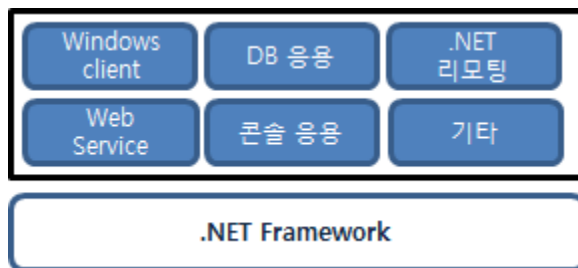


## 1. C# 소개

C#은 .NET Framework 기반에서 동작하는 프로그램을 개발할 때 사용하는 프로그래밍 언어입니다. C# 언어는 표현력이 뛰어나면서도 단순하고 배우기 쉽게 되어 있습니다. 이미 C나 C++, Java에 익숙한 사용자라면 쉽게 사용할 수 있으며 높은 생산성을 발휘할 것입니다.

C# 언어를 이용하면 .NET Framework 기반에서 동작하는 다양한 범위의 프로그램을 개발할 수 있습니다. 일반적인 Windows 클라이언트 응용 프로그램에서 XML Web Services와 분산 구성 요소, 클라이언트/서버 응용 프로그램이나 데이터베이스 응용 프로그램 등 다양한 형태의 응용 프로그램을 제작할 수 있습니다.

이처럼 C#언어를 사용하면 .NET Framework 기반에서 동작하는 다양한 프로그램을 개발할 때 효과적입니다. 이 책에서는 C#언어에 대해서 다루고 있으며 이를 이용하여 다양한 종류(예: Windows 클라이언트 응용)의 프로그램을 개발하는 방법에 대해서는 다루지 않습니다.



[그림 1.1] .NET 프레임워크에서 동작하는 다양한 프로그램

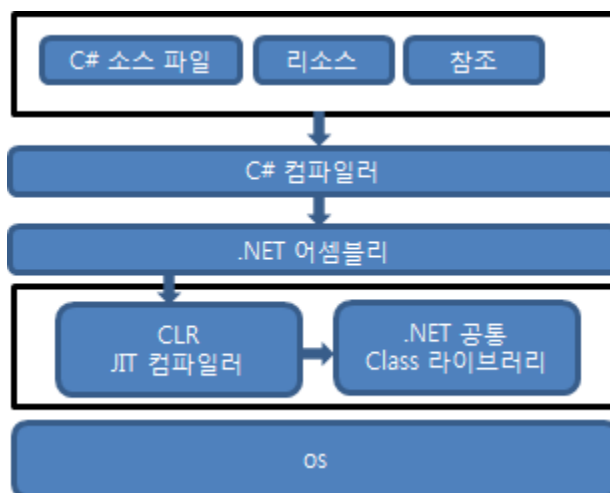
## 1. 1 .NET Framework 구조

.NET Framework를 구성하는 핵심 요소는 CLR(공용 언어 런타임)입니다. CLR은 코드들을 관리하는 주요 역할을 담당합니다. 이와 같은 코드를 관리 코드라고 하며 CLR에 의해 관리되는 코드들은 형식이 안전함을 보장받게 됩니다. CLR에서는 형식 안전성 이외에도 메모리 관리, 쓰레드 실행 및 코드 실행과 기타 시스템 서비스 등을 관리합니다.

그리고, .NET Framework에서는 CTS(공용 형식 시스템)을 통해 견고한 코드를 보장하기 위해 형식에 대해 약속하고 있습니다. CTS는 관리되는 모든 형식은 자기 기술적이며 이 때문에 명확하게 개체들을 사용할 수 있습니다.

또한, CLS(공용 언어 사양)를 제시를 함으로써 자신이 만든 라이브러리를 다른 언어를 사용하는 개발자가 사용할 수 있게 점검해 줍니다. .NET 프레임워크에서 동작 가능한 프로그램을 작성할 때에는 C#언어를 비롯하여 Managed C++, Visual Basic.NET을 비롯하여 다양한 언어로 개발이 가능하며 CLS를 따른다면 교차 언어 개발도 가능합니다.

C# 소스파일을 컴파일하면 C# 컴파일러에 의해 .NET 어셈블리(.NET 프레임워크에서 동작 가능한 이진 파일)가 작성됩니다. 그리고 이를 실행하면 .NET 프레임워크의 핵심 역할을 담당하는 CLR에 의해 .NET 어셈블리를 로드하고 Jitter(Just In Time Compiler)에 의해 MSIL(.NET 어셈블리내에 코드를 MSIL이라 부름)코드를 실제 운영체제에서 동작 가능한 기계어 명령으로 변환하여 가동됩니다. 그리고 CLR은 참조하고 있는 다른 .NET 어셈블리를 로딩하는 작업과 .NET 공통 클래스 라이브러리를 사용하는 부분도 담당합니다.



[그림 1.2] .NET Framework에서 C# 프로그램 동작 구조

## 1. 2 Hello, World!

이제 C#언어로 간단하면서도 유명한 "Hello World!" 프로그램을 작성해 봅시다.

콘솔 응용 프로그램 템플릿 프로젝트를 생성하시면 하나의 프로그램 소스 파일이 자동으로 만들어집니다. 프로그램 진입점인 Main 메서드 내부에 "Hello, World!"를 콘솔 화면에 출력하는 구문을 추가해 봅시다.

▶ Program.cs

```
/*Hello, World! 프로그램: 콘솔 화면에 "Hello, World!"를 출력*/  
using System;  
namespace HelloWorld  
{  
    /// <summary>  
    /// 진입점 메서드를 포함하고 있는 클래스  
    /// </summary>  
    class Program  
    {  
        /// <summary>  
        /// 진입점 메서드  
        /// </summary>  
        /// <param name="args"> 진입점 전달 인자</param>  
        static void Main(string[] args)  
        {  
            Console.WriteLine("Hello, World!");  
        }  
    }  
}
```

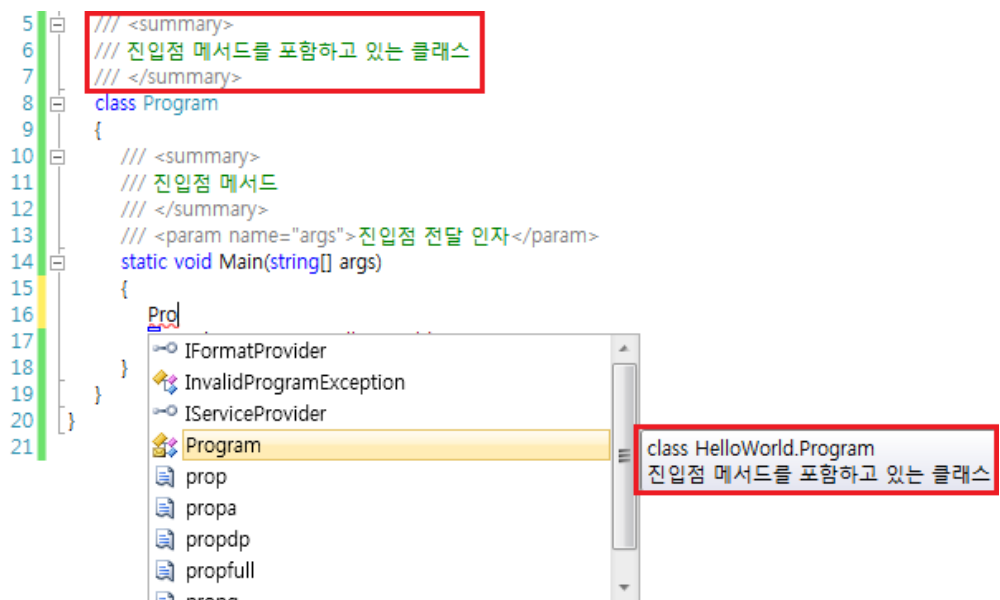
### 1.2.1 주석

주석은 실제 수행되는 코드가 아니라 개발자가 코드를 이해하기 쉽게 작성하는 것입니다. C#에서도 C언어나 C++처럼 한 줄 주석에는 //를 사용하고 특정 영역을 주석으로 만들 때는 영역의 시작에 /\*를 영역의 끝에 \*/를 사용합니다. Microsoft Visual Studio 2010에서는 주석을 만들거나 해제하기 쉽게 [그림4]처럼 툴바에 버튼을 제공하고 있습니다.



[그림 1.3] 주석 설정과 해제하는 버튼

사용자가 정의하는 형식이나 형식 내 멤버에 대한 주석은 /// 를 사용하면 편집 창에서 이를 사용할 때 향상된 인텔리센스를 통해 주석의 내용을 확인할 수 있어 개발이 쉬워집니다.



[그림1.4] 향상된 인텔리센스

### 1.2.2 namespace

C#에서는 namespace는 형식 이름 사이에 충돌을 막기 위해 제공하고 있습니다. 만약, 참조한 두 개의 .NET 어셈블리에 같은 이름의 클래스가 정의되어 있다면 이름 충돌이 발생하여 어느 클래스를 사용하려는 것인지 개발도구는 판단할 수 없게 됩니다. C#에서는 이러한 이름 충돌을 막기 위해 namespace 문법을 제공하고 있습니다.

Console은 .NET Framework에서 제공하는 공용 클래스 라이브러리에서 System 이름 공간에 정의된 클래스입니다. 이를 사용하기 위해 using 구문을 사용하였습니다.

```
using System;
```

### 1.2.3 class

C#으로 작성한 응용 프로그램은 최소한 하나 이상의 클래스를 가지고 있어야 합니다.

### 1.2.4 Main

Main은 프로그램의 진입점으로써 정적 메서드입니다. 여러분은 목적에 따라 Main 메서드를 다음 중 한 가지 형태로 정의할 수 있습니다.

```
static void Main();  
static int Main();  
static int Main(string[] args);  
static void Main(string[] args);
```

### 1.2.5 Console.WriteLine

Console은 .NET Framework에서 제공하는 클래스 라이브러리에서 제공되는 클래스 중의 하나이며 System namespace에 정의되어 있습니다. Console 클래스는 표준 출력, 표준 입력 및 오류 스트림을 나타내며 콘솔 응용 프로그램에서 입출력 작업을 위해 자주 사용됩니다. 그중에 WriteLine은 정적 멤버 메소드로 표준 출력에 입력 인자로 전달된 문자열과 개행을 출력하는 역할을 담당합니다.

## 2. C# 구성 요소

여러분도 잘 아시는 것처럼 C#은 프로그래밍 언어 중의 하나이며 .NET Framework 기반에서 동작하는 프로그램을 작성할 때 사용됩니다. 이번 장에서는 프로그래밍 언어인 C#의 구성 요소를 개괄적으로 살펴보기로 할게요.

프로그래밍 언어는 공통으로 관리해야 하는 데이터에 대한 문법과 해야 할 일에 대한 문법들을 제공하고 있습니다. 그리고 C#과 같은 개체 지향 프로그래밍 언어는 프로그래밍 성능보다는 신뢰성과 재사용성을 높일 수 있게 만들어져 있습니다.

### 2.1 데이터에 관한 문법 사항

C#에서는 프로그램 내에 관리할 데이터를 표현하기 위한 문법 사항으로 형식과 변수를 제공하고 있습니다. 형식이란 프로그램 내에 표현할 데이터 종류에 대한 약속입니다. 이러한 형식은 변수 선언이나 개체 생성을 통해 구체화합니다. 이처럼 구체화 된 데이터에 접근하기 위해서 변수를 사용하게 됩니다.

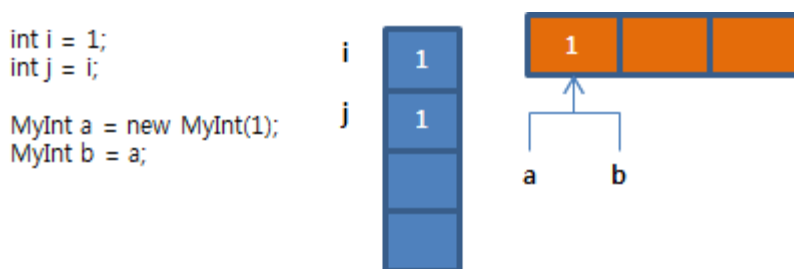
C# 형식	.NET Framework형식	설명 및 표현 범위
bool	System.Boolean	논리 값을 표현 (true 혹은 false)
byte	System.Byte	부호 있는 8비트 정수(-128~127)
sbyte	System.SByte	부호 없는 8비트 정수(0~255)
char	System.Char	유니코드 16비트 문자
decimal	System.Decimal	128비트 데이터 형식
double	System.Double	7개의 자릿수를 갖는 실수 (근사치)
float	System.Single	15~16개의 자릿수를 갖는 실수(근사치)
int	System.Int32	부호 있는 32비트 정수
uint	System.UInt32	부호 없는 32비트 정수
long	System.Int64	부호 있는 64비트 정수
ulong	System.UInt64	부호 없는 64비트 정수
object	System.Object	모든 형식의 기반 형식
short	System.Int16	부호 있는 16비트 정수
ushort	System.UInt16	부호 없는 16비트 정수
string	System.String	0개 이상의 유니코드 문자로 구성된 시퀀스

C#에서의 형식은 이미 정의되어 제공되는 기본 형식과 필요할 때 개발자가 정의해서 사용하는 사용자 정의 형식으로 나눌 수 있습니다. 그리고 형식은 구체화 된 데이터에 접근하기 위한 변수가 값을 의미하거나 생성된 개체를 참조하는가에 따라 값 형식과 참조 형식으로 나누기도 합니다. 기본 형식에는 참과 거짓을 표현할 수 있는 bool 형식과 유니코드 문자를 표현하기 위한 char 형식, 정수와 실수를 표현하기 위한 여러 형식과 문자열을 위해 string과 모든 형식의 기반 형식인 object를 제공하고 있습니다. 기본 형식으로 제공되는 형식 중에 string과 object는 참조 형식이며 나머지 형식들은 모두 값 형식입니다.

C#에서는 기본 형식 외에 필요한 형식을 정의하여 사용할 수 있게 사용자 정의 형식을 제공하고 있습니다. 사용자 정의 형식은 표현할 수 있는 값의 종류를 정의하는 열거형과 여러 가지 멤버들을 하나의 형식 내에 정의할 수 있는 구조체와 클래스 등이 있습니다. 이 중에 구조체와 열거형은 값 형식이며 클래스는 참조 형식입니다.

C#에서 값 형식들은 변수들이 독립적으로 사용됩니다. 이에 반해 참조 형식은 변수들이 관리와 힙에 할당된 개체를 참조하여 사용되기 때문에 하나의 개체를 사용하는 여러 개의 변수가 있을 수 있습니다.

C#에서 값 형식의 변수는 독립적으로 사용되므로 같은 형식의 변수로 초기화되거나 대입 연산 등이 이루어질 때 단순히 값을 복사할 뿐입니다. 즉, 값 형식에서 변수가 다르다는 것은 할당된 메모리가 다르다는 것을 의미합니다. 참조 형식은 변수는 단순히 개체를 참조하여 사용되는 이름이기 때문에 같은 개체를 참조하는 여러 개의 변수가 있을 수 있습니다.



[그림 2.1] 값 vs 참조

다음의 예제 코드는 값 형식과 참조 형식의 차이에 대한 이해를 돕기 위한 간단한 코드입니다.

▶ Ex-ValueVSReference

```
class MyInt // 사용자 정의 클래스 - 참조 형식
{
    public int Value{ get; set; } //속성
    public MyInt(int value) //생성자
    {
        Value = value;
    }
    public override string ToString() //재정의
    {
        return Value.ToString();
    }
}

class Program
{
    static void Main(string[] args)
    {
        int i = 1; //값 형식 변수 i 선언 및 초기화
        int j = i; //값 형식 변수 j 선언 및 초기화(i를 초기화에 사용)
        i++; //i를 1 증가
        Console.WriteLine("i:{0} j:{0}",i,j); //값 형식인 i와 j값을 출력
        MyInt a = new MyInt(1); // MyInt 개체를 생성하여 변수 a에 대입
        MyInt b = a; //MyInt 형식 변수 b 선언 및 초기화(a를 초기화에 사용)
        a.Value++; //a의 속성 Value를 1 증가
        Console.WriteLine("a:{0} b:{0}", a, b); //참조 형식인 a와 b를 출력
    }
}
```

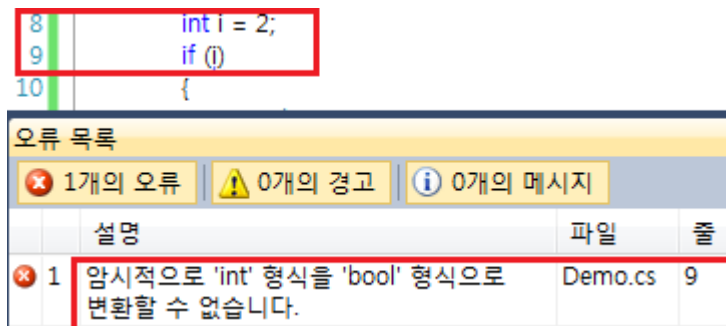


## 2.2 해야 할 일에 관한 문법 사항

컴퓨터 프로그램에는 관리해야 할 데이터 이외에도 동작해야 할 논리를 표현하는 문법이 필요할 것입니다. C#에서는 프로그램의 논리를 표현하기 위해 연산자와 식, 문 등을 제공하고 있습니다. 수행해야 할 코드에서 값이나 변수, 속성, 리터럴, 연산자, 함수 호출 등의 코드 조각을 식(Expressions)이라 하고 이러한 식으로 구성하여 세미콜론으로 구분하는 것을 문(Statements)이라 부릅니다.

식에는 단순히 변수이름을 사용하는 것과 리터럴 상수를 사용하는 것, 메서드 이름과 인자를 사용하는 것이 있습니다.

```
static void Main(string[] args)
{
    string s = string.Empty;
    s = "Hello"; //리터럴 상수 사용
    string s2 = s; //변수 이름을 사용
    Console.WriteLine(s2); //호출문
}
```



[그림 2.2] 조건식에 bool 형식이 아닌 형식을 사용했을 때의 오류 화면

```

static void Main(string[] args)
{
    int i = 0; //선언문
    Console.WriteLine("수를 입력하세요.."); //호출식(문)
    try //예외 처리문의 try 블록
    {
        i = int.Parse(Console.ReadLine());
        if ((i % 2) == 0) //조건문
        {
            Console.WriteLine("입력한 수 {0}는 짝수입니다..", i);
        }
        else
        {
            Console.WriteLine("입력한 수 {0}는 홀수입니다. ", i);
        }

        int sum = 0;
        for (int index = 1; index < i; index++) //반복문
        {
            sum += index;
        }

        Console.WriteLine("1~{0}까지의 합은 {1}입니다.", 1, i, sum);
    }
    catch (Exception e) //예외 처리문의 catch 블록
    {
        Console.WriteLine("예외가 발생하였습니다. {0}", e.Message);
    }
}

```

### 2.2.1 if - else 문

조건식으로 bool 형식이 올 수 있으며 조건식의 결과에 따라 수행 여부를 결정합니다.  
조건이 참일 때 수행하는 구문만 정의할 때는 else 부분을 생략할 수 있습니다.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("이름을 입력하세요.");
        string name = Console.ReadLine();

        if (name != string.Empty) //이름을 입력하였을 때
        {
            Console.WriteLine("{0}의 주소를 입력하세요. ", name);
            string addr = Console.ReadLine();

            if (addr != string.Empty) //주소를 입력하였을 때
            {
                Console.WriteLine("{0}의 주소는 {1}입니다. ", name, addr);
            }
        }
        else //이름을 입력하지 않았을 때
        {
            Console.WriteLine("이름을 입력하지 않았습니다.");
        }
    }
}
```

### 2.2.2 switch 문

여러 가지 경우의 코드에서 원하는 코드만 수행할 수 있습니다. 각 경우에 수행할 코드는 case문을 사용하며 일치하는 case문부터 시작합니다. 그리고 break문을 만나면 switch문을 빠져나갑니다. case문으로 표현하지 않은 때에 수행할 코드는 default문을 사용합니다.

그리고 C언어와 C++언어와 다르게 문자열을 사용할 수 있어 개발에 편의성을 높였습니다. 또한, C언어와 C++과 다르게 case문에 수행할 코드가 있을 때에는 break문을 명시해야 합니다.

```
static void Main(string[] args)
{
    string name = string.Empty;
    Console.WriteLine("이름을 입력하세요..");
    name = Console.ReadLine();

    switch (name)
    {
        case "홍길동": Console.WriteLine("휘리릭~"); break;
        case "강감찬":
        case "을지문덕 ": Console.WriteLine("이라~"); break;
        default: Console.WriteLine("음냐뤄~"); break;
    }
}
```

### 2.2.3 while , do while 문

while문과 do while문은 조건식이 참일 동안 반복해서 수행합니다. if문과 마찬가지로 조건식은 bool 형식이 와야 합니다. 그리고 반드시 한 번은 수행해야 한다면 do while문을 사용할 수 있습니다.

반복문을 수행 도중에 빠져나가려고 할 때에는 break문을 사용합니다. 또한, 반복문의 수행 도중에 반복문 시작으로 가고자 할 때는 continue문을 사용합니다. for문을 이용하여 반복문을 표현할 때도 break와 continue에 대한 사항은 같습니다.

```
static void Main(string[] args)
{
    int i = 0;
    while (i < 10)
    {
        Console.WriteLine("{0}", i);
        i++;
    }

    do
    {
        i++;
        Console.WriteLine("do-while:{0}", i);
    } while (i < 10);
}
```

## 2.2.4 for 문

while문 처럼 조건식이 참일 동안 반복해서 수행합니다. 반복문을 수행하기 전에 초기화와 반복문에 조건을 변화시키는 부분이 있어서 초기 구문을 생략하는 개발자의 실수가 줄어듭니다.

```
static void Main(string[] args)
{
    for(int i = 0; i<10; i++)
    {
        Console.WriteLine("for:{0}", i);
    }
}
```

## 2.2.5 foreach문

배열과 같은 컬렉션에 있는 각 요소에 대해 같은 작업을 수행할 때 사용됩니다. 사용자가 정의한 형식을 foreach문에서 사용하려면 GetEnumerator, MoveNext, Reset, Current 멤버가 있어야 합니다. 이 부분은 7장 인터페이스와 컬렉션에서 다루겠습니다.

```
static void Main(string[] args)
{
    int[] arr = new int[]{2,3,4,5,6};
    foreach (int i in arr)
    {
        Console.WriteLine(i.ToString());
    }
}
```

### 3. 형식 개요

C#은 강력한 형식의 언어로 모든 변수와 상수 및 메서드 시그니처의 입력 매개변수와 반환 값 등의 형식을 지정해야 합니다.

형식은 분류 방법에 따라 기본 형식과 사용자 정의 형식으로 나눌 수도 있으며 값 형식, 참조 형식, 포인터 형식으로 구분할 수도 있습니다.

기본 형식에는 true와 false를 값으로 가질 수 있는 bool 형식과 유니코드 문자를 표현하기 위한 char 형식, 정수와 실수를 표현하기 위한 여러 가지 형식과 문자열을 위해 string과 모든 형식의 기반 형식인 object를 제공하고 있습니다. 기본 형식으로 제공되는 형식 중에 string과 object는 참조 형식이며 나머지 형식들은 모두 value 형식입니다.

C# 형식	.NET Framework형식	설명 및 표현 범위
bool	System.Boolean	논리 값을 표현 (true 혹은 false)
byte	System.Byte	부호 있는 8비트 정수(-128~127)
sbyte	System.SByte	부호 없는 8비트 정수(0~255)
char	System.Char	유니코드 16비트 문자
decimal	System.Decimal	128비트 데이터 형식
double	System.Double	7개의 자릿수를 갖는 실수 (근사치)
float	System.Single	15~16개의 자릿수를 갖는 실수(근사치)
int	System.Int32	부호 있는 32비트 정수
uint	System.UInt32	부호 없는 32비트 정수
long	System.Int64	부호 있는 64비트 정수
ulong	System.UInt64	부호 없는 64비트 정수
object	System.Object	모든 형식의 기반 형식
short	System.Int16	부호 있는 16비트 정수
ushort	System.UInt16	부호 없는 16비트 정수
string	System.String	0개 이상의 유니코드 문자로 구성된 시퀀스

C#에서는 기본 형식외에 필요한 형식을 정의하여 사용할 수 있습니다. 사용자 정의 형식은 표현할 수 있는 값의 종류를 정의하는 열거형과 여러가지 멤버를 캡슐화할 수 있는 구조체와 클래스 등이 있습니다.

C#의 형식은 값 형식과 참조 형식으로 나눌 수 있는데 구조체와 열거형은 값 형식이고 클래스는 참조 형식입니다. C#에서 값 형식들은 변수들이 독립적으로 사용됩니다. 이에 반해 참조 형식은 변수들이 관리화 힙에 할당된 개체를 참조하여 사용되므로 하나의 개체를 사용하는 여러 개의 변수가 있을 수 있습니다. 그리고 참조 형식의 개체는 관리화 힙에 할당되어 .NET에 의해 생명 주기를 관리되어 형식 안정성을 제공합니다.

### 3.1 object

C#의 object는 .NET Framework의 Object 형식을 부르는 형식 이름입니다. 앞서서도 계속 얘기를 했던 것처럼 object 형식은 모든 형식의 기반이 되는 형식입니다. C#에서 사용자 정의 형식을 정의하면 묵시적으로 object 형식에서 파생됩니다. 또한, 개발자는 파생 관계를 문법적으로 표현할 필요도 없고 표현해서도 안 됩니다.

먼저, object 형식을 구성하는 멤버들을 살펴봅시다.

object 형식에서 생성자는 기본 생성자를 제공하고 있습니다. 기본 생성자란 입력 매개 변수가 없는 생성자를 말합니다.

`public object()`

object 형식에서 접근 지정이 public으로 지정된 멤버 메서드는 다음과 같습니다.

▶ object 형식의 public으로 접근 지정된 멤버 메서드

```
public virtual bool Equals(object obj)
public static bool Equals (object obj1, object obj2)
public virtual int GetHashCode()
public Type GetType()
public static bool ReferenceEquals (object obj1,object obj2)
public virtual string ToString ()
```

C#에서 static 키워드가 명시된 멤버를 정적 멤버라고 합니다. 정적 멤버는 형식 명을 통해 접근이 가능한 멤버로 개체의 멤버가 아니라 형식의 멤버입니다. 그리고 virtual 키워드가 있는 메서드는 가상 메서드로 파생 형식에서 재정의할 수 있습니다. 재정의란 기반 형식에서 정의한 것을 무효화시키고 새롭게 정의하는 것을 말합니다. 이에 대한 자세한 사항은 6장 캡슐화와 7장 상속과 다형성에서 자세히 설명하겠습니다.



### 3.1.1 Equals, ReferenceEquals

Equals 메서드와 ReferenceEquals 메서드는 두 개의 개체가 같은지를 비교할 때 사용하는 메서드이며 정적 멤버와 개체의 멤버가 있습니다. 정적 멤버는 입력 매개변수로 전달된 두 개의 인자가 같은지를 판단하며 개체의 멤버는 입력 매개변수로 전달된 인자와 자신이 같은지를 판단합니다. 기본적으로는 참조된 개체가 같은지를 판단하지만 개체의 멤버 Equals는 가상 메서드이므로 목적에 따라 값이 같은지 판단하게 재정의 가능합니다. 그리고 값 형식과 string 형식에서는 개체의 멤버 Equals를 값이 같은지를 판단하도록 재정의되어 있습니다. ReferenceEquals는 참조된 개체가 같은지를 판단합니다.

#### ▶ 값 형식의 Equals, ReferenceEquals 메서드

