제5장
닷넷 기본과 어셈블리

이 장에서는 닷넷 환경에서 프로그램을 작성하기 위한 기본 개념에 대하여 다룹니다. 특히 어셈블리의 개념과 어셈블리를 분석할 수 있는 ILDASM에 대하여 공부합니다.

Step1 : 어셈블리 이해하기

Step2 : ILDASM 도구로 어셈블리 분석하기

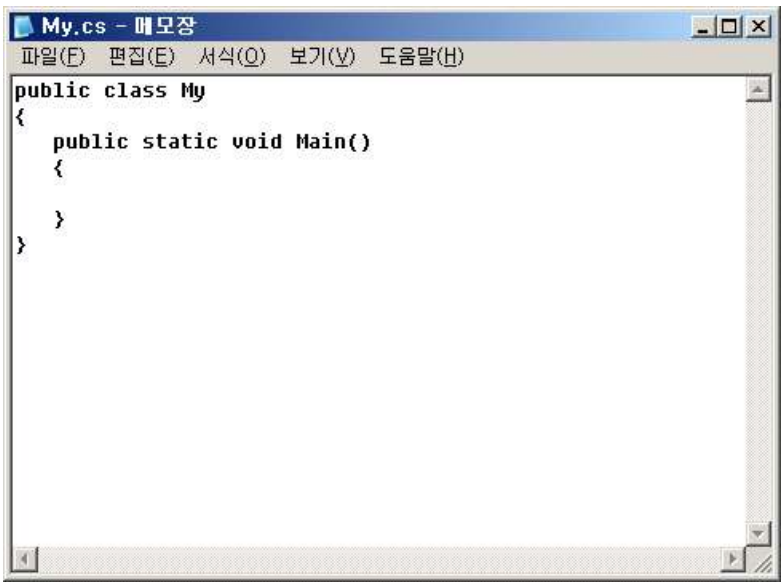
Step3 : DLL 어셈블리 분석하기

Step4 : CLR과 기본 클래스 라이브러리

Step1 어셈블리 이해하기

메모장을 이용하여 다음 코드를 작성한 후 my.cs파일로 저장합니다.

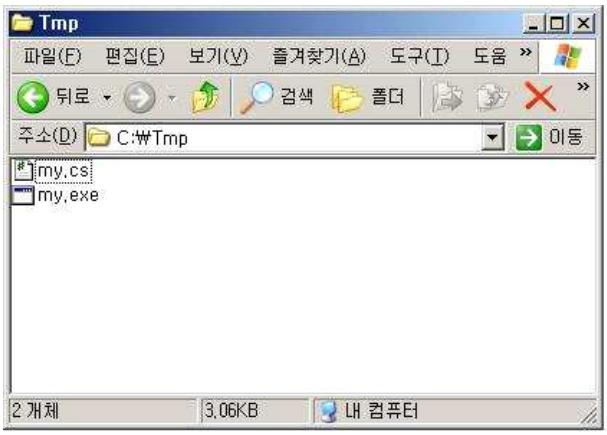
```
public class My
{
   public static void Main()
   {

   }
}
```



작성한 프로그램을 도스 환경에서 컴파일해 보겠습니다. 도스 창을 이용하여 다음과 같이 컴파일합니다. csc는 c-sharp compiler 이니셜입니다.

```
C:\Tmp>csc my.cs
```

탐색기를 이용하여 생성된 파일을 확인해 봅니다.



참고로 C# 컴파일러는 다음 폴더에 있습니다.

C:\WINDOWS\Microsoft.NET\Framework\v3.5

my.exe 파일은 닷넷 환경에서 실행되는 파일로서 이를 어셈블리(assembly)라고 합니다. 이 파일은 아래아 한글 실행 파일 hwp.exe 등과 같은 기존의 윈도우 실행 파일과 전혀 다릅니다. 무엇이 다른지를 이해하기 위해 기존 실행 파일의 문제점을 짚어보도록 하겠습니다. 기존 윈도우 실행 파일의 문제점은 크게 다음과 같이 두 가지로 설명할 수 있다.

- 기존 실행 파일에는 특정 CPU에서 실행되는 기계어로 구성되어 있어서 다른 플랫폼(가령 유닉스) 환경에서는 실행할 수 없습니다.
- 기존 실행 파일만 보면 코드가 어떤 내용인지 알 방법이 없습니다. 가령 dll로 작성되어 있는 라이브러리 파일에는 함수 프로토타입에 관한

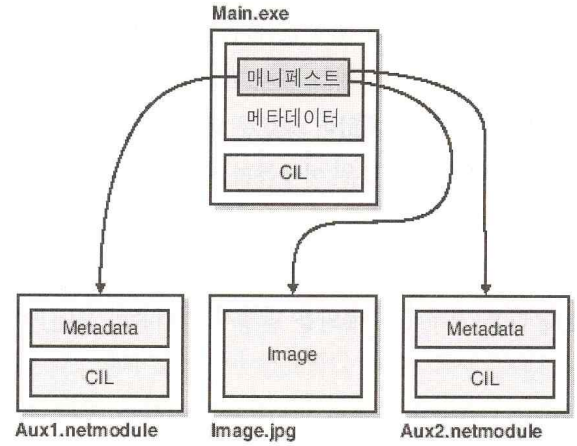
정보가 존재하지 않아서 이 라이브러리를 사용하려면 해당 헤더 파일도 가지고 있어야 합니다.

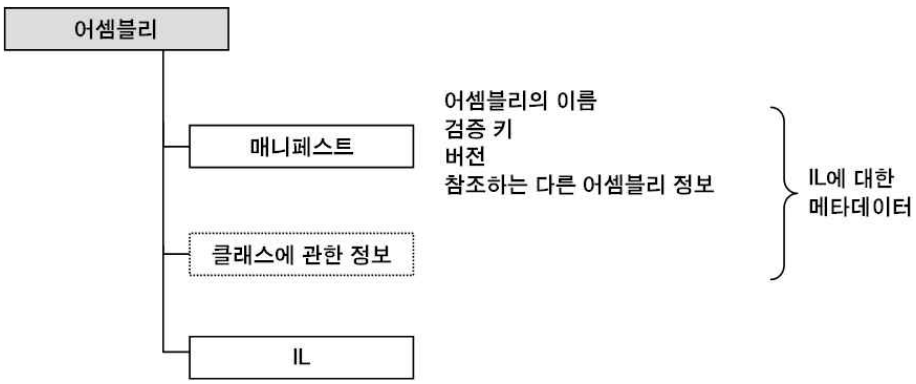
어셈블리는 위와 같은 문제점을 해결하였습니다. 즉,

- 어셈블리는 특정 CPU에서 실행되는 기계어가 아니라 런타임(가상 머신)에서 실행되는 **중간 형태의 언어**인 IL(Intermediate Language)로 구성됩니다. 그럼으로써 런타임이 설치되어 있기만 하면 다른 플랫폼에서도 실행할 수 있습니다.
- 어셈블리에는 실행 코드인 IL 뿐만 아니라 이에 대한 정보인 메타데이터(metadata)라는 것이 존재합니다. 가령 어셈블리에 있는 함수의 프로토타입, 클래스 정보 등이 들어있습니다. 그럼으로써 어셈블리 파일 하나만 있으면 사용이 가능합니다. 헤더 파일이 따로 필요하지 않습니다.

이러한 어셈블리는 다음과 같은 **특징**을 가지고 있습니다.

- 어셈블리는 설치 및 배포를 위한 최소의 단위입니다.
- csc 컴파일러가 생성하는 exe, dll, 그리고 각종 리소스 파일이 모두 어셈블리입니다.
- 결국, 닷넷 응용을 만든다는 것은 어셈블리(assembly)를 만드는 것을 의미합니다.
- 어셈블리는 IL과 IL에 대한 정보인 메타데이터로 구성됩니다.
- 메타데이터에는 매니페스트(menifest)라는 메타데이터가 있는데, 여기에는 자신이 사용하는 다른 어셈블리에 관한 목록(manifest)이 들어있으며, 이외에도 어셈블리의 이름, 버전 번호 등이 들어 있습니다.
- 메타데이타에서 매니페스트 뿐만 아니라 어셈블리에서 사용하는 클래스에 관한 정보가 들어 있습니다.



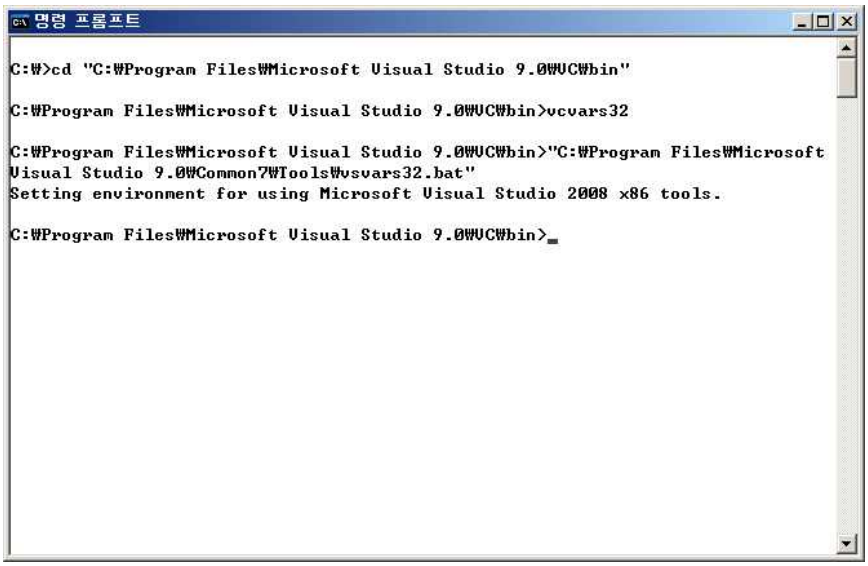


ILDASM 도구로 어셈블리 분석하기

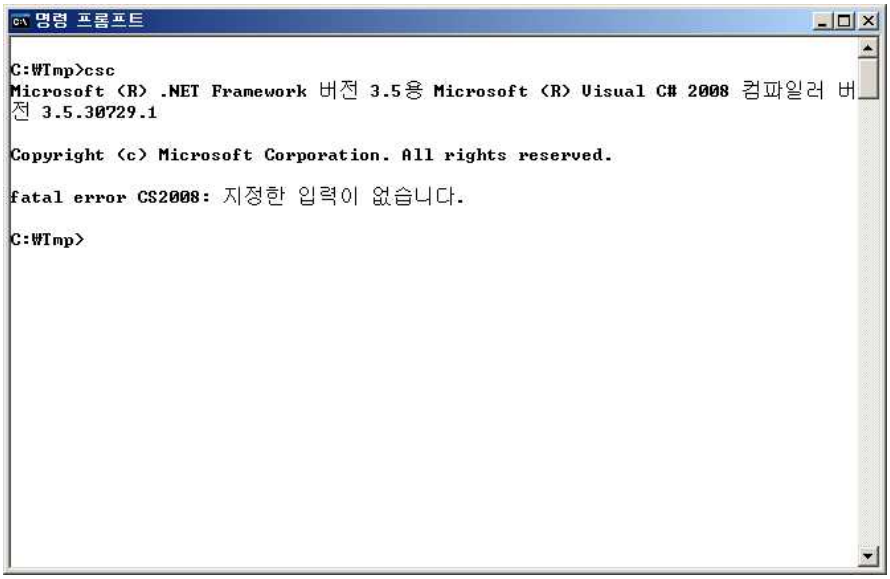
ildasm.exe 도구를 이용하면 앞서 생성한 어셈블리 my.exe에 들어있는 IL과 메타데이터를 사람이 읽을 수 있도록 보여줍니다. 도스창 아무데서나 ildasm.exe과 csc.exe 컴파일러를 실행할 수 있도록 하기 위해 두 실행 파일이 있는 경로를 환경변수 Path에 추가합니다. 방법은 간단합니다. 도스창을 이용하여 다음 폴더로 이동합니다.

C:\Program Files\Microsoft Visual Studio 9.0\VC\bin

그리고 vcvars32.bat 파일을 실행합니다.

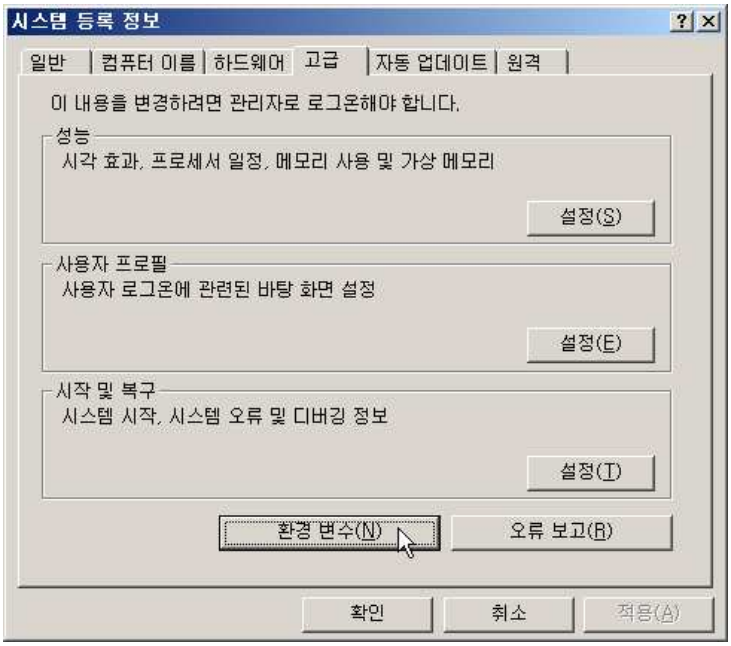


이제 다른 폴더로 이동하여도 csc.exe와 ildasm.exe 파일을 실행할 수 있습니다. 다음은 C:\Tmp 폴더로 이동한 후 csc.exe를 실행한 모습입니다.

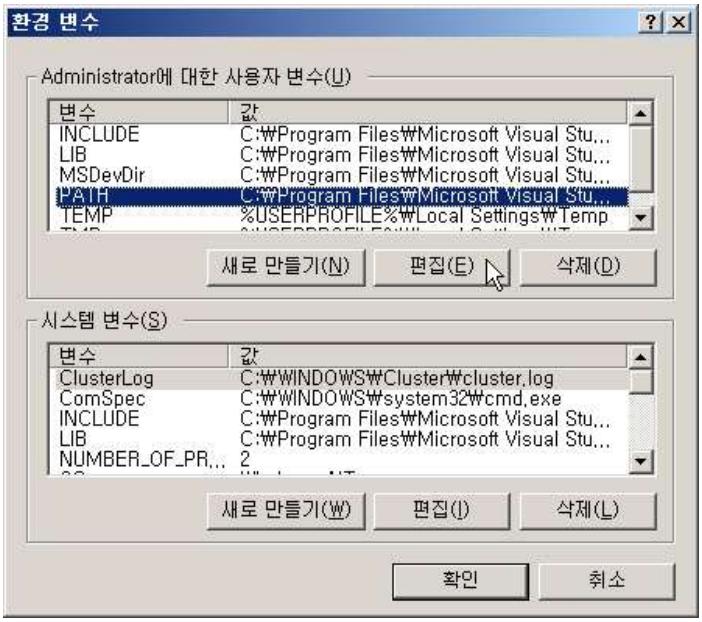


하지만 매번 vcvars32.bat 파일을 찾는 것이 번거로우므로 이 파일을 c:\ 폴더에 복사해 두면 이후 쉽게 실행할 수 있을 것입니다.

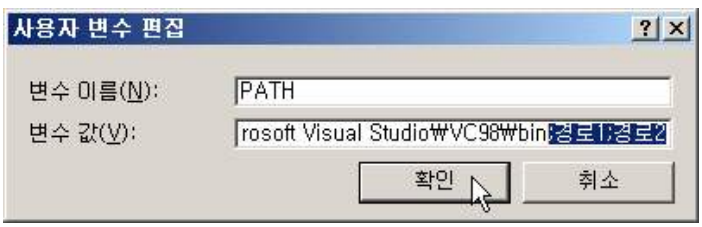
혹은 아예 Path 환경 변수에 경로를 설정합니다. 바탕 화면의 내 컴퓨터 > [오른쪽 버튼 클릭] > 속성 > [고급 탭 선택] > 환경 변수 버튼을 눌러 설정할 수 있습니다.



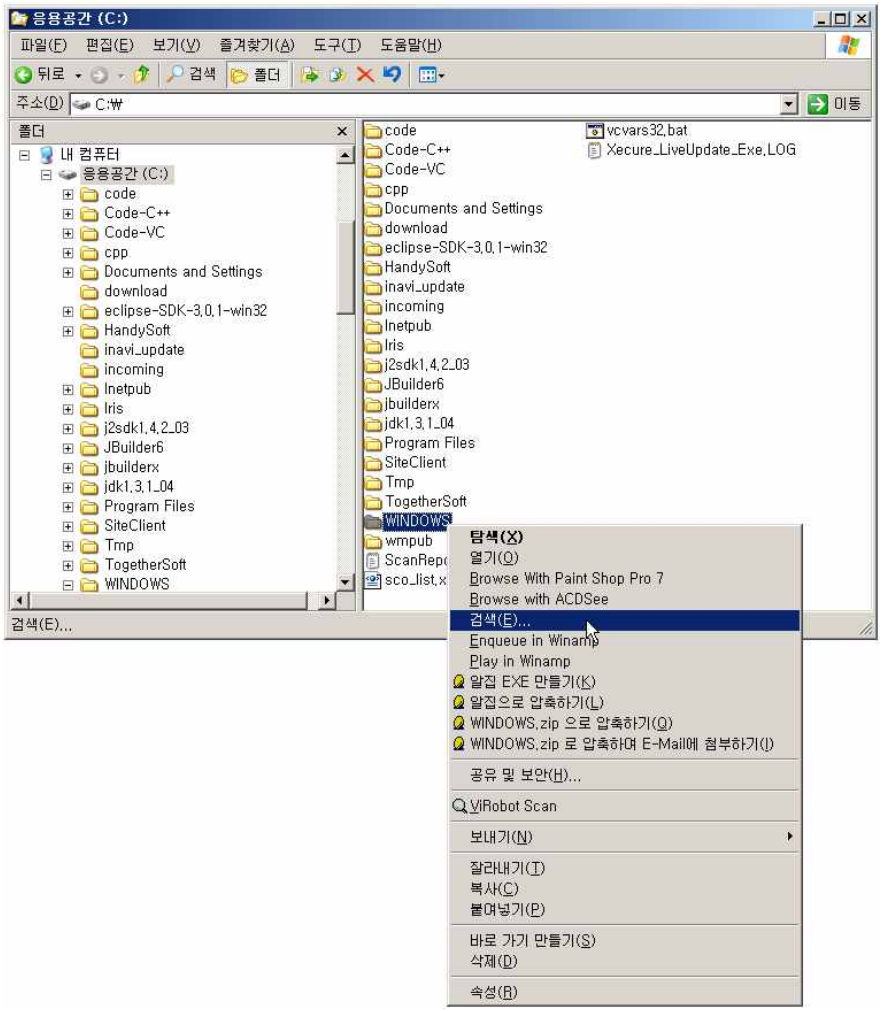
환경변수 PATH를 선택한 후 편집 버튼을 누릅니다.



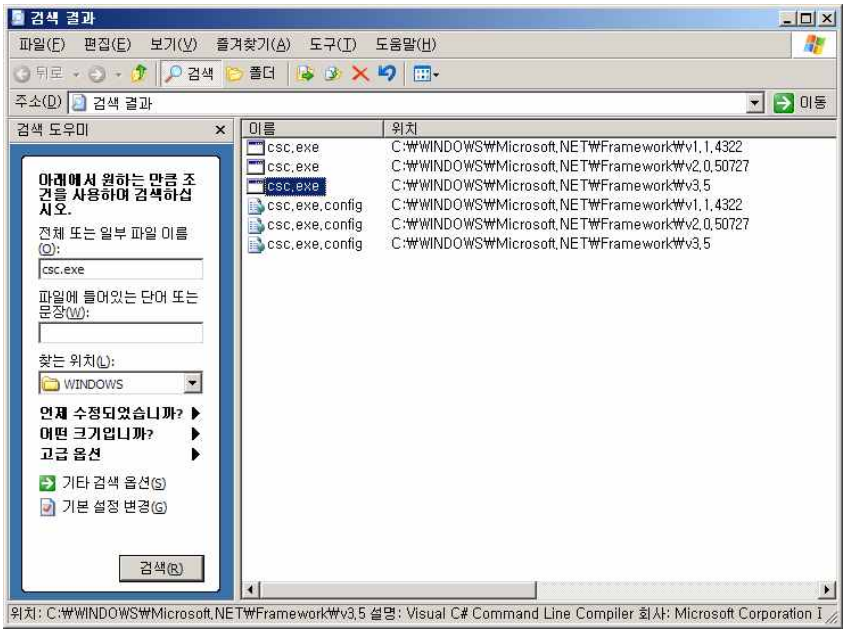
그러면 다음과 같이 편집 대화상자가 표시됩니다. 아래와 같이 csc.exe 실행 파일이 들어있는 폴더 경로1과 ildasm.exe 실행 파일이 있는 폴더 경로2를 세미콜론(;)으로 분리하여 입력합니다.



경로1과 경로2를 알아내기 위하여 다음과 같이 탐색기를 이용하여 C:\Windows 폴더에서 csc.exe 파일이 들어있는 폴더를 찾습니다.


응용공간 (C:)
파일(F) 편집(E) 보기(V) 즐겨찾기(A) 도구(T) 도움말(H)
뒤로 검색 폴더
주소(D) C:₩
이동
폴더
내 컴퓨터
응용공간 (C:)
code
Code-C++
Code-VC
cpp
Documents and Settings
download
eclipse-SDK-3.0.1-win32
HandySoft
inavi_update
incoming
Inetpub
Iris
j2sdk1.4.2_03
JBuilder6
jbuilderx
jdk1.3.1_04
Program Files
SiteClient
Tmp
TogetherSoft
WINDOWS
wmpub
ScanRep
sco_list.x
vcvars32.bat
Xecure_LiveUpdate_Exe.LOG
검색(E)...
탐색(X)
열기(O)
Browse With Paint Shop Pro 7
Browse with ACDSee
검색(E)...
Enqueue in Winamp
Play in Winamp
알집 EXE 만들기(K)
알집으로 압축하기(L)
WINDOWS.zip 으로 압축하기(Q)
WINDOWS.zip 로 압축하여 E-Mail에 첨부하기(I)
공유 및 보안(H)...
ViRobot Scan
보내기(N)
잘라내기(T)
복사(C)
붙여넣기(P)
바로 가기 만들기(S)
삭제(D)
속성(R)

그 결과 다음과 같이 경로1을 알 수 있습니다.



