Microsoft

Microsoft® Visual Studio®

FIRST LOOK C# 4.0

저자 : 강보람 Visual Studio 2010 팀 블로그 멤버

Contents

들어가며	5
0. C# Chronicle	6
0.1 C# 1.0	6
0.2 C# 2.0	7
0.2.1 제네릭	7
0.2.2 익명메서드와 클로저	7
0.2.3 반복자	8
0.2.4 파셜 타입	9
0.2.5 널 가능타입	9
0.3 C# 3.0	10
0.3.1 암시적인 타입을 적용한 지역변수와 익명 타입	10
0.3.2 확장 메서드	11
0.3.3 객체와 컬렉션 초기화 생성자	11
0.3.4 람다식과 쿼리 표현식	12
0.3.5 Expression Trees	12
0.3.6 자동으로 구현되는 프로퍼티	13

Contents

1. Dynamic 14

1.1 덕 타이핑 14

1.1.1 정적 언어와 동적 언어? 17

1.2 Dynamic 타입의 추가 18

1.2.1 바인딩 과정 26

1.3 Expression Trees 27

1.3.1 Expression Trees v2 30

1.4 바인딩 더 깊게 보기 32

1.4.1 동적 수신자의 경우와 캐시 33

1.4.2 정적 수신자의 경우 34

1.5 Dynamic과 관련된 오버로딩과 형변환 35

1.5.1 오버로드 판별 36

1.5.2 Dynamic과 다른 타입간의 형변환 36

1.5.3 프로퍼티 39

1.5.4 인덱서 39

1.5.5 형변환 40

1.5.6 연산자 40

1.5.7 델리게이트 호출 41

1.6 예제 41

2. Co- Contravariance 45

2.1 일반적인 Co- Contravariance의 개념 45

2.2 타입 시스템에서 Co- Contravariance의 개념 46

3. Named and Optional Parameters 51

마치면서 53

참고자료 54

들어가며...

2002년에 정식으로 세상에 모습을 드러냈던 C#이 어느덧 4번째 버전을 눈앞에 두고 있습니다. MS진영 개발자들이 통합된 개발환경에서 새로운 패러다임에 적응하면서, 좀 더 생산성을 낼 수 있도록 C#이 그 동안 도와주었는데요. 사실, 프로젝트에 치이다 보면, C#의 내면을 상세하게 들여다볼 기회는 거의 없다고 봐도 무방합니다. 빡빡한 일정에 그저 하루하루 몰려드는 일을 처리하다 보면, 녹초가 되어서 새로운 개념을 공부하는 건 뒤로 미뤄지기 마련이죠. 그리고 프로젝트에서 써야 해서 새로운 기능을 공부한다 해도, 그저 사용법만을 이해하고 넘어가는 경우가 대부분입니다. 그래서, 어떤 언어발전의 흐름을 잡기 보다는 그저 사용법만을 이해하고 넘어가므로, 그 다음에 더 발전되어가는 상황을 제대로 따라잡기 더욱 힘들어 지는 것이죠.

C#은 현대적이고 간결한 언어를 목표로 하고 출발했으며, 여전히 같은 목표를 가지고 발전해 가고 있습니다. 그러다 보니, 각종 프로그래밍 언어에서 이슈가 되는 부분들 중, 정말 개발자의 생산성에 기여할 수 있는 부분들을 흡수해 나가고 있습니다. 이로 인해 우리가 얻을 수 있는 장점은, 새로운 언어를 배울 필요 없이 우리가 가장 익숙하게 쓰고 있는 개발 툴 위에서(VS.NET), 가장 익숙하게 사용하는 언어를 통해서(C# 혹은 VB.NET), 생산성의 향상을 도와주는 기능을 기존의 연장선상에서 첨부해서 사용할 수 있다는 점입니다. 하지만, C#은 간결함을 또 하나의 목표로 하고 있으므로 이로 인해서 좀 더 복잡한 모습을 가지게 됩니다. 즉, 기능이 계속해서 추가됨에 따라서 언어가 가지는 기능의 합집합은 점점 더 커져갑니다. 하지만, 언어에 추가되는 기능 때문에 기존에 동작하는 기능의 사용법이 바뀌는 건 막아야 하며, 새로 추가되는 기능 역시 기존의 사용법과 직관적으로 유사하게 구현해서 사용자들이 기존에 해왔던 경험을 바탕으로 새로운 기능을 이용할 수 있도록 해주는 것이죠. 그로 인해 또 하나의 복잡성이 발생하기도 하는데요, 바로 기존과의 호환성을 위해서 표면아래서 수행되는 작업들 입니다. 물론, 이런 부분은 깊숙이 들여다보지 않으면 보이지 않게 설계되어 있습니다.

