[목차]

1. 확장 메소드(Extension Method)

프로그램을 작성하다 보면 이미 존재하는 클래스의 멤버에 유틸리티 성격의 메소드를 추가 하고 싶은 경우가 있다. C# 3.0 부터 등장한 확장 메소드를 사용하면 새로운 클래스를 선언하지 않고 기존 클래스에 새로운 메소드를 추가 할 수 있다. 만약 String 클래스에 나만의 유틸리티 메소드가 있다면 메소드를 확장 할 수 있는 것이다. 확장 메소드로 선언을 하려면 반드시 정적 메소드로 선언을 하여야 한다.

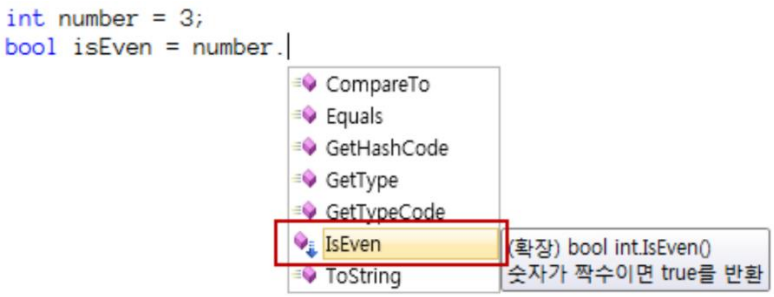
다음 예제는 int(Int32) 형식에 홀/짝수를 판단하는 확장 메소드 선언을 보여 주고 있다. 확장 메소드는 정적 메소드로만 선언이 가능하므로, 클래스도 정적 클래스로 선언을 하였다. 확장 메소드만 모아 놓은 정적 클래스를 선언하면 편할 것이다.

```
using System;

namespace Sample
{
    //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]

```
using System;

namespace Sample
{
    //Extension Method 선언을 위한 static 클래스 정의
    public static class ExtensionMethod
    {
        public static void Func(this string str, string value)
        {
            Console.WriteLine(value);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            string test = "";
            test.Func( "Hello,World!" );
        }
    }
}
```

확장 메서드 예

```
using System;
static class GlobalClass{
    // 확장 메서드는 반드시 static이어야 하고,
    // 확장하려는 타입의 매개변수를 this 예약어와 함께 명시
    public static int GetWordCount(this string txt)
    {
        return txt.Split(' ').Length;
    }
}

class Program{
    static void Main(string[] args)
    {
        string text = "Hello, World!";

        // 마치 string 타입의 인스턴스 메서드를 호출하듯이 확장 메서드를 사용
        Console.WriteLine("Count: " + text.GetWordCount()); // 출력 결과: Count: 2

        Console.WriteLine("Count: " + GlobalClass.GetWordCount(text));
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq; // IEnumerable의 확장 메서드를 호출하기 위해 네임스페이스 추가

class Program{
    static void Main(string[] args)
    {
        List<int> list = new List<int>() { 5, 4, 3, 2, 1 };

        // IEnumerable의 Min 확장 메서드 호출
        Console.WriteLine(list.Min()); // 출력 결과: 1
    }
}
```

2. Optional / Named 파라미터(Parameter)

C#은 기본적으로 메소드에 선언된 파라미터 목록을 명시적으로 사용해야만 했다. 하지만 C# 4.0 부터는 메소드를 선언할 때 파라미터의 기본값을 저정하여 만약 사용하는 시점에 파라미터가 생략이 되면 선언 시점에 설정한 기본값을 사용하게 하거나, 인자 순서를 명시적 이름로 나열할 수 있게 되었다.

```
using System;

namespace Sample
{
    class Program
    {
        //Named 파라미터 선언 : y=5, z=7의 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)
        {
            M(1, 2, 3);       //Output x=1, y=2, z=3
            M(1, 2);          //Output x=1, y=2, z=7 => 생략된 z 파리미터는 기본값 설정
            M(1);             //Output x=1, y=5, z=7


            M(1, z: 3);     //Output x=1, y=5, z=3 => y는 기본값으로 z는 명시적으로 선언
            M(x: 1, z: 3);  //Output x=1, y=5, z=3
            M(z: 3, x: 1);  //Output x=1, y=5, z=3
        }
    }
}
```

3. 가변 매개변수 : params

메서드를 정의할 때 몇 개의 인자를 받아야 할지 정할 수 없을 때가 있다. 이런 상황에서 params 예약어를 사용해 가변 인자를 지정할 수 있다.

