



## 목차

.NET Framework 버전과 기술.....	7
1) 프로그래밍 언어의 이해.....	7
✧ 컴퓨터 프로그램 언어의 종류.....	8
2) .NET Framework 의 이해.....	12
✧ .NET Framework 란?.....	12
✧ .NET Framework 에 무엇이 들어 있는가?.....	13
✧ 컴파일이란 무엇인가요?.....	15
✧ .NET Framework 컴파일은 2 번 일한다?.....	16
✧ 관리되는 코드란 .....	17
✧ Visual C# 이란 무엇인가?.....	17
✧ C# 언어의 이해 .....	18
✧ .NET Framework 연산자 .....	39
✧ 산술 연산자(수식 연산자).....	40
✧ 증감 연산자.....	40
✧ 관계연산자.....	40
✧ 할당 연산자.....	41
✧ 논리 연산자.....	41
✧ 비트 연산자.....	42
✧ 프로그램 제어 기술의 이해 .....	44
✧ 조건(분기) 문.....	44
✧ switch 문.....	45

✧ 반복문 – while .....	46
✧ 반복문 – do ~ while .....	47
✧ 반복문 – for .....	47
✧ 중첩 반복문 – for { for { for .....	48
✧ 반복문 – foreach .....	48
✧ 반복문 – break .....	48
✧ 반복문 – continue .....	48
✧ 반복문 – goto .....	49
✧ Visual Studio 와 언어 버전 .....	50
3) Windows OS 와 스레드 기술의 이해 .....	54
✧ 스레드의 종류 .....	55
✧ 스레드의 데이터 처리 .....	56
4) .NET Framework 메모리 관리와 코드 작성 .....	56
✧ 개발자에 따른 메모리 관리 .....	62
✧ 관리되지 않는 메모리 관리 .....	64
5) Windows OS 의 내부 .....	65
6) .NET Framework 와 Thread 처리기술 .....	68
다중 스레드를 사용해야 하는 경우 .....	69
다중 스레드의 장점 .....	69
다중 스레드의 단점 .....	70
스레딩 및 응용 프로그램 디자인 .....	71

스레딩 및 예외 사항 .....	71
✧ 스레드 기술의 사용 .....	72
스레드 만들기.....	72
데이터를 스레드에 전달 및 스레드에서 데이터 검색.....	72
콜백 메서드로 데이터 검색.....	73
.NET Framework 동기/비동기 프로그래밍 .....	74
1) System.Tread 네임스페이스.....	74
네임스페이스 .....	74
✧ System.Threading 클래스.....	74
구조체 .....	77
대리자 .....	78
열거형 .....	79
2) 동기와 비동기 프로그래밍.....	79
✧ 비동기 (Asynchronous: 동시에 일어나지 않는, 非同期: 같은 시기가 아닌).79	
✧ 동기 (synchronous: 동시에 일어나는, 同期: 같은 시기).....	80
3) 비동기 프로그래밍의 코드 작성.....	81
비동기 작업 형식 .....	81
제안된 비동기 시나리오.....	81
비동기 서비스 작업 구현 .....	82
4) 개체지향 처리와 데이터 주고 받기 .....	83
기본 아키텍처.....	84

Message 클래스.....	85
메시지 본문 .....	85
WCF 의 메시지 사용.....	87
메시지 헤더 .....	88
메시지 속성 .....	88
채널 스택.....	90
서비스 프레임워크에서 데이터 표시 .....	93
프로그래밍 모델.....	93
프로그래밍 모델 제한.....	96
메시지 포맷터.....	96
Serialization.....	97
5) Windows 이벤트 처리방식의 이해.....	98
이벤트 .....	98
대리자 .....	99
이벤트 데이터.....	100
이벤트 처리기.....	100
정적 및 동적 이벤트 처리기.....	101
여러 이벤트 발생시키기 .....	101
SDK 의 이해와 어셈블리.....	102
1) SDK 를 이용한 개발 .....	102
2) #define 와 #undef 등의 전처리기 .....	102

✧ #define.....	103
✧ #undef.....	104
3) 어트리뷰트 .....	104
✧ Custom Attribute 의 정의 .....	105
4) C#에서의 포인터 .....	106
5) 어셈블리의 개념과 아키텍처 .....	109
어셈블리 통합 및 구성 요소.....	109
지원되는 런타임 버전.....	110
6) Visual Studio Debug 의 이해와 예외처리 .....	112
✧ 예외처리(Exception) 배우기 .....	113
7) Visual Studio Debugging 기술 익히기 .....	117
✧ Debugging 의 기초 – 디버그 메뉴와 중단점 .....	117
✧ Debugging 의 활용과 Tip.....	119
✧ 그 외 Debugging Window.....	123
8) 개체지향이란 무엇인가? .....	125
9) 개체와 수명에 대한 이해.....	127
.NET Framework 와 개체지향 프로그래밍.....	128
개체와 클래스의 관계.....	128
✧ 메서드.....	128

## .NET Framework 버전과 기술

이곳에서는 프로그램을 시작하기 전에 알아야 하는 기본적인 것과 프로그램 개발자라는 직업 또는 앞으로 우리가 해야 할 내용에 대한 기본적인 역사와 Microsoft .NET 이라는 내용을 알아보겠습니다.

### 1) 프로그래밍 언어의 이해

컴퓨터의 시작을 이해하기 위하여 우리가 가끔(?) 또는 뭔가를 계산하기 위해서 사용하는 계산기를 떠올려 보시면 될 것 같습니다. 계산기도 일종의 기계이며, 지금은 스마트폰에 있는 하나의 앱을 이용하기도 합니다. 이런 기계들이 처음에는 매우 크고 다루기에도 많은 노력이 필요했던 시기가 있었으며, 그 시기를 지나 기술의 발전으로 지금의 손안의 컴퓨터가 있는 것이라 할 수 있습니다.

기술의 발전으로 최초의 컴퓨터는 에니악(ENICA)이라 할 수 있습니다. 이 에니악은 현재 자기 집의 작은 방 또는 거실을 생각해 보시고, 그 정도 크기에 각종 전기 배선과 일종의 카드라 불리우는 것, 그리고 천공기라는 것을 통해 계산을 했던 컴퓨터 입니다.

여기서 전기 배선에 대한 것을 기억해야 합니다. 이 전기 배선이 우리가 앞으로 배워야 할 프로그래밍 언어라 할 수 있습니다.

이 에니악은 뛰어난 계산 능력과 그 당시에 신기술이지만 문제는 있었습니다. 바로 크기만 있는 것이 아니라 진공관이라는 것을 사용하였기 때문에 이 열기 또는 대단한 것이었습니다. 이후에 컴퓨터 프로그램 개발자라면 한 사람의 이름 정도는 알아야 하는 바로 존 폰 노이만(John von Neumann)이란 사람이 이룬 개선하고 향상시켜 놓은 것이 에드박(EDVAC : 1951 년)이란 기계였습니다. 이 기계가 지금의 컴퓨터는 이 에드박이 시초라 할 수 있습니다. 일단 이런 기계적인 기술의 발전은 지속적으로 발전을 하였지만 그 기계라 불리우는 컴퓨터 안에서 사용되는 기술은 발전이 거의 없었다 하여도 될 정도였습니다.

컴퓨터 안에서 사용되는 기술의 발전 없이 지속적으로 0 또는 1이라는 조합으로 계속 컴퓨터라는 기계를 다루었고, 이 조합을 Bit로 구성된 기계의 조합이었습니다. 즉 컴퓨터라는 기계를 가지고 뭔가를 하려면 0 또는 1이라는 조합을 정말 잘 맞추어야 하고 보통 수학이라고 하는 숫자와의 싸움을 하였던 것입니다. 수학이라면 보통 싫어라 하는 사람이 많을 수도 있는데 이런 0 또는 1이라는 비트 조합이라는 것을 우리 사람이 이해라 수 있는 언어로 바꾸어서 표현하고, 기계인 컴퓨터는 그 언어를 이해해서 우리가 요청한 일을 수행하는 것을 컴퓨터 프로그래밍 또는 프로그래밍 개발이라고 할 수 있습니다.

즉 컴퓨터 프로그램 언어란 "사람이 이해라 수 있는 언어로 컴퓨터란 기계에 명령을 내려서 일을 시키는 언어를 배운 것"이라 할 수 있습니다. 사실 컴퓨터는 매우 똑똑한 비서처럼 일을 하지만, 이 일을 시키는 것은 바로 우리이기 때문에 일을 잘 시켜야 을 잘 할 수 있듯이 프로그래밍 언어의 기초를 확실히 이해하고 프로그램 언어를 이용한 개발을 할 때 실수를 많이 줄일 수 있을 것입니다.

#### ☆ 컴퓨터 프로그램 언어의 종류

컴퓨터 프로그램 언어는 우리가 생활할 때 사용하는 말 또는 언어와 같다고 생각하면 될 듯 합니다. 사람이 태어나서 처음 말을 배울 때 한 글자, 한 단어를 천천히 배우듯이 배우면 됩니다. 잘못 배우게 되면 다들 아실 듯 합니다. 말을 잘못하면 큰일 일어날 수 있듯이 프로그램 언어도 잘못하면 컴퓨터에 치명적인 문제를 줄 수 있으므로 이제 프로그램 언어의 종류에 대하여 천천히 알아 보겠습니다.

#### 언어들의 종류

- 어셈블리 언어 : 초기 기계언어인 비트의 조합을 사람이 이해할 수 있는 언어로 표현된 프로그래밍 언어이지만 단순히 아기의 걸음마 수준의 언어라 생각할 수 있으며, 초기 언어로 표현하기 위한 한계를 가지고 있습니다. 따라서 이 언어를 배우기 위해서는 프로그래밍 전문 과학자 또는 천재 또는 똑똑한 사람들이 주로 이 언어를 배우고 사용했습니다.
- 포트란 : 하드웨어(기계)의 발전으로 어셈블리 언어의 단점인 표현의 한계를 대중화 되어 가고 있는 컴퓨터에서 사용하기 위한 언어로 존 배커스(John Backus)란 사람이



IBM 이란 회사에서 프로젝트 경험을 기반으로 사람의 언어에 가까운 최초의 프로그래밍 언어입니다. Fortran(포트란) 언어라는 언어를 개발하고 과학 계산등에 주로 많이 사용하였습니다.

- 코볼 : 컴퓨터라 옛날에 비해 작아지고, 대중화 되면서 과학자들이 아닌 일반 회사에서 사용하기 시작하면서 그에 맞게 고안된 프로그램 언어입니다. 이 언어는 미국 국방부를 중심으로 경성된 프로그램 언어로서 그룹 CODASYL(conference on data system Language)에 의해 1960 년 처음 제정되어 사무용 컴퓨터들이 늘어나면서 사용 되어진 언어 입니다. 이 언어는 영어회화와 비슷한 구어체 문장 형태로 기술할 수 있도록 설계되어 사용되어 1968 년 미국에서 사무처리 표준 언어로 제정되어 1974 년 미국 표준 코볼이 완성되었고, 1985 년에는 기능 보강하여 COBOL-85 를 완성합니다.
- C : 이 언어는 말이 필요 없는 언어라 할 수 있습니다. 우리가 영어를 왜 배우려 할 까요? 영어를 생각하시면 됩니다. 프로그래밍 언어를 배우는 사람이라면 한번쯤 들어봤거나 오래 전 또는 학교에서 프로그래밍 공부를 하신 분이라면 들어 봤거나 사용해 봤던 언어가 바로 C 언어 입니다. 이 언어는 1964 년 MIT 공대와 AT&T 의 벨 연구소, GE 의 멀틱스(Multics)라고 하는 운영체제의 개발에서 시작되어진 프로젝트가 있었으며, 이 프로젝트가 실패 되었지만, 이 프로젝트에 참여자인 데니스 리치(Dennis Ritchie)와 켄 톰슨(Ken Thompson)이란 분들이 운영체제 개발이라고 하는 소중한 경험을 기반으로 회사에서 새로운 운영체제 개발의 도전으로 유닉스(Unix)라는 것을 만들고, 이 언어에 사용하던 B 언어(켄 톰슨이 만든 언어)를 데니스 리치가 B 언어의 특징을 이용하여 새로운 프로그래밍 언어를 개발한 언어가 C 언어 입니다. 유닉스는 C 언어가 탄생하고 데니스 리치와 켄 톰슨이란 분은 이 언어로 새롭게 유닉스를 C 언어로 재 작성합니다. 이 당시 유닉스는 C 언어로 재 작성되어 다른 컴퓨터에도 쉽게 이식할 수 있어서 다른 대학과 기업들이 소요하고 있는 컴퓨터들도 이 유닉스를 사용하게 되어 C 언어도 자연스럽게 같이 보급되어 사용되게 됩니다.
- C++ : C 언어는 앞에서 이야기한 것처럼 다른 컴퓨터에 쉽게 이식할 수 있는 유닉스라는 언어로 만들어졌고, 이 언어가 매우 훌륭하게 만들어져 있지만 그래도 뭔가 부족하거나 뭔가 좀 아쉬운 것이 있었습니다. 이 언어의 부족하거나 아쉬운 부분을 비아네 스트러우스롭(Bjarne Stroustrup)교수가 이것을 채우게 됩니다. 바로 개체지향 이라는 것을 채우게 됩니다. 그래서 C 언어에 기능을 더한 것이라 해서 C++(우리는 보통 C 뽀뽀 또는 C 플러스 플러스)라고 합니다.

- Java : 개체지향 프로그래밍 언어로 보안성이 뛰어나고, 코드를 다른 운영체제에서 사용할 수 있는 기능을 제공하며, C++ 언어의 장점을 살리면서 90년대 컴퓨터 환경을 지원하도록 만든 프로그래밍 언어입니다.
- Basic : 1964년 미국 다트머스 대학의 존 케머니(John Kemeny)와 토마스 커츠(Thomas Kurtz)교수에 의해 탄생한 언어입니다. Basic(Beginner's All-Purpose Symbolic Instruction Code)의 약자로 베이직이라 불러주면 됩니다. 이 언어는 우리가 잘 알고 있는 사람이라 할 수 있는 Microsoft의 빌 게이트(Bill Gates)도 이 프로그래밍 언어를 사용하여 프로그램을 개발한 언어입니다. 이 언어는 컴퓨터 프로그래밍 언어가 과학자 또는 엔지니어들의 전유물이 아닌 학생부터 일반 사람들이 쉽게 프로그래밍 언어를 접할 수 있도록 하기 위한 언어를 고안해 냈고, 이 언어가 바로 베이직이라 합니다.

#### 우리가 배워야 할 언어

- ✧ Microsoft 언어 : Microsoft라는 회사는 소프트웨어 개발회사이며, 앞에서 잠시 언급한 빌 게이츠란 분이 만든 회사입니다. 1975년도에 창립한 회사라서 매우 오래된 회사입니다. 이 회사의 언어를 왜 배워야 해요? 라고 하면, 현재 우리가 사용하는 업무용 컴퓨터 또는 집에서 사용하는 컴퓨터에서 화면에 보이는 프로그램이 바로 Microsoft에서 만든 Windows라는 프로그램입니다. 따라서 이 Windows라는 환경에서 프로그램 개발할 수 있는 언어를 배우는 것입니다.
- ✧ 스마트 폰 개발 언어 : 스마트 폰에서 사용되는 환경은 크게 2가지 정도로 먼저 생각하면 될 것 같습니다. 사실 더 있지만, 현재 우리가 주로 접하는 환경이 안드로이드라는 환경과 아이폰이라는 환경이기 때문입니다. 다행히 Microsoft에서는 하나의 개발에서 Windows, 안드로이드, 아이폰에서 사용할 수 있는 프로그램을 만들 수 있도록 하고 있습니다. 단 제한이 있으며, 이를 항상 생각하고 있어야 합니다.
- ✧ Microsoft C#.NET : 일반적으로 C#이라고 하지만 정확히는 Visual C#.NET이라고 하는 것이 좋습니다. 편하게 C#이라고도 하며 우리가 앞으로 배워야 할 언어입니다.
- ✧ Microsoft .NET 언어 : Microsoft에서는 2000년도에 뭔가 고민하면서 프로그램 개발 방식에 대한 획기적인 것을 서서히 모습을 보이기 시작하는 시기입니다. 바로 .NET이라는 것을 발표하게 되고, 이 기반에서 사용되는 언어가 바로 C#, Visual Basic.NET이라는 뒤에 .NET이라는 표현을 하는 언어가 나옵니다.

앞에서 잠시 설명한 언어가 몇 가지가 있습니다. 이 언어들은 사실 일부분이라 할 수 있습니다. 잘 생각해 보면 우리가 외국을 나가게 되면 꼭 알아야 하는 언어가 있습니다. 바로 몸으로 표현하는 언어입니다. 이 몸으로 표현하는 언어가 어셈블리이고, C 또는 C++가 영어로 생각하면 될 것 같습니다. 그럼 영어라 할 수 있는 C 또는 C++를 배우면 되는 것 아닌가 생각할 수 있습니다. 앞에서 제가 C# 언어를 설명하기 전 Microsoft 언어를 보시면 PC 에서 사용하는 환경은 Windows 입니다. 네 바로 Windows 환경이기 때문에 Windows 환경에서는 .NET 언어가 바로 영어라 할 수 있습니다. 즉 .NET 언어의 하나의 C# 언어는 C++에서 좋은 것만 가지고 와서 만든 언어입니다. Microsoft 의 앤더스 해지스버그(Anders Hejlsberg)라는 분이 만들었고 현재 지속적으로 발전하고 있는 언어입니다. C# 언어는 Microsoft .NET 환경을 처음 발표할 때 나온 언어로 오로지 Microsoft .NET 환경만을 위한 언어입니다. 물론 다른 언어도 있지만 1999 년 처음 우리 앞에 베타라는 이름으로 나타난 C# 언어의 초기부터 지금까지 계속 사용되고 있는 언어입니다. 이 언어는 ECMA/ISO 규격에 기반을 두고 있는 Microsoft .NET 언어 중 하나이지만 특히 우리나라에서는 많은 사랑을 받고 있는 언어이기도 합니다.

## 2) .NET Framework 의 이해

C# 언어를 배우기 전에 우리는 먼저 .NET Framework 에 대해서 이해를 해야 합니다.  
아니? 왜? 무엇 때문에 C# 언어를 배우야 하는데 .NET Framework 를 이해를 해야 하는데  
하면, 바로 .NET Framework 에 있는 기능을 C# 언어로 표현해서 Windows 라는 운영 환경  
또는 운영체제라는 것에 명령을 내려야 하기 때문입니다. 운영체제라는 환경이  
Microsoft 에서는 지속적으로 개선을 통한 업그레이드가 되고 최신의 Windows  
환경에서도 항상 .NET Framework 환경은 제공되고 있으며, 이 환경에서 사용되는 언어를  
배우고, 이 언어를 이용하여 우리가 원하는 명령을 Windows 에 전달하여 우리가 원하는  
결과를 받을 수 있도록 앞으로 .NET Framework 를 배워보도록 하겠습니다.

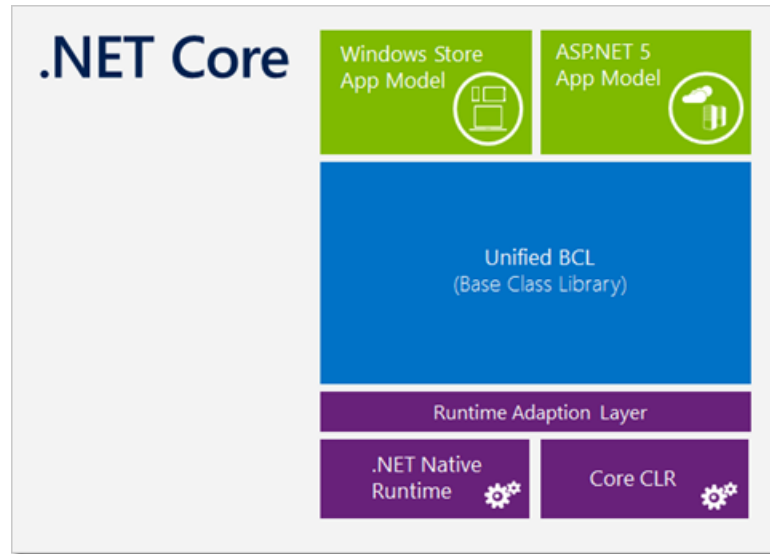
### ☆ .NET Framework 란?

.NET Framework 는 Microsoft 에서 만들어 낸 응용 프로그램 개발을 위한 새롭고도  
혁신적인 플랫폼입니다.

플랫폼(Platform) 컴퓨터의 기본이 되는 특정 프로세서와  
하나의 컴퓨터 시스템을 바탕으로 하는 운영체제를 말하기도  
합니다. 즉 여러분들이 만든 프로그램을 실행할 수 있는  
기반이 되는 시스템입니다.

이 시스템 기반으로 하는 회사가 구글, 애플등이 있으며,  
현재 Microsoft 도 플랫폼 기업이라 할 수 있습니다.

즉 우리가 프로그램을 개발할 때, 일반적인 개발은 PC 또는 데스크 탑과 노트북이라는  
환경과 Windows 운영체제에서 실행되는 프로그램을 만듭니다. 이런 프로그램을 응용  
프로그램을 만든다고 하는데, 서버라고 하는 운영체제도 있습니다. 이런 서버에서 특정  
목적으로 서비스 하는 프로그램을 만들 때에는 서버 프로그램 개발이라고도 합니다. 즉  
환경이 Windows 환경이지만 노트북, 데스크 탑, 서버 등등 환경이 매우 다양합니다.  
지금은 IoT 라는 용어가 있는 쉽게 말해 스마트 폰이나 주변 작은 기계들에게 까지  
실행되는 환경에 프로그램을 만들어야 하는 상황이라는 뜻입니다. 조금 더 쉽게 말하자면  
Windows 가 실행되는 기계가 매우 많고, 그 기계 마다 실행되는 프로그램을 만들 때마다  
프로그램 언어 기술을 새롭게 배우거나, 또는 기계 마다 새로운 프로그램 개발에 필요한  
것을 설치하고 할 일이 많아 진다는 것입니다. 이것을 좀 줄여 보자는 것입니다.



하나의 플랫폼에서 돌아간다면, 화면 크기에 맞추고 그에 맞는 실행결과만 다르게 표시하는 프로그램을 만드는데 한가지 프로그램 개발 기술을 가지고 할 수 있다면 어떨까요? .NET Framework 는 바로 이런 기반의 시작이라 할 수 있습니다. .NET Framework 는 쉽게 “프로그램을 개발하는 플랫폼” 이라 할 수 있습니다. 여기서 나는 노트북 환경과 데스크 탑에서 실행되는 프로그램을 만든다고 하면, .NET Framework 에서는 “데스크 탑 응용 프로그램을 개발하는 플랫폼”이라 할 수 있고, 서버 환경에서 실행되는 환경이라는 “서버 응용 프로그램을 개발하는 플랫폼”이라 할 수 있고, 인터넷 환경에서는 “인터넷 응용 프로그램을 개발하는 플랫폼”이라 할 수 있습니다.

다시 쉽게 표현하자면, 응용 프로그램을 만드는 플랫폼으로 어떤 기계나 환경에서도 (여기서는 Windows 에 제한을 하겠습니다.) 실행되는 프로그램을 만들 수 있도록 도와주는 훌륭한 것입니다.

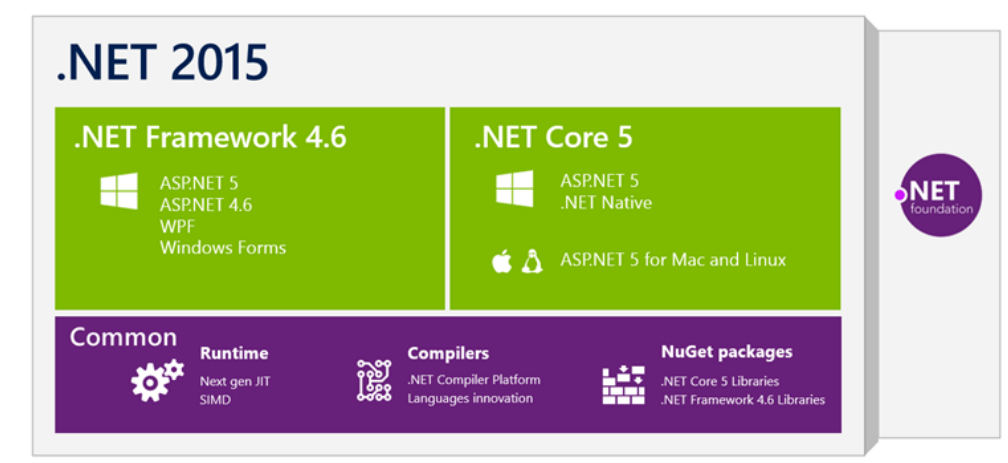
#### ✧ .NET Framework 에 무엇이 들어 있는가?

.NET Framework 에서는 .NET 기반환경에서 실행되는 프로그램을 만드는 언어들이 포함되어 있습니다. 여기에는 컴파일러도 포함되어 있고, 여러분들이 만들고자 하는 기본적인 기능을 제공도 합니다. 만약 여러분들이 만들고자 하는 기능이 없다면 Microsoft 에서는 SDK(Software Development Kit)라는 것을 제공하고 있는데 이 SDK 를 설치하면 만들고자 하는 기능을 프로그래밍 할 수 있도록 해줍니다.

Microsoft 다운로드 사이트에서는 Windows 환경에서 실행되는 모든 것을 개발할 수 있도록 추가 SDK 라는 것을 제공합니다.

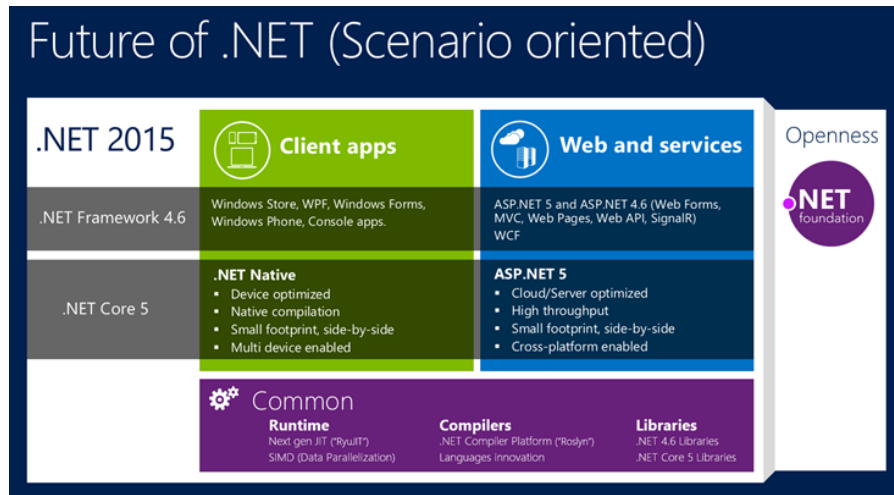
SDK 는 매우 중요한 것으로 .NET Framework SDK 를 설치하면 기본적으로 BCL(Base Class Library)이 있으며 프로그램 언어는 BCL 안에 있는 기능을 호출하여 사용하는 것을 배우는 것입니다. 쉬운 말로 우리가 원하는 기능들은 BCL 에 있는 것을 가져다가 사용하고 없는 것이 있다면 추가로 SDK 를 설치하면 된다는 것입니다.

그럼 이제 BCL(Base Class Library)에 있는 것을 호출 했다면 이것을 실행하는 환경이 있는데 Windows 에서 바로 실행되는 것이 아닙니다. 꼭 기억해야 하는 것이 바로 CLR(Common Language Runtime)이라는 것입니다. 우리가 Windows 를 설치하고 .NET Framework 만 설치하면 바로 BCL 과 CLR 이것만 있다고 보면 됩니다.



이제 마지막으로 프로그래밍 언어는 .NET Framework 설치하면 끝입니다. 지금 바로 메모장 프로그램을 이용해서 프로그램 언어로 표현하고 하나의 작업만 하면 끝입니다. 그런데 메모장으로 프로그램을 개발하면 어떨까 생각을 해 보시면 어떨까요? 아마도 당장이라도 프로그램을 그만 두고 싶거나, 이거 뭐하자는 것인지 등 의문이 많을 것입니다. 그래서 한번 프로그램 개발을 하려면 쉽게 개발하기 위하여 설치해야 하는 것들이 매우 많습니다. 설치해야 하는 것은 뒤에서 다시 이야기 하고, 그럼 .NET Framework 는 이것만

있느냐? 아닙니다. 바로 컴파일 이란 것이 있고, 여러분들이 만들어 놓은 것을 .NET Framework 의 CLR 에서 돌아갈 수 있도록 하는 작업을 하는 것이 바로 컴파일 입니다.

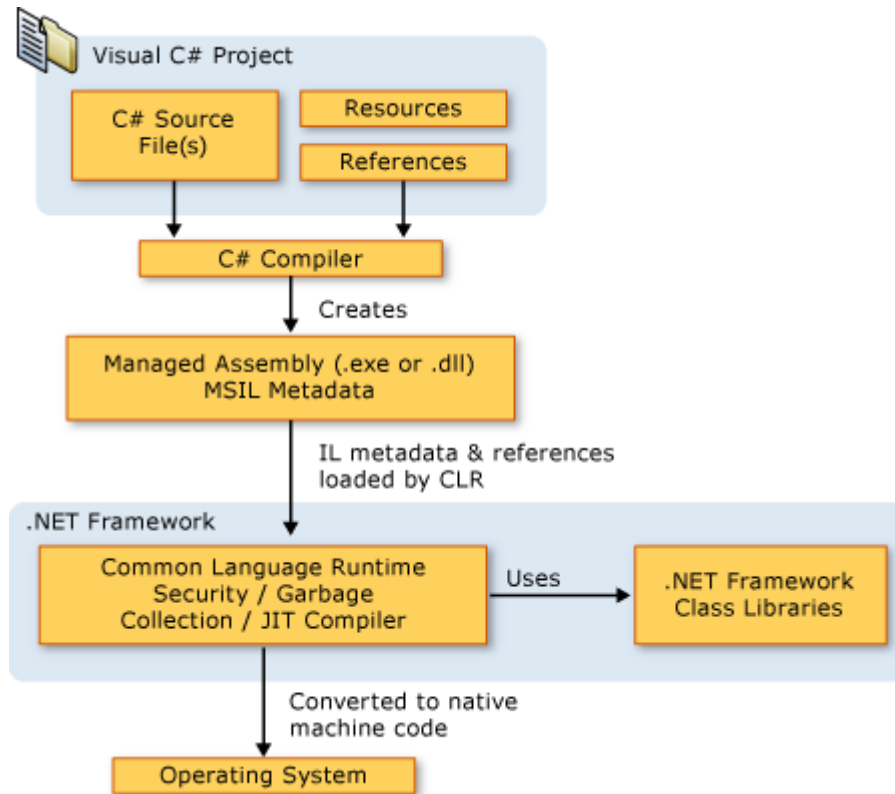


#### ☆ 컴파일이란 무엇인가요?

사실 컴파일은 마지막에 여러분들이 프로그램 언어를 다 배우고 언어로 여러분들이 하얀색 화면에 색이 있는 글씨를 이용하여 작성한 결과물을 CLR 에서 실행되게 하는 것입니다.

여기서 한가지 중요한 것은 .NET Framework 에서는 자기네 .NET 프로그램 언어끼리 서로 이해하기 편하게 주고 받는 방식이 있습니다. 쉽게 표현하자면 각 지역의 사투리가 있고, 그 사투리로 이야기를 하면 서로 이해하기 어려울 수가 있습니다. 이런 것을 해소하기 위해서 우리는 표준말이라는 것이 있고 이 표준말을 이용하면 서로 이해하기 쉬울 수 있습니다. 옛날 언어들은 자기네 표현의 말만 이해할 수 있었는데 .NET Framework 에서는 그렇지 않습니다. 바로 CTS(Common Type System) 이라는 표준말이 있습니다.

이런 표준말인 CTS 는 .NET Framework 를 사용하는 언어들 사이에서 전문용어로 상호운영성(Interoperability)를 용이하게 한다고 합니다. 어렵다면 쉬운 말로 표준어로 대화를 하면 제주도의 지역의 방언을 표준어로 대체하여 사용한다는 것입니다. 따라서 CTS 란 녀석은 .NET Framework 언어들끼리 서로 지켜야 하는 하나의 표준이라 할 수 있습니다. 이렇게 표준으로 되어 있는 프로그램 개발의 결과물을 컴파일이라는 단계를 거치고 실제 실행이 되게 됩니다.



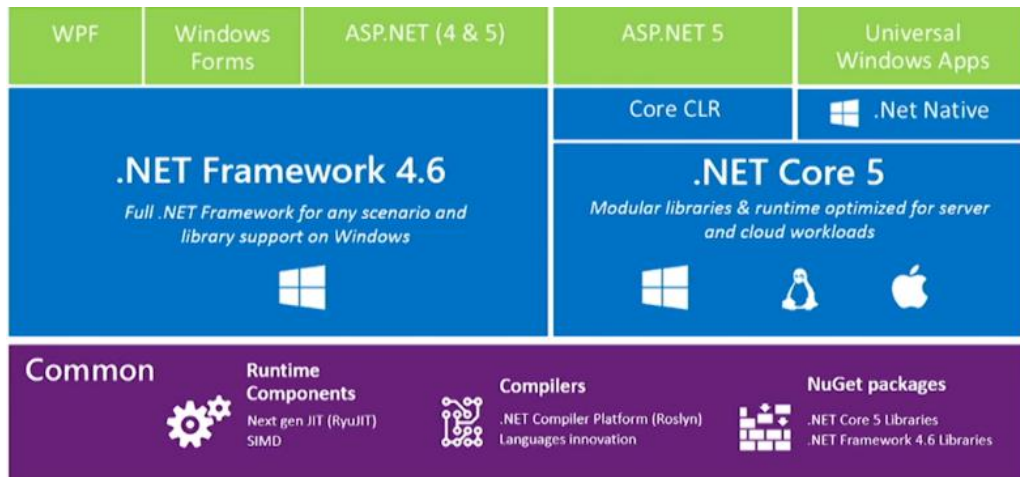
#### ☆ .NET Framework 컴파일은 2 번 일한다?

.NET Framework 의 컴파일은 다른 컴파일러와 다르게 2 번 컴파일 하게 됩니다.

첫 번째는 Microsoft .NET Framework 의 각각 언어의 컴파일러에서 컴파일을 하여 하나의 결과물을 만들어 냅니다. 이 결과물은 exe 파일 또는 dll 파일이라는 확장자를 가지고 있는 파일로 Microsoft .NET Framework 에서는 MSIL(Microsoft intermediate Language)로 만들어 집니다. 다른 말로 어셈블리 또는 닷넷 어셈블리라는 말도 하지만 정확히는 MSIL 로 컴파일 된 결과물입니다. 가끔 여기서 오래 전부터 프로그램 개발을 하신 분이라면 어셈블리 언어로 만든 결과물이 아닌가? 할 수 있습니다. 이건 절대 아닙니다. 바로 다음 두 번째 컴파일 단계에서 사용되는 결과물 입니다.

두 번째 컴파일은 MSIL 결과물을 JIT(Just In Time) Compiler 라는 녀석이 실제 실행되게 합니다. JIT 컴파일에 와서야 실제 만든 프로그램이 실행이 됩니다.





### ◇ 관리되는 코드란

관리되는 코드의 뜻은 JIT 가 실제 프로그램을 실행하게 되어도 CLR 의 역할은 계속 됩니다. CLR 은 응용 프로그램의 메모리를 관리하고, 보안을 처리하며, 언어간의 상호운용성을 지원하는 등의 지속적인 관리를 하게 됩니다. 즉 .NET Framework 환경에서 실행되는 프로그램의 결과물을 만들고 이것을 작성하는 것을 만드는 것을 코드를 작성한다 또는 "코딩을 한다"라고 합니다. 코드작성 또는 코딩이란 응용 프로그램을 개발할 때 사용하는 언어로 작성한 것을 이야기 하는 것입니다. 이런 것을 지속적으로 관리하기 때문에 관리하기 때문에 "관리되는 코드(Managed Code)"라고 부릅니다. 이와 반대로 "관리되지 않는 코드(Unmanaged Code)"라고 하면 .NET Framework 에서 실행이 되지 않는 옛날 방식 코드 작성이라 할 수 있습니다. 대표적으로 C++는 바로 이 관리되지 않는 코드라고 할 수 있으며, .NET Framework 에서는 이런 관리되지 않는 코드에 대하여 사용을 권장하지 않습니다.

### ◇ Visual C# 이란 무엇인가?

C# 언어는 .NET CLR 안에서 실행되는 응용 프로그램 개발에 사용되는 언어입니다. 이 언어는 .NET Framework 가 나올 때 나온 언어이며 C#만큼 .NET Framework 의 전용 언어라 해도 과언이 아니라 생각합니다. 물론 다른 언어도 많이 있지만 우리나라에서 C# 언어의 사용 비율이 매우 높은 편이라 할 수 있습니다.

C# 언어로 프로그램을 개발할 때 흔히 C++ 보다 못하다 하는데 이는 .NET Framework 를 조금 더 자세히 보면 도움이 되지 않을까 합니다. C++에서 할 수 있는 모든 것은 C#에서도 모두 가능합니다. 단 못하는 것은 시스템 메모리에 접근하고 조작하는 것은

C++의 고수준 기능들에 해당하기 때문에 C#에서는 권장하지 않고 만약 사용한다면 C#에서는 “안전하지 않음”이라는 것으로 표시를 하게 됩니다. 이 글이 보이면 웬지 프로그램이 이상해 질 것 같지만, 이는 프로그램 개발을 오래하시고 시스템 메모리 조작을 잘하시는 분이라는 전제 조건하에 사용하셔도 괜찮습니다. 절대 시스템 메모리 조작을 잘하시는 분이라는 전제입니다. 특히 Windows 운영체제를 자세히 알고 메모리 구조에 대하여도 잘 아시는 분이라는 전제입니다.