이번 퍼스트 룩 C#은 이런 개발자들의 현실을 감안해서, 최대한 친절하고, 자세하게 C#의 새로운 기능에 대해서 들여다 보려고 생각합니다. 다만, 집필자의 능력이 매우 부족하므로 제대로 설명을 드리지 못하거나, 개념적인 설명으로 끝나는 부분이 있겠지만, 너그러운 마음으로 봐주시고, 자신의 지식을 활용해서 여러 커뮤니티 등에서 피드백으로 이 책에서 소개하는 내용을 더 쓸모 있는 내용으로 업그레이드 시켜 주시길 기대하는 바입니다.

C# 4.0에서 추가되는 기능들은 크게 아래와 같습니다.

1. Dynamic
2. Co – Contravariance
3. Named and optional parameters
4. COM Interop

위 내용 중에서, 1번은 새로운 프로그래밍 패러다임을 받아들이는 부분이고, 2번과 3번과 4번은 기존에 존재하는 부분을 새로운 요구에 맞춰서 개선한 부분입니다. 이제 앞으로 각 부분에 대해서 하나씩 하나씩 자세하게 설명을 드려볼까 합니다.

0. C# Chronicle

chronicle은 연속적인 사건이나 이벤트를 기록한 것을 지칭할 때 쓰이는 데요. 이미 C#도 네 번째 버전이 등장했으니 chronicle로 한번 정리해 볼만한 때가 온 것 같습니다. 본격적인 이야기로 들어가기 전에, 우선 C# 1.0부터 그 동안 C#이 걸어온 길과 그 과정에서 추구한 것은 무엇이었는지, 한번 돌아보는 시간을 갖도록 하겠습니다.

0.1 C# 1.0

"C# 1.0은 훌륭하게 관리되는 개발환경을 이 땅에 정착시키는 .NET Framework의 선두주자로서의 역사적 사명을 띄고 이 땅에 태어났다."
– .NET 국민 헌장 1조 1항

지금은 CO-Evolution전략을 통해서, C#과 VB.NET이 그래도 동등하게 발전할 수 있는 기틀을 마련했습니다만, 그 당시에는 .NET 국민 헌장에서 볼 수 있듯이, C#이 단연 돋보이는 선두주자였습니다. 오로지 닷넷을 위해서 고안된 언어였기 때문이죠.

그 당시에 C# 1.0을 통해서 닷넷 뿌리내리고자 했던 건 바로, 개발자의 실수를 유발하기 쉽던 메모리 관리나 배열 범위체크 등의 이슈를 닷넷의 런타임을 통해서 자동으로 관리하고, 개발자는 그만큼 더 높은 추상화단계의 전략을 고민할 수 있도록 하자는 "Managed"의 도입이었습니다.

그 외에도 C# 1.0에는 매우 현대적인 기능과 문법적으로 편리한 코딩을 도와주는 Syntactic sugar 등이 추가되었었습니다. 그 항목을 보자면 아래와 같습니다.

* 어트리뷰트 – AOP개념으로도 생각할 수 있는 기능으로, 클래스나 메서드등에 특정 속성을 쉽게 부여할 수 있었다. 특히 웹 서비스의 [WebMethod]같은 어트리뷰트가 주목을 받았었다.

