝났다.

20.6.3.2 제어 페이지 – PPTController.xaml

ShowController 프로젝트에 PPTController.xaml이라는 이름으로 새 페이지를 하나 추가한다. 그리고 다음과 같이 Panorama 요소 하나와 ListBox를 추가해 UI를 구성한다.

```
// ======= PPTController.xaml =======
<phone:PhoneApplicationPage
    x:Class="ShowController.PPTController"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
    xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

    xmlns:controls="clr-
namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone.Controls"

    FontFamily="{StaticResource PhoneFontFamilyNormal}"
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    SupportedOrientations="Portrait" Orientation="Portrait"
    mc:Ignorable="d" d:DesignHeight="768" d:DesignWidth="480"
    shell:SystemTray.IsVisible="True">

    <Grid x:Name="LayoutRoot" Background="Transparent">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
            <TextBlock     x:Name="ApplicationTitle"     Text="Show     Controller"
Style="{StaticResource PhoneTextNormalStyle}"/>
        </StackPanel>

        <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
            <Grid.RowDefinitions>
                <RowDefinition></RowDefinition>
                <RowDefinition Height="5"></RowDefinition>
                <RowDefinition Height="100"></RowDefinition>
            </Grid.RowDefinitions>

            <controls:Panorama x:Name="panorama">
```

```
            </controls:Panorama>

            <ListBox Grid.Row="2" x:Name="imageList"
                     ScrollViewer.HorizontalScrollBarVisibility="Auto"
                     ScrollViewer.VerticalScrollBarVisibility="Disabled">

                <ListBox.ItemsPanel>
                    <ItemsPanelTemplate>
                        <StackPanel Orientation="Horizontal" />
                    </ItemsPanelTemplate>
                </ListBox.ItemsPanel>

            </ListBox>

        </Grid>
    </Grid>

</phone:PhoneApplicationPage>
```

PPTController 타입은 페이지가 로드될 때 자동으로 실행되는 OnNavigatedTo 메서드를 재정의해 초기화 작업을 시작한다.

```
// ======= PPTController.xaml.cs =======
using System;
using System.IO;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using Microsoft.Phone;
using Microsoft.Phone.Controls;
using Microsoft.Phone.Shell;
using PPTShow;

namespace ShowController
{
    public partial class PPTController : PhoneApplicationPage
    {
        string _baseUrl;

        public PPTController()
        {
            InitializeComponent();
        }

        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
            PPTDocument document = null;

            // MainPage에서 저장해 둔 PPTDocument 객체를 받아온다.
            if (PhoneApplicationService.Current.State.ContainsKey("Document") ==
                                                                        true)
            {
```

```
                document = PhoneApplicationService.Current.State["Document"]
                                                                    as PPTDocument;
            }

            if (document == null)
            {
                return;
            }

            // 쿼리 문자열로 전달된 IP주소와 포트 번호도 함께 받아온다.
            string ipAddress;
            string port;
            NavigationContext.QueryString.TryGetValue("ip", out ipAddress);
            NavigationContext.QueryString.TryGetValue("port", out port);

            if (string.IsNullOrEmpty(ipAddress) == true ||
                string.IsNullOrEmpty(port) == true)
            {
                return;
            }

            _baseUrl = string.Format("http://{0}:{1}", ipAddress, port);

            base.OnNavigatedTo(e);
        }

    }
}
```

PhoneApplicationService.Current.State["Document"] 컬렉션에 담긴 값은 PPTDocument이므로 이를 이용해 곧바로 Panorama 컨트롤과 하단의 ListBox를 초기화할 수 있다. 그 중에서 우선 코딩하기 쉬운 ListBox 먼저 채워보자.

```
// ======= PPTController.xaml.cs =======
using System;
// ……[생략]……

namespace ShowController
{
    public partial class PPTController : PhoneApplicationPage
    {
        string _baseUrl;
        // ……[생략]……

        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
            // ……[생략]……

            LoadDocument(document);

            base.OnNavigatedTo(e);
        }

        private void LoadDocument(PPTDocument document)
```

```
        {
            foreach (var item in document.List)
            {
                byte[] jpegContents = Convert.FromBase64String(item.ImageAsText);
                Image img = new Image();

                MemoryStream ms = new MemoryStream(jpegContents);
                WriteableBitmap bmp = PictureDecoder.DecodeJpeg(ms, 100, 100);
                img.Source = bmp;

                Border border = new Border();
                border.Child = img;

                imageList.Items.Add(border);
            }
        }
    }
}
```

PPTDocument의 List 속성에는 PPTPage 객체들이 있고, PPTPage의 ImageAsText에는 PPT 슬라이드의 이미지 데이터가 Base64 형식으로 인코딩된 채로 저장돼 있다. 따라서 이를 Base64 디코딩해서 바이트 배열로 변환한 다음 MemoryStream과 PictureDecoder 타입을 이용해 WritableBitmap 타입으로 변환할 수 있다. 일단 Bitmap 타입으로 변환되면 Image 요소의 Source에 지정해 그림으로 출력할 수 있다.

결국 위의 코드를 실행하면 ListBox에 들어갈 항목 하나당 다음과 같은 XAML이 구성되는 것과 같다.

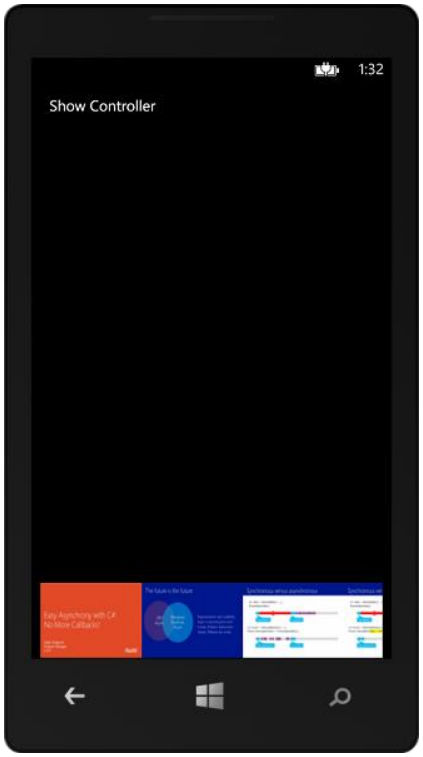
```
<Border>
    <Border.Child>
        <Image Source="……WritableBitmap……" />
    </Border.Child>
</Boder>
```

여기까지 완성하고 실행해 보자.

1. PPTShow에서 PPT 파일을 선택한다.
2. ShowController 윈도우 폰 앱을 실행하고 연결 버튼을 누른다.

그럼 다음과 같이 PPTController 페이지 하단에 PPT 슬라이드 그림 목록이 출력되는 것을 볼 수 있다.

그림 21.15 PPTController 페이지의 PPT 슬라이드 그림 목록



에뮬레이터에서 테스트하는 경우 마우스를 이용해 이미지 목록을 좌/우로 스크롤할 수 있고, 실제 윈도우 폰이라면 터치 동작을 이용해 이미지를 좌/우로 밀어낼 수 있다.

원래 우리는 이 상태에서 이미지로 출력된 슬라이드를 선택하면 PPTShow에서 진행 중인 쇼의 슬라이드도 변경하도록 구현하는 것이 목표였다. 이를 위해서는 다음과 같은 선행 과제가 필요하다.

- 쇼의 슬라이드를 변경하기 위해서는 당연히 PPT 쇼가 이미 시작된 상태여야 한다.
- 하단의 ListBox에서 현재 선택된 이미지가 어떤 것인지 구분될 필요가 있다. 이를 위해 선택된 슬라이드를 구분하기 쉽게 Border의 선을 노란색으로 굵게 표시한다.
- ListBox 내의 Image가 선택되는 이벤트 처리기를 작성한다. 그리고 해당 이벤트 처리기 내에서는 현재 선택된 Image 요소가 몇 번째 슬라이드를 표현하는 것인지 정보를 조회할 수 있어야 한다.

단계적으로 하나씩 구현해 보자. 우선 PPTShow에 ShowController가 연결됐으면 파워포인트가 쇼를 시작하게 만드는 것이 바람직하다. 따라서 LoadDocument의 마지막 단계에서 PPTShow에 쇼를 시작하라는 HTTP 요청을 전송한다.

```
// ======= PPTController.xaml.cs =======
using System;
// ……[생략]……

namespace ShowController
{
```

```
    public partial class PPTController : PhoneApplicationPage
    {
        // ……[생략]……

        private void LoadDocument(PPTDocument document)
        {
            foreach (var item in document.List)
            {
                // ……[생략]……
            }

            StartShow();
        }

        void StartShow()
        {
            string url = string.Format("{0}/startShow", this._baseUrl);
            CallUrl(url, null);
        }

        void CallUrl(string url, DownloadStringCompletedEventHandler handler)
        {
            WebClient wc = new WebClient();
            wc.Headers[HttpRequestHeader.IfModifiedSince] =
                                                    DateTime.UtcNow.ToString();

            if (handler != null)
            {
                wc.DownloadStringCompleted += handler;
            }

            wc.DownloadStringAsync(new Uri(url));
        }
    }
}
```

코드는 크게 특별한 것이 없으므로 이 상태에서 빌드한 후 테스트해보자. 이번에는 연결 버튼을 누르자마자 파워포인트의 쇼가 시작되는 것을 확인할 수 있다.