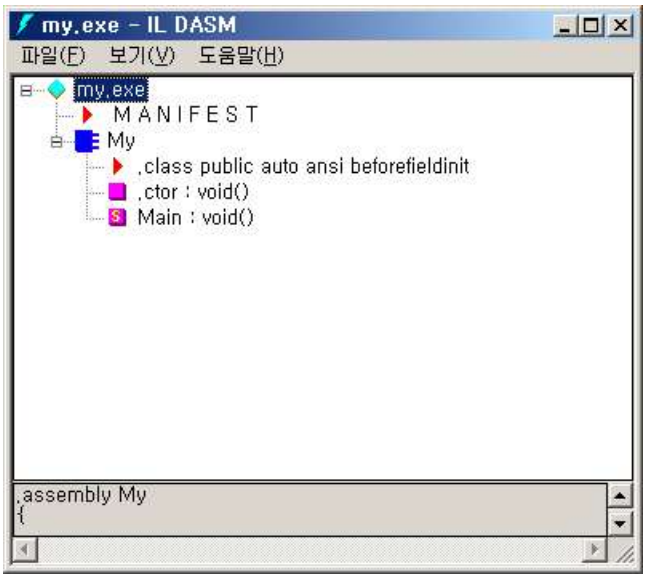
마찬가지 방법으로 C:\Program Files\ 폴더에서 ildasm.exe 파일이 있는 폴더를 찾습니다. 그 결과 찾은 폴더는 다음과 같습니다.

파일	설명	폴더
ces.exe	C# 컴파일러	C:\WINDOWS\Microsoft.NET\Framework\v3.5
ildasm.exe	중간언어 역어셈블러 도구	C:\Program Files\Microsoft SDKs\Windows\v6.0A\bin

앞으로는 도스창을 새로 실행하여도 두 도구를 바로 실행할 수 있습니다. 그러면 도스창을 실행하여 c:\TMP 폴더로 이동한 후 다음과 같이 ildasm 도구

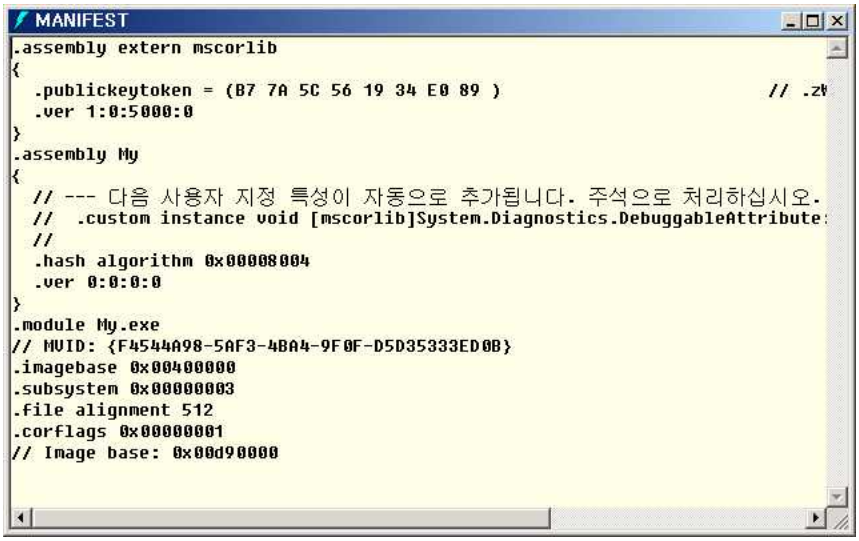
를 실행합니다. 그러면 my.exe 어셈블리 내부를 살펴볼 수 있습니다.

```
C:\Tmp>ildasm my.exe
```



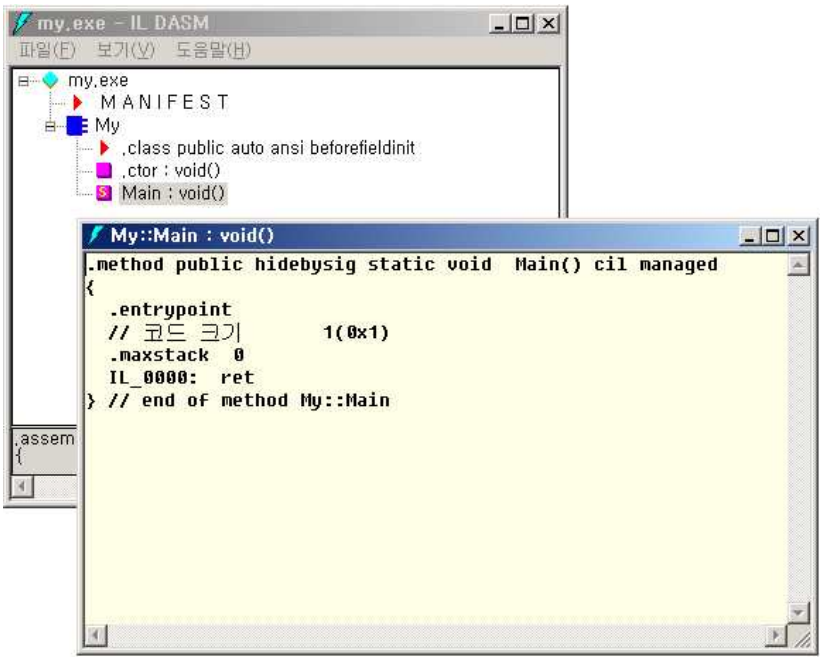
위 그림은 my.exe 어셈블리에 대한 정보 즉, 메타데이터를 보여줍니다. my.exe 어셈블리에는 위의 메타데이터뿐만 아니라 마이크로소프트에서 제안하는 중간 형태의 언어로서 실제로 .NET에 의해 실행되는 IL 코드가 들어있습니다.

위 그림의 메타데이터를 보면 MANIFEST라는 매니페스트를 볼 수 있는데, 여기에는 어셈블리 이름(My), 어셈블리 버전(0:0:0:0), 참조하는 다른 어셈블리들(mscorlib)에 관한 정보가 들어있습니다. 위 그림에서 MANIFEST 부분을 두 번 클릭하면 다음과 같은 확인할 수 있습니다.



위 그림을 보면 mscorlib 어셈블리에 publickeytoken 이라는 것이 들어있는데, 이는 mscorlib 어셈블리를 인증하기 위한 키로서, 이를 이용하여 이름은 mscorlib로 같지만 전혀 다른 어셈블리가 사용되는 것을 막을 수 있습니다.

ildasm에서 Main 함수를 두 번 클릭하면 Main 함수에 대한 IL 코드를 볼 수 있습니다.

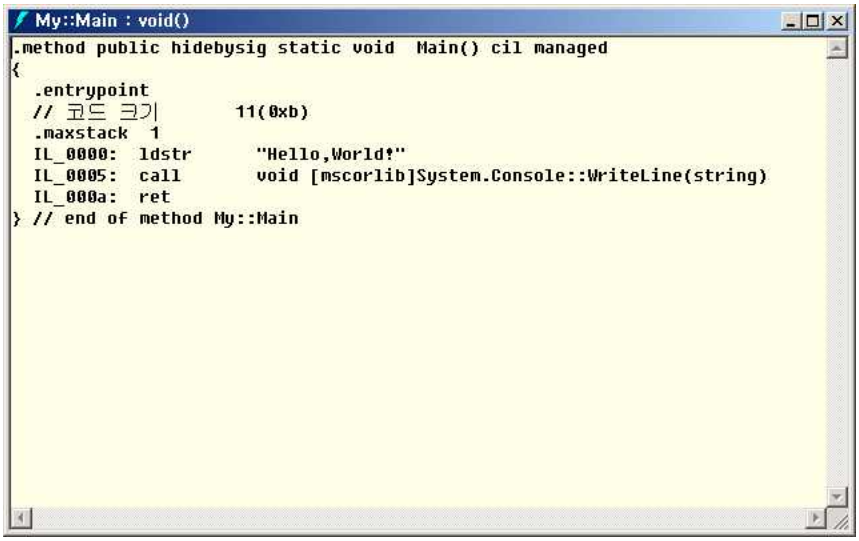


앞서 표시된 코드가 바로 마이크로소프트에서 정의한 중간 언어, 즉 IL입니다.

이제 Main 함수를 수정해 보겠습니다. 다음과 같이 Main 함수에 Hello,World! 문자열을 출력하는 코드를 작성합니다.

```
class My
{
    public static void Main()
    {
        System.Console.WriteLine("Hello,World!");
    }
}
```

ildasm 도구를 이용하여 Main 함수의 IL을 보면 다음과 같이 바뀌어 있습니다.



ldstr 명령어는 "Hello,World!"라는 문자열을 스택에 push합니다. 그 다음 호출한 call 명령어는 스택 top에 있는 문자열을 꺼낸 후(pop) WriteLine 함수를 이용하여 화면에 출력합니다.

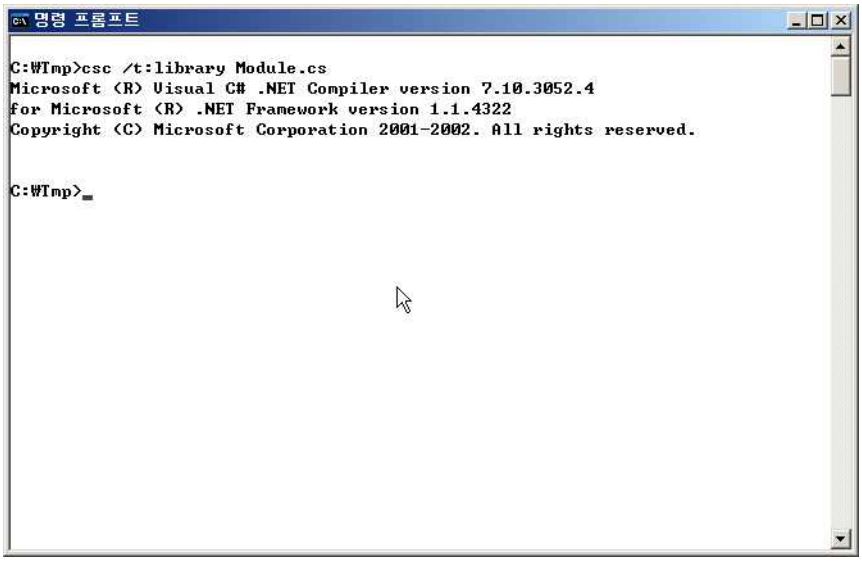
DLL 어셈블리 분석하기

이번에는 dll 형태의 어셈블리를 만들어보겠습니다. 메모장을 이용하여 다음 프로그램을 작성한 후 Module.cs 파일로 저장합니다.

```
//---------------
// Module.cs
//---------------
public class My
{
       public My()
       {

       }
}
```

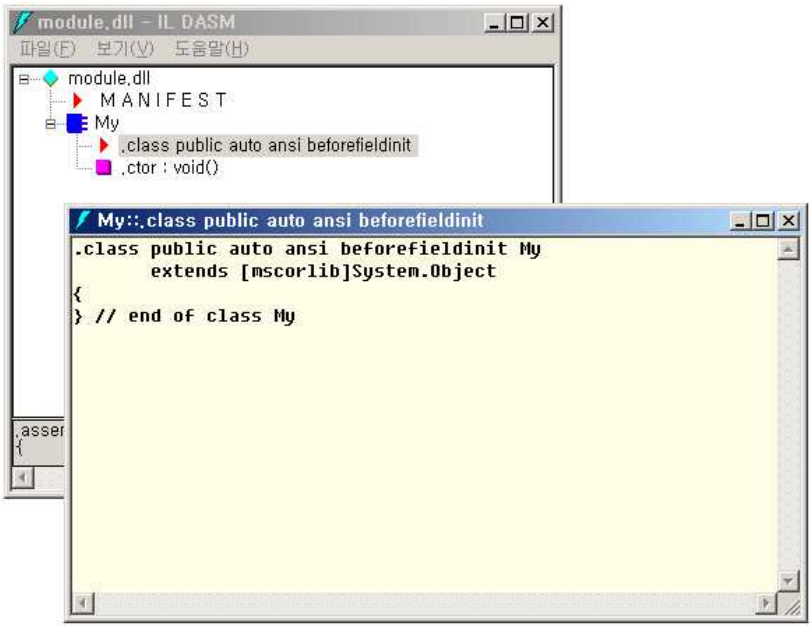
위 프로그램은 Main 함수가 없습니다. 따라서 실행파일(*.exe)로 만들 수 없으며, 다음과 같이 dll 라이브러리를 생성하도록 옵션을 입력하여 컴파일합니다.

C:\Tmp>csc /t:library module.cs



ildasm 도구를 이용하여 클래스를 본 모습입니다. My 클래스가 public으로 선언되어 있고, 또한 생성자 함수가 있음을 알 수 있습니다.

다음은 클래스 정보를 본 모습입니다.

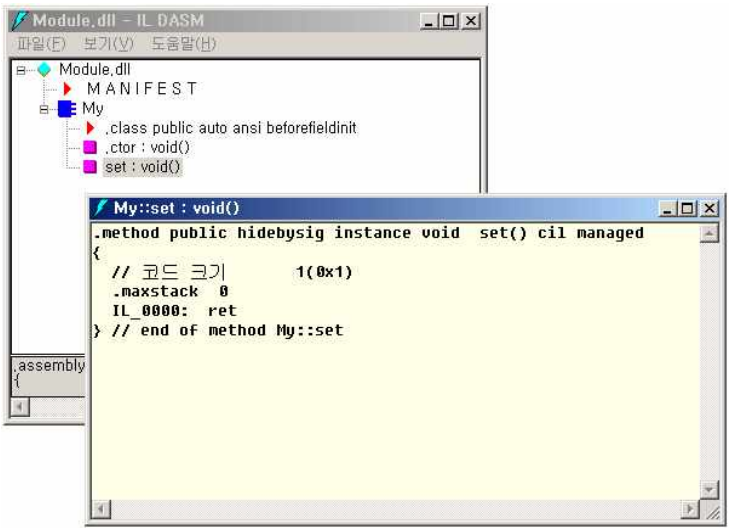


다음과 같이 멤버 함수를 추가해 봅니다.

```
//---------------
// Module.cs
//---------------
public class My
{
        public My()
        {

        }

        public void set()
```

```
        {

        }
}
```

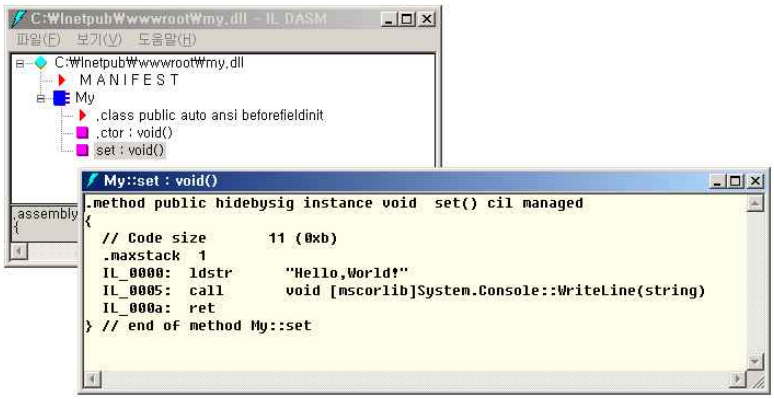
추가한 멤버 함수 IL 코드를 보면 다음과 같습니다.



set 함수는 아무런 일도 수행하지 않습니다. IL 명령어 ret를 이용하여 그냥 반환합니다. 다음은 문자열을 출력하는 코드를 작성한 예입니다.

```
//---------------
// Module.cs
//---------------

public class My
{
        public My()
        {
```

```
        }

        public void set()
        {
                System.Console.WriteLine("Hello,World!");
        }
}
```



문자열 "Hello,World!"를 스택에 로드(push)한 후(ldstr) 닷넷 프레임워크에 있는 기본 클래스 라이브러리(Basic Class Library) Console의 WriteLine 함수를 호출(call)하여 문자열을 출력합니다. 이때 WriteLine 함수는 스택 top에 있는 문자열, 즉, "Hello,World!"를 꺼내어(pop) 화면에 출력합니다.

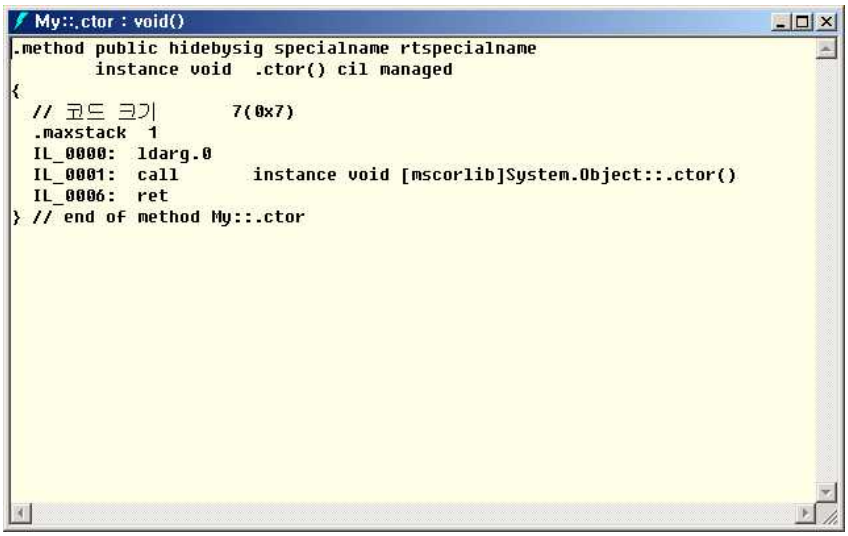
다음은 멤버 변수를 정의한 예입니다. 그리고 set 멤버 함수에 작성한 코드를 지웠습니다.

```
//---------------
// Module.cs
//---------------
```

```
using System;

public class My
{
	private int iNumber;

	public My()
	{

	}

	public void set()
	{

	}
}
```

ildasm 도구로 어셈블리를 보면 다음과 같이 멤버 변수가 있음을 알 수 있습니다.

멤버 변수를 초기화하지 않았습니다. 생성자 함수의 IL을 보겠습니다.



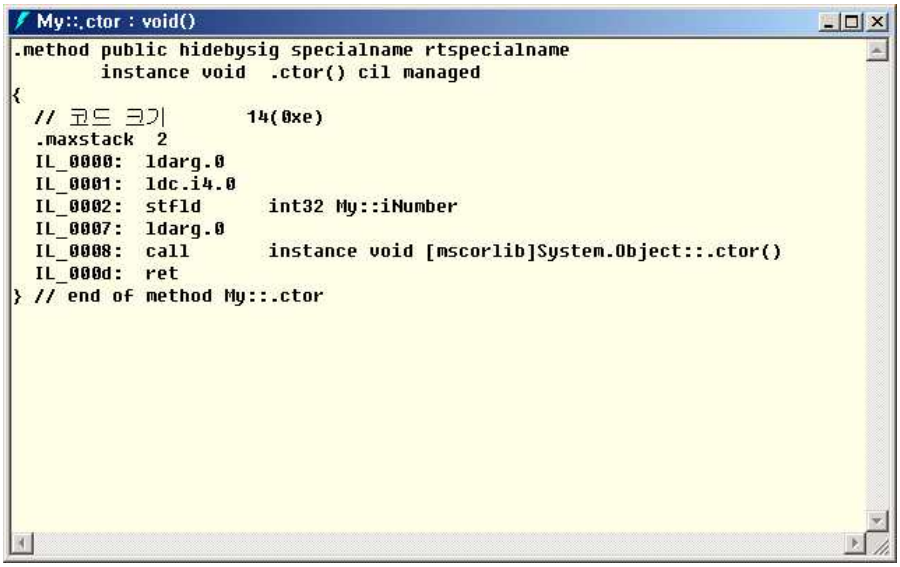
ldarg.0이라는 명령어는 현재 객체의 주소값인 this를 스택에 push하는 명령어입니다. 그 다음 call 명령어는 (this 객체가) Object 클래스로부터 상속받은 생

성자 함수를 호출합니다.

초기화를 수행하도록 코드를 작성해 보겠습니다.

```
//---------------
// Module.cs
//---------------
public class My
{
        private int iNumber = 0;

        public My()
        {

        }

        public void set()
        {

        }
}
```

생성자 함수의 IL을 다시 보면 다음과 같습니다.



ldarg.0 명령어에 의해 스택에 this 값이 push됩니다. ldc.i4.0 명령어에 의해 0 값이 스택에 push됩니다. stfld 명령어는 스택에 있는 두 값 this와 0을 pop하여 this.iNumber 멤버 변수에 0값을 저장합니다. 즉, 초기화가 수행됩니다. 이로서 멤버 변수를 초기화하면 생성자 함수에 어떤 코드가 작성되는지를 알 수 있습니다.

Set 멤버 함수에 다음 코드를 작성합니다.

```
//---------------
// Module.cs
//---------------
public class My
{
      private int iNumber = 0;
```

```
        public My()
        {

        }

        public void set(int i)
        {

        }
}
```

ildasm 도구를 이용하여 set 함수 안에 작성된 IL 코드를 보면 다음과 같습니다.

```
.method public hidebysig instance void  set(int32 i) cil managed
{
  // 코드 크기       1(0x1)
  .maxstack  0
  IL_0000:  ret
} // end of method My::set
```

32비트 변수 i를 정의하고 있음을 볼 수 있습니다. 이제 아래와 같이 작성하겠습니다.

```
//---------------
// Module.cs
//---------------
public class My
{
        private int iNumber;
```

```
        public My()
        {

        }

        public void set(int i)
        {
              iNumber = i;
        }
}
```

다음과 같이 set 멤버 함수를 호출한다고 가정하겠습니다.

```
My gildong = new My();
gildong.set(7);
```

set 멤버 함수를 파라미터 혹은 인자가 하나입니다. 하지만 실제로는 그렇지 않습니다. 여러분이 모르는 사이에 또 다른 값 this가 set 멤버 함수의 파라미터로 넘겨집니다. 결국 다음과 같이 호출하는 모양이 됩니다.

```
set(this, 7);
```

현 객체 gildong의 주소값(숨겨진 포인터 this)이 0번째 인자로 자동으로 전송되고 있다는 뜻이지요. 여러분이 기술한 7은 사실은 두 번째 파라미터로 전달됩니다.

아래의 코드를 보면 0번째 인자로 전송된 this 값(현 객체의 주소값)을 스택에 로드하기 위하여 ldarg.0가 있습니다. ldarg는 load argument의 준말입니다.

```
.method public hidebysig instance void set(int32 i) cil managed
{
  // 코드 크기       8(0x8)
  .maxstack  2
  IL_0000:  ldarg.0  //0번째 인자(숨겨진 포인터 this)를 스택에 로드함(push)
IL_0001:  ldarg.1  //1번째 인자 i에 있는 값을 스택에 로드함
//this 객체의 멤버 iNumber에 1번 인자의 값을 저장함
  IL_0002:  stfld     int32 My::iNumber
  IL_0007:  ret
} // end of method My::set
```

this.iNumber 멤버 변수에 i값을 저장합니다. 결국 다음과 같이 실행됩니다. 또한 1번째 인자로 전송된 i값을 스택에 넣은 후 stfld int32 My::iNumber를 호출하면 됩니다.

```
this.iNumber = i;
```

이제까지 작성한 클래스를 XXX 네임스페이스를 갖도록 선언해 보겠습니다.

```
//---------------
// Module.cs
//---------------

namespace XXX
{
       public class My
       {
              private int iNumber = 0;

              public My()
              {
```

```
        }

        public void set(int i)
        {
            iNumber = i;
        }
    }
}
```

ildasm 도구로 보면 XXX라는 네임스페이스에 My 클래스가 선언되어 있음을 볼 수 있습니다.