* 델리게이트 – 함수포인터를 안전하게 개량했다고 해서 '안전한 함수포인터'라는 이름으로 불리기도 했었다.

* 리플렉션 – 런타임에서 특정 객체의 멤버를 조회하고 실행도 가능한 강력한 기능을 제공했다.

* 값형 / 참조형간의 박싱&언박싱 – 모든 객체의 최상위 객체인 object타입이 소개되었고, 스택에 저장되면서 직접 값을 가지고 있는 값형과 힙에 저장되며 값에 대한 참조를 가지는 참조형이 소개되었다. 그리고 두 타입간의 변환을 위해서 값형을 객체로 포장한다는 의미의 박싱(boxing)과 포장된 객체를 벗겨내고 값을 꺼낸다는 의미의 언박싱(unboxing)이 소개되었다. 당시에는 제네릭이 도입되기 전이었으므로, ArrayList같은 컬렉션을 사용하려면 object타입으로 박싱&언박싱이 필수였다.

* foreach – 반복문을 작성하기 위한 더 간편한 문법을 제공했다.

* 프로퍼티 / 인덱서 – private필드의 입/출력을 위해서 Get/Set메서드가 작성되던 것을 탈피하고, 멤버 접근이나 인덱스를 이용한 내부 멤버의 접근을 매우 직관적으로 표현할 수 있게 도와주었다.

0.2 C# 2.0

> "제네릭이 법으로 허용되기 전까지, 우리 정말 열심히 포장을 하고, 또 포장을 풀고... 그런 무의미한 일을 반복해야 했었죠. 나중에야 안 사실이지만, F#연구소에서 열심히 연구를 한 탓에 제네릭 허용법이 통과되었더군요. 큰 빚을 졌어요."
> – Mr. 닷넷 런타임 (2002 ~ 현재)

0.2.1 제네릭

제네릭은 어떤 타입이든지 될 수 있는 타입을 가리키는 데, 이 제네릭으로 인해서, 불필요한 코딩이 줄어들고, 다양한 경우에 일반적으로 쓰일 수 있는 코드를 작성할 수 있게 되었습니다. 즉, 기존에는 컬렉션을 써도 object로 집어넣고 꺼내고 하는 작업을 해야 했는데, 이제는 필요한 타입만 넣고 꺼낼 수 있게 되었다는 말이죠.

```
//제네릭 허용법 이전의 반복 노동.
ArrayList wayTooOldList = new ArrayList();
wayTooOldList.Add(1);//포장!
wayTooOldList.Add(2);//포장!
wayTooOldList.Add(3);//또 포장!

int a = (int)wayTooOldList[0];//포장 풀기!
int b = (int)wayTooOldList[1];//포장 풀기!
int c = (int)wayTooOldList[2];//또 포장 풀기!

//제네릭 허용법 이후의 쾌적한 노동.
List<int> brandNewList = new List<int>();
brandNewList.Add(1);
brandNewList.Add(2);

int d = brandNewList[0];
int e = brandNewList[1];
```

〈코드 1〉

0.2.2 익명메서드와 클로저

그리고 익명메서드와 클로저(closure)역시 2.0에서 추가 되었습니다. 마틴 파울러의 설명에 따르면, 클로저는 코드블록을 독립적으로 사용할 수 있으며, 그 코드 블록 안에서 바깥의 지역 변수를 사용하는 것도 가능하고, 그 코드블록을 변수에 할당하는 것도 가능하다고 말합니다. (http://www.martinfowler.com/bliki/Closure.html) 그 문서에서 클로저가 안 되는 언어로 C#을 이야기 하는데요. 이 문서가 작성된 시점이 2004년 이라, 아직 C#에 클로저가 없을 때 입니다.

```
int[] nums = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int factor = 3;
Predicate<int> isPowerOfNum = delegate(int num) { return num % factor == 0; };

int[] result = Array.FindAll(nums, isPowerOfNum);
```

```
foreach (int num in result)
{
    Console.Write(" {0} ", num);
}
```

