<C# 강좌 리스트>

목차

[C# 강좌 및 동영상 목록]

[C# 동영상 강좌] 00.프롤로그 – C# 4.0

[C# 동영상 강좌] 01.C#의 발전

[C# 동영상 강좌] 02.Hello World C# - 기본 개념의 이해

[C# 동영상 강좌] 03.C# 기본 구문

[C# 동영상 강좌] 04.Data Type 1 - 값 형식

[C# 동영상 강좌] 05.Data Type 2 - 참조 형식, 변수, 상수

[C# 동영상 강좌] 06.조건식

[C# 동영상 강좌] 07.반복문

[C# 동영상 강좌] 08.예외 처리

[C# 동영상 강좌] 09.배열

[C# 동영상 강좌] 10.문자열 다루기

[C# 동영상 강좌] 11.클래스 1 - 선언하기

[C# 동영상 강좌] 12.클래스 2 - 다양한 클래스 선언

[C# 동영상 강좌] 13.프로퍼티

[C# 동영상 강좌] 14.인덱서(Indexer)

[C# 동영상 강좌] 15.메소드

[C# 동영상 강좌] 16.확장 메소드, Optional-Named 파라미터

[C# 동영상 강좌] 17.델리게이트, 이벤트

[C# 동영상 강좌] 18.Func, Action

[C# 동영상 강좌] 19.익명 형식□

[C# 동영상 강좌] 20.컬랙션

[1~20 강좌 전체 소스]

SqlerCSharp_SRC.zip

[C#강좌] 00.프롤로그 – C# 4.0

2000 년 6 월 Microsoft PDC(Professional Developers Conference) 2000 에서 Microsoft 가 닷넷(.NET) 전략에 대해 발표한 이후 Microsoft 의 제품들은 계속해서 닷넷과 통합이 이루어 지고 있다. (SQL Server, Office, Sharepoint 등) 특히 C#은 닷넷 전략이 발표 되면서 기존의 언어와는 다른 닷넷 환경에 최적화된 언어의 필요성에 의해 새롭게 탄생된 언어이다.
C# 언어도 닷넷 프레임워크(.NET Framework)의 발전과 더불어 계속해서 변화를 가져 왔으며, 닷넷 프레임워크 4.0 출시와 함께 C# 4.0 으로 큰 변화를 가져 왔다.
먼저 C# 언어에 대해 알아보기에 앞서 개발 환경을 위한 몇 가지 사항에 대해 알아 보자.

- **개발툴(IDE)**
  - Visual C# 2010 Express Edition
  - Microsoft Visual Studio 2010 Ultimate 평가판 - ISO
  - Microsoft Visual Studio 2010 서비스 팩 1

C#으로 다양한 프로그램을 개발하기 위하여 통합 개발 환경(IDE)이 필요하다. 현재 마이크로소프트는 Express 버전과 엔터프라이즈 환경을 위한 개발툴을 나누어 제공하고 있다.
Express 버전은 무료로 사용할 수 있으며, 소규모 프로젝트에 사용이 적합하다. 하지만 대부분의 회사에서 큰 규모의 프로젝트를 진행하기에는 부족함이 있을 것이다. 규모가 큰 프로젝트일수록 빌드, 테스트, 형상관리 등 다양한 지원이 원할 하게 이루어 져야 하고, 하나의 툴에서 이러한 기능이 유기적으로 잘 지원이 되어야 한다. 이러한 요구조건은 만족할 수 있는 것이 Visual Studio 2010 Ultimate 버전이다. Visual Studio 2010 의 최상위 버전으로 설계, 테스트, 빌드 등 다양한 기능을 제공하고 있다.

- **도움말**
  - MSDN
  - Visual C# 개발자 센터
  - MSDN Magazine

개발을 진행하다 보면 업무적인 부분, 프로그램 언어 자체에 대한 부분, API 사용에 대한 부분은 다양한 문제점과 만나게 된다. 이러한 내용에 대한 답을 많이 가지고 있는 곳이 바로 MSDN 이 아닐까 싶다. 마이크로소프트(Microsoft) 각 제품별로 방대한 양의 자료를 제공하고 있으며, 특히 한글로 번역이 잘 되어 있어 많은 도움을 받을 수 있을 것이다.
MSDN Magazine 은 매달 마이크로소프트에서 발행하는 개발 관련 웹진으로(약간의 비용으로 오프라인 구독도 가능) 전문적인 다양한 읽을 거리를 제공 하고 있다. 자신의 관심 분야에 대해 깊은 주제를 알고 싶다면, 좋은 참고 자료가 될 것이다.
이제 개발에 필요한 환경이 만들어 졌을 것이다. 다음 시간부터 C#에 대해 자세히 알아 보는 시간을 갖도록 하겠다.

[C#강좌] 01.C#의 발전

2000 년 6 월 Microsoft PDC(Professional Developers Conference) 2000 에서 Microsoft 가 닷넷(.NET) 전략에 대해 발표한 이후 Microsoft 의 제품들은 계속해서 닷넷과 통합이 이루어 지고 있다. (SQL Server, Office, Sharepoint 등) 특히 C#은 닷넷 전략이 발표 되면서 기존의 언어와는 다른 닷넷 환경에 최적화된 언어의 필요성에 의해 새롭게 탄생된 언어이다.

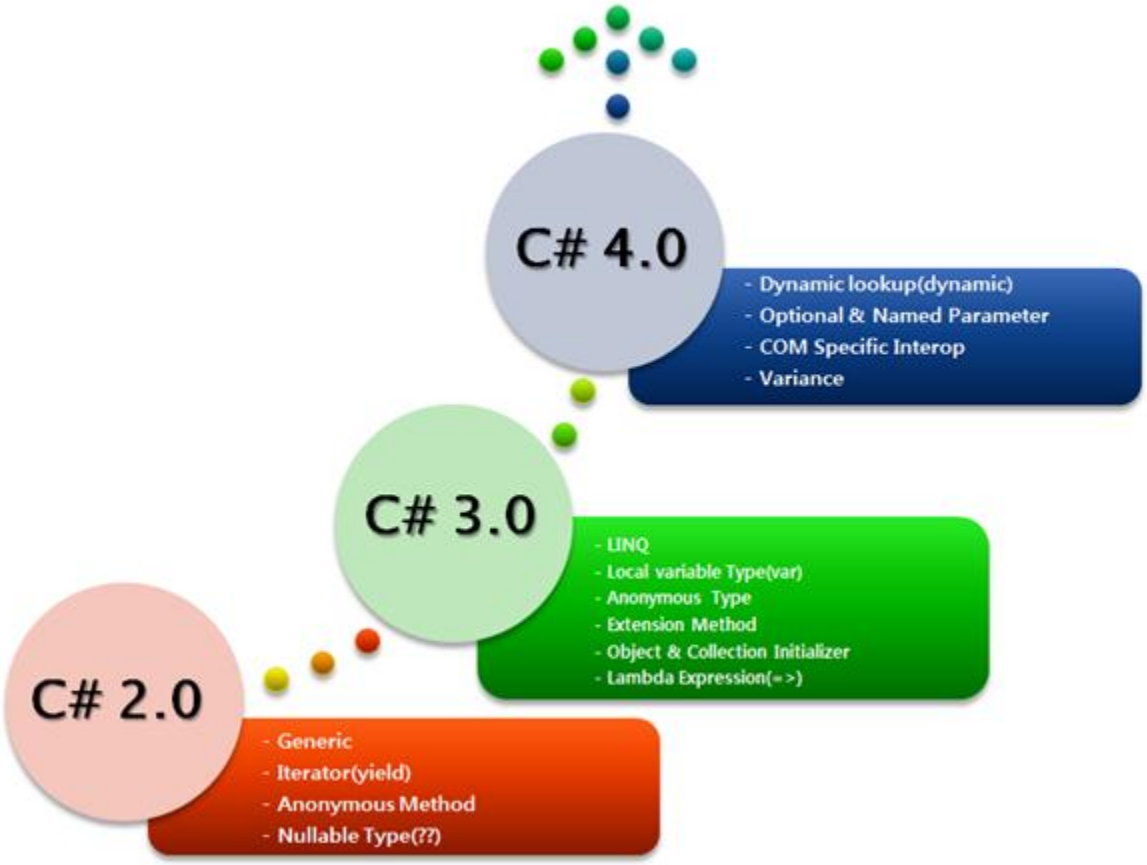
[닷넷 프레임워크 버전별 변화]

닷넷을 지원하는 언어는 Visual Studio 에서 기본으로 지원하는 C#과 더불어 Visual Basic, Visual C++, F# 이외에 Delphi, Cobol, Iron Ruby, Iron Python 등 다양한 서드파티(3rd Party)에서 지원을 하고 있다. (http://www.dotnetlanguages.net/DNL/Resources.aspx) 하지만, 닷넷을 지원하는 다양한 언어들이 이미 예전에 개발된 컴파일러 및 언어적 특성을 닷넷을 지원하기 위해 적절히 수정하여야 했고, 닷넷의 장점을 십분 발휘하기 쉽지 않았다. C#은 탄생부터 닷넷을 위해 설계가 되었으며, 닷넷의 개발 장점을 가장 잘 가지고 있는 언어가 아닌가 생각 한다. 닷넷 개발시 많은 곳에서 C#을 기본 언어로 사용하고 있으며, C++의 강력한 성능과 Visual Basic, Java 와 같은 유연함을 함께 가지고 있는 훌륭한 언어라고 볼 수 있다. 또한 C#은

Standard ECMA-334 (http://www.ecma-international.org/publications/standards/Ecma-334.htm),

ISO/IEC23270:2006
(http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=42926)

의 표준으로 등록이 되어 있다.



[C# 버전별 주요 특징]

- C# 1.0

과거 CS(Clinet-Server) 환경에서 웹 기반 솔루션(Solution)으로 빠르게 이동이 되면서, 기존 프로그램 언어와 플랫폼으로는 시장의 빠른 흐름에 대응하기에는 한계점을 가지고 있었다. Microsoft 는 이러한 문제점을 해결하기 위하여 닷넷 프레임워크 1.0 과 C# 1.0, Visual Studio 2002 를 시작으로 닷넷이 세상과 첫 만나게 된다. 기존 개발자가 모든 상황에 대한 처리 코드를 작성해야 했다면, 닷넷으로 넘어오면서 메모리 관리는 가비지 컬렉션(Garbage Collection) 으로, DLL Hell 은 전역 어셈블리를 통한 버전 관리 등과 같이 프레임워크 차원에서 관리되는 형태로 변화 되었다. 그만큼 개발자는 비즈니스를 구현하는데 집중을 할 수 있게 되었다.

- C# 2.0

현재 가장 많이 사용 되고, 현재 닷넷 기술의 기틀이 자리 잡은 중요한 시기이다. 가장 큰 특징은 제네렉(Generic) 지원이다. 이전에는 모든 컬렉션(Collection)이 오브젝트(object) 형식만 지원하여 값 형식(Value type)을 컬렉션에서 사용할 경우 참조 형식(Reference type)으로 변환하고 다시 원본

형식으로 캐스팅 하는 박싱(Boxing), 언박싱(Unboxing) 작업으로 성능에 안 좋은 영향을 미쳤었다. 하지만 제네릭 형식 선언 및 사용으로 좀더 강력한 형식 확인을 통하여 런타임 오류 및 잦은 형 변환에 따른 오버헤드를 줄 일 수 있게 되었다. 그리고 이터레이터(Iterator)를 쉽게 구현할 수 있는 yield, 델리게이트 사용이 간편해진 익명메서드(Anonymous Method), 값 형식의 널 형식(Nullable Type) 지원 및 널 형식의 기본값 설정을 쉽게 할 수 있는 "??" 연산자를 지원 하고 있다.

- C# 3.0

여러 데이터원본(Object, DB, XML, 컬렉션 등)에서 데이터 처리를 위한 통합 인프라인 LINQ(Language INtegration Query)를 지원해 주고 있다. 그리고 많은 추가된 기능들이 LINQ 와 연관을 맺고 있다. 조회데이터의 유연한 사용을 위한 Local variable type(var), 익명 형식(Anonymous type), 기존 객체의 소스 변경 없이 기능을 확장 할 수 있는 확장메서드(Extension Method), 좀더 간결한 코드로 객체 및 컬렉션 초기화를 위한 Object & Collection Initializer, 익명 메서드(Anonymous Method)를 더 쉽게 사용할 수 있는 람다식(Lambda)을 들 수 있다. 닷넷 프레임워크 3.0 은 상당히 시기적으로 아쉬운 점이 있다. 닷넷 프레임워크 2.0 기반에 획기적인 응용기술을 선보였으나 개발툴인 Visual Studio 가 뒷받침을 해 주지 않아 특히 WPF 개발에 있어서는 Visual Studio 2008(개발 코드명 Orcas) 출시 전까지는 닷넷 프레임워크 3.0 개발에 있어서 어려움이 많았다. Visual Studio 2005 에서는 Extension previews 형태로 지원하다 Visual Studio 2008 CTP, Beta 출시 이후에는 Beta 툴로 정식 프레임워크의 기술을 개발하는 상황 발생이 발생한 것이다.

- C# 4.0

닷넷 프레임워크(.NET Framework) 4.0 으로 오면서 CLR(Common Language Runtime), Visual Studio, C# 모든 것이 새롭게 탄생 되었다. C# 4.0 은 다이나믹(dynamic)을 기본 컨셉으로 Dynamic lookup(dynamic), Named and Optional Parameters, COM Interop, 공변성(Variance) 그리고 병렬처리와 같은 기술이 추가 되었다.

C# 4.0 에 대한 언어적인 세부 정보는 MSDN 에서 다운로드 가능하다.

(http://www.microsoft.com/downloads/en/details.aspx?FamilyID=dfbf523c-f98c-4804-afbd-459e846b268e)

델리게이트(delegate)를 통해 본 C#의 변화 실 예제

C# 버전별로 어떻게 변화가 이루어 졌는지 일례로 델리게이트(delegate) 사용 방법 변화를 통해서 살펴보자. 혹 현재 C#을 처음 접하게 되어 아래 내용이 이해가 잘 되지 않는다면, 먼저 다른 C# 주제를 먼저 학습한 후 읽어도 좋을 것이다.

우리나라 말로 델리게이트를 위임이라고 표현한다. 뜻 그대로 나 대신 일을 처리해 주는 것이다. 개발 측면에서는 가장 많이 사용하는 것이 이벤트(Event) 와 함께 사용이 된다. 스래드(Thread)를 사용하려면 ThreadStart 라는 델리게이트에 위임 받을 메서드를 설정 하여 사용하게 된다.

```
public delegate void ThreadStart()

public Thread( ThreadStart start )
```

C# 1.0 에서는 일반적인 컴파일러 언어의 특징처럼 명시적으로 모든 것을 선언한다.

```
//C# 1.0 명시적(Explicitly) 선언

ThreadStart tStart = new ThreadStart(DoWork);

Thread thread1 = new Thread(tStart);
```

C# 2.0 에서는 2 가지 변화를 가진다.

하나는 구문을 간소화 할 수 있는 부분, 그리고 무명 메서드를 통하여 명시적으로 메서드를 선언 하지 않아도 구문을 작성 할 수 있도록 되어 있다. 좀더 유연한 프로그램을 가능하게 해 준다.

```
// C# 2.0 묵시적(Implicitly) 선언

ThreadStart tStart = DoWork;

Thread thread1 = new Thread(tStart);

//C# 2.0 무명 메서드(Anonymous Method) 이용

Thread thread1 = new Thread

(

delegate()

{

Console.WriteLine("Do work");

}

);
```

C# 3.0 은 무명 메서드에서 좀더 발전된 람다 표현식(Lambda Expression)을 통해서 델리게이트를 선언할 수 있다. 특히 Func, Action 이란 특수 델리게이트 들을 통해서 확장 가능한 프로그램 작성이 가능하다.

```
// C# 3.0 람다 표현식(Lambda Expression)

Thread thread1 = new Thread

(

() =>

{

Console.WriteLine("Do work");

}

);
```

언어의 발전 방향을 살펴 보면 작성되는 코드를 좀더 적게 작성하면서, 직관적인 구문 형태로 변화 되는 것을 볼 수 있을 것이다.

