C# 2.0 / 3.0 New Features

작성자 : 리얼웹 개발본부 배성혁

들어가는 말…

.NET Framework 버전과 C# Specification 버전이 꼭 일치하지는 않지만, 유사하게 버전이 올라가고 있습니다. 2009년 현재 C# 4.0 Spec Preview가 공개되었고, 이를 .NET Framework 4.0, VS.NET 2010 부터 사용이 가능할 것 같습니다.

여러분은 처음 .NET Framework 또는 C#을 공부하면서, 문법, API 등을 공부하고, 쓰고 있지만 어떤 버전을 쓰고 있는지는 잘 모르셨을 것이라 생각됩니다. 하지만 S/W 개발쪽에 경력이 쌓이게 되면, version에 따라 변화되는 기능에 대해 어느정도 무의식 속에 파악하고, 수용하게 됩니다.

보통 어느정도 초보딱지를 뗀 개발자라면, 자신이 사용하는 툴이나 라이브러리를 가장 최신의 것으로 유지하려고 하는 경향이 있습니다. Early adapter라고 하지요… 하지만, 신제품 개발단계가 아닌 제품의 라이프사이클을 고려하게 되면, 꼭 최신의 제품을 사용할 수 없을 때가 더 많아지기도 합니다. – backward compatibility가 있는 제품이라면 영향이 작을 수는 있지만, Microsoft는 워낙 악명이 높습니다. 차라리 open source 진영에서 더 신경쓰고 있는 부분입니다.

제품 Spec을 결정하던가, 버전관리를 주도하는 개발자는 사용하는 제품의 새로운 버전이 출시되었을 때, 꼭 Upgrade를 해야 하는지를 고려해야 합니다. 아예 눈/귀를 닫아서는 안되고, Library 하나라도, 담당 제품의 많은 장점을 줄 수 있는 기능이 있는지?, 제품 개발 혹은 Customizing 시의 공수, 외부 인터페이스 변화 등을 고려해서, 신 버전 사용을 결정해야 합니다.

고려대상은 사용하는 제품마다 다르고, 독자 판단만으로 안되는 경우도 많습니다. 특히나 MS 제품은 무조건 신버전을 사용하도록(지들 제품 많이 팔리도록) 강요하는 경우가 많습니다. (여기에 놀아나지 말아야 하지만… 가만보면 제가 가장 많이 놀아나는 것도 같고…) – 고객 중에서도 최신버전을 좋아하는 사람. 기존 버전을 고수하는 사람 등등 참 많습니다.

어쨌든 새로운 제품에 대해서는 항상 새로운 기능이 무엇인지, 뭐가 바뀌었는지, 기존 제품과의 호환성은 어떤지를 파악해야 합니다. 새로 도입할 제품도, 현재 버전이 어떤기능을 가졌는지, 발전 단계는 어떤지, 앞으로의 로드맵에는 어떤 기능이 있는지를 조사해야 합니다.

그럼 이제부터 .NET Framework 2.0/3.0 - 이제 4.0이 공개될 건데 왠 뒷북? 이라 할 수도 있지만 사실 개발자들이 C# 2.0 spec 조차도 잘 사용하지 못하는 경우가 대부분이어서 – 의 새로운 기능에 대해 살펴보겠습니다.

먼저 C# 2.0 Spec부터 살펴보죠… 개인적으로 C# 2.0의 새로운 특성 중에 제일 반겼던 것은 Generic 을 지원한다는 것입니다. 물론 다른 좋은 기능들도 많지만, 굵은 글자로 되어 있는 부분이 제게는 도움이 많이 되었던 것들입니다.

대부분 실제 많이 사용하고 있는 기능들이므로, 설명이나 의미는 생략하겠습니다. 다만 Generic에 대해서는 다음 기회에 다시 설명하도록 하겠습니다.