```
static void Main(string[] args)
{
    int i = 2;
    int j = i;
    Console.WriteLine("i.Equals(j) => {0}", i.Equals(j));
    Console.WriteLine("object.Equals(i,j) => {0}",object.Equals(i, j));
    Console.WriteLine("object.ReferenceEquals(i,j) =>{0}", object.ReferenceEquals(i, j));
}
```

#### ▶ 결과

```
i.Equals(j) => True
object.Equals(i,j) => True
object.ReferenceEquals(i,j) => False
```

위 같은 경우에 i와 j는 값이 같습니다. 값 형식은 개체의 멤버 Equals를 값이 같은지를 판단하도록 재정의가 되어 있어 i.Equals(j) 결과는 참이 됩니다. 정적 멤버 Equals는 내부적으로 입력 인자로 전달된 첫 번째 개체의 멤버 Equals 메서드를 호출하여 결과를 반환하므로 object.Equals(i,j) 결과도 참이 됩니다.정적 멤버 ReferenceEquals는 개체가 같은지를 판별하는데 값 형식은 변수가 독립적으로 할당되어 관리되므로 i와 j는 같을 수 없습니다. 이에 object.RefernceEquals(i,j) 결과는 거짓입니다.

string은 참조 형식이지만 값 형식처럼 개체의 멤버 Equals를 값이 같은지를 판단하도록 재정의되어 있어 값을 비교합니다. 하지만 서로 다른 개체일 경우 ReferenceEquals의 결과는 거짓이 됩니다.

▶ string 형식의 Equals와 ReferenceEquals

```
static void Main(string[] args)
{
    string s1 = "hello";
    string s2 = s1;
    string s3 = string.Format("hello");

    Console.WriteLine("s1.Equals(s2) => {0}", s1.Equals(s2));
    Console.WriteLine("s1.Equals(s3) => {0}", s1.Equals(s3));
    Console.WriteLine("object.ReferenceEquals(s1,s2) =>{0}",
        object.ReferenceEquals(s1, s2));
    Console.WriteLine("object.ReferenceEquals(s1,s3) =>{0}",
        object.ReferenceEquals(s1, s3));
}
```

▶ 결과

```
s1.Equals(s2) => True
s2.Equals(s3) => True
object.ReferenceEquals(s1,s2) => True
object.ReferenceEquals(s1,s3) => False
```

위의 예에서 s1과 s2는 같은 개체를 참조합니다. 그리고 s3는 string.Format 메서드를 이용하여 새로운 개체를 생성하였습니다. s1과 s2는 Equals와 ReferenceEquals의 결과가 모두 참으로 나오지만 s1과 s3는 ReferenceEquals은 거짓이 나오는 것을 확인할 수 있습니다. 하지만 string의 Equals 메서드는 값이 같은지를 비교하여 결과를 반환하므로 s1과 s3의 Equals 결과는 참입니다.

C#에서는 string을 제외한 참조 형식의 Equals 메서드는 참조하는 개체가 같은지를 판단한다는 것에 주의하세요. 물론, 개발자가 재정의하여 목적에 맞게 변경할 수도 있습니다.

▶ 참조 형식의 Equals와 ReferenceEquals

```
class Example
{
    int val;
    public Example(int val)
    {
        this.val = val;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Example e1 = new Example(1);
        Example e2 = new Example(1);
        Console.WriteLine("e1.Equals(e2) => {0}", e1.Equals(e2));
        Console.WriteLine("object.Equals(e1,e2) => {0}", object.Equals(e1, e2));
        Console.WriteLine("object.ReferenceEquals(e1,e2) =>{0}",
            object.ReferenceEquals(e1, e2));
        Example e3 = e1;
        Console.WriteLine("e1.Equals(e3) => {0}", e1.Equals(e3));
        Console.WriteLine("object.Equals(e1,e3) => {0}", object.Equals(e1, e3));
        Console.WriteLine("object.ReferenceEquals(e1,e3) =>{0}",
            object.ReferenceEquals(e1, e3));
    }
}
```

▶ 결과

```
e1.Equals(e2) => False
object.Equals(e1,e2) => False
object.ReferenceEquals(e1,e2) =>False
e1.Equals(e3) => True
object.Equals(e1,e3) => True
object.ReferenceEquals(e1,e3) => True
```

테스트를 위해 Example 클래스를 정의하였습니다. 그리고 테스트는 두 개의 변수가 같은 값을 갖는 다른 개체를 각각 참조할 때와 두 개의 변수가 같은 개체를 참조할 때에 대하여 테스트를 하였습니다. 참조 형식의 Equals 메서드는 같은 개체인지 비교하므로 세 가지 모두 결과가 같습니다. 두 개의 변수가 같은 값을 갖는 다른 개체를 각각 참조하는 경우에는 참조한 개체가 다르므로 값이 같아도 결과는 모두 거짓입니다. 그리고 두 개의 변수가 같은 개체를 참조할 때의 결과는 모두 참입니다.

이번에는 class Example을 struct Example로 변경하여 값 형식일 경우에 어떻게 되는지에 대해 얘기를 해 봅시다. 값 형식은 개체의 멤버 Equals가 값이 같으면 참을 반환하므로 서로 다른 개체일 때도 결과는 참입니다. 더군다나 값 형식은 변수마다 독립적으로 메모리가 할당되고 관리되므로 ReferenceEquals의 결과는 모두 거짓입니다.

▶ 예제 코드의 Example 클래스를 Example 구조체로 변경했을 때 결과

```
e1.Equals(e2) => True
object.Equals(e1,e2) => True
object.ReferenceEquals(e1,e2) =>False
e1.Equals(e3) => True
object.Equals(e1,e3) => True
object.ReferenceEquals(e1,e3) =>False
```

### 3.1.2 GetHashCode

GetHashCode는 해시 테이블과 같은 자료구조를 이용하여 개체를 관리할 때 적합한 키를 반환하는 메서드입니다. 실제 반환되는 값이 어떻게 발생하는지 규칙성이 없고 유일성을 보장하지 않습니다. 확률적으로 같은 값을 반환할 경우가 적다는 이유로 고유한 키로 사용하지 마십시오. 물론, GetHashCode는 가상 메서드이기 때문에 목적에 맞게 재정의할 수 있습니다.

### 3.1.3 GetType

GetType 메서드는 형식에 대한 상세정보를 갖는 Type 개체를 반환합니다. Type 개체는 런 타임에 형식에 대한 상세정보가 필요할 때 사용되는데 여러분들은 이 책을 보시고 난 후에 동적으로 라이브러리를 사용하는 리플렉션에 대해 살펴보세요.

### 3.1.4 ToString

ToString 메서드는 자신에 대한 정보를 문자열로 반환하는 메서드입니다. 기본적으로 형식 이름을 반환합니다. 그리고 ToString은 재정의가 가능한 메서드이며 기본 형식들은 갖고 있는 값을 문자열로 변환하여 반환하고 있습니다. 여러분이 사용자 정의 형식을 만들 때 이를 재정의하여 주요한 키를 문자열로 변환하여 반환하면 많은 곳에서 쉽게 사용할 수 있습니다.

#### ► ToString 메서드 재정의

```
namespace Example
{
    class Stu//Stu 클래스에는 ToString 메서드를 재정의하였음
    {
        string name;
        public Stu(string name)
        {
            this.name = name;
        }
        public override string ToString()//재정의
        {
            return name;
        }
    }

    class Book//Book 클래스에는 ToString 메서드를 재정의하지 않았음
    {
        string name;
        public Book(string name)
        {
            this.name = name;
        }
    }
}
```

```

class Program
{
    static void Main(string[] args)
    {
        //Stu 클래스에는 ToString 메서드를 중복 정의하였음
        Stu stu = new Stu("홍길동");
        Console.WriteLine(stu.ToString()); //명시적으로 ToString 메서드 사용
        Console.WriteLine(stu);

        //Book 클래스에는 ToString 메서드를 중복 정의하지 않았음
        Book book = new Book("IT 전문가로 가는 길 Escort C#");
        Console.WriteLine(book.ToString()); //명시적으로 ToString 메서드 사용
        Console.WriteLine(book);
    }
}

```

▶ 결과

홍길동

홍길동

Example.Book

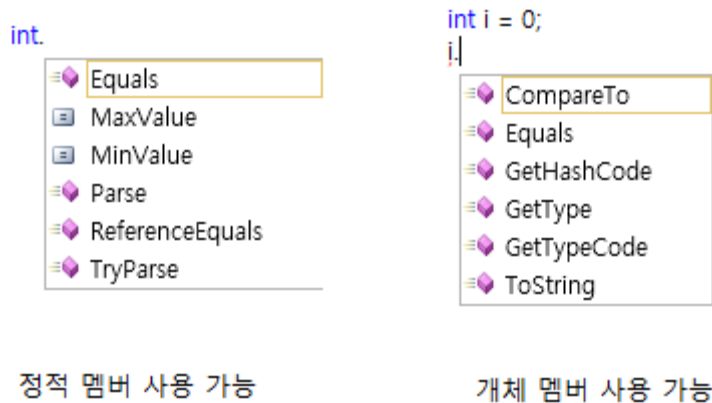
Example.Book

## 3.2 Boxing 과 UnBoxing

C#의 object 형식은 모든 형식의 기반 형식입니다. 여기에서는 값 형식들이 object 형식 변수에 대입하거나 object 개체를 값 형식 변수에 대입하여 사용할 때 어떤 메커니즘이 진행되는지 살펴보기로 합니다.

C#에서 값 형식은 구조체와 열거형으로 나눌 수 있습니다. 기본 형식에서 object와 string을 제외한 모든 기본 형식은 구조체입니다. 구조체는 여러 종류의 멤버들을 캡슐화하여 사용할 수 있습니다. 구조체는 기반 형식이 될 수 없으며 기본 생성자(매개 변수가 없는 생성자)와 소멸자를 선언할 수 없습니다. C#에서는 여러 멤버를 캡슐화하여 사용자 정의 형식을 만들 때 일반적으로 클래스를 사용하지만 단순한 경우에 구조체를 사용할 수도 있습니다.

C언어나 C++언어를 학습해 본 적이 있는 이들은 int와 같은 기본 형식들이 구조체라고 하면 이해가 가지 않을 수 있을 것입니다. 하지만, C#에서는 int는 멤버 필드와 멤버 메서드들을 캡슐화하고 있는 구조체임에 틀림이 없습니다. C#에서는 다음과 같은 표현을 할 수가 있으며 이를 통해 int도 구조체임을 알 수 있습니다.



[그림 3.1] int 형식의 멤버들

C#에서 값 형식들은 스택에 할당되고 참조 형식의 개체는 관리화 힙에 동적으로 할당됩니다. 그리고 C#의 모든 형식은 object에서 파생되었다고 하였습니다. 그러면 object 형식 변수에 값 형식을 대입하면 어디에 할당될까요?

다음 코드를 통해 설명할게요.

```
static void Main(string[] args)
{
    int a = 0;
    a = 4;
    object o = a;//Boxing
    Console.WriteLine(o.ToString());

    int b = 0;
    b = (int)o; //UnBoxing
    Console.WriteLine(b.ToString());
}
```

모든 value 형식은 object에서 파생된 형식이므로 object 형식 변수 o에 int 형식 변수를 대입할 수 있습니다. 이처럼 값 형식을 object에 대입하면 값을 대입 받기 위한 object 개체가 관리화 힙에 생성됩니다. 그리고 해당 개체에 대입한 형식이 int 형식이라는 것과 값을 보관합니다. 이러한 과정을 Boxing이라고 하며 내부적으로 오버헤드가 발생합니다.

```
object o = a;//Boxing
```

그리고 object 개체를 값 형식에 형 변환하여 대입을 하는 과정은 UnBoxing이라고 합니다. UnBoxing과정에서는 object 개체의 실제 형식이 형 변환 요청에 명시한 형식인지 확인하여 맞았다면 보관된 값을 반환합니다. 하지만 형식이 다르다면 예외를 발생합니다.

```
b = (int)o; //UnBoxing
```

이처럼 value 형식과 object 형식 사이에는 Boxing과 UnBoxing이라는 비교적 오버헤드가 큰 작업이 요구됩니다.



### 3.3 배열

여러분도 잘 알다시피 배열은 같은 형식의 여러 개의 요소(원소)를 포함하는 데이터 구조입니다. C#에서는 효과적으로 배열을 표현하기 위해 추상 클래스 Array를 제공하고 있으며 모든 배열은 Array 기반의 파생 클래스 형식 개체입니다. 먼저, 가장 단순한 1차원 배열을 선언하고 사용하는 예를 살펴보기로 합시다.

#### ▶ 1차원 배열의 선언

```
int[] arr1;  
arr1 = new int[5];  
int[] arr2 = new int[5];  
int[] arr3 = new int[] { 1, 2, 3, 4, 5 };  
int[] arr4 = new int[5] { 1, 2, 3, 4, 5 };
```

1차원 배열은 변수 선언 시에 요소 형식과 배열명 사이에 [ ]를 명시하여 선언합니다. 그리고 개체를 생성할 때에는 new 연산자와 요소 형식, [요소 개수]를 명시합니다. 1차원 배열을 선언과 동시에 초기화할 때는 초기값을 { }안에 명시하여 초기화하거나 요소의 개수를 명시하여 개체를 생성하면 됩니다. 만약, 초기값을 명시하지 않으면 기본값으로 초기화가 되며 초기값을 명시하면 요소의 개수를 명시하지 않아도 됩니다.

배열을 사용할 때에는 배열명과 인덱스 연산자([ ]) 및 요소의 상대적 위치(0부터 시작)를 통해 각 요소에 접근할 수 있으며 이 외에 기반 클래스인 Array에서 제공되는 다양한 속성과 메서드를 사용할 수 있습니다.

#### ▶ 배열의 인덱스 연산자 사용

```
int[] arr = new int[4]{10,8,1,6};  
for (int i = 0; i < arr.Length; i++)  
{  
    Console.WriteLine("{0} ", arr[i]);  
}
```

▶ Array 추상 클래스의 정적 메서드 Sort 사용

```
int[] arr = new int[4] { 5, 8, 2, 1 };
Console.WriteLine("요소의 개수:{0}", arr.Length);
foreach (int elem in arr)
{
    Console.WriteLine(elem.ToString());
}
Array.Sort(arr);
Console.WriteLine("정렬 후");
foreach (int elem in arr)
{
    Console.WriteLine(elem.ToString());
}
```

C#에서 다차원 배열은 다음과 같이 선언하여 사용할 수 있습니다.

▶ 다차원 배열 선언

```
int[,] arr1;
arr1 = new int[2, 3];
arr1 = new int[,] { { 1, 2, 3 }, { 4, 5, 6, } };
int[, ] arr2 = new int[4, 2, 3];
int[,] arr3 = new int[,]{{1,2},{4,5},{5,6}};
int[,] arr4 = new int[3,2] { { 1, 2 }, { 4, 5 }, { 5, 6 } };
```

C#에서 다차원 배열을 선언할 때 [ ]안에逗를 통해 차수를 결정할 수 있습니다. 그리고 개체를 생성할 때는 각 차수의 요소 개수를 명시하거나 초기값을 명시하여 개체를 생성해야 합니다. 다차원 배열을 사용할 때도 각 차수의 상대적 거리를 인덱스 연산자 내부([상대적 거리, 상대적 거리])에 명시하여 사용합니다.

▶ 다차원 배열에서 인덱스 연산자([ ]) 사용

```
int[,] arr = new int[3, 2] { { 1, 2 }, { 4, 5 }, { 5, 6 } };
Console.WriteLine("차수:{0} 요소의 개수:{1}", arr.Rank, arr.Length);
arr[2, 1] = 8;
for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < 2; j++)
    {
        Console.WriteLine("arr[{0},{1}]:{2}", i, j, arr[i, j]);
    }
}
foreach (int elem in arr)
{
    Console.WriteLine(elem.ToString());
}
```

C#에서는 가변 길이의 배열을 지원합니다. 가변 배열은 배열을 요소로 하는 배열로 각 요소 배열의 크기를 다르게 지정할 수 있습니다.

▶ 가변 길이의 배열의 선언

```
int[][] arr1;
arr1 = new int[2][];
arr1[0] = new int[3];
arr1[1] = new int[6];
int[][] arr2 = new int[3][];
arr2[0] = new int[3];
arr2[1] = new int[] { 1, 2, 3, 4 };
arr2[2] = new int[2] { 1, 2 };
int[][] arr3 = new int[][] { new int[] { 1, 2, 3 }, new int[] { 4, 5, 6 } };
```

가변 배열은 요소를 지정하지 않으면 초기값인 null로 초기화됩니다. 그리고 각 요소에 배열 개체가 올 수 있으므로 이를 이용하면 다양한 형태의 배열을 사용할 수 있습니다.

▶ 가변 길이 배열의 사용 예

```
int[][] array = new int[][] { new int[] { 1, 2, 3 }, new int[] { 4, 5 } };

foreach (int[] elemArray in array)
{
    foreach (int elem in elemArray)
    {
        Console.Write(elem.ToString());
    }
    Console.WriteLine();
}
```

여기에서는 가장 기본적인 컬렉션인 배열에 대해서만 다루었으며 좀 더 다양한 컬렉션에 대해서는 8장 컬렉션과 인터페이스에서 설명할게요.

### 3.4 string

C#에서는 문자열을 string 형식으로 제공하고 있으며 이는 .NET Framework에서 제공하는 System.String 형식을 C#에서 부르는 형식 이름입니다. string은 참조 형식임에도 값 형식처럼 사용할 수 있습니다. string 개체를 생성하는 방법은 여러 가지가 있습니다. 그 중에서 자주 사용되는 몇 가지를 나열한다면 다음과 같습니다.

▶ string 개체 생성하는 방법

```
string s1;
s1 = "hello";
string s2 = "yahoo";
char[] chArr = new char[] { 'a', 'b', 'c', 'd', 'e', 'f' };
string s3 = new string(chArr);
string s4 = new string(chArr, 2, 3);
string s5 = new string('a', 4);
```

string 형식의 변수를 선언하고 초기화하지 않으면 null을 기본값으로 갖습니다. string 형식의 변수를 선언할 때 초기화를 하지 않는 것은 많은 응용 개발 시에 논리적 버그가 발생하여 개발 비용이 늘어나는 원인이 될 수 있습니다. 될 수 있으면 string 형식의 변수는 선언과 동시에 초기화를 해 주시기 바랍니다. 마땅히 초기화할 값이 없을 때는 string 클래스의 정적 멤버인 Empty를 사용하시기 바랍니다.

```
string str = string.Empty;
```

string 형식에서는 == 연산자와 != 연산자는 값 형식처럼 문자 컬렉션이 같은지를 판단합니다.

▶ string 형식의 ==연산자와 != 연산자

```
char[] chArr = new char[] { 'a', 'b', 'c', 'd', 'e', 'f' };  
string str1 = new string(chArr);  
string str2 = new string(chArr);  
Console.WriteLine(str1 == str2);  
Console.WriteLine(string.ReferenceEquals(str1, str2));
```

▶ 실행 결과

True

False

위의 코드를 실행하면 str1==str2의 결과가 True입니다. 실제 str1과 str2는 서로 다른 개체임에도 컬렉션 내용이 같은지를 가지고 판단하기 때문입니다. 물론, 참조하는 개체를 비교하는 string.ReferenceEquals 메서드를 호출한 결과는 False입니다.

string의 다른 비교 연산자들(<, <=, >, >=)은 제공하지 않습니다. 대신 멤버 메서드로 CompareTo를 제공함으로써 이를 대신할 수 있습니다.

```
int difference = str1.CompareTo(str2);
```

string은 + 연산자를 사용하여 단일 문자열을 만들 수 있습니다.

```
string s1 = "hello";  
string s2 = "yahoo";  
string s3 = s1 + " " + s2 + " abc";
```

특정 포맷에 맞는 문자열을 만들고자 한다면 string 클래스의 정적 메서드인 Format을 사용하면 됩니다.

```
string str = string.Format("이름:{0,-5} 나이:{1,10}", "홍길동", 23);
```

이 외에도 string에는 다양한 멤버들을 제공하고 있어 다양한 작업을 할 수 있습니다.

다음은 문자열을 수정할 때 사용되는 메서드의 예제 코드입니다. 각 메서드에서는 원하는 작업을 수행한 결과 문자열을 반환하며 원본 문자열은 바뀌지 않습니다.

#### ▶ 문자열 수정

```
string s = " ab cde";  
Console.WriteLine(s.Insert(3, "EHCLUB")); //부분 문자열 삽입  
//문자열 길이가 최소 10이 될 수 있게 앞쪽에 공백 추가  
Console.WriteLine(s.PadLeft(10));  
//문자열 길이가 최소 10이 될 수 있게 앞쪽에 '*' 추가  
Console.WriteLine(s.PadLeft(10, '*'));  
//문자열 길이가 최소 10이 될 수 있게 뒤쪽에 공백 추가  
Console.WriteLine(s.PadRight(10));  
//문자열 길이가 최소 10이 될 수 있게 뒤쪽에 '*' 추가  
Console.WriteLine(s.PadRight(10, '*'));  
Console.WriteLine(s.Remove(2)); //인덱스 2부터 부분 문자열 제거  
Console.WriteLine(s.Remove(2, 2)); //인덱스 2부터 2개의 부분 문자 제거  
//소문자를 대문자로 변경 (ToLower는 대문자를 소문자로 변경)  
Console.WriteLine(s.ToUpper());  
Console.WriteLine(s.Trim()); //문자열 앞쪽과 뒤쪽에 있는 공백을 제거
```

string에서는 부분 문자열이 존재 여부를 확인하기 위한 메서드들과 위치를 찾아주는 메서드들을 통해 검색에 편의성을 제공하고 있습니다.

▶ 부분 문자열 검색

```
string str = "I am a boy. You are a girl.";
Console.WriteLine(str.Contains("boy"));           //부분 문자열 포함 여부
Console.WriteLine(str.EndsWith("girl."));         //문자열 끝 부분 비교
Console.WriteLine(str.StartsWith("You"));          //문자열 시작 부분 비교
Console.WriteLine(str.IndexOf('.').ToString());    //특정 문자가 처음 발견되는 인덱스 반환

//특정 문자열이 처음 발견되는 인덱스 반환
Console.WriteLine(str.IndexOf("You").ToString());

//특정 문자가 뒤에서 처음 발견되는 인덱스 반환
Console.WriteLine(str.LastIndexOf('a').ToString());

//특정 문자열이 뒤에서 처음 발견되는 인덱스 반환
Console.WriteLine(str.LastIndexOf(" a").ToString());
```

▶ 실행 결과

```
True
True
False
10
12
20
19
```

이 외에도 string 형식에서는 다양한 멤버들을 제공하고 있으니 사용해 보시기 바랍니다.

## 4. 값(value) 형식

C#에서 값 형식에는 구조체와 열거형이 있으며 암시적으로 System.ValueType에서 파생된 형식입니다. 형식개요에서 살펴본 바와 같이 값 형식은 각각의 변수가 독립적인 값을 갖게 되며 하나의 변수에서 다른 변수에 대입하였을 때 값을 복사합니다.

### 4.1 구조체

C#에서는 int 형식이나 bool 형식과 같은 모든 단순 형식들도 구조체입니다. 그리고 다른 많은 언어처럼 여러 멤버를 갖는 사용자 정의 구조체를 만들 수 있습니다.

#### 4.1.1 정수

C#에서는 표현할 수의 범위에 따라 여러 종류의 정수 형식을 제공하고 있습니다.

대부분 키워드가 C언어나 C++언어와 같고 할당되는 메모리 크기와 표현 범위가 비슷합니다. 이러한 이유로 무의식적으로 C#에서도 할당되는 메모리 크기나 표현 범위가 같다고 생각하면서 프로그래밍할 수 있고 이 때문에 논리적 오류를 갖게 될 수 있습니다. C#에서 char는 C언어와 C++언어와 다르게 2바이트가 할당되며 유니코드로 표현할 수 있습니다. 예를 들어, C언어에서는 ASCII 코드로 표현하기 때문에 한글의 한 글자를 하나의 문자로 표현하지 못하고 문자열로 표현해야 했지만, C#에서는 유니코드로 표현하므로 하나의 문자로 표현할 수 있습니다. 그리고 long 형식은 C언어에서 4바이트였지만 C#에서는 8바이트(64비트)를 할당받아 int 형식보다 넓은 범위의 수를 표현할 수 있습니다.

형식	메모리 크기	표현 범위
sbyte	1바이트	-128~127
byte	1바이트	0~255
char	2바이트	0~65535(유니코드)
short	2바이트	-32768~32767
ushort	2바이트	0~65535
int	4바이트	-2147483648~2147483647
uint	4바이트	0~4294967295
long	8바이트	-9223372036854775808~9223372036854775807
ulong	8바이트	0 ~ 18446744073709551615



int 와 같은 기본 형식이 구조체라는 사실은 변수나 형식을 통해 보관된 값 이외에도 사용할 수 있는 멤버들이 있다는 것에서 확인할 수 있습니다.

```
int max = int.MaxValue;
int min = int.MinValue;
Console.WriteLine(max.ToString());
Console.WriteLine(min.ToString());
```

일반적으로 가장 많이 사용하게 될 멤버로는 최종 사용자가 입력한 값을 기본 형식으로 변환할 때 사용하는 정적 멤버 메서드인 Parse와 TryParse입니다. Parse와 TryParse 메서드는 사용하는 목적에 따라 입력 인자가 다르게 중복 정의(overload)되어 있습니다. 다음은 중복 정의되어 있는 목록 중에 하나입니다.

```
public static int Parse (string s);
public static bool TryParse (string s,out int result);
```

Parse 메서드는 문자열을 입력인자로 받아 문자열을 정수로 변환하여 반환합니다. 만약, 변환할 수 없는 문자가 포함되어 있을 때는 예외를 발생합니다. TryParse 메서드는 첫 번째 입력 인자로 전달된 문자열을 정수로 변환하여 output 전달 형식의 두 번째 매개 변수에 대입해 줍니다. 만약, 정상적으로 변환되면 true를 반환하고 변환할 수 없을 때는 false를 반환합니다.