[스레드 각 버전별 처리 예]

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace SqlerCSharp.Ch01_01
{
    class Program
    {
        static void Main(string[] args)
        {
            //C# 1.0 명시적(Explicitly) 선언
            //C# 1.0 에서는 스레드 실행시 파라미터를 직접 넘길 수 없으므로 별도 처리가 필요
            ThreadExample tExam = new ThreadExample("C# 1.0 명시적(Explicitly) 선언");
            ThreadStart tStart = new ThreadStart(tExam.DoWork);
            Thread thread1 = new Thread(tStart);
            thread1.Start();
```

```
        //C# 2.0 묵시적(Implicitly) 선언
        //C# 2.0 부터 ParameterizedThreadStart 가능
        ParameterizedThreadStart tStart2 = DoWork2;
        Thread thread2 = new Thread(tStart2);
        thread2.Start("C# 2.0 묵시적(Implicitly) 선언");

        //C# 2.0 무명 메서드(Anonymous Method) 이용
        Thread thread3 = new Thread
        (
            delegate(object data)
            {
                for (int i = 0; i < 10; i++)
                {
                    Console.WriteLine("{0} : {1}", data, i);
                }
            }
        );
        thread3.Start("C# 2.0 무명 메서드(Anonymous Method) 이용");

        //C# 3.0 람다 표현식(Lambda Expression)
        Thread thread4 = new Thread
        (
            (data) =>
            {
                for (int i = 0; i < 10; i++)
                {
                    Console.WriteLine("{0} : {1}", data, i);
                }
            }
        );
        thread4.Start("C# 3.0 람다 표현식(Lambda Expression)");
    }

    public static void DoWork2(object data)
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine("{0} : {1}", data, i);
        }
    }
}

class ThreadExample
{
    string _Data = "";

    public ThreadExample(string data)
    {
```

```
            this._Data = data;
        }

        public void DoWork()
        {
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine("{0} : {1}", this._Data, i);
            }
        }
    }

}
```

[C#강좌] 02.Hello World C# - 기본 개념의 이해

C#으로 작성된 어플리케이션은 크게 클래스(Class), 네임스페이스(Namespace), 어셈블리(Assembly)와 같은 요소로 구성되어 있다.

- 클래스(Class)

프로그램 내에서 독립적으로 존재할 수 있는 최소 단위. 멤버로 메소드(Method), 프로퍼티(Property), 이벤트(Event), 델리게이트(Delegate) 등을 가진다.

- 네임스페이스(Namespace)

여러 개의 클래스들이 모인 논리적 그룹 단위.

- 어셈블리(Assembly)

클래스, 네임스페이스가 모여 생성된 물리적 파일. 간단히 말하면 빌드 후 생성되는 DLL 혹은 EXE 파일이라고 생각하면 된다.

이렇게 작성된 코드를 빌드하게 되면 MSIL(Microsoft Intermediate Language)이라고 하는 기계어 직전 단계의 언어로 구성된 어셈블리가 생성이 된다. 바로 바이너리(Binary) 형식이 아닌 MSIL 형태로 생성을 하게 되는 이유는 자바와 마찬가지로 플랫폼에 종속적이 않고, 다양한 플랫폼에 이식 가능하도록 하기 위해서 이다. 현재 Mono 프로젝트(http://www.mono-project.com/Main_Page)를 통해서 다양한 플랫폼에 적용 가능하다. 이렇게 MSIL 형태로 존재하는 어셈블리는 실행되는 시점에 Just-In-Time (JIT) 컴파일러(compiler)가 각 플랫폼에 맞는 코드로 컴파일 되며 실행이 된다.

이러한 런타임 환경에서 일어나는 다양한 일들은 닷넷 프레임워크에서 제공해 주며, 다음과 같은 구성 요소를 가지고 있다.

- CLS(Common Language Specification)

닷넷 프레임워크는 다양한 언어로 개발이 가능하다. 하지만 문제는 각 언어별로 특별히 지원하는 스펙이 존재할 것이다. 모든 프로젝트다 특정 하나의 언어로만 생성된 어셈블리만 사용하면 상관이 없으나, 개발을 하게 되면 다양한 언어로 개발된 어셈블리를 참조하게 될 것이다. 만약 참조된 어셈블리에서 지원하는 자료형 등이 해당 언어에서 지원이 되지 않는다면? 난감한 상황이 발생하게 된다. 이러한 문제점을 줄이고, 다양한 언어들 간에 호환성을 놓이기 위해 닷넷 프레임워크를 지원하는 최소한의

스펙을 정의하고 있는데 그것이 바로 CLS 이다. CLS 규칙을 따르는 것은 개발 시점에서는 약간의 제약사항이 발생을 하게 되지만 다른 언어들 간의 상호 운영성은 증대하게 된다.

- CTS(Common Type System)

CLS 를 포함하여, 닷넷에서 사용하게 되는 필드(Field), 메소드 등 모든 스펙이 정의된 것을 가리킨다.

- CLR(Common Language Runtime)

실행되는 프로그램의 메모리 관리, 보안 등 실제 운영되는 환경을 관리하게 되는 핵심 요소 이다.

- GC(Garbage Collector)

닷넷 프레임워크는 예전의 비관리 코드(Unmanaged Code)로 작성되던 시기에 개발자가 직접해주던 메모리 관리를 GC 를 통해서 자동으로 하게 된다.

[C#강좌] 03.C# 기본 구문

개발 환경 설정

C#으로 다양한 프로그램을 개발하기 위하여 통합 개발 환경(IDE)이 필요하다. 현재 마이크로소프트는 Express 버전과 엔터프라이즈 환경을 위한 개발툴을 나누어 제공하고 있다.

Express 버전은 무료로 사용할 수 있으며, 소규모 프로젝트에 사용이 적합하다. 하지만 대부분의 회사에서 큰 규모의 프로젝트를 진행하기에는 부족함이 있을 것이다. 규모가 큰 프로젝트일수록 빌드, 테스트, 형상관리 등 다양한 지원이 원할 하게 이루어 져야 하고, 하나의 툴에서 이러한 기능이 유기적으로 잘 지원이 되어야 한다. 이러한 요구조건은 만족할 수 있는 것이 Visual Studio 2010 Ultimate 버전이다. Visual Studio 2010 의 최상위 버전으로 설계, 테스트, 빌드 등 다양한 기능을 제공하고 있다.

<개발툴(IDE)>

- Visual C# 2010 Express Edition (http://www.microsoft.com/express/Windows/)
- Microsoft Visual Studio 2010 Ultimate 평가판 – ISO (http://www.microsoft.com/downloads/ko-kr/details.aspx?FamilyID=06a32b1c-80e9-41df-ba0c-79d56cb823f7)
- Microsoft Visual Studio 2010 서비스 팩 1 (http://www.microsoft.com/downloads/ko-kr/details.aspx?FamilyID=75568aa6-8107-475d-948a-ef22627e57a5)

특별한 경우를 제외하고 모든 예제는 Visual Studio 2010 Ultimate 에서 Console 프로그램으로 제작을 할 예정이다.

프로젝트 생성하기

이제 프로그램 입문자라면 한번쯤은 접하게 되는 Hello World 를 작성해 보도록 하겠다. Hello World 프로그램은 각 언어별로 가장 기본 이면서 중요한 부분을 보여 주기 때문에 한번은 꼭 접하게 되는 예제이다.

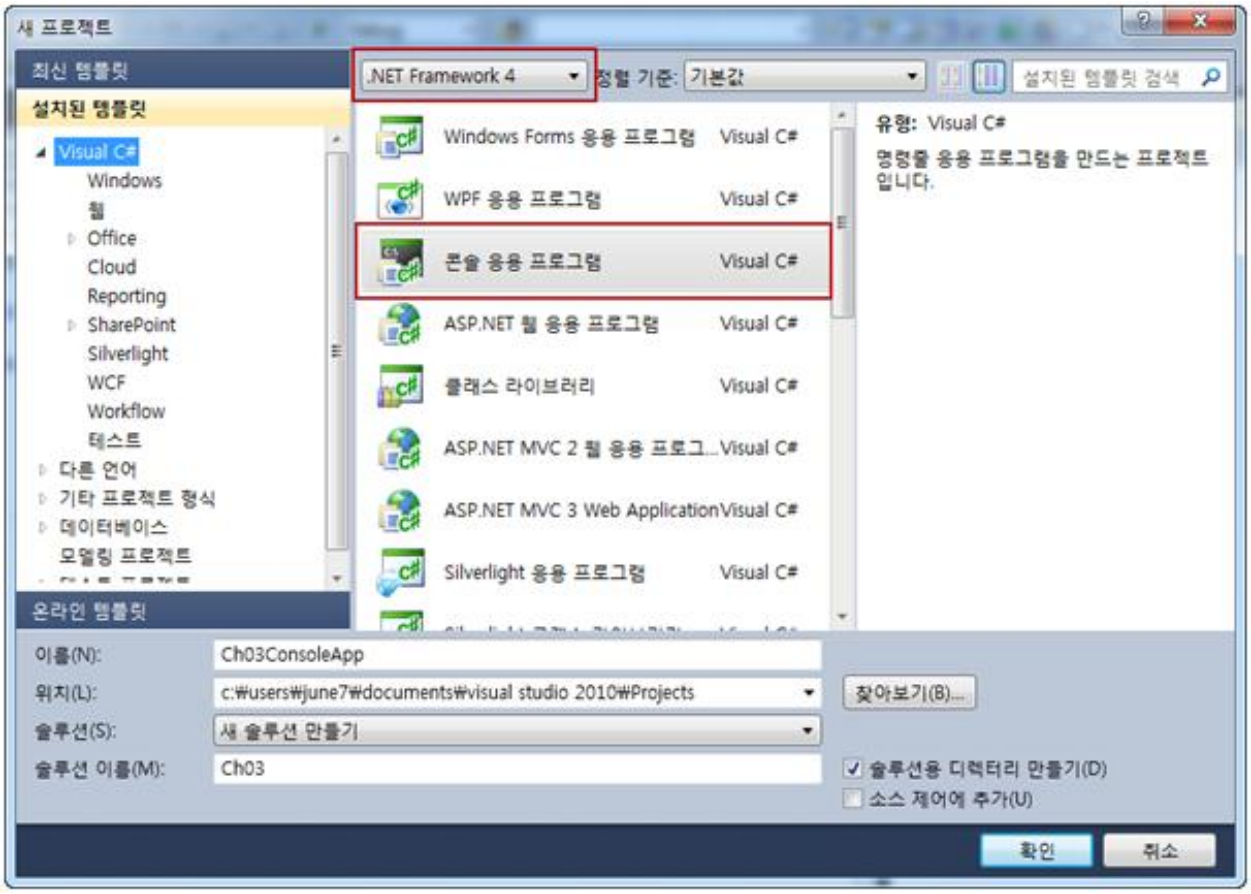
List of Hello World Programs in 200 ProgrammingLanguages(http://www.scriptol.com/programming/hello-world.php) 사이트에 방문하여 보면 참 다양한 언어가 존재하며, 그 언어별로 작성된 Hello World 예제를 감상할 수 있다.

<Hello World 프로젝트 생성>

메뉴에서 [파일] -> [새로 만들기] -> [프로젝트]를 클릭한다.

[프로젝트 생성]

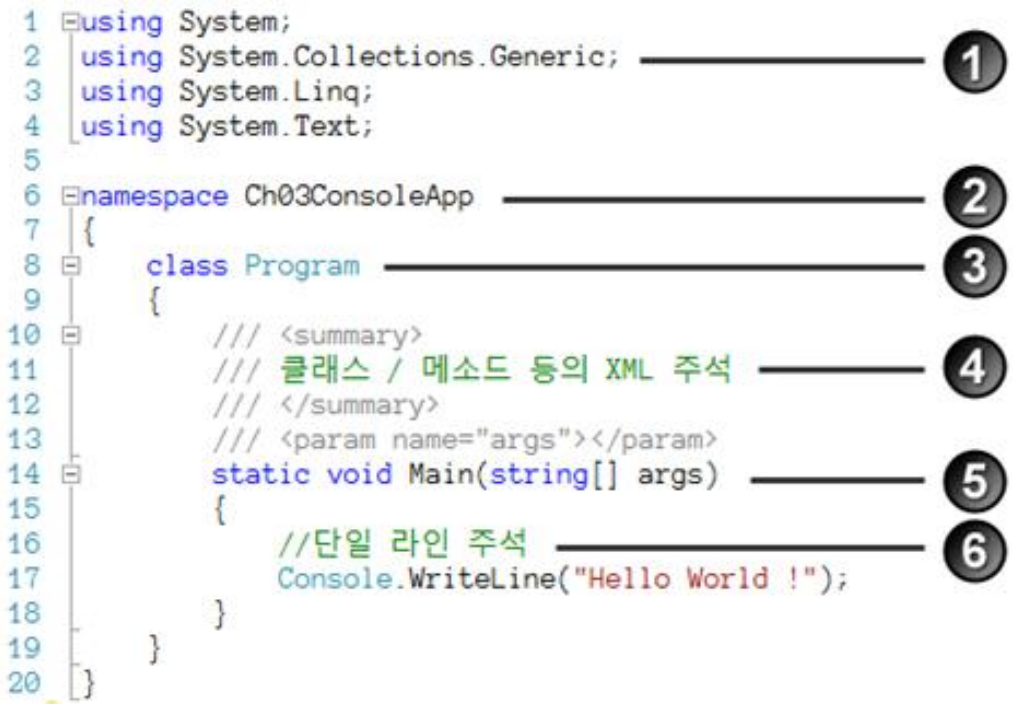
1. [새 프로젝트] 창에서 프레임워크를 [.NET Framework 4] 선택 후 적절한 프로젝트명과 위치를 지정한다. 참고로 Visual Studio 2008 부터는 .NET Framework 2.0 이상의 멀티 타겟 프레임워크를 지원하고 있다. 그러므로 하나의 툴에서 여러 프레임워크 버전의 솔류션을 개발 가능하다.



[새 프로젝트 설정]

Hello World 기본 구조

프로젝트를 생성하였으면 다음 그림과 같이 Main 메서드에 Console.WriteLine("Hello World !"); 를 작성을 하고, [Ctrl + F5] 키를 눌러 결과를 확인한다.



[C# 코드 기본 구조]

1. using 블록

   닷넷 프레임워크는 방대한 BCL(Base Class Library)을 제공해 주고 있다. 각 라이브러리는 네임스페이스와 클래스가 잘 계층 구조를 가지며 분류가 되어 있으며, 상단에 사용하고자 하는 네임스페이스를 미리 선언한다. 선언된 네임스페이스는 사용되는 시점에 네임스페이스를 생략하고 바로 클래스명으로 작성을 하고, 만약 네임스페이스를 using 구문으로 선언을 하지 않았으면 사용하는 시점에 [네임스페이스].[클래스].[메소스] 와 같이 전체 명칭을 작성해 주면 된다. 코드에서 보는 것처럼 Console.WriteLine() 메서드는 System 네임스페이스에 존재하는 클래스로 using 블록에 선언을 하였으므로 네임스페이스명을 생략하였다. System.Console.WriteLine()과 같이 작성하여도 된다.

2. 네임스페이스 선언

   PC 에서 파일 관리할 때를 생각해 보라. 각 자료의 특성에 따라 폴더를 분리하고, 그 안에서 파일을 관리한다. 마찬가지로 네임스페이스도 각 클래스들의 기능단위를 논리적으로 묶어 놓는 역할을 한다.

3. 클래스 선언

   독립적으로 존재할 수 있는 최소 단위로 모든 메서드는 반드시 클래스(혹은 struct)의 멤버이어야 한다.

4. XML 주석

   닷넷에서 제공하는 특수한 주석 구문으로 XML 주석을 기반으로 문서를 만들 수도 있으며, 일반 주석과 달리 다른 곳에서 해당 메서드 등을 참조할 때 인텔리센스(intellisense)에서 작성된 설명을 볼 수 있다.

5. Main 함수

   프로그램이 실행이 되면 최초의 진입점이 되는 곳으로 반환값은 없으며(void), 파리미터는 문자 배열(string[] args)을 받으며, 정적인 메서드(static)로 클래스의 인스턴스를 생성하지 않아도 실행 가능하다는 의미 이다.

6. 주석

   C# 에서는 주석을 크게 하나의 라인에서만 유효한 단일 라인 주석과 여러 줄에 걸쳐 적용이 되는 멀티 라인 주석을 지원에 주고 있다.

   - 단인 라인 주석 : //주석작성
   - 멀티 라인 주석 : /* 주석 작성 */

7. 구문 작성

   Console.WriteLine() 메서드는 콘솔창에 파라미터로 받은 문자열을 출력하라는 메서드 이다. 그리고 구문의 마지막에는 세미콜론(;)으로 끝나게 된다. C#은 세미콜론을 만날 때까지 하나의 의미 있는 구문으로 인식을 한다. 하나의 구문은 여러 줄로 작성할 수 있는 것이다.

Visual Studio 2010 을 이용한 솔류션 빌드 및 디버그

코드 작성이 완료가 되면 바로 실행을 하지는 않고 솔류션을 빌드하여 컴파일 오류가 없는지 확인을 하거나, 빌드와 함께 실행을 하여 논리적 오류가 없는지 디버깅 작업을 진행 하게 된다.

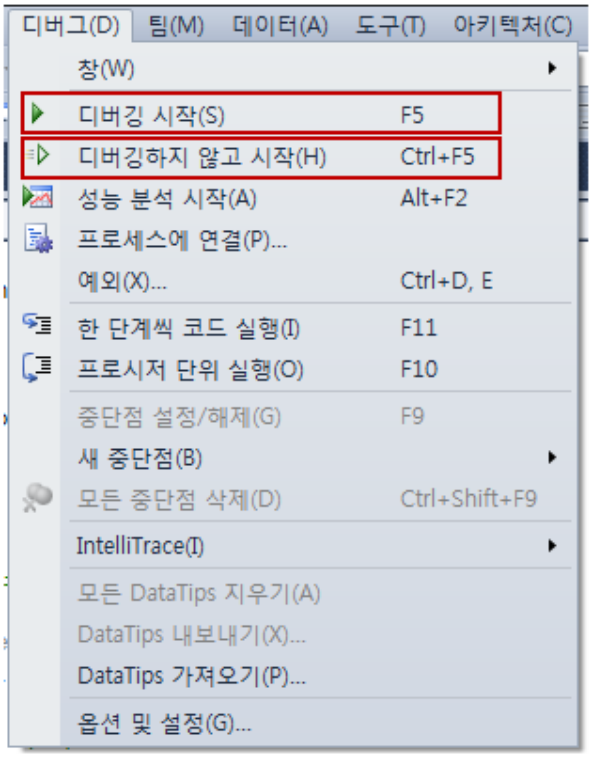
<솔류션 빌드 >

빌드는 크게 솔류션 전체를 진행하거나, 특정 프로젝트 하나만 진행을 할 수 있다. 메뉴에서 [빌드(B)] -> [솔류션 빌드(B)]를 선택 하거나 솔류션 탐색기에서 오른쪽 마우스를 클릭하여 가능하는하다. 만약 [솔류션 탐색기]가 보이지 않을 경우 메뉴에서 [보기(V)] -> [솔류션 탐색기(P)]를 선택한다.

[빌드 메뉴]

<디버그 >

최종 릴리스 버전이 나올 때까지 개발자는 계속해서 자신이 작성한 코드에 오류가 존재하지 않은지 많은 디버깅 과정을 거치게 된다. 중단점(Break Point)을 지정해 각 단계별 예외, 변수의 값 등을 확인 할 수 있으며, 실행 결과만 확인이 필요할 경우 디버깅 없이 실행도 가능하다. 보통은 단축키를 많이 사용하므로, 아래 그림에서 단축키를 확인해 두는 것이 좋다.



[디버그 메뉴]

[Hello World C#]

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch03_01
{
    class Program
    {
        /// <summary>
        /// 클래스 / 메서드 등의 XML 주석
        /// </summary>
        /// <param name="args"></param>
        static void Main(string[] args)
        {
            //단일 라인 주석
            Console.WriteLine("Hello World !");
        }
    }
}
```

[C#강좌] 04.Data Type 1 - 값 형식

프로그램은 수 많은 데이터의 입력과 연산, 그 결과의 출력이 일어나게 된다. 이러한 작업을 하기 위하여 각 데이터에 맞는 형식을 지정하고 연산 등의 작업을 하게 되는 것이다. 예를 들어 사람의 나이를 나타내는 경우 "30"과 같이 숫자로 표기도 가능하지만, 문자로 "서른" 이라고 표현을 하기도 한다. 하지만 컴퓨터는 기본적으로 같은 형식간에 연산을 지원하므로 "서른" 이라는 것을 형 변환을 통해 숫자로 바꾸어 처리를 해야 한다.