표 . C# 2.0 New Features

|  |  |
| --- | --- |
| 항목 | 설명 |
| **Generic** | 형식 매개 변수를 사용하여, run time 시에 cat, boxing 작업에 대한 위험없이 단일 class 작성 가능 |
| **Iteration**  Yield return  Yield break; | Public void GetEnumerator()  {  for(int i=0; i< \_data.Count; i++)  yield return \_data[i]; } |
| Partial class | public partial class Employee { public void DoWork() { … } }  public partial class Employee { public void GoToLunch() { … } } |
| **Nullable type** | System.Nullable 구조체의 Instance  int? num = null;  int count = num ?? 0;  int count = num.GetValueOrDefault(0); |
| **Anonymous method** | Delegate void Del(int x);  Del d = delegate() {  Console.WriteLine(“delete “ + x);  }; |
| Static class | public static class Utils {  public static void UtilityMethod() { … }  } |
| Property accessor | string FullName { get; **protected** set; } |
| Fixed size buffer | Public struct MyArray { public fixed char PathName[128]; } |
| Friendly assembly | [assembly:InternalsVisibleTo(“cs\_friend\_assemblies\_2”)] |

한가지 짚고 넘어가고 싶은 것은 Anonymous method 라는 기능입니다. 이는 java 문법에서 제공되는 기능을 차용한 것으로 보이는데, 앞으로 설명할 C# 3.0 spec에서 이를 토대로 많은 기능들을 구현할 수 있음을 알 수 있습니다.

다음으로는 C# 2.0 (.NET Framework 2.0 이 더 옳은 표현이지만, C#이 대표 Language이므로) 에서 새롭게 제공하는 delegate 에 대해 알아 보겠습니다.

표 . C# 2.0 중요 Delegate

|  |  |
| --- | --- |
| Type | Definition |
| Action | public delegate void Action<T> ( T arg )  public delegate void Action<T1, T2> ( T1 arg1, T2 arg2 )  public delegate void Action<T1, T2, T3> ( T1 arg1, T2 arg2, T3 arg3 )  public delegate void Action<T1, T2, T3, T4> ( T1 arg1, T2 arg2, T3 arg3, T4 arg4 ) |
| Func | public delegate TResult Func<T, TResult> ( T arg )  public delegate TResult Func<T1, T2, TResult> ( T1 arg1, T2 arg2 )  public delegate TResult Func<T1, T2, T3, TResult> ( T1 arg1, T2 arg2, T3 arg3 )  public delegate TResult Func<T1, T2, T3, T4, TResult> ( T1 arg1, T2 arg2, T3 arg3, T4 arg4 ) |
| Predicate | public bool Predicate<T>( T obj ) |

표 2.에 나온 Action, Func, Predicate라는 delegate는 일반적으로는 많이 사용하는 delegate를 미리 정의해 둔 것이라 볼 수 있습니다. 특징중에 Generic을 이용한 것인데, Generic을 이용하면, signature는 하나지만, 다양한 Type을 지원하는 단 한 개의 delegate만을 정의해서 사용할 수 있는 장점이 있습니다.

Action은 void function point (C Language 용어)라 할 수 있는데, 지정된 형식의 인자를 N를 받아 수행하고, 반환값이 없는 함수를 말합니다.

Func는 반환값이 있다는 차이가 있고, Predicate는 Func<T, bool>(T arg) 와 같은 뜻입니다.

위의 delegate를 보시면 메소드 인자의 개수가 최대 4개인데, 더 많은 인자를 가지는 메소드는 개발자가 추가로 정의하여 사용이 가능합니다.

사용 예는 다음과 같습니다.

|  |
| --- |
| Func<string, string> convert = delegate(string s)  {  return s.ToUpper();  }; // anonymous method  string name = "Dakota";  Console.WriteLine(convert(name));   * DAKOTA |

convert라는 delegate를 만들고, 이를 사용하는 방식을 설명했습니다. Anonymous method 를 사용하여, 실제 delegate를 위한 메소드 원형의 class에서 정의하지 않고, 내부적으로 “이름없는” 메소드를 정의하여 사용할 수 있습니다.