Step4 CLR과 기본 클래스 라이브러리

다음 프로그램을 살펴보겠습니다.

```
public class My
{
   public static void Main()
   {
       System.Console.WriteLine("Hello,World!");
   }
}
```

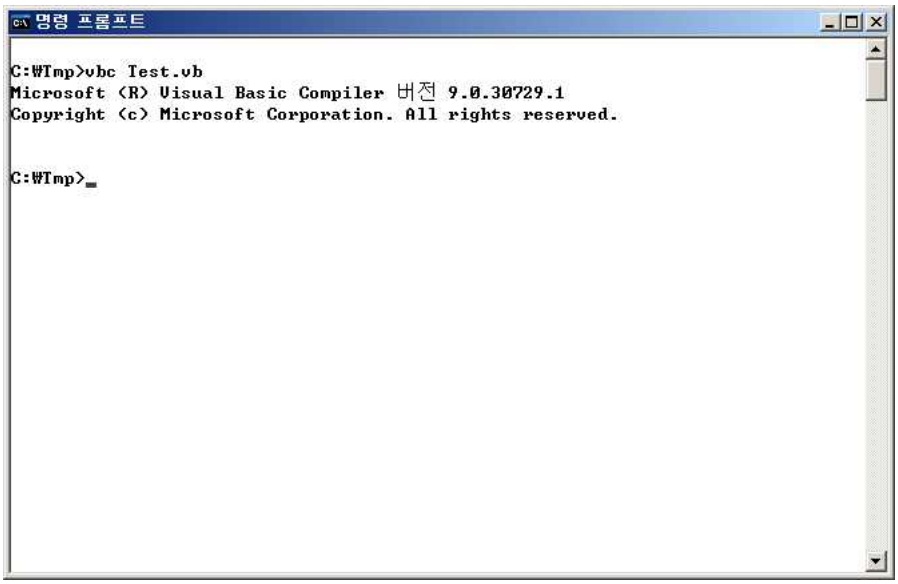
위 코드를 보면 Console 클래스의 WriteLine 정적 멤버 함수를 호출하고 있습니다. 나중에 설명하겠지만, Console과 같이 기본적으로 존재하는 클래스들을 기본 클래스 라이브러리 (Base Class Library)라고 합니다.

C#이 아닌 다른 언어를 이용하여 동일한 어셈블리를 작성해 보겠습니다. 다음과 같이 Visual Basic.NET 프로그램을 작성한 후 C:\Tmp 폴더에 Test.vb라는 파일로 저장합니다.

```
Module Test
    Sub Main()
        System.Console.WriteLine("Hello,World!")
    End Sub
End Module
```

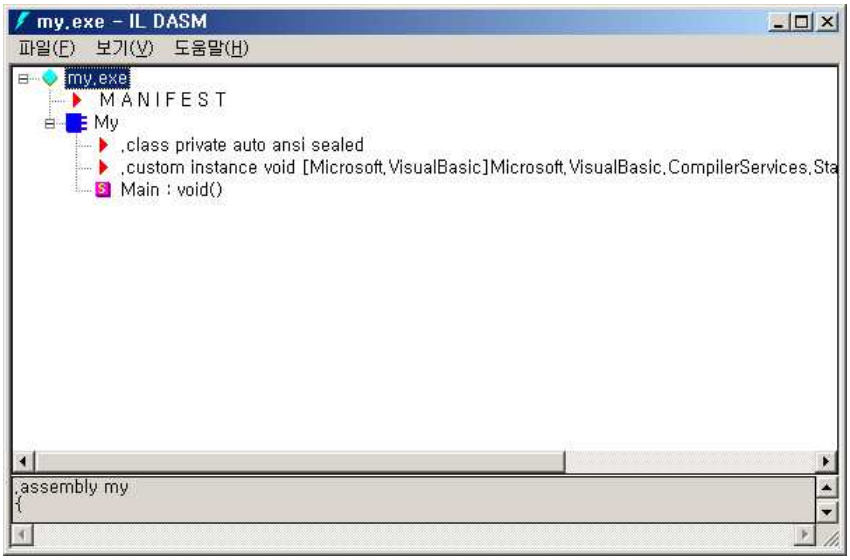
비주얼 베이직 컴파일러는 vbc.exe 입니다. 따라서 다음과 같이 컴파일합니다.

C:\Tmp>vbc My.vb

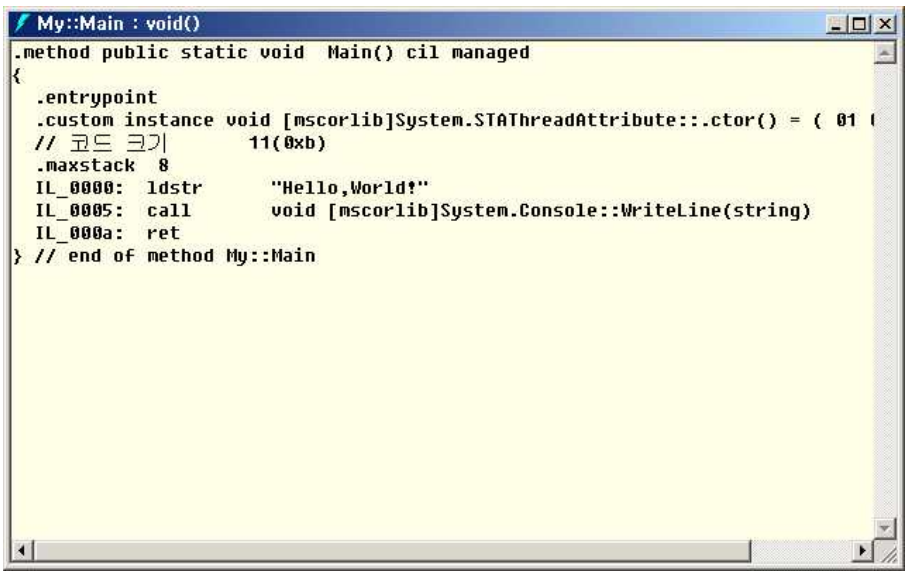


다음과 같이 입력하여 ildasm 도구를 실행하여 어셈블리를 로드합니다.

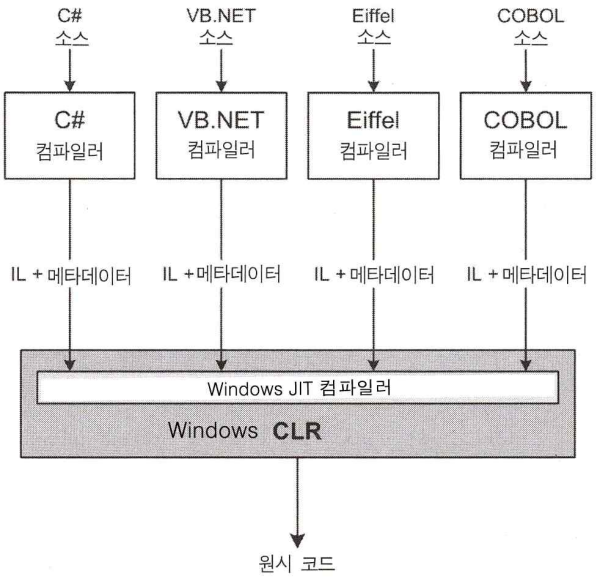
C:\Tmp>ildasm My.exe



Main 함수를 두 번 클릭하면 다음과 같이 IL 명령어들을 볼 수 있습니다.



C#으로 작성한 어셈블리와 비주얼 베이직으로 작성한 어셈블리가 IL 명령어 레벨에서는 다르지 않음을 알 수 있습니다. 이는 어떤 언어를 이용하여 작성하든지 상관없이 어셈블리는 동일하다는 것을 의미합니다.



따라서 중간 형태의 언어(IL)를 사용하는 이유는 다음과 같이 정리할 수 있습니다.

- 여러분이 사용하기 편한 언어를 이용하여 똑 같은 C# 응용 프로그램을 작성할 수 있습니다. 언어에 관계없이 생성되는 어셈블리는 같으므로 하나의 프로그램 (어셈블리)을 여러 언어를 이용하여 작성할 수도 있습니다.

- .NET 어셈블리는 운영체제 혹은 플랫폼에 관계없이 실행됩니다. .NET 런타임만 설치되어 있으면 윈도우든 UNIX든 플랫폼이든 관계없이 IL

코드를 실행할 수 있으므로 어셈블리는 플랫폼에 독립적입니다.

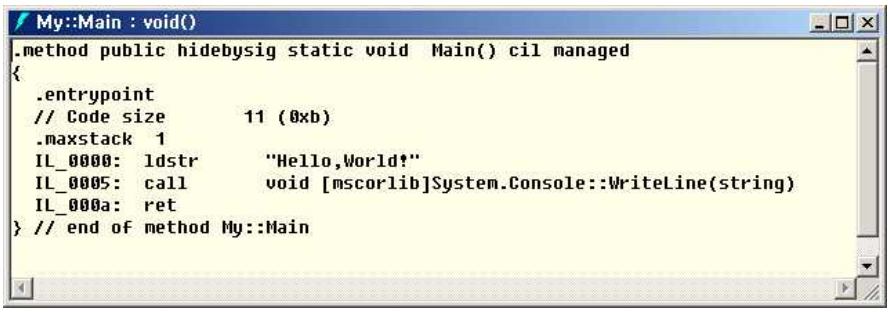
CLR(Common Language Runtime, 혹은 줄여서 런타임)은 여러분이 어셈블리를 실행할 경우 다양한 언어로 만들어질 수 있는 어셈블리를 하드 디스크에서 찾아 메모리로 로드하고 실행합니다. 예를 들어 C#으로 작성된 어셈블리를 실행하면 런타임이 이를 찾아 메모리로 로드한 후 Main 함수를 찾아 실행합니다. 또한 어셈블리의 메타데이터에 있는 정보를 이용하여 실행에 필요한 다른 어셈블리 및 모듈들을 로드하기도 합니다.

CLR의 구성 요소 중의 하나인 JIT(Just-In-Time) 컴파일러는 여러분이 my.exe를 실행하면 실행 바로 전에 my.exe 어셈블리를 해당 플랫폼(가령 윈도우 운영체제)에서 실행되는 exe 파일(가령 hwp.exe)로 just-in-time(제때에, 바로) 컴파일한 후 실행합니다. 즉, 어셈블리를 실행하면 중간 코드를 한 줄 한 줄 인터프리트 방식으로 실행하는 것이 아니라 해당 플랫폼에 맞는 원시(native, cpu-specific) 코드를 바로 생성한 후 일괄적으로 실행하기 때문에 어셈블리 실행 속도가 상당히 빠릅니다.

또한 런타임은 닷넷 응용(어셈블리)에서 동적 할당된 객체를 자동으로 제거해줍니다. 따라서 여러분은 .NET의 어떤 언어를 이용하여 프로그램을 작성할 경우 동적 할당된 객체를 직접 제거하는 코드를 작성할 필요가 없습니다. 런타임의 이러한 기능을 쓰레기 수집(garbage collection)이라고 합니다. 이처럼 .NET에 의해 객체가 제거되고 관리되는 코드를 관리 코드(managed code)라고 합니다. 앞서 작성했던 my.exe 어셈블리는 관리 코드입니다. 이에 반해 기존의 윈도우의 실행 파일, 가령 워드 프로그램이나 아래아 한글 프로그램 등을 관리되지 않는, 혹은 비관리 코드라고 합니다.

C#, Visual C++, Visual Basic 등과 같이 서로 다른 언어를 이용하여 개발한 어셈블리들이 하나의 공통 실행 환경(Common Language Runtime)에서 실행

될 수 있는 이유는 어셈블리가 모든 언어에서 공통적으로 사용하는 기본 클래스 라이브러리(Base Class Libraries)라는 것을 이용하기 때문입니다. 즉, C#을 이용하든 Visual Basic을 이용하든 컴파일할 경우 생성되는 어셈블리는 기본 클래스 라이브러리를 이용합니다. 앞서 설명했지만, 이는 ildasm 도구를 이용하면 쉽게 확인할 수 있습니다.



위 그림의 경우 기본 클래스 라이브러리에 있는 Console 클래스를 이용하여 문자열을 출력하고 있음을 볼 수 있습니다. 이때 기본 클래스 라이브러리는 C:\Windows\Framework 폴더에 있는 mscorlib.dll 파일에 작성되어 있습니다. 이외에도 많은 클래스들이 이 어셈블리에 작성되어 있습니다.

기본 클래스 라이브러리를 구성하는 네임스페이스 중 가장 상위의 네임스페이스는 System이고, 가장 상위의 기반(베이스) 클래스는 Object 클래스입니다. 자료형에는 누가 만들었느냐에 따라 두 가지로 구분할 수 있습니다. 여러분이 직접 만든(user-defined) 것과 기존에 이미 만들어진(built-in) 것으로 구분할 수 있습니다. 가령 여러분이 직접 만든 클래스나 구조체 등을 사용자 정의 자료형이라고 하고, .NET에 이미 만들어져 있는(built-in) 자료형을 기본(혹은 built-in) 자료형이라고 합니다. 기본(built-in) 자료형은 앞서 살펴본 mscorlib.dll 어셈블리 파일에 정의되어 있습니다.

- 기본(built-in) 자료형
- 사용자 정의(user-defined) 자료형

자료형을 다른 방법으로도 분류할 수 있습니다. 하나는 값(value) 자료형이고 다른 하나는 참조(reference) 자료형입니다. 값 자료형은 선언하자마자 값을 저장할 수 있는 객체가 바로 만들어지는 자료형이고, 참조 자료형은 단지 참조만 만들어지고 객체는 만들어지지 않는 자료형이다.

앞서 설명한대로 값 자료형으로 정의된 객체(혹은 변수)는 정의하자마자 바로 객체가 만들어집니다. 함수가 실행될 때 만들어지고 함수가 끝날 때 사라지는데, 이를 효과적으로 구현하기 위해 스택에 변수를 생성합니다. 참조 자료형은 객체가 만들어지지 않고 단지 참조만 만들어지기 때문에 객체에 값을 저장하거나 멤버 함수를 호출하려면 반드시 new 명령어로 객체를 실행 시간에 만들어야 합니다.

```
int a;
Point gildong = new Point();
```

한편, 참조 자료형으로 만든 객체는 .NET의 런타임이 쓰레기 수집을 통해 불필요할 경우 스스로 제거할 수 있도록 하기 위하여 메모리 영역 중 힙(heap)이라는 영역에 만들어집니다. 다음은 값 자료형과 참조 자료형에 속하는 자료형들을 기술한 것입니다.

- 값(value) 자료형 : 기본 자료형 중 Object와 String 자료형을 제외한 모든 자료형, 그리고 구조체
- 참조(reference) 자료형 : 모든 사용자 정의 자료형과 기본 자료형 중 Object와 String 자료형

.NET built-in 타입

.NET	C#	VB .NET	IL	값 또는 참조
System.Boolean	bool	Boolean	bool	값
System.Byte	byte	Byte	unsigned int8	값
System.Char	char	Char	char	값
System.DateTime	–	Date	–	값
System.Decimal	decimal	Decimal	–	값
System.Double	double	Double	float64	값
System.Int16	short	Short	int16	값
System.Int32	int	Integer	int32	값
System.Int64	long	Long	int64	값
System.object	object	object	object	참조
System.SByte	sbyte	–	int8	값
System.Single	float	Single	float32	값
System.String	string	String	string	참조
System.UInt16	ushort	–	unsigned int16	값
System.UInt32	uint	–	unsigned int32	값
System.UInt64	ulong	–	unsigned int64	값

다음은 값 자료형을 사용하는 예입니다.

```
class My
{
    public static void Main()
    {
        int a; //값
        a = 2;
        System.Console.WriteLine(a);
    }
}
```

다음은 참조(reference) 자료형인 My 클래스를 선언한 후 사용하는 예입니다.

```
public class My
{
       public void say()
       {
              System.Console.WriteLine("Hello,World!");
       }
}

class My
{
    public static void Main()
    {
        int a; //값
        a = 2;
        System.Console.WriteLine(a);

        My gildong;   // 레퍼런스 선언
        gildong = new My();   // 힙에 객체 정의
        gildong.Study();
    }
}
```

요약 정리하겠습니다. 향후 다음 내용에 익숙해지도록 하기 바랍니다.

- 어셈블리(assembly)
- 중간 언어(IL) 코드
- 메타데이터
- 매니페스트(manifest)

- ildasm 도구
- CLR(Common Language Runtime) 혹은 런타임
- JIT 컴파일러
- 쓰레기 수집
- 관리 코드
- 기본 클래스 라이브러리(Base Class Library)
- mscorlib.dll 어셈블리
- .NET 자료형

제6장
델리게이트와 닷넷 프레임워크

이 장에서는 닷넷에서 이벤트를 처리하는데 있어서 아주 중요한 델리게이트 개념을 공부합니다. 또한 이를 이용하여 특정 이벤트가 발생할 경우 실행되는 핸들러 함수를 실행하는 원리에 대하여 학습합니다. 특히 이를 기반으로 하여 닷넷 프레임워크 개념 등에 대하여 다룹니다.

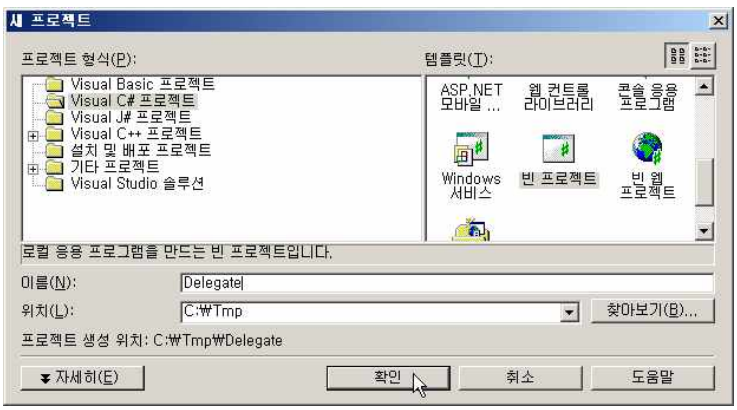
Step1 델리게이트의 의미

델리게이트(delegate, 대리인)란 무엇일까요? 생소하게 들리지만 대단한 것이 아닙니다. 델리게이트란 그저 '대신 사용할 수 있는 것'을 의미합니다. 무엇을 대신 사용하느냐 하면, C#에서는 '함수'를 대신해서 사용하는 것을 의미합니다. 결국 여러분은 이렇게 이해하고 있으면 됩니다. '아하! 델리게이트는 함수 대신에 사용하는 것이구나'.

이 델리게이트는 비록 단순한 아이디어이긴 하지만 C#의 이벤트 모델을 이해하기 위해서는 반드시 이해하고 넘어가야 하는 내용입니다. 이 절에서는 이러한 델리게이트가 C# 프로그램에서 이벤트 처리시 어떻게 이용되는지에 대해 살펴보도록 하겠습니다.

Step2 Delegate 프로젝트 생성하기

우선 파일 > 새로 만들기 > 프로젝트 메뉴 항목을 선택하여 Delegate 프로젝트를 생성합니다. 이때 아래의 새 프로젝트 대화 상자에서 오른쪽의 템플릿으로는 '빈 프로젝트'를 선택합니다.



프로젝트 형식 : Visual C# 프로젝트
템플릿 : 빈 프로젝트
이름 : Delegate

그런 다음, **프로젝트(P) > 클래스 추가(C)** 메뉴 항목을 선택하여 새로운 클래스를 추가합니다. 아래와 같이 클래스 이름에 Delegate를 입력하여 열기 버튼을 누르면 Delegate 클래스가 자동으로 작성됩니다.

Step3 코드 편집 및 작성

열기 버튼을 눌러 Delegate 클래스를 추가하면 기본적으로 아래와 같은 코드가 자동으로 작성됩니다.

```
using System;
```

```
namespace Delegate
{
	/// <summary>
	/// Delegate에 대한 요약 설명입니다.
	/// </summary>
	public class Delegate
	{
		public Delegate()
		{
			//
			// TODO: 여기에 생성자 논리를 추가합니다.
			//
		}
	}
}
```

자동으로 작성된 코드 중 다음과 같이 강조된 부분을 제거하겠습니다. 이 코드는 제거해도 무방한 코드입니다.

```
using System;

namespace Delegate
{
	/// <summary>
	/// Delegate에 대한 요약 설명입니다.
	/// </summary>
	public class Delegate
	{
		public Delegate()
		{
			//
			// TODO: 여기에 생성자 논리를 추가합니다.
```

```
                    //
                }
        }
}
```

다음은 필요 없는 코드를 제거한 모습입니다.

```
public class Delegate
{

}
```

엔트리 포인트(진입점, 운영체제가 호출하는 부분)에 해당되는 Main 함수가 반드시 존재해야 하므로 다음과 같이 Main 함수를 작성합니다.

```
public class Delegate
{
        public static void Main()
        {
                System.Console.WriteLine("Hello,World!");
        }
}
```

컴파일한 후 실행한 모습은 다음과 같습니다.

위 도스 창을 일컬어 '콘솔'이라고 합니다. 객체지향 관점에서 보면 위 윈도우는 콘솔 객체의 얼굴에 해당되겠지요. 따라서 Console.WriteLine (Hello,World!) 문장은 콘솔 객체에게 Hello,World!라는 문자열을 표시하도록 하는 것으로 생각할 수 있습니다.

문자열을 출력하고자 할 때마다 **매번 객체를 만들어** 객체의 멤버 함수 WriteLine을 호출하는 것은 귀찮습니다. 대신 **객체를 정의하지 않고서도** WriteLine 함수를 호출할 수 있도록 하기 위하여 WriteLine 함수는 정적 멤버 함수로 작성되어 있고, 따라서 위와 같이 클래스 Console을 직접 이용하여 멤버 함수 WriteLine을 호출하고 있습니다.

문자열을 출력할 때마다 매번 System 이라는 네임스페이스를 기술하는 것은 귀찮을 듯합니다. 다음과 같이 작성해도 됩니다.

```
using System;

public class Delegate
{
	public static void Main()
	{
		Console.WriteLine("Hello,World!");
```

```
        }
}
```

네임스페이스는 1)관련된 클래스들을 함께 관리할 수 있게 해 주고, 2)두 개 이상의 클래스들 간에 클래스 이름이 충돌되는 것을 막는 역할을 한다. 네임스페이스로 선언되어 있는 클래스들은 C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705 폴더에 존재한다. 일반적으로 대부분의 클래스들은 mscorlib.dll 이라는 라이브러리에 존재한다.