데이터 형식은 숫자형, 구조체(struct), 열거형(enum)과 같은 값 형식과 문자열, 클래스와 같은 참조 형식으로 나누어 진다.

- 값 형식(Value Type)
  - 숫자형 : sbyte, short, int, long, byte, ushort, uint, ulong, float, double, decimal
  - bool
  - char
  - struct
  - enum 등
- 참조 형식(Reference Type)
  - string
  - object
  - class 등

형식(Value Type)

C#에서 형식을 선언할 때 아래 표와 같이 C# 형식 혹은 System 네임스페이스에 있는 닷넷 프레임워크 형식으로 표현이 가능하다. 이 둘간의 차이점은 C# 표현은 닷넷 프레임워크 형식 표현의 별칭이다. 닷넷 초기 이 둘간의 표현 방법이 어느 것이 좋으며, 성능상에 유리한지에 대해 의견이 분분한적이 있다. 결론적으로 각 언어 별로 예전부터 형식에 대해 표현해 오던 방식이 있으며, 좀더 간략히 표현하는 것일 뿐 성능상의 차이는 존재하지 않는다. 개인적으로 다른 언어간의 연동이 주 목적이 아니라면 C# 형식으로 선언 하는 것이 코드의 가독성을 높일 수 있다. 그리고 예전 C/C++과 달리 닷넷에서 문자는 한글/영문 구분 없이 모든 한 문자는 16 비트 유니코드 문자이다. 숫자형식은 표현할 수 있는 전체 크기의 첫 번째 비트를 싸인(Sign) 비트라고 하여 양수인지 음수인지를 판단한다. 하지만 양수만 사용을 하는 되는 숫자라면(예를 들어 나이 같은) 싸인 비트가 필요 없게 된다. 이러한 싸인 비트도 모두 숫자를 표현하도록 한 것이 unsigned 형식이다. signed 형식 보다 2 배의 양수 범위를 가지게 된다.

C#에서 지원하는 형식은 다음과 같다.

구분	C# 형식	.NET Framework 형식	형식 접미사	크기(Bit)	범위
정수-signed	sbyte	System.SByte		8	-128 ~ 127
	short	System.Int16		16	-32,768 ~ 32,767
	int	System.Int32		32	-2,147,483,648 ~ 2,147,483,647
	long	System.Int64	L	64	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807
정수-unsigned	byte	System.Byte		8	0 ~ 255
	ushort	System.UInt16		16	0 ~ 65,535
	uint	System.UInt32	U	32	0 ~ 4,294,967,295
	ulong	System.UInt64	UL	64	0 ~ 18,446,744,073,709,551,615
실수	float	System.Single	F	32	±1.5e−45 ~ ±3.4e38
	double	System.Double	D	64	±5.0e−324 ~ ±1.7e308
	decimal	System.Decimal	M	128	(-7.9 x 1028 - 7.9 x 1028) / (100 - 28)
문자	char	System.Char		16 (Unicode)	U+0000 ~ U+ffff
Boolean	bool	System.Boolean		1	true, false

[C# 값 형식]

형식은 다음과 같이 선언하여 사용 가능하며 특히 접미사를 통해 초기화 되는 같을 명시적으로 선언할 수 있다. 한가지 주의 할 점은 모든 형식은 반드시 사용되기 전에 초기화 되어야 한다. 만약 초기화 하지 않은 변수가 있을 경우 컴파일 에러가 발생 한다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch04_01
{
    class Program
    {
        static void Main(string[] args)
        {
            //int 형 선언
            int number1 = 123;              // C# 표현, System.Int32 와 동일
            System.Int32 number2 = 123;     //.NET Framework 표현, int 와 동일
            Console.WriteLine("number1 : {0}", number1);
            Console.WriteLine("number2 : {0}", number2);

            //double 형 선언
            double number3 = 123D;          // "D" 형식 접미사를 통한 명시적 선언
            double number4 = 123;           //int 형식을 double 형식으로 암시적으로 형 변환
            Console.WriteLine("number3 : {0}", number3);
            Console.WriteLine("number4 : {0}", number4);

            //형 선언 후 초기화 하지 않음
            //int number5;                  //사용 전에 초기화 하지 않으면 컴파일 에러 발생
            //Console.WriteLine("number5 : {0}", number5);
        }
    }
}
```

열거형(Enumeration)

열거형은 사용자 정의된 숫자 형식으로 명칭으로 접근하거나 숫자로 형 변환이 가능하다. 열거형은 특별한 설정이 없으면 선언된 명칭 순서대로 0 부터 차례대로 내부적인 값을 가지며, enum 예약어를 사용한다.

예를 들어 다음과 같이 Country 라는 열거형을 선언하였다. Country 열거형은 Korea, China, Japan 값을 가지며, 내부적으로 Korea = 0, China = 1, Japan = 2 의 숫자값을 가지고 있다.

```
using System;

using System.Collections.Generic;

using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch04_02
{
    //열거형 선언
    public enum Country { Korea, China, Japan }

    class Program
    {
        static void Main(string[] args)
        {
            //사용
            Country location1 = Country.Korea;
            Country location2 = (Country)1;         // location2 = Country.China
            int location3 = (int)location1;         // location3 = 0 : Country.Korea

            Console.WriteLine("location1 : {0}", location1);
            Console.WriteLine("location2 : {0}", location2);
            Console.WriteLine("location3(int) : {0}", location3);
            Console.WriteLine("location3(Country) : {0}", (Country)location3);
        }
    }
}
```

Nullable 형식

int 와 같은 값 형식은 기본적으로 null 값을 가질 수 없다. 하지만 DB 와 연동을 하는 프로그램을 개발하다 보면, DB 에서는 모든 형식에 null 이 가능하다. 보통은 DB 에서 쿼리를 작성할 경우 Microsoft SQL Server 에서는 ISNull() 함수를 통하여 기본값을 지정하는 처리를 하였다. C# 3.0 에서 이러한 불편함을 해소하고 DB 와의 유연한 처리가 가능하도록 값형식에서도 null 값을 사용할 수 있게 되었다.

값 형식에 null 이 가능하도록 하려면 해당 형식선언 다음에 ?를 붙여 주면 된다. Nullable 형식은 내부적으로 System.Nullable<T> 구조체의 인스턴스로 HasValue 와 Value 라는 두 개의 프로퍼티를 가지고 있다.

- HasValue : 값이 null 이면 false, null 이 아닌 값이면 true
- Value : HasValue 가 true 이면 가지고 있는 값을 반환하며, HasValue 가 false 일 경우 InvalidOperationException 예외가 발생한다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch04_03
{
    class Program
    {
        static void Main(string[] args)
        {
            //int 에 null 가능 하도록 선언
            int? number = null;
            number = 10;                        //암시적으로 값 설정

            if (number.HasValue)                //number == null 으로 비교 가능
            {
                //Value 프로퍼티를 사용하지 않고 바로 변수명만 적어도 가능.
                Console.WriteLine(number.Value);    // Console.WriteLine(number);
            }
            else
            {
                Console.WriteLine("값이 null 입니다.");
            }
        }
    }

}
```

Nullable 형식은 유연성은 확보 할 수 있으나, 널 형식을 허용하지 않은 명시적 선언 보다는 오버헤드가 발생하므로 꼭 필요한 경우에만 사용을 해야 한다.(DB 와 연동하는 처리에는 유용할 것이다.)

?? 연산자

설정된 데이터를 연산을 할 경우 null 로 설정이 되어 있으면 올바른 사칙연산을 이루어 지지 않는다. 그러므로 DB 의 IsNull() 함수와 같이 null 일 경우 기본값 설정이 필요하다. C#에서 단일 행으로 조건식을 판단할 수 있는 ?: 연산지 이외에 특별히 ?? 연산자를 통해 쉽게 처리 가능하다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch04_04
{
    class Program
    {
        static void Main(string[] args)
        {
            int? number = null;

            //?: 연산자 사용
            int defaultNumber1 = (number == null) ? 0 : (int)number;
            Console.WriteLine("defaultNumber1 : {0}", defaultNumber1);

            //?? 연산자 사용 number 가 null 일 경우 0 을 기본값으로 설정하여 defaultNumber 에 설정
            int defaultNumber2 = number ?? 0;
            Console.WriteLine("defaultNumber2 : {0}", defaultNumber2);
        }
    }

}
```

[C#강좌] 05.Data Type 2 - 참조 형식, 변수, 상수

참조 형식(Reference Type)

참조 형식은 string, object, class 와 같은 형식으로 new 를 통하여 인스턴스가 생성이 되면 데이터를 참조하는 메모리 주소만 가지고 있으며, 실제 데이터는 분리된 공간에 저장이 된다. 참조 형식은 기존의 변수를 새로운 변수에 할당할 경우 값 전체를 복사 하는 것이 아니라, 값을 참조하는 주소를 복사하게 된다.

다음 코드에서 값 형식인 struct 와 참조 형식인 class 로 된 Employee 객체로 둘간의 차이점을 알아보자.

<struct 로 정의된 Employee>

emp1 의 값을 emp2 로 복사한 후 emp1.BirthYear 의 값을 변경을 하여도 emp2 의 값은 영향을 받지 않는다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch05_01
{
    //구조체 정의
    public struct Employee
    {
        public int BirthYear;
        public string Name;
    }

    class Program
    {
        static void Main(string[] args)
        {
            //인스턴스 생성
            Employee emp1 = new Employee();
            emp1.Name = "김도현";
            emp1.BirthYear = 1983;
```

```
            Employee emp2 = emp1;

            Console.WriteLine("emp1.BirthYear : {0}", emp1.BirthYear);      //1983
            Console.WriteLine("emp2.BirthYear : {0}", emp2.BirthYear);      //1983

            emp1.BirthYear = 1978;                  //emp1.BirthYear 값 변경
            Console.WriteLine("=== emp1.BirthYear = 1978 값 변경 ===");
            Console.WriteLine("emp1.BirthYear : {0}", emp1.BirthYear);      //1978
            Console.WriteLine("emp2.BirthYear : {0}", emp2.BirthYear);      //1983
        }
    }
}
```

<class 로 정의된 Employee>

클래스로 정의된 경우는 아래 결과처럼 emp1.BirthYear 의 변경된 값이 복사된 변수에도 영향을 미치는 것을 볼 수 있다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch05_02
{
    //클래스 정의
    public class Employee
    {
        public int BirthYear;
        public string Name;
    }

    class Program
    {
        static void Main(string[] args)
        {
            //인스턴스 생성
            Employee emp1 = new Employee();
            emp1.Name = "김도현";
            emp1.BirthYear = 1983;

            Employee emp2 = emp1;

            Console.WriteLine("emp1.BirthYear : {0}", emp1.BirthYear);      //1983
            Console.WriteLine("emp2.BirthYear : {0}", emp2.BirthYear);      //1983
```

```
            emp1.BirthYear = 1978;                  //emp1.BirthYear 값 변경
            Console.WriteLine("=== emp1.BirthYear = 1978 값 변경 ===");
            Console.WriteLine("emp1.BirthYear : {0}", emp1.BirthYear);      //1978
            Console.WriteLine("emp2.BirthYear : {0}", emp2.BirthYear);      //1978
        }
    }

}
```

변수

다양한 자료형을 사용하고, 객체의 인스턴스를 생성하기 위해서는 값을 담을 곳이 필요하게 된다. 바로 그 그릇이 바로 변수이다.

변수를 선언할 때는 몇 가지 제약 사항이 존재한다.

- 문자, 언더바(_)로 시작 가능.
- 숫자로 시작할 수 없음.
- 예약어(int, string, if, else, class 등)는 사용할 수 없음.

위와 같은 규칙에 부합하는 어떤 변수명도 사용할 수 있으나, 보통 다음과 같은 명명 방식으로 구분을 짓는다.

- Pascal casing : ClickEvent 각 의미있는 단어를 대분자로 시작한다. 보통 상수, 네임스페이스, 클래스, 메서드 와 같은 이름을 지을 때 사용한다.
- Camel casing : currentLocation 과 같이 첫문자는 소문자, 나머지 의미 있는 단어들은 대분자로 시작한다. 낙타의 등 모양과 비슷하다는 의미에서 나왔으며, 일반적으로 변수 선언시 많이 사용하게 된다.
- 헝가리안 표기법 : 버튼의 경우 btnSave 와 같이 특정 접두사(예에서는 btn)를 붙여 변수명을 만든다. 일반적으로 컨트롤 명과 같이 특별한 구분이 필요할 때 사용하게 되며, 내장 형식(int 등) 선언에서는 사용을 피하는 것이 좋다.

C#은 크게 3 가지의 변수 선언 방법이 있다.

- 명시적 형식 사용 (예: int number = 7; List members = new List();)
- var (예 : var number = 7; var members = new List();)
- dynamic (예 : dynamic number = 7; dynamic members = new List();)

명시적 형식 사용

기본이 되는 선언 방식으로 코드에서 사용하고자 하는 형식을 명시적으로 선언을 하게 된다.

<var>

C# 3.0 에서 새롭게 등장한 형식으로 자바스크립트를 개발한 경험이 있다면 이미 본적이 있을 것이다. 자바스크립트와 마찬가지로 var 로 선언된 변수는 모든 형식을 사용할 수 있지만, 차이점은 한번 선언된 형식은 불변으로 선언된 이후에 다른 형식의 값으로 설정 할 수 없다.

그럼 왜 var 형식이 나오게 된 것일까? 이전 장에서 Nullable 형식에 대해 살펴 보았을 것이다. 이와 마찬가지로 var 형식도 DB 처리와 연관이 있다. 우리가 인라인 쿼리를 작성하는 것을 생각해 보자. 하나의 테이블에서 조회를 할 경우도 있지만, 특정 컬럼만 가지고 오거나, 여러 테이블과 조인을 하여 보고자 하는 형태로 쿼리를 작성하는 경우가 더 많을 것이다.

그럼 그 조회된 결과는 물리적으로 DB 에 존재하는 테이블인가? 아니다. 쿼리가 발생할 때 마다 동적으로 생성되는 것이다. 그럼 그 조회된 결과를 C# 코드에서 받을 때는 어떻게 처리 하여야 될까? C# 3.0 이전에는 DataTable 이란 곳에 담거나, 쿼리 컬럼과 일치하는 클래스를 미리 생성해 놓아야 했다. 만약 조회 쿼리가 동적으로 변한다면? 이 또한 난감함 문제이다. 이럴 경우 익명 형식(anonymous type)과 함께 사용하게 되면 위에서 했던 고민이 해결 된다. 익명 형식은 인라인 쿼리 결과처럼 클래스를 명시적으로 선언하지 않고도 사용하고자 하는 시점에 동적으로 생성이 가능하다. 이렇게 동적으로 생성된 익명 형식을 var 타입으로 선언 변수 값으로 설정하면 되는 것이다. 익명 형식은 다음 장에서 좀더 자세히 살펴 볼 수 있다.

var 형식은 모든 형식을 설정할 수 있지만 컴파일 시점에 형식이 확정 되므로 성능상의 이슈는 없다.

- 사용 가능한 형식
    - 내장 형식(built-in type) : int, string 등
    - 익명 형식(anonymous type)
    - 사용자 정의 형식(user-defined type) : class, struct 등
    - .NET FCL : Stream, List<T> 등

이처럼 사용이 편리한 var 형식도 몇 가지 제약 사항이 존재 한다.

- 선언시 유의사항
    - var temp; //초기화기 되지 않음
    - var temp = null; //특정한 형식을 가져야 함.
    - 메서드의 파라미터로 사용할 수 없음

<dynamic>

C# 4.0 이전에는 모든 타입이 컴파일 되는 시점에 확정이 되기 때문에 VSTO(Visual Studio Tools for Office)를 이용하여 엑셀과 같은 오피스 관련 개발과 같은 런타임에 타입이 확정되는 객체의 코드 작성이 어려웠다. 하지만 C# 4.0 에서 dynamic 이라는 새로운 타입으로 보다 쉽게 코드 작성이 가능하다. dynamic 형식을 사용하기 위해서는 var 와 마찬 가지로 변수 선언시 dynamic 으로 선언해 주면 된다.