더 자세한 내용은

[ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.ko/fxref\_mscorlib/html/da586d48-5345-2de1-63f2-d6208f298942.htm](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.ko/fxref_system.core/html/a0ab867a-da51-dd82-2e1a-e87b93712102.htm)

<ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.ko/fxref_system.core/html/a0ab867a-da51-dd82-2e1a-e87b93712102.htm>

를 참고하세요.

C# 2.0의 새로운 특징은 이쯤에서 정리하고 C# 3.0의 새로운 기능에 대해서 알아봅시다.

우선 MSDN의 C# 3.0의 새로운 기능에 대한 설명은 <ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.ko/dv_csref/html/e5193336-6adb-471e-ab22-e3fc60e0fa52.htm> 에 자세히 나와 있으니 참고하세요. 특히 새로운 기능의 필요성, 다른 새로운 기능의 토대가 되는지 등등을 파악하 보시기 바랍니다.

참고로 C# 3.5의 경우에는 <ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.ko/dv_fxintro/html/1b0d91e8-2d01-47a0-affc-966894de71f8.htm> 에 있습니다. – 3.5에서는 Language 차원에서 크게 발전된 건 없습니다.

그럼 C# 3.0에서 새롭게 도입된 기능을 알아 봅시다.

표 . C# 3.0 Language enhancements

|  |  |
| --- | --- |
| 항목 | 설명 |
| Local variable type inference | 지역변수와 함께 사용될 경우 var 키워드는 초기화 문의 오른쪽에 있는 식에서 변수 또는 배열 요소의 형식을 유추하도록 **컴파일러**에게 지시한다.  var i = 5;  var s = "Hello";  var a = new[] {1,2,3};  for(var x =1; x <10; x++)  {  // some codes  }  **var 키워드는 "Variant"를 의미하지 않으며 변수가 느슨한 형식이거나 런타임에 바인딩됨을 의미하는게 아닙니다.** |
| Object initialize | 생성자를 명시적으로 호출하지 않고 개체 초기화를 수행할 수 있게 한다.  var cat = new Cat {Age = 10, Name = "Sylverster"};  Object intializer에 익명 형식 허용 / 단 nullable 형식은 불가  var o = new {Title = "Book In Action", Publisher = "Manning"}; |
| Collection initialize | List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7 };  List<int> digits2 = new List<int> { 0+1, 12 % 3, MakeInt() };  List<Cat> cats = new List<Cat> {  new Cat { Age=10, Name=“Sylverster” },  new Cat { Age=4, Name=“Peaches” },  null  }; |
| Anonymous types | 익명 형식은 형식을 먼저 명시적으로 정의할 필요 없이 읽기 전용 속성 집합을 단일 개체로 캡슐화하는 편리한 방법을 제공합니다. 형식 이름은 컴파일러에 의해 생성되고 소스 코드 수준에서 사용할 수 없습니다. 속성의 형식은 컴파일러에 의해 유추됩니다. 다음 예제에서는 Amount 및 Message라는 두 개의 속성을 사용하여 초기화되는 익명 형식을 보여 줍니다.  var o = new {Title = "Book In Action", Publisher = "Manning"}; |
| Auto-implemented properties | 속성중에 추가 논리가 필요없을때, 속성 선언을 간결하게 한다.  class MyClass  {  public double TotalAmount { get; set; }  public string Name { get; private set; } // read-only  public int CustomerId { get; protected set; } // read-only for user, writable for drived class  } |
| Extension Methods | 인스턴스 메서드 구문을 사용하여 호출할 수 있는 정적 메서드를 사용하여 기존 클래스를 확장합니다.  public static class MyExtensions {  public static int WordCount(**this** String text) {  return str.Split(new char[] { ‘ ‘, ‘.’, ‘?’, StringSplitOperations.RemoveEmptyEntries).Length;  }  }  String s = "New Features in C# 3.0";  int wordCount = s.WordCount(); |
| Lambda expression | 대리자나 식 트리에 바인딩할 수 있는 입력 매개 변수를 가진 인라인 식을 사용할 수 있게 합니다  Func<int, int> myPower2 = **x => x\*x;**  int x2 = myPower2(5); // x2 = 25 |
| Partial methods | Abstract class의 abstract method와 유사한 효과를 나타낸다. 몇가지 제한사항이 있음  // define in file1.cs  partial void onNameChanged();  // define in file2.cs  partial void onNameChanged()  {  … } |