C#언어로 프로그램을 만들 수 있는 것은 Windows 환경에서 만들 수 있는 것은 모두 개발할 수 있습니다. 큰 분류로 구분하면 다음과 같습니다.

- Windows 응용 프로그램 : Windows 환경에서 실행되는 응용 프로그램으로 노트북이나 데스크 탑에서 실행되는 프로그램입니다.
- 태블릿 응용 프로그램 : Windows 태블릿 환경에서 실행되는 응용 프로그램으로 Windows 8.0 이상에서 사용되는 프로그램이라 할 수 있습니다.  
(참고 Modern UI 개발이라 할 수 있습니다.)
- 서버 서비스 프로그램 : Window Server 에서 Background 에서 실행되는 프로그램을 개발할 수 있습니다.
- 웹 응용 프로그램 : Windows 환경의 웹 환경에서 실행되는 응용 프로그램을 개발할 때 사용됩니다. 쉽게 홈페이지 만들 때 또는 인트라넷 프로그램 만들 때 사용됩니다.
- 웹 서비스 : 분상환경이나 웹으로 서비스를 프로그램을 만들 때 사용합니다.

#### ☆ C# 언어의 이해

프로그램 언어는 크게 두 가지로 구분할 수 있습니다. 바로 정적 / 동적 언어로 구분합니다. 우리가 알고 있는 C# 언어는 바로 정적 언어에 해당됩니다.

정적 언어의 대표는 바로 C 언어인데 C 언어를 사용하기 위해서는 사전에 배워야 할 것이 많습니다. C 언어를 배우고 C++언어를 배운 사람들이 사실 C#언어를 배우면 매우 빨리 배우는데 그 이유 중에 하나라 할 수 있듯 정적 언어는 프로그램 개발을 할 때 즉 코딩을 할 때 미리 정의를 해야 하는 것들 있기 때문입니다. 그럼 이제 C# 언어도 정적 언어이기 때문에 사전에 배워야 하는 것을 배우고 가도록 하겠습니다.

이제부터 C#의 문법이라는 것을 배우지만, 실제 다른 정적 언어의 프로그램에서도 거의 비슷한 개념이기 때문에 이 부분을 확실히 이해하고 가면 다른 언어를 배울 때 많은 도움이 됩니다.

## C# 기초 문법 1 = 데이터와 저장소

데이터란 것은 무엇인가? 또 저장소는 무엇인가?

“컴퓨터에 일을 시키고 그 결과를 화면에 보는 것” 이것이 우리가 컴퓨터에 일반적으로 요청한 거의 대부분의 일 입니다. 이 일을 완료하고 화면에 보여주기 위한 작업을 하기 위해 한가지 중요한 것이 있습니다. 바로 저장소입니다.

저장소는 화면에 표시하기 위한 데이터를 담아두기 위한 곳입니다. 이 저장소에 우리가 보고자 하는 화면에 대한 데이터가 있고 이 데이터를 화면에 표시하게 됩니다. 즉 저장소가 있고, 화면에서 표시할 때 이 데이터가 숫자인지 아니면 글씨인지 이것도 사실 중요합니다. 즉 실제 데이터가 숫자, 글씨 또는 기타 어떠한 것인지 알아야 합니다. 그리고 중요한 것은 이 저장소에 숫자를 가지고 있을 것인지, 글씨를 가지고 있을 것인지도 중요합니다. 사전에 우리가 해야 하는 작업이란 바로 이 저장소에 숫자를 저장할 것인지 아니면 글씨를 저장해야 할지를 결정해야 합니다.

이것을 우리는 앞으로 데이터 형식이라 부르겠습니다.

## 데이터 저장소 = 형식을 지정을 꼭 지정

이렇게 밑줄 그어 가면서 기억하고 있어야 합니다. 데이터를 저장할 곳은 꼭 어떤 형식으로 지정해야 문제가 발생하지 않습니다. 사실 정적 언어의 한가지 특징이라 할 수 있습니다. 즉 C 언어나 다른 정적 언어들도 대부분 데이터를 저장할 형식을 사전에 지정해야 합니다.

C#에서는 이런 데이터 형식이 크게 기본 데이터 형식과 복합 데이터형식으로 구분할 수 있습니다. 우리가 배우는 것은 먼저 기본 데이터 형식을 배우고 복합은 조금 뒤에서 배우도록 하겠습니다.

## C# 기초 문법 2 = 변수는 무엇인가요?

변수라는 말이 나오면 우선 무섭게 느껴지는 분들도 있고, 수학을 좋아하는 사람이라면 아 변수를 그냥 좋아라 할 수 있고, 또는 변수 이게 뭐지 하는 사람도 있을 수 있습니다.

이 변수라는 것은 변하는 수라고 배웁니다. 그럼 쉽게 표현하자면 "바람둥이"를 떠올리면 될 듯 합니다. 너무 부정적일 수 있지만 이 부정적인 이미지로 기억하시면 쉽게 변수를 이해 할 수 있을 것이라 생각합니다.

C# 언어의 변수는 숫자를 저장하거나 글씨를 저장하거나 또는 기타 다른 내용을 저장하는 변수들이 있습니다. 즉 형식을 저장하는 곳이라 할 수 있으며, 쉽게 내가 사는 곳이 일반 주택인지, 아니면 아파트인지, 아니면 빌라인지를 결정해야 합니다.

변수를 C#에서 사용하려면 앞에서 이야기 한 것처럼 사전에 어디에 사는지 결정해야 하기 때문에 사용하기 위한 문법은 다음과 같습니다.

### [데이터 형식] [이름];

이렇게 지정을 하는 것으로 입니다.

앞의 [형식] 은 어떤 데이터 형식인지를 지정하는 것입니다. 숫자를 저장할 지 아니면 글씨를 저장할 지를 지정하는 것입니다. 다음으로는 데이터 형식은 몇 가지 정해져 있는데 이를 표시할 이름을 지정하는 것입니다. 이런 단계를 거치는 것이 바로 "변수를 지정한다"라고 표현합니다.

정적 언어에서는 일반적으로 선언되지 않는 변수는 컴파일이 되지 않기 때문에 선언되지 않는 변수는 문제가 발생하므로 꼭 선언하고 사용해야 합니다.

## C# 기초 문법 3 = 기본적인 문법

이제 C# 언어를 배워야 하는데 변수를 선언해야 하고 사용해야 합니다. 이제 간단한 문법부터 보겠습니다.

C#의 기본적인 문법은 C 또는 C++ 프로그램 개발 경험이 있으신 분들은 쉽게 이해할 수 있습니다. 그러나 처음 C#을 접하시는 분은 어려울 수 있지만 문법이기 때문에 차분히 하나씩 하나씩 천천히 따라오는 것이 좋습니다. 이유는 바로 영어를 배울 때에는 우리는 ABCD 부터 배우기 때문에 하나씩 천천히 익숙해 질 때까지의 기다림이 필요합니다.

익숙해 지면 쉽게 C 또는 C++ 언어를 배울 때 도움이 될 것입니다.

C# 컴파일러는 다른 언어와 달리 코드의 추가적인 공백들을 그냥 무시하기 때문에 혹 프로그램 코드를 작성할 때 공백을 많이 두어도 괜찮습니다. 즉 문법에서 띄어쓰기 정도는 글씨의 가독성을 위해 적당히 공백을 두어도 됩니다.

C#은 문장(statement)으로 이루어져 있으며, 하나의 문장이 끝나면 세미콜론(semicolon ";")으로 끝냅니다. 일반적으로 마침표 역할이라 할 수 있습니다. 코드의 가독성을 위해서 한 줄에 한 문장씩을 사용하는 것이 좋습니다.

C# 언어는 코드 작성을 할 때에는 블록단위로 이루어져 있습니다. 즉 코드 블록의 일부라는 뜻으로 중괄호"{"의 시작과 끝"}"으로 둘러싸여 있습니다. 이 블록 안에 코드를 작성하는 방법을 사용합니다.

```
{  
    문장 1;  
    문장 2;  
}
```

이제 C# 언어의 기초적인 문법과 변수의 선언을 배웠으면 이제부터 실제 프로그램을 해보겠습니다. 잠시 동안 우리는 메모장을 이용하여 실제 프로그램 코드 작성을 하겠습니다.

#### **C# 기초 문법 4 = 데이터 형식과 변수의 사용**

C# 언어의 데이터 형식은 사실 Microsoft 의 .NET Framework 에서 이미 정해져 있습니다. Microsoft MSDN 웹 사이트에는 이 부분을 표와 함께 설명되어 있습니다.

그럼 Microsoft 사이트에서가 아닌 여기서 보도록 하겠습니다.

데이터 타입은 크게 3 가지로 먼저 구분하겠습니다. 이 구분 방법이 아닌 다르게 구분하셔도 됩니다.

- 숫자를 저장하는 데이터 타입
- 글씨를 저장하는 데이터 타입
- 기타 저장하는 타입

이렇게 크게 대 분류로 구분하고 자세히 알아보겠습니다.

#### ① 숫자를 저장하는 데이터 타입

숫자를 저장하는 것은 정수형 데이터와 실수형 데이터로 나눌 수 있습니다.  
그리고 조금 더 세분화 하면 정수형은 음의 수와 양의 수로 나눌 수 있습니다.



정수형은 크게 숫자 1,2,3 등을 저장하고 실수형은 소수점을 저장합니다.

정수형은 여기서 + 형식의 숫자 1,2,3 등을 저장하는 것과 - 형식의 숫자 -1,-2,-3 등을 저장하는 음수를 저장하는 공간으로 저장을 합니다.

그럼 우리가 가장 많이 사용하는 숫자는 정수형으로 음수까지 저장할 수 있는 -1,-2 등도 저장하는 "int"형식으로 저장합니다.

그 외 실수형에서는 "float" 또는 "double" 둘 중에 하나만 사용합니다. 2 개의 차이는 소수점 몇 자리까지 저장하는 가의 차이입니다.

형식	범위	크기
sbyte	-128 ~ 127	부호 있는 8비트 정수
byte	0 ~ 255	부호 없는 8비트 정수
char	U+0000 ~ U+ffff	유니코드 16비트 문자
short	-32,768 ~ 32,767	부호 있는 16비트 정수
ushort	0 ~ 65,535	부호 없는 16비트 정수
int	-2,147,483,648 ~ 2,147,483,647	부호 있는 32비트 정수
uint	0 ~ 4,294,967,295	부호 없는 32비트 정수
long	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807	부호 있는 64비트 정수
ulong	0 ~ 18,446,744,073,709,551,615	부호 없는 64비트 정수

## 정수형 타입의 형식

형식	근사 범위	전체 자릿수 (Precision)
float	$\pm 1.5e-45 \sim \pm 3.4e38$	7개의 자릿수
double	$\pm 5.0e-324 \sim \pm 1.7e308$	15-16개의 자릿수

## 실수형 타입의 형식

### ② 글씨를 저장하는 데이터 타입

글씨를 저장하는 데이터 타입은 딱 하나만 기억합니다. 문자 데이터형은 "string" 이것 하나만 기억하고 사용하면 됩니다. 이 문자형의 크기는 2 바이트 이며 범위는 0 ~ 65536 입니다. C, C++ 언어와 다르게 문자는 ASCII 가 아닌 유니코드(Unicode) 방식입니다. 이점 꼭 기억해야 합니다.

### ③ 기타 저장하는 타입과 그 외 저장 및 표현

날짜를 저장하는 DateTime 과 참과 거짓을 저장하는 Boolean 타입인 "bool" 타입이 있습니다. 그 외 decimal 타입의 경우 화폐를 표시할 때 주로 사용합니다.

Microsoft .NET Framework 에서는 다음의 표와 같이 데이터 타입 중에 기본 데이터 타입을 제공하고 있습니다.

값 형식	기본값
bool	false
byte	0
char	'\0'
decimal	0.0M
double	0.0D
enum	식 (E)0으로 계산된 값입니다. 여기서 E는 열거형 식별자입니다.
float	0.0F
int	0
long	0L
sbyte	0
short	0
struct	모든 값 형식 필드를 기본값으로 설정하고 모든 참조 형식 필드를 null로 설정하여 계산된 값입니다.
uint	0
ulong	0
ushort	0

이렇게 데이터 타입의 경우 실제 선언은 다음과 같이 합니다.

```
참조 0개
static void Main(string[] args)
{
    int a = 10;
    string b = "안녕";
}
```

여기에 보면 "[데이터 형식] [이름] = 값;" 문법을 사용하고 있습니다.



"=" 이 부분은 할당이라고 하며 선언과 동시에 값을 지정하거나 넣어야 추후에 넣어야 합니다. 만약 데이터 형식에 맞추어 변수를 선언하고 사용하지 않았다면 프로그램이 실행되면서 문제가 발생할 수 있을 수 있으니 변수를 선언하고 사용하지 않는 실수를 최대한 줄어야 합니다. 다행히 우리가 앞으로 배워야 하는 도구는 이 실수를 최대한 줄여 주는 기능을 제공하기도 합니다.

문자열에서는 주의할 것이 한가지 있습니다. 바로 문자열 리터럴이라고 하는데 바로 이스케이프 시퀀스들입니다. 이는 이스케이프 다음에 오는 것을 기준으로 프로그램이 뭔가 작동을 하게 되는데 실제 많이 사용되는 경우가 있습니다. 대표적으로 추후에 데이터베이스 처리를 배울 때 사용되거나 파일 입출력에서 파일 경로명에서도 많이 사용되므로 꼭 참고하여 알아야 합니다.

#### 이스케이프 문자

w'	작은 따옴표
w"	큰 따옴표
ww	역 슬래시
w0	널 문자
wa	삐 소리 내기
wb	백스페이스
wf	폼 피드
wn	새 줄
wr	캐리지 리턴(맨 앞으로)
wt	수평 탭
wv	수직 탭

## C# 기초 문법 5 = 변수의 명명 규칙과 관례

모든 프로그램 언어를 배울 때에는 꼭 기억해야 하는 것이 있습니다. C# 언어를 배울 때에도 마찬가지 입니다. 프로그램 개발자라면 꼭 기억하면 좋을 것을 지금부터 시작합니다.

### ① 변수 명명 규칙

변수를 지정할 때 다시 말해 변수 이름을 정할 때에는 특별히 규칙이 존재합니다. 어려운 규칙이 아니므로 기억하고 있다고 사용합니다.

- 변수 이름의 첫 번째 문자는 반드시 하나의 글자 또는 밑줄문자 또는 앳("@")기호여야 합니다.

예) aaaa / bbbb / \_asdba

- 두 번째부터는 글자, 밑줄 문자, 숫자가 올 수 있습니다.

예) total10 / total\_avg

- C# 언어는 대소문자를 구분합니다.

예) total / Total 은 서로 다른 변수 입니다.

그 외 Microsoft .NET Framework 에서 사용되는 "예약어" 또는 "키워드"라는 것이 있는데 이 키워드도 사용하면 안됩니다. 즉 특정한 단어를 변수 이름으로 사용할 수 없습니다. 키워드는 다음 과 같습니다.

abstract	as	base	bool
break	byte	case	catch
char	checked	class	const
continue	decimal	default	delegate
do	double	else	enum
event	explicit	extern	false
finally	fixed	float	for
foreach	goto	if	implicit
in	in(제네릭 한정자)	int	interface

internal	is	lock	long
namespace	new	null	object
operator	out	out(제네릭 한정자)	override
params	private	protected	public
readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc
static	string	struct	switch
this	throw	True	TRY
typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual
void	volatile	while	

그 외 컨텍스트 키워드는 특정 의미를 지정하는데 사용되지만 C#의 키워드는 아닙니다. 그래도 변수명으로는 사용할 경우 한번 생각해 봐야 할 부분입니다.

<a href="#"><u>add</u></a>	<a href="#"><u>alias</u></a>	<a href="#"><u>ascending</u></a>
<a href="#"><u>async</u></a>	<a href="#"><u>await</u></a>	<a href="#"><u>descending</u></a>
<a href="#"><u>dynamic</u></a>	<a href="#"><u>from</u></a>	<a href="#"><u>get</u></a>
<a href="#"><u>global</u></a>	<a href="#"><u>group</u></a>	<a href="#"><u>into</u></a>
<a href="#"><u>join</u></a>	<a href="#"><u>let</u></a>	<a href="#"><u>orderby</u></a>
<a href="#"><u>partial(형식)</u></a>	<a href="#"><u>partial (메서드)</u></a>	<a href="#"><u>remove</u></a>
<a href="#"><u>select</u></a>	<a href="#"><u>set</u></a>	<a href="#"><u>value</u></a>
<a href="#"><u>var</u></a>	<a href="#"><u>where(제네릭 형식 제약 조건)</u></a>	<a href="#"><u>where(쿼리 절)</u></a>
<a href="#"><u>yield</u></a>		

## ② 명명 관례

변수의 이름은 매우 자주 사용되기 때문에 처음 정할 때 신중해야 합니다. 그래서 실제 개발 프로젝트가 진행되면 변수 이름에 대한 변수 명명 규칙등을 정하여 다른 개발자들과 서로 특정한 규칙을 정하기도 합니다. 그런 규칙이 없으면 몇가지 관례를 알아두면 매우 용이하며 C#의 언어의 기본 표기법도 이 관례를 일반적으로 따르고 있습니다.

가장 유명한 명명 시스템은 헝가리식 표기법(Hungarian notation)이란 것이 있습니다. 이 표기법은 변수의 형식을 의하는 소문자 접두어를 변수 이름 앞에 붙이는 것으로 예를 들면 나이를 표시할 경우 Age 라는 단어에 숫자를 저장한다면 iAge 라는 이름으로 표시하는 방법입니다. 물론 이 표기법은 하나의 관례이며 굳이 따를 필요는 없지만 알아두면 실제 개발 프로그램의 프로젝트에 있는 소스코드를 볼 때 특정 변수의 데이터 타입을 쉽게 이해하고 사용할 수 있는 장점이 있습니다. 단점은 접두어가 너무 많을 경우 변수의 형식을 표기하는 방법이 조금 주의를 해야 합니다. 특히 많은 사람들이 있을 경우 각기 다른 표기 방법을 사용하면 혼란스러울 것이므로 헝가리식 표기법을 사용할 경우 변수 명명 규칙이라는 것을 정하여 사용하는 것이 좋습니다.

그래서 요즘 변수의 이름을 정할 경우에는 용도에 기반해서 짓는 방식이 많이 선호되고 있고 실제 Microsoft .NET Framework 에서도 실제 용도에 맞는 뜻의 영어단어로 표시하고 있습니다. 그 이유는 잠수 후 Visual Studio 라는 Tool 을 사용할 경우 더 명확한 이유를 알 수 있습니다. 그렇다면 변수 이름을 정할 때 영어 단어의 뜻에 맞게 할 경우 특정 단어를 중복해서 써야 한다면, 실제 이럴 경우에는 단어의 뜻을 "\_" (언더바)를 이용하여 뜻을 길게 하는 경우도 있습니다. Microsoft .NET Framework 에서는 너무 길지 않게 사용하여도 되기 때문에 20 자에서 30 자까지 사용해도 됩니다. 물론 더 길게 사용해 되지만 프로그램 코드의 가독성을 위해서 너무 길지 않게 변수의 이름을 정하는 것이 좋습니다.

Microsoft .NET Framework 에서는 몇가지 더 사용하는 표기법이 있는데 바로 PascalCase 와 camelCase 라는 두 가지 관례가 더 있습니다. 이 두 관례 이름의 형태 자체가 둘 사이의 차이를 보여주는데 PascalCase 의 경우 변수 이름의 첫 번째 글씨의 경우대문자로 표기하는 반면, camelCase 는 첫 글씨를 소문자로 사용합니다. 다른 표현으로 Upper Case 와 Low Case 도 있기도 합니다.

PascalCase 예) Age / LastName / FirstName

camelCase 예) age / lastname / firstname

이러한 몇몇 관례는 Microsoft 에서는 간단한 변수들에는 camelCase 로 조금 더 복잡할 경우 PascalCase 를 사용하도록 권장합니다. 권장일 뿐이지 실제 프로그램 개발 프로젝트가 진행될 경우 특히 많은 개발자들이 개발하는 프로젝트의 경우에는 변수 명명규칙을 꼭 확인하고 변수이름을 정하는 것이 좋습니다.

마지막으로 `_`“(언더바)”를 변수 이름으로 사용하는 것은 이제 권장하지 않습니다.

### ③ 주석문

모든 프로그램 언어에서 중요한 것은 바로 주석(comment)입니다. 주석이란 C# 코드가 아니므로 컴파일러에서도 이 부분은 무시하게 되는데, 이를 잘 사용하여 실제 다른 개발자들이 프로그램 코드(또는 소스 코드)를 볼 경우 그 뜻을 간략히 적어 실제 프로그램 코드나 변수의 내용을 알리는 것입니다. 다른 개발자들이 보기 위한 것도 있지만, 실제 1년이 지난 소스 코드를 보고 1년전에 선언한 변수의 용도가 무엇이며 이 프로그램에서 왜 이렇게 했는지 등 또한 누가 언제 프로그램을 처음 코드를 작성하고 수정했으며, 수정한 부분은 어느 부분인지를 알 수 있게 하는 방법이 바로 주석문이기도 합니다.

주석을 작성할 때에는 C# 언어에서는 다음과 같은 방법이 있습니다.

여러 줄 주석(또는 블록 주석) : 주석의 시작을 `/*`로 시작해서 `*/`로 끝나는 것으로 여러 줄 또는 한 줄에 그 뜻을 전달할 때 사용합니다.

한 줄 주석 : `///` 역 슬래시 2개를 다음에 오는 것은 무조건 주석문으로 인식하고 그 줄이 끝나면 더 이상 주석이 아닙니다.

특수 주석 : C#언어에서는 앞에 2 가지 방식의 주석보다 한가지가 더 있습니다. 2 가지 주석 방법은 C++에서도 사용하는 방법인데 C#에서는 `///` 역 슬래시 3 개를 사용하는 특별한 종류의 주석이 있습니다. 이 주석은 Visual Studio Tool 에서 사용하면 매우 편리하고, 실제 프로그램 코드 작성을 할 때에도 도움을 주기도 합니다. 이 주석은 Visual Studio Tool 에서 자세히 알아보겠습니다.

## C# 기초 문법 6 = 상수

앞에서 변수를 기억하십니까? 변수의 반대되는 개념이 있습니다. 바로 “상수”라는 녀석입니다. 이 상수는 변수의 반대로 한 번 데이터가 저장되면 그 코드가 끝날 때까지 변하지 않고 계속 그 값을 가지고 있는 녀석입니다. C# 언어에서는 2 가지 상수가 존재합니다. 하나는 “const”로 선언해서 사용하는 것과 다른 하나는 “readonly”라는 것이 있습니다. 원래 상수는 const 로 선언해서 사용하였는데 C# 3.0 부터 readonly 가 하나 더 추가되었습니다.

const 는 코드 작성 시작 맨 앞에(조금 전문적인 말로 필드 선언부라고도 합니다.) 선언하고 초기에 값을 무조건 넣어야 합니다. 그렇지 않다면 컴파일에서 문제가 발생합니다.

사용 예) `const 자료형 상수명 = 값;`

Readonly 는 선언만하고 값은 프로그램이 시작할 때(뒤에서 클래스와 생성자를 배우는데 바로 프로그램 시작 점을 그 때 배우게 됩니다.) 값을 넣어주면 그 값을 프로그램 끝날 때 까지 가지고 갑니다.

사용 예) `readonly 자료형 상수명;`  
`프로그램 시작점에서 자료형 상수명 = 값;`

그럼 여기서 상수는 절대 변하지 않는 값이기 때문에 보통 기준이 되는 값을 정할 때 사용을 많이 합니다. 상수 이름을 정할 때에는 보통 변수 이름을 정할 때와 같이 하지만 보통은 모두 대문자로 사용하기 때문에 Upper Case 를 사용합니다.

사용 예) `const int MAX_INT = 99999999;`

### **C# 기초 문법 7 = 열거형**

상수의 선언 방법은 변수와 비슷합니다. 데이터 형식 앞에 const 라는 키워드를 사용하는 것 다를 뿐 입니다. 그렇다면 상수를 여러 개 선언해야 한다면 어떻게 해야 할까요? 예를 들어 고정된 값을 순차적으로 가지고 있는 상수를 여러 개 사용해야 하고 이 값은 정수만 저장되어 있다면 어떻게 해야 할까요? 이런 문제를 해결하기 위하여 열거형(enum)을 사용합니다. 쉽게 말해 열거형은 상수를 순차적으로 열거한 것으로 첫 번째 요소가 0, 두 번째 요소가 1 등으로 순차적으로 저장이 되는 녀석입니다.

사용 예) `enum 열거형식명 : 기반 자료형 { 상수 1, 상수 2, .....}`

여기서 기반 자료형은 모두 정수 계열(byte, sbyte, short, ushort, int, uint, long, ulong, char) 만 가능합니다. 상수를 선언할 때 int 로 선언할 경우 그 값을 넣어주지 않으면 기본적으로 0 으로 시작하고, 중간에 값을 넣어주면 넣어준 다음부터 시작합니다.

열거형은 보통 프로그램 개발 프로젝트에서 순서대로 나열된 데이터를 표시할 경우 사용하는데 컴퓨터는 문자열 보다 숫자를 빨리 처리하기 때문에 실제 문자 대신 숫자로 문자열의 의미를 대신해서 처리하는 경우가 있습니다. 대표적으로 인사정보 시스템 개발에서 직급을 문자열로 표시하면 "대표이사, 사장 등등"으로 표시하지만 시스템

내부에서는 열거형으로 0 은 대표이사, 1 은 부사장등으로 표시하여 처리하는 경우에 미리 정의된 열거형을 사용하기도 합니다.

### **C# 기초 문법 8 = Nullable 형식**

C#에서는 데이터 형식을 정할 때에는 무조건 특정한 값이 저장되어야 합니다. 그런데 당장 저장할 것이 없고 추후에 저장하려 한다면 어떻게 될까요? 쉽게 말해 내가 집을 계약했고 그 집을 특정한 날짜에 이사를 하려고 합니다. 그렇다고 그 집이 없는 것이 아니라 새로 지은 집이라 단지 비워져 있다는 것입니다. C#에서는 사실 이것을 허용하지 않습니다. 그래서 나온 것은 Nullable 형식입니다. 즉 나중에 들어갈 거니 잠시 기다려와 비슷하다고 할 수 있습니다. 사용법은 다음과 같습니다.

선언 방법 : `int? TotalValue;`

Nullable 형식을 왜 사용할까요? 앞에 선언 방법에 있는 정수형 변수 TotalValue 에 값이 설정되지 않는 상태에 값을 할당하려면, 2 가지 방법으로 사용할 수 있습니다. 그것은 특정 값의 바로 넣어주거나 TotalValue 의 값이 없으므로 거짓을 표시할 경우가 필요할 수도 있습니다. 우리는 값이 아직 없기 때문에 비워져 있다는 뜻으로 거짓을 표시하고 싶은데 C# 초기에는 이것을 표현하기에 코드를 몇 줄을 작성해야 했다면 이제 Nullable 형식으로 선언하면 바로 한 줄 코드로 거짓을 표현 수 있습니다. 즉 Nullable 형식은 바로 두 번째 형식으로 많이 사용하는 방법이 바로 Nullable 형식입니다.

### **C# 기초 문법 9 = var 형식과 Dynamic 형식 그리고 마지막 Object**

C#언어는 정적 언어로 데이터 형식이 정해지지 않으면 매우 싫어해서 프로그램이 문제가 발생하기 때문에 컴파일러에서부터 문제가 있다고 표시합니다. 이는 프로그래머의 실수를 줄여주기도 하지만 데이터를 저장할 때 그 때 그때 형식이 정해질 수도 있습니다. 그래서 C# 3.0 에서부터 그 부분을 서서히 지원하게 됩니다.

var 이라는 익명타입(Anonymous Type)입니다. 변수를 선언할 때 데이터 타입 대신 var 이라고 바꾸어서 선언만 하면 됩니다. 어렵지 않은 사용법인데 주의 점이 있습니다.

Var 는 특정한 지역에서만 사용가능 한데 바로 그 블록 안에서만 사용이 가능합니다. 조금 더 전문적으로 이야기 하면 지역 변수로만 사용할 수 있다는 것입니다. 특정 지역을 벗어나면 사용할 수 없습니다. 그런 반대로 필드 또는 전역 변수라는 말도 있는데 이는 다른 곳에서 사용되는 변수를 말합니다.

선언 방법 : `var totalint = 100;`

`var totalstring = "합계";`

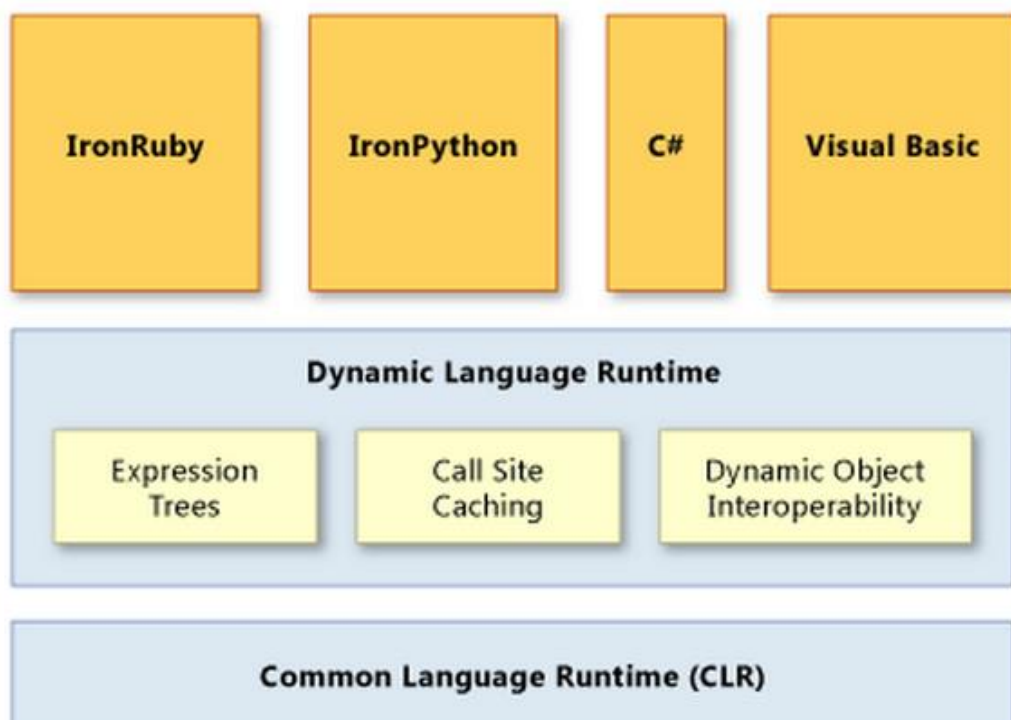
C# 4.0 부터는 다른 프로그램 언어와 호환성을 위해 하나의 형식이 더 나왔습니다. 바로 Dynamic 타입으로 이 용도는 사실 동적 언어와 관련이 있습니다. 동적 언어라 하면 값을 저장할 때 특정한 타입을 지정하지 않는 언어들로 이 언어들의 데이터를 C# 언어로 가지고 와서 사용할 때 Dynamic 타입을 사용합니다. C# 언어의 Dynamic 언어의 요소를 추가한 이유는 바로 Ruby, Python 같은 언어와 데이터를 주고 받는 상호운용성도 제공하기 위함이고도 합니다. Microsoft .NET Framework 4.0 부터 있는 DLR(Dynamic Language Runtime)을 추가함으로 이런 동적 언어의 데이터를 사용할 수 있도록 하고 있습니다.

선언 방법 : `dynamic total = 10;`

`total = "합계";`

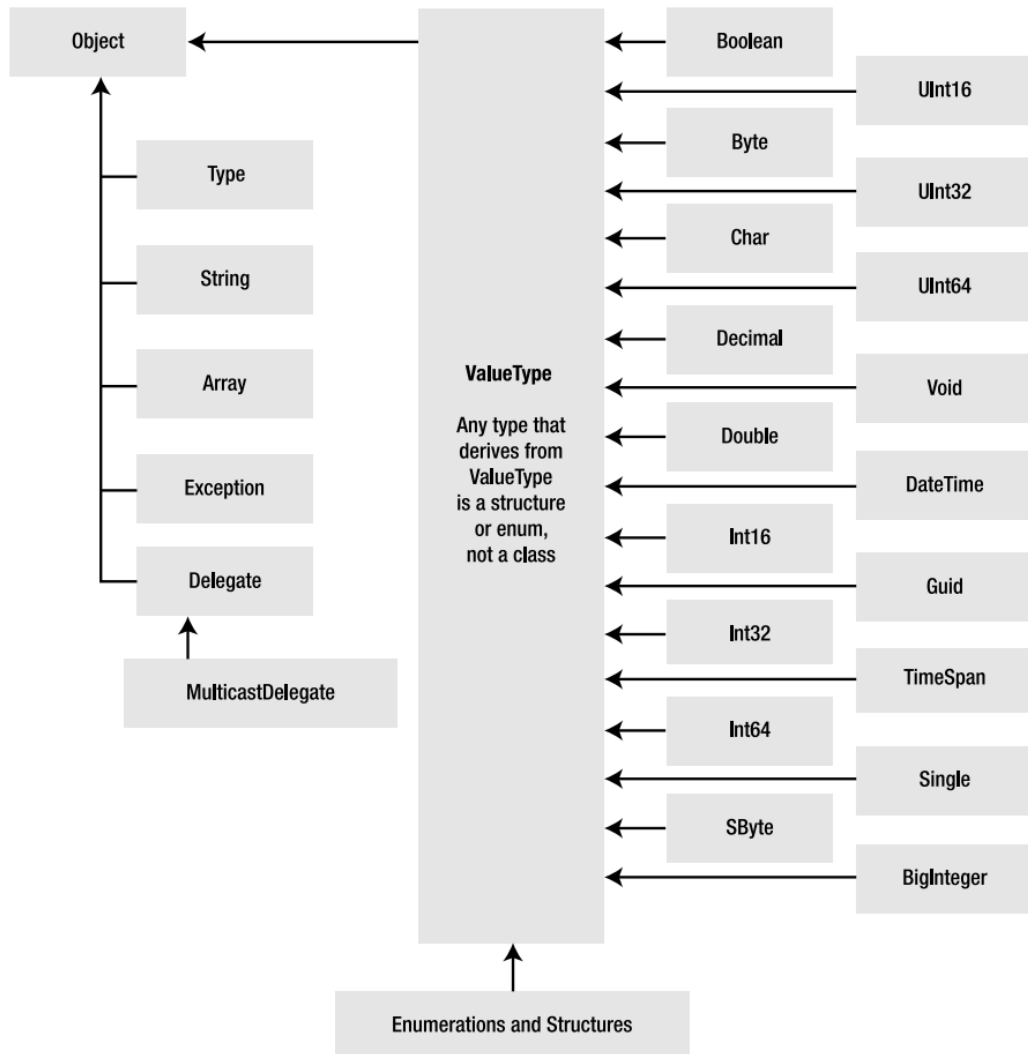
C#의 dynamic 타입은 선언할 때 데이터 타입 대신 사용하면 되고, 중간에 데이터 타입이 얼마든지 변경될 수 있습니다.

### DLR architecture





Object 타입은 모든 데이터 타입의 조상이라 할 수 있습니다. 우리가 데이터 타입을 선언할 때 object 타입으로 선언하면 Dynamic 와 비슷하게 사용할 수 있습니다. 그렇지만 object 타입을 주로 사용하는 것은 권장하지 않습니다. 이는 다음의 형 변환과 연관도 있으며 object 를 사용할 경우 메모리의 문제도 있습니다. 따라서 object 타입은 모든 데이터 타입의 조상이라는 것만 기억하고 사용을 최대한 자제 하는 것이 좋습니다.



### C# 기초 문법 10 = 형 변환

실제 프로그램에서는 데이터를 저장하고 화면에 표시할 때 문자 형태로 표시하거나 정수형 데이터를 소수점으로 변환하여 표시하는 경우가 발생합니다. 또는 날짜의 경우에는 숫자 데이터를 년도만 가지고 오거나 중간에 년월일 등을 넣어야 하는 경우도

발생합니다. 이를 위해 형 변환이라는 것을 제공합니다. 형 변환은 다음 같은 경우 사용하기도 합니다.

- 크기가 서로 다른 정수 형식
- 크기가 서로 다른 소수점 형식
- 음수나 양수 형식
- 소수점과 정수의 형식
- 문자열과 숫자 형식

이렇게 변환을 해야 하는 경우가 존재합니다. 쉬운 예로 정수 10 을 소수점이 있는 형식으로 변환 할 경우입니다. 이는 int 형식에서 float 형식으로 변환해야 하는 경우입니다. 크기가 서로 다른 정수의 경우에는 작은 자릿수에서 큰 자릿수로 표현할 경우입니다. 쉬운 예로 유튜브의 동영상 시청의 카운터가 표시할 수 있는 수의 한계가 한번 왔습니다. 더 이상 숫자를 추가할 수 없자 형 변환을 하여 표시를 하였습니다. 바로 가수 사이의 동영상입니다. 이럴 경우가 크기가 서로 다른 정수 형식의 형 변환입니다.

형 변환은 다음 같이 합니다.

```
Byte a = 127;
```

```
Int b = (int)a;
```

쉽습니다. 바로 변수 앞에 "("으로 시작해서 변환하고자 하는 데이터 타입을 쓰고 ")"로 닫고 변환 하고자 하는 변수명을 써 주면 됩니다. 그럼 소수점으로 변환할 경우에는 같은 방법으로 하면 됩니다.