```
int i1 = int.Parse("-123");
Console.WriteLine(i1.ToString());

int i2 = int.Parse("123");
Console.WriteLine(i2.ToString());

//int i3 = int.Parse("a-123"); 변환할 수 없는 문자가 있기 때문에 예외가 발생합니다.
//Console.WriteLine(i3.ToString());

//int i4 = int.Parse("123a"); 변환할 수 없는 문자가 있기 때문에 예외가 발생합니다.
//Console.WriteLine(i4.ToString());
```

만약, 다음과 같이 최종 사용자로부터 수를 입력받는 구문이 있다면 변환할 수 없는 문자가 포함된 문자열을 입력하면 예외가 발생합니다. 개발자가 테스트할 때 잘못 입력할 때를 생각하지 못하면 버그가 있는 프로그램을 만들게 됩니다.

```
Console.WriteLine("수를 입력하세요.");  
int num = int.Parse(Console.ReadLine());  
Console.WriteLine("입력한 수는 {0}입니다.", num);
```

이럴 때 개발자는 Parse 메서드를 대신하여 TryParse 메서드를 이용할 수 있습니다.

```
Console.WriteLine("수를 입력하세요.");  
int num;  
if (int.TryParse(Console.ReadLine(), out num))  
{  
    Console.WriteLine("입력한 수는 {0}입니다.", num);  
}  
else  
{  
    Console.WriteLine("잘못된 수를 입력하였습니다..");  
}
```

char 형식은 문자 리터럴 상수 표현을 이용할 수 있습니다. 이는 C언어나 C++언어와 마찬가지로 표현할 문자를 콤마로 감싸 표현하는 것입니다. 앞에서 얘기했듯이 C#언어는 유니코드로 표현하므로 한글도 하나의 문자로 표현할 수 있습니다.

```
char c = '가';  
Console.WriteLine(c.ToString());
```

그리고 char 형식은 묵시적으로 정수 형식과 형식 변환을 제공하고 있습니다.

```
int cval = c;  
Console.WriteLine(cval.ToString());
```

#### 4.1.2 부동 소수점 형식

C#에서는 실수를 표현하기 위해 float과 double 형식을 제공하고 있습니다. 그런데 실수는 0.1에서 0.2 사이에도 무한개의 실수가 존재하기 때문에 컴퓨터 프로그램에서는 정확한 수치를 저장하지 못하고 근사치를 보관합니다.

형식	메모리 크기	표현 범위
float	4바이트	$\pm 1.5e-45 \sim \pm 3.4e38$
double	8바이트	$\pm 5.0e-324 \sim \pm 1.7e308$

예를 들어, 0.0으로 초기화된 변수에 0.1씩 더해나가면 10번 반복했을 때 1.0이 될 것으로 생각할 수 있습니다. 하지만, 정확한 수치가 아닌 근사치이기 때문에 다음의 코드를 수행하면 반복문을 탈출하지 못합니다.

```
double d = 0.0;
while (d != 1.0) //무한 루프 - 0.1을 정확히 표현하지 못하기 때문에 1.0이 되지 못함
{
    Console.WriteLine(d.ToString());
    d = d + 0.1;
}
```

#### 4.1.3 decimal

C#에서는 10진수 표현에 적합한 decimal 형식을 제공하고 있습니다. decimal도 근사치를 표현하지만 오차 범위내에서 10진수로 실수 표현을 할 때 효과적으로 프로그래밍할 수 있습니다. double 형식을 사용했을 때 무한루프에 빠졌던 예제 코드를 decimal 형식으로 바꾸면 원하는 결과를 얻을 수 있음을 알 수 있습니다.

```
decimal d = 0.0m;
while (d != 1.0m)
{
    Console.WriteLine(d.ToString());
    d = d + 0.1m;
}
```

10진수로 오차 범위(96비트 정수, 소수 자리수 10의 28)내에서 표현하면 부동 소수점 표현보다 정확하게 표현할 수 있습니다.

또한, C#에서는 수를 표현하는 다양한 형식들에 대해 표현 범위가 좁은 형식을 범위가 넓은 형식으로 암시적 형식 변환을 대부분 지원합니다.

형식	변환 형식
sbyte	short, int, long, float, double, decimal
byte	short, ushort, int, uint, long, ulong, float, double, decimal
char	ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long	float, double, decimal
ulong	float, double, decimal
float	double

#### 4.1.4 bool

C#에서는 true와 false를 값으로 표현할 수 있는 bool 형식을 제공하고 있습니다.

#### 4.1.5 사용자 정의 구조체

C#에서는 사용자에게 의해 프로그램에 필요한 형식을 정의할 수 있게 구조체와 열거형 및 클래스를 지원하고 있습니다. 특히, 구조체와 클래스는 여러 개의 멤버들을 하나의 형식으로 캡슐화하여 목적에 맞게 정의할 수 있으며 많은 부분에서 비슷한 문법을 제공합니다. 이 책에서는 구조체에 대한 문법 사항을 상세하게 얘기하지 않고 클래스와 공통적인 문법 사항을 OOP의 특징별로 설명하고 있습니다.

구조체를 정의하여 사용하는 방법은 클래스와 함께 다음 장부터 자세히 다루겠습니다.

## 4.2 열거형

열거형은 enum 키워드를 사용하여 상수 집합 목록을 열거자 목록에 선언하여 사용될 수 있는 값의 종류를 사용자가 정의하는 형식입니다. 열거형을 정의할 때 나열할 수 있는 상수 집합 목록의 요소는 기본적으로 정수(int) 형식입니다.

```
enum Season{ NonSeason, Spring, Summer, Autumn, Winter }
```

위의 코드는 계절을 Season 이름의 열거형을 정의한 예입니다. 이 경우에 NonSeason부터 차례대로 0, 1, 2, 3, 4에 해당하는 값으로 정의됩니다. NonSeason은 Season 형식의 기본값으로 사용할 수 있게 선언한 것으로 반드시 있을 필요는 없지만 적절한 값이 아님을 나타내기 위해 선언하였습니다. MSDN에서는 될 수 있으면 0값을 사용하지 말거나 적절한 값이 아님을 나타내기 위한 값을 선언하여 디폴트 값으로 사용하라고 권고하고 있습니다.

### ▶ 열거형 사용 예

```
class Program
{
    enum Season { NonSeason, Spring, Summer, Autumn, Winter };
    static void Main(string[] args)
    {
        Season season = Season.NonSeason;
        Console.WriteLine("계절을 입력하세요. 봄:1 여름:2 가을:3 겨울:4 ");
        season = (Season)int.Parse(Console.ReadLine());
        Console.WriteLine("입력한 계절은 {0}입니다.", season);
    }
}
```

### ▶ 실행 결과

```
계절을 입력하세요. 봄:1 여름:2 가을:3 겨울:4
2
입력한 계절은 Summer입니다.
```

그리고 개발자는 열거형 형식에 선언한 각 열거 목록의 값을 명시적으로 지정할 수 있습니다. 목록에 명시되지 않은 것은 앞 목록의 값+1이 됩니다.

```
enum BaseScore
{
    MinScore, Bad = 60, SoSo = 75, Good = 90, HighScore = 100
}
```

C#에서의 열거형에 선언된 목록의 기본값은 int 형식을 기반으로 하지만 다른 정수 형식으로 정의할 수도 있습니다. 참고로 char 형식은 불가능합니다.

```
enum Denomination: long
{
    Kilo = 1000, Mega = 1000000, Giga = 1000000000,
    Tera = 1000000000000, Peta = 1000000000000000
}
```

그리고 C#에서는 비트 연산을 통해 여러 열거 목록을 표현하기 쉬울 때 Flags Attribute를 명시하여 열거형을 정의하는 방법을 지원합니다.

```
[Flags]
enum MyFlag{ HasCar = 0x01, Married = 0x02, HasHouse = 0x04 }
class Program
{
    static void Main(string[] args)
    {
        MyFlag flag = MyFlag.HasCar | MyFlag.Married | MyFlag.HasHouse;
        Console.WriteLine("{0}", flag);
    }
}
```

▶ 결과

HasCar, Married, HasHouse

## 5. 캡슐화

C#에서는 프로그램 목적에 맞게 사용자가 형식을 정의할 수 있습니다. C#에서 사용자가 형식을 정의하기 위한 문법으로 클래스와 구조체, 열거형 등을 제공하고 있는데 열거형에 대해서는 앞에서 다루었으며 여기에서는 클래스와 구조체에 대해 알아보시다.

클래스와 구조체는 여러 개의 멤버를 하나의 형식으로 묶어 하나의 형식으로 정의할 수 있습니다. 이와 같은 작업을 캡슐화라고 하는데 C언어에서는 구조체만 제공하고 있으며 캡슐화 대상도 멤버 필드(멤버 변수)로 한정되어 있었습니다.

C++언어에서는 클래스를 제공하고 구조체와 클래스에 캡슐화 대상이 멤버 필드뿐만 아니라 멤버 메서드도 캡슐화 대상입니다. 그리고 C언어는 캡슐화된 멤버에 접근 지정자를 제공하고 없어서 모든 멤버에 접근할 수 있습니다. 이는 정보 은닉이 필요한 멤버들을 충분히 은닉하지 못하게 되어 데이터의 신뢰성이 낮아집니다. 하지만 C++과 C#에서는 캡슐화하는 멤버에 접근 지정할 수 있어 데이터의 신뢰성을 높일 수 있습니다. 특히, C#에서는 서로 다른 모듈 사이에서의 접근 지정자를 지정하기 쉽게 되어 있어 컴포넌트 기반의 프로그램을 제작하는 경우에 더욱 높은 신뢰성을 추구할 수 있습니다.

그리고 C#에서는 다른 클래스를 기반으로 파생 클래스를 정의할 수 있도록 상속을 제공합니다. 상속을 사용하면 여러 형식의 공통적인 부분을 기반 형식에서 정의하고 이를 기반으로 파생 형식에서는 다른 부분만 정의할 수 있습니다. 또한, 기반 형식의 변수로 파생된 개체를 참조할 수 있고 파생 클래스에서 기반 클래스에 정의한 멤버 메서드를 재정의할 수 있습니다. 이러한 OOP의 특징을 다형성이라 합니다. 이번 장에서는 캡슐화에 대해서만 다루고 상속과 다형성은 다음 장에서 설명할게요.

C#에서 캡슐화에 대한 문법 사항은 크게 캡슐화할 수 있는 대상은 무엇인가에 대한 사항과 정보의 신뢰성을 유지하기 위해 접근 한정자 그리고 정적 멤버와 비 정적 멤버가 있습니다. 이번 장에서는 이러한 캡슐화에 대한 문법과 참조 형식인 클래스와 값 형식인 구조체를 사용할 때 차이에 대해서 간략하게 다루겠습니다.

그리고 C#에서 캡슐화 문법은 구조체와 클래스가 대부분 같거나 유사하므로 여기에서는 클래스를 예로 설명하겠습니다. 물론, 서로 다른 문법이 있을 때는 별도로 언급할게요.

## 5.1 캡슐화 대상(멤버)

C#에서 클래스나 구조체를 정의할 때 다양한 멤버들을 캡슐화가 가능합니다. 가장 기본적인 멤버는 데이터를 캡슐화하기 위한 멤버 필드와 수행할 작업에 대한 논리적 코드를 정의하는 메서드입니다. 그리고 사용하는 곳에서는 캡슐화된 데이터처럼 보이지만 실제로는 수행할 작업을 정의할 수 있는 특별한 메서드인 속성을 제공합니다.

배열이나 연결 리스트처럼 요소 개체들을 보관하는 컬렉션에서는 사용자가 인덱스 연산을 통해 요소에 접근할 수 있는 인덱서를 제공하고 있습니다. 그리고 정적 멤버인 상수 멤버와 읽기 전용이 있습니다. 이 외에도 개체를 생성할 때 수행할 작업을 정의할 수 있는 생성자와 메모리에서 개체를 제거할 때 작업을 정의하는 소멸자가 있습니다. 그리고 형식 내에 서브 형식을 정의할 수 있으며 연산자를 사용하였을 때의 작업을 정의할 수도 있습니다.

이 외에도 콜백에서 자주 사용하는 대리자(delegate)형식의 이벤트 멤버를 캡슐화할 수 있는데 이에 관한 사항은 대리자와 이벤트에서 설명할게요.

### 5.1.1 멤버 필드

멤버 필드는 클래스나 구조체의 캡슐화되어 있는 일부 데이터입니다. 멤버 필드는 클래스나 구조체 블록 내에 멤버 형식 및 필드 이름을 차례로 선언하면 됩니다. 또한, C#에서는 선언과 동시에 초기값을 지정할 수 있습니다.

```
class Man
{
    string name;
    int hp = 0; //멤버 필드 초기화
}
```

접근 한정자에 대한 설명하면서 다시 다루겠지만 멤버 필드는 형식 외부에서 접근할 수 없게 private으로 지정하는 것이 바람직합니다. C#에서는 접근 한정자를 명시하지 않을 때 private으로 설정되어 형식 외부에서 접근할 수 없습니다. 만약, 형식 외부에서 멤버 필드의 값을 얻어오거나 설정할 필요성이 있으면 멤버 속성이나 멤버 메서드를 이용하여 제공하는 것이 바람직합니다.



### 5.1.2 멤버 속성

멤버 속성은 멤버 필드에 있는 값을 얻어오거나 변경할 때 사용할 수 있게 제공하는 특별한 메서드입니다. 멤버 속성을 캡슐화하기 위해서는 형식과 속성 명을 선언하고 전용 필드의 값을 얻어올 때 사용하는 get 블록과 설정하는 set 블록을 선택적으로 정의할 수 있습니다. 각 블록에서는 메서드처럼 내부에서 수행할 작업에 대한 코드를 작성할 수 있으며 필요에 따라 get 블록과 set 블록의 접근 한정을 다르게 지정할 수도 있습니다. get 블록에서는 선언한 형식을 반환해야 하고 set 블록에서는 value 이름으로 전달된 값을 사용할 수 있습니다.

```
class Man
{
    string name;
    int hp = 0;

    public string Name //멤버 속성 - get 블록만 선택적으로 정의
    {
        get
        {
            return name;
        }
    }

    public int Hp //멤버 속성 - public으로 접근 지정
    {
        get //멤버 속성에 대한 접근 지정을 따름(public)
        {
            return hp;
        }
        private set //private으로 접근 지정
        {
            hp = value; //value를 전달 받은 값을 사용할 수 있음
        }
    }
}
```

이와 같은 멤버 속성은 멤버 필드의 신뢰성을 높이는 데 큰 역할을 할 수 있습니다. 외부에서 멤버 필드를 사용할 수 있게 접근을 허용한다면 잘못된 사용으로 신뢰성 없는 값을 갖게 될 수 있습니다. 예를 들어 사람의 hp를 0에서 200사이 값이 유지하게 하고 일을 하면 hp가 5가 증가하게 하려고 합니다. 사람 형식의 멤버 필드 hp 접근을 public으로 지정하여 형식 외부에서 접근할 수 있게 하면 사용하는 곳에서 잘못된 값으로 hp를 지정하는 경우가 발생할 수 있습니다.

▶ 신뢰성이 없는 값을 사용한 예

```
class Man
{
    public string name; //멤버 필드의 접근을 public으로 지정
    public int hp = 0; //멤버 필드의 접근을 public으로 지정
    public Man(string name)
    {
        this.name = name;
    }
    public void Work()
    {
        hp += 5;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Man man = new Man("홍길동");
        Console.WriteLine("{0} HP:{1}", man.name, man.hp);
        man.hp = 937; //잘못된 사용
        Console.WriteLine("{0} HP:{1}", man.name, man.hp);
    }
}
```

이 같은 경우에 개발자 사이에서는 Man을 잘못 정의한 것인지 사용을 잘못된 것인지 의견이 서로 다를 수 있습니다. OOP에서는 Man을 정의하는 곳에서 사용하는 곳에서 접근할 수 있는 멤버를 적절하게 지정할 수 있게 해 주는 문법을 제공합니다. 따라서 C#과 같은 OOP 언어에서는 위 경우 Man을 잘못 정의한 것으로 볼 수 있습니다. 다음 예는 멤버 필드의 접근을 막고 이에 대해 필요한 수준으로 접근 가능한 멤버 속성을 정의한 예입니다.

▶ 신뢰성을 높인 예

```
class Man
{
    string name; //멤버 필드
    int hp = 0; //멤버 필드

    public Man(string name) //생성자
    {
        this.name = name;
    }

    public string Name //멤버 속성
    {
        get{ return name; }
    }

    public int Hp //멤버 속성
    {
        get{ return hp; }
        private set //설정은 내부에서만 접근 가능하게 지정
        {
            if(value > 200){ value = 200; }
            if(value < 0) { value = 0; }
            hp = value;
        }
    }
}
```

```

public void Work()
{
    Hp += 5;
}
}

class Program
{
    static void Main(string[] args)
    {
        Man man = new Man("홍길동");
        Console.WriteLine("{0} HP:{1}", man.Name, man.Hp);
        man.Work();
        Console.WriteLine("{0} HP:{1}", man.Name, man.Hp);
    }
}

```

단순히 get 블록에서는 값을 반환하기만 하고 set 블록에서는 필터링 없이 value를 설정하기를 원한다면 멤버 필드를 선언하지 않고 멤버 속성의 get 블록과 set 블록을 선언만 해도 됩니다.

```

class Foo
{
    public int Exp{ get; set; }
}

```

그리고 C# 컴파일러에서는 Hp라는 멤버 속성의 get 블록은 get\_Hp, set 블록은 set\_Hp 이름의 멤버 메서드로 번역합니다. ildasm.exe 유틸리티를 이용하여 il 코드를 덤프하여 확인해 보세요.

### 5.1.3 메서드와 매개 변수 전달 방식

메서드는 수행해야 할 작업에 대한 코드가 있는 블록입니다. 메서드는 이름과 수행에 필요한 입력 매개 변수와 수행한 결과 형식을 선언하고 블록 내에서 수행할 코드를 정의해야 합니다. 메서드의 입력 매개 변수는 여러 개가 올 수 있으면 콤마를 통해 구분하게 됩니다. 그리고 반환 형식은 하나만 정의할 수 있으며 필요가 없으면 void 로 선언합니다.

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

입력 매개 변수는 전달하는 방법에는 값으로 전달하는 방법과 참조로 전달하는 방법이 있습니다. 선언문에 단순히 형식과 변수 이름만 명시하면 값을 복사하여 전달되며 out이나 ref를 명시하면 참조로 전달됩니다. 값을 복사하여 전달하면 호출한 곳의 변수와 호출 받은 곳의 변수는 독립적이며 참조로 전달되면 두 곳의 변수가 같은 개체를 참조합니다.

#### ▶ 값 방식과 참조 방식으로 매개 변수 전달

```
static void Main(string[] args)
{
    int a = 2, b = 2, c = 2;
    Foo(a, ref b, out c);
    Console.WriteLine("Main 메서드 a:{0} b:{1} c:{2}", a, b, c);
}

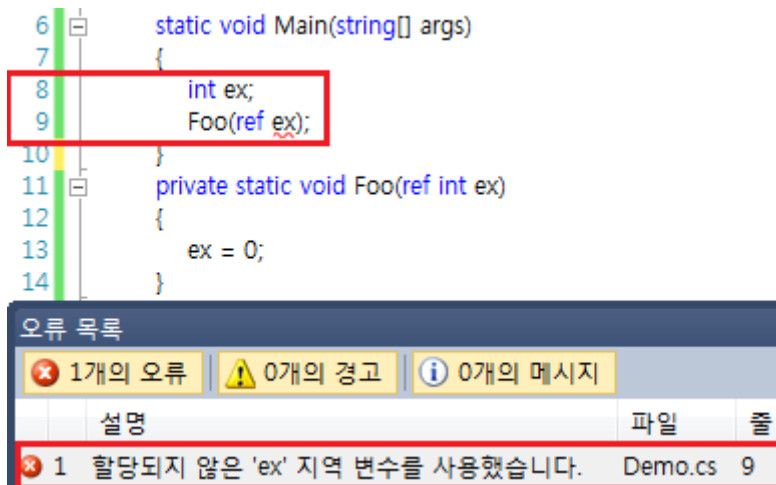
private static void Foo(int a, ref int b, out int c)
{
    a = 3;
    b = 3;
    c = 3;
    Console.WriteLine("Foo 메서드 a:{0} b:{1} c:{2}", a, b, c);
}
```

#### ▶ 실행 결과

```
Foo 메서드 a:3 b:3 c:3
Main 메서드 a:2 b:3 c:3
```

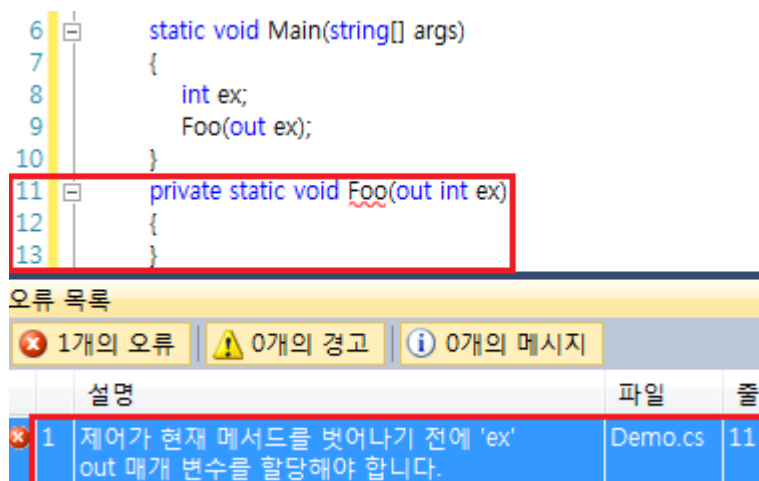
참조 방식으로 전달하는 방식에는 out과 ref이 있습니다.

ref 방식으로 전달하는 매개 변수는 메서드를 수행하는 데 필요한 인자이면서 메서드 내에서 변경되면 호출한 곳에서도 알 필요가 있을 때 사용됩니다. 이러한 이유로 ref 매개 변수는 호출하는 곳에서는 반드시 의미 있는 값을 가진 변수를 전달해야 합니다.



[그림 5.1] 의미없는 값을 가진 ref 인자 전달 시 오류 화면

out 방식으로 전달되는 매개 변수는 반환 형식이 하나라는 한계를 극복하기 위해 사용되며 호출하는 곳에서 전달하는 변수의 값을 초기화하지 않아도 됩니다. 대신 피 호출 메서드 내에서는 반드시 out 형식의 매개 변수의 값을 변경해야 합니다.



[그림 5.2] out 매개 변수를 변경하지 않았을 때 발생하는 오류 화면

다음의 예는 두 수의 차가 예측한 값과 같은지를 판별하고 실제 차이 값을 확인하는 메서드를 통해 ref 매개 변수를 간략하게 살펴봅시다. GapCheck 메서드는 두 수의 차가 ref 매개 변수로 전달된 값과 같으면 true를 반환하고 그렇지 않으면 ref 매개 변수에 차이를 대입하고 false를 반환합니다.

▶ 전달 방식이 ref인 매개 변수 사용 예

```
static void Main(string[] args)
{
    int a = 10, b = 3;
    int gap = 8;
    Console.WriteLine("예상 {0},{1} 차이: {2}",a,b,gap);
    if (GapCheck(a, b, ref gap))
    {
        Console.WriteLine("예측이 맞았군요.");
    }
    else
    {
        Console.WriteLine("예상과 다르군요.");
    }
    Console.WriteLine("실제 {0},{1} 차이: {2}",a,b,gap);
}

private static bool GapCheck(int a, int b, ref int gap) //gap:예상 값
{
    if ((a - b) == gap)
    {
        return true;
    }
    gap = a - b; //실제 차이를 대입
    return false;
}
```

▶ 실행 결과

예상 10,3 차이:7  
예상과 다르군요.  
실제 10,3 차이:7

다음은 두 수의 합과 차를 계산해 주는 메서드에 대한 예입니다. AddAndGap 메서드에 서는 두 수의 합을 반환하고 두 수의 차는 out 방식으로 전달한 변수를 통해 알 수 있습니다.

▶ out 매개 변수 사용 예

```
class Program
{
    static void Main(string[] args)
    {
        int a = 2, b = 3;
        int sum, gap;
        sum = AddAndGap(a, b, out gap);
        Console.WriteLine("sum:{0} gap:{1}", sum, gap);
    }

    //두 수의 합을 반환하고 두 수의 차이를 gap(out 방식으로 전달)에 대입하는 메서드
    private static int AddAndGap(int a, int b, out int gap)
    {
        gap = a - b;
        return a + b;
    }
}
```

▶ 실행 결과

sum:5 gap:-1



값과 참조에 관한 얘기는 형식에서도 언급한 바가 있는데 여기에서 얘기한 것은 매개 변수 전달 방식에 대한 것입니다. 앞의 예는 모두 값 형식의 인자를 값 방식과 참조 방식으로 전달하는 예를 들었는데 여기에서는 참조 형식의 인자를 값 방식과 참조 방식으로 전달하는 예를 들기로 하겠습니다.

이들에 대한 설명에 앞서 값 형식과 참조 형식의 차이에 대한 간략한 설명과 예를 들어 볼게요.

3장 형식 개요에서 값 형식의 변수는 변수 선언을 통해 메모리가 할당되어 변수마다 독립적으로 할당된 메모리를 사용할 수 있다고 하였습니다. 이에 다른 변수를 선언하고 대입을 하면 각 변수를 위해 할당된 메모리는 독립적이므로 단순히 값이 복사됩니다.

이에 반해 참조 형식의 변수는 변수 선언이 아닌 개체 생성을 통해 메모리가 할당되고 변수는 개체를 참조합니다. 이에 개체를 참조하는 변수를 다른 변수에 대입하면 두 개의 변수는 같은 개체를 참조하게 됩니다. 따라서 하나의 변수로 변경하면 다른 변수도 같은 개체를 참조하므로 개체의 정보를 확인해 보면 같이 변하는 것을 알 수 있습니다.

#### ▶ 값 형식과 참조 형식의 차이

```
using System;

namespace Ex_Method
{
    struct ExStruct //구조체 - 값 형식
    {
        public int Val{ get; set; }
    }

    class ExClass //클래스 - 참조 형식
    {
        public int Val{ get; set; }
    }
}
```

```

class Program
{
    static void Main(string[] args)
    {
        //값 형식에 관한 테스트 코드
        ExStruct es1 = new ExStruct();
        es1.Val = 4;
        ExStruct es2 = es1;
        es2.Val = 5;
        Console.WriteLine("es1.Val:{0}, es2.Val:{1}", es1.Val, es2.Val);

        //참조 형식에 관한 테스트 코드
        ExClass ec1 = new ExClass();
        ec1.Val = 4;
        ExClass ec2 = ec1;
        ec2.Val = 5;
        Console.WriteLine("ec1.Val:{0}, ec2.Val:{1}", ec1.Val, ec2.Val);
    }
}

```

▶ 실행 결과

es1.Val:4, es2.Val:5

ec1.Val:5 ec2.Val:5

이제 참조 형식을 값 방식으로 전달하는 경우에 대해 살펴봅시다.

값 형식을 값 방식으로 전달하면 호출한 곳과 피 호출 메서드의 변수는 독립적입니다. 단지 호출한 곳의 값을 복사하여 전달받는 것이며 피 호출 메서드에서 변수의 값을 변경하였을 때 호출한 곳의 변수는 아무런 영향을 받지 않습니다.

하지만 참조 형식을 값 방식으로 전달하면 호출한 곳과 피 호출 메서드의 변수는 같은 개체를 참조하게 됩니다. 따라서 피 호출 메서드에서 개체의 정보를 변경하였을 때 호출한 곳의 변수도 같은 개체를 참조하므로 변경된 정보를 알 수 있습니다.

▶ 값 형식과 참조 형식을 값 방식으로 매개 변수 전달했을 때 비교

```
class Program
{
    static void Main(string[] args)
    {
        ExStruct es = new ExStruct();
        ExampleValueType(es); // 값 형식을 값 방식으로 전달
        Console.WriteLine("es.Val:{0}", es.Val);
        ExClass ec = new ExClass();
        ExampleReferenceType(ec); // 참조 형식을 값 방식으로 전달
        Console.WriteLine("ec.Val:{0}", ec.Val);
    }
    private static void ExampleReferenceType(ExClass ec)
    {
        ec.Val = 5;
    }
    private static void ExampleValueType(ExStruct es)
    {
        es.Val = 5;
    }
}
```

▶ 실행 결과

es.Val:0

ec.Val:5

이번에는 참조 형식을 참조 방식으로 전달하는 경우에 대해 살펴보기로 합시다.

참조 형식을 값 방식으로 전달하더라도 피 호출 메서드에서 개체의 정보가 변경되었을 때 호출한 곳에서도 같은 개체를 참조하므로 해당 개체의 정보 변화를 알 수 있습니다. 그렇다면 언제 참조 형식을 참조 방식으로 전달해야 할까요?

만약, 참조 형식을 값 방식으로 전달받은 메서드에서 새로운 개체를 생성하여 대입하면 어떻게 될까요? 이 경우에 호출하는 곳에서는 호출 시에 전달한 개체를 참조하고 있기 때문에 피 호출 메서드에서 새로운 개체를 생성한 것을 알 수가 없게 됩니다. 호출한 곳에서도 새로 생성한 개체를 알아야 한다면 참조 방식으로 전달해야 합니다.

▶ 참조 형식을 값 방식과 ref 방식으로 전달했을 때 비교

```
static void Main(string[] args)
{
    ExClass ec1 = new ExClass();
    ExampleValueParam(ec1); //참조 형식을 값 방식으로 전달
    Console.WriteLine("ec1.Val:{0}", ec1.Val);
    ExClass ec2 = new ExClass();
    ExampleReferenceParam(ref ec2); //참조 형식을 ref 방식으로 전달
    Console.WriteLine("ec2.Val:{0}", ec2.Val);
}

private static void ExampleReferenceParam(ref ExClass ec)
{
    ec = new ExClass();
    ec.Val = 5;
}

private static void ExampleValueParam(ExClass ec)
{
    ec = new ExClass();
    ec.Val = 5;
}
```

▶ 실행 결과

```
ec1.val:0
ec2.Val:5
```

#### 5.1.4 인덱서

인덱서는 멤버 요소들로 구성된 컬렉션 개체의 요소에 쉽게 접근할 수 있게 해 주는 멤버입니다. 인덱서는 매개 변수가 있다는 점을 제외하면 구현 방법이 속성과 매우 흡사하며 속성처럼 get 블록과 set 블록을 선택적으로 정의할 수 있습니다.

인덱스를 캡슐화할 때에는 요소 형식, this 키워드, [매개 변수]를 선언하고 내부에 get 혹은 set 블록을 정의하면 됩니다.