〈코드 2〉

바로 위에서 보시는 코드가 클로저의 예제인데요. 강조 표시가 되어 있듯이, 익명메서드의 코드블록 안에서 바깥의 지역변수인 factor를 갖다 쓰고 있습니다. 그리고 그 코드블록을 isPowerOfNum에 저장해서, 조건을 판별하는 데 쓰고 있구요. 이런 익명메서드와 클로저를 활용한 코드는 3.0에서 람다로 발전하면서 더욱 편리하게 발전해 갑니다.

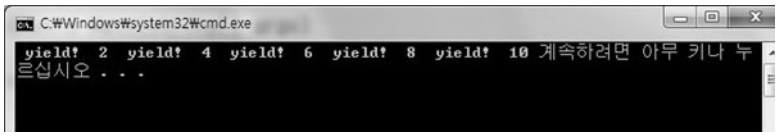
0.2.3 반복자

반복자는 점진적으로 증가하는 값이나 연속적인 값을 리턴 하는 메서드를 말합니다. IEnumerable〈T〉에서 사용할 값을 만들어서 리턴 하는 메서드를 말하는 데요. 나중에 더 자세히 설명 드리겠지만, IEnumerable〈T〉는 모든 값을 다 가지고 있는 게 아니라, 값이 요청될 때, 하나씩 값을 구해서 가져옵니다.

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

    foreach (int num in GetPowerOfNum(nums, 2))
    {
        Console.Write(" {0} ", num);
    }
}

public static IEnumerable<int> GetPowerOfNum(int[] nums, int factor)
{
    for (int i = 0; i < nums.Length; i++)
    {
        if (nums[i] % factor == 0)
        {
            Console.Write(" yield! ");
            yield return nums[i];
        }
    }
}
```

〈코드 3〉

바로 밑줄로 강조한 부분에서 값이 하나씩 만들어 지는데요. 결과를 보시면 IEnumerable의 특성이 더 명확하게 보입니다.



〈그림 1〉

보시듯이 값이 실제로 요청될 때, 그 값을 구하기 위한 연산을 하기 때문에, 조건에 맞는 값이 구해질 때 마다, "yield!"가 출력된 걸 보실 수 있습니다.

0.2.4 파셜 타입

파셜 타입(partial type)은 같은 클래스에 대한 정의를 다른 여러 파일에 나눠서 정의할 수 있도록 도와주는 기능입니다. 이 기능이 가장 먼저 눈에 띄었던 게 ASP.NET 2.0이었는데요.

```
namespace WebApplication1
{
    public partial class _Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
```

〈코드 4〉

기존에는 디자이너가 자동으로 선언한 코드와 사용자가 작성한 코드가 매우 복잡하게 얽혀서 지저분했습니다. 그래서, 파셜 타입을 통해서 자동 생성된 코드와 사용자가 작성하는 코드를 다른 파일에 나눠놓고, 컴파일 할 때 하나로 합칠 수 있도록 지원했습니다.

0.2.5 널 가능타입

값형과 참조형이 소개되면서, 참조형은 알려지지 않은 값을 가질 때, null로 선언할 수 있었지만, 값형은 빈 값이라도 반드시 내용을 가져야 했습니다. 널 가능타입(Nullable Type)은, 값형에서도 알려지지 않은 값을 가질 수 있도록 하는 기능입니다.

널 가능타입은 타입의 값과 추가적인 null상태를 가지는데요, 모든 값형은 널 가능타입이 될 수 있으며, 널 가능타입의 기본이 되는 타입과 동일한 형변환과 연산자를 지원합니다. 그리고 null value propagation을 지원합니다. null value propagation은 예를 들면, a.b().c(); 과 같은 구문이 있을 때, b의 결과 값이 null이라면, 예외발생 없이 c의 값도 null이 되는 걸 말합니다.