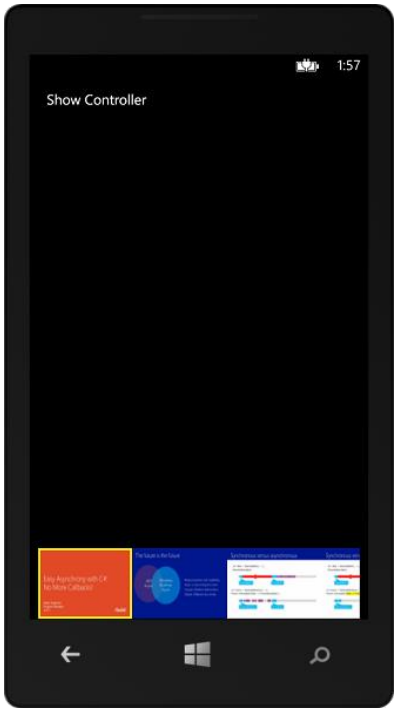
다음으로 선택된 슬라이드의 Border를 구분하기 쉽게 노란색에 굵기를 3으로 만들게끔 코드를 추가한다.

```
// ======= PPTController.xaml.cs =======
using System;
// ……[생략]……

namespace ShowController
{
    public partial class PPTController : PhoneApplicationPage
    {
        // ……[생략]……
        Border oldBorder = null;

        private void LoadDocument(PPTDocument document)
```

```
        {
            int slideIndex = 1;
            foreach (var item in document.List)
            {
                // ……[생략]……

                border.Child = img;
                border.BorderBrush = new SolidColorBrush(Colors.Yellow);

                if (slideIndex == 1)
                {
                    // 처음 연결한 시점에는 첫 슬라이드를 선택 표시
                    border.BorderThickness = new Thickness(3);
                    oldBorder = border;
                }

                // ……[생략]……

                slideIndex++;
            }

            StartShow();
        }
    }
}
```

처음 PPTShow에 연결해서 실행되는 LoadDocument 단계에서는 무조건 첫 번째 슬라이드가 선택된 것으로 가정하고 해당 슬라이드의 Border에 대해서만 굵기를 3으로 지정한다. 여기까지 진행하고 다시 실행하면 그림 21.16과 같이 첫 번째 슬라이드가 선택됐음을 쉽게 파악할 수 있는 외곽선을 볼 수 있다.

그림 21.16 첫 번째 슬라이드에 지정된 노란색 외곽선



이제 목록의 이미지를 선택한 경우 파워포인트의 쇼에서 해당 이미지의 슬라이드로 교체되는 기능을 구현할 차례다. 이를 위해 Image 요소에 Tap 이벤트를 걸어주면 사용자가 하단의 이미지 영역을 터치하거나 마우스로 선택했을 때 실행되는 코드를 정의할 수 있다.

```
// ======= PPTController.xaml.cs =======
using System;
// ……[생략]……

namespace ShowController
{
    public partial class PPTController : PhoneApplicationPage
    {
        string _baseUrl;
        // ……[생략]……

        private void LoadDocument(PPTDocument document)
        {
            int slideIndex = 1;
            foreach (var item in document.List)
            {
                byte[] jpegContents = Convert.FromBase64String(item.ImageAsText);
                Image img = new Image();

                img.Tap += img_Tap;

                // ……[생략]……
            }
```

```
            StartShow();
        }

        void img_Tap(object sender, GestureEventArgs e)
        {
            // 선택된 이미지를 PPT Show에서 선택되도록 HTTP 요청 처리
        }
    }
}
```

여기서 문제가 하나 있다. 바로 ListBox 목록에서 선택된 이미지의 슬라이드 번호를 img_Tap 이벤트 처리기에서 어떻게 구하느냐다. 즉, 선택된 객체의 문맥(context) 데이터가 필요한 상황인데, XAML 요소에서는 이런 경우 사용자가 임의의 데이터를 보관할 수 있게 Tag 속성을 제공하므로 이를 이용해 슬라이드 번호를 Image 요소와 연결할 수 있다.

```
// ======= SlideItemData.cs =======
using System;

namespace ShowController
{
    public class SlideItemData
    {
        int _slideIndex;
        public int SlideIndex
        {
            get { return _slideIndex; }
            set { _slideIndex = value; }
        }
    }
}
```

```
// ======= PPTController.xaml.cs =======
using System;
// ......[생략]......

namespace ShowController
{
    public partial class PPTController : PhoneApplicationPage
    {
        string _baseUrl;
        // ......[생략]......

        private void LoadDocument(PPTDocument document)
        {
            int slideIndex = 1;
            foreach (var item in document.List)
            {
                byte[] jpegContents = Convert.FromBase64String(item.ImageAsText);
                Image img = new Image();
                img.Tap += img_Tap;
```

```
                // 슬라이드 번호를 Image 요소의 Tag 속성에 연결
                SlideItemData tagData = new SlideItemData();
                tagData.SlideIndex = slideIndex;
                img.Tag = tagData;

                // ……[생략]……

                slideIndex++;
            }
            StartShow();
        }

        void img_Tap(object sender, GestureEventArgs e)
        {
            // 선택된 이미지의 슬라이드 번호를 Tag로부터 구한다.
            SlideItemData tagData = (sender as Image).Tag as SlideItemData;
            int slideIndex = tagData.SlideIndex;

            SetSlide(slideIndex);
            SetSelectedBorder(slideIndex);
        }

        // 슬라이드 번호로 쇼에 보일 슬라이드를 결정
        void SetSlide(int number)
        {
            string url = string.Format("{0}/setSlide/{1}", this._baseUrl, number);
            CallUrl(url, null);
        }

        // ListBox에서 선택된 슬라이드에 해당하는 Image만 Border를 3으로 설정
        void SetSelectedBorder(int slideIndex)
        {
            if (oldBorder != null)
            {
                oldBorder.BorderThickness = new Thickness(0);
            }

            Border border = imageList.Items[slideIndex - 1] as Border;
            border.BorderThickness = new Thickness(3);

            oldBorder = border;

            imageList.ScrollIntoView(border);
        }

        // ……[생략]……
    }
}
```

여기까지 구현한 결과를 테스트해보자. 이제는 마우스 또는 터치를 이용해 하단의 이미지를 선택할 수 있고 그때마다 PPTShow는 파워포인트의 쇼에서 보일 슬라이드를 변경한다.

마지막으로 Panorama에 보일 슬라이드와 메모 목록을 만들어 보자. 우선, LoadDocument에서

ListBox의 이미지 목록을 채웠던 것과 비슷하게 Panorama 컨트롤의 내용도 구성할 수 있다.

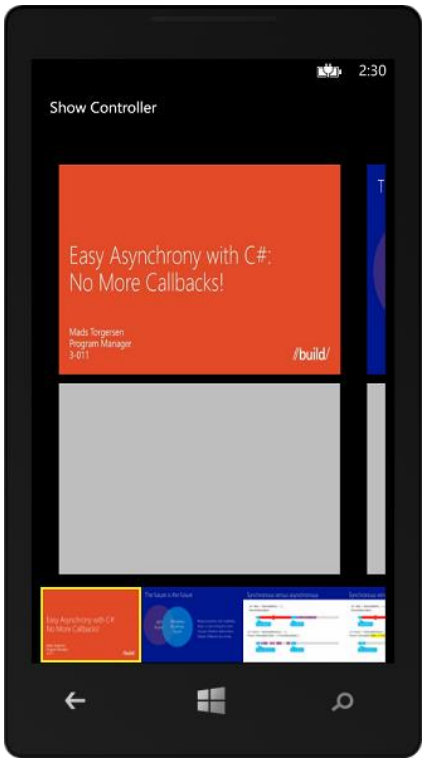
```
// ======= PPTController.xaml.cs =======
using System;
// ……[생략]……

namespace ShowController
{
    public partial class PPTController : PhoneApplicationPage
    {
        string _baseUrl;

        private void LoadDocument(PPTDocument document)
        {
            int slideIndex = 1;
            foreach (var item in document.List)
            {
                // ……[생략: ListBox의 Image/Border 구성]……

                // 파노라마 컨트롤 목록 구성
                PanoramaItem panItem = new PanoramaItem();
                panItem.Tag = tagData;
                Grid grid = new Grid();

                grid.RowDefinitions.Add(new RowDefinition());
                grid.RowDefinitions.Add(new RowDefinition());

                ms.Position = 0;
                bmp = PictureDecoder.DecodeJpeg(ms, document.Width,
                                                document.Height);

                img = new Image();
                img.Source = bmp;
                Grid.SetRow(img, 0);

                TextBox txt = new TextBox();
                txt.Text = item.Note;
                txt.IsReadOnly = true;
                Grid.SetRow(txt, 1);

                grid.Children.Add(img);
                grid.Children.Add(txt);

                panItem.Content = grid;

                panorama.Items.Add(panItem);

                slideIndex++;
            }
            StartShow();
        }

        // ……[생략]……
    }
}
```

위의 코드를 실행하면 PanoramaItem 하나당 다음과 같은 XAML이 구성되는 것과 같다.