델리게이트에 대해 공부하기 위해서 콘솔에 표시하는 문자열을 다음과 같이 바꿔봅니다.

```
using System;

public class Delegate
{
        public static void Main()
        {
                Console.WriteLine("클릭!");
        }
}
```

코드 추상화와 함수

문자열을 표시하는 코드를 다음과 같이 xxx라는 함수로 추상화해 보겠습니다.

```
using System;
```

```
public class Delegate
{
	public void xxx()
	{
		Console.WriteLine("클릭!");
	}

	public static void Main()
	{
		xxx();
	}
}
```

정적(static) 함수 Main은 객체를 정의하지 않아도 운영체제가 실행하는 엔트리 포인트입니다. 하지만 xxx 함수는 일반 멤버 함수로서 객체를 정의해야만 호출할 수 있는 함수입니다. 따라서 위와 같이 직접 함수 xxx를 호출하도록 코드를 작성할 수 없으며, 다음과 같이 객체를 정의한 후 멤버 함수를 호출합니다.

```
using System;

public class Delegate
{
	public void xxx()
	{
		Console.WriteLine("클릭!");
	}

	public static void Main()
	{
		Delegate gildong = new Delegate();
		gildong.xxx();
```

```
        }
}
```

데이터 추상화 클래스

이제 Base라는 새로운 클래스를 만든 후 xxx 함수를 그 안으로 옮겨 보겠습니다. 다음은 새로운 클래스를 선언한 모습입니다.

```
using System;

public class Base
{
}

public class Delegate
{
        public void xxx()
        {
                Console.WriteLine("클릭!");
        }

        public static void Main()
        {
                Delegate gildong = new Delegate();
                gildong.xxx();
        }
}
```

Base 클래스로 xxx 멤버 함수를 옮깁니다.

```
using System;
```

```
public class Base
{
	public void xxx()
	{
		Console.WriteLine("클릭!");
	}
}

public class Delegate
{
	public static void Main()
	{
		Delegate gildong = new Delegate();
		gildong.xxx();
	}
}
```

Base라는 새로운 클래스가 작성되었습니다. 클래스는 객체 만들라고 있는 것입니다. Main 함수에서 객체를 만들어 멤버 함수를 호출해 보겠습니다.

```
using System;

public class Base
{
	public void xxx()
	{
		Console.WriteLine("클릭!");
	}
}

public class Delegate
{
```

```
        public static void Main()
        {
                Base gildong = new Base();
                gildong.xxx();
        }
}
```

델리게이트를 이용하여 멤버 함수 호출하기

앞서 설명하기를, 함수를 대신해서 사용할 수 있는 것을 델리게이트라고 하였습니다. 그렇다면, 이제 xxx라는 함수를 대신해서 사용할 수 있는 델리게이트에 대해 알아보도록 하겠습니다.

코드가 다음과 같다면 어떨까요?

```
using System;

public class Base
{
        public void xxx()
        {
                Console.WriteLine("클릭!");
        }
}

public class Delegate
{
        public static void Main()
        {
                Base gildong = new Base();
                gildong.Click(); //Click은 xxx 함수의 델리게이트
```

```
        }
}
```

길동 객체의 xxx라는 멤버 함수를 호출했었는데 이제는 Click으로 대신 호출하고 있습니다. 만일 이렇게 하더라도 xxx 멤버 함수가 호출된다면 Click은 xxx 멤버 함수의 델리게이트가 되겠지요.

'함수 대신에 사용할 수 있는 것'을 델리게이트(delegate, 대리자)라고 했는데, 사실 C 언어에도 델리게이트가 존재합니다. C 언어에는 함수 포인터라는 것이 있습니다. 함수 포인터는 함수 대신에 사용하여 함수를 호출할 수 있지요.

예를 들어 아래의 함수는 리턴하는 값이 없고(void) 파라미터를 갖지 않는 함수의 예입니다.

```
void xxx()
{
        printf("Hello,World!");
}
```

예를 들어, 만일 FuncPtr이라는 함수 포인터가 위 xxx 함수를 가리키고 있다면 xxx 함수 대신에 FuncPtr을 기술해도 됩니다. 즉, 다음과 같이 호출해도 xxx 함수가 실행됩니다.

```
FuncPtr();
```

이처럼 함수를 대신해서 사용할 수 있는 함수 포인터 FuncPtr도 델리게이트라고 말할 수 있습니다. 물론 C 언어에서는 그냥 함수 포인터라고 하지 델리게이트라고는 하지 않습니다.

C#에서는 공식적으로 델리게이트라고 부릅니다. C 언어에서의 함수 포인터와 C#에서의 델리게이트는 기능 면에서 분명히 서로 다른 면이 있지만 '대신 사용하는 것'이라는 점에서 개념 상 서로 비슷하다고 볼 수 있습니다.

C 언어에서의 포인터 변수를 델리게이트라고 부를 수 있는데, 객체지향 프로그램에서도 변수로 구현됩니다. 하지만 객체지향 프로그램에서 변수는 곧 객체이므로 C#에서 델리게이트는 결국 객체로 구현됩니다. 변수나 객체나 모두 자료형을 이용해서 만든 인스턴스(instance)이지요?

델리게이트는 객체이므로 객체를 정의하려면 자료형이 있어야 합니다. 가령 Click이라는 델리게이트 객체를 정의하기 위한 자료형이 DelegateClass라고 한다면 다음과 같이 멤버 변수로 정의할 수 있습니다.

```
using System;

public class Base
{
        public DelegateClass Click = null;
        public void xxx()
        {
                Console.WriteLine("클릭!");
        }
}
```

델리게이트를 null 로 초기화하고 있는데, 이는 C 언어에서 포인터를 NULL로 초기화하는 것과 유사합니다. 만일 델리게이트 Click이 멤버 함수 xxx를 대신해서 사용할 수 있는 것이라고 한다면 다음과 같이 코드를 수정해도 무방하겠지요?

```
using System;

public class Base
{
	public DelegateClass Click = null;
	public void xxx()
	{
		Console.WriteLine("클릭!");
	}
}

public class Delegate
{
	public static void Main()
	{
		Base gildong = new Base();
		gildong.Click();
	}
}
```

이제 Click 객체(델리게이트)를 정의할 때 사용하고 있는 DelegateClass 자료형을 선언해야 합니다. 이때, void xxx() 등과 같은 형태의 함수를 대신해서 사용할 수 있는 델리게이트 객체를 정의하려면 다음과 같은 delegate 라는 키워드를 갖는 문법으로 선언합니다.

```
using System;

public class Base
{
	public delegate void DelegateClass();
	public DelegateClass Click = null;
```

```
        public void xxx()
        {
                Console.WriteLine("클릭!");
        }
}

public class Delegate
{
        public static void Main()
        {
                Base gildong = new Base();
                gildong.Click();
        }
}
```

그렇다면 아래와 같은 형태의 함수를 대신해서 사용할 수 있는 델리게이트 자료형은 어떻게 선언할 수 있을까요?

```
public int Add(int a, int b)
{
        return a + b;
}
```

쉽습니다. 그냥 다음과 같이 리턴값과 파라미터의 자료형만 바꾸면 됩니다.

```
public delegate int DelegateClass(int a, int b);
```

문법이니 암기하여 따를 수 밖에 없겠지요?

델리게이트가 제대로 사용되기 위해서는 마지막으로 한 가지 일이 남았습니다. 즉, Click 델리게이트를 이용할 경우 어떤 함수가 대신 호출되는지에 대

한 정보를 아직 기술하지 않았습니다. Click이 xxx 함수 델리게이트로 지정하기 위한 코드는 다음과 같습니다. Base 클래스 객체가 만들어질 때 지정되도록 하기 위하여 생성자 함수에서 작성하였습니다.

```
using System;

public class Base
{
	public delegate void DelegateClass();
	public DelegateClass Click = null;

	public Base()
	{
		Click = new DelegateClass(xxx);
	}

	public void xxx()
	{
		Console.WriteLine("클릭!");
	}
}

public class Delegate
{
	public static void Main()
	{
		Base gildong = new Base();
		gildong.Click();
	}
}
```

이제 Click은 xxx 함수 대신에 사용할 수 있는 것, 즉, 델리게이트 (객체)가

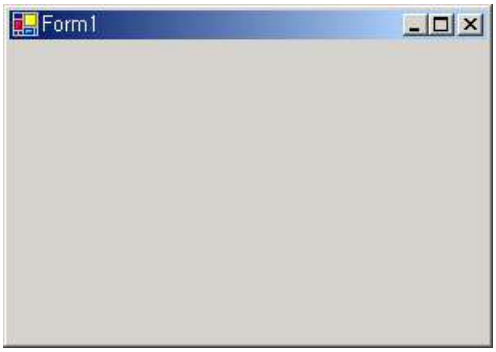
되었으며, 다음과 같이 표현할 수 있습니다.

Click	델리게이트, 델리게이트 객체, 델리게이트 인스턴스
DelegateClass	델리게이트 형(type)

이제까지 작성한 프로그램을 컴파일한 후 실행한 모습은 다음과 같습니다.

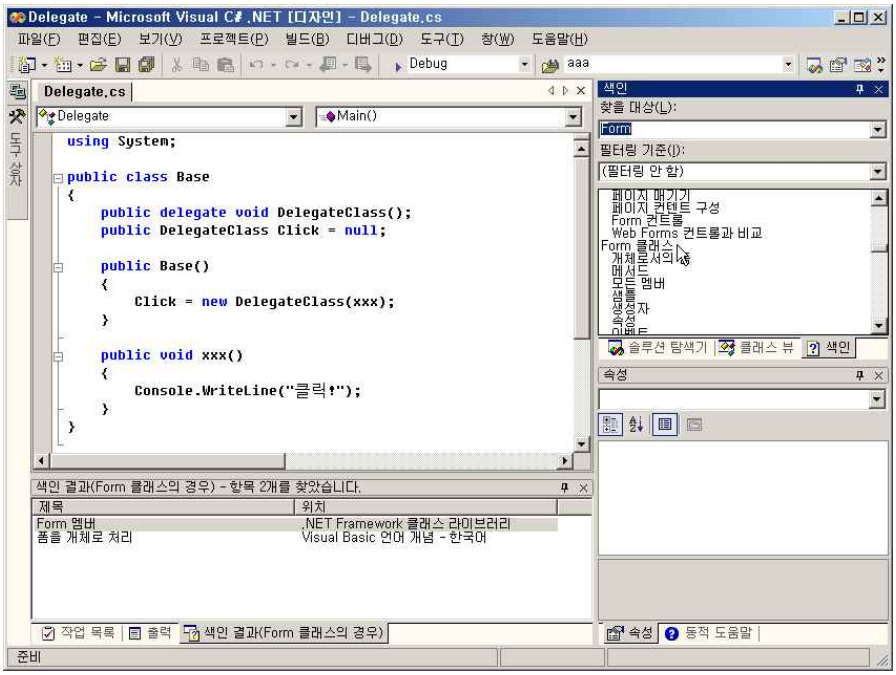
Step7 델리게이트와 이벤트

만일 어떤 프로그램을 실행하였더니 아래와 같은 폼 윈도우가 표시되었다고 가정해 보겠습니다.

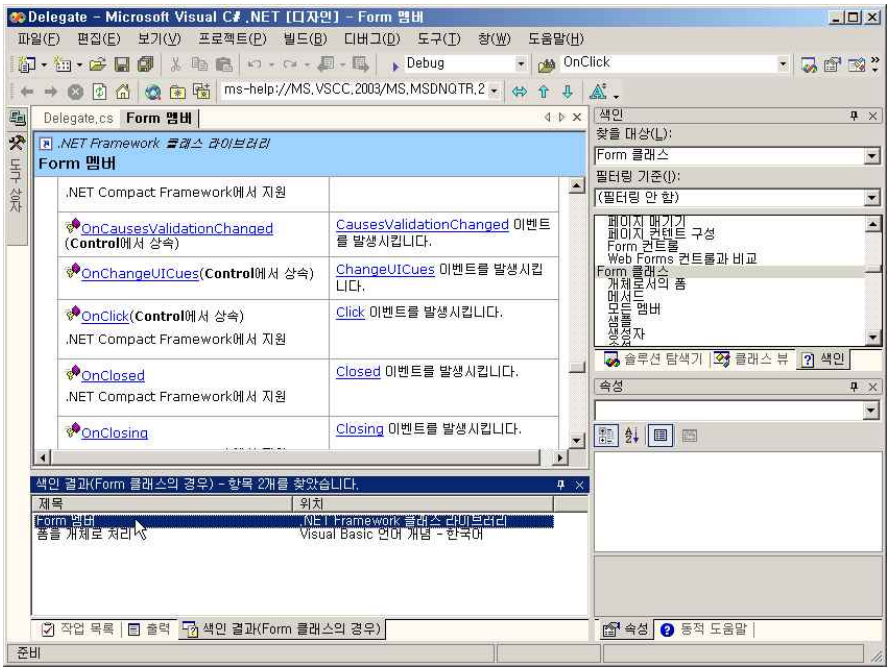


여러분이 위의 폼 윈도우를 클릭하면 '윈도우 클릭' 이벤트(event)가 발생합니다. 중간 과정을 생략하면, 여러분이 마우스로 위의 폼 윈도우를 클릭하면 결과적으로 .NET의 런타임(runtime)은 위에 표시된 폼 윈도우 객체의 OnClick이라는 가상 멤버 함수를 호출합니다. 폼 윈도우 객체의 OnClick이라는 가상 함수가 있느냐고요?

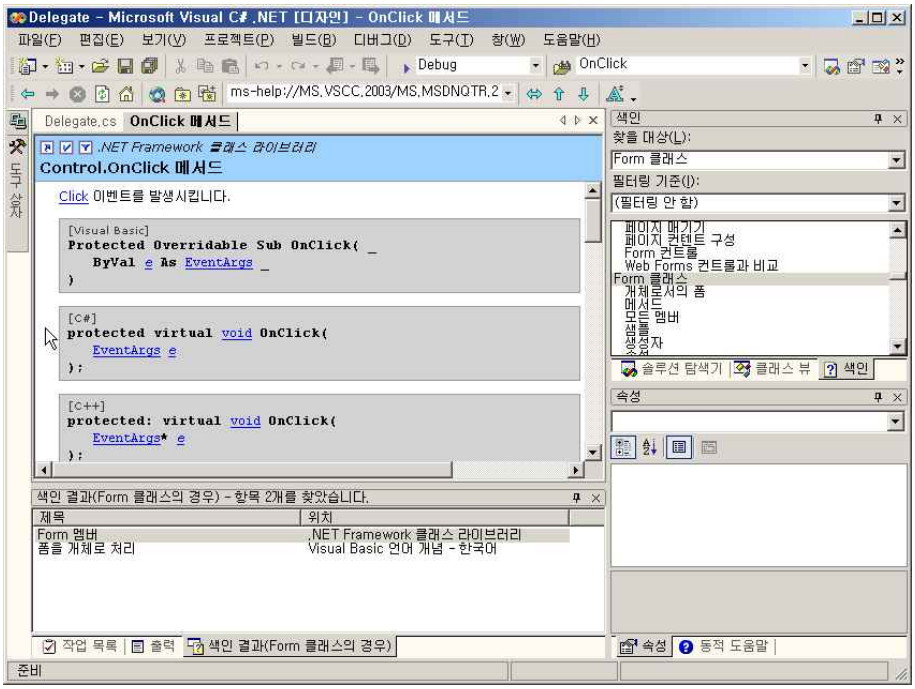
위와 같이 생긴 모든 폼 윈도우는 Form이라는 클래스로부터 상속을 받습니다. Form 클래스에 대한 도움말을 찾아보기 위해 다음과 같이 비주얼 스튜디오의 색인 탭에 Form을 입력합니다. 참고로 도움말(H) > 색인(I)... 메뉴 항목을 선택하면 됩니다.



그런 다음 오른쪽에 있는 색인 탭의 뷰에서 마우스로 **Form 클래스**를 두 번 클릭한 후 아래 부분에 있는 색인 결과 탭에서 '**Form 멤버**' 부분을 마우스로 두 번 클릭합니다. 그러면 아래와 같이 도움말이 표시됩니다.



Ctrl + F 키를 눌러 **OnClick 멤버 함수**가 있는 부분을 찾습니다. 그리고 OnClick을 마우스로 클릭하여 도움말을 봅니다. 만일 .NET 도움말이 설치되어 있지 않은 경우에는 MSDN 시디를 이용하여 먼저 설치해야 합니다.



중간 부분을 보면 다음과 같이 OnClick 멤버 함수의 원형이 있으며, virtual 키워드에 의해 이 함수가 가상 함수임을 알 수 있습니다.

```
[C#]
protected virtual void OnClick(EventArgs e);
```

아무튼, 여러분이 마우스로 폼 윈도우를 클릭하면 **폼 윈도우 객체의 OnClick 가상 멤버 함수가 자동으로 실행된다는 사실**은 절대 잊어서는 안 됩니다. 물론 .NET 런타임이 호출해 줍니다.

그런데 재미있게도 OnClick 함수내부는 다음과 같이 생겼습니다.

```
public virtual void OnClick()
```

```
{
      if(Click != null)
            Click();
}
```

델리게이트 Click이 null이 아니면, 즉, Click 델리게이트에 함수가 설정되어 있으면 그 함수를 실행하라는 의미입니다.

Form 클래스가 OnClick 가상 멤버 함수를 가지고 있음을 확인하였습니다. 마찬가지로 우리가 작성한 Base 클래스도 OnClick 멤버 함수를 가지도록 흉내를 내 보겠습니다.

```
using System;

public class Base
{
      public delegate void DelegateClass();
      public DelegateClass Click = null;

      public Base()
      {
            Click = new DelegateClass(xxx);
      }

      public void xxx()
      {
            Console.WriteLine("클릭!");
      }

      public void OnClick()
      {
```

```
            if(Click != null)
                  Click();
      }
}

public class Delegate
{
      public static void Main()
      {
            Base gildong = new Base();
            gildong.OnClick();
      }
}
```

Main 함수에서 OnClick 멤버 함수를 호출하도록 코드를 수정하였습니다.

이번 Step에서 공부한 내용을 정리해 보겠습니다. 다음 문장을 잘 음미해 보기 바랍니다.

> 폼 윈도우를 마우스로 클릭하면 이벤트가 발생하고, 이때 .NET 런타임에 의해 OnClick 가상 멤버 함수가 자동으로 실행되고, OnClick 가상 멤버 함수는 델리게이트 (객체) Click이 null이 아니라면 설정된 멤버 함수 xxx(핸들러 함수)를 호출한다.

결국, 이벤트가 발생할 경우 xxx 멤버 함수가 실행되므로 xxx 멤버 함수는 '이벤트가 발생할 경우 어떤 것을 처리(handle)할 수 있다'라는 의미로 xxx 함수는 이벤트 핸들러(handler) 함수라고 합니다. 또한 이벤트가 발생하면 결과적으로 Click 델리게이트가 호출되므로 Click 자체를 '이벤트'라고도 합니다. 결국 Click을 칭하는 말이 하나 더 생긴 셈이네요.

Click	델리게이트, 델리게이트 객체, 델리게이트 인스턴스, 이벤트
DelegateClass	델리게이트 형(type)
xxx	핸들러 함수
OnClick	이벤트 Click을 fire 하는 가상 함수

결국 위의 표를 보면, 델리게이트 객체는 이벤트임을 알 수 있습니다. OnClick 멤버 함수는 Click 이벤트가 수행되도록 하므로 'OnClick은 Click 이벤트를 시작시키는(fire) 함수다'라고 합니다.

앞으로 다음과 같은 코드를 보면 'Click 이벤트에 xxx 핸들러 함수를 연결한다'라고 생각하기 바랍니다.

```
Click = new DelegateClass(xxx);
```

Click 이벤트는 핸들러 함수를 대신 호출하기 위해 만든 **델리게이트 객체**이므로 이를 정의하기 위한 자료형도 DelegateClass 라고 하기 보다는 핸들러(handler) 함수를 호출하기 위한 자료형임을 의미하기 위하여 MyHandler 등과 같이 하는 것이 그럴 듯 해 보입니다.

```
using System;

public class Base
{
	public delegate void MyHandler();
	public MyHandler Click = null;

	public Base()
	{
		Click = new MyHandler(xxx);
	}
```

```
        public void xxx()
        {
                Console.WriteLine("클릭!");
        }

        public void OnClick()
        {
                if(Click != null)
                        Click();
        }
}

public class Delegate
{
        public static void Main()
        {
                Base gildong = new Base();
                gildong.OnClick();
        }
}
```

델리게이트가 필요한 이유

그렇다면 이러한 델리게이트는 왜 존재하는 것일까요? 단순히 델리게이트가 뭐다라는 것을 아는 것은 그리 중요하지 않습니다. 왜 필요한 지를 이해하는 것이 중요합니다. 이는 C#의 이벤트 모델을 이해하기 위해서도 반드시 필요합니다.

Click 델리게이트는 다음과 같은 자료형을 이용하여 생성된 객체입니다.

```
public delegate void MyHandler();
```

앞서 살펴보았던 xxx 멤버 함수처럼, 리턴값이 없고(void) 파라미터가 없는 함수를 대신해서 사용할 수 있습니다. 함수 이름에 상관없이 말입니다. 가령 xxx 라는 함수명을 ppp라고 바꿔도 문제없이 사용할 수 있습니다. 정리하자면 핸들러 함수 이름을 어떻게 바꾸더라도 상관이 없다는 뜻이고, 달리 해석하면 핸들러 함수 이름이 결정되지 않아도 델리게이트로 호출할 수 있다는 뜻입니다.

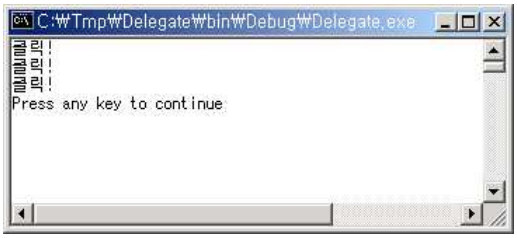
C#이든 자바든, 아니면 Visual C++든 핸들러 함수(앞의 xxx)는 사용자가 작성합니다. 함수 이름도 프로그래머가 결정하겠지요. 그러면 이런 문제를 생각해볼 수 있습니다. 앞서, 이벤트가 발생할 경우 자동으로 실행되는 함수가 핸들러 함수라고 공부했는데, 핸들러 함수 이름이 고정되어(정해져) 있지 않았는데 어떻게 자동으로 실행되도록 할 수 있을까요? 바로 델리게이트가 답입니다. 델리게이트만 있으면, 앞서 설명한 것처럼, 함수 이름이 고정되어(정해져) 있지 않더라도 문제없습니다. '왜 델리게이트가 필요하지?'라고 물어보면 이제 대답할 수 있겠지요? 즉, 고정되어(정해져) 있지 않은 핸들러 함수를 호출하기 위해서입니다.