```
using System;
namespace Sample{
    class Program    {
        static int Add( params int[] values )        {
            int result = 0;
            for (int i = 0; i < values.Length; i++)            {
                result += values[i];
            }
            return result;
        }

        static void Main(string[] args)        {
            Console.WriteLine(Add(1, 2, 3, 4, 5)); // 출력값: 15
            Console.WriteLine(Add(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)); // 출력값: 55
        }
    }
}
```

- 입력 타입을 지정할 수 없다면 모든 타입의 부모인 object를 사용한다.

```
using System;
namespace Sample{
    class Program    {
        private static void PrintAll(params object[] values)        {
            foreach (object value in values)            {
                Console.WriteLine(value);
            }
        }
        static void Main(string[] args)        {
            PrintAll(1.05, "Result", 3);
        }
    }
}
```

4. check / uncheck

정수 계열 타입의 산술 연산을 하거나, 서로 다른 정수 타입 간 형변환을 하게 되면 표현 가능한 숫자의 범위를 넘어서는 경우가 발생한다.

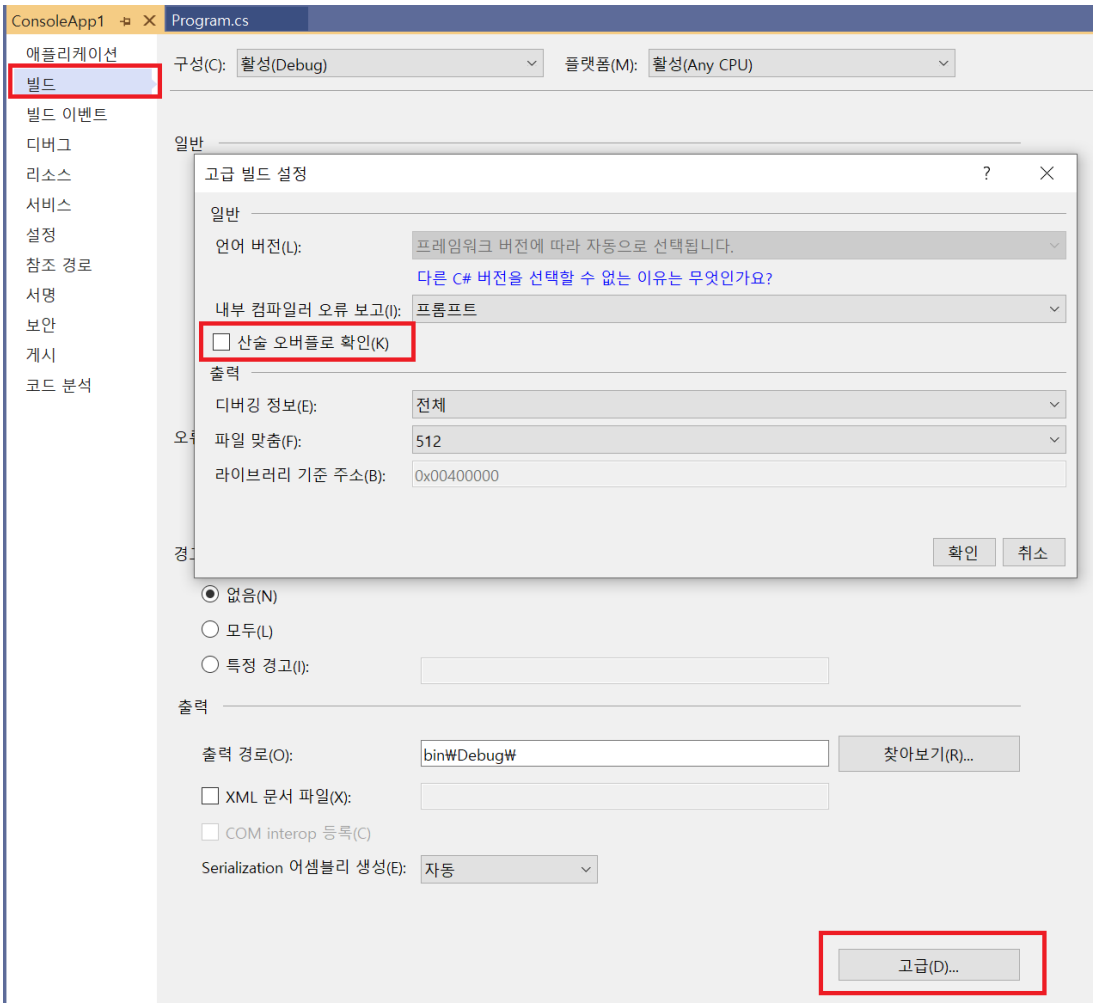
- 오버플로(overflow) 데이터의 상환값을 넘어 하한값으로 돌아가는 것
- 언더플로(underflow) 데이터의 하한값을 지나 상한값으로 돌아가는 것

연산식에서 오버프로로나 언더플로가 발생한 경우 오류를 발생하도록 명시할 수 있다. (check)

특정 산술 연산에 대해 오버플로나 언더플로가 발생해도 오류를 내지 말라고 명시할 수 있다.(uncheck)