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Image Source="……WritableBitmap……" Grid.Row="0" />
    <TextBox IsReadOnly="True" Grid.Row="1" Text="……item.Note……" />
</Grid>
```

여기까지 코드를 추가하고 변경된 사항을 확인하기 위해 앱을 실행해 보자.

그림 21.17 Panorama 컨트롤의 동작 화면



새롭게 추가된 중앙의 파노라마 컨트롤 영역을 마우스나 터치를 이용해 좌/우로 밀어보자. 그럼 그에 해당하는 이미지가 부드럽게 스크롤되면서 나타난다. 단지 아직까지는 이렇게 변경해도 PPTShow의 파워포인트 쇼에는 반영되지 않는다. 왜냐하면 Panorama 컨트롤의 선택이 바뀐 경우 이를 처리하는 코드를 아직 작성하지 않았기 때문이다. 따라서 이 작업만 마무리하면 ShowController의 구현이 모두 완료된다.

```
// ======= PPTController.xaml =======
<phone:PhoneApplicationPage
```

```
……[생략]……
    shell:SystemTray.IsVisible="True">
……[생략]……

          <controls:Panorama x:Name="panorama"
                             SelectionChanged="panorama_SelectionChanged">
          </controls:Panorama>

……[생략]……
</phone:PhoneApplicationPage>
```

```
// ======= PPTController.xaml.cs =======
using System;
// ……[생략]……

namespace ShowController
{
    public partial class PPTController : PhoneApplicationPage
    {

        // ……[생략]……

        private void panorama_SelectionChanged(object sender,
                                               SelectionChangedEventArgs e)
        {
            if (e.AddedItems.Count == 0)
            {
                return;
            }

            SlideItemData tagData = (e.AddedItems[0] as PanoramaItem).Tag
                                                        as SlideItemData;

            int slideIndex = tagData.SlideIndex;
            SetSlide(slideIndex);
            SetSelectedBorder(slideIndex);
        }