```
class Example
{
    ...
    public string this[int index]
    {
        get
        {
            return ...;
        }
        set
        {
            ...[index] = value;
        }
    }
}
```

사용하는 곳에서는 인덱스 연산자를 이용하여 원하는 요소를 참조할 수 있습니다.

```
static void Main(string[] args)
{
    Example ex = new Example();
    ....
    ex[3] = "홍길동";
    Console.WriteLine("{0}",ex[3]);
}
```

간단한 코드를 통해 인덱서에 대해 살펴봅시다. 여기에서는 키와 값을 쌍으로 하는 문자열이 보관된 사전에서 키를 입력하면 원하는 값을 찾아주는 예를 들려고 합니다.

문자열을 요소로 갖는 MyDictionary 클래스를 정의합시다. 여기에서는 요소 문자열을 3개를 갖게 정의할게요.

```
class MyDictionary
{
    string[] storage = new string[3];
}
```

MyDictionary 클래스는 요소 문자열로 "key0=value0", "key1=value1", "key2=value2"을 보관할게요.

```
public MyDictionary()
{
    for (int i = 0; i < storage.Length; i++)
    {
        storage[i] = string.Format("key{0}=value{0}", i, i);
    }
}
```

MyDictionary는 보관된 요소를 정수를 인자로 받는 인덱서와 문자열을 인자로 받는 인덱서를 제공합니다.

```
public string this[int index]
{
    get;
}
public string this[string key]
{
    get;
}
```

정수를 인자로 받는 인덱서는 요소를 보관하는 storage의 유효한 범위에 있는 인자가 아니면 해당 요소의 값을 반환하게 정의합니다.

```
public string this[int index]
{
    get
    {
        if (AvailIndex(index)) //유효한 인덱스일 때
        {
            return GetValue(storage[index]); //storage[index] 요소의 값을 반환
        }
        return string.Empty;
    }
}
```

MyDictionary에 요소 문자열을 storage에 보관하므로 유효한 인덱스는 0보다 크거나 같으면서 storage의 Length보다 작아야 합니다.

```
private bool AvailIndex(int index)
{
    return (index >= 0) && (index < storage.Length);
}
```

요소 문자열은 "키=값"형태로 보관되어 있으므로 '='의 위치를 찾아 뒤쪽에 있는 부분 문자열이 값입니다.

```
private string GetValue(string s)
{
    int index = s.Index Of('='); //'='문자가 시작되는 위치를 찾는다.
    return s.Substring(index+1); //index+1 뒤에 있는 부분 문자열을 반환한다.
}
```

문자열을 인자로 받는 인덱서는 인자로 전달된 키에 해당하는 요소를 찾은 후에 요소 문자열에서 값을 추출하여 반환해야 합니다.

```
public string this[string key]
{
    get
    {
        string element = FindKey(key); //키에 해당하는 요소 문자열을 찾는다.
        return GetValue(element); //요소 문자열에서 값을 추출하여 반환한다.
    }
}
```

키를 인자로 받아 요소를 찾는 메서드에서는 요소 문자열이 보관된 storage의 각 요소중에 시작 부분이 인자로 받은 키인 요소를 반환하면 되겠죠.

```
private string FindKey(string key)
{
    foreach (string s in storage) //storage 컬렉션에 보관된 각 요소에 대해 반복 수행
    {
        if (s.IndexOf(key) == 0) //요소 문자열이 key로 시작할 때
        {
            return s; //요소 문자열 반환
        }
    }
    return string.Empty;
}
```

이처럼 인덱서를 제공하면 사용하는 곳에서는 인덱스를 통해 원하는 요소의 값을 얻어올 수 있게 됩니다.

```
MyDictionary md = new MyDictionary();
Console.WriteLine(md[0]); //정수를 인자로 받는 인덱서 사용
Console.WriteLine(md["key2"]); //문자열을 인자로 받는 인덱스 사용
```



▶ 인덱서 캡슐화 예제 코드

```
class MyDictionary
{
    string[] storage = new string[3];
    public MyDictionary()
    {
        for (int i = 0; i < storage.Length; i++)
        {
            storage[i] = string.Format("key{0}=value{0}", i, i);
        }
    }
    public string this[int index] //정수를 인자로 받는 인덱서
    {
        get
        {
            if (AvailIndex(index)) //유효한 인덱스일 때
            {
                return GetValue(storage[index]); //storage[index] 요소의 값을 반환
            }
            return string.Empty;
        }
    }
    public string this[string key] //문자열을 인자로 받는 인덱서
    {
        get
        {
            string element = FindKey(key); //키에 해당하는 요소 문자열을 찾는다.
            return GetValue(element); //요소 문자열에서 값을 추출하여 반환한다.
        }
    }
}
```

```

private bool AvailIndex(int index)
{
    return (index >= 0) && (index < storage.Length);
}

private string FindKey(string key)
{
    foreach (string s in storage) //보관된 각 요소에 대해 반복 수행
    {
        if (s.IndexOf(key) == 0){ return s; }
    }
    return string.Empty;
}

private string GetValue(string s)
{
    int index = s.IndexOf('='); //'='문자가 시작되는 위치를 찾는다.
    return s.Substring(index+1); //index+1 뒤에 있는 부분 문자열을 반환한다.
}

public int Size //요소를 보관하는 storage의 크기를 반환
{
    get{ return storage.Length; }
}
}

class Program
{
    static void Main(string[] args)
    {
        MyDictionary md = new MyDictionary();
        Console.WriteLine(md[2]); //정수를 인자로 받는 인덱서 사용
        Console.WriteLine(md["key1"]); //문자열 인자로 받는 인덱서 사용
    }
}

```

### 5.1.5 생성자

생성자는 정의한 클래스나 구조체의 개체가 생성될 때 수행할 코드를 작성하는 메서드입니다. 생성자는 반환 형식을 명시할 수 없고 형식 이름과 같은 이름을 갖는 특별한 메서드입니다. C#에서 정의할 수 있는 생성자 종류에는 기본 생성자, 입력 매개 변수가 있는 생성자, 정적 생성자가 있습니다.

기본 생성자는 입력 매개 변수가 없는 생성자를 말합니다. 기본 생성자는 클래스에서만 명시적으로 정의할 수 있으며 구조체는 매개 변수 있는 생성자만 정의할 수 있습니다. 또한, 클래스나 구조체 내에 어떠한 생성자도 정의하지 않으면 묵시적으로 기본 생성자가 만들어지며 멤버들을 기본값으로 초기화하는 등의 작업을 수행합니다.

#### ▶ 생성자를 정의하지 않을 때

```
class Example //생성자를 정의하지 않았음
{
    public int Val
    {
        get;
        set;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Example ex = new Example();
        Console.WriteLine("ex.Val:{0}",ex.Val);
    }
}
```

그리고 매개 변수가 있는 생성자를 정의하고 기본 생성자를 정의하지 않았을 때 컴파일러는 기본 생성자를 만들지 않으므로 생성할 때 인자를 전달하지 않으면 오류가 납니다.

정적 생성자는 개체의 멤버가 아니라 정적 멤버입니다. 정적 멤버는 static 키워드를 명시하여 선언한 멤버를 말하며 사용할 때 형식명으로 접근합니다.

정적 생성자는 정적 멤버가 있을 때 정적 멤버들의 값을 초기화하거나 단 한 번만 수행되어야 하는 작업을 정의할 때 사용되며 직접 호출할 수 없으며 접근 한정자를 명시할 수도 없습니다. 그리고 정적 생성자는 해당 형식을 사용하는 어떤 코드보다도 먼저 수행됨을 보장합니다. 개체의 멤버와 정적인 멤버에 대한 자세한 사항은 5.3에서 다룰 것입니다.

▶ 정적 생성자가 호출되는 시점

```
class Example
{
    static Example() //접근 한정자를 명시할 수 없음
    {
        Console.WriteLine("정적 생성자 수행 ");
    }
    public static void Foo()
    {
        Console.WriteLine("정적 메서드 Foo 수행");
    }
}
class Program
{
    static void Main(string[] args)
    {
        Example.Foo();
    }
}
```

▶ 실행 결과

정적 생성자 수행

정적 메서드 Foo 수행

생성자의 접근 지정이 private으로 되어 있는 것을 전용 생성자라 부릅니다. 대표적인 예로 개체를 하나만 생성해야 할 때를 들 수 있습니다. 참고로 이러한 패턴을 GoF의 디자인 패턴에서는 싱글톤(Singleton) 패턴이라고 부릅니다.

▶ 전용 생성자를 이용해 단일체 구현

```
class Singleton
{
    static Singleton singleton; //단일체 정적 멤버
    public static Singleton Instance //단일체를 참조할 때 사용하는 정적 속성
    {
        get
        {
            return singleton;
        }
    }
    static Singleton() //정적 생성자
    {
        singleton = new Singleton();//단일체 생성
    }
    private Singleton() //단일체로 구현하기 위해 접근 지정을 private 설정
    {
    }
}
```

### 5.1.6 소멸자

소멸자는 개체가 소멸할 때 수행해야 할 작업에 대한 코드 블록입니다. 소멸자는 클래스에서만 정의할 수가 있고 구조체에는 정의할 수 없습니다. C++에서는 동적으로 생성된 개체를 개발자가 소멸시켜야 합니다. 하지만 C#에서는 .NET 플랫폼의 가비지 수집기가 주기적으로 개체 수명을 조사할 통해 개체를 참조하는 변수가 없는 개체를 확인하고 정책에 의해 자동으로 소멸합니다.

이에 소멸자의 접근 한정자는 개발자가 명시할 수 없습니다. 그리고 다른 메서드들과 달리 중복 정의할 수도 없습니다. 소멸자의 이름은 형식 이름 앞에 ~가 붙습니다. 그리고 가비지 수집기가 소멸되는 시점에 자동으로 호출합니다.

```
class Man
{
    ~Man()
    {
        Console.WriteLine("소멸자");
    }
}
```

컴파일러는 소멸자를 정의하면 컴파일러에서는 Finalize 메서드를 호출하는 코드로 변환됩니다. ildasm 유틸리티를 이용하여 덤프해 보면 이를 확인하실 수 있습니다.

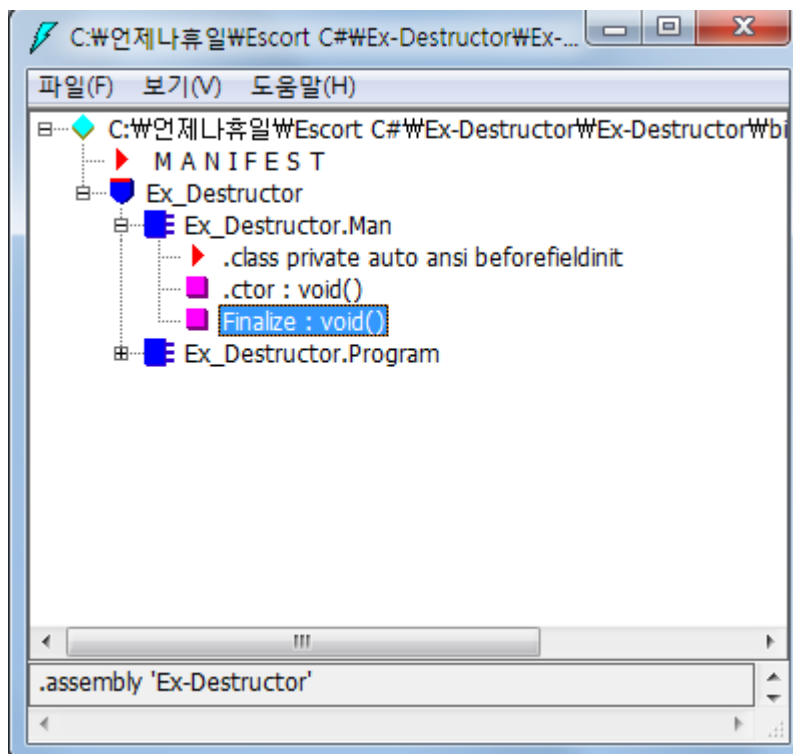
#### ▶ ildasm으로 덤프한 IL코드

```
.method family hidebysig virtual instance void
    Finalize() cil managed
{
    // 코드 크기      25 (0x19)
    .maxstack 1
    .try
    {
        IL_0000: nop
        IL_0001: ldstr      bytearray (4C C7 )           // L.
        IL_0006: call      void [mscorlib]System.Console::WriteLine(string)
        IL_000b: nop
        IL_000c: nop
    }
```

```

    IL_000d: leave.s    IL_0017
} // end .try
finally
{
    IL_000f: ldarg.0
    IL_0010: call      instance void [mscorlib]System.Object::Finalize()
    IL_0015: nop
    IL_0016: endfinally
} // end handler
IL_0017: nop
IL_0018: ret
} // end of method Man::Finalize

```



[그림 5.4] ildasm 유틸로 덤프한 화면

이처럼 개발자가 소멸자를 정의하면 컴파일러에 의해 묵시적으로 Finalize 메서드로 변환이 되고 가비지 수집기로 개체가 소멸하는 작업을 수행하게 됩니다. 만약, 소멸하는 시점에 특별히 할 작업이 없는데도 비어있는 소멸자를 정의하면 비어있는 Finalize 메서드를 수행하게 됩니다. 따라서 비어있는 소멸자는 정의하지 마시기 바랍니다.

또한, 외부 자원을 사용하는 경우에 명시적으로 자원 해제를 원한다면 Dispose 메서드를 정의하여 성능을 높일 수 있습니다.

▶ IDisposable 인터페이스 기반의 클래스 정의

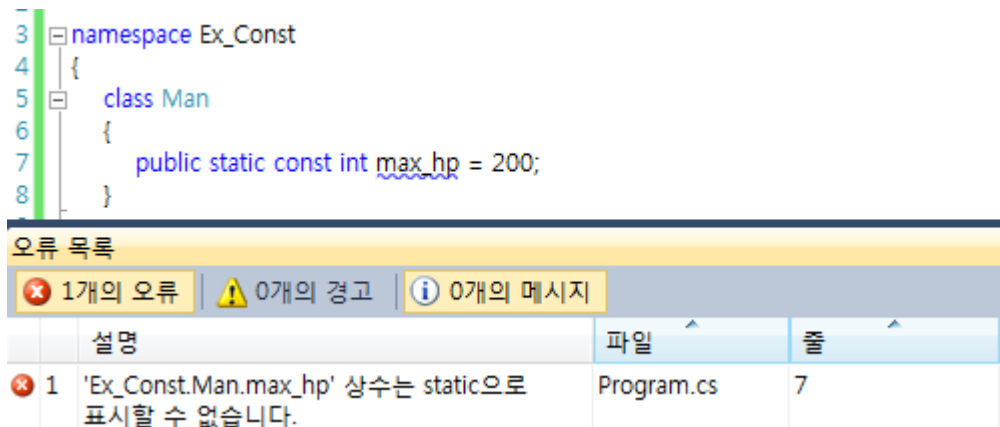
```
class Man:IDisposable
{
    bool isdispose;
    ~Man()
    {
        Dispose();
    }
    public void Dispose()
    {
        if ( ! isdispose )
        {
            //자원 해제 구문
        }
        isdispose = true;
    }
}
```



### 5.1.7 상수와 읽기 전용

C#에서는 상수에는 컴파일 시에 상수값이 결정되는 상수 멤버 필드와 런타임에 결정되는 읽기 전용을 제공하고 있습니다.

상수 멤버 필드는 const 키워드와 형식, 필드 이름과 초기값을 대입하면 됩니다. 이처럼 상수 멤버 필드를 캡슐화하면 이는 개체의 멤버가 아닌 묵시적으로 정적 멤버가 되므로 명시적으로 static 키워드를 사용할 수 없습니다.



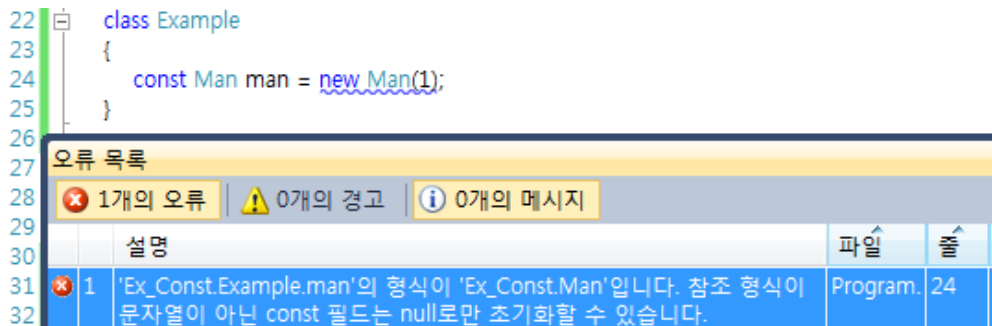
[그림 5.5] 상수 멤버 필드에 static 키워드를 명시할 경우 오류 화면

읽기 전용을 캡슐화할 때는 readonly 키워드와 형식, 이름을 선언하면 됩니다. 읽기 전용은 상수 멤버 필드와 다르게 묵시적으로 정적 멤버가 아니므로 static 키워드를 명시해야 정적 멤버가 됩니다. 그리고 읽기 전용은 생성자에서 초기화할 수 있습니다.

#### ▶ 읽기 전용 멤버를 생성자에서 초기화

```
class Man
{
    readonly int num;
    public Man(int num)
    {
        this.num = num;
    }
}
```

상수 멤버 필드는 컴파일 시에 상수값이 결정되므로 string 형식을 제외한 참조 형식을 사용하려면 상수값을 null로 초기화해야 합니다. 따라서 동적으로 생성한 개체를 상수 멤버로 두려면 읽기 전용을 사용해야 합니다.



[그림 5.6] 참조 형식의 동적인 개체를 const 멤버로 할 경우 오류 화면

### 5.1.8 연산자 중복 정의

C#에서는 클래스와 구조체의 멤버로 연산자 중복 정의를 캡슐화할 수 있습니다. 연산자 중복 정의를 하는 방법은 메서드와 비슷한데 static 키워드를 붙여 정적 멤버로 만들어야 하며 public으로 접근을 한정시키고 메서드 이름 대신 operator 키워드와 연산 기호를 명시한다는 점입니다.

그렇다고 모든 연산자를 중복 정의가 가능한 것은 아닙니다. 다음은 중복 정의가 가능한 것들입니다.

단항 연산자: +, -, !, ~, ++, --, true, false

이항 연산자: +, -, \*, /, %, &, |, ^, <, >, ==, !=, <, >, <=, >=

▶ 연산자 중복 정의

```
class Man
{
    string name;
    public Man(string name)
    {
        this.name = name;
    }

    //== 연산자 중복 정의 (!= 연산자도 같이 중복 정의해야 함)
    public static bool operator ==(Man man, string name)
    {
        return man.name == name;
    }

    //!= 연산자 중복 정의 (== 연산자도 같이 중복 정의해야 함)
    public static bool operator !=(Man man, string name)
    {
        return !(man == name);
    }
}
```

그리고 C#에서 비교 연산자를 중복 정의를 하려면 대응되는 연산자도 중복 정의해야 합니다. 즉, ==연산자를 중복 정의하려면 != 연산자도 중복 정의해야 한다는 것입니다. 이외에 묵시적 형 변환과 명시적 형 변환을 하는 제공합니다. 묵시적 형 변환은 implicit 키워드를 operator 앞에 명시하고 형 변환하고자 하는 형식을 operator 뒤에 명시하면 됩니다. 명시적 형 변환은 explicit 키워드를 사용하면 됩니다.