그럼 문자열을 경우는 조금 다릅니다. 숫자에서 문자로 반대인 문자를 숫자로 변환 할 경우에는 되지 않습니다. 다음 그림을 참조합니다

```
int a = 10;
string b = "123";
string c = (string)a;
int d = (int)b;
Console.WriteLine(a);
```

앞으로 배울 Visual Studio 에서 확인한 것으로 형 변환이 되지 않기 때문에 빨간 줄로 표시해 준 것입니다. 이를 해결하기 위해서 우리는 "Parse"라는 키워드를 기억해야 합니다. 다음 그림을 참조합니다.

```
int a = 10;
string b = "123";
string c = string.Parse(a);
int d = int.Parse(b);
Console.WriteLine(d);
```

문자열을 숫자로 바꾸는 것은 되는데 숫자를 문자열로 바꾸는 것은 되지 않는다고 Visual Studio 라는 Tool 에서 확인해 본 것입니다. 사실 문자열로 바꿀 경우에는 다른 방법을 사용합니다. 문자열로 변경하는 것은 잠시 후 배워 보겠습니다.

그럼 이 같이 형의 변환을 할 때 사실 2 가지 형의 변환이 존재합니다. 바로 명시적 형 변환과, 묵시적(또는 암묵적) 형변환 입니다. 이는 상식적으로 작은 데이터가 큰 데이터로 바꿀 경우 형 변환을 할 때 굳이 명시해 주지 않아도 되지만 큰 데이터를 작은 데이터로 변환할 경우에는 바로 변경할 수 없습니다.

```
byte bt = 127;
int e = bt;
sbyte sb = e;
```

(지역 변수) int e

암시적으로 'int' 형식을 'sbyte' 형식으로 변환할 수 없습니다. 명시적 변환이 있습니다. 캐스트가 있는지 확인하세요.

이렇게 byte 에서 int 로 변경할 경우에는 가능하지만 반대의 경우에는 명시적으로 변환을 해야 합니다.

### C# 기초 문법 11 = 문자열과 C# 6.0 의 내삽기능(String Interpolation)

우리나라는 "년 / 월 / 일"로 표시하지만, 미국의 경우 "월 / 일 / 년" 으로 표시합니다. 그 외 유럽의 어느 나라에서는 "일 / 월 / 년" 으로 표시하기도 합니다. 그리고 앞에 형 변환에서 숫자를 문자열 바꿀 경우에는 한가지 방법이 있습니다. 바로 "ToString()"을 기억합니다. 다음을 참조합니다.

```
int a = 10;
string b = "123";
string c = a.ToString();
int d = int.Parse(b);
Console.WriteLine(d);
```

문자열로 바꿀 경우 바로 ToString() 만 있으면 그 어떤 데이터 타입이라도 문자열로 바꿔준다고 기억하면 됩니다. 그럼 ToString 다음의 "()"를 이용하여 사실 문자열의 표시를 할 수 있는데 대표적으로 년월일의 날짜 표시나 숫자의 자릿수를 표현할 때 사용합니다. 그 외에 string.Format()도 있으며, 서식 문자열에 {0}, {1} 등의 표시하고 이 순에 맞게 데이터를 넣어 해당 위치에 표시되게 할 수도 있습니다. 이를 우리는 인수라고 표현합니다. 인수는 "()" 안에 넣는 것을 뜻합니다.

C#의 현재 최신 버전인 6.0 에서는 내삽(string Interpolation)이라는 기능이 추가되었습니다. 이 기능은 {0}, {1} 등의 위치를 지정하고, 그 순서에 맞게 인수를 넣어야 하지만 C# 6.0 부터는 앞에 "\$"기호를 넣으면 순서는 상관없이 사용할 수 있습니다.

```
int r = 100;
int x = 200;
string s = $"{r} x {x} = {(r * x)}";
Console.WriteLine(s);
```

### C# 기초 문법 12 = 값과 참조 형식

앞에서 데이터 형식에 대한 부분을 배웠다면 이 데이터 형식을 저장하는 곳은 어디일까? 바로 메모리에 저장한다고 많이 이야기 합니다. 여기서 메모리에 대한 부분을 조금 만 더 깊숙이 들어가 보겠습니다.

C#에서는 2 가지 메모리 영역이 있습니다. 바로 스택(Stack)과 힙(Heap)이라는 두 가지 영역이 존재합니다. 이 두 가지가 바로 값과 참조형식으로 할 수 있습니다.

데이터 형식은 바로 스택(Stack)이라는 공간에 저장합니다. 스택에 저장하는 것을 우리는 값 형식이라고 합니다. 스택은 항아리를 생각하면 됩니다. 또는 냉장고에 음식을 넣을 때 뒤에 있는 것을 꺼내야 한다면 앞에 있는 것을 꺼내고 뒤에 있는 것을 꺼내야 합니다. 바로 스택이 그런 구조입니다. 변수를 선언하면 바로 스택에 한계씩 쌓이게 되고 이를

사용할 때에는 아래부터 차근차근 순서대로 사용한다면 위에서부터 하나씩 꺼내어서 우리가 사용하는 것을 가지고 오게 됩니다. 하나씩 순서에 맞추어서 사용하고 블록의 끝을 만나는 지점인 "}" 중괄호를 만나면 모두 사라지게 됩니다. 매우 기특하게 메모리에서 모드 제거가 된다는 것입니다.

반대로 힙은 참조 형식입니다. 값 형식은 개체를 저장하는 공간으로 앞에서 데이터 타입은 object 타입을 저장하는 것입니다. 힙 형식은 스택과 조금 다른데 우리가 데이터 타입을 선언할 때 object 로 선언하면 스택에는 값이 저장되지 않고 힙의 특정 위치에 값을 저장하고 있다는 기억만 합니다. 그리고 중괄호의 끝"}"을 만나면 바로 사라지는 것은 스택이므로 특정 위치만 기억하고 있다는 것만 기억하고 힙에 있는 것은 그대로 남게 됩니다. 이렇게 되면 힙 메모리에 데이터가 계속 있게 되고 이렇게 쌓이면 메모리가 꽉차게 됩니다. 이를 위해서 C#에서는 가비지 컬렉터라는 것이 있습니다. 이 가비지 컬렉터가 바로 힙을 청소하는 청소기라 할 수 있습니다.

이 저장소에 대한 자세한 것은 뒤 저장소에 대한 이해에서 조금 더 자세히 보겠습니다.

### **C# 기초 문법 13 = 구조체**

데이터 형식을 사용하기 위해 우리는 변수를 배웠습니다. 그럼 이 변수를 우리가 마음대로 만들 수 있으면 좋지 않을까? 생각해 보셨는지? 네 바로 사용자 정의 변수라고 생각할 수 있는 것이 바로 구조체 입니다. 앞에서 잠시 이야기한 기본 데이터 형식 외에 이게 복합 데이터 형식입니다.

어렵게 생각하지 말고 기본 데이터 형식을 모아서 하나의 데이터 형식을 만든 것이기 때문에 복합 데이터 형식이라고 생각하면 쉽게 이해 할 수 있을 것이라 생각합니다. 조금 더 전문적으로 이야기 하면 값 타입의 복합 데이터 형식입니다. 구조체는 간략한 데이터 값을 저장하는 사용하는데 적당하고 생각할 수 있습니다.

구조체의 선언 방법은 다음과 같습니다.

참조 1개

```
struct MyST
```

```
{
```

```
    public int X;
```

```
    public int Y;
```

참조 0개

```
public MyST(int x, int y)
```

```
{
```

```
    this.X = x;
```

```
    this.Y = y;
```

```
}
```

```
}
```

선언은 struct로 선언하고 블록을 사용하여 그 안에 여러 개의 데이터 타입을 지정하거나 또는 초기값을 지정하여 사용하기도 합니다. 구조체는 스택을 사용한 복합 데이터 형식이므로 간단한 데이터 처리를 위해 사용하는 것을 권장합니다.

## ❖ .NET Framework 연산자

변수의 선언을 배웠다면 이제 변수를 조작하는 것을 배우게 됩니다. 변수의 조작은 사실 초등학교에서 배운 수학 또는 중고등 학교에 배운 기초적인 수학의 내용만 알면 쉽게 배울 수 있습니다.

변수를 조작하기 위해서 필요한 것은 바로 연산자(Operator)를 배우는 것입니다. 변수와 리터럴 값들을 수학에서 배우는 연산자를 이용해서 조작하는 것을 식 또는 표현식이라고 합니다. 이런 연산자의 범위는 크게 3 가지로 나눌 수 있습니다.

- 단항 연산자(unary operator) : 하나의 피연산자에 대한 작용
  - 이항 연산자(binary operator) : 두 개의 피연산자들에 대한 작용
  - 삼항 연산자(ternary operator) : 세 개의 피연산자들에 대한 작용
- 피연산자 : 연산의 대상을 뜻합니다.

연산자 타입	연산자	예제
수식 연산자	+, -, *, /, %	<code>int a = (x + y - z) * (b / c) % d;</code>
할당 연산자	=, +=, -=, *=, /=, %=	<code>int a = 100; sum += a;</code>
증가/감소 연산자	++, --	<code>int i = 1; i++;</code>
논리 연산자	&& (And),    (Or), ! (Not)	<code>if ((a &gt; 1 &amp;&amp; b &lt; 0)    c == 1    !d)</code>
관계/비교 연산자	<, >, ==, !=, >=, <=	<code>if (a &lt;= b)</code>
비트 연산자	& (AND),   (OR), ^ (XOR)	<code>byte a=7; byte b=(a &amp; 3)   4;</code>
Shift 연산자	>>, <<	<code>int i=8; i = i &lt;&lt; 5;</code>
조건 연산자	? ?? (C# 3.0 이상만 지원)	<code>int val = (a &gt; b) ? a : b; string s = str ?? "(null)";</code>

대부분의 연산자는 이항 연산자이며 삼항 연산자는 보통 선택 연산자라 할 수 있습니다.  
단항과 이항 연산자는 보통 산술(수식) 연산자들이 있습니다.

분류	연산자	예
수식 연산자	+, -, *, /, %	$a = b + c;$
증감 연산자	++, --	$a++;$
할당 연산자	=, +=, -=, *=, /=, %=	$a += b + c;$
논리 연산자	&&,   , !	$a \&\& b$
관계 연산자	<, >, ==, !=, >=, <=	$a > b$
비트 연산자	&,  , ^	$a \wedge b$

#### ◇ 산술 연산자(수식 연산자)

산술 연산자는 보통 수식 연산자라고도 하는데 수학에서 배운 사칙 연산과 같습니다.

분류	기능	예
+	양쪽 피연산자를 서로 더함	$a + b$
-	왼쪽 피연산자에서 오른쪽 피연산자를 뺌	$a - b$
*	양쪽 피연산자를 서로 곱함	$a * b$
/	왼쪽 피연산자를 오른쪽 피연산자로 나눔	$a / b$
%	왼쪽 피연산자를 오른쪽 피연산자로 나눈 뒤의 나머지를 구함	$a \% b$

#### ◇ 증감 연산자

증감 연산자는 피연산자의 값을 1 만큼 증가 또는 감소 시키는 편리한 연산자입니다.

분류	기능	예
++ (전위 증가 연산자)	피연산자의 값을 1만큼 증가	$++a;$
-- (전위 감소 연산자)	피연산자의 값을 1만큼 감소	$--a;$
++ (후위 증가 연산자)	피연산자의 값을 1만큼 증가	$a++;$
-- (후위 감소 연산자)	피연산자의 값을 1만큼 감소	$a--;$

#### ◇ 관계연산자

관계 연산자는 두 피연산자의 관계에 대한 연산을 하는 것으로 크기와 값을 비교할 때 사용하는 연산자입니다.



분류	기능	예
<	오른쪽 피연산자가 왼쪽 피연산자보다 크면 참, 작으면 거짓	$a < b$
>	왼쪽 피연산자가 오른쪽 피연산자보다 크면 참, 작으면 거짓	$a > b$
==	왼쪽 피연산자가 오른쪽 피연산자와 같으면 참, 다르면 거짓	$a == b$
!=	왼쪽 피연산자가 오른쪽 피연산자와 다르면 참, 같으면 거짓	$a != b$
>=	왼쪽 피연산자가 오른쪽 피연산자보다 크거나 같으면 참, 작으면 거짓	$a >= b$
<=	오른쪽 피연산자가 왼쪽 피연산자보다 크거나 같으면 참, 작으면 거짓	$a <= b$

#### ✧ 할당 연산자

오른쪽 부분의 피연산자를 왼쪽 부분의 피연산자에 할당하는 것으로 여기에 산술 연산자를 이용하는 경우도 있습니다.

분류	기능	예
=	오른쪽 피연산자를 왼쪽 피연산자에 할당	$a = b;$
+=	$a += b$ 는 $a = a + b$ 와 같음	$a += b;$
-=	$a -= b$ 는 $a = a - b$ 와 같음	$a -= b;$
*=	$a *= b$ 는 $a = a * b$ 와 같음	$a *= b;$
/=	$a /= b$ 는 $a = a / b$ 와 같음	$a /= b;$
%=	$a \% = b$ 는 $a = a \% b$ 와 같음	$a \% = b;$

#### ✧ 논리 연산자

연산자 중에 관계 연산자와 비슷하지만 참과 거짓을 비교할 경우에 많이 사용합니다.

논리곱 (&&) 연산자

A	B	A && B
참	참	참
참	거짓	거짓
거짓	참	거짓
거짓	거짓	거짓

논리합 (||) 연산자

A	B	A    B
참	참	참
참	거짓	참
거짓	참	참
거짓	거짓	거짓

부정 (!) 연산자

A	!A
참	거짓
거짓	참

#### ✧ 비트 연산자

논리중에 비트 연산을 할 것으로 보통 참 거짓 연상에서도 사용됩니다.

분류	기능	예
&	두 피연산자의 대응되는 비트에 논리곱을 수행	a & b
	두 피연산자의 대응되는 비트에 논리합을 수행	a   b
^	두 피연산자의 대응되는 비트에 배타적 논리합을 수행	a ^ b

이 연산자들은 수학 시간에 배웠던 것부터 새로운 것이 있을 수 있습니다. 그리고 연산자들은 우선 순위라는 것이 있습니다. 이 우선순위는 수학을 배울 때처럼 곱하기가 먼저 그리고 더하기가 뒤에 연산을 하는 것으로 수학과 같습니다. 그러나 연산의 우선순위는 실제 개발자들이 직접 괄호 "()"를 이용하여 지정하는 것이 좋습니다. 그 이유는 실제 프로그램이 실행 되면서 가끔 이상한 연산을 하게 되는데 이럴 경우 대부분은 연산의 우선순위에 맞추어서 코드를 작성하여 가끔 의도하지 않는 값을 나오는 경우가 있습니다. 이럴 경우 프로그램의 문제 해결에 많은 시간이 필요하고 코드의 가독성 측면에서도 좋지 않습니다. 될 수 있으면 연산의 우선순위는 참고만 하고 간단한 우선 순위는 사용하는 것을 권장합니다.

다음은 연산의 우선순위 입니다.

구분	연산자
주요 연산자	x.y f(x) a[x] x++ x-- new typeof checked unchecked
단항연산자	+ - ! ~ ++x --x (T)x
곱셈, 나눗셈	* / %
덧셈, 뺄셈	+ -
쉬프트	<< >>
관계연산자와 타입연산자	< > <= >= is as
Equal, Not	== !=
논리 AND	&
논리 XOR	^
논리 OR	
조건 AND	&&
조건 OR	
삼항연산자	?:
대입연산자	= += /= %= ++ -- <<= >>= &= ^=  =

## ☆ 프로그램 제어 기술의 이해

프로그램 개발을 할 때 우리가 더 알아야 하는 부분이 있습니다. 바로 프로그램의 실행 순서와 화면에 표시해야 하는 순서 등 우리가 제어해야 하는 기술은 매우 많습니다. 이런 제어의 기술을 이제부터 알아보겠습니다.

## ☆ 조건(분기) 문

분기 또는 조건 이 두 단어 모두 같은 말로 여기서는 조건문이라고 하겠습니다.

조건문은 변수의 데이터들을 비교하거나 확인해서 조건이 맞으면 실행하거나 실행하지 못하게 하는 것입니다. 이런 조건문을 배우는 이유는 특정 조건에 맞아야만 특정한 기능을 수행하거나 화면에 표시하는 것을 배우기 위함입니다.

조건에 따라 참과 거짓 또는 크거나 같음, 또는 작다 식의 조건을 이용하여 우리가 원하는 것을 처리하도록 하는 것입니다. 조건문의 다음 같이 몇 가지가 있습니다.

### 단순 조건문

```
if (조건식) {  
    // 조건이 참일 경우 실행될 문장  
}
```

### 기본 조건문

```
if (조건식) {  
    // 참일 경우에 실행될 문장  
} else {  
    // 위의 조건식에 아무것도 해당하지 않을때 실행될 문장  
}
```

### 다중 조건문

```

if (조건식) {
    // 참일 경우에 실행될 문장
} else if (조건식) {
    // 참일 경우에 실행될 문장
} else {
    // 위의 조건식에 아무것도 해당하지 않을때 실행될 문장
}

```

3 가지 정도의 조건문이 있으며 여기에 한가지 더 블록안에 조건문을 더 줄 수 있습니다. 이는 중첩 조건문이라고 합니다. 조건문의 사용할 때 주의할 점은 복잡한 다중 또는 중첩 조건문을 최대한 자제해서 사용하는 것을 권장합니다. 그 이유는 조건문을 복잡하게 하면 프로그램의 가독성과 추후 문제가 발생할 경우 처리하게 어려움이 많습니다. 이에 조금이라도 단순하게 사용할 수 있는 방법이 있다면 단순하게 하는 것을 권장합니다. 이때 또 한가지의 주의점이라면 연산자의 우선순위를 사용하여 조건문을 만드는 것도 좋은 방법이 아닙니다. 이는 우리가 의도하지 않는 결과가 가끔 나오는 경우가 있는 대 부분 조건문에서 연산자 우선순위로 인해 가끔 이상한 결과가 나오기도 합니다. 절대 실무에서는 최대한 단순하게 조건문을 만들 것을 권장합니다.

## ✧ switch 문

switch 문은 if 구문과 조금 다른데 조건 값을 기반으로 코드를 선택으로 수행할 때 사용하는 제어 기술입니다. 주의점은 조건 값을 기준으로 한다는 것입니다.

```

switch (조건식) {
    case 상수: // 만약 조건식의 결과가 이 상수와 같다면!
        // 실행될 코드
        break; // 탈출!
    case 상수:
        // 실행될 코드
        break;
    ...
}

```

switch 문은 조건식에서 값을 기준으로 case 의 상수 값에 만족하면 그 case 구문의 코드를 실행하고 break: 구문을 만나면 끝나게 됩니다. 그리고 default 라는 것이 있어서

default 구문은 조건식의 결과값이 없다면 무조건 default 구문 안에 있는 것을 실행합니다.

switch 구문은 사실 if 구문으로도 사용할 수 있습니다. 그런데 if 구문은 switch 구문으로 사용할 수 있는 조건이 제한적입니다. 즉 if 구문은 switch 구문으로 100% 바꾸기 힘들고 반대로 switch 구문은 100% if 구문으로 바꿀 수 있습니다. 사실 실제 프로그램에서 어떻게 코드를 작성하든 프로그램이 잘 돌아가서 문제만 없다면 훌륭한 프로그램이라 할 수 있습니다. 1000 줄을 아니 10000 줄의 코드를 작성하건 문제만 없다면 괜찮습니다. 그런데 사용하다가 프로그램이 느려진다면 어떨까요? 이런 것을 사전에 막는 것 중에 하나가 사실 변수 선언과 그에 맞게 조건문을 잘 사용하는 것입니다. 지금은 하드웨어가 빠르게 발전하여 메모리나 CPU 등이 빠르지만 스마트 폰등은 아직도 하드웨어가 낮은 경우도 있습니다. 이런 경우 사실 분기와 제어를 잘못하면 프로그램이 의도하지 않는 것을 실행하거나 또는 처리가 빠르게 되지 않고 잠시 멈추었다가 실행되는 것처럼 보입니다. 조건문의 경우 switch 는 조건이 맞으면 바로 해당 case 구문 안에 있는 것을 실행하지만 if 문의 경우 차례로 순차적으로 하기 때문에 앞에 조건에 맞으면 빠르지만 그렇지 않으면 매우 느릴 수 있고, 또한 if 문의 경우는 범위에 대한 것도 비교할 수 있고, switch 구문은 조건의 값을 가지고 하기 때문에 그 특징에 맞는 조건문을 사용하는 것이 좋습니다. 지금은 편하고 빨리 코드를 작성할 수 있으면 괜찮다고 할 수 있지만 서서히 경력이 쌓아가면 기본적인 것의 용도에 맞게 구현하고 사용하는 것이 추후 문제가 없는 프로그램을 만드는 길이기도 합니다.

#### ☆ 반복문 – while

반복문은 계속해서 똑같은 일을 하는 것으로 while 구문은 특정 조건이 만족할 때 까지 계속 반복하여 실행하는 구문입니다.

```
while (조건식) {  
    // 반복 실행될 코드  
}
```

### ☆ 반복문 – do ~ while

do~while 반복은 먼저 실행하고 난 다음 조건을 검사하는 것으로 특정 조건은 나중이고 먼저 블록안에 있는 것을 한번 실행하고 난 다음 조건을 검사합니다.

이 반복문은 특정한 값을 입력받고 처리할 때 그 입력 값이 맞으면 종료하거나 할 때도 사용할 수도 있습니다.

```
do {  
    // 반복 실행될 코드  
} while (조건식); // 여기서 주의하실 것은 마지막에 세미콜론이 붙는다는 겁니다.
```

### ☆ 반복문 – for

for 반복문은 실무에서도 많이 사용되는 것으로 특정 조건이 만족할 때 까지 인데 while 구문과는 다르지만 비슷한일 합니다. 똑같은 일을 하긴 하지만 for 문과 while 구문은 결정적으로 다른 것이 있습니다. 바로 조건식 다음 증감식이란 것이 있는데 바로 끝을 알 수 있다는 것이고 while 구문은 특정 조건이 만족할 때까지 이므로 조금 다르다 할 수 있습니다. 프로그램이 실행되다가 응답 없음이 발생하는 것은 경우가 있는 그 원인 중 하나가 바로 for 구문이나 while 구문에서 반복을 너무 많이 하는 경우일 수 있습니다. 그리고 while 구문의 경우 for 구문보다 무한 반복에 빠져 프로그램이 계속 응답 없음이 될 확률이 조금 더 높은 경우도 있습니다. 그렇다고 많이 높은 것은 아니고 조금 높다고 할 수 있습니다.

그리고 while 구문의 경우 절대 for 구문으로 100% 구현하는 것도 한계가 있지만 for 구문을 while 구문으로는 쉽게 바꿀 수 있습니다. 그 이유는 for 구문은 증감식으로 인해 순차적으로 증가를 하지만 while 구문은 참/거짓으로 반복을 할 수도 있기 때문입니다. 따라서 반복문을 사용할 경우에도 용도에 맞추어서 사용하기를 권장합니다.

```
for(초기식; 조건식; 증감식) {  
    // 반복 실행될 코드  
}
```

#### ✧ 중첩 반복문 – for { for { for

if 문에서 중첩 if 문이 있다면 for 구문의 경우에도 중첩 for 반복문이 있습니다. 중첩 반복문의 경우는 절대 꼭 잘 알고 써야 합니다. 잘못하면 프로그램의 실행 속도 그리고 응답 없음이 발생할 수 있기 때문입니다.

```
for(...; ...; ...) {  
    for(...; ...; ...) {  
        ...  
    }  
}
```

#### ✧ 반복문 – foreach

foreach 반복문은 개체를 배우고 사용법을 배우는 것이 좋습니다. 그렇지만 미리 알고 뒤에 개체와 배열을 배우고 연습을 하겠습니다. foreach 구문은 특정한 개체에 대하여 순차적으로 하나씩 처리하고 마지막까지 처리하고 난 다음 종료는 하는 것입니다. 즉 데이터를 순차적으로 처리하면서 마지막까지 처리를 자동으로 하는 녀석입니다. 주로 많이 쓰이는 것은 개체 또는 뒤에서 배우는 컬렉션의 개체 또는 데이터베이스의 데이터를 가져와서 데이터를 순차적으로 처리할 때 사용을 많이 합니다.

```
foreach (변수 in 배열 혹은 컬렉션) {  
    // 실행될 코드  
}
```

#### ✧ 반복문 – break

break 문은 switch 구문에서 사용되는 반복문을 빠져 나오기 위해서 사용합니다.

#### ✧ 반복문 – continue

continue 문은 조건을 검사하는 부분으로 넘어가서 반복을 계속하라고 지시하는 구문입니다. 이 구문은 break 구문과 같이 쓰기도 합니다.



### ✧ 반복문 - goto

goto 구문은 특정 위치로 이동하는 이 위치를 레이블로 표시합니다. 즉 특정 위치인 레이블로 이동하라는 명령입니다.

```
...  
goto 레이블;  
  
...  
레이블:  
    // 실행될 코드  
...
```

## ❖ Visual Studio 와 언어 버전

지금까지 코드 작성을 위한 기본적인 내용을 알아보았고, 이제는 코드 작성을 위한 기술을 알아보겠습니다.

Microsoft 에서는 .NET Framework 를 이용한 프로그램 개발에 필요한 도구를 제공하고 있습니다. 이 도구는 사실 매우 비싼 도구이기도 하지만 개발에 필요한 기본적인 것은 사실 무료로 배포하고 있습니다. 이 도구는 개발자에게도 훌륭하지만 개발 프로젝트에서 필요한 모든 기능을 가지고 있습니다. 그런데 우리나라에서는 이 도구에 대한 활용을 100%로 활용을 못하기도 하고, 실제 프로젝트에서는 개발자들만 사용을 하고 있습니다.

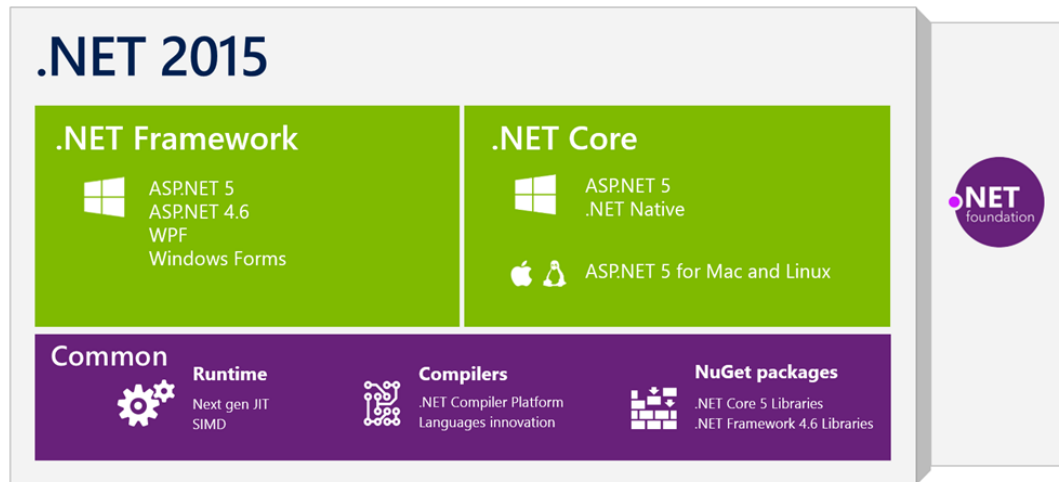
Visual Studio 라는 이 도구는 사실 개발자, 테스터, 설계자, Database 쿼리 개발자 등이 모두 사용할 수 있는 도구 입니다. 이 도구는 .NET Framework 기반에 개발에 필요한 모든 기능을 가지고 있고 최신 버전에선 스마트 폰 개발 기능이 추가 되었으며, 스마트 폰은 안드로이드, 아이폰등도 개발 할 수 있도록 하고 있습니다.

이 Visual Studio 은 아주 오래 전부터 나왔으며 다음을 참고합니다.

Product name	Codename	Internal version	Supported .NET Framework versions	Release date
Visual Studio	N/A	4.0	N/A	1995-04
Visual Studio 97	Boston	5.0	N/A	1997-02
Visual Studio 6.0	Aspen	6.0	N/A	1998-06
Visual Studio .NET (2002)	Rainier	7.0	1.0	2002-02-13
Visual Studio .NET 2003	Everett	7.1	1.1	2003-04-24
Visual Studio 2005	Whidbey	8.0	2.0, 3.0	2005-11-07
Visual Studio 2008	Orcas	9.0	2.0, 3.0, 3.5	2007-11-19
Visual Studio 2010	Dev10/Rosario	10.0	2.0, 3.0, 3.5, 4.0	2010-04-12
Visual Studio 2012	Dev11	11.0	2.0, 3.0, 3.5, 4.0, 4.5	2012-09-12
Visual Studio 2013	Dev12	12.0	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1	Upcoming

표를 참고하여 보면 .NET 은 2002 년 2 월 13 일날 발표되어 지금까지 사용되고 있으며 최신 번은 Visual Studio 2015 버전을 얼마 전에 출시 했습니다.

## The open .NET ecosystem

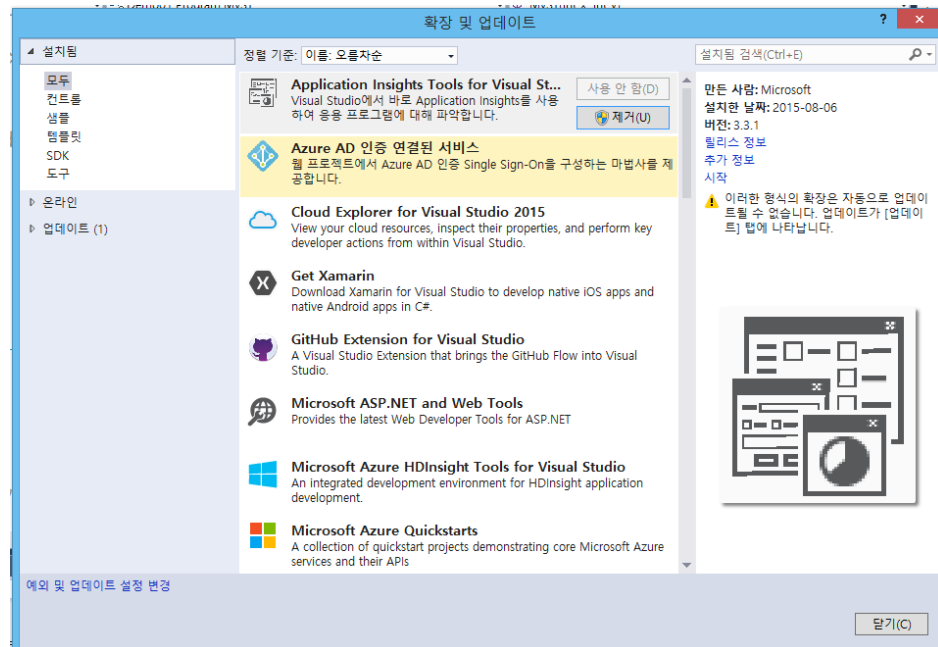


최신 .NET Framework 버전으로 앞으로 이 버전을 사용하여 실습과 연습을 하겠습니다. 사실 실무에서는 Visual Studio 2010 버전까지 사용해도 됩니다. 그 이유는 XP 환경이 남아 있다면 Visual Studio 2012 버전은 설치 및 지원이 안되기 때문입니다. 따라서 최소한 Windows Vista 부터 아니 Windows 7 이상이라면 Visual Studio 최신 버전을 사용하여 개발해도 괜찮습니다.

Visual Studio 버전	.NET	Windows
Visual Studio.NET 2002	1.0	Windows 2000
Visual Studio.NET 2003	1.1	Windows XP
Visual Studio 2005	2.0 + 3.0	Windows Vista
Visual Studio 2008	3.0	Windows XP
Visual Studio 2010	4.0	Windows 7
Visual Studio 2012	4.5	Windows 8
Visual Studio 2013	4.5.1	Windows 8.1
Visual Studio 2015	4.6	Windows 10

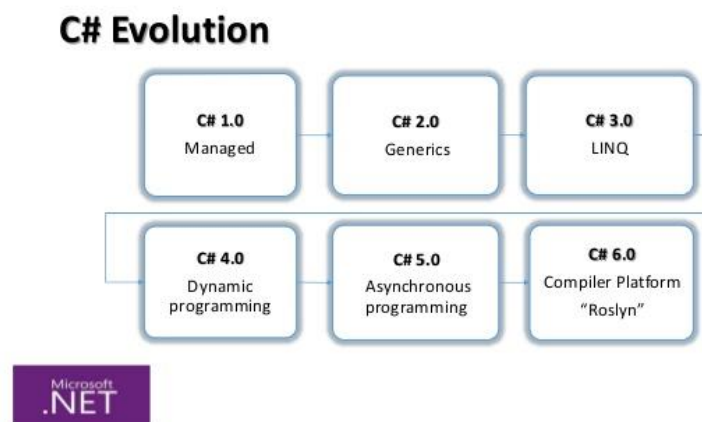
Visual Studio 은 세부적으로 개발자와 설계자들을 위한 버전으로 나누어 지는데 현재 Visual Studio Enterprise 버전은 설계 및 모든 기능을 가지고 있고 Professional 버전은 기능이 제한적인 입니다. 그렇지만 개발자들은 Professional 버전만으로 충분하고 무료 배포판의 경우에도 Professional 버전과 거의 같은 기능을 제공하므로 큰 문제 없이 .NET Framework 기반 환경에서 개발에 문제가 없을 것입니다.

Visual Studio 를 설치하면 .NET Framework SDK 는 기본적을 설치 되므로 별도의 SDK 는 설치 하지 않아도 되지만 다른 Microsoft 기반 환경이나 추가 개발을 위해서는 SDK 를 설치해야 합니다. 또한 Visual Studio 2010 부터는 기능 추가를 쉽게 할 수 있는 기능을 내장하고 있습니다.



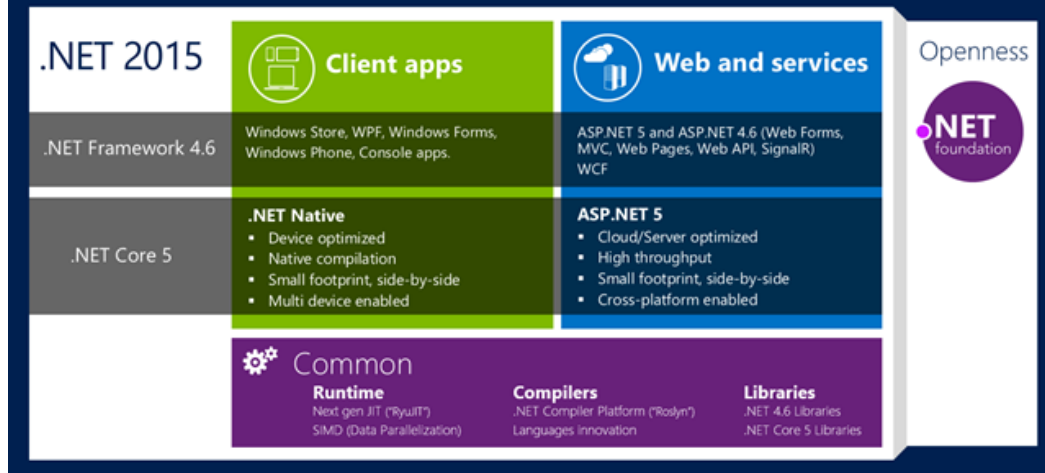
Visual Studio 를 설치하면 바로 C# 언어를 개발할 수 있는 언어를 제공하며 C# 언어를 개발할 때 그 어떤 도구보다도 개발자들에서 개발의 편의성과 문제해결을 위한 기능을 제공합니다.

C# 언어는 다음의 표를 참조합니다.



C#은 6.0 이 최신버전이며 .NET 2015 에는 다음과 같습니다.

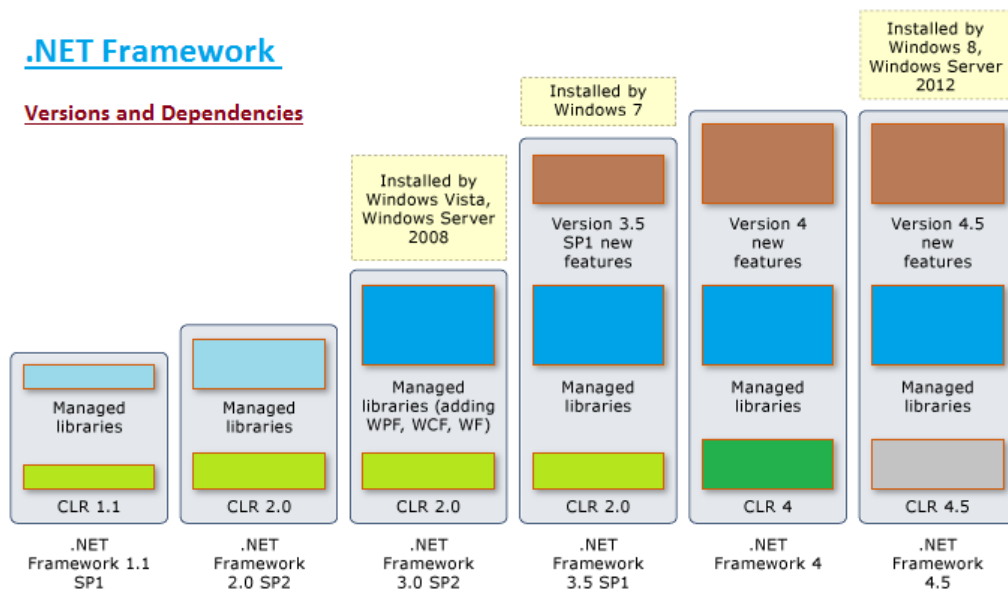
# Future of .NET (Scenario oriented)



이를 OS 와 같이 보면 다음과 같습니다.