델리게이트를 이용하면 또 다른 재미있는 결과를 얻을 수 있습니다. 코드를 다음과 같이 작성해 봅니다. 특이한 것은 = 연산자 대신에 += 연산자를 사용하고 있습니다.

```
using System;

public class Base
{
	public delegate void MyHandler();
	public MyHandler Click = null;
```

```
        public Base()
        {
                Click += new MyHandler(xxx);
                Click += new MyHandler(xxx);
                Click += new MyHandler(xxx);
        }

        public void xxx()
        {
                Console.WriteLine("클릭!");
        }

        public void OnClick()
        {
                if(Click != null)
                        Click();
        }
}

public class Delegate
{
        public static void Main()
        {
                Base gildong = new Base();
                gildong.OnClick();
        }
}
```

OnClick 멤버 함수를 보세요. OnClick 멤버 함수에서는 비록 Click 델리게이트를 이용하여 한번만 호출하고 있지만 Click 델리게이트에 xxx 함수를 세 번 설정한 결과 다음과 같이 세 번이나 호출됨을 알 수 있습니다.



결국 하나의 델리게이트를 이용하여 여러 멤버 함수를 호출하는데 대신 사용할 수도 있습니다. 이처럼 C#의 델리게이트는 여러 함수를 실행할 수도 있다는 점도 C 언어의 함수 포인터와 다릅니다.

또 다른 멤버 함수 aaa를 작성한 후 이 함수도 호출하도록 설정할 수도 있습니다.

```
using System;

public class Base
{
	public delegate void MyHandler();
	public MyHandler Click = null;

	public Base()
	{
		Click += new MyHandler(xxx);
		Click += new MyHandler(xxx);
		Click += new MyHandler(xxx);
		Click += new MyHandler(aaa);
	}

	public void xxx()
	{
```

```
            Console.WriteLine("클릭!");
        }

        public void aaa()
        {
            Console.WriteLine("Hello, World!");
        }

        public void OnClick()
        {
            if(Click != null)
                Click();
        }
}

public class Delegate
{
        public static void Main()
        {
            Base gildong = new Base();
            gildong.OnClick();
        }
}
```

다음 단계의 공부를 위하여 Base 클래스에 추가적으로 작성한 코드를 제거하겠습니다. aaa 멤버 함수를 제거하고 생성자 함수 내에 있는 코드를 정리합니다.

```
public class Base
{
        public delegate void MyHandler();
        public MyHandler Click = null;
```

```
        public Base()
        {
                Click += new MyHandler(xxx);
        }

        public void xxx()
        {
                Console.WriteLine("클릭!");
        }

        //여기에 있던 aaa 멤버 함수를 제거하였음

        public void OnClick()
        {
                if(Click != null)
                        Click();
        }
}
```

비하인드 코드, 비하인드 클래스

다음과 같이 Base 클래스에서 상속 받아 Derived라는 새로운 클래스를 작성하여 보겠습니다.

```
using System;
public class Base
{
	public delegate void MyHandler();
	public MyHandler Click = null;

	public Base()
	{
		Click += new MyHandler(xxx);
	}
	public void xxx()
	{
		Console.WriteLine("클릭!");
	}

	public void OnClick()
	{
		if(Click != null)
			Click();
	}
}
```

```
public class Derived : Base
{


}
```

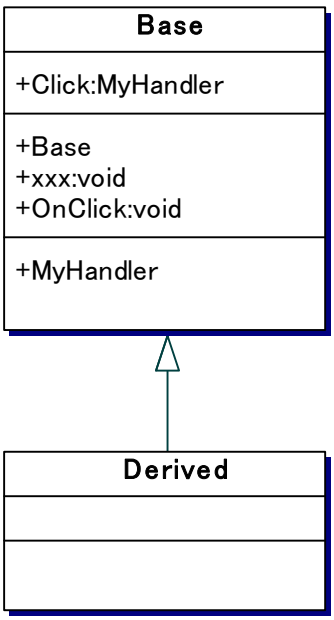
Base 클래스 안에 작성된 코드가 고스란히 Derived 클래스 안으로 들어옵니다(재사용, 상속). 이렇게 함으로써 어차피 Base 클래스에 작성된 코드가 Derived 클래스에 있는 것과 다를 바 없으므로 다음과 같이 Derived 클래스를 이용하여 객체를 정의하여도 문제가 없습니다.

```
public class Delegate
{
        public static void Main()
        {
                Base gildong = new Derived();
                gildong.OnClick();
        }
}
```

새로 만든 Derived '클래스는 객체 만들라고 있는 것'이므로 위와 같이 작성하는 것이 당연해 보입니다.

멤버 함수와 멤버 변수는 이제 Base 클래스에 있건, 혹은 Derived 클래스에 있건 상관없습니다. 현재로서는 모든 코드가 Base 클래스에 있습니다만...

앞의 코드를 보면 Main 함수에서 Derived 클래스를 이용하여 객체를 정의하였습니다. Derived 클래스를 보면 아무런 멤버도 없는 것 같지만 사실은 Base 클래스로부터 여러 멤버를 상속 받아 재사용합니다. 개념적으로 볼 때 Derived 클래스 뒤쪽(behind)에 Base 클래스가 있는 모양이며, 따라서 Base를 비하인드(behind) 클래스, 혹은 비하인드 코드라고 부릅니다. 이 개념은 특히 ASP.NET에서 주로 사용됩니다. 반대로 Derived 클래스는 프론트(front, 앞쪽) 클래스, 혹은 프론트 코드라고 할 수 있습니다.



델리게이트의 위력, Base 클래스 라이브러리화하기

델리게이트의 위력을 실감해 보겠습니다. 앞으로 여러분이 어떤 프로그램을 작성하더라도 항상 Base라는 클래스를 작성해야 한다면 Base 클래스를 여러분이 매번 작성하는 것 보다는 차라리 미리 만들어 놓고 라이브러리로 제공하면 참 좋겠지요? 그렇다면 Base 클래스를 라이브러리로 제공하는 클래스라고 가정해 보겠습니다.

하지만 문제가 생겼습니다. xxx라는 핸들러 함수 이름은 여러분과 같은 프로그래머가 직접 결정하는 것이므로 이 핸들러 함수가 Base 클래스 안에 작성되어 있으면 죽었다 깨어나도 Base 클래스는 라이브러리가 될 수 없습니다. 따라서 xxx 핸들러 함수를 Base 클래스 밖으로 끄집어내어서 다른 곳으로 옮겨야 하는데, 어디로 옮겨야 할까요? 바로 프론트 클래스인 Derived로 내려

보냅니다. 다음과 같습니다.

```
using System;

public class Base
{
	public delegate void MyHandler();
	public MyHandler Click = null;

	public Base()
	{
		Click += new MyHandler(xxx);
	}
	//여기에 있던 코드를 프론트 클래스로 옮겼습니다.
	public void OnClick()
	{
		if(Click != null)
			Click();
	}
}

public class Derived : Base
{
	public void xxx()
	{
		Console.WriteLine("클릭!");
	}
}

public class Delegate
{
	public static void Main()
	{
```

```
            Base gildong = new Derived();
            gildong.OnClick();
        }
}
```

xxx 멤버 함수 외에도 xxx와 관련된 코드도 프론트 클래스로 옮깁니다. 비하인드 클래스 Base의 생성자 함수 안을 보면 관련 코드가 있습니다. 이를 옮깁니다.

```
using System;

public class Base
{
        public delegate void MyHandler();
        public MyHandler Click = null;

        public Base()
        {
            //여기에 있던 코드를 프론트 클래스로 옮겼습니다.
        }

        public void OnClick()
        {
            if(Click != null)
                Click();
        }
}

public class Derived : Base
{
        public Derived()
        {
```

```
            Click += new MyHandler(xxx);
        }

        public void xxx()
        {
            Console.WriteLine("클릭!");
        }
}

public class Delegate
{
        public static void Main()
        {
            Base gildong = new Derived();
            gildong.OnClick();
        }
}
```

이제 Base 클래스에는 프로그래머가 마음대로 이름을 정할 수 있는 xxx 핸들러 함수 및 관련 코드는 존재하지 않으며, 따라서 라이브러리가 될 수 있습니다.

가상 함수를 이용한 이벤트 처리

한 가지 더 재미있는 경험을 해 보겠습니다. Base 클래스에 있는 OnClick을 virtual 키워드를 이용하여 가상 함수로 만듭니다.

```
using System;

public class Base
```

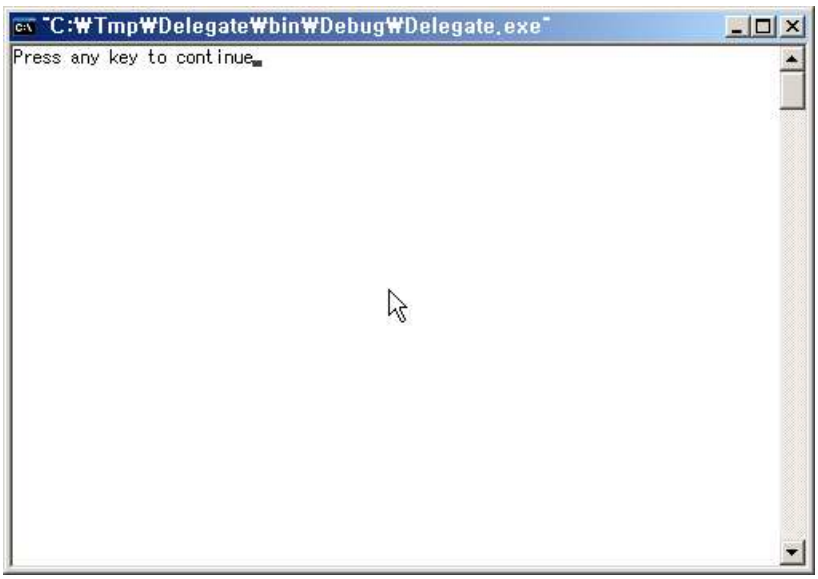
```
{
	public delegate void MyHandler();
	public MyHandler Click = null;

	public Base()
	{
	}

	public virtual void OnClick()
	{
		if(Click != null)
			Click();
	}
}
```

Derived 클래스에서 이 함수를 오버라이딩을 해 봅니다.

```
public class Derived : Base
{
	public Derived()
	{
		Click += new MyHandler(xxx);
	}

	public void xxx()
	{
		Console.WriteLine("클릭!");
	}

	public override void OnClick()
	{
```

```
        }
}
```

프로그램을 컴파일한 후 실행하면 어떤 일이 발생할까요?



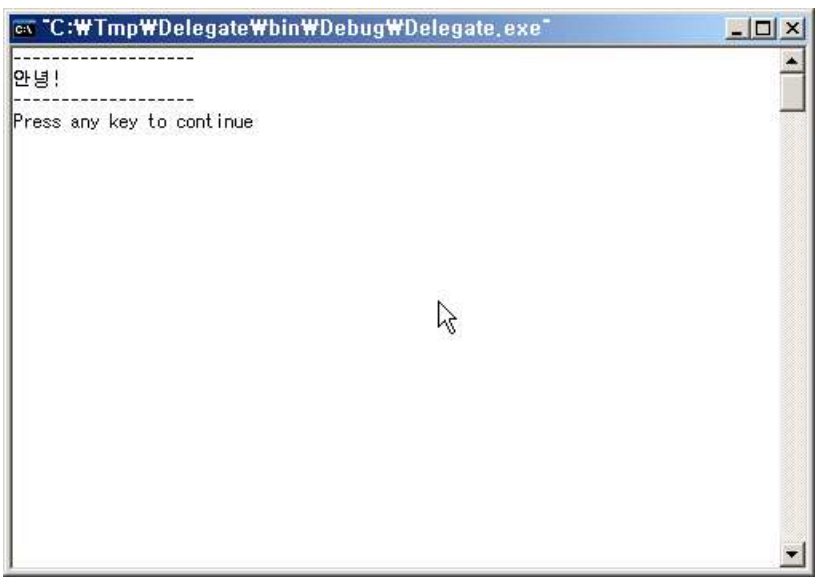
핸들러 함수 xxx가 실행되지 않습니다. 당연한 결과입니다. Base 클래스의 OnClick이 실행되지 않고 여러분이 Derived 클래스에서 오버라이딩한 OnClick 함수가 대신 실행됩니다. 하지만 오버라이딩한 OnClick 함수는 텅 비어있으므로 위와 같이 아무런 문자열도 출력되지 않는 것입니다.

이제 아래와 같이 코드를 입력해 봅니다.

```
public class Derived : Base
{
        public Derived()
        {
```

```
            Click += new MyHandler(xxx);
        }

        public void xxx()
        {
            Console.WriteLine("클릭!");
        }

        public override void OnClick()
        {
            Console.WriteLine("--------------------");
            Console.WriteLine("안녕!");
            Console.WriteLine("--------------------");
        }
}
```

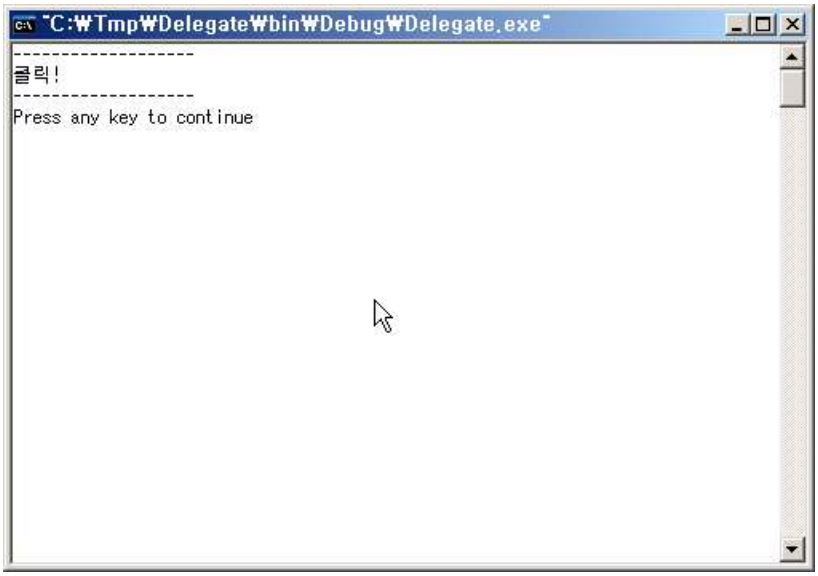
컴파일한 후 실행하면 다음과 같이 표시되겠지요.



가상함수를 다음과 같이 바꾸면 어떻게 될까요?

```
public class Derived : Base
{
	public Derived()
	{
		Click += new MyHandler(xxx);
	}

	public void xxx()
	{
		Console.WriteLine("클릭!");
	}

	public override void OnClick()
	{
		Console.WriteLine("--------------------");
		base.OnClick();
		Console.WriteLine("--------------------");
	}
}
```

Base 클래스의 OnClick 함수를 호출하고 있습니다. 이 함수는 Click 델리게이트를 이용하여 xxx 핸들러 함수를 호출합니다. 실행 결과는 다음과 같습니다.



정리하자면, 이벤트가 발생할 경우 무엇인가를 하고 싶으면 다음과 같이 두 가지 방법 중 하나를 이용할 수 있습니다.

- 핸들러 함수를 작성한다.

  핸들러 함수 xxx를 작성한 후 원하는 코드를 작성한다. 그리고 델리게이트 Click이 xxx 대신에 사용할 수 있도록 지정한다.

- 가상 함수를 오버라이딩한다.

  가상 함수 OnClick을 오버라이딩한 후 원하는 코드를 작성한다.

Step12 코드 다듬기

다음과 같이 클래스 이름을 바꿉니다.

- Base -> Form
- MyHandler -> EventHandler

바꾼 모습은 다음과 같습니다.

```
using System;

public class Form
{
	public delegate void EventHandler();
	public EventHandler Click = null;

	public Form()
	{
	}

	public virtual void OnClick()
	{
		if(Click != null)
			Click();
	}
}

public class Derived : Form
{
	public Derived()
```

```
        {
                Click += new EventHandler(xxx);
        }

        public void xxx()
        {
                Console.WriteLine("클릭!");
        }

        public override void OnClick()
        {
                Console.WriteLine("--------------------");
                base.OnClick();
                Console.WriteLine("--------------------");
        }
}

public class Delegate
{
        public static void Main()
        {
                Form gildong = new Derived();
                gildong.OnClick();
        }
}
```

Application 응용 클래스 작성하기

Main 함수 내부의 코드를 Run 함수로 추상화해 보겠습니다.

```
public class Delegate
{
	public void Run()
	{
		Form gildong = new Derived();
		gildong.OnClick();
	}

	public static void Main()
	{
		Run();
	}
}
```

하지만 Run 함수는 객체를 만들어야 호출할 수 있는 일반 멤버 함수입니다. 따라서 다음과 같이 작성해야 맞습니다.

```
public class Delegate
{
	public void Run()
	{
		Form gildong = new Derived();
		gildong.OnClick();
	}

	public static void Main()
	{
```

```
            Delegate cheolsu = new Delegate();
            cheolsu.Run();
        }
}
```

혹은 객체를 정의하지 않아도 호출할 수 있는 함수인 정적(static) 함수로 선언하면 다음과 같이 객체를 정의하지 않고도 바로 호출할 수 있습니다.

```
public class Delegate
{
        public static void Run()
        {
            Form gildong = new Derived();
            gildong.OnClick();
        }

        public static void Main()
        {
            Run();
        }
}
```

Application이라는 클래스를 작성하겠습니다.

```
public class Application
{

}
```

```
public class Delegate
{
```

```
        public static void Run()
        {
                Form gildong = new Derived();
                gildong.OnClick();
        }

        public static void Main()
        {
                Run();
        }
}
```

Run 멤버 함수를 Application 클래스로 옮깁니다.

```
public class Application
{
        public static void Run()
        {
                Form gildong = new Derived();
                gildong.OnClick();
        }
}

public class Delegate
{
        public static void Main()
        {
                Run();
        }
}
```

클래스는 객체를 만들라고 있는 것입니다. 하지만 Application의 경우 객체를 정의하지 않아도 호출할 수 있는 정적 멤버 함수를 가지고 있으므로 다음과 같이 작성할 수 있습니다.

```
public class Application
{
	public static void Run()
	{
		Form gildong = new Derived();
		gildong.OnClick();
	}
}

public class Delegate
{
	public static void Main()
	{
		Application.Run();
	}
}
```

Application 응용 클래스 라이브러리화

Derived 클래스 내부를 보면 프로그래머가 임의로 정할 수 있는 핸들러 함수 이름 xxx를 포함하고 있으므로 라이브러리화 할 수 없습니다.

```
public class Derived : Form
{
	public Derived()
	{
```

```
            Click += new EventHandler(xxx);
        }

        public void xxx()
        {
            Console.WriteLine("클릭!");
        }

        public override void OnClick()
        {
            Console.WriteLine("--------------------");
            base.OnClick();
            Console.WriteLine("--------------------");
        }
}
```

그러한 클래스 Derived를 Application 클래스에서 사용하고 있으므로 Application 클래스 또한 현재로는 라이브러리화할 수 없습니다.

```
public class Application
{
        public static void Run()
        {
            Form gildong = new Derived();
            gildong.OnClick();
        }
}
```

다음과 같이 코드를 변경하면 Application 클래스를 라이브러리화할 수 있습니다.

```
public class Application
{
```

```
        public static void Run(Form gildong)
        {
                gildong.OnClick();
        }
}

public class Delegate
{
        public static void Main()
        {
                Application.Run(new Derived());
        }
}
```

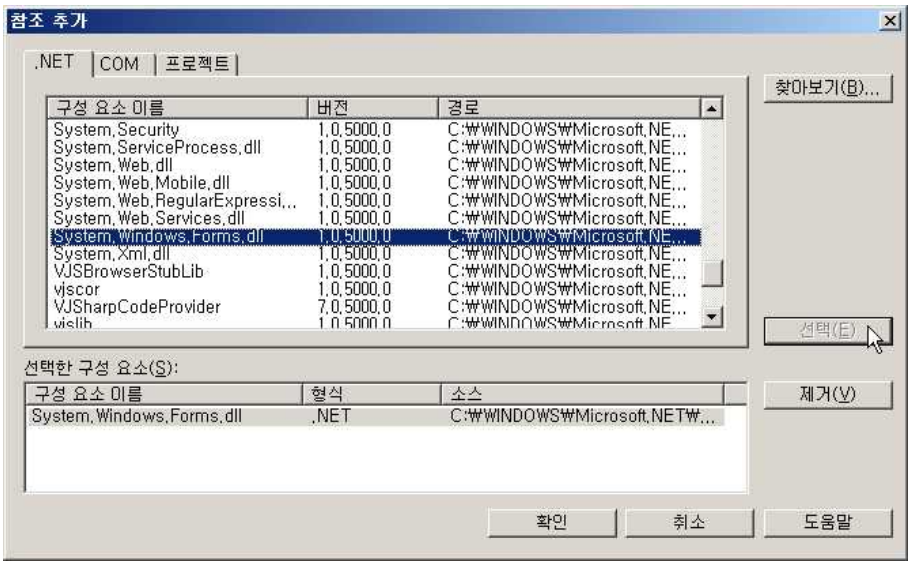
즉, Main 함수에서 Derived 객체를 만들어 넘겨줌으로써 이제 Application 클래스는 이 코드와 무관하게 되었고, 결과적으로 Application 클래스도 이제 라이브러리화할 수 있게 되었습니다.

Step15 C#의 클래스 라이브러리 이용하기

앞서 여러분은 Form 클래스와 Application 클래스를 작성하였으며, 이 클래스들은 라이브러리화할 수 있습니다. 만일 어떤 프로그램을 작성하더라도 Form 클래스와 Application 클래스를 사용하여야 한다면 이 클래스를 라이브러리로 제공하는 것이 바람직하겠지요.

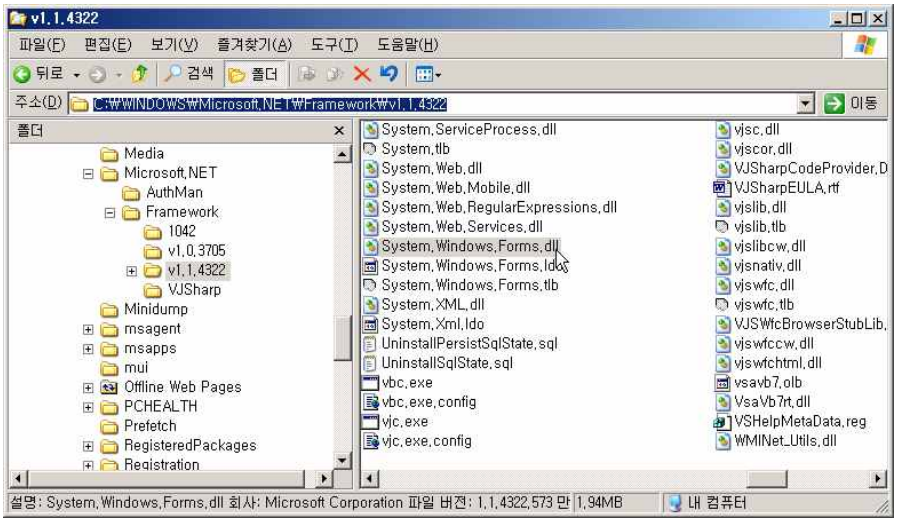
그런데 다행스럽게도 이러한 클래스가 닷넷 기본 라이브러리로 제공되고 있습니다. 따라서 지금까지 했던 것처럼 여러분이 힘들게 작성할 필요가 없겠지요. 이 두 클래스를 사용하고자 한다면 라이브러리(참조)를 추가하면 됩니다.