C# 3.0 기능 중에 몇 가지는 코딩 시에 작업을 수월하게 하고, 코드 분석을 쉽게 해 주는 문법적인 발전도 있고, Lambda expression과 같이 동적 표현이 가능케 해주는 획기적인 기능도 추가되었습니다 (lambda expression은 다음에 얘기할 LINQ에서도 설명할 예정입니다.)

요즘 Resharper 4.1 이상을 깔면 추천 코드를 제공해 주는데, 특정 타입을 “var” 예약어로 바꾸는 것을 추천하는 경우가 많습니다. “var”라는 것이 어찌보면, 모든 수형을 표현하는 것처럼 보이므로 COM+에서 여러 Language에서 상이한 수형을 표현하기 위한 VARIANT 를 연상할 수 있는데, 실제 var는 동적으로 수형이 결정되는 VARIANT 와는 달리 컴파일 시에 결정됩니다. C# 코드상에서만 결정이 안되었지, 컴파일된 IL Code 상에서는 실제 수형으로 선언되고, 사용되고 있음을 알 수 있습니다.

아래 원본 C# 코드와 Reflector를 통한 diassembling 된 코드를 비교해 보십시요.

“var”, “object initializer” 를 이용하게 되면 아래와 같이 약간의 변형을 거쳐지만, 원하는 결과가 되는 것은 마찮가지 입니다.

|  |
| --- |
| // C# 코드  public virtual Config CreateConfig(string section, string name, string value)  {  var config = new Config(name, value)  {  Application = AdminContext.Current.Application,  Enterprise = AdminContext.Current.Enterprise,  Section = section,  IsEnabled = true  };  Session.SaveOrUpdate(config);  return config;  } |
| // Deassembler code  public virtual [Config](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://RealWeb.RealAdmin:3.5.2.1002/RealWeb.RealAdmin.DomainModel.Config) [CreateConfig](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://RealWeb.RealAdmin:3.5.2.1002/RealWeb.RealAdmin.Services.ApplicationService/CreateConfig(String,String,String):RealWeb.RealAdmin.DomainModel.Config)([string](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://mscorlib:2.0.0.0:b77a5c561934e089/System.String) section, [string](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://mscorlib:2.0.0.0:b77a5c561934e089/System.String) name, [string](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://mscorlib:2.0.0.0:b77a5c561934e089/System.String) value)  {  [Config](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://RealWeb.RealAdmin:3.5.2.1002/RealWeb.RealAdmin.DomainModel.Config) <>g\_\_initLocal7 = new [Config](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://RealWeb.RealAdmin:3.5.2.1002/RealWeb.RealAdmin.DomainModel.Config/.ctor(String,String))(name, value);  <>g\_\_initLocal7.[Application](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://RealWeb.RealAdmin:3.5.2.1002/RealWeb.RealAdmin.DomainModel.Config/property:Application:RealWeb.RealAdmin.DomainModel.Application) = [AdminContext](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://RealWeb.RealAdmin:3.5.2.1002/RealWeb.RealAdmin.AdminContext).[Current](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://RealWeb.RealAdmin:3.5.2.1002/RealWeb.RealAdmin.AdminContext.Current).[Application](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://RealWeb.RealAdmin:3.5.2.1002/RealWeb.RealAdmin.AdminContext.Current/property:Application:RealWeb.RealAdmin.DomainModel.Application);  <>g\_\_initLocal7.[Enterprise](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://RealWeb.RealAdmin:3.5.2.1002/RealWeb.RealAdmin.DomainModel.Config/property:Enterprise:RealWeb.RealAdmin.DomainModel.Enterprise) = [AdminContext](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://RealWeb.RealAdmin:3.5.2.1002/RealWeb.RealAdmin.AdminContext).[Current](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://RealWeb.RealAdmin:3.5.2.1002/RealWeb.RealAdmin.AdminContext.Current).[Enterprise](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://RealWeb.RealAdmin:3.5.2.1002/RealWeb.RealAdmin.AdminContext.Current/property:Enterprise:RealWeb.RealAdmin.DomainModel.Enterprise);  <>g\_\_initLocal7.[Section](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://RealWeb.RealAdmin:3.5.2.1002/RealWeb.RealAdmin.DomainModel.Config/property:Section:String) = section;  <>g\_\_initLocal7.[IsEnabled](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://RealWeb.RealAdmin:3.5.2.1002/RealWeb.RealAdmin.DomainModel.Config/property:IsEnabled:System.Nullable%3cBoolean%3e) = true;  [Config](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://RealWeb.RealAdmin:3.5.2.1002/RealWeb.RealAdmin.DomainModel.Config) config = <>g\_\_initLocal7;  this.[Session](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://RealWeb.RealAdmin:3.5.2.1002/RealWeb.RealAdmin.Services.AdminDataServiceBase/property:Session:NHibernate.ISession).[SaveOrUpdate](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://NHibernate:2.1.0.2001:aa95f207798dfdb4/NHibernate.ISession/SaveOrUpdate(Object))(config);  return config;  } |