## .NET Framework

### Versions and Dependencies



C#과 .NET Framework 는 항상 같이 발전하고 그리고 최신의 Windows 환경 기반의 개발을 제공하기 있습니다. 앞으로 새로운 OS 와 새로운 Visual Studio 는 어떻게 바뀌는 지는 항상 Microsoft 의 Road Map 을 확인해야 합니다.

그렇지만 Visual Studio 는 개발자에 있어서 항상 좋은 개발 환경을 제공하고 Windows OS 환경에서 실행되는 프로그램을 만들 때 최고의 파트너이기도 합니다.

### 3) Windows OS 와 스레드 기술의 이해

이제부터 기초를 넘어 조금 어려운 부분으로 넘어가겠습니다.

Windows OS 즉 우리가 사용하는 Windows 운영체제에 대하여 정확히 이해하고 프로그래밍을 작성하면 그에 따른 여러가지 장점이 있습니다.

우선 Windows 운영체제를 이해가 전에 간단히 운영체제에 대하여 이해하고 Windows 운영체제에 순차적으로 논의하겠습니다. 그럼 우선 Windows OS 는 멀티 태스킹이라는 용어를 사용합니다. 다시 말해 여러가지 일을 동시에 처리한다고 합니다. 정말 그럴까요? 우리가 사용하는 Windows 운영체제는 멀티 태스킹 시스템이며 실제 일을 동시에 실행시킨다는 뜻이지만 실제로는 그렇지 않습니다. CPU 가 동시에 여러 개의 프로세스를 실행하는 것이 아니라는 뜻입니다. 정확히는 시분할 방식(Time-Division)이라는 방식으로 짧은 시간을 쪼개고 쪼개서 일정한 단위시간을 기본으로 하나씩 실행하는 형태입니다. CPU 가 돌아가면서 여러 개의 프로그램을 하나씩 실행하면서 우리가 알수 없을 정도로 짧은 시간 단위로 프로그램을 하나씩 실행하면서 마치 동시에 여러 개의 프로그램을 실행시키는 것처럼 화면에 표시를 하므로 우리는 당연히 여러 개의 프로그램을 동시에 실행하는 것처럼 알고 있는 것입니다.

그렇다면 스레드를 이해하기 앞서 왜 멀티 태스킹이라는 것인지는 잠시 후 이야기 하고, 이제 스레드를 이야기 하겠습니다.

스레드의 정체는 무엇인가? 사전적인 의미를 한번 보면 프로그램을 실행할 때 그 프로그램 내에서 실행되는 흐름의 단위를 뜻한다고 표시합니다. 쉽게 접근해보면 우리가 하나의 프로그램을 실행을 하면 그 프로그램을 실행되는 하나의 흐름 그 자체를 스레드라 할 수 있습니다. 즉 하나의 프로그램에 하나의 스레드를 가지고 있습니다. 요즘 Windows 에서는 Microsfot Word 프로그램을 실행하고, 문서를 작성하면서 다시 Microsoft Excel 를 실행할 수 있지만, 옛날의 Dos 라는 운영체제 시절에는 Microsoft Word 를 실행하고 문서를 작성하면서 동시에 Microsoft Excel 를 실행하지 못하였습니다. 즉 하나의 프로그램에 하나의 스레드를 주어서 실행했다고 이해를 하면 우선 쉽게 스레드에 접근할 수 있습니다. 그렇다면 요즘 우리가 사용하는 Windows 는 스레드 기술에 멀티라는 것을 붙여서 동시에 여러 개의 프로그램을 스레드에 생성하여 관리하고 실행하게 하는 기술을 가지고 있는 것입니다.

여기서도 잠시 그렇다면 멀티 프로세서라는 이야기도 들어봤을 것입니다.

멀티 프로세서는 가 프로세서에 독립적으로 실행되며 각각의 메모리를 별도로 차지하고 처리하는 것이고 스레드 중에 멀티 스레드는 프로세스의 내 메모리를 공유해 사용할 수 있습니다. 또한 프로세서 간의 전환 속도보다 스레드간의 전화 속도가 빠르기 때문에 우리가 프로그램을 동시에 실행한다고 하여도 스레드를 이용하여 순차적으로 실행되며 이를 빠른 주기로 반복하여 실행하기 때문에 우리는 동시에 실행된다고 이해하게 됩니다.

멀티 스레드의 장점은 CPU 가 여러 개일 경우에 각각의 CPU 가 스레드 하나씩을 담당하는 방법으로 속도를 높일 수 있는 장점이 있으며, 이러한 시스템에는 여러 스레드가 실제 시간상으로 동시에 수행될 수도 있습니다. 요즘 같은 멀티 코어일 경우 이 스레드를 하나의 코어에 하나씩 주어 실행도 가능하는 뜻입니다.

멀티 스레드의 단점은 각각의 스레드 중에 어떤 것이 먼저 실행될지 그 순서를 알 수 없다는 것입니다. 이 단점은 Windows OS 에서는 우선 순위를 이용하여 처리하는데 이 순위가 잘못되거나 이 순위를 기억 못할 경우 특정 프로그램이 중간에 멈추거나 할 경우 응용 프로그램 응답 없음이 발생 할 수 있습니다.

## ❖ 스레드의 종류

### 사용자 스레드

- 사용자 스레드는 개발자들이 만든 프로그램이 실행되는 스레드 영역이라 할 수 있습니다. 실제 우리가 만든 프로그램이 이 스레드를 사용하는 것입니다.

### 커널 스레드

- 시스템에 운영되는 즉 Windows OS 가 사용하는 영역으로 생각하고 이 커널 스레드는 함부로 접근하거나 이 영역을 잘못 사용할 경우 블루스크린 또는 운영체제에 문제를 야기 시킬 수 있습니다.

#### ✧ 스레드의 데이터 처리

스레드도 프로세스 처리를 할 때 처럼 하나의 실행 흐름으로 실행이 되기 때문에 데이터 처리에 필요한 몇가지 데이터가 필요합니다. 이 데이터는 자신만의 고유 스레드 ID, 프로그램 카운터, 레지스터 집합, 스택이 있습니다. 코드, 데이터, 파일등은 기타 자원들은 프로세스의 내의 다른 스레드와 공유하여 처리합니다.

스레드 역시 특정 데이터가 필요한 경우도 있기 때문에 멀티 스레드를 처리할 경우 이 개별 스레드만의 데이터 자료공간이 필요할 수 있습니다. 이를 TSD(Thread-Specific Data: 스레드 특정 데이터)라고 합니다.

## 4) .NET Framework 메모리 관리와 코드 작성

.NET Framework에서는 공통 언어 실행(CLR) 실행 환경고, 개발자들이 이 런타임 환경에서 실행되는 프로그램을 만드는 것입니다.

메모리 관리나 디버깅, 오류 처리들은 언어에 특정 자원이라는 것이 일반적인 의미이지만 CLR의 C는 일반'Common'을 뜻하는 것을 생각하고 .NET에서 CLR에서 실행된다는 것은 특정 .NET 언어들만 공통적으로 실행되는 환경이라는 것입니다. 즉 Windows OS 위에 .NET Framework 위에 이제 개발자들이 만든 프로그램이 실행된다는 뜻입니다.

CLR은 코드를 실행하고 관리하며, 여기서 코드는 관리 코드라고 합니다. 즉 Managed Code라고 하며 CLR에서 실행되지 않는 코드를 Unmanaged Code라고 관리되지 않는 코드라 합니다. 이런 관리형 코드는 메모리 관리, 공통 오류처리, 디버깅, 언어간의 지원, 버전, 배포, 보안, 프로파일링과 같은 CLR의 모든 서비스에 적용됩니다.

.NET Framework에서는 메모리 관리를 하기 전에 힙이라는 저장소 공간을 이해해야 합니다.

자동 메모리 관리는 관리되는 실행을 수행하는 중에 공용 언어 런타임에서 제공되는 서비스 중 하나입니다. 공용 언어 런타임의 가비지 수집기는 응용 프로그램의 메모리 할당과 해제를 관리합니다. 즉, 관리되는 응용 프로그램을 개발할 때 개발자는 메모리 관리 작업을 수행하기 위해 코드를 작성할 필요가 없습니다. 자동 메모리 관리를 사용하면 실수로 개체 비우기를 수행하지 않거나 메모리 누수를 유발하거나 또는 이미 비워진 개체를 찾기 위해 메모리에 액세스하려는 경우 등의 일반적인 문제를 해결할 수



있습니다.이 단원에서는 가비지 수집기에서 메모리를 할당하고 해제하는 방법에 대해 설명합니다.

## 메모리 할당

---

사용자가 새 프로세스를 시작하면 런타임에서는 인접한 주소 공간 영역을 이 프로세스에 예약합니다.이 예약된 주소 공간을 관리되는 힙이라고 합니다.관리되는 힙에서는 힙에 있는 다음 개체가 할당될 주소의 포인터를 관리합니다.초기에 이 포인터는 관리되는 힙의 기본 주소로 설정됩니다.모든 참조 형식은 관리되는 힙에 할당됩니다.응용 프로그램에서 참조 형식을 처음으로 만드는 경우, 이 참조 형식에 대한 메모리는 관리되는 힙의 기본 주소로 할당됩니다.응용 프로그램에서 두 번째 개체를 만드는 경우, 가비지 수집기는 첫 번째 개체와 바로 인접한 주소 공간에 두 번째 개체의 메모리를 할당합니다.주소 공간을 사용할 수 있다면 가비지 수집기는 이런 방식으로 계속해서 새 개체에 주소 공간을 할당합니다.

관리되는 힙에서 메모리를 할당하면 관리되지 않는 힙에서 메모리를 할당하는 것보다 속도가 더 빠릅니다.런타임에서는 포인터에 값을 더하여 개체에 메모리를 할당하기 때문에, 스택에서 메모리를 할당하는 속도만큼 빠릅니다.또한, 연속으로 할당된 새

개체는 관리되는 힙에 인접하여 저장되므로 응용 프로그램에서 개체에 상당히 빠른 속도로 액세스할 수 있습니다.

## 메모리 해제

---

가비지 수집기의 최적화 엔진은 수행 중인 할당에 따라 수집을 수행하기에 가장 적합한 시간을 결정합니다.수집을 수행할 때 가비지 수집기는 응용 프로그램에서 더 이상 사용되지 않는 개체에 대한 메모리를 해제합니다.가비지 수집기는 응용 프로그램의 루트를 검사하여 더 이상 사용되지 않는 개체를 결정합니다.모든 응용 프로그램에는 여러 개의 루트가 있습니다.각 루트는 관리되는 힙에 있는 개체를 참조하거나 Null 로 설정됩니다.응용 프로그램 루트에는 전역 및 정적 개체 포인터, 스레드 스택의 지역 변수 및 참조 개체 매개 변수, 및 CPU 레지스터가 들어 있습니다.가비지 수집기는 JIT(Just-In-Time) 컴파일러 및 런타임에서 관리되는 활성 루트 목록에 액세스할 수 있습니다.가비지 수집기는 이 목록을 사용하여 응용 프로그램 루트를 검사하고 이 과정에서 그래프를 만드는데, 이 그래프에는 루트에서 연결할 수 있는 모든 개체가 포함되어 있습니다.

그래프에 없는 개체는 응용 프로그램 루트에서 연결할 수 없습니다.가비지 수집기는 연결할 수 없는 개체를 가비지로 간주하고 이 개체에 할당된 메모리를 해제할

것입니다.수집을 수행할 때 가비지 수집기는 연결할 수 없는 개체에서 사용되는 주소 공간 블록을 찾기 위해 관리되는 힙을 검사합니다.연결할 수 없는 개체가 발견되면 가비지 수집기는 메모리 복사 기능을 사용하여 메모리에서 연결할 수 있는 개체를 압축합니다. 그러면 연결할 수 없는 개체에 할당된 주소 공간 블록이 해제됩니다.연결할 수 있는 개체의 메모리가 압축되면 가비지 수집기는 포인터의 위치를 적절하게 수정합니다. 그러면 응용 프로그램 루트는 개체의 새 위치를 가리킬 수 있습니다.또한 가비지 수집기는 관리되는 힙의 포인터 위치를 연결할 수 있는 마지막 개체 다음에 지정합니다.수집을 수행하는 동안 연결할 수 없는 개체의 수가 엄청나게 발견되는 경우에만 메모리를 압축한다는 점에 주목합니다.수집을 수행한 후에도 관리되는 힙에서 모든 개체가 그대로 남아 있다면 메모리 압축을 수행할 필요가 없습니다.

런타임에서는 성능 향상을 위해 큰 개체의 메모리를 별도의 힙에 할당합니다.그러면 가비지 수집기는 큰 개체에 할당된 메모리를 자동으로 해제합니다.하지만 메모리에서 큰 개체가 이동하는 것을 피하기 위해 이 메모리는 압축하지 않습니다.

세대 및 성능

-----

가비지 수집기의 성능을 최적화하기 위해 관리되는 힙은 0 세대, 1 세대 및 2 세대의 3 개 세대로 나뉩니다.런타임의 가비지 수집 알고리즘은 컴퓨터 소프트웨어 업계의 가비지 수집 체계 실험을 통해 밝혀진 일반 사실을 기반으로 하고 있습니다.첫째로, 가비지 수집기는 관리되는 전체 힙보다 관리되는 일부 힙에서 더 빠르게 메모리를 압축할 수 있습니다.둘째로, 개체가 새로울수록 수명은 더 짧아지고 개체가 오래될수록 수명은 더 길어집니다.마지막으로, 새로운 개체일수록 서로 연결되는 경향이 있어서 응용 프로그램에서 거의 동시에 액세스됩니다.

런타임 가비지 수집기는 새 개체를 0 세대에 저장합니다. 응용 프로그램 수명의 초기에 만들어져 수집 후에도 남아 있는 개체는 수준이 올라가 1 세대와 2 세대에 저장됩니다.개체 승격 과정은 이 단원의 마지막 부분에서 설명하고 있습니다.전체 힙보다 일부 관리되는 힙을 압축하는 것이 더 빠르기 때문에 가비지 수집기는 수집을 수행할 때마다 이 체계를 사용하여 전체 관리되는 힙에서 메모리를 해제하는 대신 특정 세대에서 메모리를 해제할 수 있습니다.

실제로 가비지 수집기는 세대 0 이 가득 차면 수집을 수행합니다.응용 프로그램에서 새 개체를 만들려고 할 때 세대 0 이 가득 찬 경우, 가비지 수집기에서는 개체 할당할 주소 공간이 세대 0 에 존재하지 않음을 감지합니다.그러면 가비지 수집기는 이 개체를 위해 세대 0 에서 주소 공간을 비우기 위해 수집을 수행합니다.가비지 수집기는 관리되는 힙에서 모든 개체를 검사하는 대신 먼저 세대 0 에서 개체를 검사합니다.이 방식은 가장 효율적인 방식입니다. 왜냐하면 새 개체의 수명은 짧은 성향이 있으며 수집이 수행되면 세대 0 의 상당수 개체는 더 이상 응용 프로그램에서 사용되지 않을 것이기 때문입니다.또한, 세대 0 에만 수집을 수행하더라도 응용 프로그램에서 새 개체를 계속 만들 수 있는 충분한 메모리를 확보하기도 합니다.

가비지 수집기는 세대 0 에서 수집을 수행한 후 이 단원의 메모리 해제에서 설명된 것처럼 연결할 수 있는 개체의 메모리를 압축합니다.그런 다음 가비지 수집기는 이 개체를 승격시켜서 이 부분의 관리되는 힙을 세대 1 로 간주합니다.수집 후에도 존재하는 개체는 수명이 긴 성향이 있기 때문에, 이런 개체를 상위 세대로 승격시키는 것이 당연합니다.이 결과 가비지 수집기에서 세대 0 수집을 수행할 때마다 세대 1 과 2 에서 개체를 다시 검사할 필요가 없습니다.

가비지 수집기는 세대 0에 처음으로 수집을 수행한 다음 연결할 수 있는 개체들을 세대 1로 승격시키지만, 아직도 관리되는 힙에 남아있는 개체들은 세대 0으로 간주합니다. 가비지 수집기는 세대 0이 가득 차거나 다른 수집을 수행할 필요가 있을 때까지 세대 0에서 새 개체에 대한 메모리 할당을 계속합니다. 이 시점에서 가비지 수집기의 최적화 엔진은 이전 세대에서 개체를 검사할 필요가 있는지를 결정합니다. 예를 들어, 세대 0에서 수집을 수행했음에도 불구하고 응용 프로그램에서 새 개체를 성공적으로 만드는 데 필요한 충분한 메모리를 확보할 수 없는 경우, 가비지 수집기는 세대 1에서 세대 2의 순서로 수집을 수행할 수 있습니다. 여기에서도 충분한 메모리를 확보할 수 없다면, 가비지 수집기는 세대 2, 1, 0의 순서로 수집을 수행할 수 있습니다. 각 수집이 완료되면 가비지 수집기는 세대 0에 있는 연결할 수 있는 개체를 압축하여 세대 1로 승격시킵니다. 또한, 수집이 완료된 후에 세대 1에 존재하는 개체를 세대 2로 승격시킵니다. 가비지 수집기는 세 가지 세대만을 지원합니다. 따라서 수집이 완료된 후에 세대 2에 존재하는 개체는 상위 세대로 승격시킬 없으므로, 다음 수집에서 연결할 수 없는 개체로 결정될 때까지 세대 2에 보관됩니다.

관리되지 않는 리소스의 메모리 할당 해제

---

가비지 수집기는 사용자 응용 프로그램에서 만들어지는 대부분의 개체에 대해 메모리 관리 작업을 자동으로 수행할 수 있습니다. 하지만 관리되지 않는 리소스의 경우는 명시적으로 정리할 필요가 있습니다. 가장 일반적인 형태의 관리되지 않는 리소스로는 파일 핸들, 창 핸들 또는 네트워크 연결 등의 운영 체제 리소스를 래핑하는 개체를 들 수 있습니다. 가비지 수집기에서는 관리되지 않는 리소스를 캡슐화하는 데 사용되는 관리되는 개체의 수명을 추적할 수 있지만, 리소스 정리 방법에 대한 구체적인 정보는 알 수 없습니다. 관리되지 않는 리소스를 캡슐화해 주는 개체를 만드는 경우 관리되지 않는 리소스를 정리하는 데 필요한 코드를 공용 Dispose 메서드에 제공하는 것이 좋습니다. 개체 사용자는 Dispose 메서드를 사용하여 메모리 할당을 명시적으로 해제할 수 있습니다. 관리되지 않는 리소스를 캡슐화해 주는 개체를 사용하는 경우 사용자는 Dispose 메서드를 알아 두고 필요한 경우 이 메서드를 호출해야 합니다.

메모리 관리 작업에 대한 경험은 개발 환경에 따라 다를 수 있으므로 상황에 따라서는 공용 언어 런타임에서 제공하는 자동 메모리 관리 기능에 맞춰 프로그래밍 방식을 변경해야 할 수도 있습니다.

#### ❖ 개발자에 따른 메모리 관리

##### COM 개발자

COM 개발자는 참조 횟수 구현 작업을 수동으로 하는 데 익숙합니다. 개체가 참조될 때마다 카운터가 증가합니다. 개체 참조 횟수가 범위를 벗어나면 카운터가 감소합니다. 개체의 참조 횟수가 0 이 되면 개체의 사용이 종료되고 해당 메모리가 해제됩니다.

참조 횟수 체계는 많은 버그의 원인입니다. 참조 횟수 규칙을 정확하게 준수하지 않으면 개체가 중간에 해제되거나 참조되지 않는 개체가 메모리에 축적될 수 있습니다. 순환 참조도 버그의 원인이 될 수 있습니다. 순환 참조는 자식 개체와 부모 개체에서 서로를 참조하는 경우에 발생하며, 이러한 시나리오에서는 두 개체 모두 해제되거나 소멸되지

않습니다. 이 문제를 해결하는 한 가지 방법은 부모와 자식 개체 간에 사용 및 소멸에 대한 고정된 패턴(예: 부모 개체에서 자식 개체를 먼저 삭제)을 지정하는 것입니다.

CLR 를 대상으로 하는 언어로 응용 프로그램을 개발할 때 런타임의 가비지 수집기를 사용하면 참조 횟수를 사용할 필요가 없으므로 수동 메모리 관리 체계에서 발생할 수 있는 버그가 생기지 않습니다.

### C++ 개발자

C++ 개발자는 수동 메모리 관리와 관련된 작업에 익숙해져 있습니다. C++에서 new 연산자를 사용하여 개체에 대한 메모리를 할당하면 delete 연산자를 사용하여 해당 개체의 메모리를 해제해야 합니다. 이러한 이유로, 개체를 해제하는 것을 잊어 버리면 메모리 누수가 발생하거나 이미 해제된 개체의 메모리에 대해 액세스를 시도하는 등의 오류가 발생할 수 있습니다.

Visual C++ 또는 CLR 를 대상으로 하는 기타 언어를 사용하여 응용 프로그램을 개발하면 사용자가 delete 연산자를 사용하여 개체를 해제할 필요 없이 개체가 응용 프로그램에 더 이상 사용되지 않을 때 가비지 수집기가 해당 개체를 자동으로 해제합니다.

수명이 짧은 개체의 메모리를 수동으로 관리하는 데는 메모리가 많이 소요되기 때문에 C++ 개발자는 이러한 개체를 되도록 사용하지 않았습니다. 수명이 짧은 개체에 대해 메모리를 할당하고 해제하는 작업에는 적은 양의 메모리가 소모되는데, .NET Framework 의 가비지 수집기는 실제로, 수명이 짧은 개체를 관리하기 위해 최적화되었습니다. 그렇기 때문에, 관리되는 응용 프로그램을 개발할 때 코드를 단순화하기 위해 수명이 짧은 개체를 적절하게 사용하는 것이 좋습니다.

### Visual Basic 개발자

Visual Basic 개발자들은 일반적으로 자동 메모리 관리에 익숙합니다. 이전에 사용하던 프로그래밍 방식은 .NET Framework 에서 만드는 대부분의 관리되는 개체에 적용됩니다.

그러나 관리되지 않는 리소스를 캡슐화할 때 사용하는 Dispose 메서드에 대해서는 권장되는 디자인 패턴을 사용하는 것이 좋습니다.

.NET Framework에서는 여기에서 설명하는 것보다 더 많은 CLR 대상 언어를 지원합니다. .NET Framework 가비지 수집기는 관리되는 모든 언어에 대해 자동 메모리 관리 기능을 제공합니다. 가비지 수집기는 관리되는 개체에 대한 메모리를 할당하거나 해제하며 관리되지 않는 리소스를 정리하기 위해, 필요에 따라 Finalize 메서드 및 소멸자를 실행합니다. 자동 메모리 관리를 사용하면 수동 메모리 관리 체계를 사용했을 때 발생하는 일반적인 버그를 방지할 수 있어 개발 작업을 단순화할 수 있습니다

#### ❖ 관리되지 않는 메모리 관리

사용자 응용 프로그램에서 만들어지는 대부분의 개체에 대해 .NET Framework의 가비지 수집기를 사용하여 메모리 관리를 처리할 수 있습니다. 그러나 관리되지 않는 리소스를 포함하는 개체를 만든 경우에는 응용 프로그램에서 해당 개체의 사용을 마치면 이러한 리소스를 명시적으로 해제해야 합니다. 가장 일반적인 형태의 관리되지 않는 리소스로는 파일, 창, 네트워크 연결 또는 데이터베이스 연결 등의 운영 체제 리소스를 래핑하는 개체를 들 수 있습니다. 가비지 수집기에서는 관리되지 않는 리소스를 캡슐화하는 개체의 수명을 추적할 수 있지만, 관리되지 않는 리소스를 해제하고 정리하는 방법에 대해서는 알 수 없습니다.

형식이 관리되지 않는 리소스를 사용하는 경우 다음을 수행해야 합니다.

- ❖ 삭제 패턴을 구현합니다. 이를 수행하려면 관리되지 않는 리소스의 명확한 해제를 활성화하기 위해 IDisposable.Dispose 구현을 제공해야 합니다. 개체 및 해당 개체에서 사용하는 리소스가 더 이상 필요하지 않은 경우 형식의 소비자가 Dispose를 호출합니다. Dispose 메서드가 관리되지 않는 리소스를 즉시 해제합니다.



- ✧ 형식의 소비자가 실수로 Dispose 를 호출하지 않은 경우 해제되는 관리되지 않는 리소스를 제공합니다.여기에는 두 가지 방법이 있습니다.

SafeHandle 을 사용하여 관리되지 않는 리소스를 래핑합니다.이것이 권장되는 방법입니다.SafeHandle 은 System.Runtime.InteropServices.SafeHandle 클래스에서 파생되며, 강력한 Finalize 메서드를 포함합니다.SafeHandle 을 사용하면 IDisposable 인터페이스를 간단하게 구현하고 IDisposable.Dispose 구현에서 SafeHandle 의 Dispose 메서드를 호출할 수 있습니다. Dispose 메서드가 호출되지 않는 경우 가비지 수집기에 의해 SafeHandle 의 종료자가 자동으로 호출됩니다.

Object.Finalize 메서드를 재정의합니다.종료를 사용하면 형식의 소비자가 명확하게 삭제할 수 있도록 IDisposable.Dispose 를 호출하지 못한 경우 관리되지 않는 리소스가 명확하지 않게 해제됩니다.그러나 개체 종료에 복잡하고 오류가 발생하기 쉬운 작업일 수 있기 때문에 고유한 종료자를 제공하는 대신 SafeHandle 을 사용하는 것이 좋습니다.

그러면 형식의 소비자가 IDisposable.Dispose 구현을 직접 호출하여 관리되지 않는 리소스가 사용하는 메모리를 확보할 수 있습니다. Dispose 메서드를 제대로 구현하면 SafeHandle 의 Finalize 메서드 또는 Object.Finalize 메서드의 고유한 재정의가 Dispose 메서드가 호출되지 않는 경우 리소스를 정리하는 보호 기능이 됩니다.

## 5) Windows OS 의 내부

앞에서 이야기 했듯이 Windows OS 는 멀티 태스킹을 하며, 개발자들이 만든 프로그램 중에 CLR 에서 실행되는 것은 .NET Framework 에서 관리를 하게 됩니다.

그럼 Windows OS 가 실제 부팅하는 과정일 이해하고 부팅하고 난 다음 .NET Framework 가 실행되기 까지를 보겠습니다.

1. 부트 프로그램은 먼저 CPU 이상 유무를 테스트한 후 POST(Power On Self-Test) 작업을 수행하기 위한 기본적인 테스트를 수행한다. 만약 테스트 결과가 ROM BIOS 에 저장된 값과 일치하면 POST 작업을 수행한다.
2. POST 의 첫 번째 단계으로 CPU 는 System Bus 가 정상적으로 동작하는지 테스트하기 위해 System Bus 로 특정 시그널을 보낸다. 테스트가 이상이 없다면 다음 단계로 넘어간다.
3. 다음 단계로 RTC(Real-Time Clock; or system clock)을 테스트한다. RTC 는 시스템의 전기적 신호를 동기화하기 위한 클럭으로 CMOS 를 구성하는 장치에 칩 형태로 존재한다. RTC 테스트가 이상이 없다면 다음 단계로 넘어간다.
4. 다음 단계로 시스템의 비디오 구성요소들(비디오 메모리 등)을 테스트한다. 이 과정이 완료된 후에야 비로소 표준출력을 이용해 부팅과정에서 정의된 출력을 볼 수 있다.
5. 다음 단계로 RAM 을 테스트한다. RAM 을 테스트 하는 장면은 컴퓨터 부팅 과정에서 모니터 화면을 유심히 본 사람이라면 누구나 한번쯤 보았을 것이다. 현재 RAM 에는 ROM BIOS 와 비디오 BIOS 에서 읽어들이는 데이터가 존재할 것이다. 따라서 해당 데이터가 정상적인지 테스트를 하게 된다.
6. 다음 단계로 키보드가 정상적으로 연결되어 있는지 혹은 눌려진 키가 없는지 테스트 한 후 이상이 없으면 다음 과정으로 넘어간다. 부팅 과정에서 키보드의 키가 눌러져 있는 경우 빼익~!@ 하는 연속적인 비프음 소리를 들어본 경험이 있을 것이다. 또한 POST 과정에 키보드에 대한 테스트 과정이 포함되기 때문에 키보드가 연결되지 않은 시스템은 POST 과정을 완료하지 못하고 부팅되지 않는 것을 경험해 봤을 것이다.
7. 다음 단계로 시스템에 연결된 모든 드라이브(플로피, CD, 하드디스크 등)에 신호를 보내 정상적으로 동작하는지를 테스트한다.
8. 다음 단계로 앞서 수행한 POST 의 결과가 CMOS(RTC/NVRAM)에 저장된 값과 일치하는지 검사한다.
9. 다음 단계로 SCSI BIOS 와 같은 추가적인 BIOS 를 RAM 에 로드하고 Plug and Play 를 실행하여 운영체제 로드를 위한 기본적인 구성을 RAM 에 로드한다.
10. 다음 단계로 부트 프로그램은 운영체제를 로드하기 위해 인식한 드라이브 내에서 첫 번째 섹터를 읽는다. 드라이브의 첫 번째 섹터에는

MBR(Master Boot Record)이 위치한다. MBR 섹터의 마지막 2 바이트는 정상적인 MBR 을 알려주는 시그니처 "0x55AA" 값을 가진다. MBR 의 앞부분 446 바이트의 16 비트 부트 코드를 수행하다가 오류가 발생하면 적절한 오류메시지를 출력한다.

11. 다음 단계로 **MBR 에서 부팅 가능한 파티션을 찾는 작업**을 수행한다. MBR 의 오프셋(offset) 446~509 까지 64 바이트가 파티션의 정보를 나타내는데 각 파티션은 16 바이트를 사용한다. 따라서 기본적으로 4 개의 파티션에 대한 정보 저장이 가능하다. 각 파티션의 첫 번째 바이트는 부팅 가능한 파티션인지를 나타낸다. 값이 0x80 을 가지면 부팅 가능하고 0x00 이면 부팅 가능한 파티션이 아니다. 만약 파티션이 부팅 가능하다면 해당 파티션의 시작 위치로 이동한다. 이동하게 되면 MBR 과 유사한 형태의 첫 번째 섹터가 나온다. 이 섹터를 VBR(Volume Boot Record)라고 한다. 이때 부터는 운영체제에서 정의된 부팅 과정이 수행된다. 앞서 파티션이라고 언급했는데 볼륨 부트 레코드라고 하는 것에 다소 오해가 있을지 모르겠지만 일반적으로 볼륨 당 하나의 운영체제의 부팅이 가능하기 때문에 VBR 이라고 한다.
12. VBR 에 클러스터크기, MFT 위치, 전체 섹터 등 해당 볼륨의 추가적인 정보 외에도 부팅에 필요한 시스템 파일의 위치와 실행할 수 있는 코드가 포함되어 있다. 이러한 코드는 NT Loader(NTLDR)에 의해 로드되어 실행된다. 먼저 NTLDR 은 부팅 옵션 및 부팅 메뉴가 정의되어 있는 BOOT.INI 파일을 로드한다. 이후 윈도우 이외의 다른 운영체제와 듀얼 부팅을 한다면 BOOTSEC.DOC 파일을 로드한다. 또한 SCSI 드라이브가 장착되어 있다면 해당 드라이브 실행을 위한 NTBOOTDD.SYS 파일을 로드한다.
13. 이후 NTLDR 은 NTDETECT.COM 을 로드하여 설치된 하드웨어와 관련 구성 파일들을 찾아 실행하도록 한다.
14. NTDETECT 에 의해 수행된 결과는 NTOSKRNL.EXE(NT OS KERNEL)에 적용된다. 이후 NTOSKRNL.EXE 은 커널(Kernel), HAL(Hardware Abstraction Layer), 시스템 레지스트리 정보를 로드한다.

15. 다음으로 TCP/IP 와 관련된 네트워크 드라이버들을 로드하고 로그인 화면을 보여준다. 사용자가 로그인에 성공하면 사용자에게 대한 레지스트리 정보를 가져와 사용자 환경을 구성한다.
16. 로그인 과정에서 Plug and Play 에 의해 새로운 장치가 발견된다면 DRIVER.CAB 파일에서 관련 드라이버를 로드하여 해당 장치를 마운트시킨다. 관련 드라이버가 없다면 드라이버 설정하는 다이얼로그를 보여주게 된다.
17. 이러한 과정이 지나면 윈도우의 쉘인 explore.exe 가 실행된 화면을 보게 된다.

Windows OS 에 따라 조금씩 과정이 다르지만 이 대부분 비슷하게 실행이 됩니다.

마지막에 우리가 보는 화면은 Explorer.exe 를 실행하는 것입니다. 그렇다면 이제 .NET Framework 는 어디에 위치를 할지 한번 봅시다. 요즘 Windows OS 는 실제 그래픽 카드 또는 프린터 드라이버까지 .NET Framework 로 만들기 때문에 13 번과 14 번 사이쯤이 맞을 것 같습니다. 이는 Windows OS 마다 조금씩 다를 수 있습니다.

Windows OS 에서는 커널 모드 위에 .NET Framework 가 실행되기 때문에 실제 우리는 이러한 부분을 몰라도 되지만 Windows OS 위에 .NET Framework 가 있다고 알고 있어야 합니다.

## 6) .NET Framework 와 Thread 처리기술

운영 체제에서는 프로세스를 사용하여 실행 중인 다양한 응용 프로그램을 구분합니다. 스레드는 운영 체제에서 프로세서 시간을 할당하는 기본 단위로 두 개 이상의 스레드가 해당 프로세스 내에서 코드를 실행할 수 있습니다. 각 스레드에서는 예외 처리기와 스케줄링 우선 순위, 시스템에서 일정을 잡을 때까지 스레드 컨텍스트를 저장하는 데 사용되는 구조 집합을 유지 관리합니다. 스레드 컨텍스트는 스레드의 CPU 레지스터와 스택 집합을 비롯하여 실행을 다시 시작하기 위해 스레드에서 필요한 모든 정보를 스레드 호스트 프로세스의 주소 공간에 포함합니다.

.NET Framework 는 System.AppDomain 으로 표시되며 응용 프로그램 도메인이라는 간단한 관리되는 하위 프로세스로 운영 체제 프로세스를 분할합니다.

`System.Threading.Thread`로 표시되는 하나 이상의 관리되는 스레드는 동일한 관리되는 프로세스 내의 하나 이상의 응용 프로그램 도메인에서 실행될 수 있습니다. 각 응용 프로그램 도메인은 단일 스레드로 시작되지만 해당 응용 프로그램 도메인의 코드는 추가 응용 프로그램 도메인과 추가 스레드를 만들 수 있습니다. 따라서 관리되는 스레드가 동일한 관리되는 프로세스 내의 응용 프로그램 도메인 사이를 자유롭게 이동할 수 있습니다. 한 스레드만 여러 응용 프로그램 도메인 사이를 이동하는 경우도 있습니다.

선점 멀티태스킹을 지원하는 운영 체제는 여러 프로세스에서 여러 스레드를 동시에 실행하는 효과를 냅니다. 각 스레드에 프로세서 시간 간격을 차례로 할당하여 프로세서 시간을 필요로 하는 스레드에 사용 가능한 프로세서 시간을 분배하면 가능합니다. 현재 실행 중인 스레드는 해당 시간 간격이 경과되면 일시 중단되고 다른 스레드의 실행이 다시 시작됩니다. 시스템이 한 스레드에서 다른 스레드로 전환될 때 선점 스레드의 스레드 컨텍스트를 저장하고 스레드 큐에 저장된 다음 스레드의 스레드 컨텍스트를 다시 로드합니다.

시간 간격의 길이는 운영 체제와 프로세서에 따라 달라집니다. 각 시간 간격이 짧기 때문에 프로세서가 한 개만 있더라도 여러 스레드가 동시에 실행되는 것처럼 보입니다. 실제로 다중 프로세서 시스템의 경우 실행 가능한 스레드가 사용 가능한 프로세서에 분배됩니다.

### **다중 스레드를 사용해야 하는 경우**

---

사용자 상호 작용이 필요한 소프트웨어는 다양한 사용자 경험을 제공하기 위해 최대한 빨리 사용자 동작에 반응해야 합니다. 동시에 사용자에게 데이터를 표시하는 데 필요한 계산을 최대한 빨리 수행해야 합니다. 응용 프로그램에서 실행 스레드를 한 개만 사용할 경우 ASP.NET 을 사용하여 만든 `.NET Framework Remoting` 이나 `XML Web services` 를 비동기 프로그래밍과 결합하여 다른 컴퓨터의 처리 시간을 함께 사용함으로써 사용자에 대한 응답성을 향상시키고 응용 프로그램의 데이터 처리 시간을 줄일 수 있습니다. 프로세서 사용이 많은 입출력 작업을 수행할 때 I/O 완료 포트를 사용하여 응용 프로그램의 응답성을 향상시킬 수 있습니다.

### **다중 스레드의 장점**

---

두 개 이상의 스레드를 사용하는 것은 사용자에 대한 응답성을 향상시키고 동시에 작업을 완료시키기 위해 필요한 데이터를 처리할 수 있는 강력한 기술입니다. 프로세서가 한 개인 컴퓨터의 경우, 다중 스레드에서 사용자 이벤트 사이의 짧은 시간 간격을

이용하여 백그라운드에서 데이터를 처리하면 이러한 효과를 낼 수 있습니다. 예를 들어, 사용자가 스프레드시트를 편집하는 동안 다른 스레드에서 해당 스프레드시트의 다른 부분을 다시 계산할 수 있습니다.