```
using System;
namespace Sample {
    class Program    {
        static void exam1()        {
            short c = 32767;
            c++;
            Console.WriteLine(c);
        }
        static void exam2()        {
            short c = 32767;
            checked
            {
                c++; //System.OverflowException: 산술 연산으로 인해 오버플로가 발생했습니다.
            }
        }
        static void exam3()        {
            short c = 32767;
            unchecked
            {
                c++; // 오류가 발생하지 않는다.
            }
        }
        static void Main(string[] args)        {
            exam3();
        }
    }
}
```

- C# 컴파일러 수준에서 checked 상황을 전체 소스코드에 걸쳐 강제로 적용할 수 있다.



[Visual Studio 2019, 프로젝트 속성 >> 빌드 : 고급]

5. var 예약어

C# 3.0 컴파일러부터는 타입 추론(type inference) 기능이 추가되면서 메서드의 지역변수 선언을 타입에 관계없이 var 예약어로 쓸 수 있다.

```
using System;
class Program{
    static void Main(string[] args)    {
        int i = 5;
        var j = 6;
        Console.WriteLine(i.GetType().FullName); // 출력 결과: System.Int32
        Console.WriteLine(j.GetType().FullName); // 출력 결과: System.Int32
    }
}
```

* var 키워드가 있는 자리는 결국 C# 컴파일러에 의해 실제 타입으로 치환된다.

var예약어는 코드의 가독성을 낮게 하지만, 복잡한 타입의 경우 오히려 var 예약어를 사용하면 코드가 간결해 지는 장점이 있다.

```
using System;
using System.Collections.Generic;
using System.Text;
class Program{
    static void Main(string[] args)    {
        List<int> numbers1 = new List<int>(new int[] { 1, 2, 3, 4, 5 });
        List<int> numbers2 = new List<int>(new int[] { 6, 7, 8, 9, 10 });

        Dictionary<string, List<int>> dict = new Dictionary<string, List<int>>();
        dict["first"] = numbers1;
        dict["second"] = numbers2;

        foreach (KeyValuePair<string, List<int>> elem in dict)          {
            Console.WriteLine(elem.Key + ": " + Output(elem.Value));
        }

        foreach (var elem in dict)  {// C# 컴파일러는 var를 KeyValuePair<string, List<int>>로 대체
                Console.WriteLine(elem.Key + ": " + Output(elem.Value));
        }
    }
```

```
    private static string Output(List<int> list)      {
        StringBuilder sb = new StringBuilder();
        foreach (int elem in list)            {
            sb.AppendFormat("{0},", elem);
        }
        return sb.ToString().TrimEnd(',');
    }
}
```

6. dynamic 예약어

var 예약어와 비슷한 기능을 갖는다.

Var 예약어는 C# 컴파일러가 빌드 시점에 초기값과 대응되는 타입으로 치환한다.

Dynamic 은 해당 프로그램이 실행되는 시점에 타입을 결정한다.

- 동적 언어인 Ruby, Python도 닷넷 프레임웍에서 사용될 수 있도록 IronPython, IronRuby 으로 포팅되었다.
- 동적 언어로 만들어진 프로그램의 타입 시스템을 C#과 같은 정적 언어에서 연동하는 방법을 제공하기 위해 dynamic 이 제공된다.