Deassembled code에서 보듯이 var는 Config 수형으로 선언되어 있습니다. 확실히 VARIANT 수형과는 다른 것입니다. (C# 4.0 spec에 보면 dynamic language 기능이 포함되는데, 이때는 기존 variant와 비슷한 수형이 나올 것입니다.)

다음으로 .NET Framework 3.0 부터 아주 새로운 개념이 추가 되었는데, [LINQ](http://msdn.microsoft.com/ko-kr/library/bb397897.aspx) (**L**anguage **IN**tegrated **Q**uery) 입니다.

이 부분에 대한 자세한 설명은 MSDN에 있으니 참고하시고, 다른 글로서 설명을 드리겠습니다.

이 글에서는 LINQ의 기본적인 문법을 설명하고, 위의 C# 3.0 에서 추가된 특징이 어떻게 활용되는지 (반대로 LINQ 라는 개념을 구현하기 위해 C# 3.0의 새로운 기능들이 만들어졌다라고 볼 수 있습니다.)

LINQ를 사용하는데는 크게 두가지 방식이 있습니다.

1. Query Expression ( SQL 문과 비슷 )
2. Query Operator (Extension Methods 사용)

우선 2가지 방식의 예를 봅시다.

1. Query Expression

Query Expression은 우리가 잘 알고 있는 Database 질의 형식과 유사하다. 즉 SQL 문법과 유사하게 사용할 수 있고, 기본적으로 메모리에 적재된 Object들로부터 질의를 수행하고, 결과를 받을 수 있다. (물론 LINQ to SQL, LINQ to XML 등 다양한 질의 대상이 추가될 수 있다)

표 . Query Expression Format

|  |
| --- |
| **from** [ type ] id **in** source  [ **join** [ type ] id **in** source **on** expr **equals** expr [ into id ] ]  { **from** [ type ] id **in** source | **let** id = expr | **where** condition }  [ **orderby** ordering, ordering, … ]  **select** expr | **group** expr **by** key  [ into id query ] |

Query Expression 예를 들어 보자.