응용 프로그램을 수정하지 않고 두 개 이상의 프로세서를 사용하는 컴퓨터에서 실행하더라도 사용자 만족도가 훨씬 향상됩니다. 단일 응용 프로그램 도메인에서 다중 스레드를 사용하여 다음 작업을 수행할 수 있습니다.

- 네트워크를 통해 웹 서버 및 데이터베이스와 통신
- 시간이 오래 걸리는 작업 수행
- 우선 순위가 다른 작업 구분. 예를 들어, 우선 순위가 높은 스레드는 시간 위험 작업을 관리하고 우선 순위가 낮은 스레드는 다른 작업을 수행합니다.
- 백그라운드 작업에 시간을 할당하면서 사용자 인터페이스의 응답성

### 다중 스레드의 단점

---

가능한 스레드를 적게 사용하여 운영 체제 리소스 사용을 최소화하고 성능을 향상시키는 것이 좋습니다. 스레딩에는 응용 프로그램을 디자인할 때 고려해야 할 리소스 요구 사항과 잠재적 충돌이 있습니다. 리소스 요구 사항은 다음과 같습니다.

- 프로세스, **AppDomain** 개체 및 스레드에서 필요한 컨텍스트 정보는 시스템 메모리에 저장됩니다. 따라서 만들 수 있는 프로세스, **AppDomain** 개체 및 스레드의 수는 사용 가능한 메모리에 의해 제한됩니다.
- 많은 수의 스레드를 관리하려면 상당한 프로세서 시간이 소비됩니다. 스레드가 너무 많으면 대부분의 스레드에서 작업이 크게 진행되지 못합니다. 현재 스레드가 대부분 한 프로세스에 있으면 다른 프로세스에 있는 스레드의 일정은 자주 잡히지 않습니다.
- 많은 스레드로 코드 실행을 제어하면 복잡하고 버그가 많이 발생할 수 있습니다.
- 스레드를 삭제하려면 삭제 후 발생하게 될 문제와 그 처리 방법에 대해 알아야 합니다.

리소스에 공유 액세스를 제공하면 충돌이 생길 수 있습니다. 충돌을 피하려면 공유 리소스에 대한 액세스를 제어하거나 동기화해야 합니다. 같거나 다른 응용 프로그램 도메인에서 액세스를 제대로 동기화하지 못하면, 교착 상태 또는 경쟁 상태와 같은 문제가 발생할 수 있습니다. 교착 상태에 빠지면 두 개의 스레드가 서로 완료되기를 기다리면서 모두 응답을 멈추게 되고, 경쟁 상태에 빠지면 두 이벤트의 타이밍에 대한 예기치 못한 심각한 종속성으로 인해 비정상적인 결과가 발생할 수 있습니다. 시스템에서

다중 스레드 간의 리소스 공유를 조정하는 데 사용되는 동기화 개체를 제공합니다. 스레드 수를 줄이면 리소스를 동기화하기가 더 쉽습니다.

동기화가 필요한 리소스는 다음과 같습니다.

- 시스템 리소스(예: 통신 포트)
- 여러 프로세스에서 공유하는 리소스(예: 파일 처리)
- 다중 스레드에서 액세스하는 단일 응용 프로그램 도메인의 리소스(예: 전역, 정적, 인스턴스 필드)

## 스레딩 및 응용 프로그램 디자인

---

일반적으로 ThreadPool 클래스를 사용하면 다른 스레드를 차단하지 않고 작업 스케줄링이 필요 없는 간단한 작업에 대해 여러 스레드를 매우 쉽게 처리할 수 있습니다. 하지만 다음과 같이 사용자 고유의 스레드를 만들어야 하는 여러 가지 이유가 있습니다.

- 작업에 특정 우선 순위를 지정해야 할 경우
- 실행 시간이 오래 걸려서 다른 작업을 차단해야 할 경우
- 모든 **ThreadPool** 스레드는 다중 스레드 아파트에 있는데 일부 스레드를 단일 스레드 아파트에 두어야 하는 경우
- 스레드에 안정적인 ID가 필요할 경우. 예를 들어, 스레드를 중단하거나 일시 중단, 이름으로 검색할 경우 전용 스레드를 사용해야 합니다.
- 사용자 인터페이스와 상호 작용하는 백그라운드 스레드를 실행해야 하는 경우 .NET Framework 버전 2.0에서는 이벤트를 사용하여 사용자 인터페이스 스레드로 크로스 스레드 마샬링과 통신하는 BackgroundWorker 구성 요소를 제공합니다.

## 스레딩 및 예외 사항

---

스레드에서는 예외가 처리됩니다. 일반적으로 백그라운드 스레드를 포함한 스레드의 처리되지 않은 예외로 인해 프로세스가 종료됩니다. 다음은 이 규칙의 세 가지 예외 사항입니다.

- Abort가 호출되었으므로 스레드에서 ThreadAbortException이 throw 됩니다.
- 응용 프로그램 도메인이 언로드 중이므로 AppDomainUnloadedException이 throw 됩니다.

- 공용 언어 런타임 또는 호스트 프로세스에서 스레드를 종료합니다.

## ❖ 스레드 기술의 사용

운영 체제 프로세스가 만들어지면 운영 체제에서 원본 응용 프로그램 도메인을 포함하여 해당 프로세스에서 코드를 실행할 스레드를 삽입합니다. 이때부터 반드시 운영 체제 스레드를 만들거나 삭제하지 않아도 응용 프로그램 도메인을 만들고 삭제할 수 있습니다. 실행 중인 코드가 관리 코드인 경우 Thread 형식의 정적 CurrentThread 속성을 검색하여 현재 응용 프로그램 도메인에서 실행 중인 스레드의 Thread 개체를 가져올 수 있습니다. 이 항목에서는 스레드 만들기를 설명하고 데이터를 스레드 프로시저로 전달하는 대체 방법을 설명합니다.

### 스레드 만들기

---

새 Thread 개체를 만들면 관리되는 스레드가 새로 만들어집니다. Thread 클래스에 ThreadStart 대리자 또는 ParameterizedThreadStart 대리자를 사용하는 생성자가 있습니다. 대리자는 Start 메서드를 호출할 때 새 스레드가 호출하는 메서드를 래핑합니다. Start 를 두 번 이상 호출하면 ThreadStateException 이 throw 됩니다.

Start 메서드는 새 스레드가 실제로 시작되기 전에 즉시 반환되는 경우가 종종 있습니다. ThreadState 및 IsAlive 속성을 사용하여 언제든지 스레드의 상태를 결정할 수 있지만 스레드의 활동을 동기화하는 데 이러한 속성을 사용하면 안 됩니다.

### 데이터를 스레드에 전달 및 스레드에서 데이터 검색

---

.NET Framework 버전 2.0 에서 ParameterizedThreadStart 대리자는 Thread.Start 메서드 오버로드를 호출할 때 데이터를 포함하는 개체를 스레드에 간단하게 전달할 수 있는 방법을 제공합니다. 코드 예제를 보려면 ParameterizedThreadStart 를 참조하십시오.

Thread.Start 메서드 오버로드에서 모든 개체를 허용하기 때문에 ParameterizedThreadStart 대리자를 사용하는 것은 데이터를 전달하는 데 있어 형식이 안전한 방법이 아닙니다. 대안은 스레드 프로시저 및 도우미 클래스의 데이터를



캡슐화하고 `ThreadStart` 대리자를 사용하여 스레드 프로시저를 실행하는 것입니다.이 방법은 다음 두 가지 코드 예제에서 보여 줍니다.

데이터를 비동기 호출로부터 반환할 장소가 없기 때문에 이러한 대리자 모두에 반환 값이 없습니다.스레드 메서드의 결과를 검색하려면 두 번째 코드 예제에서와 같이 콜백 메서드를 사용합니다.

### **콜백 메서드로 데이터 검색**

---

다음 예제에서는 스레드에서 데이터를 검색하는 콜백 메서드를 보여 줍니다.또한 데이터 및 스레드 메서드를 포함하는 클래스의 생성자는 콜백 메서드를 나타내는 대리자를 받아들이므로 스레드 메서드가 종료되기 전에 콜백 대리자를 호출합니다.

## .NET Framework 동기/비동기 프로그래밍

### 1) System.Threading 네임스페이스

















System.Threading 네임스페이스는 다중 스레드 프로그래밍을 사용할 수 있도록 하는 형식을 포함합니다. 하위 네임스페이스는 동시 및 비동기 코드를 작성하는 작업을 단순화하는 형식을 제공합니다.

#### 네임스페이스

네임스페이스	설명
<u>System.Threading</u>	<u>System.Threading</u> 네임스페이스는 다중 스레드 프로그래밍을 할 수 있는 클래스와 인터페이스를 제공합니다. 이 네임스페이스는 스레드 작업 및 데이터 액세스를 동기화하기 위한 클래스( <u>Mutex</u> , <u>Monitor</u> , <u>Interlocked</u> , <u>AutoResetEvent</u> 등) 외에 시스템에서 제공하는 스레드 풀을 사용할 수 있도록 하는 <u>ThreadPool</u> 클래스와 스레드 풀 스레드에 대해 콜백 메서드를 실행하는 <u>Timer</u> 클래스도 포함합니다.
<u>System.Threading.Tasks</u>	<b>System.Threading.Tasks</b> 네임스페이스는 동시 및 비동기 코드를 작성하는 작업을 단순화하는 형식을 제공합니다. 주요 형식은 대기하고 취소될 수 있는 비동기 작업을 나타내는 <u>System.Threading.Tasks.Task</u> 와 값을 반환할 수 있는 작업인 <u>System.Threading.Tasks.Task&lt;TResult&gt;</u> 입니다. <u>System.Threading.Tasks.TaskFactory</u> 클래스는 작업을 만들고 시작하는 정적 메서드를 제공하고, <u>System.Threading.Tasks.TaskScheduler</u> 클래스는 기본 스레드 예약 인프라를 제공합니다.

#### ❖ System.Threading 클래스

System.Threading 네임스페이스는 다중 스레드 프로그래밍 가능 클래스 및 인터페이스를 제공합니다. 스레드 작업 및 데이터에 대한 액세스를 동기화하기 위한 클래스 외에도 (Mutex, Monitor, Interlocked, AutoResetEvent, 등), 이 네임스페이스에 포함 한 ThreadPool 시스템에서 제공 하는 스레드 풀을 사용할 수 있는 클래스 및 Timer 스레드 풀 스레드에서 콜백 메서드를 실행 하는 클래스입니다.


클래스	설명
 <a href="#"><u>AbandonedMutexException</u></a>	가져오면 throw 되는 예외는 Mutex 해제 하지 않고 종료 하 여 다른 스레드가 중단는 개체입니다.
 <a href="#"><u>AsyncLocal&lt;T&gt;</u></a>	비동기 메서드와 같은 지정된 비동기 제어 흐름에 로컬인 앰비언트 데이터를 나타냅니다.
 <a href="#"><u>AutoResetEvent</u></a>	대기 중인 스레드에 이벤트가 발생했음을 알립니다. 이 클래스는 상속될 수 없습니다.
 <a href="#"><u>Barrier</u></a>	여러 작업이 여러 단계에 걸쳐 특정 알고리즘에서 병렬로 함께 작동할 수 있도록 합니다.
 <a href="#"><u>BarrierPostPhaseException</u></a>	경우에 throw 되는 예외 사후 단계 작업의 한 <a href="#"><u>Barrier</u></a> 실패
 <a href="#"><u>CancellationTokenSource</u></a>	취소되도록 <a href="#"><u>CancellationToken</u></a> 에 신호를 보냅니다.
 <a href="#"><u>CompressedStack</u></a>	설정 및 압축 된 현재 스레드의 스택 캡처하기 위한 메서드를 제공 합니다. 이 클래스는 상속될 수 없습니다.
 <a href="#"><u>CountdownEvent</u></a>	수가 0 에 도달하는 경우 신호를 받는 동기화 기본 형식을 나타냅니다.
 <a href="#"><u>EventWaitHandle</u></a>	Represents a 스레드 동기화 이벤트입니다.
 <a href="#"><u>ExecutionContext</u></a>	현재 스레드의 실행 컨텍스트를 관리합니다. 이 클래스는 상속될 수 없습니다.
 <a href="#"><u>HostExecutionContext</u></a>	캡슐화 하 고 스레드 간에 호스트 실행 컨텍스트를 전파 합니다.
 <a href="#"><u>HostExecutionContextManager</u></a>	공용 언어 런타임 호스트 흐름 또는 실행 컨텍스트의 마이그레이션에 참여 하도록 허용 하는 기능을 제공 합니다.
 <a href="#"><u>Interlocked</u></a>	다중 스레드에서 공유하는 변수에 대한 원자 단위 연산을 제공합니다.
 <a href="#"><u>LazyInitializer</u></a>	초기화 지연 루틴을 제공합니다.
 <a href="#"><u>LockRecursionException</u></a>	잠금에 대한 재귀 정책과 맞지 않는 방식으로 잠금을 재귀적으로 시작할 때 throw 되는 예외입니다.
 <a href="#"><u>ManualResetEvent</u></a>	하나 이상의 대기 중인 스레드에 이벤트가 발생했음을 알립니다. 이 클래스는 상속될 수 없습니다.

 ManualResetEventSlim

슬림 다운 버전을 제공 ManualResetEvent 합니다.

 Monitor

개체에 대한 액세스를 동기화하는 메커니즘을 제공합니다.

 Mutex

프로세스 간 동기화에 사용할 수도 있는 동기화 기본 형식입니다.


 Overlapped

Win32의 관리되는 표현을 제공

**OVERLAPPED** 구조에서 정보를 전송하는 메서드를 포함하는 Overlapped 인스턴스는 NativeOverlapped 구조입니다.

 ReaderWriterLock

단일 작성기 및 다중 판독기를 지원하는 잠금을 정의합니다.


 ReaderWriterLockSlim

여러 스레드에서 읽을 수 있도록 허용하거나 쓰기를 위한 단독 액세스를 허용하여 리소스에 대한 액세스를 관리하는 데 사용되는 잠금을 나타냅니다.

 RegisteredWaitHandle

호출하는 경우 등록된 핸들을 나타내는

RegisterWaitForSingleObject 합니다. 이 클래스는 상속될 수 없습니다.

 Semaphore


리소스 또는 리소스 풀에 동시에 액세스할 수 있는 스레드 수를 제한합니다.

 SemaphoreFullException


카운트가 이미 최대값에 도달한 세마포에 대해 Semaphore.Release 메서드를 호출한 경우 throw되는 예외입니다.

 SemaphoreSlim

리소스 또는 리소스 풀에 동시에 액세스할 수 있는 스레드 수를 제한하는 Semaphore 대신 사용할 수 있는 간단한 클래스를 나타냅니다.

 SynchronizationContext

다양한 동기화 모델에서 동기화 컨텍스트를 전파하기 위한 기본 기능을 제공합니다.

 SynchronizationLockException














메서드가 지정된 Monitor에 대해 잠금을 소유하도록 호출자에게 요구하지만 해당 잠금을 소유하지 않는 호출자가 해당 메서드를 호출할 때 throw되는 예외입니다.

 Thread

스레드를 만들고 제어하며, 해당 속성을 설정하고, 상태를 가져옵니다.


 ThreadAbortException








Abort 메서드를 호출할 때 throw되는 예외입니다. 이 클래스는 상속될 수 없습니다.

 <a href="#"><u>ThreadExceptionEventArgs</u></a>	<a href="#"><u>ThreadException</u></a> 이벤트에 대한 데이터를 제공합니다.
 <a href="#"><u>ThreadInterruptedException</u></a>	경우에 throw 되는 예외는 <a href="#"><u>Thread</u></a> 대기 상태에서는 중단 합니다.
 <a href="#"><u>ThreadLocal&lt;T&gt;</u></a>	데이터의 스레드 로컬 저장소를 제공합니다.
 <a href="#"><u>ThreadPool</u></a>	작업 실행, 작업 항목 게시, 비동기 I/O 처리, 다른 스레드 대신 기다리기 및 타이머 처리에 사용할 수 있는 스레드 풀을 제공합니다.
 <a href="#"><u>ThreadStartException</u></a>	내부 운영 체제 스레드가 사용자 코드를 실행할 수 없는 상태로 시작된 후 관리되는 스레드에서 실패가 발생한 경우에 throw 되는 예외입니다.
 <a href="#"><u>ThreadStateException</u></a>	경우에 throw 되는 예외는 <a href="#"><u>Thread</u></a> 에 잘못 된 <a href="#"><u>ThreadState</u></a> 메서드 호출에 대 한 합니다.
 <a href="#"><u>Timeout</u></a>	무한 시간 제한 간격을 지정하는 상수를 포함합니다. 이 클래스는 상속될 수 없습니다.
 <a href="#"><u>Timer</u></a>	지정된 간격으로 메서드를 실행하는 메커니즘을 제공합니다. 이 클래스는 상속될 수 없습니다.
 <a href="#"><u>Timer</u></a>	이 유형에 대 한 .NET Framework 소스 코드를 찾아보려면 참조는 <a href="#"><u>Reference Source</u></a> 합니다.
 <a href="#"><u>Volatile</u></a>	휘발성 메모리 작업을 수행하기 위한 메서드를 포함합니다.
 <a href="#"><u>WaitHandle</u></a>	공유 리소스에 대한 단독 액세스를 기다리는 운영 체제 관련 개체를 캡슐화합니다.
 <a href="#"><u>WaitHandleCannotBeOpenedException</u></a>	시스템 뮤텍스, 세마포 또는 이벤트 대기 핸들 존재 하지 않는 열려고 시도할 때 throw 되는 예외입니다.
 <a href="#"><u>WaitHandleExtensions</u></a>	대기 핸들에 대한 안전한 핸들을 사용하여 작업하기 위해 편의 메서드를 제공합니다.









## 구조체


---

구조체	설명
 <a href="#"><u>AsyncFlowControl</u></a>	마이그레이션 또는 스레드 간 실행 컨텍스트의 흐름을 복원 하는 기능을 제공 합니다.

 <u>AsyncLocalValueChangedArgs&lt;T&gt;</u>	변경 알림을 등록하는 <u>AsyncLocal&lt;T&gt;</u> 인스턴스에 데이터 변경 정보를 제공하는 클래스입니다.
 <u>CancellationToken</u>	작업이 취소되어야 한다는 알림을 전파합니다.
 <u>CancellationTokenRegistration</u>	에 등록 된 콜백 대리자는 <u>CancellationToken</u> 합니다.
 <u>LockCookie</u>	단일 작성기/다중 판독기 의미 체계를 구현 하는 잠금을 정의 합니다. 값 형식입니다.
 <u>NativeOverlapped</u>	비관리 코드에서 볼 수 있고 끝에 추가 예약 필드가 포함된 Win32 OVERLAPPED 구조체와 같은 레이아웃이 있는 명시적 레이아웃을 제공합니다.
 <u>SpinLock</u>	잠금을 얻으려는 스레드가 잠금을 사용할 수 있을 때까지 루프에서 반복적으로 확인하면서 대기하는 기본적인 상호 배타 잠금을 제공합니다.
 <u>SpinWait</u>	회전 기반 대기를 지원합니다.







## 대리자

대리자	설명
 <u>ContextCallback</u>	새 컨텍스트 내에서 호출할 메서드를 나타냅니다.
 <u>IOCompletionCallback</u>	스레드 풀에서 I/O 작업이 완료 되 면 오류 코드에 바이트 수, 및 겹쳐진 값 형식을 받습니다.
 <u>ParameterizedThreadStart</u>	실행 되는 메서드를 나타낸다는 <u>Thread</u> 합니다.
 <u>SendOrPostCallback</u>	메시지가 동기화 컨텍스트로 디스패치될 때 호출할 메서드를 나타냅니다.
 <u>ThreadExceptionHandler</u>	처리 하는 메서드를 나타내는 <u>ThreadException</u> 의 이벤트는 <u>Application</u> 합니다.
 <u>ThreadStart</u>	실행 되는 메서드를 나타낸다는 <u>Thread</u> 합니다.
 <u>TimerCallback</u>	호출을 처리 하는 메서드를 나타낸다는 <u>Timer</u> 합니다.
 <u>WaitCallback</u>	스레드 풀 스레드에 의해 실행될 콜백 메서드를 나타냅니다.

 WaitOrTimerCallback

될 때 호출 될 메서드를 나타내는 WaitHandle 신호를 받거나 시간 초과 합니다.

## 열거형

열거형	설명
 <u>ApartmentState</u>	아파트 상태를 지정 된 <u>Thread</u> 합니다.
 <u>EventResetMode</u>	나타냅니다 여부는 <u>EventWaitHandle</u> 신호를 받은 후 자동 또는 수동으로 재설정 됩니다.
 <u>LazyThreadSafetyMode</u>	<u>System.Lazy&lt;T&gt;</u> 인스턴스가 여러 스레드 간 액세스를 동기화하는 방법을 지정합니다.
 <u>LockRecursionPolicy</u>	동일한 스레드에서 잠금을 여러 번 시작할 수 있는지 여부를 지정합니다.
 <u>ThreadPriority</u>	예약 우선 순위를 지정 된 <u>Thread</u> 합니다.
 <u>ThreadState</u>	실행 상태를 지정 된 <u>Thread</u> 합니다.

## 2) 동기화 비동기 프로그래밍

### ☆ 비동기 (Asynchronous: 동시에 일어나지 않는, 非同期: 같은 시기가 아닌)

비동기란 말 그대로 동시에 일어나지 않는다는 의미입니다. 무엇이 같이 일어나지 않는다는 말일까요? 바로 **요청과 그 결과가 동시에 일어나지 않을 거라는 약속**입니다. 즉, 요청한 그 자리에서 바로 결과가 주어지는 것이 아니라, 이따가 줄게라고 약속하는 것이죠.

비동기식 전송이란 데이터를 전송할 때 하나의 글자를 나타내는 부호의 전후에 START BIT 와 STOP BIT 를 넣어서 블록의 동기화를 취해주는 방식으로 START-STOP 전송 방식이라고도 한다. 비트열을 전송하지 않을 때는 송수신기의 회선은 휴지 상태 (idle, 항상 1)로 있다가 데이터 전송시에 ST 상태(0)를 전송하여 수신측은 타임슬롯의 1/2 시간 동안 0 상태를 유지함을 감지하여 데이터 수신을 준비한다. 글자를 구성하는 각 비트의 길이는 통신속도에 따라 정해진다.

300~2400 BPS 정도의 비교적 저속 데이터 전송에 사용된다. 단점은 프레임 에러(Frame error)가 발생할 수 있으며, 문자당 2~3 비트의 오버헤드가 있다.

#### ☆ 동기 (synchronous: 동시에 일어나는, 同期: 같은 시기)

동기란 동시에 일어난다는 말입니다. 무엇이 동시에 일어날까요? **요청과 그 결과가 동시에 일어난다는 약속**입니다. 바로 요청을 하면 그 요청한 자리에서 바로 결과가 주어져야 한다는 말이죠. 시간이 얼마가 걸리든지 상관없습니다, 동기방식으로 하겠다는 것은 시간이 얼마가 걸리든 요청한 그 자리에서 결과를 주겠다는 약속이기 때문이죠. 즉, 요청과 결과가 한자리에서 동시에 일어난다는 말입니다. 전송매체로 연결되는 두 장치 간에 데이터를 교환하기 위해 전송되는 비트들의 타이밍(전송율, 전송시간, 간격)이 송신측과 수신측이 정확히 송수신할 수 있게 시간을 맞추는 것이다.

##### i. 동기식 전송

데이터 신호와 별도의 클럭 신호를 전송

Manchester 인코딩(Bi-phase encoding)

동기식 전송(Synchronous Transmission)

동기식 전송은 2 대의 송수신 시스템이 통신 시에 시차가 있을 경우 보내온 데이터를 잘못 해석할 가능성을 막기 위해 양방향 시차를 맞추어 수신자가 정확히 수신할 수 있는 기술이다. 한 글자 단위가 아닌 미리 정해진 수만큼 의 글자열을 한 블록으로 만들어 일시에 전송한다.

데이터블럭의 전후에 특정한 제어 정보를 삽입하며, 데이터 블럭과 전후의 제어 정보를 합쳐서 프레임이라고 한다

##### i. 글자 위주 동기 방식

문자를 블럭의 선두에 붙여 동기를 취하는 방식으로 데이터 블럭을 일련의 문자 (8 비트)로 취급하며, 제어 정보도 문자로 구성된다. 한 프레임에는 한 개 이상의 동기화 문자로 시작된다. 동기화 문자는 데이터 블럭의 시작을 알리는 비트패턴 (SYN), 데이터 블럭의 마지막을 알리는 비트패턴(ETX)등으로 구성된다.



### 3) 비동기 프로그래밍의 코드 작성

대부분의 응용 프로그램은 메서드를 비동기적으로 호출합니다. 이렇게 하면 메서드 호출이 실행되는 동안 응용 프로그램이 유용한 작업을 계속할 수 있기 때문입니다. WCF(Windows Communication Foundation) 서비스와 클라이언트는 두 개의 다른 응용 프로그램 수준에서 비동기 작업 호출에 참여할 수 있으므로 WCF 응용 프로그램에서 보다 유연하게 상호 작용과 균형을 맞춰 처리량을 최대화할 수 있습니다.

#### 비동기 작업 형식

---

WCF의 모든 서비스 계약은 매개 변수 형식 및 반환 값에 관계없이 WCF 특성을 사용하여 클라이언트와 서비스 간의 특정 메시지 교환 패턴을 지정합니다. WCF에서는 인바운드 및 아웃바운드 메시지를 자동으로 적절한 서비스 작업이나 실행 중인 클라이언트 코드로 라우팅합니다.

클라이언트는 특정 작업의 메시지 교환 패턴을 지정하는 서비스 계약만 소유하며 기본 메시지 교환 패턴이 확인되는 한 클라이언트가 선택한 프로그래밍 모델을 개발자에게 제공할 수 있습니다. 또한 지정된 메시지 패턴이 확인되는 한 서비스도 임의의 방식으로 작업을 구현할 수 있습니다.

서비스나 클라이언트 구현에서 서비스 계약이 독립되면 WCF 응용 프로그램에서 다음과 같은 비동기 실행을 구현할 수 있습니다.

- 클라이언트는 동기 메시지 교환을 사용하여 요청/응답 작업을 비동기적으로 호출할 수 있습니다.
- 서비스는 동기 메시지 교환을 사용하여 요청/응답 작업을 비동기적으로 구현할 수 있습니다.
- 클라이언트나 서비스의 구현에 관계없이 메시지 교환은 단방향일 수 있습니다.

#### 제안된 비동기 시나리오

---

작업 서비스 구현에서 I/O 작업을 수행하는 경우와 같이 블로킹 호출을 만드는 경우 서비스 작업 구현에서 비동기 접근 방법을 사용하는 것이 좋습니다. 비동기 작업 구현 중인 경우에는 비동기 작업 및 메서드를 호출하여 비동기 호출 경로를 최대한 확장해 보세요. 예를 들면 `BeginOperationOne()` 내에서 `BeginOperationTwo()`를 호출합니다.

- 다음과 같은 경우에는 클라이언트 또는 호출 응용 프로그램에서 비동기 접근 방법을 사용하세요.
- 중간 계층 응용 프로그램에서 작업을 호출하는 경우. (이러한 시나리오 추가 정보는 [중간 계층 클라이언트 응용 프로그램을 참조하세요.](#))
- ASP.NET 페이지 내에서 작업을 호출하는 경우에는 비동기 페이지를 사용하세요.
- Windows Forms 또는 WPF(Windows Presentation Foundation) 등의 단일 스레드 응용 프로그램에서 작업을 호출하는 경우. 이벤트 기반의 비동기 호출 모델을 사용하는 경우에는 결과 이벤트가 UI 스레드에서 발생하기 때문에 사용자가 직접 여러 스레드를 처리할 필요 없이 응용 프로그램에 응답이 추가됩니다.
- 일반적으로 동기 호출과 비동기 호출 중에서 선택해야 하는 경우에는 비동기 호출을 선택하세요.

## **비동기 서비스 작업 구현**

---

다음 세 가지 방법 중 하나를 사용하여 비동기 작업을 구현할 수 있습니다.

1. 작업 기반 비동기 패턴
2. 이벤트 기반 비동기 패턴
3. IAsyncResult 비동기 패턴

### **작업 기반 비동기 패턴**

---

작업 기반 비동기 패턴은 가장 쉽고 단순하기 때문에 비동기 작업을 구현하는 데 가장 선호하는 방법입니다. 이 방법을 사용하려면 서비스 작업을 구현하고 반환 형식으로 `Task<T>`를 지정하면 됩니다. 여기서, T는 논리 연산에서 반환하는 형식입니다.

### **이벤트 기반 비동기 패턴**

---

이벤트 기반 비동기 패턴을 지원하는 서비스에는 `MethodNameAsync`라는 작업이 하나 이상 있습니다. 이러한 메서드는 현재 스레드에서 동일한 작업을 수행하는 동기 버전을 미리링할 수 있습니다. 또한 이 클래스에는 `MethodNameCompleted` 이벤트가 있을 수 있고 `MethodNameAsyncCancel` 메서드가 있거나 단순히 `CancelAsync` 메서드가 있을 수

있습니다. 작업을 호출하려는 클라이언트는 작업이 완료될 때 호출할 이벤트 처리기를 정의합니다.

### ***IAsyncResult 비동기 패턴***

---

.NET Framework 비동기 프로그래밍 패턴을 사용하고 `AsyncPattern` 속성이 **true** 로 설정된 **<Begin>** 메서드를 표시하여 서비스 작업을 비동기 방식으로 구현할 수 있습니다. 이 경우 비동기 작업은 동기 작업과 동일한 형태로 메타데이터에 노출됩니다. 즉, 요청 메시지와 관련 응답 메시지가 포함된 단일 작업으로 노출됩니다. 그런 다음 클라이언트 프로그래밍 모델에서는 둘 중 하나를 선택할 수 있습니다. 즉, 서비스가 호출될 때 요청-응답 메시지 교환이 발생하는 한 이 패턴을 동기 작업이나 비동기 작업으로 나타낼 수 있습니다.

일반적으로 시스템의 비동기 특성을 사용하는 경우 스레드에 대한 종속성을 사용하지 않아야 합니다. 작업 디스패치 처리의 다양한 단계에 데이터를 전달하는 가장 안정적인 방법은 확장을 사용하는 것입니다.

예제를 보려면 방법: 비동기 서비스 작업 구현을 참조하세요.

클라이언트 응용 프로그램에서 호출되는 방식에 관계없이 비동기적으로 실행되는 계약 작업 X를 정의하려면 다음을 수행합니다.

- 패턴 **BeginOperation** 및 **EndOperation** 을 사용하여 두 개의 메서드를 정의합니다.
- **BeginOperation** 메서드는 작업에 대한 *in* 및 *ref* 매개 변수를 포함하고 `IAsyncResult` 형식을 반환합니다.
- **EndOperation** 메서드는 *out* 및 *ref* 매개 변수뿐 아니라 `IAsyncResult` 매개 변수를 포함하고 작업 반환 형식을 반환합니다.

## 4) 개체지향 처리와 데이터 주고 받기

WCF(Windows Communication Foundation)는 일종의 메시징 인프라입니다. WCF 는 메시지를 받고, 처리하고, 추가 작업을 위해 사용자 코드로 디스패치하거나, 사용자 코드에서 제공된 데이터로부터 메시지를 생성하고 이 메시지를 대상에 전달할 수 있습니다. 고급 개발자를 대상으로 한 이 항목에서는 메시지 및 포함된 데이터를 처리하기

위한 아키텍처에 대해 설명합니다. 데이터를 주고 받는 방법을 보다 간단하게, 작업에 초점을 두고 설명하는 내용은 서비스 계약에서 데이터 전송 지정을 참조하십시오.

#### 참고:

이 항목에서는 WCF 개체 모델 검사로는 보이지 않는 WCF 구현 세부 정보에 대해 설명합니다. 문서화된 구현 세부 정보와 관련하여 두 가지 주의해야 할 점이 있습니다. 첫째, 설명은 간결하지만 실제 구현은 최적화 또는 다른 이유로 인해 훨씬 복잡할 수 있습니다. 둘째, 문서화된 사항을 비롯한 특정 구현 세부 정보에 의존해서는 안 됩니다. 이러한 사항은 버전이 변경되는 경우 또는 서비스 릴리스에서 예고 없이 변경될 수 있기 때문입니다.

#### 기본 아키텍처

WCF 메시지 처리 기능의 핵심은 Message 클래스이며, 이에 대해서는 Message 클래스 사용에 자세히 설명되어 있습니다. WCF의 런타임 구성 요소는 연결 지점이 되는 **Message** 클래스와 함께 두 개의 주요 부분, 즉, 채널 스택과 서비스 프레임워크로 구분할 수 있습니다.

채널 스택은 유효한 **Message** 인스턴스와 메시지 데이터를 주고받는 일부 동작 간의 변환을 담당합니다. 보내는 쪽에서는 채널 스택이 유효한 **Message** 인스턴스를 사용하고, 처리를 수행한 다음 논리적으로 메시지 보내기에 해당하는 동작을 수행합니다. 이러한 동작은 구현에 따라 TCP 또는 HTTP 패킷 보내기, 메시지 큐에서 메시지 대기시키기, 메시지를 데이터베이스에 쓰기, 파일 공유를 위해 메시지 저장하기 또는 기타 동작이 될 수 있습니다. 가장 일반적인 동작은 네트워크 프로토콜을 통해 메시지를 보내는 것입니다. 받는 쪽에서는 이와 반대의 작업을 수행합니다. 동작(TCP 또는 HTTP 패킷의 도착 또는 기타 동작)을 감지하면 처리를 수행한 다음 채널 스택에서 이 동작을 유효한 **Message** 인스턴스로 변환합니다.

**Message** 클래스 및 채널 스택을 직접 사용함으로써 WCF를 사용할 수 있습니다. 그러나 이 방법은 어렵고 시간이 많이 걸립니다. 또한 **Message** 개체는 메타데이터 지원을 제공하지 않으므로, 이 방법으로 WCF를 사용할 경우 강력한 형식의 WCF 클라이언트를 생성할 수 없습니다.

따라서 WCF에는 **Message** 개체를 생성하고 받는 데 사용할 수 있는 간편한 프로그래밍 모델을 제공하는 서비스 프레임워크가 포함되어 있습니다. 서비스 프레임워크는 서비스 계약의 개념을 통해 서비스를 .NET Framework 형식으로 매핑하고, OperationContractAttribute 특성으로 표시된 단순한 .NET Framework 메서드인 사용자 작업으로 메시지를 디스패치합니다. 자세한 내용은 서비스 계약 디자인을 참조하십시오. 이러한 메서드에는 매개 변수와 반환 값이 있을 수 있습니다. 서비스 쪽에서는 서비스

프레임워크가 들어오는 **Message** 인스턴스를 매개 변수로 변환하고, 반환 값을 나가는 **Message** 인스턴스로 변환합니다. 클라이언트 쪽에서는 반대로 수행됩니다. 예를 들어 아래의 FindAirfare 작업을 살펴 봅니다.

FindAirfare 를 클라이언트에서 호출했다고 가정합니다. 클라이언트의 서비스 프레임워크는 *FromCity* 및 *ToCity* 매개 변수를 나가는 **Message** 인스턴스로 변환하고, 보낼 채널 스택에 이를 전달합니다.

서비스 쪽에서는 **Message** 인스턴스가 채널 스택으로부터 도착할 때 서비스 프레임워크가 메시지에서부터 관련 데이터를 추출하여 *FromCity* 및 *ToCity* 매개 변수를 채운 다음, 서비스측 FindAirfare 메서드를 호출합니다. 메서드가 반환되면 서비스 프레임워크는 반환된 정수 값과 *IsDirectFlight* 출력 매개 변수를 사용하여 이 정보가 포함된 **Message** 개체 인스턴스를 만듭니다. 그런 다음 **Message** 인스턴스를 클라이언트로 다시 보낼 채널 스택에 전달합니다.

클라이언트 쪽에서는 응답 메시지가 포함된 **Message** 인스턴스가 채널 스택으로부터 나타납니다. 서비스 프레임워크는 반환 값과 *IsDirectFlight* 값을 추출하고 이러한 값을 클라이언트의 호출자에게 반환합니다.

## Message 클래스

**Message** 클래스는 메시지를 추상적으로 표현하도록 만들어졌으나 해당 디자인은 SOAP 메시지에 강하게 연결되어 있습니다. **Message** 에는 메시지 본문, 메시지 헤더 및 메시지 속성이라는 세 가지 주요 정보가 포함되어 있습니다.

### 메시지 본문

메시지 본문은 메시지의 실제 데이터 페이로드를 나타내는 부분입니다. 메시지 본문은 항상 XML Infoset 으로 표시됩니다. 단, 이는 WCF 에서 만들어지거나 받은 모든 메시지가 XML 형식이어야 한다는 것을 의미하지는 않습니다. 메시지 형식은 메시지 본문의 해석 방법을 결정하는 채널 스택에 따라 달라집니다. 채널 스택은 메시지를 XML 로 보내거나, 다른 형식으로 변환하거나, 아예 전부 생략할 수도 있습니다. 물론 WCF 에서 제공하는 대부분의 바인딩과 함께 메시지 본문은 SOAP 봉투의 본문 섹션에서 XML 콘텐츠로 표현됩니다.

**Message** 클래스가 본문을 나타내는 XML 데이터와 함께 반드시 버퍼를 포함하지는 않습니다. 논리적으로는 **Message** 가 XML Infoset 을 포함하지만, 이 Infoset 은 동적으로 생성될 수 있으며, 메모리에 물리적으로 존재하지는 않습니다.