        // ……[생략]……
    }
}
```

마지막으로 완성된 앱을 테스트해보고 정상적으로 동작하는지 확인한다.

21.7 개선 사항

필자의 경험상 응용 프로그램 개발은 제품의 시작일 뿐이다. 만들어진 OfficePresenter 프로그램을 배포하면 각양각색의 사용자에게서 버그 리포트 및 개선 사항들이 올라올 것이다. 어쩌면 개발하는 도중 테스트를 진행하면서 내부적으로 이런 문제점들이 이미 알려졌을 수도 있다.

앞에서도 언급했듯이 이 모든 기능을 완벽하게 구현하고 응용 프로그램을 배포할지, 아니면 차후 버전에서 개선할 것으로 결정하고 진행할지는 기획팀과의 협의하에 적절하게 판단해야 한다. 일례로 특정 기능이 확률적으로 1% 내의 사용자에게만 발생하는 특별한 문제라면 제품의 릴리즈 시점을 미뤄서까지 구현해야 하는가는 생각해 볼 문제다.

당연히 이번 장에서 만든 프로그램도 몇 가지 개선 사항이 필요하지만 지면상 모든 내용을 다루기는 불가능하므로 나머지 부분은 독자의 몫으로 남긴다. 아래는 필자가 뽑아본 OfficePresenter 프로그램의 문제점이다.

- **[문제점 1]** 이 프로그램으로 대규모 세미나에서 프레젠테이션을 진행 중인데 참가자 가운데 누군가가 PPTShow의 IP를 알아내고는 파워포인트를 임의로 제어하는 상황이 벌어졌다.
  [해결 방안] PPTShow의 화면에서 간단하게 "암호"를 입력받을 수 있게 만든다. ShowController에서 PPTShow로 접속할 때 반드시 해당 암호를 입력해야만 제어할 수 있게 만든다.

- **[문제점 2]** 슬라이드를 이미지로 변경한 크기가 현재는 Width=480, Height=358로 고정돼 있는데, 향후 지원될 다양한 크기의 윈도우 폰 화면에서는 바뀔 필요가 있다.
  [해결 방안] ShowController 측에서 이미지의 크기를 getSnapshot을 요청할 때 선택하게 한다.

- **[문제점 3]** 슬라이드에 "애니메이션"이 포함된 경우 이를 제어하는 방법이 구현돼 있지 않다.
  [해결 방안] PPTShow에서 getSnapshot으로 전달하는 PPTPage 정보에 각 슬라이드에 포함된 애니메이션의 수를 구해서 함께 전달해야 한다. ShowController에서는 애니메이션 수에 따라 사용자가 더블-탭을 누르는 동작을 하면 다음 애니메이션이 나타나도록 제어하는 기능을 추가해야 한다.

- **[문제점 4]** 예쁘지 않은 사용자 UI
  [해결 방안] Blend를 쓸 줄 아는 디자이너를 구할 수 있다면 가장 좋겠지만 그것이 불가능하다면 응용 프로그램의 외양이라도 그려줄 수 있는 디자이너가 필요하다. 혹은 아이콘이라도 말이다.

이제 모든 과정이 끝났다. 그럼 다음 단계로는 뭘 해야 할까? 이 책을 덮고 또 다른 주제를 공부해야 할지, 아니면 자신만의 응용 프로그램을 구상해서 만들어갈지는 여러분의 몫이다.

부록

A. C# 12 언어 명세

ECMA 표준에 제출된 PDF 문서는 C# 5.0을 기준으로 하며 다음 URL에서 확인할 수 있다.

- ECMA-334: **https://www.ecma-international.org/publications/standards/Ecma-334.htm**

C# 6.0은 draft 상태로 다음 페이지를 통해 제공되며 좌측 하단의 "Download PDF" 메뉴를 선택하면 PDF 파일로 다운로드할 수 있다.

- C# 6.0 draft: **https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/**

이후의 업데이트는 마이크로소프트의 온라인 문서를 통해 최신 C# 정보를 공개하고 있다.

- C# 온라인 문서: **https://learn.microsoft.com/en-us/dotnet/csharp/index**

B. C# 12 연산자와 문장 부호

C# 12의 연산자와 우선 순위에 대한 정보는 아래의 문서에서 최신 정보를 확인할 수 있습니다.

C# operators and expressions (C# reference) **https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/**

표 B.1 연산자와 문장 부호

[ ]	( )	.	!.	::	+	-	*	/	%
&	|	^	!	~	=	<	>	?:	++
--	&&	||	<<	>>	==	!=	<=	>=	+=
-=	*=	/=	%=	&=	|=	^=	<<=	>>=	->
??	??=			=>	Null-	^	..		

			conditional (?. or ?[)	(index)	(range)		

표 B.2 연산자 우선순위

우선순위	연산자 범주	연산자
1	기본	x.y x?.y x?[y] x!.y f(x) a[x] x++ x— x! new typeof checked unchecked default nameof delegate sizeof stackalloc x->y
2	단항	+x -x !x ~x ++x --x ^x (T)x await &x *x true and false
3	범위	x..y (range)

4	switch 식, with 식	switch, with
5	승제	x * y x / y x % y
6	가감	x + y x – y
7	시프트	x << y x >> y x >>> y
8	관계 및 형식 테스트	x < y x > y x <= y x >= y is as
9	비교	x == y x != y
10	비트 논리곱	x & y
11	논리 XOR	x ^ y
12	비트 논리합	x | y
13	조건 논리곱	x && y
14	조건 논리합	x || y
15	Null 결합	x ?? y
16	3항 조건	c ? t : f
17	할당 및 람다 식	x = y x += y x -= y x *= y x /= y x %= y x &= y x |= y x ^= y x <<= y x >>= y x ??= y =>

* 1번부터 가장 우선순위가 높으며, 같은 그룹 내의 연산자는 우선순위가 같다.

C. C# 12 예약어

C# 12의 최신 예약어는 아래의 웹 페이지에서 확인할 수 있다.

C# Keywords **https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/index**

다음은 위의 내용을 그대로 가져온 것이다.

표 C.1 C# 12 키워드 77개

abstract	as	base	bool	break	byte
case	catch	char	checked	class	const
continue	decimal	default	delegate	do	double
else	enum	event	explicit	extern	false
finally	fixed	float	for	foreach	goto
if	implicit	in	int	interface	internal
is	lock	long	namespace	new	null
object	operator	out	override	params	private
protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string
struct	switch	this	throw	true	try
typeof	uint	ulong	unchecked	unsafe	ushort
using	virtual	void	volatile	while	

표 C.2 C# 12 문맥 키워드(Contextual Keywords) 46개

add	and	alias	ascending	args	async
await	by	descending	dynamic	equals	file
from	get	global	group	init	into
join	let	managed (function pointer calling convention)	nameof	nint	not
notnull	nuint	on	or	orderby	partial (type)
partial	record	remove	required	scoped	select

(method)					
set	unmanaged (function pointer calling convention)	unmanaged (generic type constraint)	value	var	when (filter condition)
where (generic type constraint)	where (query clause)	with	yield		

D. ASCII 코드

10진수	ASCII	10진수	ASCII	10진수	ASCII	10진수	ASCII
0	NULL	**32**	SP	**64**	@	**96**	`
1	SOH	**33**	!	**65**	A	**97**	a
2	STX	**34**	"	**66**	B	**98**	b
3	ETX	**35**	#	**67**	C	**99**	c
4	EOT	**36**	$	**68**	D	**100**	d
5	ENQ	**37**	%	**69**	E	**101**	e
6	ACK	**38**	&	**70**	F	**102**	f
7	BEL	**39**	'	**71**	G	**103**	g
8	BS	**40**	(	**72**	H	**104**	h
9	HT	**41**	)	**73**	I	**105**	i
10	LF	**42**	*	**74**	J	**106**	j
11	VT	**43**	+	**75**	K	**107**	k
12	FF	**44**	,	**76**	L	**108**	l
13	CR	**45**	-	**77**	M	**109**	m
14	SO	**46**	.	**78**	N	**110**	n
15	SI	**47**	/	**79**	O	**111**	o
16	DLE	**48**	0	**80**	P	**112**	p
17	DC1	**49**	1	**81**	Q	**113**	q
18	DC2	**50**	2	**82**	R	**114**	r
19	DC3	**51**	3	**83**	S	**115**	s
20	DC4	**52**	4	**84**	T	**116**	t
21	NAK	**53**	5	**85**	U	**117**	u