```
static void Main(string[] args)
{
    int? num = null;
    Console.WriteLine(num.ToString());
}
```

〈코드 5〉

즉, 위와 같은 코드에서 num의 값이 null이지만, num.ToString()의 호출은 예외 없이 빈칸이 출력되게 됩니다.

0.3 C# 3.0

"LINQ등장 전에는 우리 반복문의 인권은 없었지요. 늘 우리에게 '너희들이 산업 최전선에서 싸우는 산업 전사다!', '애국자다!' 같은 말을 하면서 노동을 미화시켰지만, 결국 우리에게 남은 건, 무한히 반복되는 단순업무였죠. '생활의 달인' 같은 거 나가면 뭘 합니까. 생활은 하나도 나아지는 게 없는데 말이죠. LINQ시행령은 그런 의미에서 반복문의 인권 선언과 같은 거였죠."
– "그 시절을 회고하며", Loop junior 3세

LINQ는 Language Integrated Query의 약자로서, 함수형 언어의 개념을 도입해서, 데이터 추출과 같은 작업에서 추상화 단계를 굉장히 높여줘서, 코딩 량도 줄었으며, 사고과정도 단순해 지도록 한 히트 상품입니다. LINQ에 익숙해지면, 기존방식으로 루프 문을 쓰는 게, 너무 불편하게 느껴지기도 할 정도로 편의성을 높여준 기능입니다. 그리고 C# 3.0에 추가된 기능은 거의 모두가 LINQ를 지원하기 위해서 추가 된 기능으로 봐도 무방할 정도로 C# 3.0의 핵심은 LINQ였습니다.

LINQ에 대한 설명을 하려면, 너무 길어지기 때문에, 위에서 잠깐 언급 드렸던, IEnumerable과 List의 차이점에 대한 설명은 http://www.vsts2010.net/43를 LINQ를 쓴 코드와 C# 2.0의 익명메서드를 사용한 코드의 차이점은 http://www.vsts2010.net/48을 참고하시면 될 거 같습니다.

0.3.1 암시적인 타입을 적용한 지역변수와 익명 타입

암시적인 타입을 적용한 지역변수(Implicitly typed local variable)은 쉽게 var타입을 이야기 합니다. 어떤 타입이 리턴 될지 명확하지 않은 경우에, 그냥 var타입으로 받아서 컴파일러가 타입 유추를 하도록 하는 기능입니다.

그리고 익명 타입(Anonymous type)은 LINQ의 쿼리 중간 과정으로 생기는 새로운 타입을 굳이 class로 정의하지 않아도, 필요할 때 익명 타입으로 객체를 만들어 낼 수 있는 기능입니다.

```
int num = 1;
string[] names = { "언냐2", "유어폰", "안드로메다폰" };

var pairs = names.Select(name => new { num = num++ , name = name}).ToList();

foreach (var pair in pairs)
{
    Console.WriteLine("num : {0}, name : {1}",pair.num, pair.name);
}
```

〈코드 6〉

보시면, names의 값과 번호를 뭉쳐서 새로운 타입을 만들어 내는데, new 뒤가 빈칸입니다. 즉, 미리 정의된 class는 없지만 임시로 타입을 하나 만들어 내는 것이죠. 이렇게 생성된 타입은 class가 정의되지 않았기 때문에, 변수를 선언할 수가 없는데요. 그래서 var타입이 등장해서, 타입을 알아서 유추해 줍니다. 그리고 아래쪽에서 그 임시타입의 멤버를 접근해서 값을 가져오는 걸 볼 수 있습니다.

0.3.2 확장 메서드

기존에 만들어진 타입에 추가로 메서드를 붙일 때 사용합니다. LINQ사용시, 헬퍼메서드의 개념으로도 사용되곤 합니다.