## 데이터를 메시지 본문에 넣기

데이터를 메시지 본문에 넣기 위한 일관된 메커니즘은 없습니다. **Message** 클래스에는 추상 메서드인 OnWriteBodyContents 가 있으며, 이 메서드는 XmlDictionaryWriter 를 사용합니다. **Message** 클래스의 각 서브클래스는 이 메서드를 재정의하고, 해당 콘텐츠를 작성하는 기능을 담당합니다. 메시지 본문은 **OnWriteBodyContent** 를 생성하는 XML Infoset 을 논리적으로 포함합니다. 다음 **Message** 서브클래스 예제를 참조하십시오.

물리적으로 `AirfareRequestMessage` 인스턴스에는 두 개의 문자열("fromCity" 및 "toCity")만 포함됩니다. 그러나 논리적으로는 이 메시지에 다음 XML infoset 이 포함됩니다.

### 복사

```
<airfareRequest>
  <from>Tokyo</from>
  <to>London</to>
</airfareRequest>
```

물론, 서비스 프레임워크를 사용하여 작업 계약 매개 변수로부터 이전과 같은 메시지를 만들 수 있기 때문에 일반적으로는 이 방식으로 메시지를 만들지 않습니다. 또한 **Message** 클래스에는 일반 콘텐츠 형식을 가진 메시지를 만드는 데 사용할 수 있는 정적 **CreateMessage** 메서드가 있습니다. 이러한 일반 콘텐츠 형식에는 빈 메시지, DataContractSerializer 와 함께 XML 로 serialize 된 개체가 포함된 메시지, SOAP 오류가 포함된 메시지, XmlReader 로 표시되는 XML 이 포함된 메시지 등이 있습니다.

## 메시지 본문에서 데이터 가져오기

메시지 본문에 저장된 데이터를 추출하는 방법으로는 다음 두 가지가 있습니다.

- WriteBodyContents 메서드를 호출하고 XML 작성기에서 전달하면 한 번에 메시지 본문 전체를 가져올 수 있습니다. 전체 메시지 본문이 이 작성기에 작성되어 있습니다. 전체 메시지 본문을 한 번에 가져오는 것을 *메시지 쓰기*라고도 합니다. 쓰기는 주로 메시지를 보낼 때 채널 스택에서 수행합니다. 채널 스택의 일부는 일반적으로 전체 메시지 본문에 대한 액세스 권한, 인코딩 권한, 전송 권한을 가집니다.
- 메시지 본문에서 정보를 가져오는 또 다른 방법은 GetReaderAtBodyContents 를 호출하고 XML 판독기를 가져오는 것입니다. 그런 다음 필요에 따라 판독기에서 메서드를 호출하여 메시지 본문에 순차적으로 액세스할 수 있습니다. 메시지 본문을 하나씩 가져오는 것을 *메시지 읽기*라고도 합니다. 메시지 읽기는 주로 메시지를 받을 때 서비스 프레임워크에서 사용합니다. 예를 들어

**DataContractSerializer** 를 사용 중일 때 서비스 프레임워크는 본문에 대해 XML 판독기를 가져오고 이를 deserialization 엔진에 전달합니다. 그러면 이 엔진은 메시지를 요소별로 읽기 시작하고 해당 개체 그래프를 생성합니다.

메시지 본문은 한 번만 검색할 수 있습니다. 이를 통해 정방향 스트림을 사용할 수 있습니다. 예를 들어 FileStream 으로부터 읽은 **OnWriteBodyContents** 재정의의 쓰고, 결과를 XML Infoset 으로 반환할 수 있습니다. 파일의 시작 부분으로 "되돌아갈" 필요가 없습니다.

**WriteBodyContents** 및 **GetReaderAtBodyContents** 메서드는 해당 메시지 본문이 전에 검색된 적이 없는지 확인한 다음 **OnWriteBodyContents** 또는 **OnGetReaderAtBodyContents** 를 각각 호출하기만 하면 됩니다.

## WCF 의 메시지 사용

대부분의 메시지는 *보낼* 메시지(채널 스택에서 보낼 서비스 프레임워크에서 만드는 메시지) 또는 *들어오는* 메시지(채널 스택에서 전송되어 서비스 프레임워크에서 해석되는 메시지) 중 하나로 분류될 수 있습니다. 또한 채널 스택은 버퍼링 모드 또는 스트리밍 모드로 작동할 수 있으며 서비스 프레임워크는 스트리밍 또는 비스트리밍 프로그래밍 모델을 노출할 수 있습니다. 다음 표는 이러한 조합에 따라 발생 가능한 경우를 간략한 구현 정보와 함께 보여 줍니다.

메시지 유형	메시지의 본문 데이터	쓰기(OnWriteBodyContents) 읽기(OnGetReaderAtBodyContents)	
		구현	구현
비스트리밍 프로그래밍 모델에서 만들어진 나가는 메시지	메시지 작성에 필요한 데이터(예: serialize 하는 데 필요한 개체 및 <b>DataContractSerializer</b> 인스턴스)*	저장된 데이터를 기반으로 메시지를 작성하는 사용자 지정 논리(예: <b>DataContractSerializer</b> 의 <b>WriteObject</b> 호출. 단 이 serializer 를 사용 중인 경우)*	<b>OnWriteBodyContents</b> 를 호출하고, 결과를 버퍼링하고, 버퍼를 통해 XML 판독기를 반환
스트리밍된 프로그래밍 모델에서 만들어진 나가는 메시지	작성할 데이터가 포함된 <b>Stream</b> *	<b>IStreamProvider</b> 메커니즘을 사용하여 저장된 스트림에서 가져온 데이터 작성*	<b>OnWriteBodyContents</b> 를 호출하고, 결과를 버퍼링하고, 버퍼를 통해 XML 판독기를 반환

스트리밍	네트워크를 통해	<b>WriteNode</b> 를 사용하여	
채널	<b>XmlReader</b> 와 함께	저장된 <b>XmlReader</b> 로부터	저장된 <b>XmlReader</b> 반환
스택에서	들어오는 데이터를	콘텐츠 작성	
들어오는	나타내는 <b>Stream</b> 개체		
메시지			

스트리밍되지

않은 채널	<b>XmlReader</b> 와 함께	<b>WriteNode</b> 를 사용하여	
스택에서	본문 데이터가 포함된	저장된 <b>XmlReader</b> 로부터	저장된 lang 반환
들어오는	버퍼	콘텐츠를 작성합니다.	
메시지			

\* 이러한 항목은 **Message** 서브클래스에서 직접 구현되는 것이 아니라 [BodyWriter](#) 클래스의 서브클래스에서 구현됩니다. **BodyWriter** 에 대한 자세한 내용은 [Message 클래스 사용](#)을 참조하십시오.

## 메시지 헤더

메시지에는 헤더가 포함될 수 있습니다. 헤더는 논리적으로 이름, 네임스페이스 및 다른 속성과 연관된 XML Infoset 으로 구성됩니다. **Message** 의 **Headers** 속성을 사용하여 메시지 헤더에 액세스합니다. 각 헤더는 [MessageHeader](#) 클래스로 표현됩니다. 일반적으로 메시지 헤더는 SOAP 메시지를 사용하기 위해 구성된 채널 스택을 사용할 때 SOAP 메시지 헤더로 매핑됩니다.

메시지 헤더로 정보를 가져오고, 메시지 헤더에서 정보를 추출하는 것은 메시지 본문을 사용하는 것과 비슷합니다. 스트리밍이 지원되지 않으므로 프로세스가 다소 간단합니다. 동일한 헤더의 콘텐츠에 두 번 이상 액세스할 수 있으며, 헤더가 항상 버퍼링되도록 하여 임의의 순서로 액세스할 수 있습니다. 헤더를 통해 XML 판독기를 가져오기 위해 사용할 수 있는 일반 용도의 메커니즘은 없지만 WCF 의 내부에 이러한 기능을 통해 읽을 수 있는 헤더를 나타내는 **MessageHeader** 서브클래스는 있습니다. 이 **MessageHeader** 형식은 사용자 지정 응용 프로그램 헤더와 함께 메시지가 나타날 때 채널 스택에서 만듭니다. 이를 통해 서비스 프레임워크는 **DataContractSerializer** 와 같은 deserialization 엔진을 사용하여 이러한 헤더를 해석합니다.

자세한 내용은 [Message 클래스 사용](#)을 참조하십시오.

## 메시지 속성

메시지에는 속성이 포함될 수 있습니다. 속성은 문자열 이름과 관련된 모든 .NET Framework 개체입니다. **Message** 의 **Properties** 속성을 통해 속성에 액세스합니다.



일반적으로 각각 SOAP 본문과 SOAP 헤더로 매핑되는 메시지 본문과 메시지 헤더와는 달리, 메시지 속성은 일반적으로 메시지와 함께 보내거나 받지 않습니다. 메시지 속성은 주로 채널 스택의 다양한 채널 간이나 채널 스택과 서비스 모델 간에 메시지에 대한 데이터를 전달하기 위한 통신 메커니즘으로 존재합니다.

예를 들어 WCF 의 일부로 포함된 HTTP 전송 채널은 클라이언트로 응답을 보낼 때 "404(찾을 수 없음)", "500(내부 서버 오류)" 등의 다양한 HTTP 상태 코드를 만들 수 있습니다. 이 채널은 회신 메시지를 보내기 전에 **Message** 의 **Properties** 에 HttpResponseMessageProperty 형식의 개체가 포함된 "httpResponse"라는 속성이 포함되어 있는지 여부를 확인합니다. 이러한 속성이 있으면 Statuscode 속성을 살펴 보고 해당 상태 코드를 사용합니다. 이러한 속성이 없으면 기본 "200(확인)" 코드를 사용합니다.

자세한 내용은 Message 클래스 사용을 참조하십시오.

### **메시지 전체**

지금까지 메시지의 여러 부분에 각각 액세스하는 데 사용되는 메서드를 살펴 보았습니다. 그러나 **Message** 클래스는 메시지 전체를 대상으로 작동하는 메서드도 제공합니다. 예를 들어 **WriteMessage** 메서드는 XML 작성기에 전체 메시지를 작성합니다.

이렇게 하려면 전체 **Message** 인스턴스와 하나의 XML Infoset 간에 매핑이 정의되어야 합니다. 이러한 매핑은 실제로 존재하며, WCF에서는 SOAP 표준을 사용하여 이 매핑을 정의합니다. **Message** 인스턴스를 XML Infoset 으로 작성할 때, 결과 Infoset 은 해당 메시지가 포함된 유효한 SOAP 봉투입니다. 따라서 **WriteMessage** 는 일반적으로 다음 단계를 수행합니다.

1. SOAP 봉투 요소 열기 태그를 씁니다.
2. SOAP 헤더 요소 열기 태그를 쓰고, 모든 헤더를 작성하고, 헤더 요소를 닫습니다.
3. SOAP 본문 요소 열기 태그를 씁니다.
4. **WriteBodyContents** 또는 동등한 메서드를 호출하여 본문을 작성합니다.
5. 본문 및 봉투 요소를 닫습니다.

이전 단계는 SOAP 표준과 밀접하게 연관되어 있습니다. 여러 버전의 SOAP 가 존재하기 때문에 발생하는 문제가 있습니다. 예를 들어 사용 중인 SOAP 버전을 모르면 SOAP 봉투 요소를 올바르게 작성할 수 없습니다. 또한 복잡한 SOAP 별 매핑을 완전히 해제하는 것이 좋은 경우도 있습니다.

이러한 목적을 위해 **Version** 속성이 **Message** 에서 제공됩니다. 이 속성은 메시지를 작성할 때 사용할 SOAP 버전으로 설정하거나, SOAP 별 매핑을 방지하도록 **None** 으로 설정할 수 있습니다. **Version** 속성이 **None** 으로 설정되면 전체 메시지를 대상으로

작동하는 메서드는 메시지가 본문으로만 구성되어 있는 것처럼 작동합니다. 예를 들어 **WriteMessage** 는 앞에서 설명한 여러 단계를 수행하는 대신 **WriteBodyContents** 만 호출합니다. 들어오는 메시지에서 **Version** 이 자동으로 감지되고 올바르게 설정됩니다.

## 채널 스택

### 채널

앞서 언급한 대로 채널 스택은 나가는 **Message** 인스턴스를 일부 동작(예: 네트워크를 통한 패킷 전송)으로 변환하거나 일부 동작(예: 네트워크 패킷 수신)을 들어오는 **Message** 인스턴스로 변환하는 역할을 담당합니다.

채널 스택은 시퀀스의 순서로 하나 이상의 채널로 구성됩니다. 나가는 **Message** 인스턴스는 스택의 첫 번째 채널(*맨 위 채널*이라고도 함)로 전달되며, 이 채널은 해당 인스턴스를 스택의 다음 아래 채널로 전달합니다. 메시지는 *전송 채널*이라고도 하는 마지막 채널에서 종료됩니다. 들어오는 메시지는 전송 채널에서 시작되며, 스택의 아래 채널에서 위 채널로 전달됩니다. 메시지는 보통 맨 위 채널에서부터 서비스 프레임워크로 전달됩니다. 이는 응용 프로그램 메시지의 일반적인 패턴이지만 일부 채널은 조금 다르게 작동할 수 있습니다. 예를 들어 위의 채널로부터 전달되는 메시지 없이 자체 인프라 메시지를 보낼 수 있습니다.

채널은 스택을 통과하므로 여러 방법으로 메시지에서 작동할 수 있습니다. 가장 일반적인 작업은 나가는 메시지에 헤더를 추가하고, 들어오는 메시지의 헤더를 읽는 것입니다. 예를 들어 채널은 메시지의 디지털 서명을 계산하고 이를 헤더로 추가할 수 있습니다. 또한 채널은 들어오는 메시지에서 이 디지털 서명 헤더를 검사하고, 유효한 서명이 없는 메시지가 채널 스택에서 자체 방식을 만들지 못하도록 차단합니다. 채널은 종종 메시지 속성을 설정하거나 검사하기도 합니다. 메시지 본문은 수정이 허용되는 경우에도 일반적으로 수정되지 않습니다. 예를 들어 WCF 보안 채널이 메시지 본문을 암호화할 수 있습니다.

### 전송 채널 및 메시지 인코더

스택의 가장 아래쪽에 있는 채널은 나가는 **Message** 를 다른 채널에서 수정된 대로 일부 동작으로 실제로 변환하는 기능을 담당합니다. 받는 쪽에서 이 채널은 다른 채널이 처리하는 **Message** 로 일부 동작을 변환하는 채널입니다.

앞에서 설명한 대로 일부 예제만 제공하기 위해서도 다양한 동작이 있을 수 있습니다. 이러한 동작으로는 다양한 프로토콜을 통한 네트워크 패킷 주고받기, 데이터베이스의 메시지 읽기 또는 쓰기, 메시지 큐에서 메시지 대기시키기 또는 제거하기 등이 있습니다. 이러한 모든 동작에는 한 가지 공통점이 있습니다. 즉, 보내기, 받기, 읽기, 쓰기, 큐에

대기시키기 또는 큐에서 제거하기 등을 수행할 수 있는 실제 바이트 그룹과 WCF **Message** 인스턴스 간의 변형이 필요합니다. **Message** 를 바이트 그룹으로 변환하는 프로세스를 *인코딩*이라 하며, 바이트 그룹으로부터 **Message** 를 만드는 역 프로세스를 *디코딩*이라 합니다.

대부분의 전송 채널은 *메시지 인코더*라는 구성 요소를 사용하여 인코딩 및 디코딩 작업을 수행합니다. 메시지 인코더는 MessageEncoder 클래스의 서브클래스입니다.

**MessageEncoder** 는 다양한 **ReadMessage** 및 **WriteMessage** 메서드 오버로드를 포함하여 **Message** 와 바이트 그룹 사이에서 변환합니다.

보내는 쪽에서는 버퍼링 전송 채널이 상위 채널로부터 **Message** 개체를 받아서 **WriteMessage** 로 전달합니다. 버퍼링 전송 채널은 바이트 배열을 다시 가져온 다음 이를 사용하여 해당 동작(예: 이러한 바이트를 유효한 TCP 패킷으로 패키징화하고 올바른 대상으로 이를 보냄)을 수행합니다. 스트리밍 전송 채널은 먼저 **Stream** 을 만든 다음(예: 나가는 TCP 연결을 통해), 메시지를 작성하는 적합한 **WriteMessage** 오버로드로 보내기 위해 필요한 **Stream** 과 **Message** 를 모두 전달합니다.

받는 쪽에서는 버퍼링 전송 채널이 (들어오는 TCP 패킷 등으로부터) 들어오는 바이트를 배열로 추출하고, **ReadMessage** 를 호출하여 채널 스택 위쪽으로 전달할 수 있는 **Message** 개체를 가져옵니다. 스트리밍 전송 채널은 **Stream** 개체(예: 들어오는 TCP 연결을 통한 네트워크 스트림)를 만들고 이를 **ReadMessage** 로 전달하여 **Message** 개체를 다시 가져옵니다.

전송 채널과 메시지 인코더를 반드시 분리해야 하는 것은 아니지만 메시지 인코더를 사용하지 않는 전송 채널을 작성할 수 있습니다. 단, 분리를 하게 되면 컴퍼지션이 쉽다는 장점이 있습니다. 전송 채널이 기본 **MessageEncoder** 만 사용하는 경우에는 모든 WCF 또는 타사 메시지 인코더와 함께 사용할 수 있습니다. 마찬가지로, 일반적으로 모든 전송 채널에서 동일한 인코더를 사용할 수 있습니다.

### **메시지 인코더 작업**

인코더의 일반적인 작업을 설명하려면 다음 네 가지 경우를 고려하는 것이 좋습니다.

작업	설명
인코딩, 버퍼링	버퍼링 모드에서는 인코더가 보통 다양한 크기의 버퍼를 만든 다음 이를 통해 XML 작성기를 만듭니다. 그런 다음 인코딩되는 메시지에서 <u>WriteMessage</u> 를 호출합니다. 이 메시지는 이 항목의 <b>Message</b> 에 대한 이전 단원에서 설명한 것처럼 <b>WriteBodyContents</b> 를 사용하여 헤더와 본문을 차례로 작성합니다. 그런 다음 사용할 전송 채널에 대해 바이트 배열로 표시되는 버퍼의 콘텐츠가 반환됩니다.

인코딩, 스트리밍	스트리밍 모드에서는 작업이 위의 모드와 비슷하지만 좀 더 간단합니다. 버퍼링할 필요가 없기 때문입니다. XML 작성기는 일반적으로 스트림을 통해 만들어지며, <b>Message</b> 에서 <b>WriteMessage</b> 가 호출되어 이 작성기에 이를 작성합니다.
디코딩, 버퍼링	버퍼링 모드에서 디코딩할 때는 버퍼링된 데이터가 포함된 특별한 <b>Message</b> 서브클래스가 일반적으로 만들어집니다. 메시지의 헤더가 읽히고, 메시지 본문에 배치된 XML 판독기가 만들어집니다. 이는 <b>GetReaderAtBodyContents</b> 와 함께 반환될 판독기입니다.
디코딩, 스트리밍	스트리밍 모드에서 디코딩할 때는 특별한 메시지 서브클래스가 일반적으로 만들어집니다. 스트림은 모든 헤더를 읽고 이를 메시지 본문에 배치할 수 있는 고급 기능입니다. 그런 다음 XML 판독기가 스트림을 통해 만들어집니다. 이는 <b>GetReaderAtBodyContents</b> 와 함께 반환될 판독기입니다.

인코더는 다른 기능도 수행할 수 있습니다. 예를 들어 인코더는 XML 판독기 및 작성기를 풀링할 수 있습니다. 필요할 때마다 새 XML 판독기나 작성기를 만들려면 비용이 많이 듭니다. 그러므로 일반적으로 인코더는 구성 가능한 크기의 작성기 풀과 판독기 풀을 유지 관리합니다. 위에서 설명한 인코더 작업에 대한 설명에서 "XML 판독기/작성기 만들기"라는 구문이 사용될 때 이는 일반적으로 "풀에서 하나를 사용하거나, 사용 가능한 것이 없으면 하나를 만든다"는 것을 의미합니다. 인코더(및 디코딩하는 동안 인코더가 만드는 **Message** 서브클래스)는 작성기와 반환기가 더 이상 필요하지 않을 때(예: **Message** 가 닫힐 때) 이들을 풀로 반환하는 논리를 포함합니다.

WCF 는 자체에서 추가 사용자 지정 형식을 만들 수 있긴 하지만 세 가지 메시지 인코더를 제공합니다. 제공되는 형식에는 텍스트, 이진 및 MTOM(Message Transmission Optimization Mechanism)이 있습니다. 자세한 내용은 메시지 인코더 선택에 설명되어 있습니다.

### ***IStreamProvider*** 인터페이스

스트리밍된 본문을 포함하고 있는 보내는 메시지를 XML 작성기에 쓸 때 **Message** 는 **OnWriteBodyContents** 구현에서 다음과 유사한 호출의 시퀀스를 사용합니다.

- 스트림 앞에 필요한 정보를 씁니다(예: 여는 XML 태그).
- 스트림을 씁니다.
- 스트림 뒤에 필요한 정보를 씁니다(예: 닫는 XML 태그).

이는 텍스트 XML 인코딩과 유사한 인코딩을 사용하여 작동합니다. 그러나 일부 인코딩은 XML infoset 정보(예: XML 요소의 시작 및 끝 태그)를 요소 내에 포함된 데이터와 함께 사용하지 않습니다. 예를 들어 MTOM 인코딩에서 메시지는 여러 부분으로 나누어집니다. 하나의 부분에는 실제 요소 콘텐츠의 다른 부분에 대한 참조가 포함될 수 있는 XML infoset 이 포함됩니다. XML Infoset 이 일반적으로 스트리밍된 콘텐츠와 비교하여 작으므로

InfoSet 을 버퍼링하고 쓴 다음 스트리밍된 방법으로 콘텐츠를 쓰는 것이 좋습니다. 즉, 이는 닫는 요소 태그가 작성될 때까지는 스트림을 쓰면 안 된다는 것을 의미합니다.

이를 위해 **IStreamProvider** 인터페이스가 사용됩니다. 인터페이스에는 작성할 스트림을 반환하는 GetStream 메서드가 있습니다. **OnWriteBodyContents** 에서 스트리밍된 메시지 본문을 쓰는 올바른 방법은 다음과 같습니다.

1. 스트림 앞에 필요한 정보를 씁니다(예: 여는 XML 태그).
2. 쓸 스트림을 반환하는 **IStreamProvider** 구현과 함께 **IStreamProvider** 를 사용하는 **XmlDictionaryWriter** 에서 **WriteValue** 오버로드를 호출합니다.
3. 스트림 뒤에 필요한 정보를 씁니다(예: 닫는 XML 태그).

이 방법을 사용하면 **GetStream** 을 호출하고 스트리밍된 데이터를 작성할 때 XML 작성기를 선택할 수 있습니다. 예를 들어 텍스트 및 이진 XML 작성기는 이를 즉각 호출하고 시작 및 끝 태그 사이에서 스트리밍된 콘텐츠를 작성합니다. MTOM 작성기는 메시지의 적절한 일부를 쓸 준비가 되면 나중에 **GetStream** 을 호출할지 결정할 수 있습니다.

## 서비스 프레임워크에서 데이터 표시

이 항목의 "기본 아키텍처" 단원에서 설명한 대로 서비스 프레임워크는 WCF 의 일부이며 메시지에 대해 사용자 정의 프로그래밍 모델과 실제 **Message** 인스턴스 간의 변환을 담당합니다. 일반적으로 메시지 교환은 서비스 프레임워크에서

**OperationContractAttribute** 특성과 함께 표시된 .NET Framework 메서드로 나타납니다. 메서드는 일부 매개 변수에서 사용할 수 있으며, 반환 값 또는 out 매개 변수(또는 둘 다)를 반환할 수 있습니다. 서비스 쪽에서는 입력 매개 변수가 들어오는 메시지를 나타내며, 반환 값 및 out 매개 변수는 나가는 메시지를 나타냅니다. 클라이언트 쪽에서는 그 반대입니다. 매개 변수 및 해당 반환 값을 사용하여 메시지를 설명하는 프로그래밍 모델은 서비스 계약에서 데이터 전송 지정에서 자세하게 설명합니다. 이 단원에서는 간략한 개요만 제공합니다.

## 프로그래밍 모델

WCF 서비스 프레임워크는 메시지를 설명하기 위해 다음과 같이 다섯 가지의 서로 다른 프로그래밍 모델을 지원합니다.

### 1. 빈 메시지

이는 가장 간단한 경우입니다. 들어오는 빈 메시지를 설명하기 위해 입력 매개 변수를 사용하지 않습니다.

나가는 빈 메시지를 설명하기 위해 void 반환 값을 사용하고 out 매개 변수를 사용하지 않습니다.

이는 단방향 작업 계약과는 다릅니다.

SetDesiredTemperature 예제에서는 양방향 메시지 교환 패턴을 설명합니다. 작업에서 메시지가 반환되지만 비어 있습니다. 작업에서 오류를 반환할 수 있습니다. "Lightbulb 설정" 예제에서 메시지 교환 패턴은 단방향이므로 설명을 위해 나가는 메시지가 없습니다. 이 경우 서비스는 클라이언트로 반환되는 모든 상태를 통신할 수 없습니다.

## 2. Message 클래스 직접 사용

작업 계약에서 직접 **Message** 클래스(또는 해당 서브클래스 중 하나)를 사용할 수 있습니다. 이 경우에는 서비스 프레임워크가 추가 처리 없이 작업에서 채널 스택으로, 또는 그 반대로 **Message** 를 전달합니다.

**Message** 를 직접 사용하는 두 가지 주요 사례가 있습니다. 사용자의 메시지를 설명하기에 충분한 유연성을 제공하는 다른 프로그래밍 모델이 없을 때 이를 고급 시나리오에 사용할 수 있습니다. 예를 들어 메시지 헤더가 될 파일의 속성과 메시지 본문이 될 파일의 내용으로 메시지를 설명하기 위해 디스크의 파일을 사용할 수 있습니다. 그런 다음, 다음과 유사한 항목을 만들 수 있습니다.

작업 계약에서 **Message** 의 두 번째 일반적인 용도는 서비스가 특정 메시지 콘텐츠에 대해 고려하지 않고 메시지에서 검정 상자의 역할을 수행하는 경우입니다. 예를 들어 메시지를 다른 여러 받는 사람에게 전달하는 서비스가 있을 수 있습니다. 계약은 다음과 같이 작성될 수 있습니다.

Action="\*" 줄은 메시지 디스패치를 효과적으로 해제하고, IForwardingService 계약으로 보낸 모든 메시지가 ForwardMessage 작업에 대해 자체 방식을 사용하도록 해줍니다. 일반적으로 디스패처는 메시지의 "Action" 헤더를 검사하여 어떤 작업을 사용할지 결정합니다. Action="\*"는 "Action 헤더의 가능한 모든 값"을 의미합니다.) Action="\*" 및 매개 변수로서의 Message 사용을 조합하면 가능한 모든 메시지를 수신할 수 있기 때문에 이 조합은 "유니버설 계약"으로 알려져 있습니다. 가능한 모든 메시지를 보낼 수 있으려면 Message 를 반환 값으로 사용하고 **ReplyAction** 을 "\*"로 설정해야 합니다. 이렇게 하면 사용자가 반환하는 **Message** 개체를 사용하여 이 헤더를 제어할 수 있으므로 서비스 프레임워크가 자체 Action 헤더에 추가되지 않도록 할 수 있습니다.

## 3. 메시지 계약

WCF에서는 메시지 계약이라는 메시지를 설명하기 위해 선언적인 프로그래밍 모델을 제공합니다. 이 모델은 메시지 계약 사용에 자세히 설명되어 있습니다. 기본적으로 전체

메시지는 [MessageBodyMemberAttribute](#) 및 [MessageHeaderAttribute](#) 같은 특성을 사용하는 단일 .NET Framework 형식으로 표시되어, 메시지 계약 클래스의 어떤 부분이 메시지의 어떤 부분에 매핑되는지 설명합니다.

**Message** 클래스를 직접 사용하는 것만큼 많은 제어를 할 수는 없지만, 메시지 계약을 사용하여 결과 **Message** 인스턴스에 대해 여러 가지 제어를 할 수 있습니다. 예를 들어 메시지 본문은 종종 여러 가지 정보로 구성되며, 이러한 정보는 각각 자체 XML 요소로 표시됩니다. 이러한 요소는 본문에서 직접 발생할 수도 있고(*bare* 모드), 포함된 XML 요소에서 *wrapped*가 될 수도 있습니다. 메시지 계약 프로그래밍 모델을 사용하면 bare-wrapped 모드를 결정하고 래퍼 이름과 네임스페이스를 제어할 수 있게 됩니다.

메시지 계약의 다음 코드 예제에서는 이러한 기능을 보여 줍니다.

**MessageBodyMemberAttribute**, **MessageHeaderAttribute** 또는 기타 관련 특성과 함께 serialize 하도록 표시된 항목이 serialize 가능해야 메시지 계약에 참여할 수 있습니다. 자세한 내용은 이 항목의 뒷부분에 있는 "Serialization" 단원을 참조하십시오.

#### 4. 매개 변수

여러 데이터에 대해 작동하는 작업을 설명하려는 개발자는 종종 메시지 계약이 제공하는 일정 정도의 제어를 필요로 하지 않습니다. 예를 들어 새 서비스를 만들 때 사용자는 보통 bare-wrapped 모드나 래퍼 요소 이름을 결정하지 않으려고 합니다. 이러한 결정을 내리려면 웹 서비스와 SOAP에 대해 잘 알고 있어야 합니다.

WCF 서비스 프레임워크는 사용자에게 이러한 선택을 강요하지 않고도 관련된 여러 가지 정보를 보내거나 받기 위해 최상의 상호 운용성을 가진 SOAP 표현을 자동으로 선택할 수 있습니다. 이러한 여러 가지 정보를 작업 계약의 매개 변수 또는 반환 값으로 설명하기만 해도 이를 수행할 수 있습니다. 예를 들어 다음 작업 계약을 확인해 보십시오.

서비스 프레임워크는 customerID, item, quantity 라는 세 가지의 정보를 모두 메시지 본문에 삽입하고, SubmitOrderRequest 라는 래퍼 요소에서 이들을 래핑하도록 자동으로 결정합니다.

보다 복잡한 메시지 계약 또는 **Message** 기반 프로그래밍 모델로 이동해야 하는 특별한 이유가 없다면 작업 계약 매개 변수의 간단한 목록으로 보내거나 받을 정보를 설명하는 것이 좋습니다.

#### 5. 스트림

메시지 계약의 단독 메시지 본문 부분으로 또는 작업 계약에서 **Stream**(또는 해당 서브클래스 중 하나)을 사용하는 것은 위에서 설명한 항목과 별도의 프로그래밍 모델로

고려될 수 있습니다. 이 방법으로 **Stream** 을 사용하면 사용자의 자체 스트리밍 호환 가능 **Message** 서브클래스를 간결하게 작성할 수 있으므로, 사용자의 계약을 스트리밍 방식으로 유일하게 사용할 수 있습니다. 자세한 내용은 [큰 데이터 및 스트리밍을 참조하십시오](#).

**Stream** 또는 해당 서브클래스 중 하나를 이 방법으로 사용할 때는 serializer 가 호출되지 않습니다. 나가는 메시지의 경우 특별한 스트리밍 **Message** 서브클래스가 만들어지고, **IStreamProvider** 인터페이스의 단원에서 설명한 대로 스트림이 작성됩니다. 들어오는 메시지의 경우 서비스 프레임워크가 들어오는 메시지에 대해 **Stream** 서브클래스를 만들고 이를 해당 작업에 제공합니다.

### 프로그래밍 모델 제한

위에서 설명한 프로그래밍 모델은 임의로 조합할 수 없습니다. 예를 들어 작업이 메시지 계약 형식을 수락할 경우 해당 메시지 계약이 유일한 입력 매개 변수가 되어야 합니다. 뿐만 아니라 해당 작업은 빈 메시지(void 의 반환 형식) 또는 기타 메시지 계약을 반환해야 합니다. 이 항목에서는 이러한 프로그래밍 모델 제한에 대해 [메시지 계약 사용](#), [Message 클래스 사용](#) 및 [큰 데이터 및 스트리밍](#)이라는 각 특정 프로그래밍 모델을 설명합니다.

### 메시지 포맷터

위에서 설명한 프로그래밍 모델은 *메시지 포맷터*라는 구성 요소를 서비스 프레임워크로 플러그 인함으로써 지원됩니다. 메시지 포맷터는 클라이언트 및 서비스 WCF 클라이언트에서 각각 사용하기 위해 [IClientMessageFormatter](#) 또는 [IDispatchMessageFormatter](#) 인터페이스, 또는 둘 다를 구현하는 형식입니다.

메시지 포맷터는 일반적으로 동작별로 플러그 인됩니다. 예를 들어 [DataContractSerializerOperationBehavior](#) 는 데이터 계약 메시지 포맷터를 플러그 인합니다. 이는 서비스 쪽의 [ApplyDispatchBehavior](#) 메서드에서 [Formatter](#) 를 올바른 포맷터로 설정하거나 클라이언트 쪽의 [ApplyClientBehavior](#) 메서드에서 [Formatter](#) 를 올바른 포맷터로 설정함으로써 수행됩니다.

다음 표에서는 메시지 포맷터가 구현할 수 있는 메서드를 나열합니다.

인터페이스	메서드	작업
<b>IDispatchMessageFormatter</b>	<a href="#">DeserializeRequest</a>	들어오는 <b>Message</b> 를 작업 매개 변수로 변환합니다.
<b>IDispatchMessageFormatter</b>	<a href="#">SerializeReply</a>	작업 반환 값/out 매개 변수로부터 나가는 <b>Message</b> 만들기



<b>IClientMessageFormatter</b>	<u>SerializeRequest</u>	작업 매개 변수로부터 나가는 <b>Message</b> 만들기
<b>IClientMessageFormatter</b>	<u>DeserializeReply</u>	들어오는 <b>Message</b> 를 반환 값/out 매개 변수로 변환

## Serialization

메시지 내용을 설명하기 위해 메시지 계약 또는 매개 변수를 사용하는 경우에는 항상 serialization 을 사용하여 .NET Framework 형식과 XML Infoset 표현 간의 변환을 수행해야 합니다. Serialization 은 WCF 의 다른 위치에서 사용됩니다. 예를 들어 **Message** 에는 개체로 deserialize 된 메시지의 본문 전체를 읽는 데 사용할 수 있는 제네릭 GetBody 메서드가 있습니다.

WCF 는 매개 변수 및 메시지 부분의 serialize 와 deserialize 를 위해 "즉시 사용할 수 있는" 두 가지 serialization 기술, 즉, **DataContractSerializer** 및 **XmlSerializer** 를 지원합니다. 그뿐 아니라 사용자 지정 serializer 도 작성할 수 있습니다. 그러나 WCF 의 다른 부분(예: 제네릭 **GetBody** 메서드 또는 SOAP 오류 serialization)은 XmlObjectSerializer 서브클래스(XmlSerializer 가 아닌 **DataContractSerializer** 및 NetDataContractSerializer)만 사용하도록 제한되거나, 심지어 **DataContractSerializer** 만 사용하도록 하드 코드될 수도 있습니다.

**XmlSerializer** 는 ASP.NET 웹 서비스에서 사용되는 serialization 엔진입니다.

**DataContractSerializer** 는 새 데이터 계약 프로그래밍 모델을 인식하는 새 serialization 엔진입니다. **DataContractSerializer** 는 기본 선택이며, DataContractFormatAttribute 특성을 사용하는 작업별 기준으로는 **XmlSerializer** 를 사용하도록 선택할 수 있습니다.

**DataContractSerializerOperationBehavior** 및 XmlSerializerOperationBehavior 는 **DataContractSerializer** 와 **XmlSerializer** 각각에 대해 메시지 포맷터에서의 플러그 인을 담당하는 작업 동작입니다. **DataContractSerializerOperationBehavior** 동작은 실제로 **NetDataContractSerializer** 를 비롯하여 **XmlObjectSerializer** 에서 파생되는 모든 serializer 와 함께 작동할 수 있습니다. 자세한 내용은 독립 실행형 Serialization 사용에 설명되어 있습니다. 해당 동작은 **CreateSerializer** 가상 메서드 오버로드 중 하나를 호출하여 serializer 를 가져옵니다. 다른 serializer 를 플러그 인하려면 새 **DataContractSerializerOperationBehavior** 서브클래스를 만들고, 두 **CreateSerializer** 오버로드를 모두 재정의합니다.

## 5) Windows 이벤트 처리방식의 이해

.NET Framework에서의 이벤트는 대리자 모델에 기반합니다. 대리자 모델은 구독자는 공급자를 등록하고 공급자로부터 알림을 수신할 수 있도록 하는 관찰자 디자인 패턴을 따릅니다. 이벤트 전송자와 이벤트가 발생했음을 알리고, 이벤트 수신자는 해당 알림을 수신하고 그에 대한 응답을 정의합니다. 이 문서에서는 대리자 모델의 주요 구성 요소, 응용 프로그램에서 이벤트를 사용 하는 방법 및 코드에서 이벤트를 구현하는 방법에 대해 설명합니다.

Windows 8.x 스토어 응용 프로그램에서 이벤트 처리에 대한 자세한 내용을 보려면 [이벤트 및 라우트된 이벤트 개요 \(Windows 스토어 응용 프로그램\)](#) 을 참조하십시오.