```
dynamic value = "설정값";
```

엑셀 워크북을 오픈하는 예제 코드를 보자. 다음은 C# 4.0 이전 방식의 코드 이다. 사용하고자 하는 오피스 객체에 대해 다이나믹 룩업(Dynamic Lookup)을 할 수 없기 때문에 코드가 길다.

```
Excel.Workbook readWorkbook = excelAppMain.Workbooks.Open(workbookPath
, System.Type.Missing
, false
, System.Type.Missing
, System.Type.Missing
, System.Type.Missing
, System.Type.Missing
, System.Type.Missing
, ""
, true
, System.Type.Missing
, System.Type.Missing
, true
, System.Type.Missing
, System.Type.Missing);
Excel.Worksheet readSheet = (Excel.Worksheet)readWorkbook.ActiveSheet;
```

하지만 C# 4.0 에서는 다이나믹 룩업(Lookup)이 가능해져 상당히 코드가 간결해 진 것을 볼 수 있다.

```
Excel.Workbook readWorkbook = excelAppMain.Workbooks.Open(workbookPath);

Excel.Worksheet readSheet = readWorkbook.ActiveSheet;
```

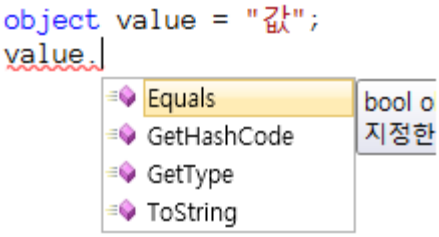
<object vs. var vs. dynamic>

여러 가지 변수 설정 방식 및 형식 설정에 대해 살펴 보았다. 하지만 기존의 object, var, dynamic 형식이 비슷한 면이 존재 하면서도 각각의 차이점이 존재한다.

공통점은 모두 다양한 형식을 값으로 설정이 가능하다.

차이점은 아래 그림을 보면 분명히 보인다.

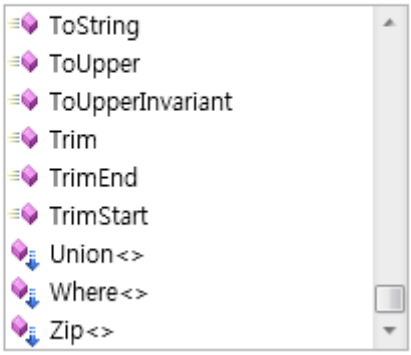
object 에 할당된 형식은 어떤 형식이든 object 로 변환이 된다. 인텔리센스를 보면 이해가 빠를 것이다. 다시 원본 형식을 사용하고 싶으면 형 변환 작업이 필요하다.



[object 로 선언하였을 경우 인텔리센스 표시]

var 형식은 값이 설정되는 시점에 형식을 추정하여 설정된 형식 그 자체 이다. 아래 인텔리센스를 보면 문자열 값을 설정 하였을 경우 문자열 관련 메서드가 보이는 것을 확인 할 수 있다.

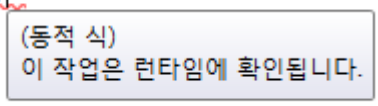
```
var value = "값";
value.
```



[var 로 선언하였을 경우 인텔리센스 표시]

dynamic 은 인텔리센스가 동작하지 않는다. 왜냐하면 컴파일 시점에는 어떤 형식인지 확인을 하지 않고 런타임 환경에서 확인을 하기 때문이다. 이러한 경우는 개발자가 사용하고자 하는 메서드 등의 멤버에 대해 정확히 알고 있다는 가정이 필요하다. 만약 잘못된 메서드를 사용하였어도 컴파일에서는 밸리데이션(Validation)을 하지 않기 때문에 아무 빌드 오류를 발생하지 않기 때문이다. 그러므로 성능상의 이슈와 런타임 환경에서 형식이 확정 되기 때문에 예기치 않은 오류를 야기 할 수 있다. 주의 깊게 사용하여야 한다.

```
dynamic value = "값";
value.
```



[dynamic 으로 선언하였을 경우 인텔리센스 표시]

	object	var	dynamic
설정 가능한 형식	모든 형식	null 등 일부를 제외한 대부분의 형식	모든 형식
설정 값의 원본 유지	값을 설정하면 object 형식 그 자체	원본 형식을 추정	개발 시점에서는 원본 형식을 알 수 없음
형식 확정	컴파일(object 자체)	컴파일	런타임
바인딩 방식	정적	정적	동적

[object, var, dynamic 비교]

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch05_03
{
    class Program
    {
        static void Main(string[] args)
        {
            string value1 = "값 1";
            var value2 = "값 2";
            dynamic value3 = "값 3";

            Console.WriteLine("value1 : {0}", value1);
            Console.WriteLine("value2 : {0}", value2);
            Console.WriteLine("value3 : {0}", value3);
        }
    }

}
```

상수

상수는 프로그램 전체에서 변경이 되지 않는 값을 설정할 때 사용 된다. 학창 시절 수학 시간을 생각해 보기 바란다. 원주율, 자연로드 e 등 수학에는 많은 상수들이 존재 한다. 이처럼 프로그램에서도 한번 설정하고 불변하는 값을 설정 할 수 있다.

C#에서 상수는 const 예약어를 통해 설정이 가능하다.

```
const float PI = 3.14F;
```

이와 더불어 C#에서는 const 와 비슷한 readonly 를 지원한다.

공통점은 모두 초기화 이후에는 값을 변경 할 수 없다. 하지만 const 는 선언할 때만 초기화될 수 있다. readonly 는 선언할 때 또는 생성자에서 초기화될 수 있다. 따라서 readonly 필드의 값은 사용된 생성자에 따라 다르다. const 는 컴파일 타임 상수, readonly 는 런타임 상수로 불리우기도 한다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch05_04
{
    class Program
    {
        //클래스 정의
        public class Product
        {
            public const int ConstPrice = 1000;      //const 필드 설정
            public readonly int ReadOnlyPrice;       //reonly 필드 설정

            public Product()
            {
                this.ReadOnlyPrice = 2000;          //생성자에서 초기값 설정
            }

            public Product(int price)
            {
                this.ReadOnlyPrice = price;         //외부에서 받은 값으로 설정
            }
        }

        static void Main(string[] args)
        {
            //상수값 가져오기 : 상수는 정적인 멤버 이므로 객체를 생성할 필요가 없다.
            Console.WriteLine("ConstPrice={0}", Product.ConstPrice);

            //기본 생성자로 설정된 값
            Product item1 = new Product();
            Console.WriteLine("new Product() : ReadOnlyPrice={0}", item1.ReadOnlyPrice);

            //생성자에서 특정값을 받음.
            Product item2 = new Product(3000);
            Console.WriteLine("new Product(3000) : ReadOnlyPrice={0}", item2.ReadOnlyPrice);
        }
    }
}
```

형 변환(Type Conversion / Casting)

데이터를 처리하다 보면 문자열에서 숫자로, 큰 숫자 형식에서 작은 숫자 형식으로 등 다양한 값의 변경이 필요하다. 숫자의 경우도 우리가 보기에는 같은 값이라도 int, short 과 같이 각각의 형식이 정해져 있다. 이러한 데이터 형식의 변환을 말한다.

<암시적(Implicit) 형 변환>

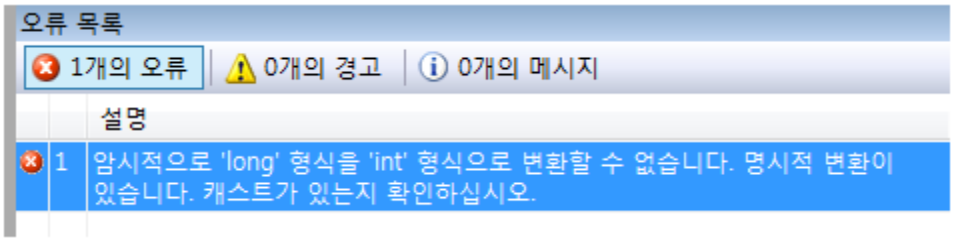
말 그대로 특별한 특별한 구문 작성 없이도 자동으로 형 변환이 일어 나는 것을 말한다. 다음 코드를 보면 int 형을 더 큰 long 형식으로 변환하고 있다. 보통 상위 형식으로 가는 업 캐스팅(Up Casting)이 일어난다. 반대의 경우인 다운 캐스팅(Down Casting)의 경우는 명시적으로 변환이 필요하다.

```
int number = 1;

long bigNumber = number;
```

<명시적(Explicit) 형 변환>

다운 캐스팅 혹은 object 에서 숫자와 같이 다른 형식으로 형 변환을 하려면 명시적인 형 변환 구문이 필요하다.

```
//다운 캐스팅

long bigNumber = 1;

int number = (int)bigNumber; // int number = bigNumber; -> 컴파일 오류

//object 형식을 int 형식으로 변경

object obj = 3;

number = (int)obj;
```



[다운 캐스팅을 명시적으로 하지 않았을 경우 오류]

<Convert 클래스>

닷넷 프레임워크에서 제공하는 기본 데이터 형식을 변환하기 위해서 Convert 클래스의 Convert.ToInt32()와 같은 정적(Static) 메서드를 제공하고 있다. Covert 클래스로 다양한 입력 값을 변환 가능하며, Boolean, Char, SByte, Byte, Int16, Int32, Int64, UInt16, UInt32, UInt64, Single, Double, Decimal, DateTime 및 String 형식을 지원한다.

Convert 클래스에 대한 세부 메서드에 대한 내용은 MSDN 에서 확인 가능하다.

(http://msdn.microsoft.com/ko-kr/library/system.convert_members(v=VS.95).aspx)

```
string str = "3";

int number = Convert.ToInt32(str);
```

<Parse 메서드>

특별히 변환하고자 하는 원본 형식이 문자열(String)일 경우 사용 가능한 Parse 메서드가 있다. Parse 메서드는 Int32.Parse()와 같이 각 형식의 정적 메서드로 존재한다.

```
string str = "3";

int number1 = Int32.Parse(str);

// 다음과 같이 C# 형식 표현(int)으로 사용해도 된다.

int number2= int.Parse(str);
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch05_05
{
    class Program
    {
        static void Main(string[] args)
        {
            string numberString = "123";
```

```
            //Convert 클래스 이용
            int number1 = Convert.ToInt32(numberString);

            //Parse 메소드 이용
            //int number2 = Int32.Parse(numberString);
            int number2 = int.Parse(numberString);

            Console.WriteLine("number1 : {0}", number1);
            Console.WriteLine("number2 : {0}", number2);
        }
    }

}
```

박싱(Boxing)과 언박싱(unboxing)

박싱(Boxing)은 값 형식(int 등)을 참조 형식(object)으로 변환하는 것을 말하며, 언박싱(unboxing)은 그와 반대로 참조 형식을 값 형식으로 변환 하는 것을 말한다. 박싱은 암시적, 언박싱은 명시적으로 이루어져야 한다. 박싱된 데이터는 힙(Heap) 영역에, 언박싱된 데이터는 스택(Stack) 영역에 각각 저장이 되므로 잦은 박싱, 언박싱 작업은 메모리 복사가 계속 일어 나므로 성능에 나쁜 영향을 미친다. 그러므로 꼭 필요한 경우에 최소한으로 사용하는 것이 좋다.

[C#강좌] 06.조건식

if, else if, else

if 문의 조건식이 참(true)이면 if 구문 블록을 실행하게 되고 거짓(false) 이면 else if, 혹은 else 문으로 분기 한다. if 문에서 else if 는 여러 번 올 수 있으나 else 는 한번만 올 수 있다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch06_01
{
    //열거형 선언
    public enum Country { Korea, China, Japan };

    class Program
    {
        static void Main(string[] args)
        {
            Country myCountry = Country.Korea;

            if (myCountry == Country.Korea)
            {
                Console.WriteLine("한국");
            }
            else if (myCountry == Country.Japan)
            {
                Console.WriteLine("일본");
            }
            else if (myCountry == Country.China)
            {
                Console.WriteLine("중국");
            }
            else        //Country 에 속하는 항목이 없을 경우
            {
                Console.WriteLine("선택된 나라가 없습니다.");
            }
        }
    }

}
```

switch

switch 문은 if 으로 동일한 처리가 가능하지만, 열거형과 같이 선택 항목 리스트 중에서 선택할 경우 효과적으로 코드를 작성 할 수 있다. switch 문은 각 선택 항목을 나타내는 case 문 다음에 break 가 나와야 하며, 모든 조건을 만족하지 않는 기본 처리를 위하여 default 문을 제공해 주고 있다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch06_02
{
    //열거형 선언
    public enum Country { Korea, China, Japan };

    class Program
    {
        static void Main(string[] args)
        {
            Country myCountry = Country.Korea;

            switch (myCountry)
            {
                case Country.Korea:
                    Console.WriteLine("한국");
                    break;
                case Country.China:
                    Console.WriteLine("일본");
                    break;
                case Country.Japan:
                    Console.WriteLine("중국");
                    break;
                default:                //Country 형식에 속하는 항목이 없을 경우
                    Console.WriteLine("선택된 나라가 없습니다.");
                    break;
            }
        }
    }

}
```

[C#강좌] 07.반복문

for

가장 많이 사용하게 되는 반복문 중 하나로 for(초기값;조건식;증감식) 형태를 가진다. 조건식을 만족하는 동안 루프를 돌며 처리를 하게 된다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch07_01
{
    class Program
    {
        static void Main(string[] args)
        {
            int result = 0;

            for (int i = 1; i <= 10; i++)
            {
                result += i;          //result = result + i; 축약 표현
            }

            Console.WriteLine("for 문 1~10 더하기 : {0}", result);
        }
    }

}
```

foreach

for 문과 비슷한 형태로 foreach(아이템 자료형 in 열겨형 객체) 형식을 가지며, 배열이나 컬랙션과 같은 곳에서 데이터를 하나씩 가져 올 때 유용하다. 아래 코드를 보면 먼저 배열에 1 부터 10 까지의 수를 설정하고 foreach 문을 통해 하나씩 가져 오는 것을 볼 수 있다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch07_02
{
    class Program
    {
        static void Main(string[] args)
        {
            int result = 0;
            int[] number = new int[10];

            for (int i = 0; i < 10; i++)
            {
                number[i] = i + 1;
            }

            foreach (int i in number)
            {
                result += i;
            }

            Console.WriteLine("foreach 문 1~10 더하기 : {0}", result);
        }
    }

}
```

while

while 문은 설정된 조건에 맞는 동안 계속해서 반복 실행되는 구조이다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch07_03
{
    class Program
    {
        static void Main(string[] args)
        {
            int result = 0;

            int startNumber = 1;
            int maxValue = 10;

            while (startNumber <= maxValue)
            {
                result += startNumber;
                startNumber++;        //startNumber = startNumber + 1; 축약 표현
            }

            Console.WriteLine("while 문 1~10 더하기 : {0}", result);
        }
    }

}
```

do-while

while 문 조건에 따라 한번도 실행이 되지 않을 수 있지만, do-while 문은 무조건 최소 1 회는 반드시 실행이 이루어 진다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch07_04
{
    class Program
    {
        static void Main(string[] args)
        {
            int result = 0;

            int startNumber = 1;
            int maxValue = 10;

            do
            {
                result += startNumber;
                startNumber++;
            } while (startNumber <= maxValue);

            Console.WriteLine("do-while 문 1~10 더하기 : {0}", result);
        }
    }

}
```

break, continue

반복문 혹은 조건문을 사용하다 보면 특정 조건에 따라 해당 루프를 빠져 나가야 하는 경우가 발생을 한다. 그러한 경우에 break. continue 문을 사용한다.

- break : 루프를 무조건 탈출
- continue : 현재 처리를 건너 뛰고 다음 루프 실행

다음 코드에서 보면 for 문은 1 부터 100 까 동작하도록 되어 있으나 i > 10 이면 루프를 완전히 탈출하고, i 가 짝수 이면 현재 처리를 건너 뛰고 다음 루프로 가도록 처리하고 있다. 결국 1 부터 10 까지 숫자는 홀수만 더한 결과가 나올 것이다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch07_05
{
    class Program
    {
        static void Main(string[] args)
        {
            int result = 0;

            for (int i = 1; i <= 100; i++)
            {
                if (i > 10)             //i 가 10 보다 크면 for 문을 빠져 나간다.
                {
                    break;
                }

                if ((i % 2) == 0)       //짝수이면 처리를 건너 뛴다.
                {
                    continue;
                }

                result += i;
            }

            Console.WriteLine("break-continue 1~10 중 홀수만 더하기 : {0}", result);
        }
    }

}
```

goto 문

일반적으로 코드는 위에서 아래도 순차적으로 실행이 이루어 진다. 하지만 간혹 로직을 구성하다 보면 순차적이 아닌 특정 위치로 바로 이동해서 다음 실행이 이루어 져야 하는 경우가 발생한다. 이러할 경우 goto 문을 통해 처리 가능하다. 하지만 goto 문은 처리의 가독성을 많이 저해 하므로 꼭 필요한 경우가 아니면 사용을 피하는 것이 좋다.