System.Windows.Forms.dll 라이브러리 파일을 선택한 후 확인 버튼을 누르면 추가됩니다.

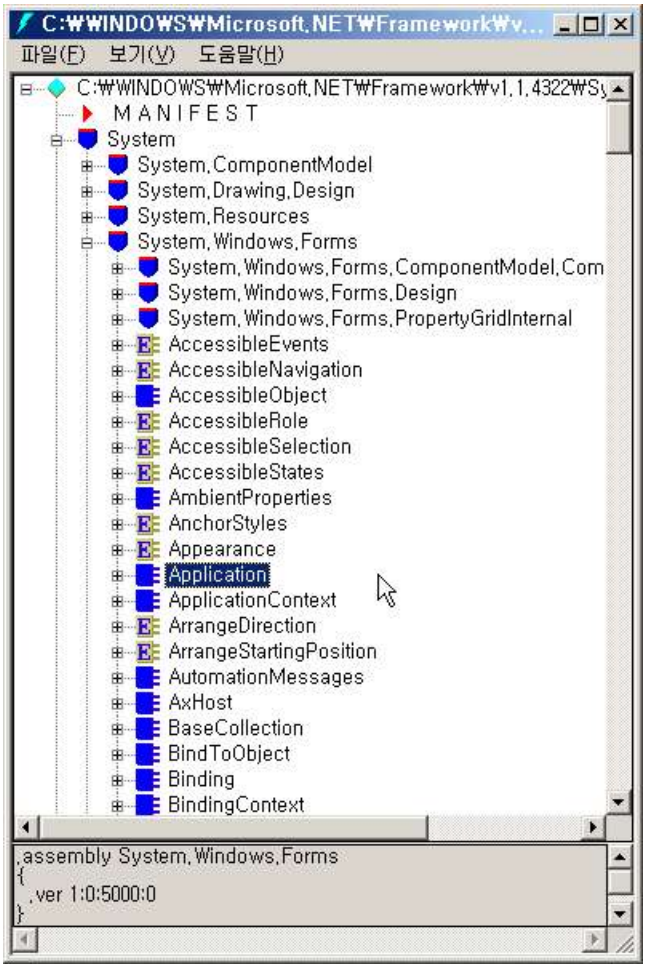


사실 이는 다음과 같이 C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322 폴더에 있는 파일로서 Form 클래스와 Application 클래스, 그리고 기타 여러

클래스들을 가지고 있습니다.



앞서 공부했던 ildasm.exe 도구를 이용하여 내부를 보면 다음과 같이 클래스가 있음을 볼 수 있습니다.



아무튼, Form 클래스와 Application 클래스가 있는 라이브러리를 프로젝트에 추가하였으므로 우리가 작성한 Form 클래스와 Application 클래스는 제거해도 됩니다.

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

```
public class Derived : Form
{
	public Derived()
	{
		Click += new EventHandler(xxx);
	}

	public void xxx()
	{
		Console.WriteLine("클릭!");
	}

	public override void OnClick()
	{
		Console.WriteLine("--------------------");
		base.OnClick();
		Console.WriteLine("--------------------");
	}
}

public class Delegate
{
	public static void Main()
	{
		Application.Run(new Derived());
	}
}
```

앞서 우리가 직접 작성했던 Form 클래스를 보면 델리게이트 객체를 만들기 위한 EventHandler 클래스를 다음과 같이 선언했었습니다.

```
public delegate void EventHandler( );
```

따라서 델리게이트는 리턴값이 없고 파라미터가 없는 함수를 대신 실행할 수 있습니다. 하지만 닷넷 기본 라이브러리에 있는 Form 클래스를 보면 델리게이트 객체를 만들기 위한 EventHandler 클래스는 다음과 같이 선언되어 있습니다.

```
public delegate void EventHandler(Object o, EventArgs e);
```

따라서 델리게이트는 위와 같이 두 개의 파라미터가 있는 함수를 대신 실행할 수 있습니다. 결국 xxx라는 핸들러 함수를 다음과 같이 두 개의 파라미터를 갖도록 선언해야 합니다.

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

public class Derived : Form
{
	public Derived()
	{
		Click += new EventHandler(xxx);
	}

	public void xxx(Object o, EventArgs e)
	{
		Console.WriteLine("클릭!");
	}

	protected override void OnClick()
	{
		Console.WriteLine("--------------------");
		base.OnClick();
		Console.WriteLine("--------------------");
```

```
        }
}

public class Delegate
{
        public static void Main()
        {
                Application.Run(new Derived());
        }
}
```

또한 닷넷 기본 라이브러리 Form 클래스에 선언되어 있는 OnClick 가상 함수도 다음과 같이 파라미터 하나를 갖습니다.

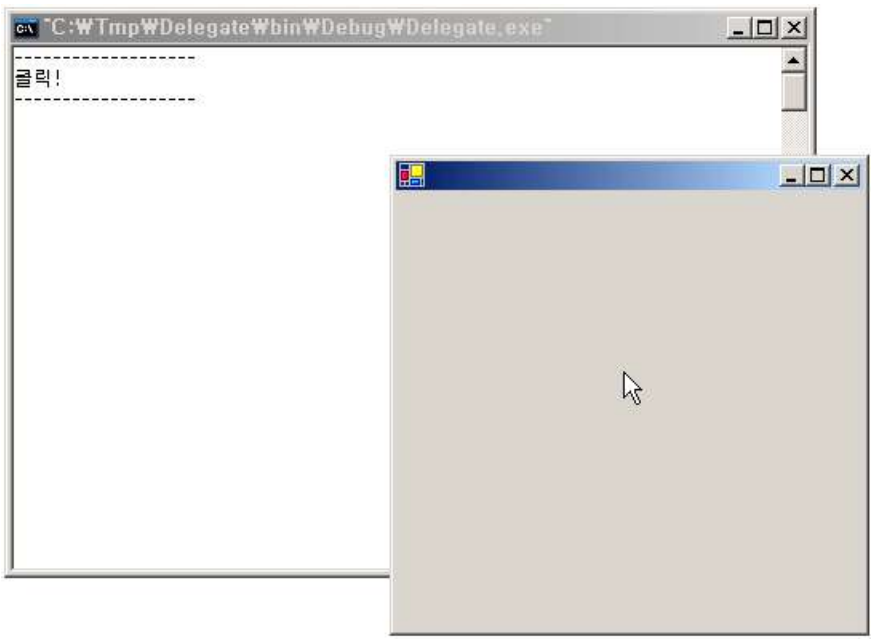
protected virtual void OnClick(EventArgs e);

따라서 가상 함수 OnClick을 재정의할 때 파라미터를 기술해야 합니다.

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

public class Derived : Form
{
        public Derived()
        {
                Click += new EventHandler(xxx);
        }

        public void xxx(Object o, EventArgs e)
        {
                Console.WriteLine("클릭!");
```

```
        }

        protected override void OnClick(EventArgs e)
        {
                Console.WriteLine("--------------------");
                base.OnClick(e);
                Console.WriteLine("--------------------");
        }
}
```

프로그램을 컴파일한 후 실행합니다. 그리고 폼 윈도우에서 마우스 버튼을 클릭해 봅니다. 다음은 실행한 모습입니다.



Step16 이벤트 처리 과정

이제까지 작성한 프로그램을 다시 한 번 보겠습니다.

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

public class Derived : Form
{
	public Derived()
	{
		Click += new EventHandler(xxx);
	}

	public void xxx(Object o, EventArgs e)
	{
		Console.WriteLine("클릭!");
	}

	protected override void OnClick(EventArgs e)
	{
		Console.WriteLine("--------------------");
		base.OnClick(e);
		Console.WriteLine("--------------------");
	}
}

public class Delegate
{
	public static void Main()
	{
		Application.Run(new Derived());  //(A)
```

```
        }
}
```

두 개의 클래스를 작성하였습니다. Delegate 클래스와 Derived 클래스입니다. Derived 클래스는 Form 클래스로부터 상속을 받고 있습니다. 클래스는 객체 만들라고 있는 것이므로 이 두 클래스로 두 개의 객체를 정의해야 합니다. 그런데 Delegate 클래스는 객체를 정의하지 않아도 닷넷 런타임이 실행할 수 있는 정적 함수 Main을 가지고 있으므로 객체를 정의할 필요가 없습니다.

Derived 클래스의 경우에는 객체를 정의하고 있습니다. Main 함수의 (A) 부분에서 객체를 만들고 있습니다.

Derived 클래스와 Form 클래스와 Is-a 관계로서 결국 Derived 클래스로 만든 객체는 Form 객체나 마찬가지입니다. 자, 그렇다면 질문 하나. Application 클래스의 정적 멤버 함수 Run가 실행되면 어떤 일이 발생할까요? (1)가장 먼저 수행하는 일은 폼 객체의 얼굴에 해당하는 윈도우를 표시하는 일입니다. 여러분이 프로그램을 실행한 결과 폼 윈도우가 표시되는 것을 쉽게 확인할 수 있습니다.

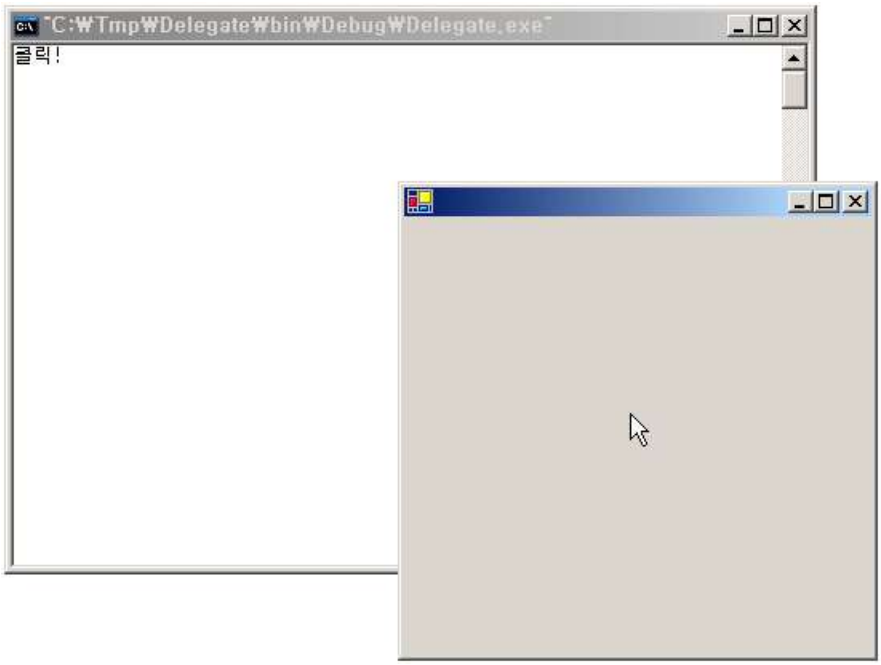
그 다음 수행하는 일은 무엇일까요? (2)여러분이 마우스를 클릭하여 Click 이벤트를 발생시키면 Run 함수에서는 폼 객체의 OnClick 가상 함수(여러분이 재정의한 함수)를 자동으로 호출한다는 것입니다. 그 결과 도스 창에 문자열이 출력됩니다. 여러분이 윈도우를 닫아 프로그램을 종료시킬 때까지 Run 함수는 계속해서 수행됩니다. 윈도우를 닫아야만 Run 함수도 비로소 종료됩니다.

그러면 다음과 같이 OnClick 가상 함수를 재정의하지 않으면 어떻게 될까요?

```
using System;
```

```
using System.Windows.Forms;

public class Derived : Form
{
	public Derived()
	{
		Click += new EventHandler(xxx);
	}

	public void xxx(Object o, EventArgs e)
	{
		Console.WriteLine("클릭!");
	}

	//이곳에 있던 OnClick 가상 함수를 제거하였습니다.
}

public class Delegate
{
	public static void Main()
	{
		Application.Run(new Derived());
	}
}
```

그럴 경우 원래 Form 클래스에 있던 가상 함수 OnClick이 실행되겠지요. 그 함수에서는 Click 델리게이트를 이용하여 xxx 멤버 함수를 호출할 것입니다. 실행 결과는 다음과 같습니다.

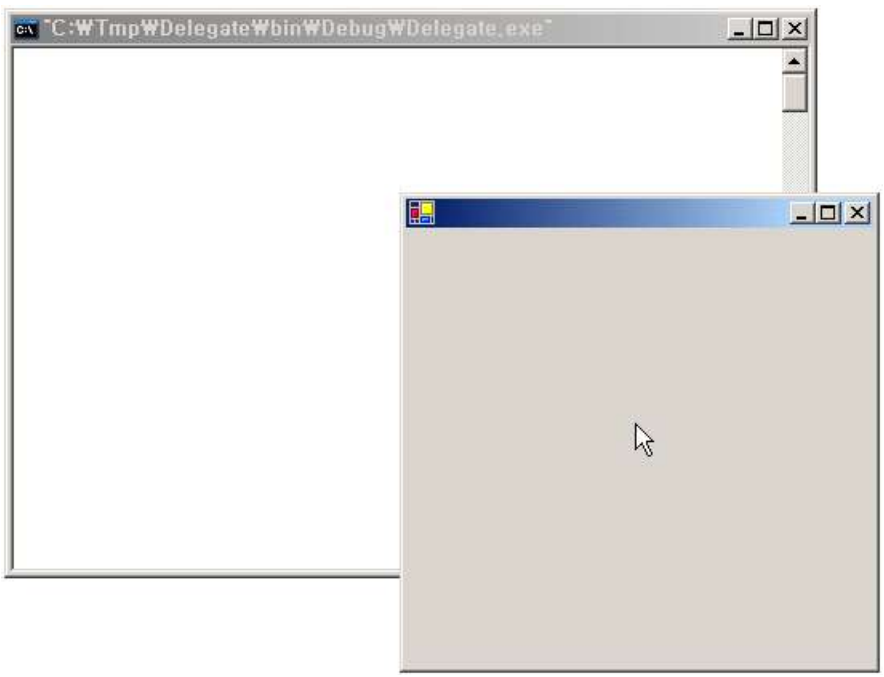


핸들러 함수 xxx 까지도 작성하지 않으면 어떻게 될까요?

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

public class Derived : Form
{
	public Derived()
	{
		//여기에 있던 코드를 제거하였습니다.
	}

	//여기에 있던 xxx 함수를 제거하였습니다.
}
```

```
public class Delegate
{
	public static void Main()
	{
		Application.Run(new Derived());
	}
}
```

컴파일하고 실행해 봅니다. 하지만 폼 윈도우에서 마우스 버튼을 클릭하여도 아무런 반응도 일어나지 않습니다. 델리게이트 Click이 수행할 함수가 설정되지 않으므로 아무런 함수도 수행되지 않기 때문입니다.



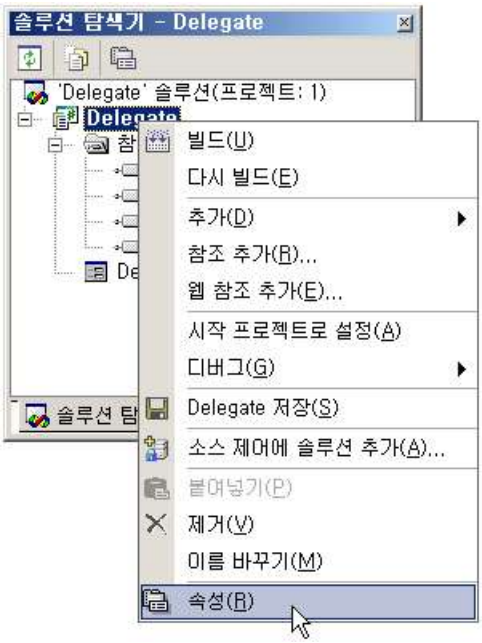
정리하자면, 프로그램을 실행하면 폼 윈도우 객체의 윈도우가 화면에 표시되고, 여러분이 표시된 폼 윈도우 위에서 마우스 버튼을 클릭하면 닷넷 프레임

워크는(구체적으로 Run 함수에서는) Form 객체의 가상 함수 OnClick을 자동으로 호출합니다. OnClick 가상 함수에서는 Click 델리게이트가 null이 아닐 경우, 즉, 등록되어 있는 멤버 함수가 있을 경우 그 멤버 함수 xxx를 실행합니다.

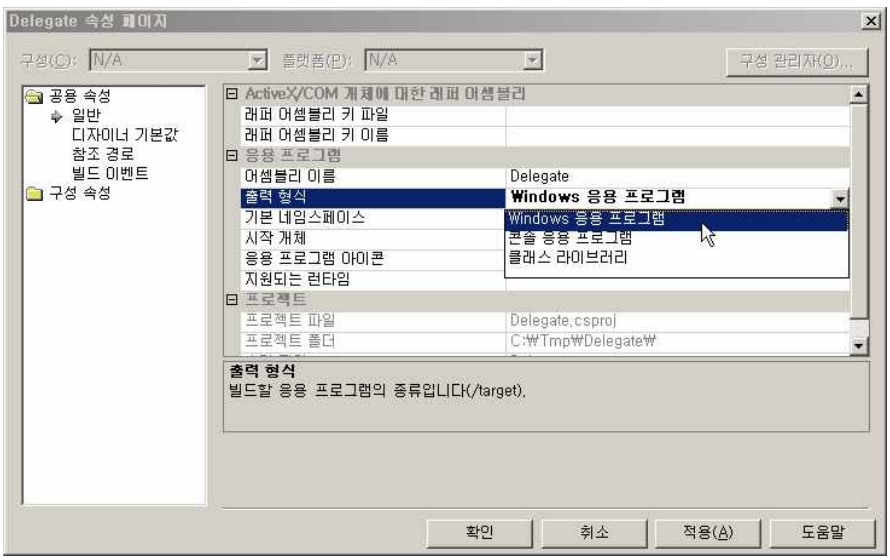
폼 윈도우에서 마우스 버튼을 클릭하면 Click 이벤트가 발생하였다라고 하며, Click 이벤트가 발생하면 OnClick 가상 함수 -> Click 델리게이트 -> 등록되어 있는 멤버 함수 xxx가 실행되며, 따라서 등록되어 있는 멤버 함수에서 이벤트에 반응 및 처리를 할 수 있다고 하여 이벤트 핸들러 함수라고 부릅니다.

윈도우 응용 설정하기

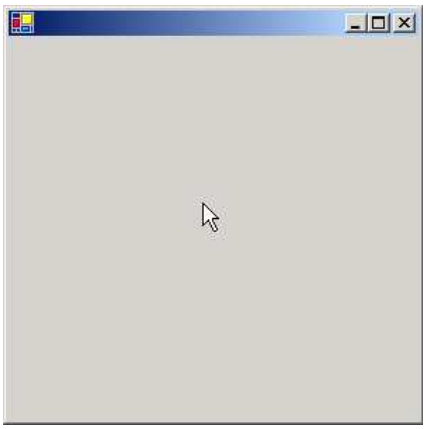
앞서 작성한 프로그램을 실행하면 폼 윈도우가 표시됩니다. 윈도우가 표시되므로 윈도우 응용이기도 하지만 도스 창도 같이 표시됩니다. 도스 창이 표시되지 않도록 하기 위하여 다음과 같이 Delegate 프로젝트에서 오른쪽 마우스 버튼을 눌러 속성(R) 메뉴를 선택합니다.



그런 다음, 출력 형식으로 Windows 응용 프로그램을 선택합니다.



이제 다시 컴파일한 후 실행해 봅니다.



도스 창이 표시되지 않고 윈도우만 표시되며, 따라서 이제는 윈도우 응용이 되었습니다.

Step18 코드 다듬기

Derived 클래스 이름을 MainForm으로 바꾸겠습니다.

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

public class MainForm : Form
{
	public MainForm()
	{
		Click += new EventHandler(xxx);
	}
```

```
        public void xxx(Object o, EventArgs e)
        {
                Console.WriteLine("Hello,World!");
        }
}

public class Delegate
{
        public static void Main()
        {
                Application.Run(new MainForm());
        }
}
```

생성자 함수 안에 작성된 코드를 InitializeComponent라는 멤버 함수로 추상화하겠습니다.

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

public class MainForm : Form
{
        public MainForm()
        {
                InitializeComponent();
        }

        private void InitializeComponent()
        {
                Click += new EventHandler(xxx);
        }
```

```
	public void xxx(Object o, EventArgs e)
	{
		Console.WriteLine("Hello,World!");
	}
}

public class Delegate
{
	public static void Main()
	{
		Application.Run(new MainForm());
	}
}
```

네임스페이스를 지정해 보겠습니다.

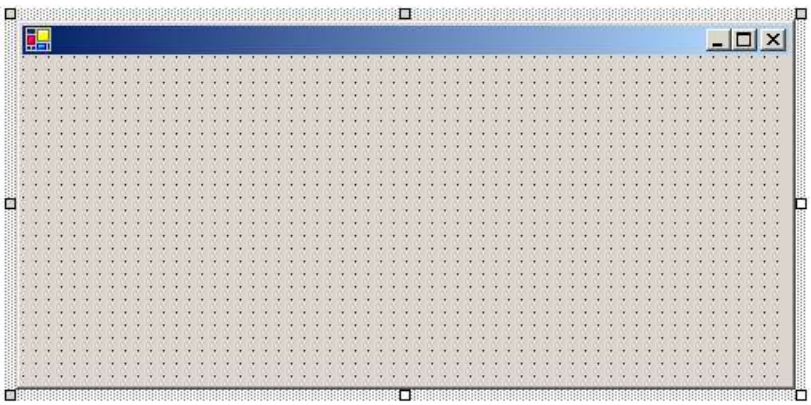
```
using System;
using System.Windows.Forms;

namespace XXX
{
	public class MainForm : Form
	{
		public MainForm()
		{
			InitializeComponent();
		}

		private void InitializeComponent()
		{
			Click += new EventHandler(xxx);
		}
```

```
        public void xxx(Object o, EventArgs e)
        {
            Console.WriteLine("Hello,World!");
        }
    }

    public class Delegate
    {
        public static void Main()
        {
            Application.Run(new MainForm());
        }
    }
}
```

Step19 디자인 편집기를 이용한 UI 디자인

다음과 같이, 솔루션 탐색기에서 Delegate.cs 파일을 두 번 클릭하면 폼을 비주얼 디자인할 수 있습니다.