22	SYN	**54**	6	**86**	V	**118**	v
23	ETB	**55**	7	**87**	W	**119**	w
24	CAN	**56**	8	**88**	X	**120**	x

25	EM	**57**	9	**89**	Y	**121**	y
26	SUB	**58**	:	**90**	Z	**122**	z
27	ESC	**59**	;	**91**	[	**123**	{
28	FS	**60**	<	**92**	₩	**124**	|
29	GS	**61**	=	**93**	]	**125**	}
30	RS	**62**	>	**94**	^	**126**	~
31	US	**63**	?	**95**	_	**127**	DEL

E. 프로그래밍 기본 지식

프로그래밍 언어를 배우는 이유는 프로그램을 만들기 위해서다. 하지만 단순히 언어의 문법만을 배워서는 활용 사례가 극히 제한된다. 이를테면, 언어 자체는 다른 프로그램과의 통신을 지원하지 않기에 통신을 하려면 운영체제에서 제공하는 통신 수단을 알아야 할 필요가 있으므로 자연스럽게 운영체제와 관련된 지식 또한 필요하다. 따라서 여기서는 운영체제를 비롯해 프로그래밍할 때 익숙해져야 할 기본적인 용어를 알아본다.

E.1 하드웨어 관련 용어

E.1.1 중앙 처리 장치(CPU)

CPU(Central Processing Unit)는 고유한 기계어를 가지며, 그것을 해석할 수 있는 장치다. 일반적으로 윈도우 운영체제가 설치된 개인용 컴퓨터에는 대부분 인텔과 AMD에서 만든 CPU가 장착된다. 반면 최근 급증하는 모바일 기기에는 ARM이라는 회사에서 만든 CPU와 호환되는 제품들이 주로 장착되고 있다. 여러분이 C#으로 만드는 프로그램들은 기본적으로 인텔 CPU와 호환되는 컴퓨터에서 실행되지만, ARM 계열에서 실행하는 것도 가능하다.

E.1.2 레지스터(Register)

CPU 내에 존재하는 기억장소다. 한 개의 레지스터에 담을 수 있는 데이터의 크기는 우리가 일반적으로 부르는 16비트/32비트/64비트 CPU라는 명칭에서 쉽게 알 수 있다. 즉, 64비트 CPU는 레지스터 하나에 담을 수 있는 비트의 수가 64개임을 의미한다. 인텔/AMD CPU에서는 하위 호환성을 확보하기 위해 16비트 응용 프로그램은 32비트/64비트 CPU에서도 실행할 수 있고, 32비트 역시 64비트 CPU에서 실행 가능하다.

CPU가 다르면 레지스터의 종류, 수, 이름도 바뀐다. 특히, 인텔 호환 CPU와 ARM 계열의 CPU는 완전히 상이한 레지스터를 가지고 있다. 지원되는 명령어 집합과 레지스터의 차이로 인해 인

텔 호환 CPU에서 실행되는 응용 프로그램은 ARM 계열의 CPU에서 실행되지 않는다.

여러분이 만들게 될 C# 프로그램은 디스크 상에 파일로 존재하다가 실행되는 시점에 메모리로 올라오고, 다시 메모리의 내용이 레지스터로 옮겨져서 해석된다.

E.1.3 x86, x64

인텔의 초기 CPU는 80286, 80386, 80486, 80586(펜티엄)과 같은 식으로 이름이 매겨졌고, 가운데 숫자만 변경해 80x86과 같이 표현할 수 있다. 이를 줄여서 x86이라는 이름이 붙게 됐으며, 대체로 인텔 CPU와 호환되는 제품을 일컬어 x86이라고 한다. 흔히 x86 서버라고 하면 인텔/AMD CPU가 탑재된 서버를 의미한다.

그런데 32비트/64비트를 구분할 때도 x86이라는 용어를 사용한다. 즉, x86은 32비트를, x64는 64비트를 의미하기도 한다.

C#을 이용해 프로그램을 만들 때 EXE/DLL 파일을 x86용(32비트)으로 생성할지, x64용(64비트)으로 생성할지 결정할 수 있다.

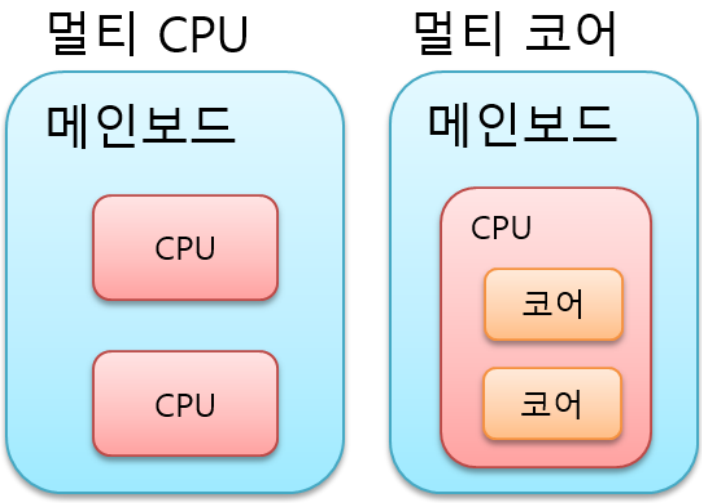
E.1.4 멀티 CPU와 멀티 코어

초기 개인용 컴퓨터 시장에서는 컴퓨터 한 대에 한 개의 CPU만 장착했지만, 점차 x86 시스템이 서버용 컴퓨터로 사용되면서 2개 이상의 CPU가 장착되는 멀티 CPU를 지원하기 시작했다. 최근에는 CPU의 집적도가 향상되면서 기존의 CPU 자원을 하나의 작은 코어로 만들 수 있게 됐다. 이로써 CPU 한 개에 여러 개의 코어를 집적한 멀티 코어 제품이 등장한다. 우리가 흔히 말하는 듀얼(dual) 코어는 한 개의 CPU에 두 개의 코어를 내장한 것으로, 이는 기존의 CPU 두 개가 수행했던 일을 하나의 CPU에서 할 수 있게 됐음을 의미한다.

참고!	실제로 초기 인텔 제품의 경우 한 개의 CPU 다이(Die)에 두 개의 물리 CPU를 집적시킨 적이 있다.

다음은 멀티 CPU와 멀티 코어의 차이점을 보여준다.

그림 E.1 멀티 CPU와 멀티 코어



최근에는 서버용 컴퓨터에 16코어까지 탑재된 CPU가 등장하고 있으며, 이 제품을 두 개 장착하면 하나의 컴퓨터에 32개의 코어가 탑재된 시스템이 만들어질 수 있다.

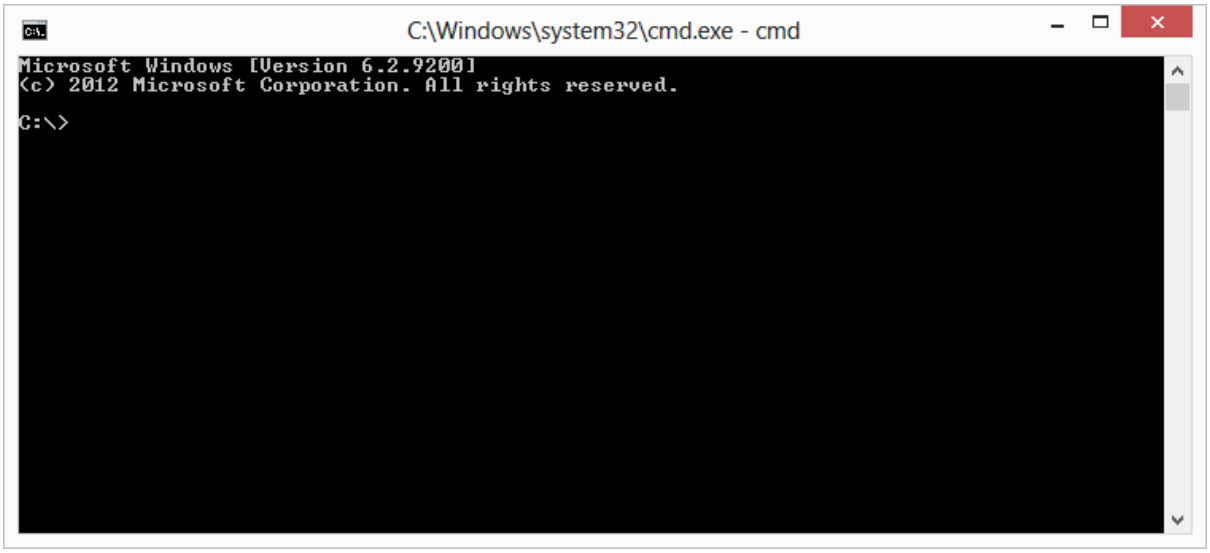
CPU/코어가 많으면 동시에 실행할 수 있는 명령어가 증가하므로 전반적으로 시스템 성능이 향상된다.

E.2 운영체제 관련 용어

E.2.1 도스(DOS)

개인용 컴퓨터 시장에서 의미가 있었던 최초의 운영체제는 마이크로소프트에서 만든 MS-DOS(Microsoft Disk Operating System)다. 이것은 16비트 운영체제로 인텔의 8086/8088 CPU에 적합하게 설계됐다. DOS 운영체제는 문자 방식 사용자 인터페이스(CUI: Character User Interface)로도 유명하다. 지금도 DOS의 흔적을 만날 수 있는데, 윈도우의 "명령 프롬프트(Command Prompt)"가 바로 그것이다. 즉, 과거의 DOS 운영체제는 아래의 그림이 모니터 화면 전체를 차지했었다고 보면 된다.

그림 E.2 명령 프롬프트

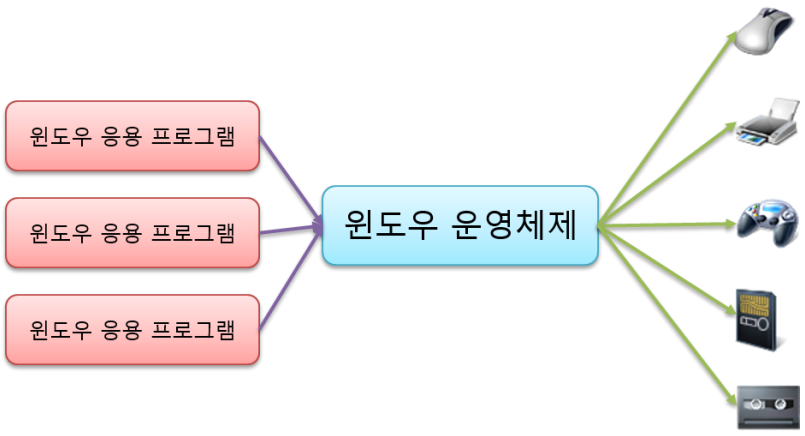


윈도우에서 처음 명령 프롬프트를 지원했을 때는 기존 DOS 프로그램과의 호환을 위해서였으나 차츰 유닉스의 셸(Shell)처럼 윈도우에서도 셸의 역할을 담당하게 된다. C#에서도 응용 프로그램을 개발할 때 명령 프롬프트를 이용해 실행하는 프로그램을 만들 수 있다. 이런 응용 프로그램 유형을 "콘솔 응용 프로그램(Console Application)"이라 한다.

E.2.2 윈도우 운영체제

마이크로소프트가 MS-DOS에 이어 출시한 그래픽 기반의 사용자 인터페이스(GUI: Graphic User Interface)를 제공하는 운영체제다. DOS에 비해 화려해진 UI와 함께 윈도우는 컴퓨터에 연결되는 모든 장치를 추상화해서 프로그래머에게 공통된 소프트웨어 개발 방식을 제공하는 것이 주요 목표였다.

그림 E.3 각종 장치를 추상화한 윈도우 운영체제



개발자는 윈도우에서 제공되는 SDK(Software Development Kit)를 이용하면 윈도우 운영체제가 설치된 어떤 컴퓨터에서든 동일한 외형과 기능을 지닌 응용 프로그램을 만들 수 있다.

윈도우는 대표적으로 개인용과 서버용으로 두 가지로 나눠서 출시되고 있다. 개인용을 서버와 대비시켜 클라이언트용이라고도 부르며, 윈도우 95, 윈도우 98, 윈도우 ME를 거쳐 윈도우 2000 프로페셔널 버전이 나왔다. 서버용으로는 별도로 윈도우 NT 3.x, 윈도우 NT 4를 거쳐 윈도우 2000 서버 버전이 나온다. 이처럼 윈도우 2000 버전을 시작으로 클라이언트 운영체제가 서버 버전의 커널을 공유하기 시작한다. 이후 서버와 클라이언트 제품이 점차로 비슷한 시기와 사용자 인터페이스를 탑재해서 출시된다. 윈도우 XP와 윈도우 서버 2003, 윈도우 비스타와 윈도우 서버 2008, 윈도우 7과 윈도우 서버 2008 R2, 윈도우 8과 윈도우 서버 2012로 쌍을 이뤄 각각 클라이언트와 서버 버전이 출시되고, 최근에는 윈도우 10과 윈도우 서버 2016까지 나왔다. 사실상 윈도우 비스타 이후부터는 서버와 클라이언트 버전이 완전히 동일한 소스코드를 공유하고 있다.

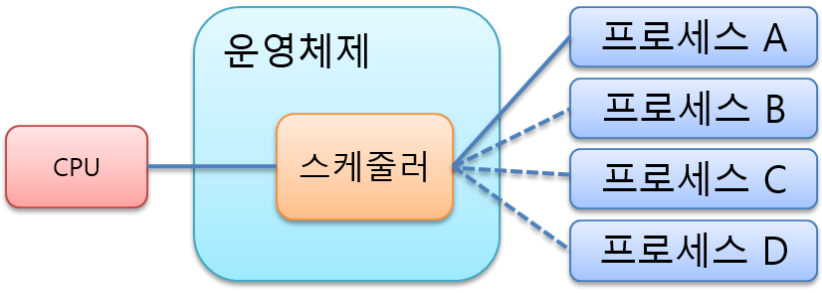
윈도우 7까지는 인텔 호환 CPU만을 지원했으나 윈도우 8부터 ARM CPU를 지원하기 시작했다.

E.2.3 멀티 태스킹/ 다중 프로세스

DOS 운영체제에서는 하나의 실행 프로그램이 화면 전체를 차지했다. 따라서 원칙적으로는 한 번에 하나의 프로그램만 실행됐지만, 윈도우 운영체제로 오면서 다중 프로세스(Multiple processes) 실행이 가능하게끔 바뀌었다. 즉, 사용자는 화면에 여러 개의 프로그램을 띄울 수 있었고 그 프로그램들은 "프로세스(Process)"라는 개별 단위로 실행됐다.

초기 개인용 컴퓨터에는 CPU가 하나였다는 점을 상기하자. CPU는 윈도우에 실행된 응용 프로그램을 각각 짧은 시간 동안 실행을 전환시키면서 마치 여러 개의 프로세스가 동작하는 것처럼 구현한다. 예를 들어, '그림 E.4'를 보면 현재 A 프로세스에 CPU 자원이 할당돼 있다. 운영체제의 스케줄러는 A 프로세스에 일정 시간 동안 CPU 자원을 할당하고 이후 B 프로세스로 CPU를 할당하고 C, D 프로세스까지 이런 식으로 실행하면서 반복한다.

그림 E.4 멀티 태스킹



이때 운영체제는 프로세스마다 문맥(Context)이라는 정보를 유지한다. 그리고 스케줄러가 현재 실행 중인 프로세스를 멈추고 다른 프로세스를 실행하는 시점에 문맥 전환(Context Switching)이 발생한다. A 프로세스에서 B 프로세스로 문맥 전환이 발생할 때 스케줄러는 A 프로세스의 실행 정보를 A 프로세스 문맥에 보관하고, B 프로세스의 문맥 정보를 CPU로 불러와 실행한다. 그러다

다시 A 프로세스를 실행하도록 스케줄링되면 운영체제는 기존에 보관된 A 프로세스의 문맥을 CPU로 가져와 이전에 마지막으로 실행했던 명령어 지점부터 다시 실행을 계속한다.

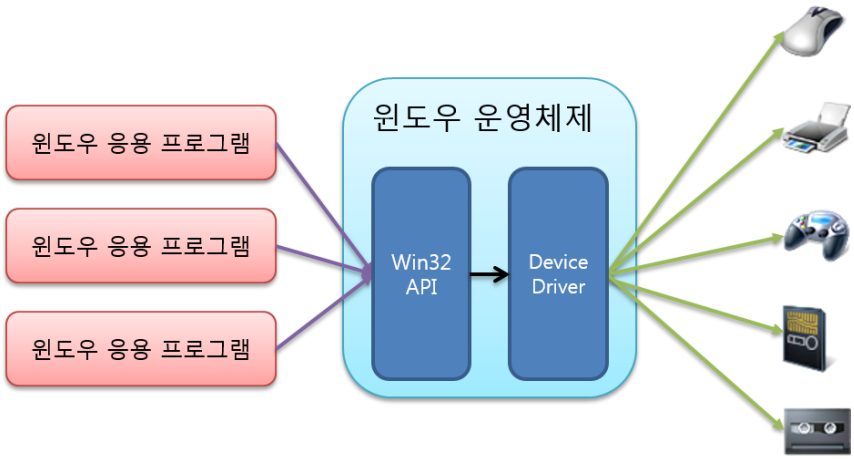