goto 문을 사용하려면 점프하고자 하는 곳에 아래 코드 예처럼 라벨명:(콜론)으로 설정을 한 후, goto 라벨명; 으로 지정을 하면 해당 goto 문을 만났을 경우 바로 라벨이 지정된 곳으로 바로 이동하며, 중간의 코드는 무시하게 된다. 또한 goto 문은 현재 위치에서 위, 아래 어느 곳으로도 이동이 가능하다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch07_06
{
    class Program
    {
        static void Main(string[] args)
        {
            int result = 0;

            for (int i = 1; i <= 10; i++)
            {
                result += i;

                if (i == 5)
                {
                    goto Jump;  //Jump 라벨로 이동
                }
            }

            Console.WriteLine("for 문 1~10 더하기 : {0}", result);

Jump:       //goto 문이 이동 할 라벨 지정
            Console.WriteLine("goto 점프 : {0}", result);
        }
    }

}
```

[C#강좌] 08.예외 처리

데이터 처리를 위해서는 시스템 내부적으로 연산하는 데이터도 있지만 외부에서 오게 되는 경우도 많이 존재한다. 사용자이든, 시스템 간 연계이든 많은 데이터들이 서로 통신을 하게 되고, 그러한 연산 과정에서 예기치 않은 오류 등이 많이 발생을 한다. 개발자가 모든 상황에 대해 인지하고 그에 맞는 해결책을 코드에 모두 작성해 놓으면 좋겠지만 사실상 불가능한 일이다. 또한 프로그램 자체의 문제가 아닌 메모리, OS 등 외부적인 요인에 의해서도 오류가 발생되기도 한다. 이러한 여러 예외 상황에서 프로그램이 완전히 동작하지 않는 상황을 미리 방지하기 위해 꼼꼼한 예외 처리는 필수 적이다.

try ~ catch ~ finally

try 문은 반드시 최소한 catch 문 혹은 finally 문 하나의 코드 블록은 존재하여야 한다.

try 블록에는 잠재적으로 예외를 발생할 수 있는 코드를 작성하면 되며, 만약 예외가 발생할 경우 catch 블록에서 처리를 하게 된다. 복수개의 catch 문이 올 경우 작은 범위의 예외에서 큰 범위로 작성해 주어야 한다. 한가지 주의 할 점은 닷넷 프레임워크 2.0 부터 StackOverflowException 은 catch 블록에서 처리할 수 없으며 바로 프로세스가 종료된다.

finally 문은 try 혹은 catch 문의 이후 반드시 실행이 되며 보통 리소스(DB 커넥션, 객체 Dispose 등) 정리 등의 코드를 작성한다.

예외 처리 작성 구분은 다음과 같은 유형으로 분류 할 수 있다.

- try ~ catch
- try ~ finally
- try ~ catch ~ finally

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch08_01
{
    class Program
    {
        static void Main(string[] args)
        {
            Foo("123");         //올바르게 처리
```

```
            Foo(null);          //ArgumentNullException 발생
            Foo("일이삼");      //Exception 발생
        }

        static void Foo(string data)
        {
            //복수개의 catch 문이 올 수도 있다.
            try
            {
                int number = Int32.Parse(data);
                Console.WriteLine("number : {0}", number);
            }
            catch (ArgumentNullException ex)    //입력 인자가 없을 경우
            {
                Console.WriteLine("ArgumentNullException 처리 : {0}", ex.Message);
            }
            catch (Exception ex)                //최상위 예외 형식
            {
                Console.WriteLine("Exception 처리 : {0}", ex.Message);
            }
        }
    }

}
```

throw

throw 문은 예외를 코드 작성자가 명시적으로 예외를 발생시킬 때 사용한다. 예외 처리의 중요성을 얘기하면서 모든 예외를 처리하지 않고 왜 예외를 명시적으로 발생을 시키는 것일까? 그 이유는 공용 모듈과 같은 프로그램 개발할 경우 만들어 놓은 API 를 다른 어디선가 참조하여 사용하게 된다. 하지만 이미 얘기 되었듯 입력 값에 대한 모든 밸리데이션을 할 수 없으며, 올바르지 않은 입력 인자 혹은 처리 값인 경우 명시적으로 예외를 발생 시키고, 사용자로 하여금 올바른 방향으로 처리할 수 있도록 유도 할 수 있다. 닷넷 FCL 도 수 많은 예외 클래스들이 존재하며, 잘못된 입력 값이 있을 경우 예외를 발생하여 오류의 원인을 명확히 확인 가능하게 제공한다.

다음 예제는 ShowMessage 메서드의 파라미터인 message 가 널(null)값이면 ArgumentNullException 을 발생하도록 되어 있다. ShowMessage 를 호출하는 곳에서는 ArgumentNullException 에 대한 예외 처리를 작성해 주면 된다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch08_02
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                ShowMessage("메시지를 출력 합니다.");
                ShowMessage(null);
            }
            catch (ArgumentNullException ex)
            {
                //예외처리
                Console.WriteLine("ArgumentNullException 처리 : {0}", ex.Message);
            }
        }

        //throw 구문 작성
        static void ShowMessage(string message)
        {
            if (message == null)
                throw new ArgumentNullException("message");

            Console.WriteLine(message);
        }
    }

}
```

[C#강좌] 09.배열

배열은 동일한 데이터 형의 연속된 공간을 말한다. 데이터를 처리하다 보면 특정 형식의 데이터들이 반복되는 것을 보게 된다. 대표적으로 DB 의 테이블을 생각해 보면, 테이블에 선언된 컬럼의 데이트 그룹인 행이 반복되게 된다.

배열 선언

배열은 차원(Rank), 인덱스(Index), 값(Element)를 가지고 있다.

- 차원(Rank) : 값 그룹의 개수(선언시 컴마(,)로 구분된 개수)
- 인덱스(Index) : 개별 값을 찾아 갈 수 있는 주소
- 요소(Element) : 개별 값

배열 인덱스의 값은 0 으로 시작한다.

1 차원/2 차원 배열

<1 차원 배열>

단일행으로 이루어진 배열로 Rank=1 인 배열이다. 다음 예제는 foreach 문과 for 문을 이용해 배열의 값에 접근하는 방법을 보여 주고 있다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch09_01
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] array1D = { 1, 2, 3, 4, 5 };

            Console.WriteLine("Rank:{0}, Length:{1}", array1D.Rank, array1D.Length);

            Console.WriteLine("1 차원 배열 - foreach");
            //foreach 문 사용
            foreach (int value in array1D)
            {
```

```
            Console.WriteLine(value);
        }

        Console.WriteLine("1 차원 배열 - for");
        //for 문 사용
        for (int i = 0; i < array1D.Length; i++)
        {
            Console.WriteLine("array[{0}] : {1}", i, array1D[i]);
        }
    }
  }
}
```

<2 차원 배열>

테이블 형태를 생각하면 된다. 행과 열이 동시에 존재한다. Rank=2 인 배열이다. string[,] array2D = new string[3,5]; 와 같이 컴마(,)로 차원을 구분한다. 아래 예제에서 각 차원의 개수를 동적으로 가져오기 위하여 GetLength 메소드를 이용하고 있다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch09_02
{
  class Program
  {
    static void Main(string[] args)
    {
      string[,] array2D = new string[3, 5];

      Console.WriteLine("2 차원 배열");
      //배열 값 설정
      for (int i = 0; i < array2D.GetLength(0); i++)      //array2D.GetLength(0) : 3
      {
        for (int j = 0; j < array2D.GetLength(1); j++)  //array2D.GetLength(1) : 5
        {
          array2D[i, j] = String.Format("{0}-{1}", i, j);
        }
      }

      //for 문 사용
      for (int i = 0; i < array2D.GetLength(0); i++)      //array2D.GetLength(0) : 3
      {
```

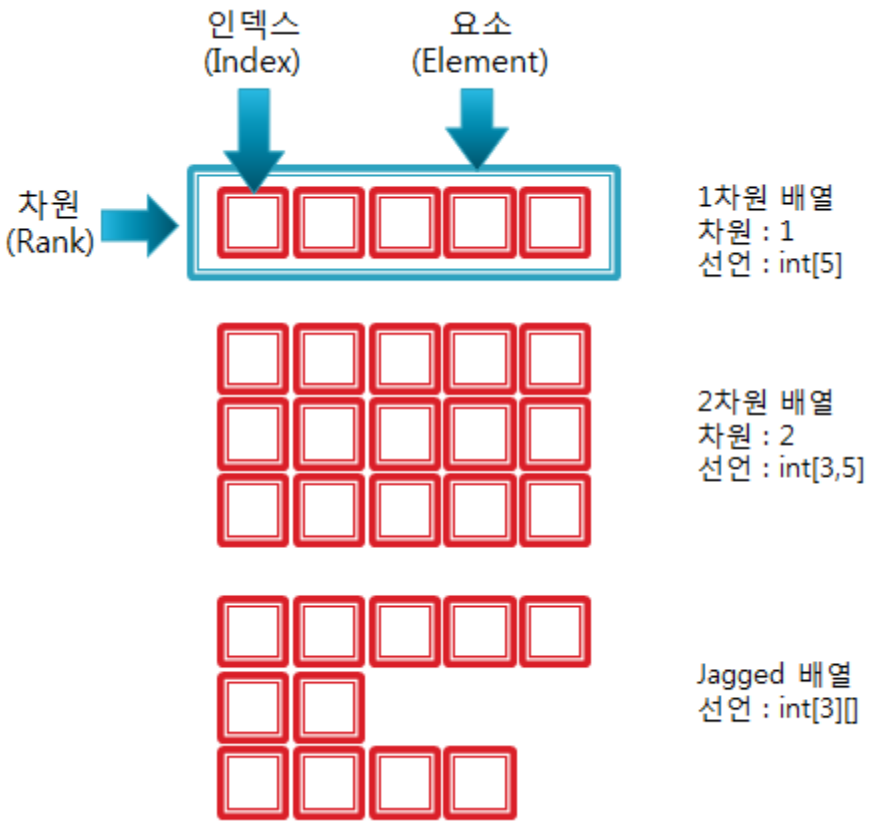
```
            for (int j = 0; j < array2D.GetLength(1); j++)  //array2D.GetLength(1) : 5
            {
                Console.Write("({0}) ", array2D[i, j]);
            }
            Console.WriteLine();
        }
    }
}

}
```

Jagged 배열

n 차원 배열은 행과 열의 개수가 동일하다. 하지만 Jagged 배열은 독립적인 1 차원 배열들의 복합이다. Jagged 배열 선언은 string[][] arrayJagged = new string[3][]; 와 같이 선언을 한다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch09_03
{
    class Program
    {
        static void Main(string[] args)
        {
            string[][] arrayJagged = new string[3][];
            arrayJagged[0] = new string[5];    //각 1 차원 배열 별로 크기가 다르게 지정.
            arrayJagged[1] = new string[2];
            arrayJagged[2] = new string[3];

            Console.WriteLine("Jagged 배열");
            //배열 값 설정
            for (int i = 0; i < arrayJagged.GetLength(0); i++)
            {
                for (int j = 0; j < arrayJagged[i].Length; j++)
                {
                    arrayJagged[i][j] = String.Format("{0}-{1}", i, j);
                }
            }

            //for 문 사용
            for (int i = 0; i < arrayJagged.GetLength(0); i++)
            {
                for (int j = 0; j < arrayJagged[i].Length; j++)
                {
```

```
                Console.Write("({0}) ", arrayJagged[i][j]);
            }
            Console.WriteLine();
        }
    }
  }
}
```



[배열의 구조]

배열 복사

배열을 값을 다른 배열에 복사해야 할 경우가 발생을 한다. 배열의 복사는 Array 클래스의 정적 메소드인 Coopy 를 사용거나, Buffer.BlockCopy 를 이용할 수 있다. 아래 예제를 실행하여 보면 Buffer.BlockCopy 가 좀더 좋은 성능을 보여 준다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;

namespace SqlerCSharp.Ch09_04
{
    class Program
    {
        static void Main(string[] args)
        {
            Stopwatch sw1 = Stopwatch.StartNew();
            for (int i = 0; i < 10000000; i++)
            {
                int[] source = new int[100];
                int[] target = new int[100];

                Buffer.BlockCopy(source, 0, target, 0, 100);
            }
            sw1.Stop();

            Stopwatch sw2 = Stopwatch.StartNew();
            for (int i = 0; i < 10000000; i++)
            {
                int[] source = new int[100];
                int[] target = new int[100];

                Array.Copy(source, 0, target, 0, 100);
            }
            sw2.Stop();

            Console.WriteLine("Buffer.BlockCopy : {0} Milliseconds", sw1.ElapsedMilliseconds);
            Console.WriteLine("Array.Copy : {0} Milliseconds", sw2.ElapsedMilliseconds);
        }
    }
}
```

배열 정렬

배열에 입력된 값을 정렬 하려면 C# 3.0 부터는 LINQ 를 이용 가능 하지만 간단한 정렬은 Array 클래스의 Sort 메소드를 이용할 수 있다. 초기값이 있는 배열을 오름 차순으로 정렬을 하며, Reverse 메소드를 통해 내림 차순 정렬이 가능하다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch09_05
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] array1 = new int[] { 3, 7, 6, 2, 8, 9, 5, 1, 4 };

            Console.WriteLine("==정렬 전==");
            foreach (var item in array1)
            {
                Console.Write("{0} ", item);
            }

            Console.WriteLine("₩r₩n==Array.Sort==");
            Array.Sort(array1);
            foreach (var item in array1)
            {
                Console.Write("{0} ", item);
            }

            Console.WriteLine("₩r₩n==Array.Reverse==");
            Array.Reverse(array1);
            foreach (var item in array1)
            {
                Console.Write("{0} ", item);
            }

            Console.WriteLine();
        }
    }

}
```

[C#강좌] 10.문자열 다루기

String 이란?

프로그램 개발을 진행할 경우 가장 많이 사용하게 되는 자료형 중 하나가 아닐까 싶다. 내부적으로 처리된 결과를 사용자 화면에 출력을 하거나 이기종간의 통신을 위한 XML 의 기본 자료형이 되기도 한다. 닷넷 프레임워크에서 string 은 유니코드 기반 문자의 집합으로 영어권과 비영어권 문자의 구분이 없이 2 바이트(byte)이다.

string 은 한번 생성된 객체는 read-only 이다. 다음 코드를 보자. str1 이란 문자열 생성 후 str1 에 값을 추가 하였다. str1 은 같은 객체일까? 아니다. 변수의 명을 같을 지라도, 내부적으로 새로운 객체를 생성해 값을 생성 후 복사하는 과정이 이루어 진다. 그러므로 보통 값을 반복적으로 자주 변할 경우 string 으로 처리하기 보다는 StringBuilder 를 사용할 것을 권고하고 있다.

```
string str1 = "문자열";

str1 += "추가";
```

string 은 System.String 에 정의 되어 있으며, 유용한 메소드 및 프로퍼티를 가지고 있다. 다음은 많은 string 의 멤버 중 주요한 몇 가지 이다. string 의 전체 멤버에 대한 내용은 MSDN 을 참고하기 바란다.(http://msdn.microsoft.com/ko-kr/library/system.string.aspx)

멤버	유형	설명
Length	프로퍼티	현재 문자열의 문자 수를 가져온다.
Format	메소드	문자열은 지정된 서식 표현으로 변환한다.
Replace	메소드	문자열 내용을 지정된 문자열로 변경한다.
Split	메소드	문자열의 특정 구분자를 기준으로 문져열 배열을 반환한다.
Substring	메소드	문자열 일부를 자른다.
ToLower	메소드	문자열 전체를 소문자로 변환한다.
ToUpper	메소드	문자열 전체를 대문자로 변환한다.
Trim	메소드	현재 문자열의 앞뒤 공백 혹은 지정된 특정 문자를 제거 한다.

[string 주요 멤버]

문자열 포맷(Console.WriteLine, String.Format)

문자열을 있는 그대로의 값으로도 표현을 하지만, 금액을 나타내는 경우 천의 자리마다 컴마(예: 1,000)를 찍는등 사용자가 좀더 쉽게 이해 할 수 있도록 ToString 메소드나, String.Format, Console.WriteLine 과 같은 포맷을 지정할 수 있는 메소드를 이용해서 다양한 형태의 포맷을 지정하게 된다.

<포맷 지정하기>

문자열의 포맷은 "{" 와 "}" 사이에 지정을 하게 된다. 위치 지시 번호는 값이 오는 순서로 0 부터 지정을 하게된다. 두 번째는 문자열의 전체 길이를 지정한다. 길이를 지정할 경우 양수는 오른쪽 정렬, 음수는 왼쪽 정렬을 나타낸다.