```
public static implicit operator string(Man man)
{
    return man.name;
}
```

## 5.2 개체의 멤버와 정적 멤버

클래스나 구조체에 캡슐화할 수 있는 멤버들은 어떠한 것들이 있는지 살펴보았는데 중간마다 개체의 멤버와 정적인 멤버라는 얘기가 나왔던 것을 기억하시죠. 이제 이들에 대해 좀 더 자세히 살펴봅시다.

클래스나 구조체를 정의하였다는 것은 형식을 정의한 것을 의미하며 해당 형식에 맞는 실질적인 대상을 개체라고 합니다. 개체의 멤버는 개체의 데이터나 이를 사용하기 위한 메서드 등을 얘기합니다. 이에 반해 정적인 멤버는 개체에 상관없이 해당 형식에 공통으로 사용되는 멤버입니다.

클래스나 구조체에 캡슐화된 멤버들 중에 static 키워드나 const 키워드가 붙어 있는 멤버들은 정적인 멤버이며 이 외에 다른 멤버들은 개체의 멤버입니다. public으로 접근이 지정된 멤버들 중에 정적인 멤버는 클래스의 이름으로 접근하고 개체의 멤버는 변수로 접근합니다. 그리고 정적인 멤버는 개체의 멤버를 사용할 수 없습니다.

클래스는 class 키워드 앞에 static 키워드를 명시하면 정적 멤버만 캡슐화 가능한 정적 클래스가 만들어집니다. 프로그램 전체에서 사용하는 상수와 자주 사용되는 기능을 정적 클래스에 캡슐화하여 사용할 수 있을 것입니다. 이 외에도 네이티브 코드로 작성된 라이브러리를 마이그레이션할 때도 자주 사용됩니다.

### ▶ 정적 클래스

```
static class EHGlobal
{
    public const int max_man = 100;
    public static int GetNum()
    {
        int num = 0;
        try
        {
            num = int.Parse(Console.ReadLine());
        }
        catch{ }
        return num;
    }
}
```

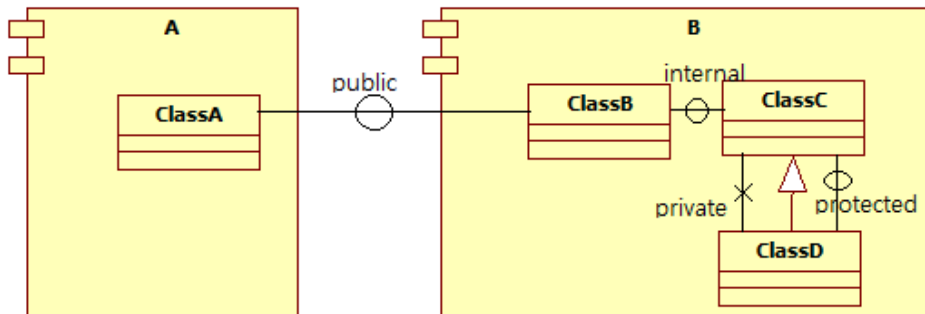
### 5.3 접근 한정자

C#에서는 정의하는 형식을 사용하는 범위나 형식 내의 멤버에 대해 사용할 수 있는 범위를 접근 한정자를 통해 지정할 수 있습니다.

형식 정의문 앞에 올 수 있는 접근 한정자에는 `public`과 `internal`이 있는데 명시하지 않으면 `internal`로 지정됩니다. `public`으로 접근 지정하면 다른 .NET 어셈블리에서도 접근할 수 있지만 `internal`로 지정하면 같은 .NET 어셈블리에서만 접근할 수 있습니다. 예를 들어 라이브러리를 만들 때 라이브러리 내에서만 접근할 멤버들은 `internal`로 외부에서 사용해도 되는 멤버는 `public`으로 지정합니다.

형식 내의 멤버에 대한 접근 한정자는 `public`, `protected`, `private`, `internal`, `protected internal`이 있습니다. 멤버에 접근 한정자가 명시되어 있지 않으면 `private`으로 지정됩니다. `private`은 해당 형식 내부에서만 접근할 수 있게 함으로써 잘못된 사용을 차단합니다. 이에 반해 `public`은 모든 곳에서 접근할 수 있으므로 외부 .NET 어셈블리에서도 접근해도 되는 멤버에만 지정합니다. 그리고 같은 .NET 어셈블리 내에 있지만, 형식 외부에서 접근해도 되는 멤버는 `internal`로 지정합니다. `protected`는 파생 클래스 형식에서는 접근해도 되지만 다른 형식에서는 사용하면 안 되는 멤버를 지정합니다. 그리고 같은 .NET 어셈블리와 파생된 형식에서 접근할 수 있게 하려면 `protected internal`로 지정합니다.

여기에서는 접근 한정을 `private`와 `internal`로 했을 때의 차이를 통해 정보 은닉하여 데이터의 신뢰성을 높이는 수단으로 사용하는 것을 다룰 것입니다. `protected` 접근 한정자는 6장 상속과 다형성에서 다룰 것이며 `public`에 대한 부분은 9장 .NET 어셈블리에서 다루겠습니다. 참고로, C++에서 제공되는 접근 한정자 `public`은 C#에서 제공하는 `internal`과 같은 수준의 접근 한정자이며 `protected`와 `private`은 동일하다고 볼 수 있습니다.



[그림 5.7] 접근 한정자의 접근 가능 여부

다음은 학생 클래스에 각 멤버의 접근 한정을 필요한 수준으로 변경하여 신뢰성을 높인 예입니다.

▶ 필요한 수준으로 접근 지정하여 신뢰성을 높인 예

```
class Student
{
    const int max_iq = 200;
    int iq = 100; //접근 수준: private

    internal int Iq
    {
        get //접근 수준: internal
        {
            return iq;
        }

        private set //접근 수준 private
        {
            if ((iq + value) > max_iq)
            {
                iq = max_iq;
            }
            else
            {
                iq += value;
            }
        }
    }

    internal void Study(int scnt) //접근 수준: internal
    {
        Iq += scnt;
    }
}
```

```

class Program
{
    static void Main(string[] args)
    {
        Student stu = new Student();
        Console.WriteLine("공부할 시간을 입력하세요.");
        int scnt = int.Parse(Console.ReadLine());
        stu.Study(scnt); //접근수준이 private인 멤버 필드 iq를 직접 사용 못함

        //접근 수준이 internal인 멤버 속성 Iq를 통해 개체의 iq를 얻어옴
        Console.WriteLine("아이큐:{0}", stu.Iq);
    }
}

```

▶ 실행 결과

공부할 시간을 입력하세요.

120

아이큐:200

먼저, iq에 대한 형식 외부에서 직접적인 사용을 막기 위해 private으로 지정하였습니다(접근 한정자를 명시하지 않았을 때는 private으로 지정됨). 물론, 학생 개체를 원하는 시간 만큼 공부시킬 수 있게 Study 메서드는 여전히 internal입니다. 그리고 사용하는 곳에서 학생 개체의 iq를 확인할 수 있게 멤버 속성 Iq를 두었고 get 블록과 set 블록의 접근 한정을 비대칭으로 지정하였습니다. get은 속성 Iq에 지정한 internal이므로 학생 클래스 외부에서 접근할 수 있지만, set은 private으로 명시하였기 때문에 외부에서 접근하지 못하게 됩니다. 이처럼 접근 한정자를 통해 필요한 수준으로 정보 은닉하여 신뢰성을 높일 수 있습니다. 이를 통해 개발 단계에서 버그를 줄일 수 있을 것입니다.

개발 시에 적절하게 접근 한정자를 지정하는 것은 많은 노하우가 필요합니다. 초기 개발자는 될 수 있으면 접근 한정자는 디폴트를 사용하세요. 그리고 반드시 접근이 필요한 곳 이 생기면 그 위치에서 접근할 수 있을 정도로 접근 수준을 변경하세요. 이 같은 습관을 갖는다면 보다 신뢰성 있는 프로그램을 제작하실 수 있을 것입니다. 그리고 멤버 필드는 언제나 접근 한정자를 디폴트를 사용하시고 필요한 경우에는 해당 멤버 필드에 대해 접근 가능한 멤버 속성이나 멤버 메서드를 제공하십시오.

## 6. 상속과 다형성

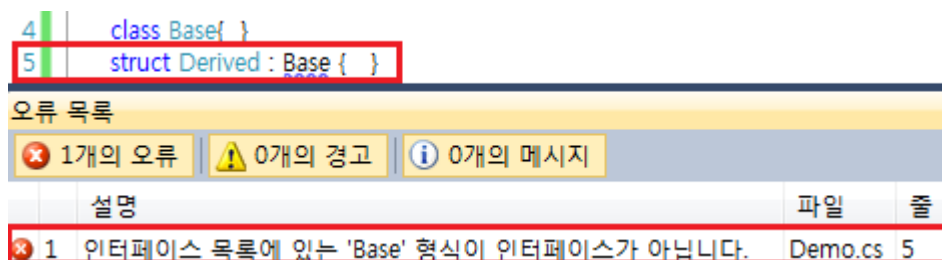
C#에서는 다른 형식을 기반으로 파생된 형식을 정의할 수 있습니다. 이러한 OOP 언어의 특징을 상속이라고 합니다. C#에서 상속은 클래스와 인터페이스가 기반 형식이 될 수 있고 구조체는 기반 형식으로 인터페이스만 가능합니다.

이처럼 상속으로 표현된 기반 클래스와 파생 클래스의 관계를 일반화 관계라고 합니다. C#에서는 기반 형식의 변수로 파생된 형식의 개체를 참조할 수 있어 사용하는 곳에 편의를 제공합니다. 그리고 기반 형식에 정의한 멤버 메서드를 파생 형식에서 재정의할 수 있습니다. 이처럼 기반 형식의 멤버 메서드를 재정의하여 사용하는 변수의 형식의 멤버 메서드가 아닌 실제 참조하고 있는 개체의 멤버 메서드가 수행되게 할 수 있습니다. 이 같은 특징을 다형성이라고 합니다. 이번 장에서는 C#에서 제공하는 상속과 다형성에 대해 살펴보시다.

### 6.1 상속

C#에서 파생된 형식을 정의할 때 기반 형식을 명시하여 상속을 표현합니다. 구조체는 기반 형식으로 인터페이스만 사용할 수 있지만, 클래스는 기반 형식으로 클래스와 인터페이스 모두 사용 가능합니다. 기반 형식에서 파생 형식을 정의할 때는 파생 형식 명 뒤에 콜론을 추가하고 기반 형식의 이름을 지정합니다.

```
class Base
{
}
class Derived: Base
{
}
```



[그림 6.1] 클래스 기반의 구조체를 정의할 때 오류 화면



파생 형식의 개체는 생성 과정에서 기반 형식의 개체 부분을 포함하여 생성됩니다.

다음의 예는 Walk 메서드가 있는 Man 클래스를 기반으로 파생된 Student 클래스를 정의한 예입니다. 예를 보시면 파생된 Student 개체를 참조하는 변수를 통해 기반 형식인 Man의 멤버 메서드인 Walk를 사용하는 것을 보실 수 있습니다.

▶ Man 클래스 기반의 파생 클래스 Student 정의

```
class Man
{
    internal void Walk()
    {
        Console.WriteLine("걷다.");
    }
}
class Student:Man //Man 클래스를 기반으로 파생
{
    internal void Study()
    {
        Console.WriteLine("공부하다.");
    }
}
class Program
{
    static void Main(string[] args)
    {
        Student student = new Student();
        student.Walk(); //기반 클래스 Man의 멤버를 사용
        student.Study();
    }
}
```

▶ 실행 결과

걷다.

공부하다.

이처럼 파생된 형식의 개체에서는 기반 형식의 멤버를 포함하게 되는데 프로그램에서 파생된 형식을 정의할 때는 "파생된 형식은 기반 형식이다."라는 논리가 성립될 때 사용하십시오.

### 6.1.1 protected

파생된 형식에서도 기반 형식에서 private으로 접근 지정된 멤버는 보이지 않으므로 자신에게 포함된 멤버지만 접근할 수 없습니다.

The screenshot shows a C# code editor with the following code:

```

5 class Man
6 {
7     int hp = 100;
8     internal void Walk()
9     {
10        Console.WriteLine("걸다.");
11        hp += 2;
12    }
13 }
14 class Student:Man
15 {
16     internal void Study()
17     {
18        Console.WriteLine("공부하다.");
19        hp -= 2;
20    }
21 }

```

Below the code, the 'Error List' (오류 목록) window is open, showing one error:

설명	파일	줄	열
1 보호 수준 때문에 'Ex_Inheritance.Man.hp'에 액세스할 수 없습니다.	Program.	19	13

[그림 6.2] 기반 형식의 private으로 접근 지정된 멤버에 파생된 형식에서 접근 불가

형식 외부에서는 접근을 막고 파생된 형식에서는 접근할 수 있게 하려면 protected로 접근 지정하면 됩니다. 하지만 멤버 필드는 접근 지정을 private으로 하시고 파생된 형식에서 접근할 수 있게 하려면 멤버 속성을 통해 접근할 수 있게 하는 것이 바람직합니다.

▶ 멤버 속성을 통해 멤버 필드에 접근

```
class Man
{
    int hp = 100; //private으로 접근 지정(디폴트 접근 지정)

    protected int Hp //파생 형식에서 접근 가능
    {
        get
        {
            return hp;
        }
        set
        {
            hp = value;
        }
    }
    internal void Walk()
    {
        Console.WriteLine("걷다.");
        Hp += 2;
    }
}
```

### 6.1.2 생성

파생 형식의 개체가 생성될 때는 기반 형식의 생성자 수행 후에 파생 형식의 생성자가 수행됩니다.

#### ▶ 파생 형식의 개체 생성 과정

```
class Man
{
    internal Man()
    {
        Console.WriteLine("Man 생성자");
    }
}
class Student : Man //Man 클래스 기반으로 파생
{
    internal Student()
    {
        Console.WriteLine("Student 생성자");
    }
}
class Program
{
    static void Main(string[] args)
    {
        Student student = new Student();
    }
}
```

#### ▶ 실행 결과

Man 생성자

Student 생성자

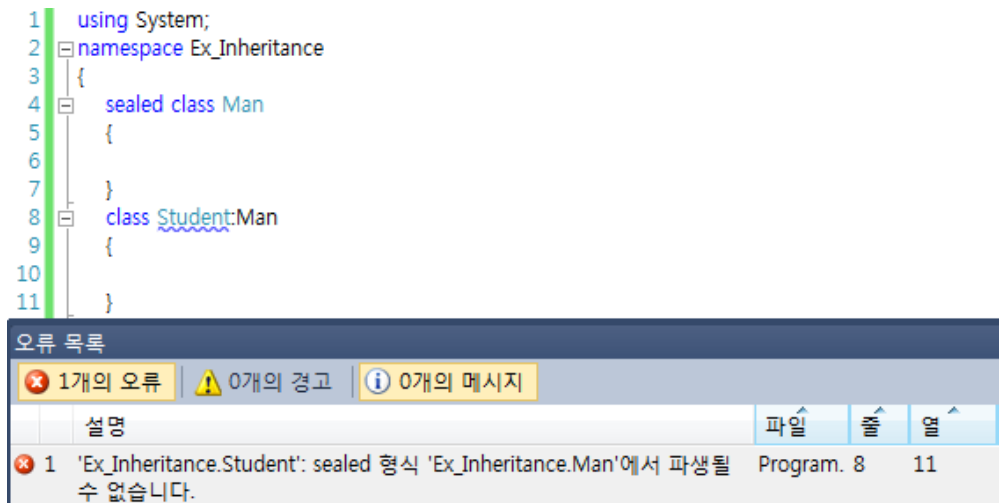
만약, 기반 형식의 기본 생성자가 없고 매개 변수가 있는 생성자만 있으면 어떻게 해야 할까요? 이때는 다음의 예처럼 파생 형식의 생성자를 캡슐화할 때 base 키워드와 기반 형식 생성에 필요한 인자를 이용하여 초기화해야 합니다.

▶ 기반 형식에 기본 생성자가 없을 때 파생 형식 생성자에서 초기화

```
class Man
{
    string name;
    internal Man(string name)
    {
        this.name = name;
    }
}
class Student:Man
{
    internal Student(string name):base(name) //기반 클래스 생성 초기화
    {
    }
}
class Program
{
    static void Main(string[] args)
    {
        Student student = new Student("홍길동");
    }
}
```

### 6.1.3 봉인(sealed) 클래스

C#에서는 클래스를 정의할 때 다른 클래스의 기반 형식으로 사용할 수 없게 정의할 수 있습니다. 이와 같은 클래스를 봉인 클래스라 부르며 class 앞에 sealed 키워드를 명시하면 됩니다. 값 형식은 기반 형식이 될 수 없으므로 묵시적으로 봉인된 형식입니다.



[그림 6.3] 봉인 클래스를 기반 형식으로 사용하려고 할 때 오류 화면

좀 더 자세한 사항은 6.2 다형성에서 다시 다루기로 하겠습니다. 그리고 인터페이스 기반으로 파생하는 부분은 7장 인터페이스와 컬렉션에서 설명할게요.

## 6.2 다형성

OOP 언어는 캡슐화, 상속과 더불어 중요한 특징으로 다형성이 있습니다. C#에서의 다형성은 크게 두 가지로 얘기합니다. 첫째로 변수는 여러 형식의 개체를 참조할 수 있다는 것입니다. C#에서는 기반 형식의 변수로 파생된 개체를 참조할 수 있습니다. 두 번째로 변수를 통해 메서드를 호출했을 때 구체적인 동작이 다를 수 있다는 것입니다.

C#에서는 기반 형식의 멤버 메서드를 추상 메서드와 가상 메서드로 지정할 수 있는데 파생된 형식에서 재정의하면 기반 형식의 변수로 파생된 개체를 참조했을 때 변수의 형식에 정의된 멤버가 아닌 참조된 실제 개체의 멤버가 수행이 됩니다.

### 6.2.1 기반 형식의 변수로 파생된 개체를 참조

C#에서는 다형성을 제공하여 기반 형식의 변수로 파생된 개체를 참조할 수 있습니다. 이러한 특징은 하나의 기반 형식에서 파생된 다양한 형식 개체를 사용할 때 같은 방식으로 사용할 수 있는 편의성을 제공합니다. 특히, C#에서는 모든 형식이 묵시적으로 object에서 파생되므로 편의성은 극대화됩니다.

#### ▶ 기반 형식 변수로 파생 개체 참조

```
class Man
{
}

class Stu : Man //기반 형식 Man에서 파생
{
}

class Program
{
    static void Main()
    {
        Man man = null;
        man = new Stu(); //기반 형식 변수 Man으로 파생 형식 Stu 개체 참조
    }
}
```

## 6.2.2 is 연산자와 as 연산자를 이용한 하향 캐스팅

이처럼 기반 형식의 변수로 파생된 개체를 참조할 수 있다는 특징은 사용의 편의성이 제곱합니다. 하지만 프로그램에서 파생 형식에만 캡슐화된 멤버에 접근하여 사용해야 할 때도 있습니다. 이 때 is 연산자와 as 연산자를 사용하면 파생 개체를 참조할 수 있습니다.

### ▶ 용어: 상향 캐스팅, 하향 캐스팅

기반 형식의 변수로 파생된 형식 개체를 참조하는 것을 상향 캐스팅이라 하고 기반 형식의 변수에 참조하는 개체를 파생 형식 변수로 참조하는 것을 하향 캐스팅이라고 합니다.

C#에서는 기반 형식의 변수로 파생된 형식 개체를 참조하는 상향 캐스팅은 묵시적으로 사용할 수 있지만, 기반 형식의 변수가 참조하는 파생 개체를 파생된 형식 변수로 하향 캐스팅하는 것은 묵시적으로 사용할 수 없고 as 연산자나 is 연산자를 사용해야 합니다.

as 연산자는 이항 연산자로 좌항에 개체를 참조하는 변수가 오고 우항에 참조하려는 형식을 명시하여 사용합니다. 만약, 해당 변수가 참조하는 개체가 우항에 오는 형식이 다르다면 해당 형식의 개체 참조를 반환하고 그렇지 않으면 null을 반환합니다. 이처럼 as 연산은 참조 형식에 사용할 수 있습니다.

### ▶ as 연산으로 파생된 형식 개체 참조

```
Stu stu = man as Stu; //파생된 형식 개체 참조
if (stu != null) //man이 Stu개체일 때
{
    stu.Study();
}
else
{
    Console.WriteLine("Stu 형식 개체가 아닙니다. ");
}
```



is 연산자는 이항 연산자로 좌항에 변수가 오고 우항에 형식 명을 명시하면 해당 변수가 해당 형식으로 호환할 수 있는지를 반환합니다. is 연산은 as 연산과 달리 값 형식에서도 사용할 수 있으며 형 변환하기 전에 호환 여부를 확인하기 위해 사용됩니다.

▶ as 연산으로 파생된 형식 개체 참조

```
Man man = new Stu();
if (man is Stu) //man이 Stu개체일 때
{
    Stu stu = (Stu)man;
    stu.Study();
}
object obj = 3;
if (obj is int) //obj가 int 형식일 때
{
    int i = (int)obj;
    Console.WriteLine(i.ToString());
}
```

특히 컬렉션을 통해 요소 개체가 특정 형식일 때 수행할 때 하향 캐스팅을 사용합니다.

▶ as 연산자를 이용하여 파생 개체 참조

```
class Man
{
    internal string Name{ get; private set; }
    internal Man(string name)
    {
        Name = name;
    }
    internal void Eat()
    {
        Console.WriteLine("{0} 밥을 먹다.",Name);
    }
}
```

```

class Stu : Man //기반 형식 Man에서 파생
{
    internal Stu(string name): base(name){ }
    internal void Study()
    {
        Console.WriteLine("{0} 공부하다.",Name);
    }
}
class Program
{
    static void Main()
    {
        Man[] people = new Man[2];
        people[0] = new Stu("홍길동"); //기반 형식으로 파생 개체 참조
        people[1] = new Man("강감찬");
        Stu stu = null;
        foreach (Man man in people)
        {
            man.Eat();
            stu = man as Stu; //as 연산으로 파생 개체 참조
            if (stu != null) //참조한 개체가 Stu 형식 개체일 때
            {
                stu.Study();
            }
        }
    }
}

```

▶ 실행 결과

홍길동 밥을 먹다.

홍길동 공부하다.

강감찬 밥을 먹다.

### 6.2.3 new 키워드를 이용한 무효화, base 키워드를 이용한 무효화 된 멤버 사용하기

C#에서 파생된 형식에서 기반 형식에 정의된 멤버와 같은 이름의 멤버를 new 키워드를 사용하여 캡슐화하면 기반 형식의 멤버는 무효화됩니다. 이 때는 사용하는 곳의 형식에 따른 멤버가 사용됩니다. 즉, 기반 형식의 변수로 접근하면 기반 형식의 멤버가 사용되고 파생 형식의 변수로 접근하면 파생 형식의 멤버가 사용됩니다.

#### ▶ new 키워드를 이용한 무효화

```
class Man
{
    internal void Work()
    {
        Console.WriteLine("일을 하다.");
    }
}

class Student : Man
{
    internal new void Work() //new 키워드를 이용하여 Man 형식의 Work메서드 무효화
    {
        Console.WriteLine("공부하다.");
    }
}

class Program
{
    static void Main()
    {
        Student student = new Student();
        student.Work();
        Man man = student;
        man.Work();
    }
}
```

▶ 실행 결과

공부하다.

일을 하다.

그리고 C#에서는 파생 형식에서 무효화 된 기반 클래스의 멤버를 사용할 수 있게 base 키워드를 제공하고 있습니다. 만약, 파생 형식에서 base 키워드를 통해 멤버를 호출하면 무효화 된 기반 클래스의 멤버가 사용됩니다.

▶ base 키워드로 무효화 된 멤버 사용

```
class Man
{
    internal void Work()
    {
        Console.WriteLine("일을 하다.");
    }
}
class Student : Man
{
    internal new void Work() //new 키워드로 기반 형식 Man의 Work 메서드 무효화
    {
        base.Work(); //base 키워드로 무효화 된 Work 메서드 사용
        Console.WriteLine("공부하다.");
    }
}
class Program
{
    static void Main()
    {
        Student student = new Student();
        student.Work();
    }
}
```

▶ 실행 결과

일을 하다.

공부하다.

#### 6.2.4 virtual 키워드를 이용한 가상화 및 override를 이용한 재정의

new 키워드로 기반 클래스의 멤버를 무효화 할 때는 사용하는 변수 형식의 멤버가 사용 됩니다. 이러한 특징은 기반 형식의 변수로 다양한 파생 개체를 참조할 때 실제 개체에 정의된 멤버가 사용되지 않으므로 다형성의 장점을 충분히 사용하지 못합니다.

C#에서는 기반 형식에서 정의한 멤버를 파생 형식에서 재정의하면 변수의 형식이 아닌 개체의 형식의 멤버가 동작하게 virtual 키워드와 override 키워드를 제공합니다. 기반 형식에서 virtual 키워드를 명시하여 멤버를 선언하면 가상 멤버가 됩니다. 이때 파생 형식에서 기반 형식의 가상 멤버를 재정의할 때는 override 키워드를 명시합니다.

▶ virtual로 가상 메서드 선언, override로 재정의

```
class Man
{
    internal virtual void Work() //virtual 키워드로 가상 메서드 선언
    {
        Console.WriteLine("일을 하다.");
    }
}

class Student : Man
{
    internal override void Work() //override로 기반 형식의 가상 메서드 재정의
    {
        Console.WriteLine("공부하다.");
    }
}

class Program
{
    static void Main()
    {
        Man man = new Student();
        man.Work();
    }
}
```

▶ 실행 결과

공부하다.

### 6.2.5 abstract 키워드를 이용한 추상화

C#에서는 상속을 통해서만 사용할 수 있는 기반 클래스를 만들 수 있습니다. 이러한 클래스를 만들기 위해서는 abstract 키워드를 이용하여 클래스를 정의하면 되는데 이를 추상 클래스라 합니다. C#에서 추상 클래스는 개체를 생성할 수 없으며 단지 상속을 통해 기반 클래스 역할만 할 수 있습니다.

```
9  abstract class Man
10 {
11     internal Man()
12     {
13     }
14 }
15
16 class Program
17 {
18     static void Main(string[] args)
19     {
20         Man man = new Man();
21     }
22 }
```

오류 목록

1개의 오류   0개의 경고   0개의 메시지				
	설명	파일	줄	열
1	'Ex_Polymorphism.Man' 추상 클래스 또는 인터페이스의 인스턴스를 만들 수 없습니다.	Program.	20	23

[그림 6.4] 추상 클래스 형식의 개체를 생성하려고 할 때 오류 화면

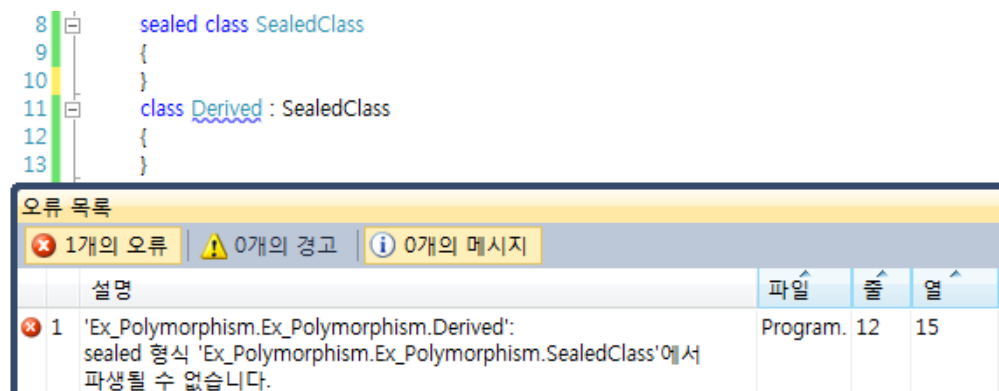
추상 클래스에는 추상 메서드를 캡슐화할 수 있는데 `abstract` 키워드를 명시하고 메서드의 수행 코드는 정의하지 않습니다. 이처럼 추상 메서드를 캡슐화하면 파생 형식에서는 추상 멤버를 재정의해야 개체를 생성할 수 있게 됩니다.

▶ `abstract` 키워드를 이용한 추상 클래스 정의 및 추상 메소드 선언

```
abstract class Man //추상 클래스
{
    internal abstract void Work(); //추상 메서드
}
class Student : Man
{
    internal override void Work()//기반 형식 Man의 추상 메서드 Work 재정의
    {
        Console.WriteLine("공부하다.");
    }
}
```

#### 6.2.5 sealed 키워드를 이용한 봉인

C#에서는 특정 형식을 기반 형식으로 사용하지 못하게 봉인시킬 수 있습니다. 봉인 형식을 정의할 때는 형식 명 앞에 `sealed` 키워드를 명시합니다. 그리고 파생 형식에서 재정의한 메서드에 `sealed` 키워드를 명시하여 이후에 파생 형식에서 재정의하지 못하게 봉인 메서드를 만들 수 있습니다.