```
using System;
class Program
{
    static void Main(string[] args)     {
        dynamic d = 5;
        int sum = d + 10;
        Console.WriteLine(sum);

        DynamicSample();
    }
    static void DynamicSample()
    {
        /*
        var d = 5;
        d = "test"; // 컴파일 오류: d == System.Int32로 결정되기 때문에 문자열을 받을 수 없음

        d.CallFunc(); // 컴파일 오류: System.Int32 타입에는 CallFunc 메서드가 없음
        */
        dynamic d2 = 5;

        d2 = "test"; // d2는 형식이 결정되지 않았기 때문에 다시 문자열로 초기화 가능
        d2.CallFunc(); // 실행 시에 d2 변수의 타입으로 CallFunc을 호출하기 때문에
                       // 컴파일 오류가 발생하지 않음. 하지만 실행 시에는 오류 발생

    }
}
```

7. nullable 형식

데이터베이스의 NULL과 유사한 개념(값이 존재하지 않는다)

int? number = null;

```
using System;
class Program{
    static void Main(string[] args)
    {
        Nullable<int> intValue = 10;
        int target = intValue.Value;
        intValue = target;
        Console.WriteLine(intValue);      //intValue.Value;

        double? temp = null;
        Console.WriteLine(temp);                // 공백출력
        Console.WriteLine(temp.HasValue);    // false 출력

        temp = 3.141592;
        Console.WriteLine(temp);                // 3.141592
        Console.WriteLine(temp.HasValue);    // true 출력
    }
}
```

회원 정보가 아래와 같이 정의되었다고 가정하자.

만약, 가입한 사람이 결혼 여부를 입력하지 않았다면?

```
public class SiteMember
{
    bool _getMarried;
    public bool GetMarried
    {
        get { return _getMarried; }
        set { _getMarried = value; }
    }
}
```

아래와 같이 수정 가능하다.

Nullable<T> 타입은 일반적인 값 형식에 대해 null 표현이 가능하게 하는 역할을 한다.

축약형으로 값 형식에 ? 문자를 붙이는 표현도 지원한다.(컴파일러는 Nullable<T>형식으로 변환해 줌)

```
Nullable<bool> _getMarried;
public Nullable<bool> GetMarried
{
    get { return _getMarried; }
    set { _getMarried = value; }
}
```

```
bool? _getMarried;
public bool? GetMarried
{
    get { return _getMarried; }
    set { _getMarried = value; }
}
```

```
using System;

public class SiteMember
{
    bool? _getMarried = null;
    public bool? GetMarried     {
        get { return _getMarried; }
        set { _getMarried = value; }
    }
}

class Program
{
    static void Main(string[] args)
    {
        SiteMember sm = new SiteMember();
        Console.WriteLine("결혼여부 : " + ((sm.GetMarried ==null) ? "미입력": "입력"));
    }
}
```

8. ?? 연산자

?? 연산자는 null값을 가진 참조형 변수를 손쉽게 처리할 수 있는 연산자이다.

피연산자1 ?? 피연산자2 >> 참조 형식의 피연산자1이 null이 아니라면 그 값을 그대로 반환하고, null이라면 피연산자2의 값을 반환한다.

```
using System;
namespace Sample{
    class Program    {
        static void Main(string[] args)        {
            string txt = null;

            //기존 방식
            if (txt == null)
                Console.WriteLine("(null)");
            else
                Console.WriteLine(txt);
            //?? 방식
            Console.WriteLine(txt ?? "(null)");
        }
    }
}
```

```
using System;
namespace Sample{
    class Program    {
        static void Main(string[] args)        {
            int? a = null;
            int b = 3;
            var output = a ?? b;
            var type = output.GetType();
            Console.WriteLine($"Output Type :{type}");
            Console.WriteLine($"Output value :{output}");
        }
    }
}
```

```
using System;
namespace Sample{
    class Program    {
        static void Main(string[] args)        {
            object o = null;
            var output = o?.ToString() ?? "Default Value";
            Console.WriteLine(output); //Default Value
        }
    }
}
```

```
using System;
namespace Sample{
    class Program    {
        static string GetName() { return "홍길동"; }

        static void Main(string[] args)        {
            string name = GetName();
            if (name == null)
                name = "Unknown!";

            string name1 = GetName() ?? "Unknown!";
        }
    }
}
```

```
using System;
namespace Sample{
    public class Person    {
        public int Age { get; set; }
        public string Name { get; set; }
    }

    class Program    {
        static Person GetPerson1() { return null; }
        static Person GetPerson2() { return new Person() { Age = 10, Name = "ccm" }; }

        static void Main(string[] args)        {
            Person person = GetPerson1();
            var age = person?.Age; // 'age' will be of type 'int?', even if 'person' is not null
            Console.WriteLine(age);
        }
    }
}
```

9. yield return / break

기존 IEnumerable, IEnumerator 인터페이스를 이용해 구현하였던 열거 기능을 쉽게 구현할 수 있다.