실제로 비주얼 디자인을 해 봅니다. 다음과 같이 마우스로 윈도우 크기를 변경해 봅니다.



폼 위에서 오른쪽 버튼을 클릭한 후 코드 보기(C) 메뉴 항목을 선택하면 비주얼 디자인한 결과 자동으로 생성된 소스 코드를 볼 수 있습니다.

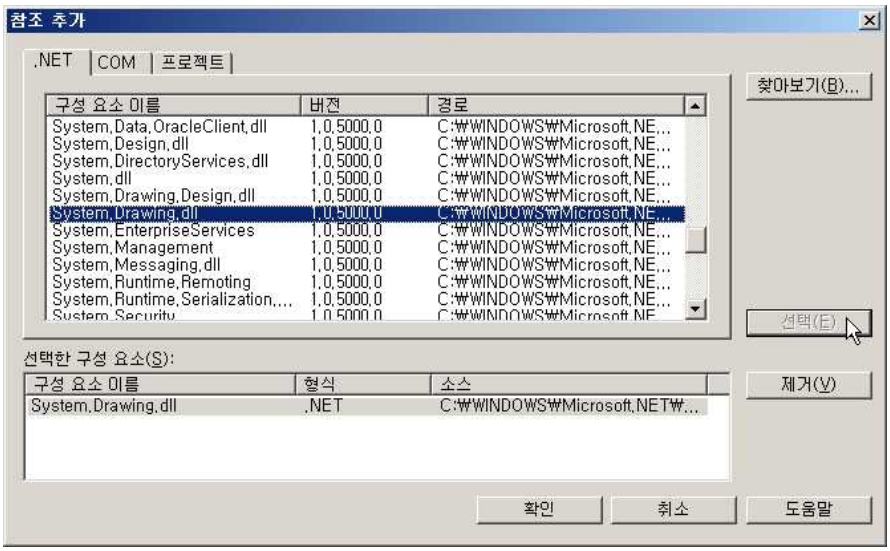
자동으로 작성된 코드는 다음과 같습니다.

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

namespace XXX
{
	public class MainForm : Form
	{
		public MainForm()
		{
			InitializeComponent();
		}

		private void InitializeComponent()
		{
			//
			// MainForm
			//
			AutoScaleBaseSize =
				new System.Drawing.Size(6, 14);
			ClientSize = new System.Drawing.Size(480, 205);
			Name = "MainForm";
			Click += new System.EventHandler(this.xxx);
		}

		public void xxx(Object o, EventArgs e)
		{
			Console.WriteLine("Hello,World!");
```

```
        }
    }

    public class Delegate
    {
        public static void Main()
        {
            Application.Run(new MainForm());
        }
    }
}
```

하지만 이 프로그램을 컴파일할 경우 오류가 발생할 것입니다. UI를 비주얼 디자인한 결과 코드 생성기가 일련의 코드를 작성했는데, 여기에서 System.Drawing 네임스페이스에 있는 Size라는 클래스를 사용하고 있습니다. 하지만 이 라이브러리는 System.Drawing.dll 파일에 있으며, 따라서 이 라이브러리를 프로젝트에 추가해야 합니다. 다음과 같이 참조(라이브러리) 추가 메뉴를 선택합니다.

그런 다음 System.Drawing.dll 파일을 선택하여 확인 버튼을 누릅니다.

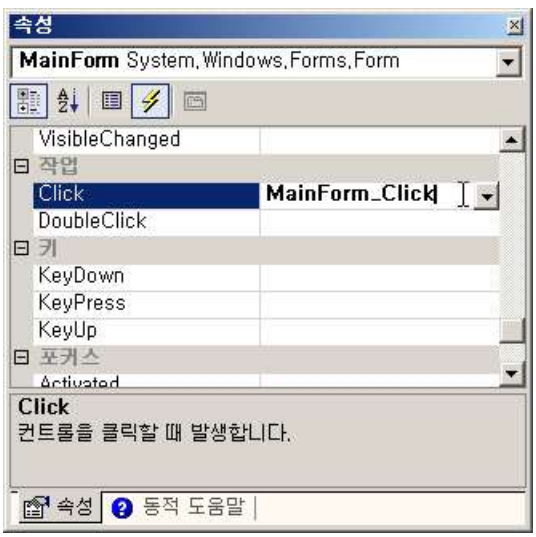


이제 컴파일할 경우 문제가 없을 것입니다.

Step20 마술사를 이용한 코드 생성

사용자 인터페이스뿐만 아니라 프로그램 코드 까지도 마우스를 이용하여 작성할 수 있습니다. 솔루션 탐색기에서 Delegate.cs 파일을 두 번 클릭하여 폼을 표시합니다. 그런 다음 속성 탭에서 번개 모양의 이벤트 버튼을 클릭하면 다음과 같은 화면이 표시됩니다.

Click 이벤트가 발생할 때 자동으로 실행되는 핸들러 함수 이름이 xxx로 되어 있습니다. 이 함수는 메인 폼 윈도우를 클릭할 경우 자동으로 실행되는 핸들러 함수이므로 함수 이름을 MainForm_Click 이라고 변경하면 좋습니다.



MainForm_Click을 입력한 후 엔터 키를 누르면 다음과 같이 코드가 자동으로 작성됩니다.

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

namespace XXX
{
	public class MainForm : Form
	{
		public MainForm()
		{
			InitializeComponent();
		}

		private void InitializeComponent()
		{
			//
			// MainForm
			//
			AutoScaleBaseSize  =  new  System.Drawing.Size(6,
14);
			ClientSize = new System.Drawing.Size(480, 205);
			Name = "MainForm";
			Click += new System.EventHandler(MainForm_Click);

		}

		public void xxx(Object o, EventArgs e)
		{
			Console.WriteLine("Hello,World!");
		}
```

```
        private void MainForm_Click(object sender, EventArgs e)
        {

        }
    }

    public class Delegate
    {
        public static void Main()
        {
            Application.Run(new MainForm());
        }
    }
}
```

이제 마우스로 메인 폼 윈도우를 클릭할 경우 MainForm_Click 멤버 함수가 자동으로 실행됩니다. 여기에서 Click! 문자열을 윈도우로 출력해 보겠습니다.

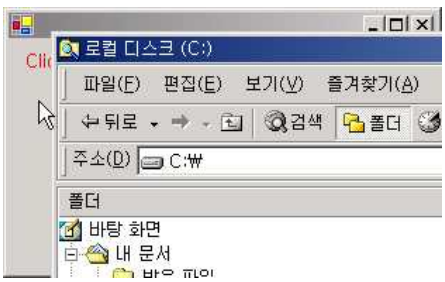
```
using System;
using System.Windows.Forms;
using System.Drawing;

namespace XXX
{
    public class MainForm : Form
    {
        public MainForm()
        {
            InitializeComponent();
        }
```

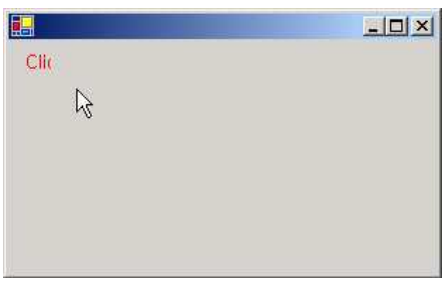
```
            private void InitializeComponent()
            {
                  //
                  // MainForm
                  //
                  AutoScaleBaseSize  =  new  System.Drawing.Size(6,
14);
                  ClientSize = new System.Drawing.Size(480, 205);
                  Name = "MainForm";
                  Click                  +=                  new
System.EventHandler(this.MainForm_Click);

            }

            public void xxx(Object o, EventArgs e)
            {
                  Console.WriteLine("Hello,World!");
            }

            private void MainForm_Click(object sender, EventArgs e)
            {
                  Graphics g = this.CreateGraphics();
                  g.DrawString("Click!",          Font,          new
SolidBrush(Color.Red), 10, 10);
            }
      }

      public class Delegate
      {
            public static void Main()
            {
                  Application.Run(new MainForm());
            }
```

```
        }
}
```

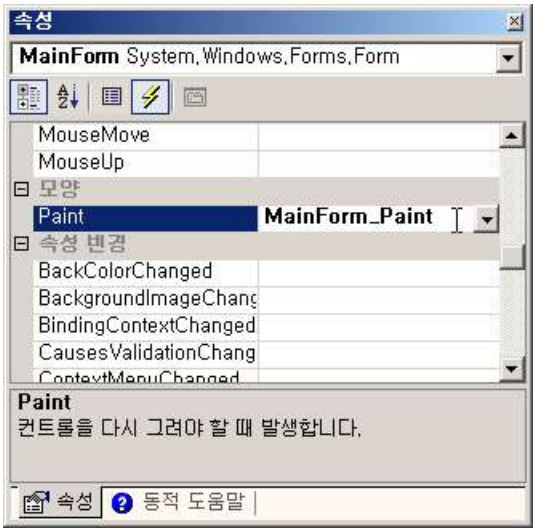
Graphics, Font, SolidBrush 등과 같은 클래스는 System.Drawing 네임스페이스로 선언되어 있는데, 매번 클래스 앞에 이를 기술하지 않아도 되도록 위와 같이 using 문을 선언하였습니다.

프로그램을 컴파일한 후 실행해 봅니다. 하지만 안타깝게도 출력한 문자열이 다른 윈도우로 가려지면 지워져 버립니다.





Paint라는 이벤트가 발생할 경우 실행되는 핸들러 함수 MainForm_Paint를 추가합니다.



엔터 키를 입력하면 새로 추가된 MainForm_Paint 함수로 이동합니다. MainForm_Click 핸들러 함수에 있던 코드를 여기로 옮깁니다.

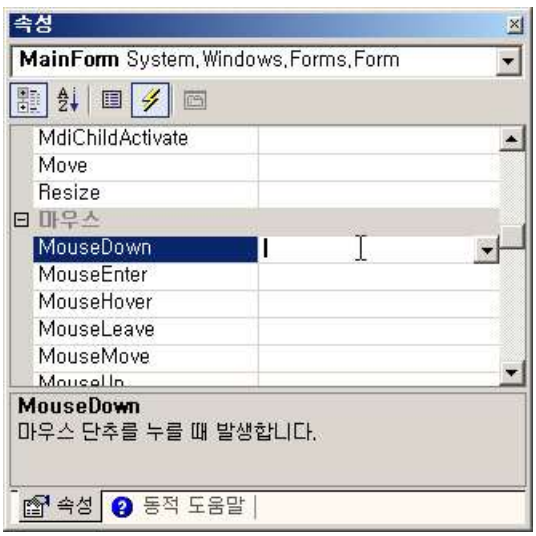
```
private void MainForm_Click(object sender, System.EventArgs e)
{
      //여기에 있던 코드를 아래 함수로 옮겼습니다.
}

private          void          MainForm_Paint(object          sender,
System.Windows.Forms.PaintEventArgs e)
{
      Graphics g = this.CreateGraphics();
      g.DrawString("Click!",  Font,  new  SolidBrush(Color.Red),  10,
10);
}
```

Paint 이벤트는 윈도우가 다시 그려져야 할 필요가 있을 때마다 자동으로 발

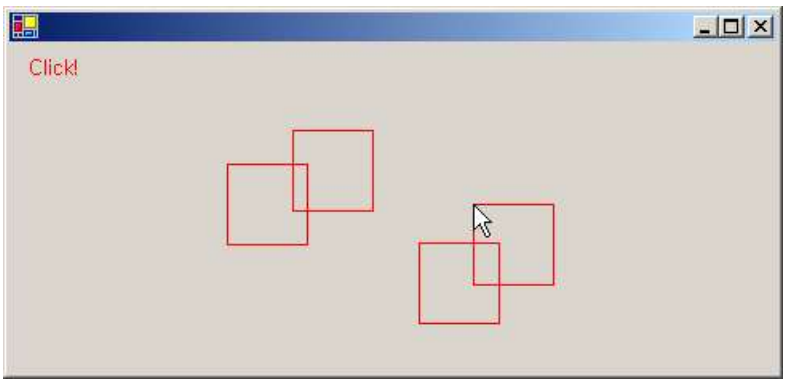
생하며, 따라서 위의 MainForm_Paint 핸들러 함수는 윈도우가 다시 그려져야 할 때마다 자동으로 실행됩니다.

예를 들어, 폼 윈도우를 일부 가렸다가 복원하면 자동으로 Paint 이벤트가 발생하며, 따라서 앞서 실행할 때와 같이 문자열이 지워지는 문제를 해결할 수 있습니다.

MouseDown 이벤트는 폼 윈도우 위에서 마우스 버튼을 누를 경우 발생합니다. 이때 자동으로 실행되는 이벤트 핸들러 함수 MainForm_MouseDown를 정의해 보겠습니다. 핸들러 함수 명을 직접 입력할 필요없이 마우스로 두 번 클릭해도 됩니다.



```
private void MainForm_MouseDown(object sender,
System.Windows.Forms.MouseEventArgs e)
{
        Graphics g = this.CreateGraphics();
        g.DrawRectangle(new Pen(Color.Red), e.X, e.Y, 50, 50);
}
```



제7장
마술사를 이용한 C# 프로그래밍

비주얼 스튜디오 환경에서 프로그래밍은 쉽고 효율적입니다. 특히 마술사를 이용하게 되면 직접 프로그램 코드를 작성하지 않고 많은 코드를 자동으로 생성할 수 있습니다. 이 장에서는 마술사를 이용하여 자동으로 프로그램을 작성하는 방법에 대하여 학습합니다.

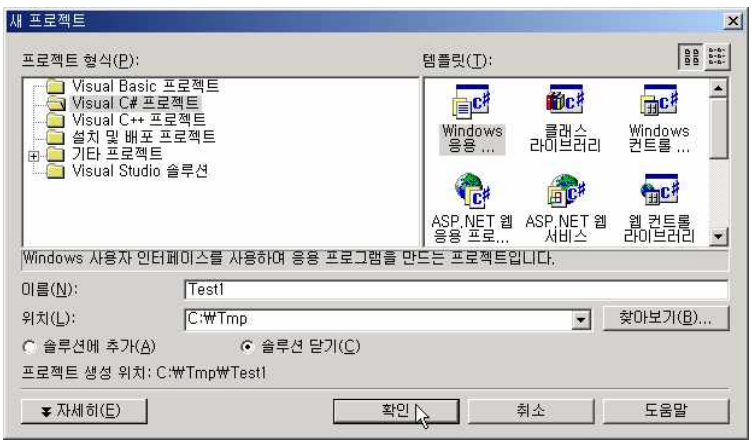
Step1 : 프로젝트(Test1) 생성 및 사전 작업
Step2 : 폼 윈도우 비주얼 디자인
Step3 : 핸들러 함수 작성 연습하기
Step4 : 핸들러 함수 제거 연습하기
Step5 : 가상함수 오버라이딩 연습하기

이 장에서는 몇 가지 C# 코딩 연습을 해 보고 이후에 실제로 몇 가지 응용 프로그램을 작성해 보겠습니다.

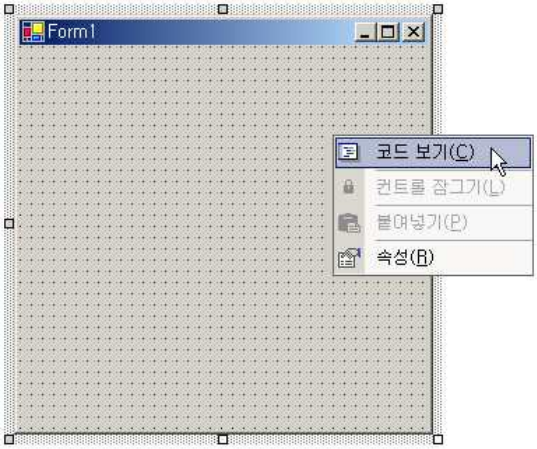
Step1 프로젝트 작성 및 사전작업

Windows 응용 프로그램 프로젝트를 생성하도록 하겠습니다. 다음과 같이 입력한 후 확인 버튼을 누릅니다.

프로젝트 형식	Visual C# 프로젝트
템플릿	Windows 응용 프로그램
이름	Test1



프로젝트를 생성하면 다음과 같이 폼 디자인 템플릿이 나타나는데, 여기에서 마우스의 오른쪽 버튼을 누른 후 코드 보기 메뉴를 선택합니다.



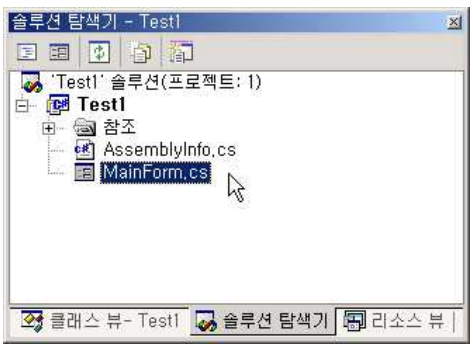
그러면 프로그램 소스가 오픈되는데, 여기에서 먼저 Form1 클래스 이름을 MainForm 으로 변경합니다. 클래스 이름을 MainForm으로 바꾸는 이유는, 이 클래스 객체의 얼굴이 메인 폼 윈도우에 해당되기 때문입니다. 참고로 Visual C++에서는 메인프레임(MainFrame)이라고 합니다.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace Test1
{
	/// <summary>
	/// Summary description for Form1.
	/// </summary>
	public class MainForm : System.Windows.Forms.Form
	{
		/// <summary>
		/// Required designer variable.
		/// </summary>
```

```
        private System.ComponentModel.Container components =
null;

        public MainForm()
        {
            //
            // Required for Windows Form Designer support
            //
            InitializeComponent();

            //
            // TODO: Add any constructor code after
InitializeComponent call
            //
        }

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        protected override void Dispose( bool disposing )
        {
            if( disposing )
            {
                if (components != null)
                {
                    components.Dispose();
                }
            }
            base.Dispose( disposing );
        }

        Windows Form Designer generated code

        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
```

```
            Application.Run(new MainForm());
        }
    }
}
```

클래스 이름을 변경했으므로 파일 이름도 바꾸도록 합니다. 이는 다음과 같이 솔루션 탐색기를 이용하여 Form1.cs 파일을 MainForm.cs 파일로 이름을 변경하면 됩니다.



MainForm 클래스 내부를 보면 폼 윈도우와는 별로 상관없어 보이는 코드가 있습니다. Main 함수가 그것입니다. 즉, 다음과 같습니다.

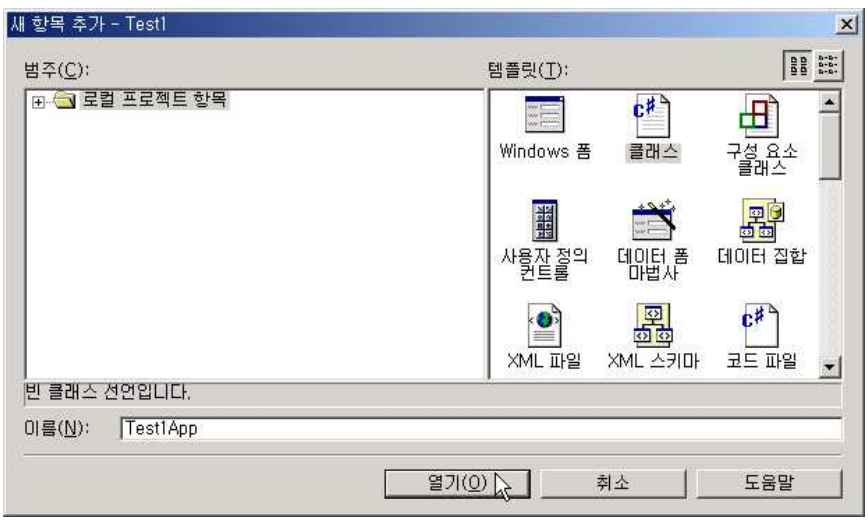
```
static void Main()
{
    Application.Run(new MainForm());
}
```

Application 클래스의 정적(static) 함수 Run은 메시지 루프라고 합니다. 이 메시지 루프가 실행되는 한은 계속해서 프로그램이 실행됩니다. 만일 이 메시지 루프가 종료되면 프로그램도 종료됩니다. 결국 Main 함수에서 호출하고 있는 Run 메시지 루프는 애플리케이션이 실행되는 한 계속해서 실행되는

코드이며 폼 윈도우 자체와는 별로 상관이 없으므로 따로 추상화하여 클래스로 작성하도록 하겠습니다. 이름하여 애플리케이션 클래스로 추상화하겠습니다. 프로젝트 이름이 Test1이므로 애플리케이션을 의미하는 App를 연결하여 Test1App라는 클래스를 작성하도록 하겠습니다. 사실은 Visual C++의 경우에도 이와 동일하게 클래스 이름이 결정되는데, C#의 Test1App와는 조금 다르게 CTest1App라는 클래스로 결정됩니다.

뭐 그냥 코드를 그대로 두지 왜 복잡하게 이렇게 하고 있지?라고 생각할 수 있습니다만 객체지향의 최대 장점인 코드 재사용으로 인한 이점을 최대한 얻으려면 역할 분담이 되어야 합니다. 그런 의미에서 응용프로그램 수행과 관련된 코드를 따로 떼어내서 새로운 클래스로 추상화하는 것입니다. 추상화 및 코드 재사용에 관한 내용은 3장에서 자세히 설명하도록 하겠습니다.

프로젝트(P) > 클래스 추가(C) 메뉴 항목을 선택하여 Test1App 클래스를 추가합니다.



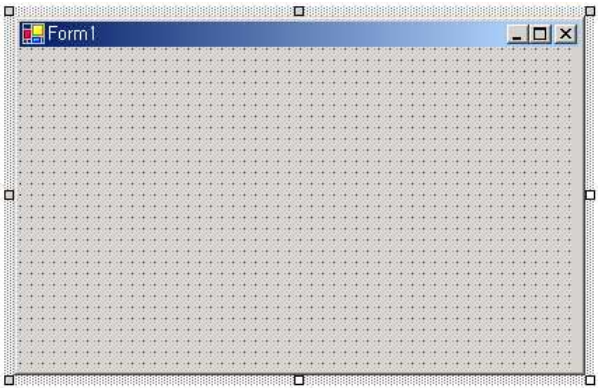
MainForm 클래스에 존재하는 Main 함수를 Test1App 클래스로 옮깁니다. 이때 Main 함수 내에 작성되어 있는 Application라는 클래스는 System.Windows.Forms 네임스페이스에 존재하며, 따라서 다음과 같이 using System.Windows.Forms; 문장을 입력해야 오류가 발생하지 않습니다.

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

namespace Test1
{
	/// <summary>
	/// Summary description for Test1App.
	/// </summary>
	public class Test1App
	{
		public Test1App()
		{
			//
			// TODO: Add constructor logic here
			//
		}

		/// <summary>
		/// The main entry point for the application.
		/// </summary>
		[STAThread]
		static void Main()
		{
			Application.Run(new MainForm());
		}
	}
}
```

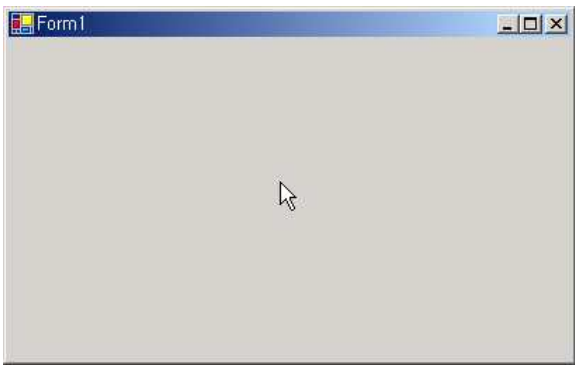
클래스가 하나였었는데 이제 두 개가 되었군요. 사실 **OOP는 클래스를 계속 해서 만들어 가는 과정**입니다. 물론 클래스는 객체를 만들기 위한 것입니다.