다중 프로세스가 가능하다고 해서 반드시 멀티 태스킹(Multi-tasking)이 되는 것은 아니다. 다중 프로세스는 단지 여러 개의 프로세스가 동시에 운영체제에 의해 로드됐음을 의미할 뿐이다. 만약 운영체제에서 그중 하나만 실행하도록 구현돼 있다면 해당 운영체제는 멀티 태스킹을 지원한다고 볼 수 없다.

E.2.4 Win32 응용 프로그램 인터페이스(API)

윈도우 운영체제에서 동작하는 모든 프로그램은 운영체제 측에서 제공하는 기능을 이용한다. 그 기능을 API(Application Programming Interface)라고 하며, 확장자가 DLL인 파일에서 제공된다.

'그림 E.3'을 좀 더 구체화하면 다음 그림과 같이 표현할 수 있다.

그림 E.5 Win32 API 공통 인터페이스를 응용 프로그램에 제공



보다시피 윈도우 프로그램은 각종 장치를 직접 제어하지 않는다. 윈도우는 연결될 장치에 대해 그 장치의 사용법을 가장 잘 아는 제조사가 장치 드라이버(Device Driver)를 함께 제공하게 한다. 그리고 API를 통해 장치 드라이버의 기능을 노출한다. 그 덕분에 장치 드라이버가 바뀌었다고 해서 윈도우 응용 프로그램이 바뀌지는 않는다. 이 상황을 "프린터"로 설명해 보자. 프로그램에서는 인쇄를 위해 Win32 API를 사용해 인쇄 작업을 수행할 수 있다. 윈도우는 A 제조사에서 만든 프린터와 장치 드라이버가 있다면 그것을 사용해 인쇄 작업을 연결한다. 또는 B 제조사에서 만든 프린터가 있다면 역시 그 장치 드라이버를 이용해 인쇄 작업을 수행함으로써 윈도우 응용 프로그램과 장치 드라이버 사이의 직접적인 의존성을 제거한다.

윈도우 운영체제가 16비트이던 윈도우 3.1에서는 Win16 API라는 이름으로 불리다가 32비트 운영체제인 윈도우 95가 나오면서 Win32 API라고 불리기 시작했다. 현재 64비트 운영체제까지 나와서 내부적으로 Win64 API가 나오긴 했지만 Win32라는 표현이 워낙 널리 퍼져서 윈도우에서 제공

하는 API에 대한 일반적인 호칭으로 "Win32 API"를 주로 사용한다.

처음 Win16 API에서 Win32 API로 넘어올 때는 많은 부분이 변경되어 이전하기가 쉽지 않았지만, Win32 API에서 Win64로 넘어가는 과정은 잘 만들어진 프로그램의 경우 다시 컴파일하는 것만으로도 가능할 정도로 하위 호환성이 잘 지켜진다.

Win32 API를 호출해서 동작하는 응용 프로그램은 일반적으로 모든 종류의 윈도우에서 실행된다. 단지 새로운 운영체제에서는 이전 운영체제에서 제공되지 않는 새로운 API를 제공하고, 서버 운영체제는 클라이언트 운영체제가 제공하는 것보다 더 많은 API를 제공하기도 한다.

C# 응용 프로그램에서는 Win32 API를 직접 호출할 수 있지만 이렇게 만든 프로그램은 리눅스에서 실행할 수 없다. 따라서 특별한 목적이 없다면 사용하지 않기를 권장한다.

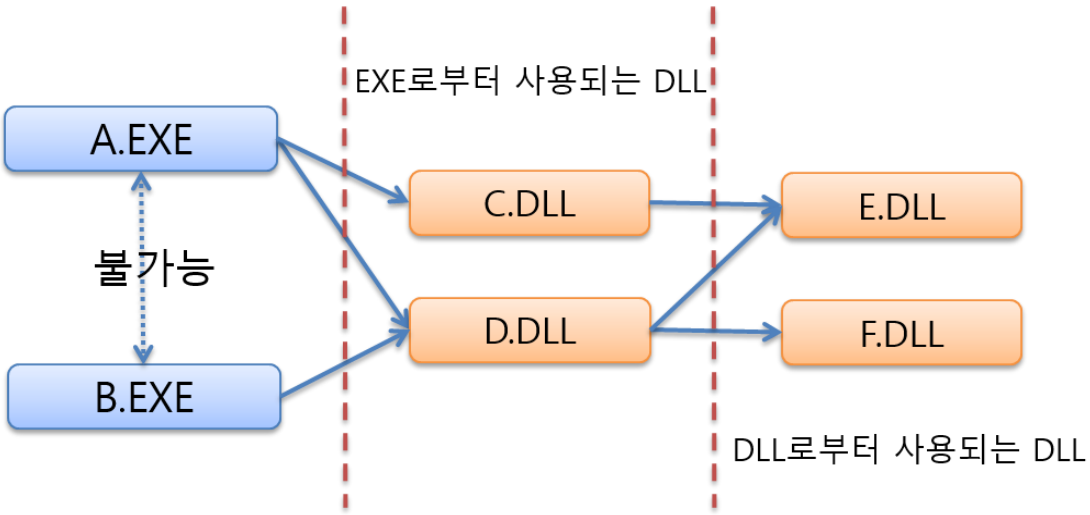
E.2.5 윈도우 응용 프로그램

윈도우 운영체제에서 실행되는 프로그램을 말한다. 모든 윈도우 응용 프로그램은 Win32 API를 사용해 운영체제에서 제공하는 기능들과 협력한다. 윈도우에서는 Win32 API를 동적 링크 라이브러리(DLL: Dynamic Link Library) 파일로 제공한다. 탐색기를 열어 C:\Windows\System32 폴더를 보면 무수히 많은 DLL 파일이 있는 것을 확인할 수 있는데, 대부분 윈도우 운영체제에서 제공하는 기능에 해당한다.

참고!	응용 프로그램 개발자가 직접 사용하는 기본 DLL은 kernel32.dll, user32.dll, gdi32.dll 정도가 있다. Visual C++ 언어를 사용하는 윈도우 프로그래머는 최소한 그 3개의 DLL에서 제공하는 API의 사용법에 익숙해져야 한다.

윈도우에서 사용자가 실행하는 파일은 보통 확장자가 EXE인데, EXE 파일은 DLL 파일과 구조가 동일하다. 다만 DLL은 내부에 구현된 API를 외부에서 참조할 수 있다. 따라서 EXE 파일은 다른 EXE 파일의 기능을 사용할 수 없지만 DLL 파일은 로드해서 사용할 수 있다.

그림 E.6 DLL과 EXE의 참조 관계



이러한 이유로 재사용 가능한 코드는 컴파일해서 DLL 파일에 모아두고 서로 다른 EXE에서 공유해서 사용하는 방식이 일반적이다. 예를 들어, 파일을 암호화/복호화하는 코드를 작성했다고 가정해 보자. 그 코드를 DLL 파일에 넣어두면 A.EXE, B.EXE에서는 그 DLL을 로드해서 암호화/복호화하는 API를 호출해서 사용할 수 있다. 반면 DLL이 아닌 EXE 파일에 넣었다면 그와 같이 재사용하는 것이 불가능하다.

대부분의 윈도우 응용 프로그램은 1개의 EXE 파일과 여러 개의 DLL 파일로 구성된다. 이는 C#으로 프로그램을 만들 때도 마찬가지다.

E.2.6 32비트 응용 프로그램

32비트 CPU의 기계어로 번역된 프로그램을 일컫는다. 32비트 응용 프로그램의 특징은 해당 프로그램이 사용할 수 있는 메모리의 주소가 2^{32}(4GB)라는 제약이 있다. 4GB라는 용량이 크다고 느낄 수도 있지만 최근의 서버용 응용 프로그램에서는 4GB로도 모자라는 경우가 종종 발생한다. 메모리 용량뿐 아니라, CPU와 메모리 간의 데이터 이동이 32비트 단위로 발생하고 CPU 내의 레지스터 크기가 32비트이므로 한 번에 32비트 데이터만큼의 계산만 할 수 있다.

64비트 윈도우의 경우 32비트에 대한 호환성을 유지하고 있어 별다른 변경 없이 기존의 32비트 응용 프로그램을 그대로 실행할 수 있다. 대신 64비트의 혜택을 고스란히 제공받지 못하고 32비트의 제약을 그대로 유지한 채 실행된다. 게다가 32비트 응용 프로그램을 64비트 환경에서 실행하기 위해 중간에 변환을 위한 층을 하나 두는데, 이를 WoW64(Windows 32bit on Windows 64bit)라고 한다. 따라서 32비트 응용 프로그램을 32비트 운영체제에서 실행하는 것보다 64비트 운영체제에서 실행했을 때 속도가 더 느려진다는 단점이 있다.

참고!	32비트 윈도우 운영체제는 4GB 가상 메모리 중 2GB 영역을 운영체제 전용으로 예약하기 때문에 사실상 응용 프로그램에서 사용할 수 있는 메모리 주소는 2GB에 불과하다.

E.2.7 64비트 응용 프로그램

64비트 운영체제에서만 실행되는 응용 프로그램이다. 마이크로소프트에서는 서버 운영체제의 경우 Windows 서버 2008 R2 제품 이후로 64비트 운영체제만 출시하고 있으므로 서버용 응용 프로그램에 관심이 있다면 64비트 환경에 익숙해지는 것이 좋다.

64비트는 메모리 주소를 2^{64}로 지정할 수 있는데, 이론상 16EB(Exabyte)의 메모리를 다룰 수 있지만 현실적인 이유로 윈도우에서는 커널 모드의 주소 공간으로 8TB(Terabyte)를, 사용자 모드의 주소 공간으로 8TB를 각각 예약한다.

CPU와 메모리 간의 데이터 이동이 64비트 단위로 발생하고 CPU 내의 레지스터 크기가 64비트이므로 한 번에 64비트 데이터만큼 계산할 수 있다. 이로 인해 경우에 따라 32비트 응용 프로그램과 비교해 2배 가까운 속도 향상이 가능해진다.

주의!	32비트 실행 파일과 64비트 실행 파일은 동일한 프로세스에 로드될 수 없다. 예를 들어, 32비트로 컴파일된 DLL은 64비트로 컴파일된 EXE 또는 DLL과 함께 사용할 수 없으며, 그 반대도 마찬가지다.

C#은 32비트와 64비트 응용 프로그램을 모두 만들 수 있다. 또한 "AnyCPU"라는 모드가 추가돼 있는데, 이 모드로 만들어진 C# 응용 프로그램은 32비트 운영체제에서는 32비트로, 64비트 운영체제에서는 64비트로 실행된다는 특징이 있다.

E.2.8 윈도우 이외의 운영체제

마이크로소프트 윈도우 운영체제 이외에도 개발자로서 주목해야 할 운영체제들이 있다.

- 유닉스(Unix)
  특정 분야를 제외하고는 현재 유닉스가 그렇게 중요한 위치를 차지하고 있지는 않지만, 현대 운영체제의 개념을 확립했다는 점에서 의미가 있다.
- 리눅스(Linux)
  유닉스에 뿌리를 두고 있지만, 개인이 만들어 오픈소스로 공개한 운영체제다. 무료로 공개된 운영체제라는 특징으로 인해 클라이언트 및 서버 운영체제에 걸쳐 폭넓게 사용되고 있다.
- 맥 OS X(Mac OS X)
  애플의 맥 하드웨어를 기반으로 실행되는 운영체제로서 GUI 기반이지만 유닉스에 뿌리를 두고 있어 터미널 모드에서 사용되는 명령어는 유닉스/리눅스와 호환되는 부분이 많다.

그리고 다음의 운영체제들은 보통 모바일 기기에 설치된다.

- iOS
  애플의 휴대전화 및 태블릿 제품인 아이폰(iPhone), 아이패드(iPad)에 탑재된다.
- 안드로이드

리눅스를 모바일 환경에 최적화시킨 운영체제다. 구글에서 개발했고 무료로 배포하기 때문에 각종 모바일 기기에 사용되고 있다.

- 윈도우 폰 OS
  윈도우 역시 핸드폰을 위한 경량화된 운영체제를 개발했고, 현재 윈도우 폰 10까지 출시됐다.

컴퓨터에 사용되는 운영체제는 대체로 x86 호환 시스템에서 운영되는 반면, 모바일/임베디드 기기에 사용되는 운영체제는 대부분 ARM 계열의 CPU에서 동작한다.

E.3 프로그래밍 용어

E.3.1 기계어

기계어(Machine language)란 CPU가 해석할 수 있는 2진수의 모음이다. 예를 들어, x86 CPU에서 EAX라고 불리는 레지스터의 값을 1과 비교하는 연산에 해당하는 기계어는 "1000 0011 1111 1000 0000 0001"이고 16진수로 표현하면 "83 F8 01"에 해당한다. 기계어도 하나의 프로그래밍 언어다. 단지 특별한 문법이 없고 그 기계어가 실행되는 CPU가 정한 규칙에 따라 2진 숫자를 나열해야 한다.

지금 세대는 잘 믿기지 않겠지만 과거에는 이처럼 숫자로 된 기계어로 프로그램을 만들던 적도 있었다.

E.3.2 어셈블리어, 소스코드, 컴파일

숫자로만 이뤄진 기계어로 프로그램을 만들기란 너무 불편하다. 이런 기계어에 의미를 부여할 수는 없을까? EAX 레지스터와 1을 비교한다는 작업을 숫자가 아닌 문자열로 "CMP EAX, 1"과 같이 표현하면 어떨까?