```
using System;
using System.Collections.Generic;

namespace Sample
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] intList = new int[] { 1, 2, 3, 4, 5 };
            List<string> strings = new List<string>();

            strings.Add("Anders");
            strings.Add("Hejlsberg");

            foreach (int n in intList)
            {
                Console.Write(n + ",");
            }

            Console.WriteLine();

            foreach (string txt in strings)s
            {
                Console.Write(txt + " ");
            }
        }
    }
}
```

10.IEnumerable, IEnumerator 를 구현하여 foreach문에서 사용가능토록 코드 구현

```
using System;
using System.Collections;
using System.Collections.Generic;

public class NaturalNumber : IEnumerable<int>
{
    //IEnumerable<int> 재정의
    public IEnumerator<int> GetEnumerator()
    {
        return new NaturalNumberEnumerator();
    }

    //IEnumerable 재정의
    IEnumerator IEnumerable.GetEnumerator()
    {
        return new NaturalNumberEnumerator();
    }

    public class NaturalNumberEnumerator : IEnumerator<int>
    {
        int _current;

        public int Current
        {
            get { return _current; }
        }

        object System.Collections.IEnumerator.Current
        {
            get { return _current; }
        }

        public void Dispose() { }

        public bool MoveNext()
        {
```

```
                _current++;
                return true;
            }

            public void Reset()
            {
                _current = 0;
            }
        }
}

class Program
    {
        static void Main(string[] args)
        {
            NaturalNumber number = new NaturalNumber();
            foreach (int n in number) // 출력 결과: 1부터 자연수를 무한하게 출력
            {
                Console.Write(n + " ");
            }
    }
}
```

yield return 을 이용한 자연수 표현

```
using System;
using System.Collections.Generic;
class YieldNaturalNumber{
	public static IEnumerable<int> Next(int max)	{
		int _start = 0;
		while (true)
		{
			_start++;
			//if (max < _start) {
		//		yield break; // max만큼만 루프를 수행한 후 열거를 중지한다.
			//}
			yield return _start;
		}
	}
}

class Program{
	static void Main(string[] args)	{
		foreach (int n in YieldNaturalNumber.Next(100))		{
			Console.Write(n + " ");
		}
	}
}
```

* Next 메서드가 호출되면 yield return 에서 값이 반환되면서 메서드 실행을 중지

* 하지만 내부적으로는 yield return 이 실행된 코드의 위치를 기억해 두었다가 다음에 다시 한번 메서드가 호출되면 처음부터 코드가 실행되지 않고 마지막 yield return 문이 호출되었던 코드의 다음 줄부터 실행을 재개한다.(무한 반복됨)

* 주석을 제거하면 100까지 출력 된다.

11.익명 메서드

익명 메서드란 이름이 없는 메서드로서 델리게이트에 전달되는 메서드가 일회성으로만 필요할 때 편의상 사용된다.

```
using System;
using System.Threading;
class Program{
    static void Main(string[] args)    {
        Thread thread = new Thread(ThreadFunc);
        thread.Start();
        thread.Join();
    }

    private static void ThreadFunc(object obj)
    {
        Console.WriteLine("ThreadFunc called!");
    }
}
```

익명메서드로 쓰레드 함수 사용

```
using System;
using System.Threading;
class Program{
    static void Main(string[] args)    {
        Thread thread = new Thread(
            delegate (object obj)
            {
                Console.WriteLine("ThreadFunc in anonymous method called!");
            });

        thread.Start();
        thread.Join();
    }
}
```

변수에 담은 익명 메서드

```
using System;

class Program
{
    delegate int? MyDivide(int a, int b);

    static void Main(string[] args)
    {
        MyDivide myFunc = delegate (int a, int b)
        {
            if (b == 0)
            {
                return null;
            }

            return a / b;
        };

        Console.WriteLine("10 / 2 == " + myFunc(10, 2)); // 출력 결과: 10 / 2 == 5
        Console.WriteLine("10 / 0 == " + myFunc(10, 0)); // 출력 결과: 10 / 2 ==
    }
}
```

- C# 컴파일러는 익명 메서드가 사용되면 빌드 시점에 중복되지 않은 특별한 문자열을 하나 생성해 메서드 이름으로 사용하고 deleage 예약어 다음의 코드를 분리해 해당 메서드의 본체로 넣는다.

12.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 선언이 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 BirthYear는 설정 값이 숫자 이므로 int로, Name은 문자열이 설정되었으므로, string으로 형식을 추정한다.

```
using System;
namespace Sample{
    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);
            }
        }
    }
}
```