코드 . Query Expression Sample

예제 코드는 시스템에 수행중인 모든 프로세스 중에 메모리가 20Mb 이상을 사용하는 프로세스를 찾아, 사용하는 메모리 양을 기준으로 정렬을 수행하고, 그 프로세스의 속성 중에 Id, Process Name 만으로 새로운 anonymous type을 생성해서 반환하도록 한다.

기존 코드로 이 작업을 수행하려면, Filtering, Sorting, 새로운 Class 정의 등 상당히 많은 코드를 구현해야 하지만, 예제 코드를 사용하면 (질의 문법에 약간이라도 익숙하다면) 직관적이고, 코드도 상당히 간단해 진다.

1. Query Operator

Query operator는 query expression과는 달리, 메소드 호출을 이용하여 질의를 수행하므로, 기존 C# Language 문법과 유사하다. 다만 C#의 새로운 특징인 (사실 새로운 기능이 아니더라도 가능하게 할 수는 있다. – Nhibernate의 ICriteria, DetachedCriteria가 그런 경우이다.) extension methods 를 이용하여, 코드를 일목요연하게 표현 할 수 있다.

코드 . Query Operator Sample

Query expression과 같은 작업을 수행하는 코드지만, 기존 C# 코드와 큰 차이가 없다. 다만 Lambda expression에 대한 공부는 좀 해야겠지요…

두 예제 모두 붉은 글자로 코드상에서 사용된 C# 3.0의 새로운 특징을 명시했습니다. 많은 특징들을 LINQ가 이용하므로, 여러분 중 LINQ를 쓰시고자 한다면, C# 3.0의 새로운 특징들을 먼저 공부해야 합니다. 물론 이러한 새로운 특징들이 꼭 LINQ 기술을 쓸 때만 유용한 것은 아니고, 전통적인 코드상에서도 상당한 잇점을 제공합니다.

그럼 이 두 가지 방식 중에 어떤 걸 이용해야 할까요?

답은 “당신 맘대로” 입니다.

그럼 넌? 저요? 전 Query operator 방식을 더 선호합니다.

이유는?

뭐 여러가지가 있겠지만, 가장 먼저 생각나는 건 메소드 사용이 익숙하고, 이해하기에도 더 쉽거든요.

또다른 큰 이유는 Query Operator는 확장이 가능하지만, Query expression은 예약어 방식이라, MS 말고는 확장이 곤란합니다.

표 5.에서는 Query operator와 expression 간의 대응관계를 표시하고 있는데, 실제 예약어 방식인 query expression에서는 제공하지 않는 부분이 많습니다. 다만 query operator의 메소드 명이 제게는 직관적이지 않은 부분은 불만입니다.

표 . Query Operator & Expression

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Expression | Operator | Expression |
| All | N/A | OrderBy | Orderby …, … |
| Any | N/A | OrderByDescending | Orderby …, … descending |
| Average | N/A | Select | Select |
| Cast | From int I in numbers | SelectMany | Multiple from clauses |
| Count | N/A | Skip | N/A |
| Distinct | N/A | SkipWhile | N/A |
| GroupBy | Group … by … [ into …] | Sum | N/A |
| GroupJoin | Join … in … on … equals … into … | Take | N/A |
| Join | Join … in … on … equals … | TakeWhile | N/A |
| LongCount | N/A | ThenBy | Orderby …,… |
| Max | N/A | ThenByDescending | Orderby …,… descending |
| Min | N/A | Where | Where |
|  |  |  |  |

이제까지 C# Language의 버전별 새로운 특징을 살펴봤고, 마지막에는 LINQ에 대한 간략한 소개를 했습니다. 앞서 설명했지만, 새로운 버전으로 Upgrade시에는 새로운 기능이 뭐가 있고, 어떤 장점이 있는지 파악하고 선택적으로 사용하는 것이 우선입니다. 특히나 C# Language 는 가장 기본이 되는 특성이므로, 꼭 알아둬야 하고, 자신의 것으로 만들 필요가 있습니다.