```
기계어: 1000 0011 1111 1000 0000 0001
➔ Compare EAX and 1
➔ CMP EAX, 1
```

보다시피 훨씬 이해하기 쉬워졌다. 이처럼 기계어로 하는 작업에 문자열로 의미를 붙여 만들어진 언어가 바로 어셈블리어(Assembly Language)다. 이 언어는 기계어와 거의 일대일로 대응된다.

여기서 "CMP EAX, 1"은 어셈블리어로 작성한 소스코드(Source code)에 해당한다. 물론 CPU는 소스코드를 직접 이해할 수 없으므로 그에 대응되는 기계어인 1000 0011 1111 1000 0000 0001로 바꾸는 작업이 필요하다. 이 과정을 컴파일(Compile)이라고 하며, 그와 같은 작업을 하는 프로그램을 컴파일러(Compiler)라고 한다. 컴파일러 중에서도 어셈블리 언어의 컴파일러를 특별히 어셈

블러(Assembler)라고 한다.

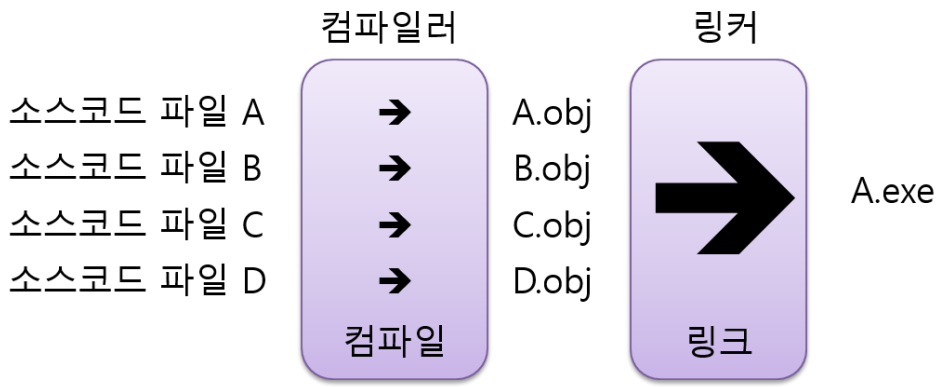
E.3.3 컴파일러, 링커, 빌드

프로그래밍 언어 가운데 컴파일러(Compiler)를 제공하는 것이 많다. 하지만 CPU는 그러한 언어의 고유 문법에 맞춰 작성된 소스코드를 이해할 수 없기 때문에 반드시 기계어로 바꿔야 한다. 이 과정을 좀 더 자세하게 설명해 보자. 관례상 어셈블리어로 소스코드를 작성하면 확장자가 asm으로 된 파일에 저장한다. 어셈블러로 asm 파일을 컴파일하면 확장자가 obj인 파일이 생성된다. 그런데 왜 exe가 아닌 obj일까?

프로그램을 한 개의 소스코드 파일로 만드는 것은 대단히 불편한 일이다. 물론 규모가 작은 프로그램이라면 한 개의 파일로도 충분하겠지만 대부분의 의미 있는 프로그램들은 여러 개의 파일로 나눠서 작성한다. 따라서 한 개의 asm 파일에 대해 한 개의 exe 파일로 출력할 수는 없다. 하나의 프로그램을 구성하기 위해 만들어진 모든 소스코드 파일을 컴파일하면 각각에 해당하는 obj 파일이 나오게 하고, 그 obj 파일들을 모아서 하나의 exe로 만드는 작업을 하게 된다.

앞에서 설명한 내용을 토대로 다시 컴파일러를 정의해 보자. 컴파일러란 소스코드에서 그것이 의미하는 기계어로 변환해 obj 파일에 담는 역할을 한다. 그리고 obj 파일을 모아서 하나의 exe 파일을 만드는 작업을 링크(Link)라고 하며, 그 작업을 수행하는 프로그램을 링커(Linker)라고 한다.

그림 E.7 컴파일러와 링커의 역할



프로그램을 하나 만드는 과정은 컴파일 과정과 링크 과정으로 분리돼 있지만, 일반적으로 "컴파일한다"라는 표현에는 그 두 가지 과정이 모두 포함된다. 최근에는 "컴파일 + 링크" 과정을 합쳐 '빌드(build)'라는 표현이 많이 쓰이는 추세다.

E.3.4 인터프리터 언어

어셈블리어는 컴파일을 통해 소스코드를 중간 파일(obj)로 출력하고 다시 링크 과정을 거쳐 실행 가능한 파일로 출력한다. 이런 언어를 컴파일러 언어라고 한다. 반면 인터프리터 언어

(Interpreter Language)는 명시적인 컴파일 과정 없이 소스코드에서 곧바로 프로그램을 실행하는 기능을 제공한다.

대표적인 예로 자바스크립트(JavaScript)가 있다. 자바스크립트는 실행을 위해 별도로 컴파일러를 사용하지 않는다. 단지 HTML 페이지 안에 자바스크립트 코드를 넣어두면 웹 브라우저가 소스코드의 문장을 하나씩 실행한다.

인터프리터 언어는 실행 시점에 소스코드를 해석한다는 특징이 있다. 반면 컴파일러 언어는 소스코드를 컴파일 과정에서 실행 가능한 기계어로 번역해 파일로 저장해 둔다. 이런 차이점으로 인해 일반적으로 실행 속도는 인터프리터 언어보다 컴파일러 언어가 더 빠르다.

C#은 컴파일러 언어라서 실행하기 전에 반드시 빌드 과정을 통해 결과 파일을 만들어야 한다.

E.3.5 저급/고급 프로그래밍 언어

기계어와 어셈블리어를 저급언어(Low Level Language)라고 한다. 비록 어셈블리어가 기계어보다 쉬워진 것은 사실이지만 개발자는 좀 더 편한 구문을 원했다. 예를 들어, "CMP EAX, 1" 같은 식의 문법을 좀 더 추상화해서 다음과 같이 표현하는 것도 가능하다.