{위치 지시 번호, 문자열 정렬 지정} 예:{0,5}

{위치 지시 번호:문자열 포맷} 예:{0:0000#}

만약 "{", "}" 를 출력 하려면 "{{", "}}" 와 같이 두 번을 작성해 주면 된다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch10_01
{
    class Program
    {
        static void Main(string[] args)
        {
            int number1 = 1;
            int number2 = 123;

            Console.WriteLine("number 1 : |{0,5}|, number2 : {1:0000#}"
                , number1, number2);        //number 1 : |    1|, number2 : 00123
            Console.WriteLine("number 1 : |{0,-5}|, number2 : {1:0000#}"
                , number1, number2);        //number 1 : |1    |, number2 : 00123

            int number3 = 12345;
            Console.WriteLine("number 3 : {{ {0} }}", number3); // number 3 : { 12345 }
        }
    }
}
```

<천 단위 컴마(,) 지정>

회계처리를 하게 되는 프로그램은 대부분 천 단위 마다 컴마를 찍고 특별히 마이너스 금액의 경우 -12,345 가 아닌 (12,345) 과 같은 표현을 사용하는 경우가 있다. 포맷을 지정할 때 세미콜론(;)으로 양수, 음수의 포맷을 같이 지정하는 방법으로 쉽게 처리 가능하다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch10_02
{
    class Program
    {
        static void Main(string[] args)
        {
            int number1 = 12345;
            int number2 = -12345;

            Console.WriteLine("number 1 : {0:#,#;(#,#)}", number1);            //number1 : 12,345
            Console.WriteLine("number 2 : {0}", number2.ToString("#,#;(#,#)")); //number2 : (12,345)
        }
    }

}
```

<특정 길이까지 남는 앞 부분을 0 으로 채우기>

코드값이 숫자값으로 되어 있을 경우 보통은 자릿수를 맞추게 된다. 예를 들어 코드가 5 자리이고 입력값이 123 이면 00123 과 같이 표현을 하게 된다. 포맷을 지정할 경우 자릿수만큼 0 을 입력해 주면 된다. "#"은 10 진수 자리를 나타낸다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch10_03
{
    class Program
    {
        static void Main(string[] args)
        {
            int number1 = 1;
```

```
            int number2 = 123;
            int number3 = 1234;

            Console.WriteLine("number 1 : {0:000##}", number1);             // number1 : 00001
            Console.WriteLine("number 2 : {0:0000#}", number2);             // number2 : 00123
            Console.WriteLine("number 3 : {0}", number3.ToString("00000")); // number3 : 01234
        }
    }
}
```

<축약 문자열(@)>

폴더 경로와 같은 "₩"와 같은 특수 문자를 표현할 경우 "₩₩"와 같이 두 개를 입력해야 한다. 축약 문자열을 표현하는 "@"를 사용하면 특수 문자가 아닌 입력 문자열 차체로 인식을 한다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch10_04
{
    class Program
    {
        static void Main(string[] args)
        {
            string systemFolder1 = "C:₩₩Windows₩₩System32";
            string systemFolder2 = @"C:₩Windows₩System32";

            Console.WriteLine("systemFolder1 : {0}", systemFolder1);
            Console.WriteLine("systemFolder2 : {0}", systemFolder2);
        }
    }

}
```

앞에서 언급된 내용 이외에 DateTime 및 숫자에 대한 다양한 포맷을 지정하는 방법은 아래의 MSDN 문서에서 찾아 볼 수 있다.

- 형식 서식 지정(http://msdn.microsoft.com/ko-kr/library/26etazsy.aspx)
- 사용자 지정 숫자 서식 문자열 (http://msdn.microsoft.com/ko-kr/library/0c899ak8.aspx)
- 사용자 지정 날짜 및 시간 형식 문자열 (http://msdn.microsoft.com/ko-kr/library/8kb3ddd4.aspx)

StringBuilder

위에서 얘기 한 것처럼 string 은 한번 생성이 되면 Read-Only 객체이다. 그러므로 값이 계속해서 변하는 빈도가 많을 경우 성능에 좋지 않은 영향을 미칠 수 있다. StringBuilder 는 값의 복사 없이 버퍼의 크기만큼 동적으로 값 변경을 처리 할 수 있다. StringBuilder 는 System 네임스페이스가 아닌 System.Text 네임스페이스에 존재하며, 프로그램 상단에 using System.Text; 와 같이 미리 선언하여야 한다.

아래 코드는 string 과 StringBuilder 의 성능을 비교한 것으로 각 실행 시간을 측정한 것이다. 실제 실행 시간은 컴퓨터의 사양에 따라 달라 지지만, StringBuilder 가 더 성능이 빠른 것을 확인 할 수 있다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;          //StringBuilder
using System.Diagnostics;   //Stopwatch

namespace SqlerCSharp.Ch10_05
{
    class Program
    {
        static void Main(string[] args)
        {
            string text1 = "";

            Stopwatch sw1 = Stopwatch.StartNew();
            for (int i = 0; i < 50000; i++)
            {
                text1 += i.ToString();
            }
            sw1.Stop();

            StringBuilder text2 = new StringBuilder();

            Stopwatch sw2 = Stopwatch.StartNew();
            for (int i = 0; i < 50000; i++)
            {
                text2.Append(i.ToString());
            }

            sw2.Stop();

            Console.WriteLine("String : {0} Milliseconds", sw1.ElapsedMilliseconds);
            Console.WriteLine("StringBuilder : {0} Milliseconds", sw2.ElapsedMilliseconds);
        }
    }

}
```

이처럼 StringBuilder 가 string 보다 우수한 것으로 보인다. 그러면 모든 문자열 처리를 할 때 StringBuilder 를 사용하면 되지 않을까? 모든 일에 일장일단이 있는 것처럼 StringBuilder 도 다음의 사항을 고려하여 올바르게 사용하여야 한다.

- 문자열의 변경 처리가 많이 일어 나지 안을 경우는 성능 차이가 미미하다.
- 적은 변경을 StringBuilder 로 선언할 경우 코드의 가독성이 좋지 않다.
- 문자열의 변화가 클 경우 StringBuilder 의 생성자에서 사이즈를 미리 정해 놓아야 한다. 왜냐하면 기본 생성자에서 정한 크기를 벗어날 경우 StringBuilder 의 사이즈를 확대하면서 값 복사 등의 연산이 일어나게 되는데, 이러한 작업은 비용이 많이 발생하는 작업이다. 예 : StringBuilder text = new StringBuilder(10000);

[C#강좌] 11.클래스 1 - 선언하기

클래스(Class)

클래스는 C# 프로그램에서 독립적으로 존재하는 최소단위로 다양한 멤버를 정의할 수 있으며, 객체지향(Object Oriented) 프로그램의 기본 구조라 할 수 있다. 우리 주변에 자동차를 생각해 보자. 자동차를 만들기 위한 설계도가 있고, 그 설계도에 따라 실제 만들어진 구매 가능한 차가 존재할 것이다. 여기서 설계도는 클래스이고 실제 만들어진 차가 바로 인스턴스(Instance)라 할 수 있다. 잘 정의된 클래스를 바탕으로 인스턴스가 프로그램에서 실제 다양한 처리를 하게 된다.

클래스 선언은 다음과 같이 어셈블리 혹은 네임스페이스에서 클래스에 접근을 관리하는 접근 제한자를 지정할 수 있으며, 필드, 메소드, 이벤트와 같은 멤버를 정의 할 수 있다.

접근제한자 class 클래스명

{

//멤버 정의

}

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch11_01
{
    //클래스 선언
    public class Employee
    {
        public string Name;
    }

    class Program
    {
        static void Main(string[] args)
        {
            Employee emp = new Employee();
            emp.Name = "김수영";
```

```
            Console.WriteLine("emp.Name : {0}", emp.Name);
        }
    }

}
```

클래스 멤버

클래스 멤버는 크게 데이터를 저장을 하는 부분과 기능적인 부분으로 분류가 가능하다.

- 데이터 저장
    - 필드(Field)

클래스 레벨에서 정의되는 변수, 상수 등을 말한다.

- 기능적인 분류
    - 메소드(Method)

다양한 기능을 처리하게 한다. 처리결과를 반환하거나, 결과의 반환 없이 연산을 하기도 한다.

- 프로퍼티(Property)

멤버 변수와 비슷하지만, 멤버 변수는 값을 설정할 때 밸리데이션을 처리할 수 없다. 하지만 프로퍼티는 get/set 을 통하여 입/출력을 관리 할 수 있으며, 값 설정전에 다양한 처리가 가능하다.

- 생성자

클래스의 인스턴스를 생성하기 위해 여러 가지 초기화 작업을 수행하는 코드를 작성하게 된다. 객체 생성시 반드시 호출이 되는 특수 메소드이다.

- 인덱서

배열과 비슷한 개념으로 클래스의 객체를 배열과 같은 형태로 정의해서 사용가능하며, 다양한 컬랙션 객체 등에서 사용된다.

- 이벤트

버튼 클리과 같은 컨트롤 혹은 다양한 클래스들의 이벤트를 정의하고 사용 가능하다.

- 델리게이트

메소드가 처리해야 할 일을 위임 받아 처리를 하게 되며, 보통 이벤트와 함께 사용이 된다.

접근 제한자

클래스에 정의된 멤버는 클래스 내에서만 사용을 제한 하거나 외부에서 아무 제한 없이 호출되거나, 특정 조건(상속과 같은)에서만 사용이 가능하도록 할 수 있다. 이러한 기능을 위하여 각 멤버에 접근 제한자(Access Modifier)를 지정하게 된다.

- public

클래스 내/외부, 어셈블리, 네임스페이스 어디에서도 제한 없이 접근이 가능하다.

- private

클래스 내부에서만 접근이 가능하며, 외부에서는 접근을 할 없다. 보통 외부에서 값을 변경하지 못하도록 보호 하는 경우에 사용하게 된다.

- protected

클래스 내부 혹은 상속으로 이루어진 파생 클래스에서 접근이 가능하다.

- internal

동일 어셈블리에서는 public 과 같은 속성을 가지며, 다른 어셈블리에서는 호출을 할 수 없다.

- protected internal

protected 와 internal 의 속성을 모두 가지고 있는 것으로, 파생 클래스 혹은 동일 어셈블리 내에서 접근이 가능하다.

[C#강좌] 12.클래스 2 - 다양한 클래스 선언

생성자(Initializer)

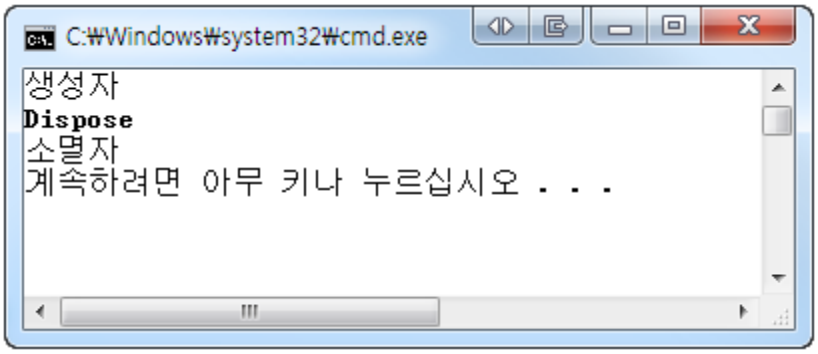
생성자는 클래스가 최기화 될 때 실행되는 코드이다. 생정자는 클래스명과 동일한 이름으로 메소드처럼 정의를 한다. 다음 예제를 보면 Employee 클래스는 birthYear, name 은 인자로 받아 초기화를 하게 되어 있다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch12_01
{
    public class Employee
    {
        public int BirthYear;                       //필드 선언
        public string Name;                          //필드 선언

        public Employee() { }                        //기본 생성자 선언

        public Employee(int birthYear, string name)      //생성자 오버로딩(overloading)
        {
            this.BirthYear = birthYear;              //초기화 코드
            this.Name = name;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Employee emp = new Employee();
            emp.BirthYear = 1988;
            emp.Name = "강철중";

            Console.WriteLine("BirthYear : {0}, Name : {1}", emp.BirthYear, emp.Name);
        }
    }

}
```

소멸자(Finalizer)

컴퓨터의 자원은 한정 적이다. 그러므로 작업이 마무리 되었으면 연산을 위하여 사용되었던 리소스 및 메모리를 정리하여야 한다. 소멸자는 가비지 컬렉터(Garbage Collector)가 수행 직전 실행이 되는 특수한 메소드로 클래스 명 앞에 ~(tilde)를 붙이면 된다. 한가지 주의 할 점은 닷넷 프레임워크는 기본적으로 Garbage Collector 를 수집하는 시점을 임의로 조정할 수 없으므로 소멸자가 실행되는 시점 또한 통제를 할 수 없다. 소멸자와 더불어 C# 에서는 IDisposable 인터페이스 상속을을 통하여 리소스 정리하는 방법이 존재한다. 일반적으로 소멸자에서는 unmanaged 객체를, IDisposable 인터페이스에서는 managed 객체의 리소스 정리 코드를 작성하는 것이 일반 적이다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch12_02
{
    //생성자와 소멸자의 각 시점 코드
    public class InitFinalizer : IDisposable
    {
        public InitFinalizer()
        {
            Console.WriteLine("생성자");
        }

        ~InitFinalizer()
        {
            Console.WriteLine("소멸자");
        }

        public void Dispose()
        {
            Console.WriteLine("Dispose");
        }
    }


    class Program
    {
        static void Main(string[] args)
        {
            //호출
            InitFinalizer initTest = new InitFinalizer();
            initTest.Dispose();
        }
    }
}
```



[생성자, 소멸자, Dispose 처리 순서]

Sealed 클래스

객체지향 프로그램에서 가장 많이 얘기 되는 내용 중에 하나가 상속 관계이다. 인터페이스, 추상클래스뿐만 아니라 클래스 간의 상속을 통하여 다양한 설계 및 기능 구현이 가능하다. 하지만 모든 클래스가 상속이 필요한 것은 아니며, 정의된 클래스가 더 이상 상속이 필요지 않을 수도 있을 것이다. 이러할 경우 Sealed 클래스로 선언을 한다. Sealed 클래스로 선언을 하면, 다른 클래스의 부모 클래스가 될 수 없으며, 더 이상 상속 하지 않는다는 명시적 표현이다. Sealed 클래스로 선언을 하면 인스턴스 생성에 성능 향상을 가져 올 수 있다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch12_03
{
   //sealed 클래스 선언
   public sealed class Employee
   {
      public int BirthYear;                    //필드 선언
      public string Name;                       //필드 선언

      public Employee() { }                     //기본 생성자 선언

      public Employee(int birthYear, string name)     //생성자 오버로딩(overloading)
      {
```

```
            this.BirthYear = birthYear;              //초기화 코드
            this.Name = name;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Employee emp = new Employee();
            emp.BirthYear = 1988;
            emp.Name = "강철중";

            Console.WriteLine("BirthYear : {0}, Name : {1}", emp.BirthYear, emp.Name);
        }
    }
}
```

Static 클래스

클래스의 인스턴스를 생성할 필요 없이 클래스의 멤버가 모두 정적(Static)인 요소로만 구성이 될 겨우 클래스를 Static 클래스로 선언이 가능하다. Static 클래스로 선언을 하면 인스턴스 생성이 불가하다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch12_04
{
    //Extension Method 선언을 위한 static 클래스 정의
    public static class ExtensionMethod
    {
        public static bool IsEven(this int number)
        {
            //모듈러연산(%)이 아닌 비트연산 이용
            bool isEven = ((number & 0x1) == 0) ? true : false;

            return isEven;
        }
    }
```

```
    class Program
    {
        static void Main(string[] args)
        {
            int number = 3;
            Console.WriteLine("{0} 짝수 ? {1}", number, number.IsEven());
        }
    }

}
```

Partial 클래스

클래스의 기능이 많을수록 작성되는 코드가 많아 진다. 그리고 프로젝트의 규모가 클수록 형상관리를 통하여 소스(Source)관리를 하게 된다. 그리고 기본적으로 클래스는 물리적으로 같은 파일내에 모든 코드가 작성이 되어야 한다. 하지만 WinForm, WPF 같은 디자인 관련 코드와 비즈니스 관련 코드가 하나의 클래스에 존재할 경우, 여러명이 동시에 작업을 할 수가 없다. 이러한 문제점을 Partial 클래스를 통해서 해결 가능하다. Partial 클래스는 하나의 클래스를 물리적으로 서로 다른 파일에서 동시에 구현이 가능하다. 또한 Partial 클래스는 Partial 메소드를 선언할 수 있다. 단 Partial 메소드 선언에는 액세스 한정자 또는 virtual, abstract, override, new, sealed 나 extern 한정자를 사용할 수 없다.

```
//PartialTest.partial01.cs
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch12_05
{
    public partial class PartialTest
    {
        public void PartialTest01Method()
        {
            Console.WriteLine("PartialTest01Method");
        }

        partial void PartialTestPartialMethod();
    }
}
```

```
//PartialTest.partial02.cs
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch12_05
{
    public partial class PartialTest
    {
        public void PartialTest02Method()
        {
            Console.WriteLine("PartialTest02Method");
            PartialTestPartialMethod();
        }

        partial void PartialTestPartialMethod()
        {
            Console.WriteLine("PartialTestPartialMethod 구현");
        }
    }
}
```

```
//호출
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch12_05
{
    class Program
    {
        static void Main(string[] args)
        {
            PartialTest test = new PartialTest();
            test.PartialTest01Method();
            test.PartialTest02Method();
        }
    }
}
```

Object Initializer

훌륭한 설계도가 있으면 구체적인 제품을 만들어야 한다. 정의된 클래스에서 실제 사용 시점에 인스턴스를 생성해야 한다. Employee 클래스의 인스턴스를 생성할 때 일반적으로 다음과 같이 작성을 한다.

Employee emp = new Employee();

emp. BirthYear = 1988;

emp.Name = "강철중";

하지만 C# 3.0 부터 Object Initializer 를 통하여 다음과 같이 인스턴스 생성과 함께 필드 초기화를 할 수 있다. DB 처리를 ORM(Object Relational Mapping)을 이용할 때 정의된 클래스는 DB 의 테이블 스키마가 되며, 생성된 인스턴스는 DB 에 입력된 행이라 할 수 있다. 이렇듯 인스턴스 생성 시점에 값을 초기화 하는 코드가 필요할 경우 코드의 가독성도 확보 하면서 간결한 코드 작성이 가능해 진다.