```
8 sealed class SealedClass
9 {
10 }
11 class Derived : SealedClass
12 {
13 }
```

**오류 목록**

1개의 오류 0개의 경고 0개의 메시지

설명	파일	줄	열
1 'Ex_Polymorphism.Ex_Polymorphism.Derived': sealed 형식 'Ex_Polymorphism.Ex_Polymorphism.SealedClass'에서 파생될 수 없습니다.	Program.	12	15

[그림 6.5] 봉인 클래스를 기반 클래스로 사용할 때 오류 화면

## 7. 인터페이스와 컬렉션

C#에서는 인터페이스를 통해 기능 구현에 대한 약속을 추상화할 수 있습니다. 인터페이스는 명시적으로 추상 형식이며 클래스나 구조체에서 이를 구현 약속(상속)하면 약속된 기능들을 구현해야 합니다.

이러한 인터페이스를 이용하면 같은 인터페이스 기반의 여러 개체를 인터페이스 형식 변수로 사용하는 다형성의 장점을 누릴 수 있습니다. 또한, 이를 프로그래밍에 사용하면 같은 인터페이스를 구현 약속된 개체의 사용 방법을 별도로 익힐 필요가 없게 됩니다.

그리고 C#에서는 데이터나 개체를 보관할 수 있는 여러 종류의 컬렉션을 제공하고 있으며 필수적인 인터페이스 기반으로 정의되어 있어서 효과적으로 프로그래밍할 수 있습니다.

### 7.1 인터페이스

인터페이스는 기능 구현에 대한 약속입니다. 인터페이스의 멤버로는 메서드와 속성, 이벤트, 인덱서를 캡슐화할 수 있으며 인터페이스 내부에서는 구체적인 구현을 하지 않습니다. 대신 인터페이스를 구현 약속하는 클래스나 구조체에서는 해당 인터페이스에 약속된 모든 멤버를 구체적으로 구현을 제공해야 합니다.

이와 같은 인터페이스를 사용하면 비슷한 기능이 필요한 여러 형식의 개체를 같은 방법으로 사용할 수 있는 편의성을 제공할 수 있습니다. 또한, 특정 인터페이스를 기반의 개체를 인터페이스 형식 변수로 인자를 전달받아 처리함으로써 인터페이스에 약속된 기능만 접근할 수 있게 할 수 있어 신뢰성도 높아집니다.

이 외에도 C#에서는 여러 인터페이스를 구현 약속이 가능하여 효과적으로 프로그래밍할 수 있습니다.



### 7.1.1 인터페이스 정의 및 구현 약속

인터페이스를 정의할 때는 멤버를 구체적인 구현하지 않고 캡슐화해야 합니다. 그리고 각 멤버는 묵시적으로 접근 한정자 public이며 접근 한정자를 지정할 수 없습니다.

#### ▶ 인터페이스 정의

```
interface IStudy
{
    void Study(); //접근 한정자를 지정할 수 없음
}
```

그리고 정의된 인터페이스를 구현 약속하는 경우에는 파생과 같은 방법으로 구현 약속할 인터페이스를 지정할 수 있습니다. 그리고, 구현 약속한 인터페이스에 멤버들은 구체적으로 구현해야 합니다. 인터페이스 기반의 형식에서 약속한 멤버를 구현하는 방법에는 묵시적 인터페이스 구현과 명시적 인터페이스 구현이 있습니다. 묵시적 인터페이스 구현에서는 약속된 멤버와 같은 시그니처를 갖게 선언하고 public으로 접근 지정해야 합니다.

#### ▶ 묵시적 인터페이스 구현

```
class Student : IStudy
{
    public void Study() //public으로 접근 지정해야 함
    {
    }
}
```

명시적 인터페이스 구현에서는 인터페이스 이름을 선언부에 명시하며 접근 한정자를 명시하지 않습니다.

#### ▶ 명시적 인터페이스 구현

```
class Student : IStudy
{
    IStudy.Study(){ }
}
```

### 7.1.2 다중 인터페이스 구현 약속

C#에서는 기반 클래스는 하나밖에 지정하지 못하지만 기반 인터페이스는 여러 개를 지정할 수 있습니다.

#### ▶ 명시적 인터페이스 구현

```
interface IStudy
{
    void Study();
}
interface ISleep
{
    void Sleep();
}
class Student : IStudy, ISleep
{
    public void Study()
    {
        Console.WriteLine("학생이 공부하다.");
    }
    public void Sleep()
    {
        Console.WriteLine("학생이 잠을 자다.");
    }
}
```

만약, 기반 클래스에서 파생도 받아야 하고 인터페이스 구현 약속도 해야 한다면 기반 클래스 이름을 맨 앞에 명시하여야 합니다.

▶ 명시적 인터페이스 구현

```
class Man
{
    public void Think()
    {
        Console.WriteLine("생각하다.");
    }
}
interface IStudy
{
    void Study();
}
class Student : Man, IStudy
{
    void IStudy.Study()
    {
        Console.WriteLine("학생이 공부하다. ");
    }
}
```

### 7.1.3 명시적 인터페이스 구현

같은 이름의 멤버가 캡슐화되어 있는 서로 다른 인터페이스를 기반으로 형식을 정의할 때는 명시적 인터페이스 구현으로 이름 충돌을 피해야 합니다.

#### ▶ 명시적 인터페이스 구현으로 이름 충돌

```
interface IStudy
{
    void Study();
    void Work(); //ITeach 인터페이스에도 같은 이름의 멤버가 있음
}
interface ITeach
{
    void Teach();
    void Work(); //IStudy 인터페이스에도 같은 이름의 멤버가 있음
}
class PartTimeTeacher : IStudy, ITeach
{
    void IStudy.Study()
    {
        Console.WriteLine("공부하다.");
    }
    void IStudy.Work() //이름 충돌을 막기 위해 명시적 인터페이스 구현
    {
        Console.WriteLine("강의를 듣다.");
    }
    void ITeach.Teach()
    {
        Console.WriteLine("강의하다.");
    }
    void ITeach.Work() //이름 충돌을 막기 위해 명시적 인터페이스 구현
    {
        Console.WriteLine("연구하다.");
    }
}
```

이처럼 명시적으로 인터페이스 구현하면 사용하는 곳에서는 인터페이스 형식 변수로 개체를 참조해야 약속된 멤버를 사용할 수 있습니다.

▶ 명시적 인터페이스 구현으로 이름 충돌

```
class Program
{
    static void Main(string[] args)
    {
        PartTimeTeacher pt_teacher = new PartTimeTeacher();
        IStudy istudy = pt_teacher as IStudy; //IStudy 인터페이스 참조
        if (istudy != null)
        {
            istudy.Study();
        }
    }
}
```

[그림 7.1]은 명시적 인터페이스 구현한 개체 멤버를 직접 접근할 때 오류 화면입니다.

```
71 class PartTimeTeacher : IStudy, ITeach
72 {
73     void IStudy.Study()
74     {
75         Console.WriteLine("공부하다.");
76     }
77     void IStudy.Work()
81     void ITeach.Teach()
85     void ITeach.Work()
89 }
90 class Program
91 {
92     static void Main(string[] args)
93     {
94         PartTimeTeacher pt_teacher = new PartTimeTeacher();
95         pt_teacher.Study();
96     }
97 }
98 }
99 }
```

오류 목록	파일	줄	프로젝트
1 'Ex_Interface.PartTimeTeacher'에 'Study'에 대한 정의가 없고 'Ex_Interface.PartTimeTeacher' 형식의 첫 번째 인수를 허용하는 확장 메서드 'Study'(가) 없습니다. using 지시문 또는 어셈블리 참조가 있는지 확인하십시오.	Program	95	Ex_Interface

[그림 7.1] 명시적 인터페이스 구현한 개체 멤버를 직접 접근할 때 오류 화면

#### 7.1.4 인터페이스에 캡슐화 가능한 멤버

인터페이스에는 메서드 외에도 속성, 이벤트, 인덱서를 멤버로 캡슐화할 수 있습니다. 속성과 인덱서는 get블록과 set블록은 원하는 블록만 것을 선택적으로 약속할 수 있습니다

다음은 인터페이스에 멤버 속성과 멤버 인덱서를 캡슐화하는 예제 코드입니다. 인터페이스 IMemo에서는 기록할 수 있는 메모의 최대 개수를 가져올 수 있는 MaxCount 속성과 원하는 위치에 메모를 설정하거나 가져올 수 있는 인덱스를 약속하였습니다.

##### ▶ 인터페이스에 멤버 속성과 인덱스를 캡슐화

```
interface IMemo
{
    int MaxCount { get; }
    string this[int index] { get; set; }
}
```

이번에는 IMemo를 구현 약속하는 Note 클래스를 만듭시다. Note 클래스에는 IMemo에서 약속된 멤버들을 구체적으로 구현해야 합니다.

##### ▶ 속성과 인덱스를 캡슐화한 인터페이스를 구현 약속

```
class Note : IMemo
{
    public int MaxCount
    {
        get {}
    }
    public string this[int index]
    {
        get{ }
        set{ }
    }
}
```

Note 클래스 생성자에서는 노트의 페이지 수를 입력 인자로 받아 최대 보관할 수 있는 메모의 개수를 정하고 메모를 남길 수 있는 페이지들을 생성합니다. 이를 위해 MaxCount의 set 블록을 추가하고 메모를 남길 수 있는 페이지를 위해 문자열 배열을 멤버로 추가할게요. 물론, set 블록은 노트 내부에서만 접근할 수 있게 private으로 접근 지정해야겠지요. 그리고 인덱스에서는 접근하려고 하는 index가 유효하면 남겨진 메모를 설정하거나 가져오기를 할 수 있게 구현합니다.

▶ Note 클래스 구현

```
class Note : IMemo
{
    string[] pages = null;

    public int MaxCount //IMemo에서 약속한 속성 구현
    {
        get; //IMemo에서 약속한 블록
        private set; //IMemo에서 약속하지 않은 블록이지만 추가했음
    }
    public string this[int index] //IMemo에서 약속한 인덱스 구현
    {
        get
        {
            if (AvailIndex(index))
            {
                return pages[index];
            }
            return null;
        }
        set
        {
            if (AvailIndex(index))
            {
                pages[index] = value;
            }
        }
    }
}
```

```

public Note(int max_memo)
{
    MaxCount = max_memo;
    pages = new string[max_memo];
}
private bool AvailIndex(int index)
{
    return (index >= 0) && (index < MaxCount);
}
}

```

다음은 Note 클래스를 사용하는 간단한 예제입니다.

▶ Note 클래스 사용 예

```

class Program
{
    static void Main(string[] args)
    {
        Note note = new Note(3);
        Console.WriteLine("최대 {0}개의 메모를 남길 수 있습니다. ", note.MaxCount);
        note[0]="1시 강의";
        note[1]="5시 미팅";
        Console.WriteLine("메모 리스트");
        for (int i = 0; i < note.MaxCount; i++)
        {
            Console.WriteLine("{0}:{1}", i + 1, note[i]);
        }
    }
}

```

▶ 실행 결과

최대 3개의 메모를 남길 수 있습니다.

1:1시 강의

2:5시 미팅

3:



## 7.2 컬렉션

프로그래밍하다 보면 여러 개의 개체를 구조적으로 관리하는 것은 자주 발생합니다. C#에서는 요소 개체의 집합체인 컬렉션을 다양하게 제공하고 있으며 같은 인터페이스를 기반으로 구현 약속하여 하나의 컬렉션 사용법을 익히면 다른 컬렉션을 사용법을 익히기 쉽습니다.

C#에서 제공되는 컬렉션에는 하나의 개체로 보관하는 컬렉션들과 키와 값을 쌍으로 보관하는 컬렉션들이 있습니다. 하나의 개체로 보관하는 컬렉션들은 IList 인터페이스를 기반이거나 ICollection 인터페이스를 기반으로 정의되어 있습니다. 그리고 키와 쌍으로 보관하는 컬렉션들은 IDictionary 인터페이스를 기반으로 정의되어 있습니다.

또한, IList와 IDictionary 인터페이스는 모두 ICollection 인터페이스를 기반으로 정의되어 있어서 같은 방법으로 사용할 수 있습니다.

그리고 C#의 일반적인 컬렉션은 어떠한 형식 요소든지 보관할 수 있습니다. 만약 특정 형식의 개체만 보관할 수 있게 보관할 요소의 형식을 명확하게 명시하여 강력하게 형식 지정하는 제네릭 컬렉션을 제공하고 있습니다.

다음은 C#에서 제공되는 컬렉션 중에서 자주 사용되는 것들입니다.

기반 인터페이스	클래스 이름	설명
IList	Array	배열 형식의 기반 클래스, 용량이 고정
	ArrayList	자동 확장이 되는 배열의 고급 버전
	List<T>	제네릭 컬렉션
IDictionary	Hashtable	해쉬 함수를 통해 빠른 검색이 가능
	SortedList	키를 기준으로 정렬
	SortedList<TKey,TValue>	제네릭 컬렉션
	Dictionary<TKey,TValue>	제네릭 컬렉션
	SortedListDictionary<TKey,TValue>	제네릭 컬렉션, 키를 기준으로 정렬
ICollection	Queue	선입선출(FIFO)
	Queue<T>	제네릭 컬렉션
	Stack	후입선출(LIFO)
	Stack<T>	제네릭 컬렉션

### 7.2.1 IEnumerable, IEnumerator 인터페이스

ICollection 인터페이스는 C#에서 제공하는 다양한 컬렉션 클래스들의 기반이 되는 인터페이스입니다. 그리고 ICollection 인터페이스는 IEnumerable 인터페이스를 기반으로 확장된 인터페이스입니다. C#에서 제공되는 컬렉션 개체가 foreach 구문을 통해 보관된 각 요소에 공통적인 작업을 수행할 수 있는 것도 내부적으로 IEnumerable 인터페이스를 기반으로 정의되었기 때문입니다.

IEnumerable 인터페이스에는 foreach 구문에서 필요한 멤버들을 약속한 IEnumerator 개체를 반환하는 GetEnumerator 메서드를 제공하고 있습니다.

#### ▶ IEnumerable, IEnumerator에 약속된 멤버들

```
interface IEnumerable
{
    IEnumerator GetEnumerator();
}

interface IEnumerator
{
    bool MoveNext(); //다음 요소 위치로 이동, 더 이상 없으면 false 반환
    void Reset();    //초기 상태로 바꿈
    Object Current   //현재 위치의 요소를 가져오기
    {
        get;
    }
}
```

## 7.2.2 ICollection 인터페이스

ICollection 인터페이스는 제네릭이 아닌 모든 컬렉션의 기반 인터페이스입니다. 제네릭 컬렉션은 ICollection<> 인터페이스를 기반으로 정의되어 있는데 여기서는 ICollection 인터페이스에 대해 살펴볼게요.

ICollection은 C#에서 제공하는 제네릭이 아닌 모든 컬렉션의 기반 인터페이스로 컬렉션에 보관된 요소의 개수와 다른 컬렉션에 보관된 요소를 복사하는 메서드, 열거자, 동기화 메서드가 약속되어 있습니다. 물론, 열거자는 ICollection이 IEnumerable을 기반으로 정의되어 있기 때문입니다.

### ▶ ICollection 인터페이스의 약속된 멤버

```
//System.Collections에 정의되어 있음
public interface ICollection : IEnumerable
{
    void CopyTo(Array array, int index); //보관된 요소들을 array에 복사
    int Count //보관된 요소 개수 가져오기
    {
        get;
    }
    bool IsSynchronized //동기화 가능 여부
    {
        get;
    }
    object SyncRoot //동기화 대상 개체 가져오기
    {
        get;
    }
}
```

ICollection의 Count속성은 보관된 요소 개수를 가져올 때 사용합니다. C#에 제네릭이 아닌 모든 컬렉션은 ICollection을 기반으로 정의되어 있어서 배열이나 ArrayList 등의 개체에서 사용할 수 있습니다.

▶ ICollection 인터페이스의 Count 속성

```
class Program
{
    static void Main(string[] args)
    {
        int[] arr = new int[3]{1,2,4};
        View(arr);
        ArrayList ar = new ArrayList();
        ar.Add(2);
        ar.Add(3);
        View(ar);
    }
    private static void View(ICollection ic)
    {
        Console.WriteLine("Count:{0}", ic.Count);
        foreach (object obj in ic)
        {
            Console.Write("{0} ",obj);
        }
        Console.WriteLine();
    }
}
```

▶ 실행 결과

Count:3

1 2 4

Count:2

2 3

CopyTo 메서드는 보관된 요소들을 입력 인자로 전달된 1차원 배열의 특정 인덱스에 복사하는 메서드입니다.

▶ ICollection 인터페이스의 CopyTo 메서드

```
class Program
{
    static void Main(string[] args)
    {
        int[] srcarr = new int[3]{1,2,4};
        int[] dstarr = new int[5]{11,12,13,4,15};
        //dstarr개체의 인덱스 2위치에 srcarr 개체에 보관된 요소들을 복사
        srcarr.CopyTo(dstarr, 2);
        View(dstarr);
    }
    private static void View(ICollection ic)
    {
        Console.WriteLine("Count:{0}", ic.Count);
        foreach (object obj in ic)
        {
            Console.Write("{0} ",obj);
        }
        Console.WriteLine();
    }
}
```

▶ 실행 결과

```
Count:5
11 12 1 2 4
```

이 외에 스레드를 사용하는 비동기식 프로그래밍에서 동기화 가능 여부를 확인하는 속성과 동기화에 사용하는 개체를 참조하는 속성이 있습니다. 여기서는 이에 대한 설명은 생략하겠습니다.

### 7.2.3 IList인터페이스

IList 인터페이스는 배열과 ArrayList의 기반 인터페이스입니다. IList 인터페이스에는 인덱스로 요소를 참조할 수 있는 멤버들을 약속하고 있습니다. 그리고 IList는 ICollection 인터페이스 기반의 형식이므로 앞에서 살펴본 ICollection에 약속한 멤버들에 대한 약속을 포함하게 됩니다. 여기에서는 새롭게 추가된 약속들에 대해서만 다룰게요.

ICollection 인터페이스에서는 요소를 추가할 때 사용하기 위해 Add 메서드와 Insert 메서드를 제공하고 있습니다. Add 메서드는 차례대로 보관할 때 사용하고 Insert 메서드는 원하는 인덱스 위치에 보관할 때 사용합니다.

`int Add(object value);` //요소를 추가하는 메서드

`void Insert(int index, object value);` //요소를 특정 인덱스 위치에 보관하는 메서드

만약, Add메서드로 1을 보관하고 Insert 메서드를 이용하여 인덱스 0에 2를 보관하면 보관된 순서는 2, 1이 됩니다. 다시 Add 메서드를 이용하여 3을 보관하면 2, 1, 3 순으로 보관되겠죠.

▶ Add메서드와 Insert 메서드를 이용하여 요소 보관

```
static void Main(string[] args)
{
    ArrayList ar = new ArrayList();
    ar.Add(1);
    ar.Insert(0, 2);
    ar.Add(3);
    foreach (int a in ar)
    {
        Console.WriteLine("{0} ", a);
    }
}
```

▶ 실행 결과

2 1 3

보관된 요소를 제거할 때는 Remove 메서드와 RemoveAt 메서드, Clear 메서드를 사용합니다. Remove 메서드는 제거할 요소를 인자로 전달하면 보관된 해당 요소를 제거해 줍니다. RemoveAt 메서드는 특정 인덱스 위치에 보관된 요소를 제거하며 Clear 메서드는 보관된 모든 요소를 제거합니다.

`void Remove(object value);` //요소를 제거하는 메서드

`void RemoveAt(int index);` //특정 인덱스에 보관된 요소를 제거하는 메서드

`void Clear();`//보관된 전체 요소를 제거하는 메서드

▶ Remove, RemoveAt, Clear 메서드를 이용하여 보관된 요소 제거

```
static void Main(string[] args)
{
    ArrayList ar = new ArrayList();
    for (int i = 1; i <= 4; i++)
    {
        ar.Add(i);
    }

    ar.Remove(3); //보관된 요소 중에 3이 있으면 제거 (1, 2, 4)
    ar.RemoveAt(0); //인덱스 0에 있는 요소 제거(2, 4)
    foreach (int element in ar)
    {
        Console.WriteLine(element);
    }
    ar.Clear(); //보관된 모든 요소 제거
    Console.WriteLine("보관된 요소 개수:{0}", ar.Count);
}
```

▶ 실행 결과

2

4

보관된 요소 개수:0

그리고 Contains 메서드를 이용하여 특정 요소가 보관되어 있는지 확인할 수 있으며 인덱서를 이용하여 보관된 요소를 변경하거나 참조할 수 있습니다. 이 외에도 컬렉션의 크기가 고정된 사이즈인지 확인하는 IsFixedSize 속성과 읽기만 가능한지에 대해 가져오기를 할 수 있는 IsReadOnly속성을 제공하고 있습니다.

```
bool Contains(object value); //요소가 보관되었는지 확인하는 메서드
object this[int index]{ get; set; } //인덱서
bool IsFixedSize { get; } //고정 사이즈 속성 - 가져오기
bool IsReadOnly { get; } //읽기만 가능한지에 대한 속성 - 가져오기
```

▶ 보관된 요소 확인, 참조 및 변경

```
static void Main(string[] args)
{
    ArrayList ar = new ArrayList();
    ar.Add(3); //3
    ar.Add(2); //3, 2
    ar[0] = 4; //인덱스 0에 보관된 요소를 4로 변경 (4, 2)
    for (int i = 1; i <= 5; i++)
    {
        if (ar.Contains(i)) //i가 보관되어 있을 때
        {
            int index = ar.IndexOf(i); //보관된 인덱스를 얻어옴
            Console.WriteLine("인덱스 {0}에 {1}보관되어 있음, index, i);
        }
    }
    for (int i = 0; i < ar.Count; i++)
    {
        Console.WriteLine("ar[{0}]: {1}", i, ar[i]); //인덱스 i에 있는 요소 출력
    }
}
```



IList 인터페이스를 기반으로 정의한 형식은 모두 예제 코드와 같은 방식으로 사용할 수 있습니다.

▶ IList에서 약속한 멤버들

```
interface IList:ICollection
{
    int Add(object value); //요소를 추가하는 메서드
    void Clear(); //보관된 전체 요소를 제거하는 메서드
    bool Contains(object value); //요소가 보관되었는지 확인하는 메서드
    int IndexOf(object value); //요소가 보관된 인덱스를 계산하는 메서드
    void Insert(int index, object value); //요소를 특정 인덱스 위치에 보관하는 메서드
    bool IsFixedSize { get; } //고정 사이즈 속성 - 가져오기
    bool IsReadOnly { get; } //읽기만 가능한지에 대한 속성 - 가져오기
    void Remove(object value); //요소를 제거하는 메서드
    void RemoveAt(int index); //특정 인덱스에 보관된 요소를 제거하는 메서드
    object this[int index] { get; set; } //인덱서

    #region ICollection 인터페이스에서 약속한 멤버
    void CopyTo(Array array, int index);
    int Count { get; }
    bool IsSynchronized { get; }
    object SyncRoot { get; }
    #endregion

    #region IEnumerable 인터페이스에서 약속한 멤버
    IEnumerator GetEnumerator();
    #endregion
}
```

## 7.2.4 IDictionary 인터페이스

IDictionary 인터페이스는 키와 값을 쌍으로 보관하는 컬렉션들의 기반 형식입니다. 그리고 IList 인터페이스처럼 IDictionary 인터페이스도 ICollection 인터페이스를 기반으로 정의한 형식입니다.

IDictionary 인터페이스에는 키와 값을 쌍으로 보관할 때 사용하는 Add 메서드를 제공하고 있으며 내부 규칙에 따라 보관될 위치를 결정하게 됩니다. 따라서 IList 인터페이스와 다르게 특정 위치에 보관하는 Insert 메서드는 제공하지 않습니다. 그리고 IDictionary 인터페이스에는 같은 키를 가진 요소를 보관할 수 없습니다. 만약, Add 메서드를 이용하여 같은 키를 가진 요소를 보관하려고 시도하면 예외가 발생합니다.

`void Add(object key, object value);` //키와 값을 쌍으로 보관하는 메서드

▶ Add 메서드를 이용하여 요소(키와 값을 쌍으로 함) 보관

```
static void Main(string[] args)
{
    Hashtable ht = new Hashtable();
    ht.Add("홍길동", "율도국");
    ht.Add("장언휴", "이에이치");

    foreach (DictionaryEntry d in ht)
    {
        Console.WriteLine("{0}:{1}",d.Key,d.Value);
    }
}
```

▶ 실행 결과

홍길동:율도국  
장언휴:이에이치

IDictionary 인터페이스에서는 인덱서를 사용할 때 키를 입력 인자로 전달하면 값을 참조할 수 있습니다. 그리고 인덱서를 대입 연산자 좌항에 사용하면 보관된 요소의 값이 바뀌게 됩니다. 만약, 해당 키를 갖는 요소가 없다면 키와 값을 쌍으로 보관해 줍니다. Add 메서드에서는 같은 키가 보관되어 있을 때 예외를 발생하지만, 인덱스는 예외를 발생하지 않고 보관된 값을 변경합니다.