### 이벤트

---

이벤트는 동작의 발생을 알리기 위해 개체에서 보내는 메시지입니다. 작업은 속성의 값을 변경하는 등 일부 다른 프로그램 논리에 의해 발생할 수도 있고, 단추 클릭 같은 사용자 상호 작용에 의한 것일 수도 있습니다. 이벤트를 발생시키는 개체를 *이벤트 전송자* 라고 합니다. 이벤트 전송자는 어떤 개체 또는 메서드가 발생시키는 이벤트를 수신 (처리) 할지 모릅니다. 이벤트는 일반적으로 이벤트 전송자의 멤버입니다. 예를 들어 `Click` 이벤트는 `Button` 클래스의 멤버이고, `PropertyChanged` 이벤트는 `INotifyPropertyChanged` 인터페이스를 구현 하는 클래스의 멤버입니다.

이벤트를 정의하기 위해서는 **event** (C#) 또는 **Event** (Visual Basic) 키워드를 이벤트 클래스의 시그니처에서 사용하고 이벤트 대리자의 형식을 지정합니다. 대리자는 다음 섹션에 설명되어 있습니다.

일반적으로 이벤트를 발생 시키려면 **protected** 및 **virtual** (C#) 또는 **Protected** 및 **Overridable** (Visual Basic) 라고 표시된 메서드를 추가합니다. 이 **OnEventName** 메서드의 이름을 짓습니다. 예 : `OnDataReceived` . 메서드는 이벤트 데이터 개체를 지정하는 매개 변수를 취해야 합니다. 이 메서드를 사용하면 파생된 클래스를 이용해 이벤트를 발생시키는 논리를 재정의할 수 있습니다. 파생된 클래스는 등록된 대리자가 이벤트를 수신할 수 있도록 해주는 기본 클래스의 **OnEventName** 메서드를 항상 호출해야 합니다.

## 대리자

---

대리자는 메서드에 대한 참조를 가지는 형식입니다. 대리자는 이것이 참조하는 메서드에 대한 반환 형식 및 매개 변수를 보여주는 서명을 사용하여 선언되고, 해당 서명과 일치하는 메서드만을 참조할 수 있습니다. 그렇기 때문에 대리자는 형식이 안전한 함수 포인터나 콜백과 같습니다. 대리자 선언으로 대리자 클래스를 충분히 선언할 수 있습니다.

대리자는 .NET Framework 에서 여러 용도로 사용됩니다. 이벤트의 컨텍스트에서 대리자는 이벤트 소스와 이벤트를 처리 하는 코드의 중간 (또는 포인터 같은 메커니즘) 입니다. 대리자와 이벤트의 연결은 이전 섹션의 예제에서와 같이 이벤트 선언에서 대리자 형식을 포함하는 방식으로 수행됩니다. 대리자에 대한 자세한 내용은 [Delegate](#) 클래스를 참조하십시오.

.NET Framework 에서는 대부분의 이벤트 시나리오를 지원하는 [EventHandler](#) 및 [EventHandler<TEventArgs>](#) 대리자를 제공합니다. 이벤트에 대한 데이터를 포함하지 않는 모든 이벤트에 대해 [EventHandler](#) 대리자를 사용합니다. 이벤트에 대한 데이터를 포함하는 모든 이벤트에 대해 [EventHandler<TEventArgs>](#) 대리자를 사용합니다. 이러한 대리자들은 반환 값으로 형식을 반환하지 않고 두 매개 변수(이벤트의 원본에 대한 개체 및 이벤트 데이터에 대한 개체) 를 취합니다..

대리자는 멀티캐스트로서, 하나 이상의 이벤트 처리 메서드에 대한 참조를 가질 수 있습니다. 자세한 내용은 [Delegate](#) 참조 페이지를 보십시오. 대리자는 이벤트 처리를 제어함에 있어 유연하고 정밀한 처리를 제공합니다. 대리자는 이벤트에 대해 등록된 이벤트 처리기 목록을 유지하여 이벤트를 발생시킨 클래스의 발송자 역할을 합니다.

[EventHandler](#) 및 [EventHandler<TEventArgs>](#) 대리자가 작동 하지 않는 시나리오의 경우, 대리자를 정의할 수 있습니다. 대리자를 정의해야 하는 시나리오, 예를 들어 제네릭을 인식할 수 없는 코드를 사용하여 작업 해야 하는 경우와 같은 시나리오는 매우 드뭅니다. 대리자는 선언할때 C#에선 **delegate** 를, Visual Basic에선 **Delegate** 키워드를 사용하여 표시합니다.

## 이벤트 데이터

---

이벤트와 관련된 데이터는 이벤트 데이터 클래스를 통해 제공될 수 있습니다. .NET Framework 은 응용 프로그램에서 사용할 수 있는 여러 이벤트 데이터 클래스를 제공합니다. 예를 들어, SerialDataReceivedEventArgs 클래스는 SerialPort.DataReceived 이벤트에 대한 이벤트 데이터 클래스에 해당합니다. .NET Framework 는 모든 이벤트 데이터 클래스에 대해 명명 패턴이 **EventArgs** 을 끝으로 하는 패턴을 따릅니다. 이벤트에 대한 대리자를 보는것 만으로 어떠한 이벤트 데이터 클래스가 어떤 이벤트에 연결되어 있는지 확인할 수 있습니다. 예를 들어, SerialDataReceivedEventHandler 대리자는 매개 변수 중 하나로 SerialDataReceivedEventArgs 클래스를 가집니다.

EventArgs 클래스는 모든 이벤트 데이터 클래스의 기본 형식입니다. EventArgs 는 이벤트에 연결된 모든 데이터가 없을때 사용하는 클래스 이기도 합니다. 다른 클래스에게 어떠한 데이터도 전달하지 않거나 뭔가 발생했다는 것을 알리는 이벤트를 만들고 싶은 경우, 대리자에서 두번째 변수로 EventArgs 클래스를 사용합니다. 어떠한 데이터도 제공되지 않은 경우 EventArgs.Empty 값을 전달할 수 있습니다. EventHandler 대리자는 매개변수로 EventArgs 클래스를 포함합니다.

사용자 지정된 이벤트 데이터 클래스를 만들려면, EventArgs 으로부터 파생된 클래스를 만든 후, 이벤트 관련 데이터를 전달하는데 필요한 멤버를 제공합니다. 일반적으로 .NET Framework 같은 명명 패턴을 사용하고 이벤트 데이터 클래스의 이름의 끝을 **EventArgs** 로 해야 합니다

## 이벤트 처리기

---

이벤트에 응답하려면, 이벤트 수신기에서 이벤트 처리기 메서드를 정의합니다. 이 메서드는 해당 이벤트의 대리자의 시그니처와 일치 해야 합니다. 이벤트 처리기는 사용자가 단추를 클릭한 후 사용자 입력 정보를 수집하는 등의 이벤트가 발생하는 경우 필요한 작업들을 수행합니다. 이벤트가 발생할 때 알림을 받기 위해서는 이벤트에 이벤트 처리기 메서드를 신청해야 합니다.

## 정적 및 동적 이벤트 처리기

---

.NET Framework에서는 구독자가 정적이나 동적으로 이벤트 알림을 등록할 수 있습니다. 정적 이벤트 처리기는 해당 이벤트가 처리되는 클래스의 전체 수명 동안 적용됩니다. 동적 이벤트 처리기는 프로그램 실행 도중 일반적으로 특정 조건부 프로그램 논리에 따라 명시적으로 활성화 및 비활성화됩니다. 예를 들어 특정 조건에서만 이벤트 알림이 필요하거나 응용 프로그램에서 여러 이벤트 처리기를 제공하고 런타임 조건에 따라 사용할 이벤트 처리기를 정의하는 경우 동적 이벤트 처리기를 사용할 수 있습니다. 이전 섹션의 예제에서는 이벤트 처리기를 동적으로 추가하는 방법을 보여줍니다.

## 여러 이벤트 발생시키기

---

클래스에서 여러 이벤트를 발생시키는 경우, 컴파일러가 각 이벤트 대리자 인스턴스에 대해 하나의 필드를 생성합니다. 이벤트 수가 많은 경우에는 각 대리자에 대한 필드의 저장 공간이 부족할 수도 있습니다. 이러한 경우, .NET Framework에서는 사용자가 이벤트 대리자를 저장하기 위해 선택한 다른 데이터 구조체와 함께 사용할 수 있는 이벤트 속성을 제공합니다.

이벤트 속성은 이벤트 선언과 이벤트 접근자로 구성됩니다. 이벤트 접근자는 사용자가 정의하는 메서드로서, 저장 데이터 구조에서 이벤트 대리자 인스턴스를 추가하거나 제거할 수 있게 해줍니다. 각 이벤트 대리자는 호출되기 전에 검색되어야 하기 때문에 이벤트 속성은 이벤트 필드보다 속도가 느리다는 것을 알아두시기 바랍니다. 이벤트 속성을 사용하면 사용 가능한 메모리는 큰 대신 속도는 그만큼 느립니다. 클래스에서 자주 발생되지 않는 이벤트를 정의하는 경우에는 이벤트 속성을 구현하는 것이 좋습니다.

## SDK 의 이해와 어셈블리

---

### 1) SDK 를 이용한 개발

Windows SDK(소프트웨어 개발 키트)에는 Windows 운영 체제에서 실행되는 앱을 만들 때 사용할 수 있는 헤더, 라이브러리 및 도구 모음이 포함되어 있습니다. Windows SDK 와 선택한 개발 환경에서 웹 기술(HTML5, CSS3 및 JavaScript), 네이티브(C++) 및 관리(C#, Visual Basic) 코드를 사용하여 Windows 스토어 앱(Windows 8 전용)을 개발하거나 네이티브(Win32/COM) 프로그래밍 모델을 사용하는 데스크톱 응용 프로그램이나 관리(.NET Framework) 프로그래밍 모델을 사용하는 데스크톱 응용 프로그램을 개발할 수 있습니다.

SDK 를 사용하여 Windows 8, Windows 7, Windows Vista, Windows Server 2012, Windows Server 2008 R2, Windows Server 2008 운영 체제를 대상으로 하는 응용 프로그램을 빌드할 수 있습니다.

또한 Windows SDK 에는 앱을 Windows 8 인증 프로그램과 Windows 7 로고 프로그램 기준으로 테스트하기 위한 **Windows ACK(앱 인증 키트) 2.2** 가 포함되어 있습니다.

SDK(Software Development Kit ; 윈도우 2003까지는 Platform SDK, 윈도우 비스타부터는 Windows SDK라 부름)란 마이크로소프트사에서 윈도우 응용 프로그램 제작을 위해 배포하는 컴파일러를 비롯한 각종 개발툴, 헤더 파일, 라이브러리 파일, 도움말, 소스 코드의 집합을 말합니다.

SDK는 마이크로소프트사의 웹사이트에서 무료로 받을 수 있으며, 새로운 윈도우 버전이 발표될 때마다 업데이트 됩니다.

SDK를 이용해 프로그래밍한다는 것은 언어로 개발할 때 해당 SDK의 API를 직접 호출해서 프로그램을 구현함을 뜻합니다.

API를 직접 다루기 때문에 세부 제어가 가능하고, 윈도우 운영체제가 제공하는 모든 기능을 사용할 수 있습니다.

### 2) #define 와 #undef 등의 전처리기

컴파일러에는 별도 전처리기 없지만이 절에서 설명 하는 지시문 하나 것 처럼 처리 됩니다. 도움말에서 조건부 컴파일을 사용 하 여 하. C 및 C++ 지시문과 달리 이들 지시문을 사용하여 매크로를 만들 수는 없습니다.

전처리기 지시문에는 한 줄에 명령이 하나만 있어야 합니다. 전처리기 문의 종류는 다음과 같습니다.

#if

#else

#elif

#endif

#define

#undef

#warning

#error

#line

#region

#endregion

#pragma

#pragma warning

#pragma checksum

#### ◇ #define

**#define** 지시문은 C 및 C++에서 일반적으로 수행되는 것처럼 상수 값을 선언하는 데 사용할 수 없습니다. C#의 상수는 클래스 또는 구조체의 정적 멤버로 정의하는 것이 좋습니다. 이러한 상수가 여러 개 있는 경우 별도의 "Constants" 클래스를 만들어 저장하는 것이 좋습니다.

기호를 사용하여 컴파일 조건을 지정할 수 있습니다. 이 기호를 `#if` 또는 `#elif` 로 테스트할 수 있습니다. 또한 **conditional** 특성을 사용하여 조건부 컴파일을 수행할 수도 있습니다.

기호를 정의할 수는 있지만 해당 기호에 값을 대입할 수는 없습니다. 전처리기 지시문이 아닌 모든 명령을 사용하려면 **#define** 지시문이 먼저 파일에 나타나야 합니다.

또한 `/define` 컴파일러 옵션으로 기호를 정의할 수도 있습니다. 또한 `#undef` 로 기호를 정의하지 않을 수도 있습니다.

`/define` 또는 **#define** 으로 정의한 기호는 같은 이름의 변수와 충돌하지 않습니다. 즉, 변수 이름을 전처리기 지시문에 전달해서는 안 되며 기호는 전처리기 지시문으로만 계산할 수 있습니다.

**#define** 을 사용하여 만든 기호의 범위는 해당 기호가 정의된 파일입니다.

#### ✧ **#undef**

**#undef** 를 사용하면 기호의 정의를 해제할 수 있습니다. 이때 정의 해제된 기호를 `#if` 지시문의 식으로 사용하면 식이 **false** 가 됩니다.

기호는 `#define` 지시문 또는 `/define` 컴파일러 옵션으로 정의할 수 있습니다. 지시문이 아닌 모든 문을 사용하려면 **#undef** 지시문이 먼저 파일에 나타나야 합니다.

이런 전처리기 지시문에 대하여는 전체 내용을 참고하여 알아봅니다.

### 3) 어트리뷰트

Attribute 는 클래스안에 메타정보를 포함 시킬 수 있는 기술로서 클래스, 데이터 구조, 열거자 그리고 어셈블리와 같은 프로그래밍적 요소들의 실행 시 행동에 대한 정보를 기술하고 .NET 내부에서만 사용되는 언어라고도 할 수 있습니다.



프로그래밍 요소에서 .NET Framework 에서는 어셈블리, 모듈, 클래스, 구조체, 열거형 변수, 생성자, 메소드, 프로퍼티, 필드, 이벤트, 인터페이스, 파라미터, 반환 값, 델리게이트 같은 요소를 사용할 수 있으며, 트리뷰트 역시 클래스의 하나의 종류입니다.

클래스가 아닐 경우 어트리뷰트의 문법은 다음과 같습니다.

```
[attribute(positional_parameter, name_parameter = value, ...)]
```

파라미터에는 두 종류가 있는데 하나는 위치지정 파라미터로 반드시 들어와야 합니다.

명명파라미터(name\_parameter)는 꼭 필요하지는 않은 구조를 가지면 어트리뷰트 안에 추가적인 정보를 넣을 때 이용하는 경우도 있습니다.

#### ✧ Custom Attribute 의 정의

- 어트리뷰트의 범위 지정

앞에서 Conditional Attribute 는 메소드에만 붙일 수 있다고 했다. 그렇다면 사용자가 만든 어트리뷰트에 대해 어떤 곳에서만 붙일 수 있게 하는 방법은 없을까?  
AttributeUsage 를 이용해서 사용자가 정의한 데이터 형에만 어트리뷰트를 붙일 수 있습니다.

```
[AttributeUsage(AttributeTargets.Method)]
```

AttributeUsage 역시 Attribute 이며 사용자가 정의할 어트리뷰트 앞에 사용됨으로서 범위를 지정 할 수 있다. Method 이외에 AttributeTargets.Class, AttributeTargets.Delegate, AttributeTargets.Interface, AttributeTargets.Property, AttributeTargets.Constructor 등등을 사용 할 수 있습니다.

만약 여러 개의 데이터 형에 붙일려면 '|'를 이용하면 됩니다.

[AttributeUsage(AttributeTargets.Method | AttributeTargets.Delegate)]

```
public class AdditionalInfo { ... }
```

또한 모든 데이터 형에 붙이는 것을 가능하게 하려면 `AttributeTargets.All` 이라고 해주면 됩니다.

#### - 어트리뷰트 클래스 선언

간단히 클래스에 대한 제작자, 업데이트 날짜, 최신 버전을 다운 받을 수 있는 곳 등의 정보를 담을 수 있는 어트리뷰트를 만들어 보도록 하겠습니다. 다른 클래스를 만드는 것처럼 비슷하게 클래스를 작성 하면 어트리뷰트가 구현됩니다. 모든 어트리뷰트는 `System.Attribute` 로 부터 상속 받습니다. 즉 보통 클래스를 선언하는 똑 같은 방법으로 선언하고 다만 `System.Attribute` 로부터 상속을 받으면 그 클래스가 어트리뷰트가 되는 것입니다. 일반 클래스와 구분하기 위해 접미사로 'Attribute'를 붙일 것을 권고합니다.

어트리뷰트도 클래스 이므로 생성자가 존재 합니다. 다만 어트리뷰트는 하나의 생성자만 가질 수 있습니다. 즉 생성자 오버로딩이 불가능 합니다. 그럼 추가로 받아야 하는 정보들은 어떻게 처리 할까? 어트리뷰트에서는 생성자에서 꼭 집어 넣어야 하는 데이터는 위치지정 파라미터로 받고 부가적인 데이터는 명명 파라미터를 통해 받아 들임으로서 문제를 해결합니다.

## 4) C#에서의 포인터

`unsafe` 컨텍스트 내에서는 타입이 포인터 타입, 값 타입 또는 참조 타입이 될 수 있습니다. 포인터 형식 선언은 다음 형식 중 하나를 사용합니다

```
type* identifier;  
void* identifier; //allowed but not recommended
```

포인터 형식은 다음과 같은 형식일 수 있습니다.

- sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal 또는 bool
- 임의의 열거형
- 임의의 포인터 형식
- 관리되지 않는 형식의 필드만 포함하는 임의의 사용자 정의 구조체 형식

포인터 형식은 개체에서 상속되지 않으며 포인터 형식과 **object** 는 서로 변환되지 않습니다. 또한 boxing 과 unboxing 은 포인터를 지원하지 않습니다. 그러나 다른 포인터 형식 간의 변환 및 포인터 형식과 정수 계열 형식 사이의 변환은 허용됩니다.

동일한 선언에서 여러 포인터를 선언하는 경우 별표(\*)는 기본 형식에만 함께 사용되고 각 포인터 이름의 접두사로는 사용되지 않습니다. 예를 들면 다음과 같습니다.

```
int* p1, p2, p3;    // Ok  
int *p1, *p2, *p3;  // Invalid in C#
```

개체 참조는 포인터가 해당 개체 참조를 가리키는 경우에도 가비지 수집될 수 있으므로 포인터는 참조나 참조가 들어 있는 구조체를 가리킬 수 없습니다. 가비지 수집기는 포인터 형식에서 개체를 가리키는지 여부를 추적하지 않습니다.

myType\* 형식의 포인터 변수 값은 myType 형식의 변수 주소입니다. 다음은 포인터 형식 선언의 예제입니다.

예제	설명
int* p	p 는 정수에 대한 포인터입니다.
int** p	p 는 정수에 대한 포인터를 가리키는 포인터입니다.
int*[] p	p 는 정수에 대한 포인터의 1 차원 배열입니다.
char* p	p 는 문자에 대한 포인터입니다.
void* p	p 는 알 수 없는 형식에 대한 포인터입니다.

포인터 간접 참조 연산자 \*를 사용하면 포인터 변수가 가리키는 위치의 내용에 액세스할 수 있습니다. 예를 들어, 다음 선언을 참조하십시오.

```
int* myVariable;
```

여기서 \*myVariable 식은 myVariable 에 포함된 주소에 있는 **int** 변수를 가리킵니다.

[fixed 문\(C# 참조\)](#) 및 [포인터 변환\(C# 프로그래밍 가이드\)](#) 항목에 포인터에 대한 몇 가지 예제가 나와 있습니다. 다음 예제는 **unsafe** 키워드 및 **fixed** 문에 대한 필요성과 정수 포인터를 증분하는 방법을 보여줍니다. 이 코드를 실행하려면 콘솔 응용 프로그램의 주 함수에 붙여 넣습니다. ([프로젝트 디자이너](#)에서 안전하지 않은 코드를 사용하도록 설정하십시오. 메뉴 모음에서 **프로젝트**, **속성**을 선택한 다음 **빌드** 탭에서 **안전하지 않은 코드 허용**을 선택합니다.)

void\* 형식의 포인터에는 간접 참조 연산자를 적용할 수 없습니다.그러나 캐스트를 사용하여 void 포인터를 다른 포인터 형식으로 변환하거나 반대로 변환할 수 있습니다.

포인터는 **null** 일 수 있습니다.null 포인터에 간접 참조 연산자를 적용할 때 발생하는 동작은 구현에 따라 다릅니다.

메서드 사이에 포인터를 전달하면 정의되지 않은 동작이 발생할 수 있다는 사실에 주의해야 합니다.Out 또는 Ref 매개 변수를 통해, 또는 함수 결과로 지역 변수에 포인터를 반환하는 경우를 예로 들 수 있습니다.fixed 블록에서 포인터가 설정되면 이 포인터가 가리키는 변수의 고정 상태가 해제될 수 있습니다.

다음 표에서는 안전하지 않은 컨텍스트에서 포인터에 대해 수행할 수 있는 연산자와 문을 보여 줍니다.

연산자/문	기능
*	포인터 간접 참조를 수행합니다.
->	포인터를 통해 구조체 멤버에 액세스합니다.
[]	포인터를 인덱싱합니다.
&	변수 주소를 가져옵니다.
++ 및 --	포인터를 증가 및 감소시킵니다.
+ 및 -	포인터 연산을 수행합니다.
==, !=, <, >, <=, >=	포인터를 비교합니다.
stackalloc	스택에 메모리를 할당합니다.
fixed 문	해당 주소를 찾을 수 있도록 임시로 변수를 고정합니다.

## 5) 어셈블리의 개념과 아키텍처

.NET Framework 는 한 가지 버전의 공용 언어 런타임과 형식 라이브러리를 구성하는 약 20 개의 .NET Framework 어셈블리로 구성되어 있습니다. 이러한 .NET Framework 어셈블리는 런타임에서 하나의 단위로 취급됩니다. 예를 들어, .NET Framework 버전 1.0 은 런타임 버전 1.0.3705 과 .NET Framework 어셈블리 버전 1.0.3300.0 으로 구성됩니다. 특정 버전의 .NET Framework 를 지원하는 응용 프로그램이나 구성 요소는 런타임 출시 버전과 .NET Framework 어셈블리 출시 버전에서 작동됩니다.

기본적으로 런타임은 프로세스에 로드된 런타임 버전에 속하는 .NET Framework 어셈블리 버전만 로드합니다. 응용 프로그램이 시작되면 런타임에 의해 실행된 코드에 있는 형식에 대한 참조는 모두 프로세스에 로드된 런타임과 버전 번호가 같은 .NET Framework 어셈블리로 경로가 변경됩니다. 이러한 .NET Framework 어셈블리 통합을 통해, 특별히 지정되지 않는 한 런타임이 다른 버전의 .NET Framework 에서 어셈블리를 로드하는 일이 없어집니다.

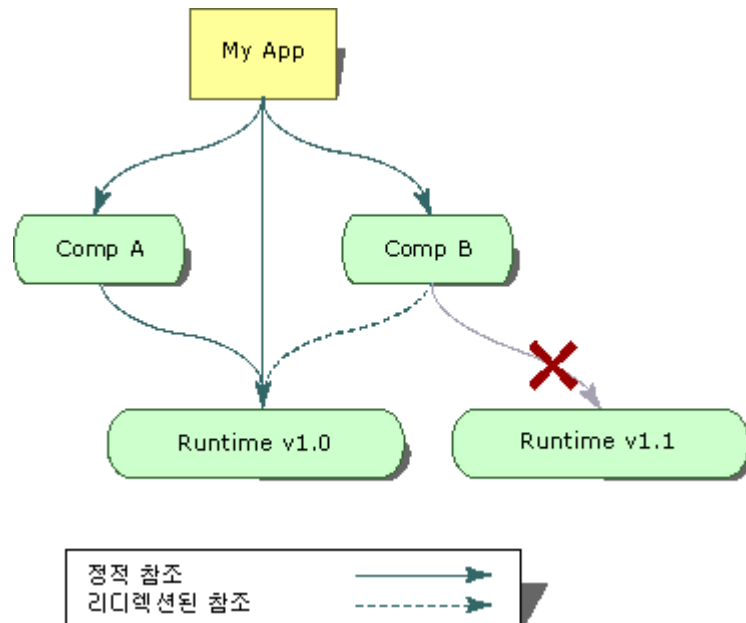
### 어셈블리 통합 및 구성 요소

---

응용 프로그램에서는 사용하는 런타임 버전을 결정합니다. 통합 과정에는 응용 프로그램이 사용할 수 있는 구성 요소도 포함됩니다. 응용 프로그램은 특정 버전의 런타임에서 실행되도록 하기 위해 사용하는 구성 요소를 리디렉션할 수 있습니다. 특정 버전의 런타임으로 컴파일한 구성 요소가 다른 버전을 사용하도록 리디렉션될 수 있습니다.

다음 그림에서 응용 프로그램 MyApp 는 두 개의 구성 요소 Comp A 와 Comp B 를 사용합니다. MyApp 와 Comp A 는 런타임 버전 1.0 으로 빌드되었으므로 런타임 버전 1.0 에 대한 정적 참조를 포함합니다. 구성 요소 Comp B 는 런타임 버전 1.1 과 함께 제공된 .NET Framework 어셈블리에 대한 정적 참조를 포함하지만 통합성을 유지하기 위해 런타임 버전 1.0 과 함께 제공된 .NET Framework 어셈블리를 사용하여 실행되도록 리디렉션됩니다.

어셈블리 통합을 통한 어셈블리 바인딩 리디렉션



응용 프로그램에서는 응용 프로그램 구성 파일에 있는 어셈블리에 대한 바인딩 리디렉션 정보를 제공하여 이러한 기본 동작을 재정의할 수 있습니다. 기본 동작을 재정의하면 다른 .NET Framework 어셈블리가 로드되는 방식에 영향을 주지 않으면서 특정 버전의 .NET Framework 어셈블리를 사용하도록 런타임을 리디렉션할 수 있습니다.

공용 언어 런타임에서는 다음 정보를 사용하여 응용 프로그램을 위해 로드할 런타임 버전을 결정합니다.

- 사용할 수 있는 런타임 버전
- 응용 프로그램이 지원하는 런타임 버전

### 지원되는 런타임 버전

런타임에서는 응용 프로그램 구성 파일과 PE(이식 가능한 실행) 파일 헤더를 사용하여 응용 프로그램이 지원하는 런타임 버전을 확인합니다. 응용 프로그램 구성 파일이 없으면 런타임에서는 응용 프로그램의 PE 파일 헤더에 지정된 런타임 버전(사용 가능한 경우)을 로드합니다.

응용 프로그램 구성 파일이 있는 경우에는 다음 프로세스의 결과에 따라 로드할 런타임 버전을 결정합니다.

1. 런타임은 응용 프로그램 구성 파일의 <supportedRuntime> 요소 요소를 확인합니다. **<supportedRuntime>** 요소에 지정되어 있는 지원되는 런타임 버전이 하나 이상 있으면 런타임은 첫 번째 **<supportedRuntime>** 요소에서 지정하는 런타임 버전을 로드합니다. 이 버전을 사용할 수 없으면 런타임은 다음 **<supportedRuntime>** 요소를 확인하고 지정된 런타임 버전을 로드하려고 시도합니다. 이 런타임 버전을 사용할 수 없으면 그 다음 **<supportedRuntime>** 요소를 확인합니다. 지정된 버전을 사용할 수 없으면 런타임은 런타임 버전을 로드하는 데 실패하고 사용자에게 메시지를 표시합니다(3 단계 참조).
2. 런타임은 응용 프로그램 실행 파일의 PE 파일 헤더를 읽습니다. PE 파일 헤더에 지정된 런타임 버전을 사용할 수 있으면 해당 버전을 로드합니다. 지정된 런타임 버전을 사용할 수 없으면 런타임은 PE 헤더의 런타임 버전과 호환된다고 Microsoft 에서 확인한 런타임 버전을 검색합니다. 해당 버전을 찾을 수 없으면 프로세스는 3 단계로 계속됩니다.
3. 런타임은 응용 프로그램에서 지원하는 런타임 버전을 사용할 수 없다는 메시지를 표시합니다. 런타임은 로드되지 않습니다.

#### 참고

레지스트리 키 HKLM\Software\Microsoft\NETFramework 에 있는 NoGuiFromShim 값을 사용하거나 환경 변수 COMPLUS\_NoGuiFromShim 을 사용하여 이 메시지가 표시되지 않게 할 수 있습니다. 예를 들어, 무인 설치나 Windows 서비스 같이 일반적으로 사용자와 상호 작용을 하지 않는 응용 프로그램의 경우 이 메시지를 표시하지 않을 수 있습니다. 이 메시지를 표시하지 않도록 설정하면 런타임은 이벤트 로그에 해당 메시지를 씁니다. 컴퓨터에 있는 모든 응용 프로그램에 대해 이 메시지를 표시하지 않으려면 레지스트리 값 NoGuiFromShim 을 1로 설정하고, 특정한 사용자 컨텍스트에서 실행되는 응용 프로그램에 대해 이 메시지를 표시하지 않으려면 COMPLUS\_NoGuiFromShim 환경 변수를 1로 설정합니다.

#### 참고

런타임 버전이 로드된 후에 어셈블리 바인딩 리디렉션을 통해 개별 .NET Framework 어셈블리의 다른 버전이 로드되도록 지정할 수 있습니다. 이러한 바인딩 리디렉션은 리디렉션되는 특정 어셈블리에만 영향을 줍니다.

## 6) Visual Studio Debug 의 이해와 예외처리

프로그램에서 문제가 발생하면 이를 해결해야 하기 위한 도구가 필요합니다. 그 도구를 Visual Studio 에서 제공하고 있습니다. 그리고 프로그램에 문제가 있다면 이를 찾는 것을 "버그(Bug)를 찾는다"라고 하며, 이 버그는 프로그램의 문제가 발생한 부분을 뜻합니다.

버그는 사실 벌레를 뜻하는 말이지만, 개발자들에게는 심각한 부분일 수 있는 부분이 바로 그 버그입니다. 버그를 찾지 못하면 프로그램을 실행할 수 없기 때문입니다.

이 버그를 잡기 위한 일련의 과정을 디버깅(Debugging)이라고 합니다.

아무리 숙련된 프로그램 개발자여도 실수를 하기 마련이고 이 실수가 극히 적기 때문에 숙련된 사람이라고도 하며, 이 실수로 인한 문제 해결을 빨리 하기도 합니다. 그렇지만 가끔 오타 하나 찾는 것도 힘들 수도 있는 것이 사실입니다.

빨리 프로그램을 만든다고 좋은 개발자가 아니라 생각합니다. 바로 문제 없는 프로그램을 만드는 개발자가 좋은 개발자가 아닐까 합니다. 그렇지만 상황이 그렇지 않기 때문에 우리는 제한된 시간에 프로그램을 개발하는 상황이 많습니다. 이런 상황에서 빨리 문제를 해결해야 합니다.

그럼 이 문제의 개념을 접근할 때 우리는 프로그램 개발자들이 만든 소스 코드의 문제인지 아니면 외부 환경의 요인으로 인해 프로그램이 문제가 발생했는지를 찾는 것이 중요합니다.

Debug 는 바로 개발자들의 만든 소스 코드에서 찾는 것을 이야기 합니다. 개발자들이 제한된 시간에 소스 코드를 작성하다 보니 크고 작은 실수를 하게 되거나, 가끔 논리적 오류를 실수로 넘어가는 경우가 있을 수 있습니다. 즉 소스 코드의 논리적 오류를 찾는 것이 바로 Debug 입니다.

그렇다면 오타를 친 것은 어떻게 찾을 것인지는 Visual Studio Tool 거의 해주고 있습니다.

즉 오타의 경우 Visual Studio Tool 의 인텔리 센스가 어느 정도 찾아주기 때문에 Debug 는 개발자의 논리적인 실수와 문제를 잡는데 사용하는 기술을 뜻합니다.

이제 외부의 요인으로 프로그램의 문제가 발생했다면 어떨까요? 프로그램 잘 실행되고 있는데 천재 지변으로 네트워크의 문제가 발생했다고 하면 네트워크 문제로 프로그램이 정지되거나 문제가 발생하면 안되지 않을까 합니다. 즉 이런 부분을 처리하는 것이 바로 예외 처리라고 합니다.

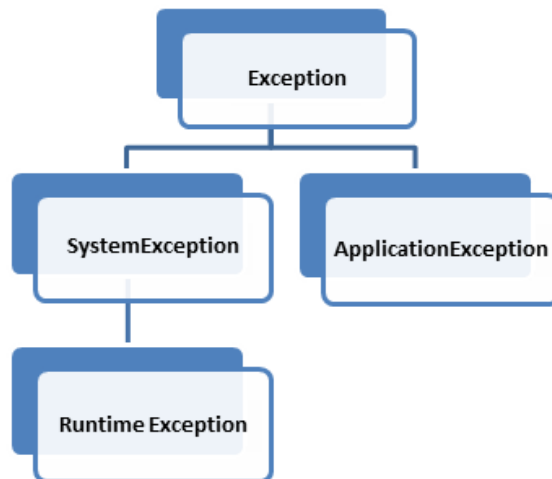


디버그	예외처리
개발자의 논리적 실수 문제 찾기	외부 요인에 의한 상황에 대한 문제
Tool을 이용하여 해결	Exception 을 이용하여 해결
코드 단계	실행단계

디버깅은 Tool 사용이므로 우선 예외처리부터 배우도록 하겠습니다.

#### ☆ 예외처리(Exception) 배우기

예외 처리는 프로그램이 실행 중에 발생하는 상황으로 만에 하나 일어날까 한 상황을 처리하는 것을 기준으로 하는 것이 좋습니다. 물론 실제 그렇지 않지만 예외 처리를 꼭 하도록 하는 코드 작성을 버릇처럼 하면 실행 중에 발생하는 상황을 사전에 방지 할 수 있습니다. 실행 중에 발생하는 환경에는 외부 요인도 있지만 사실 개발자가 데이터의 크기를 너무 작게 설정하여 나중에 너무 많은 데이터가 저장되면 문제가 발생하여 프로그램이 정지 되기도 합니다. 따라서 이런 문제의 해결은 디버그 보다 실행 환경에서 잡아주기 위하여 예외 처리 하는 것을 꼭 해야 합니다. 쉽게 말해 개발자가 논리적으로도 발생할 수 상황에 프로그램에 문제 없이 계속 실행되게 한다거나 하는 등의 조치를 취하는 방법도 있습니다.



이제 이 예외 처리를 위한 방법을 배우겠습니다.

### try ~ catch 구문

예외 처리를 하기 위하여 우리는 한가지 구문을 배웁니다. 바로 try ~ catch 구문입니다. 이 구문은 try 의 블록 안에는 정상적으로 처리되는 코드를 작성하고 catch 블록에는 문제가 발생할 시 즉 예외가 발생할 때를 위한 구문을 작성합니다.

```
try
{
    // 일반적인(정상)일 경우 처리 하는 부분
}
catch (Exception e)
{
    // 예외가 발생할 경우 처리하는 부분
}
```

너무 어렵지 않습니다. 쉽습니다. 단 어려운 부분은 바로 catch 부분입니다. 보통 코드는 try 괄호 안에 작성하면 되고 예외가 발생할 경우에만 catch 구문을 작성하면 됩니다.

catch 구문안에는 Exception 에 정의된 예외를 기술하는데 이 예외는 사실 매우 많습니다.

모든 System.Exception 이란 곳에서 나오는데 이 곳에서는 예외 상황에 대하여 많은 것들을 가지고 있습니다. 즉 상황에 맞는 예외 사항을 처리하는 구문을 넣는데 이 예외 처리를 하는 것이 너무 다양하다 보니 이것을 모두 사용할 수 없습니다. 예를 들어 Database 의 예외 처리를 하려면 이 Database 를 파일을 읽다가 문제가 생기면 파일 예외 처리를 해야 합니다. 그래서 사실 Exception 이란 녀석이 최상의 조상에 해당되어서 해당 예외 처리 대신 Exception 을 사용하는 경우가 많습니다.

그래서 예외처리를 할 때에는 먼저 해당 문제에 대한 예외처리를 하고 그 다음 조상의 예외 처리를 하는 것이 좋습니다.

다음은 그 예 입니다.

```
try
{
    // 일반적인(정상)일 경우 처리 하는 부분
}
catch (SQLException ex)
{
    // SQL Database 전용 예외처리 부분
}
catch (Exception e)
{
    // 모든 예외가 발생할 경우 처리하는 부분
}
```

이렇게 예외 처리를 할 때에는 먼저 해당 전용 예외 처리를 구현하고 그 다음 일반적인 모든 예외 처리를 할 수 있도록 구현합니다. catch 는 사실 여러 개를 사용해도 되기 때문입니다.