Employee emp = new Employee

{

BirthYear = 1988,

Name = "강철중"

};

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch12_06
{
    public class Employee
    {
        public int BirthYear;                    //필드 선언
        public string Name;                       //필드 선언

        public Employee() { }                     //기본 생성자 선언
```

```
    public Employee(int birthYear, string name)      //생성자 오버로딩(overloading)
    {
        this.BirthYear = birthYear;                  //초기화 코드
        this.Name = name;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Employee emp = new Employee();
        emp.BirthYear = 1988;
        emp.Name = "강철중";

        Console.WriteLine("BirthYear : {0}, Name : {1}", emp.BirthYear, emp.Name);

        //Object Initializer
        Employee emp2 = new Employee
        {
            BirthYear = 1888,
            Name = "홍길동"
        };

        Console.WriteLine("BirthYear : {0}, Name : {1}", emp2.BirthYear, emp2.Name);
    }
}

}
```

[C#강좌] 13.프로퍼티

프로퍼티는 사용하는 입장에서는 클래스의 필드와 차이점이 없어 보인다. 하지만, 필드의 경우 값이 설정되는 시점에 밸리데이션을 할 수 없으며, 읽기 전용, 쓰기 전용과 같은 제한을 할 수 없다. 객체 지향 관점에서는 불필요한 정보를 숨길 수 있는 정보은닉의 효과도 가져 온다.

프로퍼티 선언

프로퍼티는 get/set 구문을 사용하며 선언을 하며, get/set 구문에 public, private 와 같은 접근 제한자 설정이 가능하다. 아래 예제에서는 set 구문에서 입력값에 대한 밸리데이션 처리를 하였다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch13_01
{
    public class Employee
    {
        private int birthYear;
        private string name;

        public int BirthYear
        {
            get
            {
                return this.birthYear;
            }
            set
            {
                //입력 범위가 아닐 경우 예외 발생
                //별도 로직 처리가 가능하다.
                if (value < 1900)
                {
                    throw new ArgumentException("1900 년 이상만 지정이 가능합니다.");
                }

                this.birthYear = value;
            }
```

```
        }

        public string Name
        {
            get
            {
                return this.name;
            }
            set
            {
                this.name = value;
            }
        }

        //프로퍼티에 특별한 로직이 없을 경우 자동 구현 프로퍼티(Auto Implemented Property) 사용
        //public int BirthYear { get; private set; }
        //public string Name { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Employee emp = new Employee();
                emp.BirthYear = 1888;           //예외 발생 : 1900 이상만 가능
                emp.Name = "홍길동";

                Console.WriteLine("BirthYear : {0}, Name : {1}", emp.BirthYear, emp.Name);
            }
            catch (ArgumentException ex)
            {
                Console.WriteLine("예외 발생 : {0}", ex.Message);
            }
        }
    }

}
```

자동 구현 프로퍼티(Auto Implemented Property)

DB 의 스키마를 위한 클래스를 만들 경우 보통은 get/set 구문에서 특별한 처리 없이 다음과 같이 작성을 한다. 하지만 특별한 처리 구문이 없을 경우 작성되는 코드를 줄이기 위해 C# 3.0 부터 지원하는 자동 구현 프로퍼티를 사용하면 보다 간결한 코드를 작성할 수 있다. 자동 구현 프로퍼티는 별도 구문 없이 get/set 으로만 선언을 하면 되며, 반드시 get/set 이 한 쌍으로 존재하여야 한다. 만약 읽기 전용일 경우는 set 구분의 접근 제한자를 private 로 선언해 주면 된다.

```
public class Employee

{

public int BirthYear { get; private set; }

public string Name { get; set; }

}
```

[C#강좌] 14.인덱서(Indexer)

프로퍼티가 클래스의 필드처럼 사용이 되는 것처럼 인덱서는 클래스 자체를 배열과 같이 사용할 수 있도록 선언하는 것이다. String 클래스처럼 인덱스를 통해 값에 접근을 할 수 있다.

선언은 this 구분에 get / set 프로퍼티를 정의해 주면 된다. 다음은 제네릭 형태의 인덱서를 구현한 것이다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch14_01
{
    //제네릭(Generic) 형태의 인덱서 선언
    public class Product<T>
    {
        public T[] ProductItem;

        //기본 생성자일 경우 기본 배열 크기 설정
        public Product() : this(10) { }

        public Product(int size)
        {
            this.ProductItem = new T[size];
        }

        public T this[int index]
        {
            get
            {
                return ProductItem[index];
            }
            set
            {
                ProductItem[index] = value;
            }
        }
    }
```

```
    class Program
    {
        static void Main(string[] args)
        {
            Product<string> item = new Product<string>();
            item[0] = "꿀과메기";
            item[1] = "새욱깡";
            item[2] = "오징어아몬드";

            Console.WriteLine("item[0] : {0}", item[0]);
            Console.WriteLine("item[{0}] : {1}", item.ProductItem.Count() - 1
            , item[item.ProductItem.Count() - 1]);

            //오류 : 기본 생성자에서는 크기가 10 으로 정의
            //Console.WriteLine("item[11] : {0}", item[11]);
        }
    }

}
```

[C#강좌] 15.메소드

메소드 선언

메소드는 반환값, 파리미터 목록을 가질 수 있다. 반환값이 없는 메소드일 경우 반환되는 형식이 아닌 void 라고 표시를 하게 된다. 그리고 클래스의 인스턴스를 생성하여야만 사용하는 일반적인 형식과 정적인 메소드로 나눌 수 있다. 정적인 메소드는 static 키워드로 명시를 하여 선언한다. 정적인 메소드는 클래스의 인스턴스 생성 없이 바로 클래스명.메소명 으로 사용하게 된다.

접근제한자 반환형식 메소드명(파라미터)

{

//코드작성

}

ref, out, params

메소드에서 파라미터를 선언할 때 다음과 같은 특별한 키워드 사용이 가능하다.

- ref : 파라미터를 참조형식으로 사용하는 것으로 연산하는 메소드에서 파라미터가 변경이 되면 원본의 값도 같이 변화가 일어난다.
- out : ref 와 마찬가지로 참조 형식의 파라미터 이지만 출력 전용으로 호출하는 입장에서는 초기화할 필요가 없으며, 연산하는 곳에서는 반드시 초기화 하여야 한다.
- params : 파라미터가 고정된 형태가 아니라 가변적인 것으로 넘길 수 있습니다. 배열과 다른 점은, 파라미터를 배열로 선언을 하면 배열 그 자체 이지만, params 는 동일한 형식의 파라미터 개수가 하나 이상으로 선언가능 하다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch15_01
{
    class Program
    {
```

```
        //ref
        public static void PassByRef(ref int number)
        {
            number++;
        }

        //out
        public static void PassByOut(out int number)
        {
            number = 1;
            number += 2;
        }
        //params
        public static void PassByParams(params int[] args)
        {
            Console.WriteLine("===PassByParams===");

            foreach (var item in args)
            {
                Console.WriteLine(item);
            }
        }

        static void Main(string[] args)
        {
            //ref 호출
            int number1 = 1;
            Console.WriteLine("number1 초기값 : {0}", number1);
            PassByRef(ref number1);
            Console.WriteLine("PassByRef(ref number1) : {0}", number1);

            //out 호출 : 인자를 초기화 하지 않음
            int number2;
            PassByOut(out number2);
            Console.WriteLine("PassByOut(out number2) : {0}", number2);

            //params 호출
            PassByParams(1);
            PassByParams(1, 2);
            PassByParams(new int[] { 1, 2, 3 });    //파라미터가 가변이기 때문에 배열도 가능
        }
    }

}
```

메소드 오버로딩(Overloading)

메소드 오버로딩은 동일한 메소드명을 사용하면서 파라미터가 다른 시그니처를 같는 것을 말한다. 동일한 처리를 하는 메소드를 서로 다른 이름으로 나타낼 경우 코드의 복잡도가 증가한다. 다음 예제에서 Add 라는 메소드를 입력 타입에 따라 메소드 명을 지은것과 메소드 오버로딩을 이용하였다. 가독성 및 메소드를 호출 하는 입장에서는 메소드 오버로딩을 하는 것이 더 효율적이다.

```
//입력형식에 따라 메소드명 생성

int AddInt (int x, int y);

float AddFloat (floatx, float y);

//동일한 메소드명으로 오버로딩

int Add (int x, int y);

float Add (float x, float y);
```

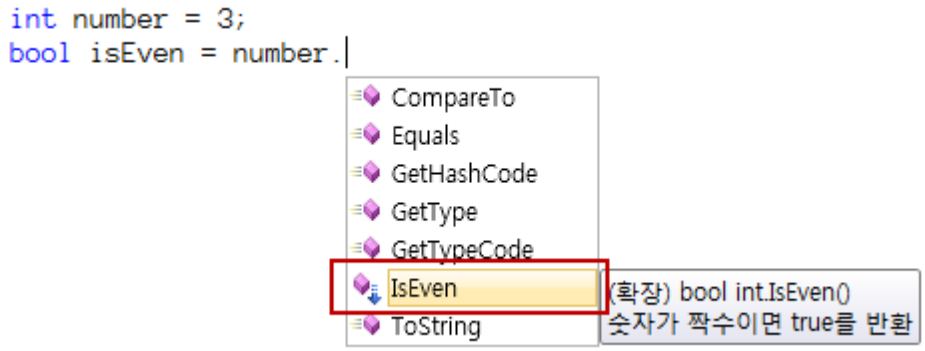
[C#강좌] 16.확장 메소드, Optional-Named 파라미터

확장 메소드(Extension Method)

프로그램을 작성하다 보면 이미 존재하는 클래스의 멤버에 유틸리티 성격의 메소드를 추가 하고 싶은 경우가 있다. 모든 소스를 개발자가 가지고 있으면 메소드를 추가 한 후 리빌드를 하면 되지만, FCL 뿐만 아니라, 3rd 파티의 많은 라이브러리를 사용하게 되므로 어쩔 수 없이 별도 클래스를 만들어 메소드를 선언하여 사용하였다. 하시만 C# 3.0 부터 등장한 확장 메소드를 사용하면 새로운 클래스를 선언하지 않고 기존 클래스에 새로운 메소드를 추가 할 수 있다. 만약 FCL 의 String 클래스에 나만의 유틸리티 메소드가 있다면 메소드를 확장 할 수 있는 것이다. 확장 메소드로 선언을 하려면 반드시 정적 메소드로 선언을 하여야 한다.
다음 예제는 int(Int32) 형식에 홀/짝수를 판단하는 확장 메소드 선언을 보여 주고 있다. 확장 메소드는 정적 메소드로만 선언이 가능하므로, 클래스도 정적 클래스로 선언을 하였다. 확장 메소드만 모아 놓은 정적 클래스를 선언하면 편할 것이다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch16_01
{
    //Extension Method 선언을 위한 static 클래스 정의
    public static class ExtensionMethod
    {
        public static bool IsEven(this int number)
        {
            //모듈러연산(%)이 아닌 비트연산 이용
            bool isEven = ((number & 0x1) == 0) ? true : false;

            return isEven;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            int number = 3;
            Console.WriteLine("{0} 짝수 ? {1}", number, number.IsEven());
        }
    }

}
```

확장 메소드로 선언을 하면 다름 그림과 같이 이미 존재하였던 메소드 처럼 해당 객체에서 바로 사용이 가능하다. 그리고 확장 메소드의 아이콘은 기존 메소드 아이콘과 다르게 아래도 향한 화살표시가 있어 쉽게 구분이 가능하다.



[인델리센스에서 보여지는 확장 메소드]

Optional / Named 파라미터(Parameter)

C#은 기본적으로 메소드에 선언된 파라미터 목록을 명시적으로 사용해야만 했다. 하지만 VSTO 와 같은 COM 과의 연동 프로그램을 작성할 경우, 사용하지 않는 파라미터 혹은 생략 가능한 목록까지 모두 작성해야만 하기 때문에 Visual Basic 과는 다르게 코드가 복잡하였다. 하지만 C# 4.0 부터는 메소드를 선언할 때 파라미터의 기본값을 저정하여 만약 사용하는 시점에 파라미터가 생략이 되면 선언시점에 설정한 기본값을 사용하게 하거나, 인자 순서를 명시적 이름로 나열할 수 있게 되었다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch16_02
{
    class Program
    {
        //Named 파라미터 선언 : y=5, z=7 의 기본값을 지정
        public static void M(int x, int y = 5, int z = 7)
        {
            Console.WriteLine("Output x={0}, y={1}, z={2}{3}", x, y, z, Environment.NewLine);
        }

        static void Main(string[] args)
        {
            Console.WriteLine("public static void M(int x, int y = 5, int z = 7)₩r₩n");
```

```
            Console.WriteLine("M(1, 2, 3);");
            M(1, 2, 3);      //Output x=1, y=2, z=3

            Console.WriteLine("M(1, 2);");
            M(1, 2);         //Output x=1, y=2, z=7 => 생략된 z 파리미터는 기본값 설정

            Console.WriteLine("M(1);");
            M(1);            //Output x=1, y=5, z=7

            Console.WriteLine("M(1, z:3);");
            M(1, z: 3);     //Output x=1, y=5, z=3 => y 는 기본값으로 z 는 명시적으로 선언

            Console.WriteLine("M(x:1, z:3);");
            M(x: 1, z: 3);  //Output x=1, y=5, z=3

            Console.WriteLine("M(z:3, x:1);");
            M(z: 3, x: 1);  //Output x=1, y=5, z=3
        }
    }
}
```

여기서 문제 하나를 내 보겠다. 다음과 같이 선언된 메소드가 존재할 경우 M(5)는 몇 번의 메소드가 호출이 될 것인가?

1. M(string x, int y = 1);

2. M(object x);

3. M(int x, string y = "Hello");

4. M(int x);

정답은 4 번 M(int x); 이다. 1 번 M(string x, int y=1) 은 숫자를 string 으로 변환 할 수 없다. 2, 3, 4 항목 중 2 번 보다는 3, 4 번 항목이 boxing 작업이 일어 나지 않아도 되기 때문에 더 적당 하다. 3, 4 번 중 파라미터 생략이 이루어진 형태 보다 정확한 시그니처가 맞는 4 번이 적당한 것이다.

이처럼 Optional / Named 파라미터 선언을 잘 못할 경우 사용하는 입장에서는 호출하는 메소드가 무엇인지 모호해지는 경우가 발행하게 된다.

[C#강좌] 17.델리게이트, 이벤트

델리게이트(Delegate)

메소드가 처리해야 할 것을 델리게이트에 위임을 한다. 그리고 델리게이트만 호출을 하게 되면, 델리게이트가 위임 받은 메소드 들이 실행이 되는 형태이다. 델리게이트는 이벤트와 함께 많이 사용이 된다. 델리게이트는 위임을 받을 메소드의 파라미터, 리턴값의 시그니처만을 정의한다. 그러므로 공용모듈을 설계 할 때, 특정 메소드가 아닌 비슷한 기능을 하는 다양한 메소드 실행이 가능하므로 상당히 유용하게 사용가능 하다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch17_01
{
    class Program
    {
        //델리게이트 선언
        delegate int Calculate(int number1, int number2);

        private static  int Add(int number1, int number2)
        {
            return number1 + number2;
        }

        private static int Subtract(int number1, int number2)
        {
            return number1 - number2;
        }

        static void Main(string[] args)
        {
            Calculate add = new Calculate(Add);
            Calculate sub = new Calculate(Subtract);

            Console.WriteLine("Calculate(Add) 1+2 : {0}", add(1, 2));
            Console.WriteLine("Calculate(Calculate) 1-2 : {0}", sub(1, 2));
        }
    }
}
```

무명 메소드(Anonymous Method)

무명 메소드 사용은 이전 장에서 잠깐 살펴 보았을 것이다. 여러 단란에서 말하고 있지만, C# 초기에는 무엇인가를 사용하려면 명시적 선언이 필요하였다. 델리게이트도 마찬가지로 명시적으로 선언하고 사용해야만 했지만, 예를 들어 버튼 클릭 이벤트에 상황에 따라 동적으로 동적으로 델리게이트 구문이 등록하여야 할 때 일일이 델리게이트 선언은 부담이 된다. 하지만 익명 메소드를 사용하면 명시적 선언 없이 필요한 시점에 즉시 구문 작성을 가능하게 해 준다. 위의 예제에서 선언한 델리게이트를 이용하여 작성하였다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch17_02
{
    class Program
    {
        //델리게이트 선언
        delegate int Calculate(int number1, int number2);

        static void Main(string[] args)
        {
            //무명 메소드(Anonymous Method) 이용
            Calculate multiplyAnonymousMethod
                = new Calculate(delegate(int number1, int number2) { return number1 * number2; });

            Console.WriteLine("Calculate(multiplyAnonymousMethod) 2*3 : {0}"
             , multiplyAnonymousMethod(2, 3));

            //무명 메소드(Anonymous Method)로 선언한 부분을 람다식으로 변경
            Calculate divideLamda = new Calculate((number1, number2) => number1 / number2);
            Console.WriteLine("Calculate(divideLamda) 4/2 : {0}", divideLamda(4, 2));
        }
    }
}
```

이벤트(Event)

UI(User Interface)를 가지고 있은 프로그램은 각종 컨트롤에서 발생하는 이벤트에 따라 다양한 처리를 하게 된다. 그리고 많은 BCL 에서도 현재 객체의 상태 변화를 감지할 수 있는 다양한 이벤트를 제공하여 각 상황에 맞게 처리할 수 있도록 제공해 주고 있다. 이벤트는 어떠한 처리가 즉시 일어 나는 것이 아니라 사용자의 행동이나 다른 모듈에서의 처리 완료 시점을 알 수 없을 때 유용한 패턴을 제공하여 준다.