```
public static class ExtensionExam
{
    public static int GetMiddleNumber(this int[] nums)
    {
        int middleIndex = nums.Count() / 2;
        return nums.ElementAt(middleIndex);
    }
}

public class HelloDynamic
{
    static void Main(string[] args)
    {
        int[] nums = new int[] { 2, 9, 3, 4, 1, 7, 8, 5, 6 };
        Console.WriteLine(nums.GetMiddleNumber());
    }
}
```

〈코드 7〉

코드에서 강조 된 부분들이 중요한 부분입니다. 클래스와 메서드는 static으로, 그리고 this키워드와 함께, 확장 메서드를 붙일 타입을 지정합니다. 그러면, int[]타입인 nums에서 확장 메서드를 사용할 수 있게 되는 거죠.

0.3.3 객체와 컬렉션 초기화 생성자

기존에는, 객체를 생성할 때, 객체의 멤버필드에 값을 할당하려면, 할당하려는 값을 받아들이는 생성자를 따로 정의해줘야 했습니다. 그리고 필드의 숫자가 바뀌면, 그에 맞게 생성자를 수정해주곤 했었죠. 그런데, 특별히 생성자를 정의하지 않더라도 public으로 선언된 멤버변수나 프로퍼티에 원하는 만큼 값을 할당해줄 수 있으며, 컬렉션을 생성할 때도, 원하는 개수만큼 컬렉션에 포함될 요소를 명시해줄 수 있습니다.

```
public class Customer
{
    public string name;
    public string address;
}

public class HelloDynamic
{
    static void Main(string[] args)
    {
        Customer _boram = new Customer
        {
            name = "Sebastian Boram",
            address = "above the world"
        };
```

```
        List<Customer> _customers = new List<Customer>
        {
            _boram,
            new Customer{
                name = "William Sesik",
                address = "below the world"
            }
        };
    }
}
```

〈코드 8〉

위와 같이, 객체를 생성할 때, 멤버에 직접 값을 할당할 수도, 안 할 수도 있으며, 컬렉션을 초기화 할 때, 요소들을 직접 정의해줄 수도 있습니다.

0.3.4 람다식과 쿼리 표현식

람다식은 위에서 설명 드렸던, 익명메서드를 훨씬 깔끔하게 다듬어서 나온 것이구요. 쿼리 표현식은 컬렉션에 대해서 데이터를 추출할 때, 데이터 집합에 대한 쿼리를 C#으로 작성하는 것을 말합니다.

LINQ를 통해서 쿼리를 하는 방법은 크게 두 가지가 있는데요, 미리 정의된 메서드와 람다식의 조합으로 사용하는 방법과, SQL쿼리와 유사한 쿼리구문을 통해서 가져오는 방법이 있습니다.

```
int[] nums = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

IEnumerable<int> nums1 =
    from num in nums
    where (num % 2).Equals(0)
    orderby num descending
    select num;

IEnumerable<int> nums2 =
    nums.Where(num => (num % 2).Equals(0))
    .OrderByDescending(num => num);
```

〈코드 9〉

위의 두 가지 방식은 당연하지만, 같은 결과를 반환합니다.

0.3.5 Expression Trees

Expression Trees는 LINQ의 중추에 자리하고 있는 데요. 이 Expression Trees를 버전업 해서 C# 4.0에서도 매우 중요하게 쓰이기 때문에, 추후에 자세하게 설명을 드리겠습니다.

0.3.6 자동으로 구현되는 프로퍼티

기존의 프로퍼티는 특별한 get/set룰을 적용하지 않고, 그저 private필드의 값을 읽기 위해서 쓰이는 경우에도, get/set내부의 코드를 작성해줘야 했습니다. 그런데, 이제는 단순 읽기/쓰기를 위한 프로퍼티라면, 변수조차 선언할 필요 없이 자동으로 구현이 됩니다.