```
object this[object key]{ get; set; } //인덱서
```

▶ Hashtable의 인덱서 사용

```
static void Main(string[] args)
{
    Hashtable ht = new Hashtable();
    ht["홍길동"] = "율도국";
    ht["장언휴"] = "이에이치";
    ht["홍길동"] = "대한민국";

    foreach (DictionaryEntry d in ht)
    {
        Console.WriteLine("{0}:{1}",d.Key,d.Value);
    }
}
```

▶ 실행 결과

```
장언휴:이에이치
홍길동:대한민국
```

IDictionary 인터페이스에서는 특정 키에 해당하는 요소를 제거하는 Remove 메서드를 제공합니다. IList 인터페이스에서는 특정 위치에 요소를 제거하는 RemoveAt 메서드를 제공하지만, IDictionary 인터페이스에서는 키를 기준으로 키와 값을 쌍으로 보관하기 때문에 RemoveAt 메서드를 제공하지 않습니다. 하지만 보관된 모든 요소를 제거하는 메서드인 Clear는 제공합니다.

`void Remove(object key);` //특정 키에 해당하는 요소를 제거하는 메서드

`void Clear();` //보관된 모든 요소를 제거하는 메서드

▶ IDictionary 개체에 보관한 요소 제거하기

```
static void Main(string[] args)
{
    Hashtable ht = new Hashtable();
    ht["홍길동"] = "울도국";
    ht["장언휴"] = "이에이치";
    ht["김 구"] = "대한민국";

    ht.Remove("홍길동");
    foreach (DictionaryEntry d in ht)
    {
        Console.WriteLine("{0}:{1}", d.Key, d.Value);
    }

    ht.Clear();
    Console.WriteLine("보관된 요소 개수:{0}", ht.Count);
}
```

▶ 실행 결과

김 구:대한민국

장언휴:이에이치

보관된 요소 개수:0

IDictionary 인터페이스에서는 특정 키의 요소가 보관되었는지 확인하는 Contains 메서드를 제공합니다. 그리고 키를 보관하는 컬렉션을 가져오는 Keys 속성과 값을 보관하는 컬렉션을 가져오는 Values 속성을 제공합니다. 특이한 사항으로는 키와 값으로 보관을 하기 때문에 foreach 문에서 열거할 때 DictionaryEntry 형식으로 보관된 요소를 접근하게 제공하고 있습니다. 이를 위해 IDictionaryEnumerator를 반환하는 GetEnumerator 메서드를 제공하고 있습니다.

▶ IDictionary 인터페이스에서 약속한 멤버들

```
static void Main(string[] args)
{
    Hashtable ht = new Hashtable();
    ht["홍길동"] = "율도국";
    ht["장언휴"] = "이에이치";

    Console.WriteLine("키 항목");
    foreach (string s in ht.Keys)
    {
        Console.WriteLine(" {0}", s);
    }
    Console.WriteLine("값 항목");
    foreach (string s in ht.Values)
    {
        Console.WriteLine(" {0}", s);
    }
    Console.WriteLine("키:값");
    foreach (DictionaryEntry d in ht)
    {
        Console.WriteLine(" {0}:{1}", d.Key, d.Value);
    }
}
```

이 외에도 컬렉션이 고정 사이즈인지 읽기만 가능한지를 가져오기 가능한 IsFixedSize 속성과 IsReadOnly 속성을 제공합니다.

▶ IDictionary 인터페이스에서 약속한 멤버들

```
interface IDictionary:ICollection
{
    void Add(object key, object value); //키와 값을 쌍으로 보관하는 메서드
    void Clear(); //보관된 모든 요소를 제거하는 메서드
    bool Contains(object key); //보관된 요소의 특정 키가 있는지 확인하는 메서드
    IDictionaryEnumerator GetEnumerator(); //foreach에서 요소를 열거하기 위한 메서드
    bool IsFixedSize{ get; } //고정 사이즈 속성 - 가져오기
    bool IsReadOnly{ get; } //읽기만 가능한지에 대한 속성 - 가져오기
    ICollection Keys{ get; } //키 컬렉션 속성 - 가져오기
    void Remove(object key); //특정 키에 해당하는 요소를 제거하는 메서드
    ICollection Values{ get; } //값 컬렉션 속성 - 가져오기
    object this[object key]{ get; set; } //인덱서

    #region ICollection 멤버
    void CopyTo(Array array, int index);
    int Count{ get; }
    bool IsSynchronized{ get; }
    object SyncRoot{ get; }
    #endregion

    #region IEnumerable 멤버
    IEnumerator IEnumerable.GetEnumerator();
    #endregion
}
```

이처럼 C#에서 제공하는 컬렉션은 일반적으로 필요한 멤버를 추상화하여 인터페이스 기반으로 정의하였습니다. 여러분은 하나의 컬렉션을 사용할 수 있다면 다른 컬렉션도 어렵지 않게 사용할 수 있을 것입니다.

## 7.2.5 IComparable 인터페이스와 IComparer 인터페이스

IComparable 인터페이스와 IComparer 인터페이스는 개체의 값 비교를 제공하기 위해 정의되었습니다. C#의 컬렉션은 대부분 Sort 메서드를 제공하는데 IComparable 인터페이스 기반의 요소를 보관하고 있을 때 정상적으로 동작하고 그렇지 않으면 예외를 발생합니다. 그리고 IComparer 개체를 입력 인자로 받는 Sort 메서드가 중복 정의되어 있는데 정렬하는 과정에서 입력 인자로 받은 IComparer 개체를 이용합니다.

또한, C#의 System에 정의되어 있는 기본 형식들은 IComparable 인터페이스를 기반으로 정의되어 있어서 기본 형식을 보관한 컬렉션은 Sort 메서드를 이용하여 정렬할 수 있습니다.

1차원 배열은 기반 클래스인 Array 추상 클래스의 정적 메서드인 Sort를 이용하여 정렬할 수 있습니다. Array 추상 클래스의 Sort는 1차원 배열을 입력 인자로 받습니다.

### ▶ 기본 형식을 요소로 하는 1차원 배열의 정렬

```
static void Main(string[] args)
{
    int[] arr = new int[10] { 20, 10, 5, 9, 11, 23, 8, 7, 6, 13 };

    Array.Sort(arr); //정적 메서드 Sort를 이용, 입력 인자는 1차원 배열
    foreach (int i in arr)
    {
        Console.Write("{0} ",i);
    }
    Console.WriteLine();
}
```

### ▶ 실행 결과

5 6 7 8 9 10 11 13 20 23

그리고 대부분의 컬렉션에는 Sort 메서드를 개체의 멤버로도 제공하고 있습니다. Sort 메서드가 동작하기 위해서는 보관하는 요소 형식이 기본 형식이거나 IComparable 인터페이스를 기반으로 정의한 형식이어야 합니다.

▶ IComparable에서 약속한 멤버

```
interface IComparable{ int CompareTo(object obj); }
```

▶ 사용자 정의 형식 개체를 요소로 하는 컬렉션 정렬

```
class Member:IComparable
{
    string name;
    string addr;
    public Member(string name, string addr)
    {
        this.name = name;
        this.addr = addr;
    }
    public int CompareTo(object obj) //IComparable에서 약속한 메서드 구현
    {
        Member member = obj as Member;
        if (member == null)
        {
            throw new ApplicationException("Member 개체가 아닙니다.");
        }
        return name.CompareTo(member.name);
    }
    public override string ToString()
    {
        return string.Format("이름:{0} 주소:{1}",name,addr);
    }
}
```



```

class Program
{
    static void Main(string[] args)
    {
        Member[] members = new Member[3];
        members[0] = new Member("홍길동", "울도국");
        members[1] = new Member("강감찬", "대한민국");
        members[2] = new Member("장언휴", "이에이치");

        Array.Sort(members);

        foreach (Member member in members)
        {
            Console.WriteLine(member);
        }
    }
}

```

▶ 실행 결과

이름:강감찬 주소:대한민국

이름:장언휴 주소:이에이치

이름:홍길동 주소:울도국

프로그램에서 하나 이상의 정렬 기능을 제공할 때는 어떻게 할까요? 기본이 되는 값으로 비교하는 것은 IComparable 인터페이스에서 약속한 CompareTo에서 정의하면 되겠죠. 그리고 다른 값으로 비교를 원한다면 IComparer 인터페이스 기반의 형식을 정의하세요. Sort 메서드는 IComparer 개체를 입력 인자로 받는 메서드도 지원하고 있습니다.

▶ IComparer에서 약속한 멤버

```
interface IComparer
{
    int Compare(object x, object y);
}
```

IComparer 인터페이스에는 두 개의 개체를 입력 인자로 받는 Compare 메서드를 약속하고 있습니다. 결국 비교를 담당하는 부분을 별도의 형식으로 정의하는 것입니다.

▶ IComparer에서 약속한 멤버

```
class AddrComparer : IComparer
{
    public int Compare(object x, object y) // IComparer에서 약속한 기능 구현
    {
        Member mx = x as Member;
        Member my = y as Member;

        if ((mx == null) || (my == null))
        {
            throw new ApplicationException("Member 개체가 아닌 인자가 있습니다.");
        }

        return mx.Addr.CompareTo(my.Addr);
    }
}
```

```

class Member : IComparable
{
    string name;
    public string Addr
    {
        get;
        private set;
    }

    public Member(string name, string addr)
    {
        this.name = name;
        this.Addr = addr;
    }

    public int CompareTo(object obj) //IComparable에서 약속한 기능 구현
    {
        Member member = obj as Member;
        if (member == null)
        {
            throw new ApplicationException("입력 인자가 Member 개체가 아닙니다. ");
        }
        return name.CompareTo(member.name);
    }

    public override string ToString()
    {
        return string.Format("이름:{0} 주소:{1}", name, Addr);
    }
}

```

```

class Program
{
    static void Main(string[] args)
    {
        Member[] members = new Member[3];
        members[0] = new Member("홍길동", "울도국");
        members[1] = new Member("강감찬", "대한민국");
        members[2] = new Member("장언휴", "이에이치");
        Array.Sort(members); //요소의 CompareTo 메서드 이용하여 정렬
        foreach (Member member in members)
        {
            Console.WriteLine(member);
        }
        Array.Sort(members, new AddrComparer()); //IComparer 개체를 이용하여 정렬
        Console.WriteLine("-----");
        foreach (Member member in members)
        {
            Console.WriteLine(member);
        }
    }
}

```

▶ 실행 결과

이름:강감찬 주소:대한민국

이름:장언휴 주소:이에이치

이름:홍길동 주소:울도국

-----

이름:강감찬 주소:대한민국

이름:홍길동 주소:울도국

이름:장언휴 주소:이에이치

## 8. 대리자와 이벤트

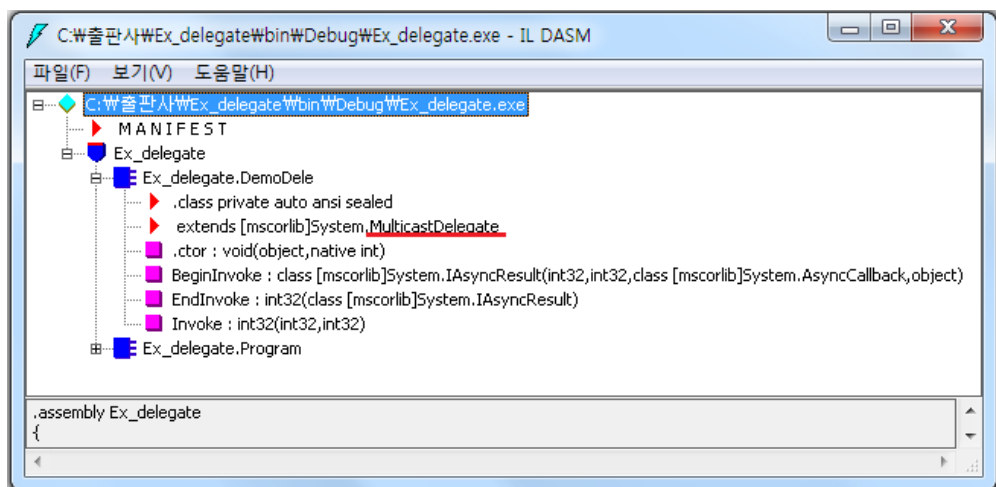
대리자는 메서드의 시그니처를 정의하는 형식으로 콜백 처리 등에서 자주 사용됩니다. 그리고 event 키워드를 명시하여 캡슐화한 대리자를 이벤트라 말합니다. 이벤트는 특정 상황이 발생하였는지를 감시하는 개체가 다른 개체에게 발생한 사실을 통보하여 처리하기 위해 캡슐화한 대리자(event 키워드를 명시한 대리자)를 말합니다. 결국 이벤트는 대리자라고 볼 수 있습니다.

### 8.1 대리자

대리자는 알고리즘을 개체화하여 인자로 전달할 때 사용되는 형식입니다. 이에 대리자를 정의할 때는 알고리즘에 필요한 인자와 리턴 형식을 명시하여 정의합니다.

`delegate` [리턴 형식] [대리자 형식 이름] ([입력 인자 리스트]);

개발자가 대리자 형식을 정의하면 컴파일러는 MulticastDelegate를 파생한 클래스를 만들어줍니다.



[그림 8.1] ildasm으로 확인한 대리자

결국 개발자가 대리자를 정의하면 .NET 어셈블리에는 MulticastDelegate를 기반으로 파생한 클래스가 만들어지는 것입니다.

▶ `delegate int DemoDele(int a, int b);`를 컴파일러가 전개한 클래스

```
class DemoDele : MulticastDelegate //MulticastDelegate에서 파생
{
    public DemoDele(object obj, IntPtr method); //obj:개체, method:멤버 메서드
    IAsyncResult BeginInvoke(int a,int b,AsyncCallback async_callback,object obj);
    int EndInvoke(IAsyncResult iar);
    int Invoke(int a, int b); //동기식 호출
};
```

대리자 개체를 생성할 때는 시그니처가 같은 메서드를 입력 인자로 생성합니다. 그리고 대리자 선언문에서 시그니처가 같은 메서드로 초기화해도 대리자 개체를 생성합니다.

▶ 대리자 개체 생성

```
void Example()
{
    DemoDele dele1 = new DemoDele(SAdd);
    DemoDele dele2 = new DemoDele(Add);
    DemoDele dele3 = Add;
}

static int SAdd(int a, int b)
{
    return a + b;
}

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

대리자 개체는 이름이 없는 메서드를 이용하여 생성할 수도 있습니다.

▶ 무명 대리자 개체 생성

```
void Example()
{
    DemoDele dele = delegate(int a, int b){ return a + b; };
}
```

대리자 개체는 메서드처럼 사용할 수 있으며 생성할 때 입력 인자로 전달한 메서드를 수행합니다. 대리자 개체를 메서드처럼 사용하면 내부적으로는 Invoke 메서드를 호출합니다. 실제로 Invoke 메서드를 호출해도 되며 이 둘의 차이는 없습니다.

▶ 무명 대리자 개체 생성

```
class Program
{
    static void Main()
    {
        DemoDele dele = Add;
        Console.WriteLine("메서드처럼 호출:{0}", dele(2, 3));
        Console.WriteLine("Invoke 메서드 호출:{0} ", dele.Invoke(2, 3));
    }
    static int Add(int a, int b)
    {
        return a + b;
    }
}
```

▶ 실행 결과

메서드처럼 호출:5

Invoke 메서드 호출:5

그리고 대리자는 서로 더하거나 빼기를 통해 수행할 메서드를 추가하거나 뺄 수 있습니다. 이는 기반 형식인 MulticastDelegate에서 제공하는 기능을 상속받기 때문이며 여러 메서드를 수행하는 대리자의 결과는 맨 마지막에 수행한 메서드의 결과입니다.

▶ 대리자의 더하기와 빼기

```
static void Main()
{
    DemoDele dele1 = Add;
    DemoDele dele2 = Sub;

    DemoDele dele3 = dele1 + dele2; // 두 개의 대리자 결합
    Console.WriteLine("{0}", dele3(4, 2));
    dele3 = dele3 - dele2; //대리자 빼기
    Console.WriteLine("{0}", dele3(4, 2));
}

static int Add(int a, int b)
{
    Console.Write ("Add ");
    return a + b;
}

static int Sub(int a, int b)
{
    Console.Write("Sub ");
    return a + b;
}
```

▶ 실행 결과

```
Add Sub 6
Add 6
```



대리자 개체의 BeginInvoke 메서드를 이용하면 수행할 메서드를 비동기적으로 호출할 수 있습니다. 하지만 BeginInvoke 메서드로 비동기로 대리자를 호출하려면 대상 메서드가 하나여야 합니다.

BeginInvoke 메서드를 호출할 때는 대리자 선언에 명시한 입력 인자와 비동기 작업이 완료 시에 수행할 종료 콜백 메서드와 종료 콜백 메서드에 전달할 인자를 전달할 수 있습니다. 종료 콜백 메서드의 시그니처는 리턴 형식이 void이며 입력 인자로 IAsyncResult입니다.

▶ 종료 콜백 메서드의 예

```
static void EndSum(IAsyncResult iar)
```

그리고 종료 콜백 메서드에서는 EndInvoke 메서드를 호출하여 비동기로 대리자를 호출한 작업을 정상적으로 종료하세요. 종료 콜백 메서드의 입력 인자로 전달된 IAsyncResult 개체를 참조 연산으로 AsyncResult 개체를 참조하여 AsyncDelegate 속성으로 대리자 개체를 가져올 수 있습니다. 그리고 대리자 개체의 EndInvoke 메서드를 호출하면 작업 결과를 반환받아 사용할 수 있습니다.

▶ 종료 콜백 메서드의 예

```
static void EndSum(IAsyncResult iar)
{
    AsyncResult ar = iar as AsyncResult;
    DemoDele dele = ar.AsyncDelegate as DemoDele;
    Console.WriteLine("수행 결과:{0}",dele.EndInvoke(iar));
}
```

종료 콜백 메서드에 null을 인자로 전달할 때는 BeginInvoke에서 전달받은 IAsyncResult 개체의 IsCompleted 속성을 확인하여 요청한 비동기 작업이 완료되었는지를 확인할 수 있습니다. 그리고 EndInvoke 메서드를 호출하여 비동기 작업을 종료할 수 있으며 작업 결과를 반환받아 사용할 수 있습니다.

▶ 비동기로 대리자 호출

```
using System;
using System.Runtime.Remoting.Messaging;
using System.Threading;
namespace Ex_delegate
{
    delegate int DemoDele(int a,int b);
    class Program
    {
        static void Main()
        {
            DemoDele dele = Sum; //Sum을 대상으로 하는 대리자 생성
            dele.BeginInvoke(1, 5, EndSum, "TEST"); //비동기로 대리자 호출
            for (int i = 0; i < 5; i++)
            {
                Console.WriteLine("Main:{0}", i);
                Thread.Sleep(100); //테스트를 위해 0.1초 멈추게 하였음
            }
            Console.ReadLine();
        }
        static int Sum(int a, int b)
        {
            int sum = 0;
            for (; a <= b; a++)
            {
                sum += a;
                Console.WriteLine("Sum:{0}", a);
                Thread.Sleep(100); //테스트를 위해 0.1초 멈추게 하였음
            }
            return sum;
        }
    }
}
```

```

static void EndSum(IAsyncResult iar)
{
    object obj = iar.AsyncState; //BeginInvoke의 마지막 전달 인자를 얻어옴

    AsyncResult ar = iar as AsyncResult;
    DemoDele dele = ar.AsyncDelegate as DemoDele; //대리자 개체 참조
    Console.WriteLine("전달받은 인자:{0} 수행 결과:{1}",obj,dele.EndInvoke(iar));
}
}
}

```

▶ 실행 결과

```

Main:0
Sum:1
Sum:2
Main:1
Main:2
Sum:3
Main:3
Sum:4
Sum:5
Main:4
전달받은 인자:TEST 수행 결과:15

```

예제 코드는 비동기 작업이 있어서 실제 수행 결과는 다양하게 나옵니다.

## 9.2 이벤트

이벤트는 특정 사건이 발생하는 것을 감시하는 개체가 이를 처리하는 개체에게 이벤트가 발생하였을 때 필요한 인자들과 함께 발생 사실을 통보하기 위한 특별한 멤버입니다. 이벤트를 감시하여 다른 개체에게 발생한 사실을 통보하는 개체를 이벤트 게시자라고 하며 이벤트가 발생하였을 때 이벤트 게시자로부터 통보받아 처리하는 개체를 이벤트 구독자라고 합니다.

C#에서 이벤트는 대리자 멤버를 캡슐화할 때 `event` 키워드를 명시하면 됩니다. 그리고 이벤트로 만들기 위해 정의한 대리자는 이벤트를 통보한 개체와 이벤트 처리에 필요한 인자를 포함하여 시그니처를 정의하도록 가이드하고 있습니다.

### ▶ 이벤트를 위한 대리자 정의

```
//obj: 이벤트를 통보한 개체 , e: 이벤트 처리에 필요한 인자  
public delegate void AddMemberEvent(object obj, AddMemberEventArgs e);
```

### ▶ EventArgs 기반의 파생 형식 정의

```
class AddMemberEventArgs:EventArgs  
{  
    public string Name{ get; private set; }  
    public string Addr{ get; private set; }  
    public Member Member{ get; private set; }  
    public AddMemberEventArgs(Member member)  
    {  
        Name = member.Name;  
        Addr = member.Addr;  
        this.Member = member;  
    }  
}
```

이벤트 게시자에는 이벤트 구독자가 이벤트 핸들러를 등록할 수 있게 이벤트 멤버의 가시성을 public으로 지정하는 것이 일반적입니다. 그리고 이벤트 게시자는 특정한 상황이 발생하면 이벤트 인자를 생성하여 등록된 이벤트 핸들러를 호출하여 전달합니다.

▶ 이벤트 게시자

```
class MemberManager
{
    public event AddMemberEvent AddMemberEventHandler = null;
    ... 중략...
    protected virtual void OnAddMember(Member member)
    {
        if (AddMemberEventHandler != null) //등록된 이벤트 핸들러가 있을 때
        {
            AddMemberEventHandler(this, new AddMemberEventArgs(member));
        }
    }
}
```

이벤트 구독자는 이벤트 처리기를 이벤트 게시자에게 등록하는 부분이 있어야 하며 등록이 되어 있으면 이벤트가 발생하면 이벤트 게시자가 등록한 이벤트 처리기를 호출해 줍니다. 이벤트 처리기에서는 해당 이벤트가 발생했을 때 처리할 구문을 작성하는 메서드입니다.

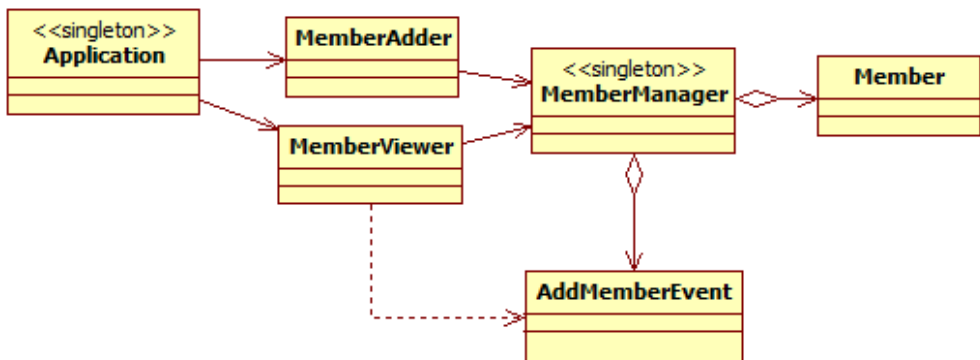
▶ 이벤트 구독자

```
class Application
{
    Application()
    {
        mm.AddMemberEventHandler +=
            new AddMemberEvent(OnAddMemberEventHandler); //등록
    }
    // 이벤트 처리기
    void OnAddMemberEventHandler(object obj, AddMemberEventArgs e)
    {
        // 이벤트 처리 구문
    }
}
```

다음은 이벤트를 사용하는 예를 보여주는 회원 관리 프로그램입니다.

프로그램에는 회원 관리 프로그램을 구동하는 Application이 있으며 회원 정보를 추가하는 MemberAdder가 있습니다. 그리고 회원 정보를 출력하는 MemberViewer가 있고 회원 정보를 관리하는 MemberManager가 있습니다.

MemberManager는 회원 추가 요청이 있을 때 새로운 회원 정보일 때 회원을 추가합니다. 그리고 회원이 추가되면 MemberViewer가 화면에 추가된 회원 정보를 출력합니다. 이를 위해 MemberManager는 회원 정보 추가 이벤트를 멤버로 캡슐화합니다. 즉, 이벤트 게시자 역할을 수행합니다. MemberViewer에는 MemberManager에게 이벤트 처리기를 등록하여 회원 정보가 추가될 때 이를 통보받아 화면에 회원 정보를 출력하는 이벤트 구독자입니다.



[그림 45] 클래스 다이어그램

먼저, 간단하게 Member 클래스를 정의합니다. 여기에서는 회원의 이름과 주소를 멤버 속성으로 갖게 정의할게요.

▶ 대리자 개체 사용 예

```
public class Member
{
    public string Name
    {
        get;
        private set;
    }
    public string Addr
    {
        get;
        private set;
    }
    public Member(string name, string addr)
    {
        Name = name;
        Addr = addr;
    }
}
```



그리고 회원이 추가될 때 이벤트 처리에 필요한 이벤트 인자를 정의합시다. 이벤트 처리기에서는 추가된 회원 정보가 필요하겠죠. 그리고 회원 이름과 주소를 바로 접근할 수 있으면 좀 더 편의성이 높아질 것입니다.

▶ 대리자 개체 사용 예

```
public class AddMemberEventArgs:EventArgs
{
    public string Name
    {
        get;
        private set;
    }
    public string Addr
    {
        get;
        private set;
    }
    public Member Member
    {
        get;
        private set;
    }
    public AddMemberEventArgs(Member member)
    {
        Name = member.Name;
        Addr = member.Addr;
        this.Member = member;
    }
}
```

회원이 추가될 때 이벤트를 사용할 것이므로 대리자 형식을 하나 정의해야겠죠.

▶ 대리자 정의

```
//obj: 이벤트를 통보한 개체, e: 이벤트 처리에 필요한 인자
```

```
public delegate void AddMemberEvent(object obj, AddMemberEventArgs e);
```

이벤트 게시자는 MemberManager 개체입니다. 멤버 이벤트를 캡슐화해야겠죠.

```
public event AddMemberEvent AddMemberEventHandler = null;
```

그리고 회원 정보를 보관하는 컬렉션을 멤버로 캡슐화합니다. 여기서는 회원 이름을 키로하고 멤버 개체를 값으로 하는 Dictionary를 사용할게요.

```
Dictionary<string, Member> members = new Dictionary<string, Member>();
```

회원을 추가하는 메서드를 추가합니다. 입력 인자로 이름과 주소를 받게 합니다. 그리고 인자로 받은 이름이 이미 컬렉션에 있다면 추가 실패를 반환하고 그렇지 않으면 회원 추가하고 참을 반환하면 되겠죠.

```
internal bool AddMember(string name, string addr)
{
    if (members.ContainsKey(name))
    {
        return false;
    }
    OnAddMember(new Member(name, addr));
    return true;
}
```

OnAddMember 메서드에서는 컬렉션에 회원 개체를 보관하는 작업과 등록된 이벤트 핸들러가 있는지 확인하여 이를 호출해 주어야겠죠.

```
protected virtual void OnAddMember(Member member)
{
    members[member.Name] = member;
    if (AddMemberEventHandler != null)
    {
        AddMemberEventHandler(this, new AddMemberEventArgs(member));
    }
}
```

그리고 여기서 MemberManager는 단일체로 정의할게요.

```
static public MemberManager Singleton
{
    get;
    private set;
}
static MemberManager()
{
    Singleton = new MemberManager();
}
private MemberManager()
{
}
```

▶ 이벤트 게시자 정의

```
class MemberManager
{
    public event AddMemberEvent AddMemberEventHandler = null;
    Dictionary<string, Member> members = new Dictionary<string, Member>();
    static public MemberManager Singleton{ get; private set; }

    static MemberManager()
    {
        Singleton = new MemberManager();//단일체 생성
    }
    private MemberManager(){ } //단일체로 구현하기 위하여 private으로 접근 지정
    protected virtual void OnAddMember(Member member)
    {
        members[member.Name] = member;
        if (AddMemberEventHandler != null) //이벤트 핸들러가 등록되어 있을 때
        {
            AddMemberEventHandler(this, new AddMemberEventArgs(member));
        }
    }
    internal bool AddMember(string name, string addr)
    {
        if (members.ContainsKey(name))
        {
            return false;
        }
        OnAddMember(new Member(name, addr));
        return true;
    }
}
```

이벤트 구독자인 MemberViewer에서는 생성자에서 이벤트 게시자인 MemberManager 단일체를 참조하여 이벤트 처리기를 등록해야겠죠.

```
public MemberViewer()
{
    MemberManager mm = MemberManager.Singleton;
    mm.AddMemberEventHandler +=
        new AddMemberEvent(mm_AddMemberEventHandler);
}
```

그리고 이벤트 처리기에서는 이벤트 처리 인자로 회원 정보를 얻어와서 화면에 출력합니다.

#### ▶ 대리자 정의

```
class MemberViewer
{
    public MemberViewer()
    {
        MemberManager mm = MemberManager.Singleton;

        //이벤트 처리기 등록
        mm.AddMemberEventHandler +=
            new AddMemberEvent(mm_AddMemberEventHandler);
    }

    void mm_AddMemberEventHandler(object obj, AddMemberEventArgs e)
    {
        Console.WriteLine("회원이 추가되었습니다.");
        Console.WriteLine("이름:{0} 주소:{1}", e.Name, e.Addr);
    }
}
```

나머지 부분도 구현해 봅시다.

MemberAdder에서는 회원을 추가하기 위해 사용자와 상호 작용하는 부분을 작성할게요. 왜 이렇게 기능이 작게 분리하여 구현하는 것인지 의문인 분들도 계실 겁니다. 지금 예제는 이벤트를 설명하기 위한 프로그램으로 큰 의미를 부여할 필요는 없습니다. 다만 윈도우즈 응용 프로그램이라고 생각하시면 메인 창과 회원 정보 추가 창을 분리하여 구현할 수 있을 것이며 프로그램 구조를 비슷하게 콘솔 응용으로 표현한 것이라 생각하세요.