### *finally*

finally 는 catch 블록 다음에 사용하는 것으로 이 부분은 try 의 부분이 실행되건 catch 부분이 실행되건 무조건 실행되는 블록을 뜻합니다. 이 부분에서 사실 공통된 또는 꼭 실행되는 부분을 작성합니다.

```
try
{
    // 일반적인(정상)일 경우 처리 하는 부분
}
catch (SQLException ex)
{
    // SQL Database 전용 예외처리 부분
}
catch (Exception e)
{
    // 모든 예외가 발생할 경우 처리하는 부분
}
finally
{
    // try 또는 catch 다음 무조건 처리되는 부분
}
```

### *throw*

throw 는 예외가 발생하지 않았지만 직접 예외를 발생할 때 사용합니다. 즉 사용자 정의 예외 처리를 할 경우에 사용합니다.

```
try
{
    // 일반적인(정상)일 경우 처리 하는 부분
    throw new // 사용자 정의 예외처리(Exception 호출함)
}
catch (Exception e)
{
    // 모든 예외가 발생할 경우 처리하는 부분
}
```

간단히 예외 처리에 대하여 알아 보았는데 사실 예외 처리는 보험이라고 생각하면  
좋습니다. 보험을 가입할 때 언제 이 보험을 쓸지 모르지만 막상 그 상황에 되면 보험을  
가입한 것이 매우 좋을 것입니다. 대표적으로 자동차 보험은 1년 무사고일 경우 보험을  
돌려받지 못하지만 사고가 났을 때에는 매우 유용하기 때문입니다. 보험이라 생각하시고  
꼭 코드에 작성하기를 권장하지만, 보험이기 때문에 너무 무리하게 많이 사용하거나  
적절히 사용하는 것이 좋습니다. 너무 무분별하게 사용하는 것은 그리 권장하지 않습니다.

## 7) Visual Studio Debugging 기술 익히기

이제 프로그램적으로 해결하는 예외 처리를 알아보았으면 이제 틀 사용법을 알아보겠습니다.

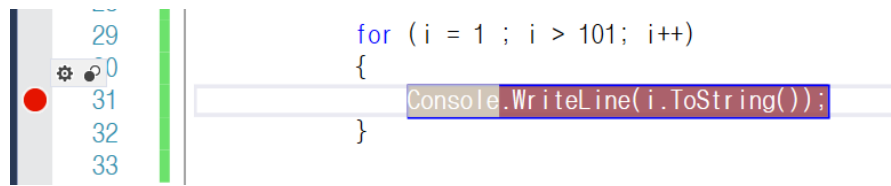
사실 Visual Studio 의 최대 장점 중에 하나가 이 Debugging 기술이 뛰어나기 때문입니다. 몇 가지 기본적인 기술을 알아보고 실제 이 기술을 잘 익혀서 사용하면 문제해결에 많은 도움이 될 것입니다.

### ❖ Debugging 의 기초 – 디버그 메뉴와 중단점

Visual C#에서 디버깅을 하기 위해서는 가장 먼저 해야 하는 것은 바로 "중단점(Break Point)"를 설정하는 것입니다. 중단점은 프로그램의 실행 중에 우리가 문제가 있다고 하는 곳에 설정하는 곳에서 잠시 Visual Studio 가 멈추게 됩니다. 이 중단점은 프로그램 실행 중에 멈춤 부분부터 문제를 찾기 시작하는 것입니다. 이 중단점은 문제가 있을 만한 부분에 중단점을 설정하는 것으로 단축키와 메뉴와 마우스를 통한 중단점 설정을 할 수 있습니다. 사실 처음에는 메뉴를 이용하고 익숙해 지면 마우스 보다 키보드의 단축키를 많이 사용합니다. 단축키는 "F9"이며 메뉴에는 "디버그(D)"메뉴를 선택하고 "중단점 설정/해제(G) F9"를 선택하면 됩니다. 이 때 마지막으로 선택한 소스 코드의 맨 앞에서 중단점이 설정됩니다.

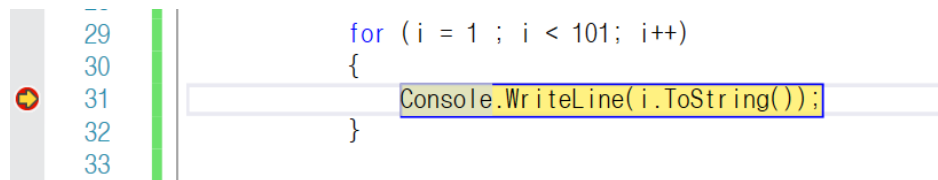


중단점을 설정하고 해제 하는 방법은 같으며 설정한 모습은 다음과 같습니다.

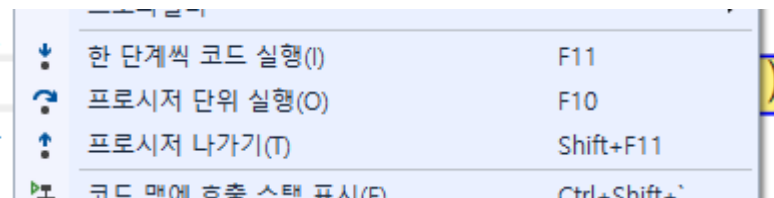


해제를 하면 편집기 화면 맨 앞에 빨간 점이 사라집니다.

디버깅을 설정했으면 이제 디버깅을 시작해야 하는데 이 방법은 디버그 메뉴에서 디버깅 시작이나 단축키로는 "F5"를 선택합니다. 그 외에 Visual Studio 화면의 Tool bar 라는 곳에서도 사용할 수 있으며 처음에는 메뉴를 사용하지만 익숙해 지면 바로 단축키를 사용합니다.

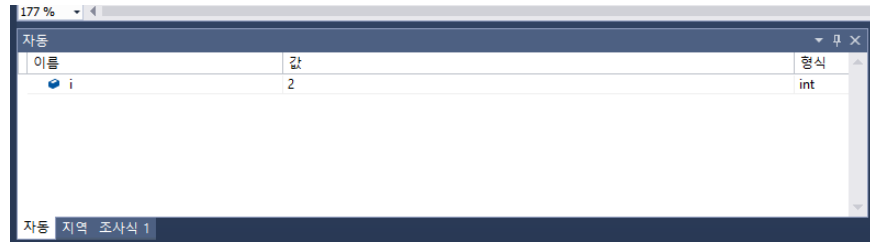


바로 그림에 보는 것처럼 화면에 노란색으로 표시되며 중단점 설정에 있는 곳에 프로그램을 잠시 중지하고 있는 것입니다. 이를 다시 한 줄씩 실행하는 방법은 메뉴에서 "한 단계씩 코드 실행" 또는 단축키로 "F11"을 사용하여 프로그램을 한 줄씩 실행하면서 문제점을 찾을 수 있습니다.

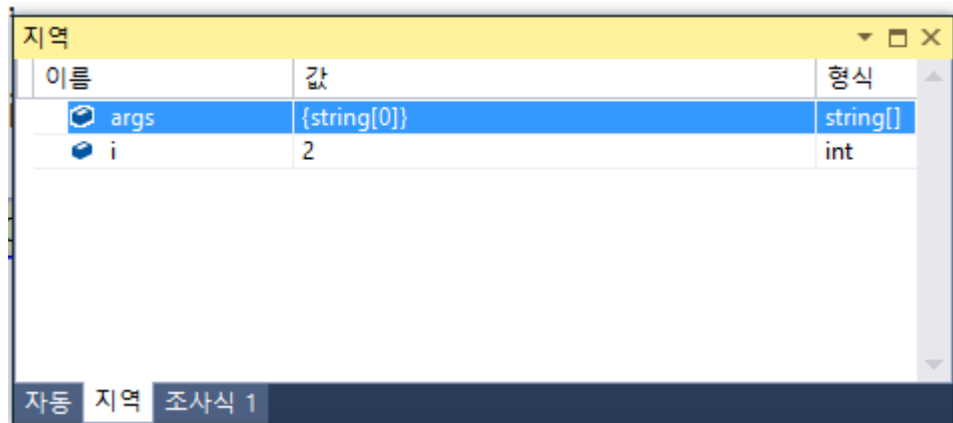


한 단계씩 코드 실행은 그 블록 안에서 한 줄씩 하는 방법이고 프로시전 단위로 하는 방법은 한 줄씩 실행하는 것입니다. 단계하고 조금 다른 것이 다음 명령어가 있는 곳으로 이동한다는 것이 다릅니다.

그럼 디버깅을 하면서 우리가 해야 할 것은 문제 해결인데, 어떤 것을 찾아야 할 까요 여기서 한번 생각해 보면 바로 `int i` 의 값이 변경되는 것을 찾는 것입니다. 이 프로그램의 소스는 `i` 를 순차적으로 출력하는 것으로 `i` 의 변경된 값을 확인 하는 방법 입니다. 이 방법은 Visual Studio 창에 "자동"이라는 것이 있으며 여기서 우리가 원하는 `i` 의 변경된 값을 볼 수 있습니다.



이 창에서는 "자동"으로 현재 블록에 있는 변수의 값과 형식, 이름을 확인하여 표시 줍니다. "지역"은 그 블록 안에 있는 모든 변수의 값을 표시해 주며, "조사식 1"에서는 수동으로 우리가 변수의 값을 이름에 입력하면 그 값을 표시해 줍니다. 이 값들은 모두 실시간으로 값이 변경되면 표시를 해 주기 때문에 지속적으로 볼 수 있는 장점이 있습니다.

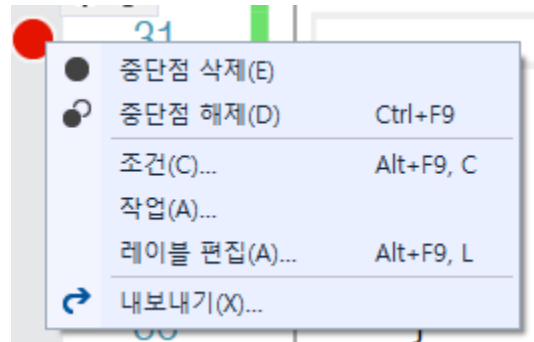


그럼 여기까지가 기본적인 디버깅을 하기 위한 방법을 본 것입니다.

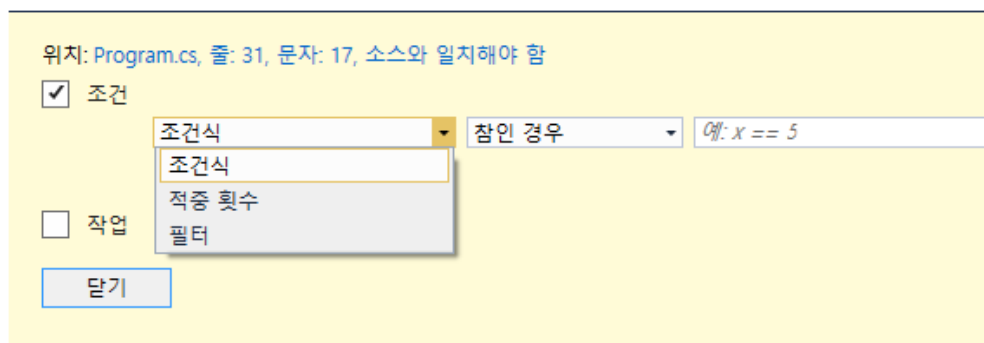
#### ✧ Debugging 의 활용과 Tip

디버깅을 하기 위한 기본적인 중단점을 배웠다면 소스 코드에서 반복문의 경우 100 번 반복을 하면서 60 번 정도에 문제가 발생했다면 어떻게 멈출까요? 이럴 때 사용하는 것이

중단점에 조건을 주는 것입니다. 그렇지 않고 순차적으로 할 경우 우리는 반복문 안에서 60 번 정도의 버튼을 선택해야 합니다. 60 번 정도면 할 수 있지 하면 1000 번, 10000 번 등의 반복 횟수가 많다면 어떨까요? 이럴 때를 위해 Visual Studio 에서 제공하는 기능을 사용하면 됩니다. 바로 중단점 조건 입니다.

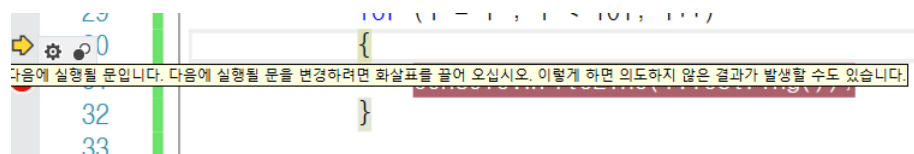


조건을 입력해서 특정 조건이 맞으면 그 때에 비로서 프로그램 실행을 일시 중지 합니다.



이렇게 일시 정지를 할 때 조건이나 적중 횟수를 활용할 경우 반복이나 특정 조건에 맞을 때 정지 되므로 매우 편리하게 사용할 수 있습니다.

디버깅에서 이제는 다음에 실행될 문의 변경을 할 수 있도록 합니다. 즉 내가 현재 멈춤이 된 부분이 아니라 다른 부분으로 이동할 수 있습니다. 디버깅을 할 경우 중단점에 잠시 멈춤이 되고 이 부분을 노란색으로 표시를 합니다. 이 노란색의 맨 앞 중단점 부분에서 마우스로 선택 후 디버깅 이전이나 다음으로 넘길 수 있습니다. 보통은 이전으로 이동하는데 값을 초기화 해서 다시 디버깅 할 경우 유용하게 사용할 수 있습니다.



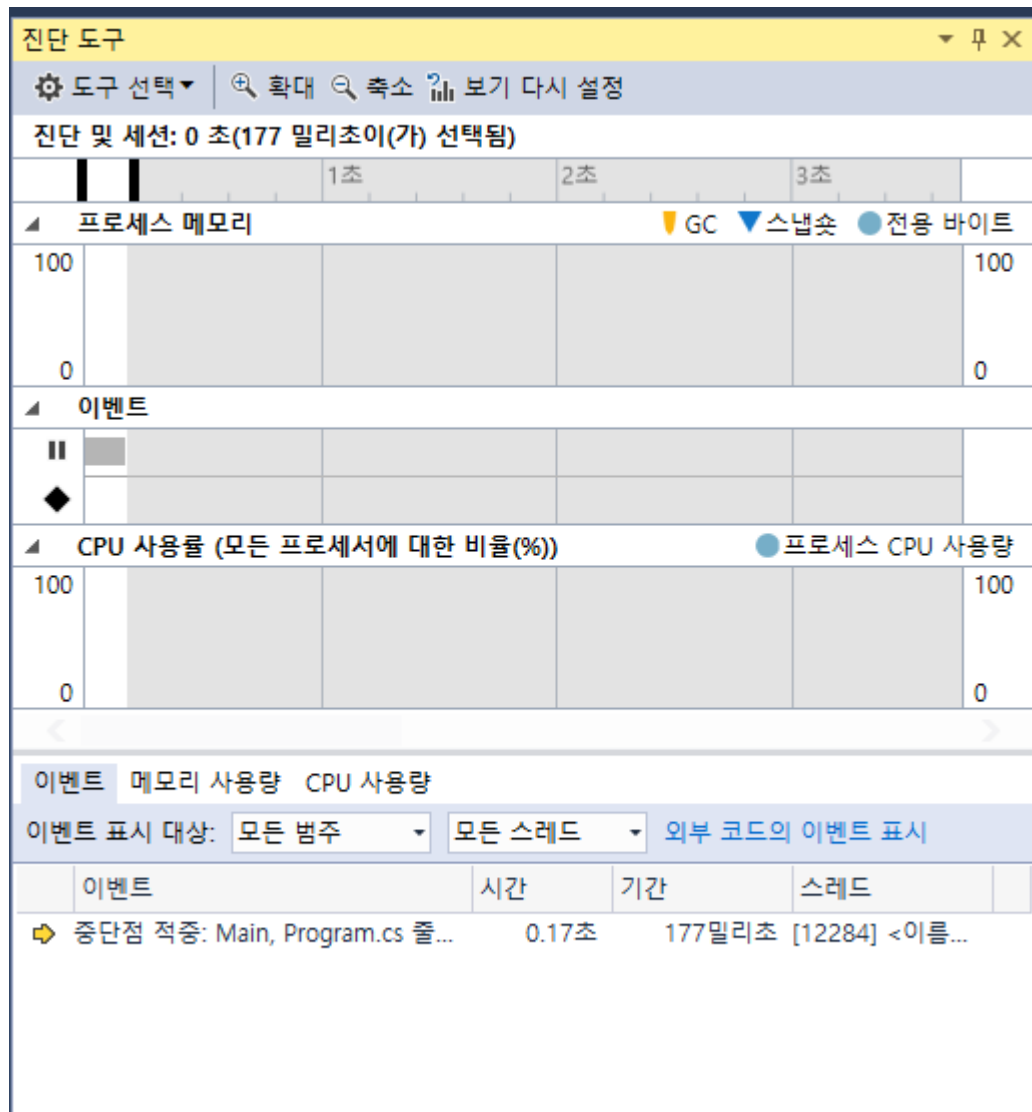


이렇게 하는 이유는 디버깅 중에 소스 코드를 변경하고 특정 시점으로 다시 이동하여 디버깅을 할 수 있기 때문입니다. 다시 말해 즉시 코드를 변경하고 다시 코드를 디버깅할 수 있도록 되어 있습니다.

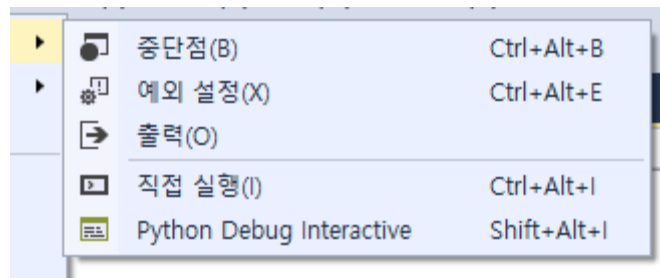
### Visual Studio 2015 의 새로운 도구 - 진단도구

Visual Studio 2015 버전에서는 사실 새로운 도구에 가까운 "진단 도구"가 있습니다. 이 진단 도구는 매우 유용하게 사용될 수 있는데 프로그램의 실행 시간과 CPU 사용률, 프로세스 메모리 사용률을 표시해 줍니다.

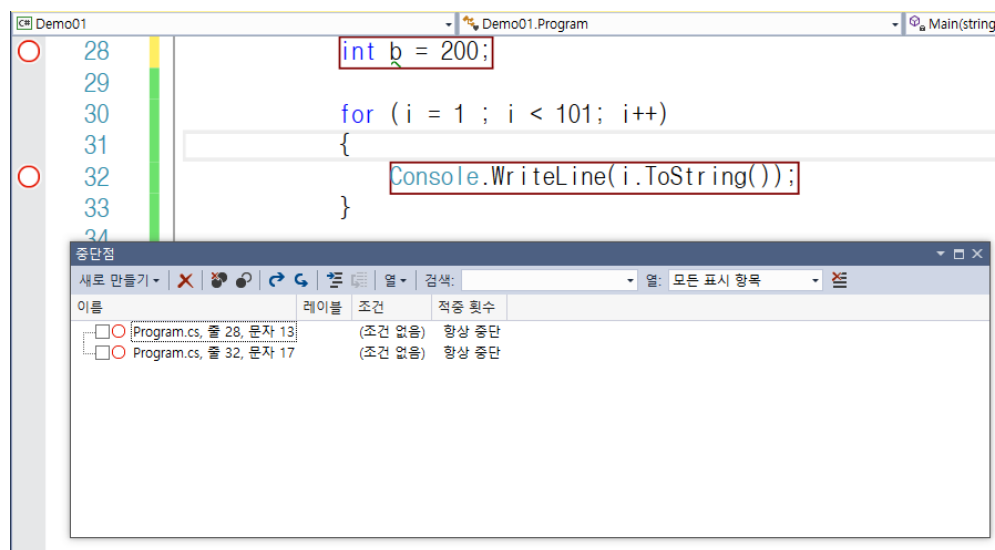
이 도구는 원래 고급 진단 또는 성능 테스트를 할 때 주로 사용했는데 디버깅에서 바로 표시해 주는 기능이 Visual Studio 2015 부터 바로 표시를 해줍니다.



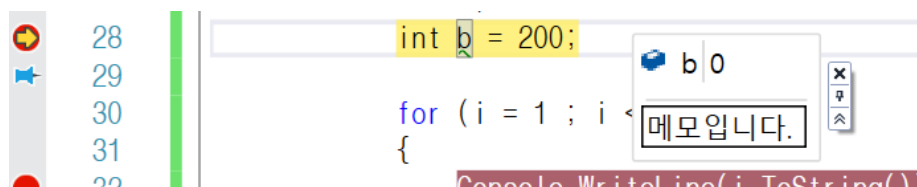
디버깅을 위한 중단점을 설정할 경우 중단점을 한눈에 보고 싶다면 디버그 메뉴에 있는 중단점 창을 이용합니다.



이 창을 이용하면 매우 중요한 디버깅 정보를 한눈에 볼 수 있으며 중단점을 설정한 것을 외부로 내보내기와 들여오기가 가능합니다. 그 외 조건과 적중횟수들을 우리가 한눈에 알 수도 있습니다. 그 외 중단점에서 잠시 디버깅을 하지 않기 위해서 중단점 해제를 할 수 있는데 이는 디버깅을 하고 난 다음 디버깅을 하지 않고 프로그램을 실행해 보기 위해 잠시 사용하는 기술입니다.



마지막으로 디버깅을 할 경우 우리는 특정 부분에 메모를 할 수 있는 기능을 제공합니다. 이 기능은 Visual Studio 2010 부터 제공하고 있습니다.



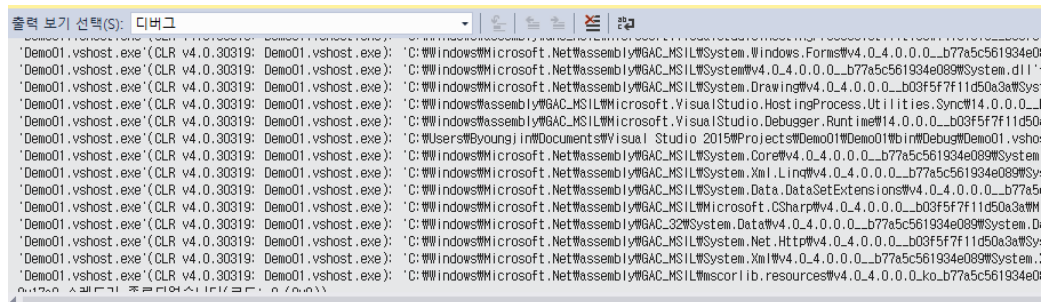
이 기능은 특정 변수에 마우스를 선택하면 바로 핀 같은 아이콘이 표시됩니다. 이 핀을 이용하여 잠깐 동안 디버깅에 대한 메모를 적어 넣을 수 있으며, 메모용도 또는 다른 디버깅에 대한 메모를 다른 사람에게 전달하는 용도로 사용해 좋습니다.

이 기능을 DataTips 라고 부르며 디버깅이 멈추면 중단점과 같은 위치에 핀이 있다고 표시해 줍니다. 이는 다른 디버깅 기능처럼 외부로 내보내기와 가져오는 것들을 지원하게 됩니다.



## ☆ 그 외 Debugging Window

그 외 Debugging 할 때 표시되는 창은 매우 많습니다. 그렇지만 우리가 확인해야 할 창은 출력 입니다. 이 출력을 잘 보면 이 프로그램을 실행하기 위한 디버그 로그들을 표시합니다.



이 기능은 실제 .NET Framework 에서 사용되는 DLL 등 실제 프로그램에 실행되기 전에 .NET Framework 에서 실행되는 것을 보여주는 내용입니다. 것에 대한 목록을 보여줍니다. 그 외 이 출력창에서는 어느 지점 어느 줄에서 문제가 생겼는지도 표시해 주기 때문에 추후에 사용할 수 있습니다.

예외 설정 창 또한 우리가 활용할 수 있는데 이는 C# 언어에서는 “Common Language Runtime Exceptions” 항목에서 우리가 예외사항이 발생할 경우 중단을 할 수 있도록 합니다. 이 예외 설정은 디버깅을 할 때 예외 사항을 만나면 예외 사항을 처리하고 디버깅에서 걸리지 않게 되는데 이를 Tool 단위 기능에서 제공하여 개발자들이 혹여 예외 처리가 발생했는데 디버깅을 하지 못하는 넘어가는 실수를 방지할 수 있는 좋은 기능입니다. 그 외 “Managed Debugging Assistants”와 “Win32 Exceptions”를 활용하면 Windows 의 문제와 관리되는 코드에서 발생하는 부분을 개발자가 모르고 넘어갈 수 있는 부분을 확인할 수 있으며, 기본적으로 이 기능 중에 몇 가지는 기본적으로 설정되어 있습니다.

## 예외 설정



### 발생된 경우 중단

- ▶ ☒ C++ Exceptions
- ▶ ☒ Common Language Runtime Exceptions
  - ☐ <일부 Common Language Runtime Exceptions이(가) 이 목록에 없음>
  - ☐ Microsoft.JScript.JScriptException
  - ☐ System.AccessViolationException
  - ☒ System.AggregateException
  - ☐ System.AppDomainUnloadedException
  - ☐ System.ApplicationException
  - ☐ System.ArgumentException
  - ☐ System.ArgumentNullException
  - ☐ System.ArgumentOutOfRangeException
  - ☐ System.ArithmeticException

## 8) 개체지향이란 무엇인가?

우리가 지금까지 프로그램을 연습한 것은 사실 개체 지향 프로그래밍을 한 것은 아닙니다. 순차적으로 한 줄 씩 코드를 실행하는 단계라 할 수 있는 데 이는 전통적인 프로그램 코드 작성입니다. 그리고 이것을 흔히 절차적(Procedural) 프로그래밍이라고 합니다. C 언어와 C++ 언어의 차이가 바로 여기서 나타나게 됩니다.

Microsoft 의 C# 언어는 개체 지향 언어입니다. 이 개체 지향 언어로 프로그래밍 하는 것을 개체 지향 프로그래밍(Object Oriented Programming)이라고 합니다. 줄여서 OOP 라는 말로도 하는데 OOP 는 나와서 사용된 지는 오래된 프로그램 입니다. 개체 또는 객체라고도 하지만 여기서는 개체라는 표현으로 하겠습니다. 그 이유는 Microsoft 설명서의 한글판에서는 개체로 번역하고 있기 때문에 될 수 있으면 Microsoft 의 번역을 따르도록 하겠습니다. 그렇다고 객체라는 것이 다른 것이 아닙니다. 개체와 객체는 같은 말이고 같은 뜻입니다.

전통적인 프로그래밍에서는 일정의 흐름이 단순하고 직선적으로 많이 코딩을 했다면 OOP 에서는 이런 직선적이기 보다 필요에 따라 그 기능을 갔다가 쓰고 필요 없으면 잠시 뒤에 쉬게 둔다고 보시면 됩니다. 그럼 그 기능의 모음을 우리는 개체라고 정의할 수 있습니다. 우리가 이해가 쉽게 다른 표현으로 하지만 일정의 변수라고 생각하면 됩니다. 뒤에서 배우게 되는 클래스가 있습니다. 이 클래스의 변수가 바로 개체라고 할 수 있습니다. 사실 다른 책이나 IT 전공자들 중에서는 개체 지향 프로그래밍에 대하여 처음에는 매우 어렵게 설명되거나 어렵게 인식되기도 합니다. 그렇지만 쉽게 조금 더 쉽게 생각하면 우리 자신을 개체라고 생각하면 됩니다.

아침에 일어나서 무엇을 하는지 한번 생각해 보면 됩니다. "나"라고 하는 개체가 잠에서 일어나서 세수하고 밥 먹는 것을 비유하여 설명을 하겠습니다. "나" 자신이 프로그램이 눈을 뜨면 실행이 된다고 생각하고, 눈을 뜬 다음 오늘은 욕실에 가서 세수하고 이빨을 닦습니다. 그 다음 아침을 먹고 회사에 출근을 합니다. 여기서 기능 즉 개체라 할 수 있는 것은 세수를 하는 것, 이빨을 닦는 것, 밥을 먹는 것, 회사에 출근하는 것 이 하나 하나의 실행을 할 하는 기능입니다. 저녁에 와서 잠을 자고 그 다음날 우리는 똑같이 하지 않고 눈을 뜨고 세수하고, 밥을 먹은 다음에 이빨을 닦습니다. 이렇게 해도 큰 문제는 없습니다.

개체 지향 프로그래밍이란 바로 내가 필요에 따라 그 기능을 가져다가 쓰면 된다는 것입니다. 조금 더 전문적으로 이야기 하면, 응용 프로그램의 빌딩 블록을 만들고 이 블록을 가져다 쓰기 위해 기능을 만듭니다. 이런 기능을 만들 때 하나의 추상적인 엔터티라는 것일 수도 있습니다. 개체는 사실 구조체와 비슷하게 필요한 것을 미리

만들어 놓고 필요에 따라 그것을 호출하여 갔다가 쓰면 됩니다. 이런 개념을 클래스와 네임스페이스 그리고 생성자와 소멸자들을 배움으로써 사용하게 됩니다.

그렇다면 UML 이란 말을 들어보셨을 것 입니다. 이 UML 은 OOP 를 사람이 이해하기 쉽게 그림으로 표현하는 것이 UML 입니다. 그 이유는 OOP 프로그램을 하지 않고 다른 사람에게 전달하기 어렵기 때문에 우선 UML 이란 것으로 OOP 프로그램의 구조를 그림으로 그리고, 이것을 다른 사람과 공유하는 것이 바로 UML 입니다.

그럼 우리가 배워야 하는 것은 바로 다음과 같습니다.

- ① 필드와 속성
- ② 메서드
- ③ 클래스
- ④ 생성자와 소멸자
- ⑤ 인터페이스
- ⑥ 상속
- ⑦ 연산자 오버로딩
- ⑧ 이벤트

이러한 것을 우리는 앞으로 배우게 됩니다.

사실 개체 지향 프로그래밍은 처음에는 매우 어렵게 느낄 수 있습니다. 그런데 절대 기억해야 할 것이 있습니다. 영화 해리포터에서 주인공 해리포터는 그 어떤 훌륭한 마법사도 모두 학생이었으며, 모두 배움의 과정이 있었다는 것을 이야기 합니다. 우리도 하나씩 배우고 그 것을 실천하여 서투르지만 계속해서 사용하면 훌륭한 마법사처럼 훌륭한 프로그래머 또는 개발자가 될 수 있습니다.

## 9) 개체와 수명에 대한 이해

사실 개체를 이야기할 때에는 사람과 비유하는 것이 적당하지 않을까 합니다. 그 이유는 개체는 수명이란 것이 있습니다. 개체를 저장하는 공간은 사실 앞에서 잠깐 언급한 힙이라는 메모리입니다. 이 메모리는 한계가 있고, 정기적으로 정리를 합니다. 사실 사람도 태어나서 일생을 우리가 잘 지내거나 또는 문제가 있지만 언젠가는 그 끝이 있습니다. 그 끝을 개체도 가지고 있습니다. 그래서 사람과 비유하는 것이 어떨까 생각합니다.

C# 와 .NET Framework 에서는 모든 것이 사실 개체 입니다. 우리가 몇 번 연습하고 디버깅을 봤지만 이 모든 것이 사실 개체라 할 수 있습니다. 그래서 C# 언어에서는 개체를 잘 다루는 것이 매우 중요합니다. 그리고 개체는 수명주기(Lifecycle)이 있다는 것을 꼭 기억해야 합니다.

이 개체는 C#에서는 “현재 쓰이고 있음”이 보통이며 그 외 다른 2 가지 있습니다.

- 생성
- 소멸

이렇게 2 가지가 있으며 우리는 이 것을 뒤에서 알아보겠습니다.

그럼 이런 것을 알기 위해 OOP 기법들을 자세히 알아야 합니다.

## .NET Framework 와 개체지향 프로그래밍

### 개체와 클래스의 관계

앞에서 언급한 것처럼 사실 개체와 클래스는 밀접한 연관 관계가 있습니다. 클래스는 사실 개체를 구현하는 하나의 단위라 할 수 있습니다. 그럼 이 클래스에 앞서 배워야 할 것이 하나 있습니다. 바로 메서드(Method 또는 메소드라고도 하는데 뜻은 같습니다.)입니다.

#### ☆ 메서드

메서드는 다른 말로 함수(function)이라고 할 수 있는데 C#언어에서는 메서드라고 합니다. 메소드라고도 하지만 우리는 Microsoft 의 번역을 기준으로 하기 때문에 메서드라는 표현을 하겠습니다. 이 메서드는 바로 기능을 정의하는 공간입니다. 앞에서 개체를 설명할 때 밥을 먹는 것을 예를 들면, 밥을 먹기 위해 우리는 식탁에 앉아서 손으로 숟가락과 젓가락을 이용하여 사용하여 밥을 먹습니다. 그럼 밥을 먹기 위해 우리는 손을 움직이고 숟가락과 젓가락을 이용하는 바로 "이 손을 움직여서 숟가락을 잡고 입에 넣어"라는 명령을 내리는 하나의 절차를 메서드에 정의를 내려서 밥을 먹을 때에는 이 메서드를 호출하여 사용합니다.

메서드는 그 안에서 사용해야 할 특정 데이터를 매개변수(parameter)라는 형태를 제공할 수 있는데 이는 기본적인 도구를 미리 주어진다고 생각하면 됩니다. 그리고 밥을 먹을 때 그 밥이 맛있다 맛없다를 표현할 수 있는데 이것을 반환값(return value)이라고 합니다. 물론 개체이므로 이런 표현을 하지 않아도 됩니다. 그럼 이제 처음 봤던 Hello C# 연습을 잠시 보겠습니다.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace HelloWorld
8  {
9      참조 0개
9      class Program
10     {
11         참조 0개
11         static void Main(string[] args)
12         {
13             Console.WriteLine("안녕하세요 C# 입니다.");
14         }
15     }
16 }
```



사실 처음으로 C# 언어에 대하여 Visual Studio 로 설명을 시작하는 것과 다름없는데 앞에서 몇 번 연습을 해본 환경이라 이제 어느 정도 익숙한 환경이라 생각합니다.

여기에서 중요한 것은 static 부분입니다. 바로 여기가 메서드를 정의하는 부분입니다. 메서드를 정의할 때에는 다음과 같이 합니다.

```
[접근 지정자] 반환형식 메소드명(매개변수_리스트) {  
    // 실행될 코드  
}
```

이렇게 접근 지정자와 반환형식 그리고 메서드 이름과 매개변수를 정의하고 중괄호 안에 코드를 작성하는 것입니다.

그럼 개체 지향에서는 바로 이 메서드를 이용하여 기능을 정하고 호출하는 것입니다. 즉 처음 시작은 Main 이라고 하는 메서드에서 시작으로 이 안에 다른 메서드를 호출하여 필요에 따라 그 메서드를 호출하고 그 일이 다 끝나면 그 메서드를 종료하게 되고, 더 이상 할 일이 없으면 그 Main 을 종료하여 그 일을 마치는 것입니다.

그럼 여기서 이제 이 메서드에 대하여 조금 더 자세히 알아보겠습니다.

메서드를 필요에 따라 갖다 쓴다고 했는데 이 메서드를 사실 아무나 갖다 쓴다는 것은 사실 어려울 수 있습니다. 즉 내꺼를 다른 사람에게 사용하게 해 줄 수 있으니까 이를 사실 접근 제한자로 클래스에서 조금 더 자세히 알아보겠습니다.

메서드에서는 사실 private, public, protected 이 세가지를 주로 많이 씁니다.

보통은 private 은 내꺼만을 쓴다는 것이고, protected 는 내 자손만, public 은 모든 사람이 내꺼를 서도 된다는 뜻입니다. 몇 가지 더 있지만 클래스에서 조금 더 자세히 보겠습니다.

그 다음 Main 함수에 보면 static 을 사용하는 경우 입니다. 그럼 이 static 키워드는 무슨 역할을 하는 것일까? 그리고 이건 왜 써야 하는지 입니다. 사실 .NET Framework 코드를 작성하면 가끔 static 코드를 보는 경우가 있는 이때 이 용도에 맞추어서 사용해야 합니다. 일반적으로 우리는 static 을 사용하지 않는데 static 을 Main 에서는 꼭 써야 합니다. 그 이유는 개체를 생성하지 않고 직접 메서드를 호출 할 수 있습니다. 조금 더 깊숙히 이야기 하면 이 static 을 사용하게 되면 프로그램 내에서 하나의 메모를 생성하고 모든 개체에서는 공통적으로 사용하는 전역변수의 개념으로 사용할 때 사용합니다.

절대적이고 유일무이한 하나의 메모리만 생성하기 때문에 모든 개체에서 뭔가 선언할 필요 없이 바로 사용할 수 있습니다. 즉 모든 개체에서 공통으로 사용하는 전역 변수의 개념으로 사용하면 됩니다. 한 문장으로 정리하면 static 멤버는 모든 개체가 어떠한 값을 공통으로 공유해야 할 경우 사용하는 변수라고 할 수 있습니다. 그리고 Main에서는 static 을 사용합니다.

그 다음 반환형식을 정의 하는데 이 형식은 앞에서 배웠던 데이터 형식을 정의 하거나 또는 개체를 정의하면 됩니다. 그럼 return 이라는 것이 메서드의 마지막 실행할 때 꼭 그 타입으로 값을 돌려줘야 한다는 뜻입니다. 앞의 예에서는 밥을 먹고, 맛이 있다/없다 둘 중 하나만 선택할 수 있도록 반환형식을 정의해야 한다는 것입니다. 반환형식이 없다면 "void" 라고 정의하면 됩니다. void 로 지정된 메서드라도 return 을 사용할 수 있으며 이때에는 return 다음에 아무것도 쓰지 말고 ";"를 사용하면 됩니다.

그 다음은 메서드 명이며 이 메서드 명은 변수 명명 규칙처럼 만들면 되는데 이때 중요한 것은 대소문자를 구분하여 잘 사용하도록 합니다.

마지막으로 매개변수 입니다. 이 매개변수에 값을 전달하는 것을 "인수의 전달"이라고 합니다. 이를 전달할 때에는 보통 Pass by Value 방식을 따르는데 매개변수에 해당하는 데이터 타입과 변수명을 사용하면 됩니다.

다른 인수의 전달 방법으로는 Pass by Reference 방식이 있는데 이걸 Call by Reference 와 Call by Value 를 사용하기도 합니다.

Call by reference 는 Call by value 와는 달리, 변수의 주소값을 매개변수로 보냅니다. 직접 원래의 변수를 참조하는 방법입니다.