다음 예제는 FileSystemWatcher 를 통해 특정 폴더를 감시하면 파일의 변화를 감지하는 예이다. 감시하는 폴더는 Environment.CurrentDirectory 를 통해 프로그램이 실행의 실행파일이 존재하는 폴더를 지정하였으며, FileSystemWatcher 의 Changed, Created, Deleted, Renamed 이벤트를 등록하여 *.txt 파일의 변화가 있을 때 마다 콘솔창에 알림을 보낸다. 아래 예제에서 한가지 주의 할 점은 콘솔 프로그램은 기본적으로 실행 후 바로 종료가 되므로 명시적으로 종료하기 전까지 대기하도록 while (Console.Read() != 'q') ; 를 작성해 두어야 한다. 코드 작성이 완료 되었으면, 프로그램 실행 폴더로 이동하여 텍스트파일을 생성, 이름 변경등 다양한 작업을 해보기 바란다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;        //FileSystemWatcher 사용

namespace SqlerCSharp.Ch17_03
{
    public class EventTest
    {
        //선언 - 특정 폴더의 파일 변화 감시
        FileSystemWatcher watcher = new FileSystemWatcher();

        public void Run()
        {
            //실행되는 현재 폴더 설정
            watcher.Path = Environment.CurrentDirectory;

            Console.WriteLine(Environment.CurrentDirectory);

            //변경내역을 조사할 형식 설정
            watcher.NotifyFilter = NotifyFilters.LastAccess | NotifyFilters.LastWrite
            | NotifyFilters.FileName | NotifyFilters.DirectoryName;

            //감시를 할 파일 확장자 설정
            watcher.Filter = "*.txt";

            //이벤트 등록
```

```
            watcher.Changed += new FileSystemEventHandler(watcher_Changed);
            watcher.Created += new FileSystemEventHandler(watcher_Changed);
            watcher.Deleted += new FileSystemEventHandler(watcher_Changed);
            watcher.Renamed += new RenamedEventHandler(watcher_Renamed);

            //감시 시작
            watcher.EnableRaisingEvents = true;
        }

        void watcher_Renamed(object sender, RenamedEventArgs e)
        {
            Console.WriteLine("변경전 : {0}{2}변경후 : {1}"
            , e.OldName, e.Name, Environment.NewLine);
        }

        void watcher_Changed(object sender, FileSystemEventArgs e)
        {
            Console.WriteLine(@"변경유형 : {0}{3}영향받은 파일 : {1}₩{2}"
            , e.ChangeType, e.FullPath, e.Name, Environment.NewLine);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            EventTest eventTest = new EventTest();
            eventTest.Run();
            //콘솔 프로그램이 종료되지 않도록 설정
            Console.WriteLine("Press ₩'q₩' to quit the sample.");
            while (Console.Read() != 'q') ;
        }
    }

}
```

[C#강좌] 18.Func, Action

LINQ(Language-Integrated Query)에서 Where 확장 메서드(Extension Method)의 경우처럼, 여러 조건으로 필터링하는 메소드를 만든다고 가정해 보자. 검색 조건은 상황에 따라 아주 다양할 것이다. 그럼 모든 상황별 메서드를 모두 만들 것인가? 사실상 불가능 하다. 그래서 Where 확장메서드는 필터링을 처리하는 델리게이트를 파라미터 받을 수 있게 되어있다.

필요한 그때 그때 필터링하는 기능을 만들어서 델리게이트로 넘기면 된다. 기능 구현은 닷넷 프레임워크 2.0 의 익명메서드(Anonymous Method)나 닷넷 프레임워크 3.5 의 람다식(Lambda Expressions)를 통해 명시적 메서드 구현 없이도 쉽게 사용 가능하다. 여기서 또 한가지 의문을 가질 수 있다. 실제적은 기능 구현은 익명메서드나 람다식으로 하면 되지만 정작 Where 같은 메서드를 정의할 때 쓰일 델리게이트는 어떻게 할 것인가? 다양한 파리미터 조건을 가지고 싶다면? 그에 맞게 델리게이트를 명시적으로 모두 정의할 것인가? 이 또한 쉽지 않다. 그래서 닷넷 프레임워크에서는 이런 고민들 쉽게 해결할 특수(?) 델리게이트를 지원해 주고 있다.

바로 "Func"와 "Action" 델리게이트이다. 이들 델리게이트는 [표 1]과 [표 2]에서 보는 것처럼 제네릭(generic) 타입으로 정의되어 있으며, 몇 가지 유형으로 오버라이드(override) 되어 있어 다양한 요구 조건을 모두 만족 할 수 있다. "Func"와 "Action" 의 차이점은 위임된 기능이 처리되고 반환되는 결과값이 있느냐, 없느냐의 차이만 있다. 닷넷 프레임워크 4.0 으로 오면서 "Func"와 "Actin" 델리게이트에 설정할 수 있는 파라미터 목록도 크게 늘어도 최대 16 개까지 지원하며, 이전 보다 더 유연한 프로그램 개발이 가능해 졌다.

- Func : 내부 처리 완료 후 반환되는 결과값이 있음. 특정 조건으로 필터링하고 그 결과를 반환하는 Where 같은 메소드를 구현할 때 쓰면 유용할 것이다.
- Action : 내부 처리 완료 후 반환되는 결과값이 없음. 어떤 처리 진행을 보여 주는 UI 업데이트 같은 처리를 할 때 유용할 것이다.

delegate void Action <T> (T arg) 는 이미 닷넷프레임워크 2.0 부터 존재 하였으며 다른 델리게이트들과 다르게 mscorlib(mscorlib.dll) 어셈블리에 정의되어 있다(네임스페이스는 동일하게 System 에 존재한다).

[정의]

- 네임스페이스: System
- 어셈블리: System.Core (System.Core.dll)

delegate void Action ( );

delegate void Action <T> (T arg);

delegate void Action <T1, T2> (T1 arg1, T2 arg2);

delegate void Action <T1, T2, T3> (T1 arg1, T2 arg2, T3 arg3);

delegate void Action <T1 ,T2, T3, T4> (T1 arg1, T2 arg2, T3 arg3, T4 arg4);

...

delegate void Action<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16>(T1 arg1, ..., T16 arg16)

[Action 델리게이트]

delegate TResult Func <TResult> ( );

delegate TResult Func <T, TResult> (T arg);

delegate TResult Func <T1, T2, TResult> (T1 arg1, T2 arg2);

delegate TResult Func <T1, T2, T3, TResult> (T1 arg1, T2 arg2, T3 arg3);

delegate TResult Func <T1, T2, T3, T4, TResult> (T1 arg1, T2 arg2, T3 arg3,T4 arg4);

...

delegate TResult Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16, TResult>(T1, arg1, ..., T16 arg16)

[Func 델리게이트]

예를 하나 살펴 보자. 숫자로 구성된 배열이 존재하고, 다양한 조건으로 배열에서 값을 가져오는 메서드를 정의하고 싶다고 가정하자. 여기서는 "다양한 조건으로 검색이 가능하도록"이 중요한 포인트이다. 검색을 다양하게 하려면 호출되는 시점에 필터링 조건을 주면 된다. 해당 값이 조건이 맞는지 확인해서 bool 형식을 반환하는 코드가 필요하므로 "Func"를 사용하여 다음과 같이 메소드를 정의 하였다.

```
private static List<int> FilterOfInts(int[] source, Func<int, bool> filter)
{
    List<int> result = new List<int>();
    foreach (int i in source) { if (filter(i)) { result.Add(i); } }
    return result;

}
```

[숫자 배열에서 여러 조건으로 필터링 할 수 있는 메서드 정의]

이제 정의된 메서드를 통해, 명시적으로 정의된 메서드 혹은 익명메서드, 람다식 모두를 사용하여 원하는 결과를 볼 수 있다. [코드 2]를 보면 FilterOfInts 를 모두 호출하지만 람다식으로 다양한 조건을 파라미터로 넘기고 있다. 각 조건에 맞는 메소드를 모두 구현하는 것이 아니라 단일 메소드를 호출하므로 보다 유연한 모듈 개발이 가능한 것을 확인 할 수 있다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections;

namespace SqlerCSharp.Ch18_01
{
    class Program
    {
        private static List<int> FilterOfInts(int[] source, Func<int, bool> filter)
        {
            List<int> result = new List<int>();
            foreach (int i in source)
            {
                if (filter(i))
                {
                    result.Add(i);
                }
            }

            return result;
        }
```

```
        static void Main(string[] args)
        {
            int[] source = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

            //===홀수 Filter===
            List<int> oddNumbers = FilterOfInts(source, i => ((i & 1) == 1));

            Console.WriteLine("=== 홀수 Filter ===");
            foreach (var item in oddNumbers)
            {
                Console.WriteLine(item);
            }

            //===짝수 Filter===
            List<int> evenNumbers = FilterOfInts(source, i => ((i & 1) == 0));

            Console.WriteLine("=== 짝수 Filter ===");
            foreach (var item in evenNumbers)
            {
                Console.WriteLine(item);
            }

            //===소수 Filter===
            List<int> primeNumbers = FilterOfInts(source,
            checkNumber =>
            {
                BitArray numbers = new BitArray(checkNumber + 1, true);
                for (int i = 2; i < checkNumber + 1; i++)
                {
                    if (numbers[i])
                    {
                        for (int j = i * 2; j < checkNumber + 1; j += i) { numbers[j] = false; }
                        if (numbers[i]) { if (checkNumber == i) { return true; } }
                    }
                }
                return false;
            });

            Console.WriteLine("=== 소수 Filter ===");
            foreach (var item in primeNumbers)
            {
                Console.WriteLine(item);
            }
        }
    }
}
```

[FilterOfInts 를 통한 다양한 필터 조건을 사용]

[C#강좌] 19.익명 형식

익명 형식(Anonymous Type)은 메소드 내에서 즉시 데이터 구조를 정의하여 사용할 수 있도록 도와 준다. 우리는 이전에 데이터의 구조를 정의 하려면 클래스 혹은 구조체(Struct)를 미리 정의해서 사용해야만 했다. 하지만 익명 형식을 사용하면 사용하고자 하는 시점에 바로 변수와 같이 정의해서 사용 가능하다. 익명이란 의미에서도 알 수 있듯이 익명 형식은 클래스 이름과 같은 것을 명시적으로 지정하지 않는다. 그리고 익명 형식은 var 형식으로 선인이 되어야 한다. 왜냐 하면 명시적 형식이 아니기 때문에 특정 형식으로는 선언해서 사용을 할 수 없다. 익명 형식에서 정의하는 필드의 자료형은 명시적으로 선언을 할 필요가 없으며, 설정하는 값에 따라 알아서 형식을 추정하게 된다. 개발하는 입장에서는 정말 간편하게 사용이 가능하다.

아래 예제에서 BirthYear 는 설정 값이 숫자 이므로 int 로, Name 은 문자열이 설정되었으므로, string 으로 형식을 추정한다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch19_01
{
    class Program
    {
        static void Main(string[] args)
        {
            //익명 형식 선언
            var emp = new { BirthYear = 1978, Name = "김도현" };

            Console.WriteLine("=== 익명 형식 선언 ===");
            Console.WriteLine("BirthYear : {0}, Name : {1}", emp.BirthYear, emp.Name);

            //익명 형식으로 된 배열 선언
            var emps = new[]
            {
                new { BirthYear = 1978, Name = "김도현" },
                new { BirthYear = 1983, Name = "서정인" }
            };

            Console.WriteLine("=== 익명 형식으로 된 배열 선언 ===");
            foreach (var item in emps)
            {
                Console.WriteLine("BirthYear : {0}, Name : {1}", item.BirthYear, item.Name);
            }
        }
    }
}
```

[C#강좌] 20.컬랙션

컬랙션은 배열과 비슷하게 각 요소(element)를 인덱스를 통하여 접근이 가능하며, foreach 문을 이용하여 쉽게 처리 가능하다. 제네릭 형식의 컬랙션의 경우 IEnumerable<T> 와 IEnumerator<T> 인테페이스(Interface)를 상속받고 있으며, 사용자 정의 컬랙션도 구현 가능하다. 과거 닷넷 프레임워크 1.x 버전에서는 object 형식의 컬랙션만이 존재 하였으나, 닷넷 프레임워크 2.0 부터 다양한 제네릭 형식의 컬랙션을 제공하고 있다.

Non-Generic(닷넷 프레임워크 1.x)	Generic(닷넷 프레임워크 2.0+)
CollectionBase	Collection<T>
ReadOnlyCollectionBase	ReadOnlyCollection<T>
ArrayList	List<T>
Hashtable	Dictionary<TKey, TValue> ConcurrentDictionary<TKey, TValue>
Queue	Queue<T> ConcurrentQueue<T>
Stack	Stack<T> ConcurrentStack<T>
SortedList	SortedList<TKey, TValue>

[Non-제네릭에 대응되는 제네릭 클래스 비교]

List<T>

가장 많이 사용하는 컬렉션 중 하나로 가장 배열과 비슷한 구조를 가진다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch20_01
{
    class Program
    {
        static void Main(string[] args)
        {
            List<string> words = new List<string>();    //string 형식의 List 생성
            words.Add("A");
            words.Add("B");

            Console.WriteLine(@"words.Add(""A""); words.Add(""B"");");
            foreach (var item in words)
            {
                Console.WriteLine("words : {0}", item);
            }

            words.AddRange(new[] { "C", "D" });        //배열을 현재 리스트에 추가

            Console.WriteLine(@"words.AddRange(new[] { ""C"", ""D"" }); ");
            foreach (var item in words)
            {
                Console.WriteLine("words : {0}", item);
            }

            words.Insert(0, "E");                    //0 번째에 추가

            Console.WriteLine(@"words.Insert(0, ""E""); ");
            foreach (var item in words)
            {
                Console.WriteLine("words : {0}", item);
            }

            words.InsertRange(0, new[] { "F", "G" });   //0 번째에 배열 값 추가

            Console.WriteLine(@"words.InsertRange(0, new[] { ""F"", ""G"" });");
            foreach (var item in words)
            {
                Console.WriteLine("words : {0}", item);
```

```
        }

        words.Remove("E");                          //값을 이용한 제거

        Console.WriteLine(@"words.Remove(""E""); ");
        foreach (var item in words)
        {
            Console.WriteLine("words : {0}", item);
        }

        words.RemoveAt(3);                          //인덱스를 이용한 제거

        Console.WriteLine("words.RemoveAt(3);");
        foreach (var item in words)
        {
            Console.WriteLine("words : {0}", item);
        }

        words.RemoveRange(0, 2);                    //범위를 이용한 제거

        Console.WriteLine("words.RemoveRange(0, 2);");
        foreach (var item in words)
        {
            Console.WriteLine("words : {0}", item);
        }
    }
  }

}
```

Dictionary<TKey, TValue>

이름에서도 알 수 있듯이 사전과 같이 키와 값으로 이루어진 컬랙션이다. LIST<T>는 특정 항목에 접근을 하려면 인덱스로 찾아 가지만 Dictionary<TKey, TValue>는 입력된 특정 키를 통해 접근이 가능하다. 한가지 유의할 점은 키는 대소문자를 구분을 한다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch20_02
{
    class Program
    {
        static void Main(string[] args)
        {
            Dictionary<string, int> month = new Dictionary<string, int>();
            month.Add("Jan", 1);
            month["Feb"] = 2;
            month["Feb"] = 222;                         //Feb 키가 존재하므로 값을 업데이트 한다.
            month["Mar"] = 3;
            month["Apr"] = 4;
            month["May"] = 5;
            month["June"] = 6;
            month["July"] = 7;
            month["Aug"] = 8;
            month["Sept"] = 9;
            month["Oct"] = 10;
            month["Nov"] = 11;
            month["Dec"] = 12;

            Console.WriteLine(month["Feb"]);                //222
            Console.WriteLine(month.ContainsKey("Jan"));    //true
            Console.WriteLine(month.ContainsValue(1));      //true

            int val = 0;
            if (!month.TryGetValue("jAN", out val))         //키는 대소문자를 구분한다.
                Console.WriteLine("No value");
        }
    }

}
```

Collection Initializer

마찬가지로 처음 컬랙션을 생성하는 시점에 초기화를 하려면 이전에는 인스턴스 생성 후 각 요소에 값을 입력하는 형식이었다. 하지만 다음 코드와 같이 C# 3.0 부터 배열처럼 처음 인스턴스 생성하는 시점에 컬랙션을 초기화 할 수 있다. 특히 Object Initializer 와 함께 사용하면 간결한 코드 작성이 가능하다. 다음 코드는 C# 2.0 에서 "new 식은 형식 뒤에 () 또는 []가 필요합니다." 라고 컴파일 오류를 발생한다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SqlerCSharp.Ch20_03
{
    class Employee
    {
        public int BirthYear {get; set;}
        public string Name { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            //컬랙션의 선언시 값 초기화
            List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

            //Object Initializer 와 함께 사용 가능
            List<Employee> emp = new List<Employee>
            {
                new Employee(){ BirthYear = 1978, Name = "김도현"},
                new Employee(){ BirthYear = 1983, Name = "이정연"},
                new Employee(){ BirthYear = 1970, Name = "유인혜"}
            };

            foreach (var item in emp)
            {
                Console.WriteLine("BirthYear : {0}, Name : {1}", item.BirthYear, item.Name);
            }
        }
    }
}
```