폼 윈도우 비주얼 디자인

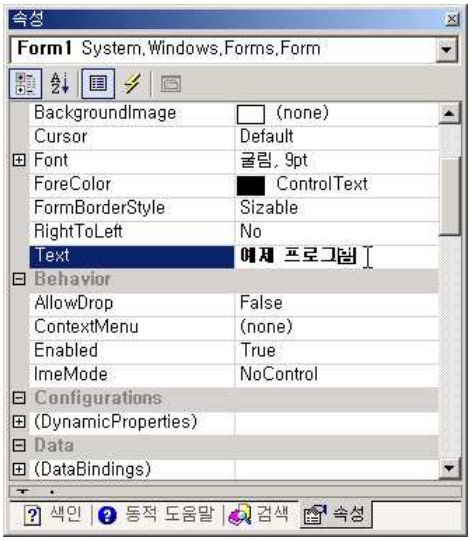
Visual C#이라는 이름에서 Visual의 의미는 마우스나 키보드를 이용하여 눈으로 보면서(visually) 프로그램을 작성할 수 있기 때문에 생겨난 이름입니다. 먼저 아래의 폼 윈도우 템플릿 크기를 원하는 크기로 변경해 보기 바랍니다.



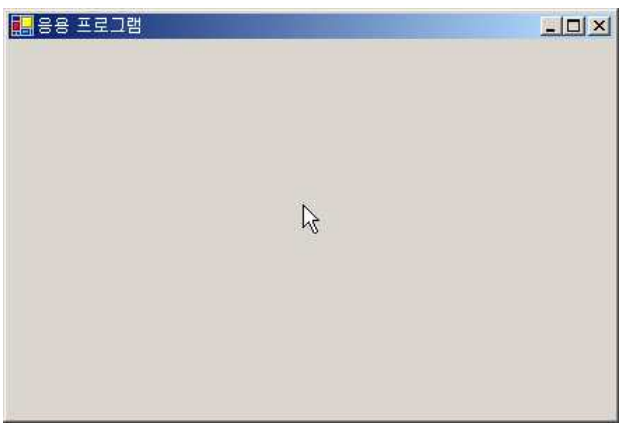
폼 크기를 변경한 후 프로그램을 컴파일한 후 실행한 모습은 다음과 같습니다.



이제 속성 창을 이용하여 타이틀 바 문자열 지정해 보겠습니다. 방법은 다음과 같이 Text 속성에 예제 프로그램 이라는 문자열을 입력하면 됩니다.



프로그램을 컴파일한 후 실행하면 다음과 같이 타이틀 바에 문자열이 표시됩니다.



Step3 핸들러 함수 작성 연습하기

핸들러 함수(handler function)란 이벤트가 발생할 경우 자동으로 실행되는 함수를 말합니다. 그러면 이벤트란 무엇일까요? 이벤트란 대개의 경우 마우스나 키보드를 이용하여 무엇인가를 하는 행위를 말합니다. 예를 들어 앞서 표시된 폼 윈도우를 마우스 버튼으로 클릭하면 '마우스 버튼 클릭 이벤트(Click)가 발생'합니다. 물론 마우스나 키보드와 직접적으로 관련이 없는 이벤트도 존재합니다.

여러분의 얼굴을 누군가가 막대기로 아프게 꾹 찌르면 어떤 일이 발생할까요? 아마도 '아야!' 하고 아파하겠지요? 즉 여러분이 반응합니다. 옆에 있는 사람이 대신 아파해 줄 수는 없는 노릇이지요. 앞서 표시된 폼 윈도우는 폼 윈도우 객체의 얼굴입니다. 실제 객체는 메모리에 존재하고 그 객체의 얼굴이 모니터 화면에 표시되는 셈이지요. 이처럼 얼굴, 혹은 윈도우를 갖는 객체를 윈도우 객체라고 합니다. 그렇다면 저 폼 윈도우를 마우스로 꾹 누르면 어떤 일이 발생할까요? 한 번 해 보세요.

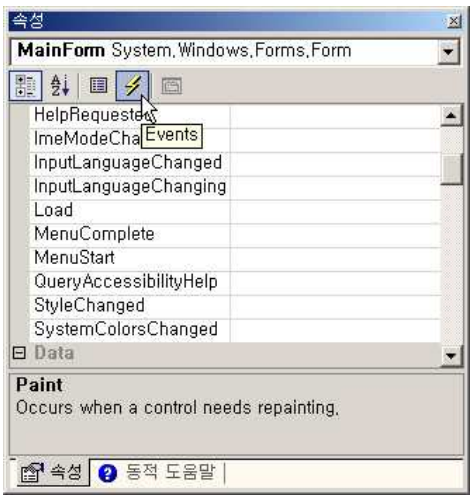
아무런 일도 발생하지 않는다구요? 아마도 윈도우 객체는 성격이 좋은가 봅니다. 하지만 틀렸습니다. 윈도우 객체는 어떤 일을 수행하였습니다. 사실은 폼 윈도우 객체의 OnClick이라는 가상 함수가 자동으로 실행되었습니다. 가상 함수에 대해서는 4장에서 자세히 다루도록 하겠습니다. 가상 함수는 .NET 프레임워크의 핵심으로 공부해 보면 정말 재미있음을 알 수 있습니다. 그런데 이 OnClick 가상 함수에서는 Click 이벤트가 발생할 때 자동으로 실행되는 함수를 호출합니다.

그런데 문제는 Click 이벤트가 발생할 경우 자동으로 실행되는 핸들러 함수를 아직 등록하지 않았다라는 것이지요. 그렇다면 이제 '마우스 버튼 클릭(Click) 이벤트'가 발생할 경우 아야!라는 문자열을 표시해보도록 하겠습니다.

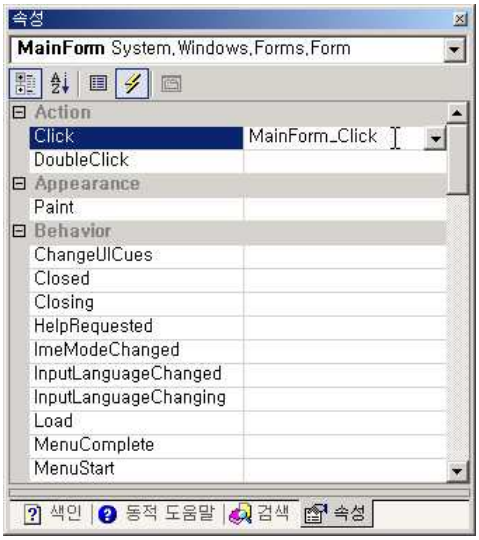
무엇을 해야 할까요? 그렇습니다. 다음과 같은 코드를 작성해야 합니다.

- 핸들러 함수를 작성한다.
- Click 이벤트가 발생할 때 실행되도록 등록한다.

그런데 이 모든 것을 한꺼번에 할 수 있는 방법이 있습니다. 뭐 방법은 간단합니다. 우선, 다음과 같이 속성 창에서 번개 모양과 같이 생긴 이벤트 아이콘을 클릭합니다.



그리고 아래 그림과 같이 Click 이벤트 오른쪽에 MainForm_Click 이라는 함수 이름을 입력합니다.



그리고 마지막으로 엔터(Enter) 키를 눌러줍니다.

모든 게 끝났습니다. 결과적으로 아래와 같이 MainForm_Click 이라는 함수가 자동으로 작성되는데, 이 함수에서 Close 멤버 함수를 호출하도록 코드를 작성합니다.

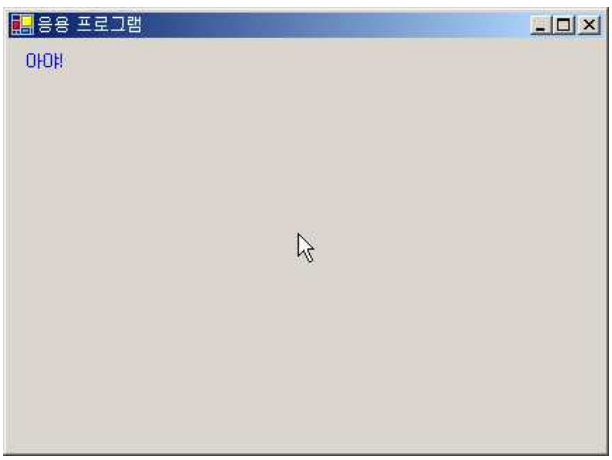
```
            private      void      MainForm_Click(object      sender,
System.EventArgs e)
            {

            }
```

위 코드가 작성되었을 뿐만 아니라 Click 이벤트가 발생할 경우 위의 함수가 자동으로 실행되도록 하기 위한 코드도 어딘가에 자동으로 작성되어 있습니다. 이것에 대한 설명은 5장에서 자세히 설명합니다. 이제 아래와 같은 코드를 입력합니다.

```
            private      void      MainForm_Click(object      sender,
System.EventArgs e)
            {
                  Graphics g = this.CreateGraphics();

                  g.DrawString("아야!",          Font,          new
SolidBrush(Color.Blue), 10, 10);
            }
```

객체지향이므로 그림을 그리는 화가 객체 g를 생성해서 그 객체에게 10, 10 위치에 아야!라는 문자열을 출력하도록 명령을 내리고 있습니다.

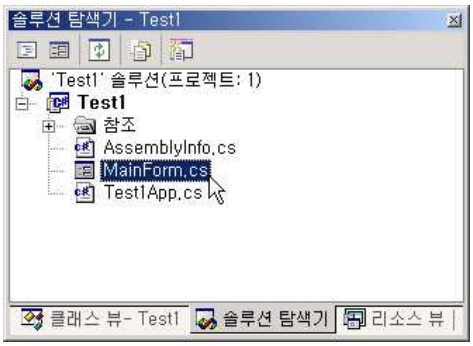
결국 이렇게 하면 폼 윈도우를 클릭(Click)할 경우 MainForm_Click 함수가 자동으로 실행되며, 결과적으로 다음과 같이 문자열이 표시됩니다.



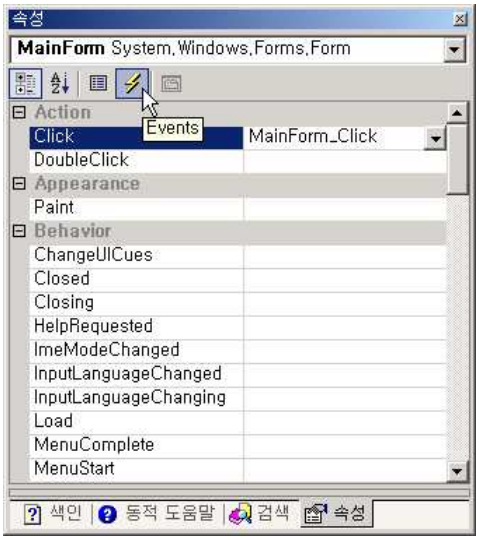
이벤트와 핸들러 함수 이외에 사실은 델리게이트라는 것을 이해해야 내부 메커니즘을 훤히 알 수 있는데, 이에 대해서는 6장에서 자세히 설명하였습니다.

Step4 핸들러 함수 제거 연습하기

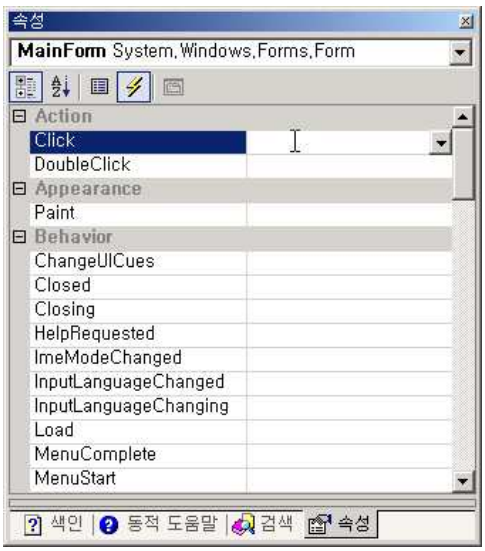
앞서 작성한 핸들러 함수를 제거해 보겠습니다. 솔루션 탐색기에서 MainForm.cs 파일을 두 번 클릭합니다.



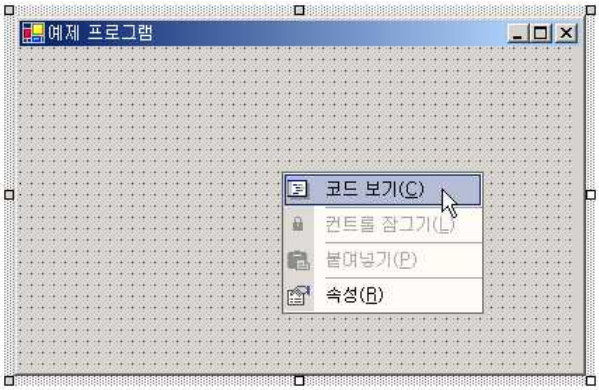
그런 다음 속성 창에서 Events 아이콘을 클릭하면 Click 이벤트가 발생할 경우 MainForm_Click 핸들러 함수가 실행되도록 작성되어 있음을 알 수 있습니다.



이제 아래와 같이 앞서 작성했던 MainForm_Click 문자열을 제거합니다.



Ctrl + S 키를 누르면 변경한 내용이 저장됩니다. 그리고 난 후 다음과 같이 폼 윈도우 클래스의 소스 코드를 오픈합니다.



그러면 해당 핸들러 함수가 제거된 것을 볼 수 있습니다. 하지만 핸들러 함수가 제거되지 않는 경우도 있습니다. 이 경우에는 여러분이 직접 핸들러 함수를 제거하면 됩니다.

다른 방법으로 핸들러 함수 작성해 보겠습니다. 속성 창에서 다음과 같이 이벤트 Click을 두 번 클릭하면 자동으로 MainForm_Click 핸들러 함수가 자동으로 작성됩니다.

그리고 다음 코드가 자동으로 오픈됩니다.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
```

```
namespace Test1
{
	/// <summary>
	/// Summary description for Form1.
	/// </summary>
	public class MainForm : System.Windows.Forms.Form
	{
		/// <summary>
		/// Required designer variable.
		/// </summary>
		private System.ComponentModel.Container components =
null;

		public MainForm()
		{
			//
			// Required for Windows Form Designer support
			//
			InitializeComponent();

			//
			// TODO: Add any constructor code after
InitializeComponent call
			//
		}

		/// <summary>
		/// Clean up any resources being used.
		/// </summary>
		protected override void Dispose( bool disposing )
		{
			if( disposing )
			{
				if (components != null)
				{
					components.Dispose();
				}
			}
			base.Dispose( disposing );
```

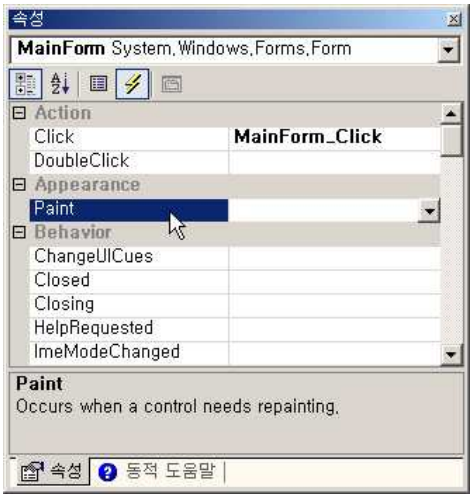
```
            }

            Windows Form Designer generated code

            private      void      MainForm_Click(object      sender,
System.EventArgs e)
            {

            }
        }
}
```

폼 윈도우가 다시 그려져야 할 때마다 Paint 이벤트가 발생하는데, 이때 응용 프로그램 내부적으로 폼 윈도우 객체의 OnPaint 가상 함수가 자동으로 실행됩니다. 그러면 OnPaint 가상 함수에서는 Paint 이벤트에 연결된 핸들러 함수가 자동으로 실행됩니다.

아직 Paint 이벤트에 등록되어 있는 핸들러 함수가 존재하지 않으므로 다음 코드를 작성하도록 하겠습니다.

- MainForm_Paint 핸들러 함수를 작성한다.
- Paint 이벤트가 발생할 때 실행되도록 등록한다.

간단합니다. 다음과 같이 마우스로 Paint를 두 번 클릭하기만 하면 되겠지요?



자동으로 작성된 핸들러 함수는 다음과 같습니다.

```
            private      void      MainForm_Paint(object      sender,
PaintEventArgs e)
            {

            }
```

만일 다음과 같이 코드를 입력하면 어떻게 될까요?

```
            private      void      MainForm_Paint(object      sender,
PaintEventArgs e)
            {
                  Graphics g = e.Graphics;

                  g.DrawString("Hello,World!",       Font,       new
SolidBrush(Color.Blue), 10, 10);
            }
```

이를 컴파일한 후 실행한 모습은 다음과 같습니다. 위 코드에서 작성한대로 10, 10 위치에 Hello,World! 문자열이 항상 표시됩니다.

Step5 가상함수 오버라이딩 연습하기

어떤 이벤트가 발생하면 응용프로그램 내부에서는 결과적으로 On이벤트 라는 가상 함수를 호출합니다. 예를 들어 폼 윈도우를 클릭할 경우 발생하는 Click 이벤트가 발생할 경우 OnClick 가상 함수가 이미 있어서 그것이 자동으로 호출됩니다. 그러면 OnClick 함수에서는 이벤트에 연결되어 있는 핸들러 함수(가령 MainForm_Click)를 호출합니다.

다른 예로, Paint 이벤트가 발생할 경우 OnPaint 가상 함수가 이미 있어서 그것이 자동으로 호출됩니다. OnPaint 가상 함수에서는 Paint 이벤트에 연결되어 있는 핸들러 함수(가령 MainForm_Paint)를 호출합니다.

따라서 이벤트를 처리하는 손쉬운 방법은 이제까지 해 왔던 대로 핸들러 함

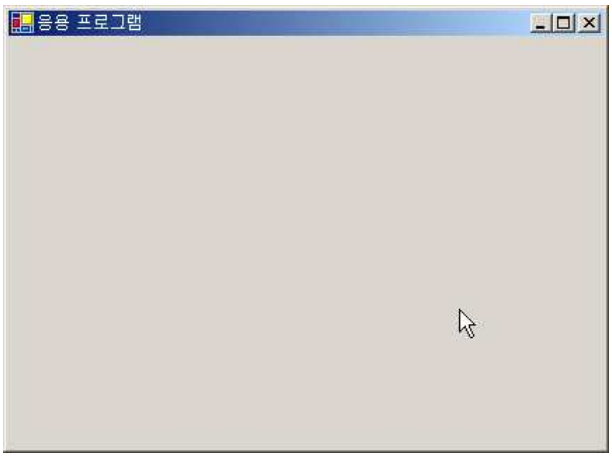
수를 작성한 후 이를 원하는 이벤트에 연결하는 것입니다. 하지만 핸들러 함수를 작성하지 않고서도 이벤트를 처리할 수 있습니다. 가상 함수를 직접 오버라이딩하는 방법이 그것입니다. 즉, MainForm_Click 이라는 핸들러 함수를 작성하지 않고, 대신에 OnClick 가상 함수를 직접 오버라이딩하면 됩니다. 물론 오버라이딩한 가상 함수에 여러분이 원하는 코드를 입력해야 하겠지요?

다음은 Click 이벤트가 발생할 경우 응용프로그램 내부에서 자동으로 실행되는 OnClick 가상 함수를 오버라이딩한 결과입니다.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace Test1
{
	/// <summary>
	/// Summary description for Form1.
	/// </summary>
	public class MainForm : System.Windows.Forms.Form
	{
		/// <summary>
		/// Required designer variable.
		/// </summary>
		private  System.ComponentModel.Container  components  =
null;

		public MainForm()
		{
			//
			// Required for Windows Form Designer support
			//
			InitializeComponent();
```

```
            //
            // TODO: Add any constructor code after
InitializeComponent call
            //
        }

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        protected override void Dispose( bool disposing )
        {
            if( disposing )
            {
                if (components != null)
                {
                    components.Dispose();
                }
            }
            base.Dispose( disposing );
        }

        Windows Form Designer generated code

        private void MainForm_Click(object sender,
System.EventArgs e)
        {
            Graphics g = this.CreateGraphics();

            g.DrawString("아야!", Font, new
SolidBrush(Color.Blue), 10, 10);
        }

        protected override void OnClick(EventArgs e)
        {

        }
    }
}
```

프로그램을 컴파일한 후 실행해 보세요.



아무리 마우스 버튼을 클릭해도 '아야!'라는 문자열이 표시되지 않습니다. 즉, MainForm_Click 핸들러 함수가 실행되지 않습니다. 왜 그럴까요?

그 이유는 이렇습니다. 비록 여러분이 눈으로 확인할 수는 없지만, 기존에 Form 클래스에 있는 OnClick 가상 함수, 즉, 여러분이 오버라이딩한 함수 말고 기존에 이미 존재하고 있는 함수에서는 MainForm_Click 핸들러 함수를 호출합니다.

하지만 아래와 같이 여러분이 오버라이딩한 OnClick 가상 함수에서는 MainForm_Click 핸들러 함수를 호출하는 코드는 고사하고 아무런 코드도 없습니다. 결국 여러분이 폼 윈도우를 클릭할 경우 아무런 일도 수행하지 않습니다. 가상 함수 오버라이딩은 정말 중요한 내용입니다. OOP의 하이라이트라고나 할까요.

```
protected override void OnClick(EventArgs e)
{
```

```
}
```

그렇다면, 위와 같이 OnClick 가상 함수를 오버라이딩하더라도 MainForm_Click 핸들러 함수가 실행되도록 하려면 어떻게 해야 할까요? Form 클래스에 작성되어 있는 기존의 OnClick 함수를 호출하기만 하면 됩니다. 다음과 같이 말이죠.

```
protected override void OnClick(EventArgs e)
{
      base.OnClick(e);
}
```