```
//기존의 방식
private int num;
public int Num
{
    get {return num;}
    set { num = value; }
}

//새로운 C# 3.0의 자동구현 프로퍼티
public int Num2
{
    get;
    set;
}
```

〈코드 10〉

이제, C# 1.0 부터 C# 3.0 까지의 변화를 간략하게 설명을 다 드린 것 같습니다. 이제 다음 부터는 앞으로 다가올 C# 4.0의 변화에 대해서 소개를 드리겠습니다. 혹시 위의 설명된 내용에 대한 세미나를 보고 싶으신 분은 (http://msdn.microsoft.com/ko-kr/ee309490.aspx에 들어가셔서, VSTS2010팀이 진행한 세션에서 "C# 연대기"를 보실 수 있습니다.)

1. Dynamic

우선은, C# 4.0의 가장 큰 변화라고 생각하는 Dynamic에 대해서 설명 드리겠습니다. Dynamic이라고 하면 굉장히 추상적인 느낌도 드는데요. 동적인 모든 걸 통칭하기 위해서 이렇게 적었습니다. 즉, C#은 static한 언어인데요, C#에서 dynamic하게 프로그래밍을 할 수 있으며, dynamic한 언어와의 연동도 지원한다는 그 모든 걸 통칭하기 위해서 적어본 제목인 거죠. 우선, 이 장에서 사용될 기본적인 용어에 대해서 설명을 해보겠습니다.

- Type
타입은 메모리덩어리에 대한 메타데이터로서, 그 메모리공간에 저장되어 있는 데이터의 종류를 구분한다.
(http://www.artima.com/weblogs/viewpost.jsp?thread=7590)

- Statically typed language (정적 프로그래밍 언어)
하나의 변수이름이 하나의 타입, 그리고 하나의 객체와 묶인다. 객체와 연결되지 않을 때, 그 값은 null이 된다.
그리고 변수이름과 묶일 수 있는 객체는 변수와 묶여 있는 타입과 같은 타입의 객체이다.
(http://pythonconquerstheuniverse.wordpress.com/2009/10/03/static-vs-dynamic-typing-of-programming-languages/)

즉, [string abc = "abcde";] 와 같은 구문이 있다고 할 때, 변수이름인 abc는 string과 묶이고, "abcde"라는 값을 가진 객체와 묶인다. 그리고 한번 abc라는 이름이 string이라는 타입과 묶인 이상, 이 abc와 묶일 수 있는 객체는 string타입의 객체만 가능하다.

- Dynamically typed language (동적 프로그래밍 언어)
변수이름 하나와 객체 하나를 묶으며, 실행 중에 다른 타입의 객체와 묶는 것도 가능하다.
(출처 위와 동일)

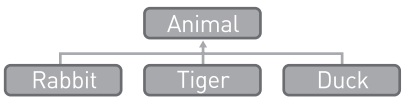
즉,
```
dynamic abc = "abcde";
abc = new List<int>();
```
위와 같이 abc가 "abcde"와 묶였지만, 바로 abc와 전혀 다른 타입인 List<int>와 묶는 것도 가능하다는 것이다.

〈표 1〉

위에서 간단한 용어에 대해서는 설명을 드렸지만, 동적 프로그래밍 언어에서 가장 중요한 개념인 덕 타이핑은 조금 큰 주제라 따로 설명을 드리도록 하겠습니다.

1.1 덕 타이핑

덕 타이핑은 위키피디아에 따르면 Alex Martelli가 처음 사용했다고 나옵니다.
(http://en.wikipedia.org/wiki/Duck_typing)
의미를 보자면, "오리의 한 종류인지 검사하지 않고 오리처럼 꽥꽥하는지, 오리처럼 걷는지 등등 오리가 하는 행동들을 할 수 있는지 검사한다"고 합니다. 즉, 기존의 정적 언어에서 하는 강한 타입체크를 보자면, 무조건 타입을 가지고 판단을 합니다. 함수에 매개변수로 넘기는 상황을 보자면, 파라미터에 명시된 타입과 일치하던가, 그 타입을 상속한 타입이어야만 합니다. 즉,



〈그림2〉

위와 같은 타입의 계층이 형성되어 있고, 아래와 같은 코드가 있을 때.