```
if (n == 1)
{
   n = n * 2;
}
```

어셈블리어를 기계어로 변환한 것처럼, 위의 구문도 적절하게 기계어로 변환하는 또 다른 컴파일러를 제공할 수 있다면 가능한 일이다. 이렇게 탄생한 언어가 바로 고급 프로그래밍 언어(High Level Language)다. 초기의 포트란(Fortran), 코볼(Cobol), 파스칼(Pascal), C 언어와 같은 컴파일러 언어와 BASIC, LISP 같은 인터프리터 언어가 모두 고급 언어에 속하며, 이후에 나온 프로그래밍 문법을 가진 모든 언어도 여기에 속한다. C# 역시 고급 언어의 하나다.

E.3.6 네이티브 언어

컴파일러가 출력한 결과물이 특정 CPU를 위한 기계어일 때 이 언어를 네이티브 언어(Native Language)라고 한다. 어셈블리어는 당연히 네이티브 언어이고, 이후의 초기 언어는 대부분 여기에 속한다. 일반적으로 네이티브 언어로 만들어진 실행 파일은 속도가 빠르다는 특징이 있다. 왜냐하면 출력 결과물이 기계어라서 CPU에 의해 곧바로 해석될 수 있기 때문이다. 이런 특징은 한편으로는 단점으로 작용할 수 있는데, 가령 x86용 실행 파일은 ARM CPU에서 실행할 수 없다.

대표적인 네이티브 언어로 C, C++ 언어가 있다. 마이크로소프트의 경우 네이티브 언어라는 말과 함께 비관리 언어(Unmanaged Language)라는 말도 함께 쓰고 있다.

E.3.7 프로세스 가상 머신(VM)

자바와 닷넷 프레임워크의 공통점은 프로세스 가상 머신(Virtual Machine)에 있다. 그러한 환경의 프로그램을 실행하면 프로세스 내에 작은 가상 머신이 생성된다. 가상 머신은 말 그대로 "가상의 기계" 역할을 한다.

일반적으로 VM에서 해석하는 기계어는 그 VM이 실행되는 컴퓨터의 CPU에서 해석되는 기계어와는 다르다. 자바의 경우 자바 가상 머신에서 해석되는 기계어를 바이트코드(Bytecode)라 부르고, 닷넷 프레임워크에서는 중간 언어(IL: Intermediate Language) 코드라고 한다.

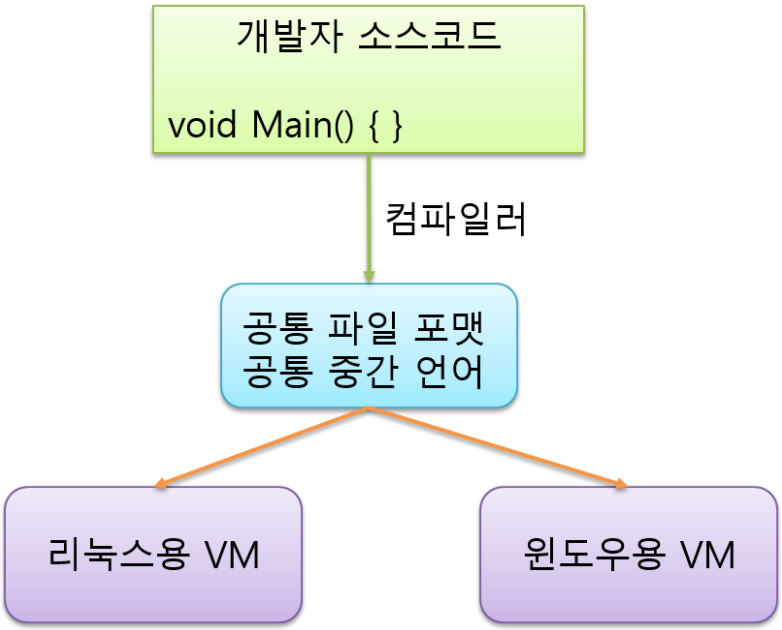
참고!	보통은 가상 머신의 고유한 기계어를 모두 중간 언어라고 통칭한다.

중간 언어는 "공통 기계어"로서의 역할을 수행한다. 개발자가 작성한 VM용 프로그램은 중간 언어로 된 파일로 만들어진다. 기존의 네이티브 언어로 된 프로그램이 특정 CPU에 종속되는 기계어를 생성한 것과 대비된다. 물론 VM의 중간 언어는 CPU가 곧바로 실행할 수 없다. 이러한 이유로 VM은 중간 언어를 CPU가 실행할 수 있는 기계어로 변환하는 작업도 담당한다.

VM 환경의 또 다른 특징은 "공통 실행 파일 포맷"에 있다. 윈도우에서 C/C++ 언어의 소스코드가 컴파일되면 윈도우 운영체제가 자체적으로 정의한 실행 파일 포맷에 맞게 기계어를 담는다. 마찬가지로 리눅스는 나름대로의 실행 파일을 정의하고 있으며, 이는 윈도우의 실행 파일 포맷과 완전히 다르다. 따라서 VM에서 정의된 공통 실행 파일 포맷에 따라 컴파일러들이 실행 파일(예: EXE 또는 DLL)을 생성해 준다면 VM은 그 파일을 손쉽게 메모리로 읽어와 실행할 수 있다.

기존에는 개발자가 만든 실행 프로그램이 해당 환경에 종속됐다. 예를 들어, 윈도우 환경에서 만든 프로그램은 오직 윈도우에서만 실행됐다. 하지만 중간 언어를 다루는 VM이 제공되면서 상황이 완전히 바뀐다.

그림 E.8 VM의 역할



'그림 E.8'처럼 윈도우나 리눅스, 심지어 ARM CPU 기반의 운영체제에서도 중간 언어를 해석할 수 있는 VM만 제공된다면 개발자가 만든 프로그램을 실행할 수 있게 된 것이다.

VM의 대표적인 사례가 바로 자바의 JVM(Java Virtual Machine)과 닷넷 프레임워크의 CLR(Common Language Runtime)이다.

E.3.8 가상 머신 지원 언어

이를테면, 자바와 C#이 대표적인 언어에 해당한다. 이러한 언어로 작성된 소스코드를 컴파일하면 기계어로 된 결과물을 산출하지 않는다. 대신 각각 JVM과 CLR에서 지원되는 중간 언어로 결과물을 생성한다. 이러한 출력물은 일반적인 기계어를 담고 있지 않으므로 직접 실행될 수 없고 반드시 가상 머신(JVM 또는 CLR) 내에서만 동작할 수 있다.

비교를 위해 기존의 네이티브 언어로 만들어진 결과물의 특징을 살펴보자.

- 네이티브 언어로 컴파일된 결과물은 그것이 실행될 CPU의 기계어로 돼 있다.
- 네이티브 언어로 컴파일된 결과물은 그것이 실행될 운영체제에서 제공되는 시스템 함수를 사용한다(윈도우의 경우 Win32 API를 사용해서 제작되지만 리눅스에는 Win32 API가 없으므로 실행할 수 없다).
- 네이티브 언어로 컴파일된 결과물은 그것이 실행될 운영체제에서 요구하는 파일 포맷으로 돼 있다.(윈도우와 리눅스의 실행 파일 포맷은 다르다.)

따라서 네이티브 언어로 컴파일된 실행 파일은 CPU 및 운영체제가 다른 환경에서는 정상적으로 실행되지 않는다. 이런 제약은 중간 언어를 둔 프로세스 가상 머신으로 해결할 수 있다.

가상 머신을 지원하는 언어는 단지 공통된 실행 파일 포맷에 미리 약속된 중간 언어로 된 결과물을 만들어낸다. 그리고 각 운영체제에서는 그러한 중간 언어를 실행 가능하게끔 번역하는 프로

세스 가상 머신을 만들면 된다. 따라서 자바와 C#으로 만든 응용 프로그램은 대상이 되는 환경에 적합한 프로세스 가상 머신(JVM 또는 CLR) 환경을 미리 설치해 둬야 한다.

F. 참고 자료

- 마이크로소프트 개발자 웹 사이트
  - **http://msdn.microsoft.com**

- C# Language Design
  - **https://github.com/dotnet/csharplang**

- 필자의 웹 사이트
  - **http://www.sysnet.pe.kr**

- 마이크로소프트 사내 개발자들의 블로그
  - **http://blogs.msdn.com/**

- MSDN Magazine
  - **http://msdn.microsoft.com/en-us/magazine/default.aspx**

- CLR via C# 4th Edition 한국어판(김명신, 남정현 공역, 비제이퍼블릭, 2014년 11월 25일 출간)

- The C# Programming Language (Fourth Edition) 한국어판(김도균, 안철진 역, 에이콘출판사, 2012년 6월 29일 출간)

- C# 5.0 언어 명세: https://www.ecma-international.org/publications/standards/Ecma-334.htm

- 닷넷의 가비지 컬렉션: https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/

- C# 오픈소스 컴파일러 – Roslyn(https://github.com/dotnet/roslyn)