▶ MemberAdder

```
class MemberAdder
{
    public void AddMember()
    {
        MemberManager mm = MemberManager.Singleton;

        Console.WriteLine("이름을 입력하세요.");
        string name = Console.ReadLine();
        Console.WriteLine("주소를 입력하세요.");
        string addr = Console.ReadLine();
        if (mm.AddMember(name, addr) == false)
        {
            Console.WriteLine("{0} 정보는 이미 존재합니다.", name);
        }
    }
}
```

마지막으로 Application 클래스와 진입점을 작성해 봅시다.

Application 클래스도 단일체로 정의할게요. 그리고 멤버 필드에 MemberAdder 개체와 MemberViewer 개체를 캡슐화할게요. Application 개체가 생성될 때 이들 개체는 생성하고 구동이 되면 최종 사용자와 상호 작용을 하면 회원을 추가해 나갈게요.

사용자와 상호 작용을 하는 부분은 아주 간단하게 회원 추가를 할 것인지 프로그램을 종료할 것인지만 선택할 수 있게 하겠습니다.

#### ▶ Application

```
class Application
{
    MemberAdder ma = null;
    MemberViewer mv = null;

    internal static Application Singleton //단일체에 대한 속성
    {
        get;
        private set;
    }

    static Application()
    {
        Singleton = new Application();//단일체 생성
    }

    private Application()//단일체로 표현하기 위해 private으로 접근 지정
    {
        ma = new MemberAdder();
        mv = new MemberViewer();
    }
}
```

```

internal void Run()
{
    bool check = false;

    do
    {
        Console.WriteLine("회원 추가:[1]");
        Console.WriteLine("다른 키를 누르면 종료됩니다.");

        check = (Console.ReadKey().Key == ConsoleKey.D1);
        if (check)
        {
            Console.WriteLine();
            ma.AddMember();
        }
    } while (check);
}

class Program
{
    static void Main(string[] args)
    {
        Application application = Application.Singleton;
        application.Run();
    }
}

```



## 9. .NET 어셈블리

.NET 어셈블리는 .NET 프레임워크 응용 프로그램을 구성하는 기본 컴포넌트입니다. 컴포넌트 기반의 프로그래밍은 모듈의 재사용성을 높여줍니다. .NET 응용 프로그램을 제작할 때 미리 작성된 라이브러리를 사용할 수 있는데 이 때 사용하는 라이브러리도 .NET 어셈블리이며 실행 파일도 .NET 어셈블리입니다.

.NET 어셈블리는 EXE 파일 혹은 DLL 파일 형태로 만들 수 있으며 하나 이상의 모듈을 포함할 수도 있습니다. 그리고 .NET 어셈블리는 배포를 단순화하여 COM에서 발생했던 많은 배포 문제를 해결하였습니다.

.NET 어셈블리는 자기 기술적인 메타 데이터를 갖고 있고 레지스트리 항목에 종속되지 않는 구성 요소로 디자인되어 있기 때문에 병행(Side by Side) 실행이 가능합니다. COM에서는 버전 관리 문제를 비롯하여 DLL 충돌 문제가 있었지만 .NET 어셈블리는 강력한 이름을 사용하여 DLL 충돌 문제를 해결하였습니다.

이로써 같은 명칭의 여러 어셈블리를 설치 가능할 뿐만 아니라 이를 사용하는 응용들은 자신들에 맞는 어셈블리를 사용할 수 있습니다. .NET 어셈블리를 배포하는 방법에는 응용 프로그램의 전용으로 배포하는 방법과 여러 응용 프로그램이 공용으로 사용할 수 있게 공용으로 배포하는 방법이 있습니다.

## 9.1 .NET 어셈블리 구성 요소

.NET 어셈블리는 여러 종류의 구성 요소로 구성됩니다.

첫 번째 구성 요소로 어셈블리 매니페스트가 있습니다. 어셈블리 매니페스트는 어셈블리의 버전 요구 사항과 보안 ID를 지정하는 데 필요한 모든 메타데이터와 어셈블리의 범위를 정의하고 참조를 확인하는 데 필요한 모든 메타 데이터를 포함하고 있습니다. 어셈블리 매니페스트는 어셈블리 이름, 버전 번호, 문화권, 강력한 이름의 정보, 어셈블리에 포함된 파일 목록, 형식 참조 정보로 구성됩니다.

그리고 어셈블리에 정의된 형식들에 대한 형식 메타 데이터와 실제 수행해야 할 코드인 MSIL과 리소스로 구성됩니다.

어셈블리는 하나의 물리적 바이너리로 되어 있는 단일 파일 어셈블리와 여러 개의 물리적 바이너리로 되어 있는 다중 파일 어셈블리로 나눌 수 있습니다. .NET에서는 어셈블리의 부속이 될 수 있는 모듈을 만들 수 있게 제공하고 있는데 모듈은 어셈블리에 포함됩니다. 즉, 하나의 어셈블리가 다른 어셈블리에 있는 형식을 사용하기 위해서는 단순히 사용할 어셈블리를 참조만 하면 되는데 모듈에 있는 것을 사용할 때는 모듈을 포함하여 어셈블리를 만들어야 합니다.

그리고 .NET 어셈블리를 구성하는 물리적 바이너리에는 리소스 파일, 매니페이스, DLL, 모듈 등이 있습니다.

## 9.2 어셈블리 만들기

여기서는 .NET 어셈블리를 만들어 보기로 합니다. .NET 어셈블리 중에 실행 파일을 만드는 것은 이미 알고 있으므로 라이브러리를 만드는 것을 하겠습니다. 그리고 다중 파일 어셈블리의 부속이 되는 모듈도 만들어 보기로 합니다.

### 9.2.1 라이브러리 제작

.NET에서 라이브러리는 쉽게 제작할 수 있습니다. 코드를 작성하는 부분에서 다른 어셈블리에서 사용할 수 있는 형식을 정의할 때 `public` 키워드를 명시하기만 하면 됩니다. 멤버도 마찬가지로 다른 어셈블리에서 접근을 허용하게 하려면 접근 지정을 `public`으로 하면 됩니다.

즉, 다른 어셈블리에 정의된 형식이나 멤버 중에 사용할 수 있는 형식과 멤버는 접근 지정이 `public`으로 되어 있다는 것 말고는 차이가 없습니다. COM과 Win32API에서의 복잡하고 까다로운 표현을 접근 지정자로 간단하게 해결하였습니다.

```
public class Member
{
    ...중략...
    public Member(string name, string addr)
    {
        ...중략...
    }
}
```

간단한 라이브러리를 하나 만들어 봅시다. 먼저 컴파일러 옵션을 이용하여 만드는 것을 보여 드리고 통합 개발 환경에서 만드는 것을 보여 드릴게요.

컴파일러 옵션을 이용하는 것을 보여 드리기 전에 메모장을 이용하여 소스 코드를 편집해 봅시다. 편집할 코드에는 다른 어셈블리에서 접근할 수 있는 `Member` 클래스와 접근할 수 없는 `MemberInfo` 클래스를 정의할게요.

그리고 `Member`에는 이름과 주소를 입력 인자로 받는 생성자와 이름 속성, 주소 속성의 `get` 블록을 `public`으로 접근 지정하여 다른 어셈블리에서 사용할 수 있게 합니다.

▶ 라이브러리 코드 예 (Member.cs)

```
namespace MemberLib
{
    public class Member //다른 어셈블리에서 접근 가능
    {
        MemberInfo mi = null;

        public string Name //다른 어셈블리에서 접근 가능
        {
            get
            {
                return mi.Name;
            }
        }

        public string Addr //다른 어셈블리에서 접근 가능
        {
            get
            {
                return mi.Addr;
            }
        }

        public Member(string name, string addr) //다른 어셈블리에서 접근 가능
        {
            mi = new MemberInfo(name, addr);
        }

        public override string ToString()//다른 어셈블리에서 접근 가능
        {
            return mi.ToString();
        }
    }
}
```

```

class MemberInfo //다른 어셈블리에서 접근 불가
{
    internal string Name
    {
        get;
        private set;
    }
    internal string Addr
    {
        get;
        private set;
    }
    internal MemberInfo(string name, string addr)
    {
        Name = name;
        Addr = addr;
    }
    public override string ToString()
    {
        return Name;
    }
}
}

```

Visual Studio 명령 프롬프트에서 `csc /t:library /out:MemberLib Member.cs`를 입력하시면 컴파일되어 `MemberLib.dll` 파일명으로 .NET 어셈블리가 만들어집니다.



[그림 9.1] csc 명령어를 이용하여 .NET 어셈블리 만들기

다음은 콘솔 명령어 csc의 옵션들입니다.

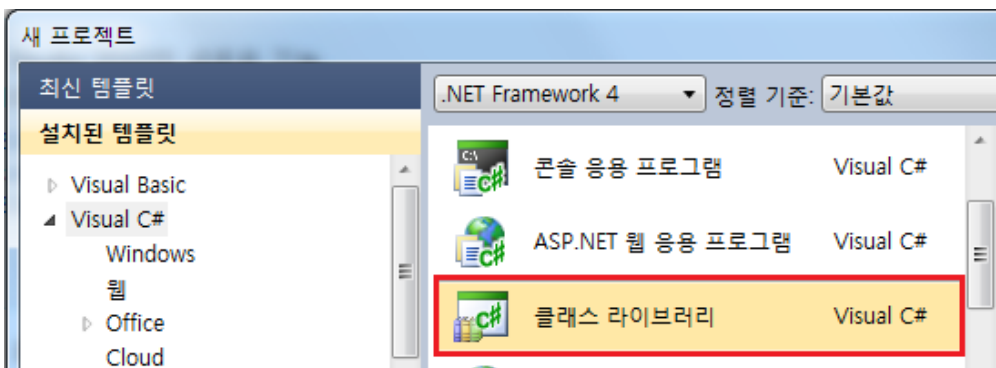
▶ 기본 옵션

/out:<file>	만들어질 어셈블리 이름(출력 파일 이름)을 지정
/target:exe	콘솔 실행 파일을 만들기 (기본) (약식: /t:exe)
/target:winexe	Windows 실행 파일을 만들기 (약식: /t:winexe)
/target:library	라이브러리를 만들기 (약식: /t:library)
/target:module	모듈을 만들기 (약식: /t:module)
/delaysign[+]	어셈블리 서명을 연기
/doc:<file>	XML 문서 파일 생성
/keyfile:<file>	강력한 이름의 키 파일을 지정
/keycontainer:<string>	강력한 이름의 키 컨테이너를 지정
/platform:<string>	실행할 수 있는 플랫폼을 지정 (x86, Itanium, x64, anycpu)

▶ 자주 사용하는 옵션

/reference:<file list>	어셈블리 참조 (약식:/r)
/addmodule:<file list>	모듈을 포함
/help	csc 커맨드 사용법 (약식:/?)
/lib:<file list>	참조를 검색할 추가 디렉터리를 지정

Microsoft Visual Studio로 라이브러리를 만들 때는 다음과 같이 새 프로젝트 템플릿에서 클래스 라이브러리를 선택하여 만들면 됩니다.



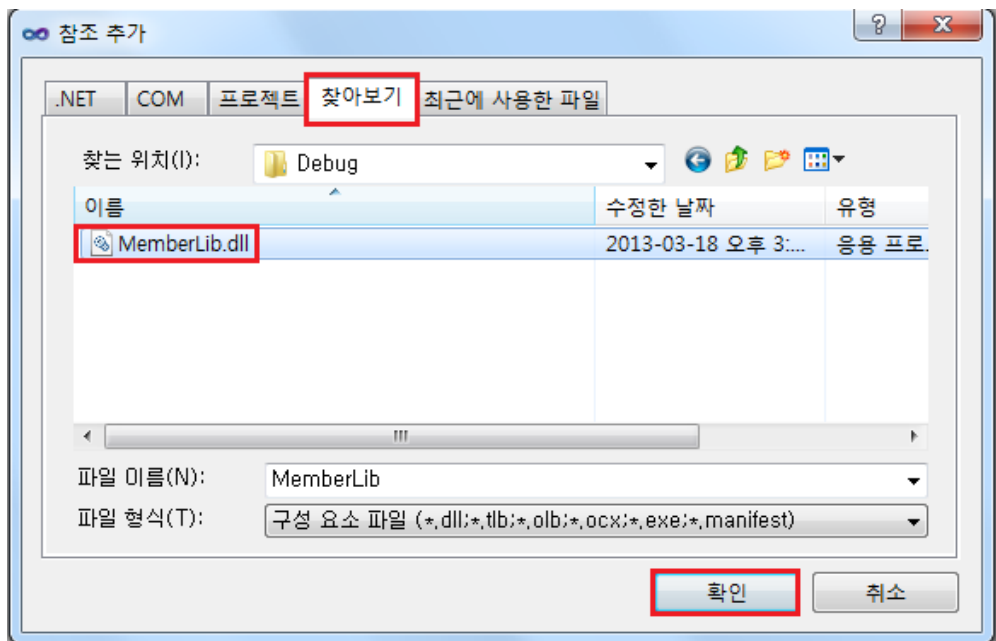
[그림 9.2] Microsoft Visual Studio에서 클래스 라이브러리 만들기

### 9.3 전용 어셈블리

이번에는 앞에서 만든 어셈블리를 전용으로 사용하는 방법에 대해 살펴보기로 합니다.

전용 어셈블리는 사용하는 응용 프로그램과 함께 어셈블리를 배포하여 해당 응용 프로그램에 의해서만 이용되는 어셈블리를 말합니다. 배포하는 위치는 응용 프로그램과 같은 디렉토리이거나 하위 디렉토리가 됩니다.

그리고 .NET에서 어셈블리 배포 방법은 기본으로 전용으로 되어 있어서 통합 개발 환경인 Microsoft Visual Studio를 가지고 개발할 때 특정 어셈블리를 참조하고 컴파일하면 자동으로 참조하는 어셈블리를 출력 폴더로 복사해 주어 개발자에게 편의성을 제공하고 있습니다. 솔루션 탐색기에서 참조 폴더에 마우스 우측 버튼을 클릭하여 나오는 컨텍스트 메뉴에서 참조 추가를 선택하여 원하는 어셈블리를 선택하세요.



[그림 9.3] 참조 추가창에서 추가할 어셈블리 선택하기

Microsoft Visual Studio에서 콘솔 응용 프로그램 프로젝트를 생성하여 작성한 어셈블리를 사용하는 코드를 작성해 봅시다.

▶ 자주 사용하는 옵션

```
using System;
using MemberLib;

namespace UsingMemberLib
{
    class Program
    {
        static void Main(string[] args)
        {
            Member member = new Member("홍길동", "대한민국");
            Console.WriteLine("이름:{0} 주소:{1}", member.Name, member.Addr);
        }
    }
}
```

▶ 실행 결과

이름:홍길동 주소:대한민국

물론, MemberLib에서 public으로 지정하지 않은 MemberInfo 형식과 멤버는 사용하지 못합니다.

CLR은 참조하는 전용 어셈블리의 위치를 찾는 일도 수행하는데 프로빙이라 합니다. 제일 먼저 응용 프로그램 디렉토리를 먼저 검사하고 없으면 구성 파일에 명시된 디렉토리를 검사하게 됩니다. 구성 파일은 응용 프로그램 구성 파일과 게시자 정책 파일, 시스템 구성 파일이 있으며 자세한 사항은 MSDN 런타임에서 어셈블리를 찾는 방법을 참고하세요.



## 9.4 공용 어셈블리

공용 어셈블리는 전역 어셈블리 캐시에 있는 어셈블리를 말합니다. 전역 어셈블리 캐시에는 어셈블리의 강력한 이름으로 배포합니다. 강력한 이름은 어셈블리 이름, 버전 번호, 문화권, 어셈블리 ID, 공개 키, 디지털 서명 등으로 구성됩니다. 이처럼 전역 어셈블리 캐시에는 같은 이름의 DLL도 강력한 이름이 다르면 배포할 수 있습니다. 따라서 서로 다른 버전을 사용하는 응용 프로그램들이 자신에게 맞는 어셈블리를 사용할 수 있으며 이를 병행(Side by Side) 실행이라 합니다.

간단하게 라이브러리를 만들고 병행 실행이 되는 것을 확인해 봅시다.

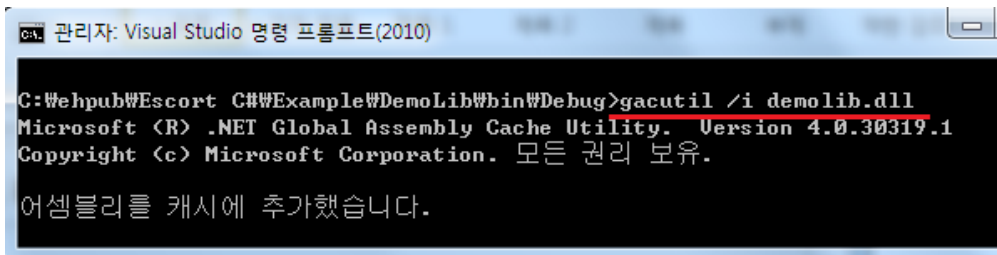
▶ DemoLib (버전 1.0.0.0)

```
using System;
namespace DemoLib
{
    public class DemoClass
    {
        public void Foo()
        {
            Console.WriteLine("Foo 1");
        }
    }
}
```

공용 어셈블리를 만들기 위해 솔루션 탐색기에 프로젝트에 우측 마우스를 클릭하여 나오는 컨텍스트 메뉴에서 속성을 선택하세요. 그리고 서명 탭을 선택하신 후에 어셈블리 서명 체크 박스를 선택하시고 강력한 이름 키 선택 콤보 박스에서 새로 만들기를 선택합니다. 강력한 이름 키 만들기 창에서 키 이름을 입력하면 프로젝트 항목에 만든 키가 보이게 됩니다.

그리고 프로젝트를 빌드하시면 출력 폴더(기본: Debug)에 DLL 파일이 만들어집니다.

이제 작성한 DLL을 전역 어셈블리 캐시에 배포합니다. Windows7부터는 보안이 강화되어 관리자 권한으로 배포해야 합니다. 관리자 권한으로 Visual Studio 명령 프롬프트를 실행한 후에 `gacutil /i demolib.dll` 을 치면 배포가 됩니다.



[그림 9.4] gacutil 명령어로 전역 어셈블리 캐시에 배포

이제 배포한 공용 어셈블리를 사용하는 응용 프로그램 프로젝트를 생성하세요. 그리고 참조 추가를 통해 demolib.dll를 추가하세요. 전용 어셈블리에서 참조 추가하는 것과 같은 방법입니다.

전용 어셈블리를 참조 추가하면 자동으로 프로젝트 출력 폴더에 참조한 어셈블리 파일이 추가되었지만, 공용 어셈블리를 참조 추가하면 프로젝트 출력 폴더에 추가되지 않습니다. 공용 어셈블리는 전역 어셈블리 캐시에 배포된 것을 사용하기 때문입니다.

그리고 프로젝트 참조 폴더에 해당 어셈블리에 오른쪽 마우스 버튼을 클릭하여 컨텍스트 메뉴에서 속성을 선택하세요. 속성 창을 보면 특정 버전을 선택할 수 있습니다. 만약, 특정 버전 속성을 false로 설정하면 새로운 버전이 배포되면 자동으로 최신 버전을 사용합니다. 하지만 특정 버전 속성을 true로 설정하면 참조한 버전의 어셈블리를 사용하게 됩니다.

여기서는 병행(Side by Side) 실행을 확인하기 위함이니 특정 버전 속성을 True로 합니다. 그리고 사용하는 코드를 작성하세요. 그리고 실행해 보세요.

▶ 공용 어셈블리(버전 1.0.0.0)를 사용하는 예제 코드

```
using DemoLib;

namespace UsingDemoV1
{
    class Program
    {
        static void Main(string[] args)
        {
            DemoClass dc = new DemoClass();
            dc.Foo();
        }
    }
}
```

▶ 실행 결과

Foo 1

그리고 DemoLib를 수정합니다.

▶ DemoLib (버전 1.0.0.1)

```
public void Foo()
{
    Console.WriteLine("Foo 2");
}
```

프로젝트 속성 창을 열어서 응용 프로그램 탭에 어셈블리 정보 버튼을 클릭합니다. 그리고 버전 정보를 수정하세요. 버전 정보를 수정하였으므로 조금 전에 만들어 배포한 어셈블리와 파일명과 확장자는 같지만 강력한 이름은 다릅니다. 전역 어셈블리 캐시에서는 강력한 이름으로 배포하기 때문에 재배포 가능합니다.

프로젝트를 빌드한 후 gacutil 명령어를 이용하여 다시 배포합니다. 그리고 방금 만들었던 응용을 실행해서 참조한 버전의 어셈블리를 사용하고 있음을 확인하세요. 그리고 새로운 버전을 참조하여 사용하는 응용 프로그램을 작성해 보세요. 소스 코드는 먼저 만들었던 것과 같게 만들어도 됩니다. 그리고 두 개의 응용 프로그램을 실행해 보시면 각각 자신이 참조하는 버전의 어셈블리를 사용함을 확인하실 수 있습니다.

참고로 프로젝트 속성 창에서 빌드 탭을 선택하여 XML 문서 파일을 체크하여 세 줄 주석을 작성하면 Visual Studio의 향상된 인텔리센스 기능을 활용할 수 있습니다.

## 9.5 모듈 작성

이번에는 .NET 어셈블리의 부속이 될 수 있는 모듈을 작성하는 방법을 알아보시다. 그리고 모듈을 포함하는 .NET 어셈블리도 만들어 봅시다.

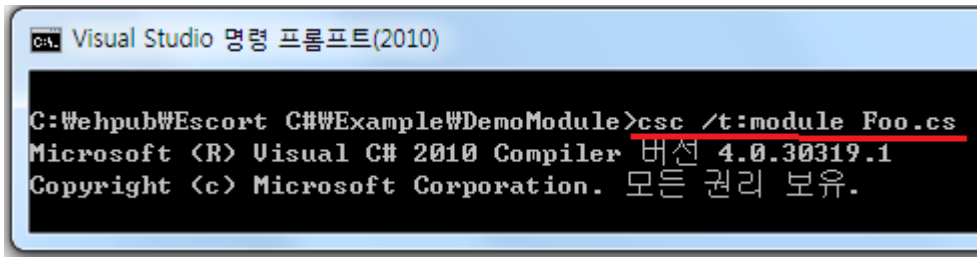
모듈은 Visual Studio 명령 프롬프트에서 csc 명령어를 이용하여 컴파일해야 합니다. 먼저 간단한 소스 코드를 편집한 후에 모듈을 만듭시다.

▶ 모듈로 만들 코드(Foo.cs)

```
namespace DemoModule
{
    public class Foo
    {
        public string Name
        {
            get;
            private set;
        }

        public Foo(string name){    Name = name;    }
        public override string ToString(){ return Name;    }
    }
}
```

그리고 Visual Studio 명령 프롬프트에서 `csc /t:module Foo.cs`를 입력하면 모듈이 만들어집니다.



```
C:\ Visual Studio 명령 프롬프트(2010)

C:\Webpub\Escort C#\Example\DemoModule>csc /t:module Foo.cs
Microsoft (R) Visual C# 2010 Compiler 버전 4.0.30319.1
Copyright (c) Microsoft Corporation. 모든 권리 보유.
```

[그림 9.5] csc 커맨드로 모듈 만들기

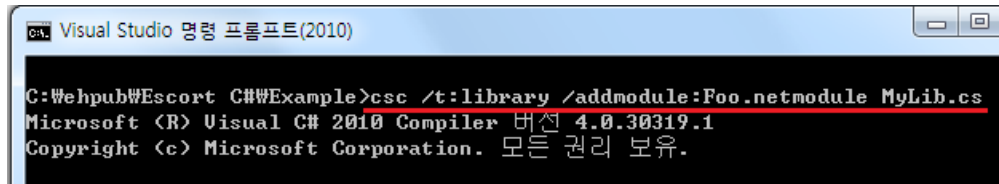
이처럼 만들어진 모듈은 .NET 어셈블리의 부속이 될 수 있습니다. 따라서 하나의 모듈을 포함하는 두 개의 .NET 어셈블리를 참조하면 충돌이 납니다. 모듈은 .NET 어셈블리가 아니라 .NET 어셈블리에 포함될 수 있는 부속임에 주의하세요.

간단하게 모듈을 포함하는 라이브러리 코드도 편집합니다.

▶ 모듈을 포함하는 라이브러리 코드(MyLib.cs)

```
using System;
using DemoModule;
namespace MyLib
{
    public class IncludeModule
    {
        public void UsingModule()
        {
            Foo foo = new Foo("테스트");
            Console.WriteLine(foo);
        }
    }
}
```

마찬가지로 Visual Studio 명령 프롬프트에서 csc 명령어를 이용하여 모듈을 포함하는 라이브러리를 제작합니다. csc /t:library /addmodule:Foo.netmodule MyLib.cs라고 입력하세요.



[그림 9.6] csc 커맨드로 라이브러리 만들기

이처럼 다른 모듈을 포함하고 있는 .NET 어셈블리를 다중 파일 어셈블리라 얘기합니다. Microsoft 사에서는 기존 COM 전략이 실패하였다는 의미에서 .NET을 만들었습니다.

많은 부분에서 변화가 있지만 그중에서도 자기 기술적인 .NET 어셈블리, 강력한 이름으로 공용 어셈블리 배포와 이로써 병행(Side by Side) 실행, 간단한 접근 지정만으로도 외부에서 접근 가능한 형식과 멤버를 지정, XML 문서 출력으로 인텔리센스 기능을 활용할 수 있게 한 점은 많은 개발자가 효과적으로 비즈니스 프로젝트를 수행할 수 있게 해 줍니다.

이제 C#언어의 기본적인 부분에 소개를 마쳤습니다. 앞으로 리플렉션, 직렬화, 리모팅, 마이그레이션 등을 익혀 보다 심화 학습을 해 나가세요. 그리고 설계 능력을 키우실 분들은 Escort GoF의 디자인 패턴 C#을 보면서 GoF의 디자인 패턴을 보시면 도움이 될 거예요. 아울러 CBD 개발 방법론에 대한 학습도 해 나가실 바랄게요.

그리고 ADO.NET, XML.NET, Windows Form with C#, ASP.NET, 웹 서비스를 학습하신 후에 WinFX인 WPF와 Silverlight, LINQ, WCF 등을 학습하시면 더욱 발전할 수 있을 것입니다. 물론, XNA나 Open API, Open CV, Open GL 등의 기술을 접목하기 위한 학습과 프로젝트 수행이나 DDK, WDF 등을 학습하는 것도 괜찮을 거예요.

처음 C#을 접하는 이들도 볼 수 있게 책을 구성해서 보다 심화된 내용을 포함하지 못하고 있지만, 반드시 다뤄야 한다고 생각하는 것들은 전달했다고 생각합니다. 아무쪼록 대한민국의 건강한 IT 환경을 구축하는데 밑알이 되길 기대할게요.