```
namespace ConsoleApplication1
{
    public class HelloDynamic
    {
        class Animal
        {
        }

        class Rabbit : Animal
        {
        }

        class Tiger : Animal
        {
        }

        class Duck : Animal
        {
            public void Quack()
            {
                Console.WriteLine("Quack!!!");
            }
        }

        static void Quack(Duck _duck)
        {
            _duck.Quack();
        }

        static void Main(string[] args)
        {
            Animal _animal = new Animal();
            Tiger _tiger = new Tiger();
            Duck _duck = new Duck();
            Rabbit _rabbit = new Rabbit();

            Quack(_animal); //(1)
            Quack(_tiger); //(2)
            Quack(_duck); //(3)
            Quack(_rabbit); //(4)
        }
    }
}
```

〈코드 11〉

위 예제코드에서, 3번 외에는 모두 오류를 내게 됩니다. 모두 duck과는 관계없는 타입이기 때문입니다. 예를 들어서 아래와 같은 메서드가 있다고 하면,

public void Feed(Animal _animal)

1. Feed(_animal);
2. Feed(_tiger);
3. Feed(_duck);
4. Feed(_rabbit);

위의 4가지 호출은 모두 유효한 호출입니다. 모두 Animal과 같은 타입이거나, 그 타입을 상속한 타입이기 때문이죠.

이렇듯이, 기존의 정적 언어에서는 타입시스템을 매우 견고하게 구축해서 안전성을 확보하려고 합니다. 타입시스템이 견고하다면, 잘못된 타입을 넘겨줄 가능성을 줄이고, 코딩 하는 시점에서 각 타입에 대한 메타데이터를 확보할 수 있기 때문에, 개발 툴에서 인텔리센스 같은 각종 편의를 제공해줄 수 있고, 컴파일러도 컴파일 하는 시점에서 타입에 대한 오류를 잡아낼 수 있습니다. 큰 시스템일 수록, 많은 사람들이 개발을 하게 되고, 그렇게 되면 다른 사람이 개발하는 모듈을 사용하는 경우가 많아집니다. 이런 경우에 견고한 타입시스템은 더욱 더 안정성을 확보할 수 있게 해주고, 협업을 용이하게 해줍니다. 그리고 초보들의 실수도 비교적 쉽게 잡아낼 수 있게 되죠.

그러면, 덕 타이핑을 사용하는 동적 언어들은 이 부분이 어떻게 다른 걸까요? 앞으로 설명드릴 dynamic타입을 사용해서 설명해보겠습니다.

```
namespace ConsoleApplication1
{
    public class HelloDynamic
    {
        public class Tiger
        {
            public void Feed(int count)
            {
                Console.WriteLine("사료 더 내놔~!!! 빨랑 내놔~~!!!!");
            }
        }

        public class Duck
        {
            public void Feed(int count)
            {
                Console.WriteLine("음 딱좋당~~~^^");
            }
        }

        public class Rabbit
        {
            public void Run()
            {
                Console.WriteLine("튀어~~~~");
            }
        }

        public static void Feed(dynamic _animal)
        {
            _animal.Feed(5);
        }

        static void Main(string[] args)
        {
            Tiger _tiger = new Tiger();
            Duck _duck = new Duck();
            Rabbit _rabbit = new Rabbit();

            Feed(_tiger); //(1)
            Feed(_duck); //(2)
            Feed(_rabbit); //(3)
        }
    }
}
```

〈코드 12〉

일단, dynamic은 object처럼 모든 타입을 담을 수 있지만, 덕 타이핑을 적용한 타입이라고 생각해보죠. 과연 위의 (1), (2), (3)번 호출은 어떻게 될까요?

이 질문에 답을 하기 위해서 어떤 정보가 필요할까요? 위에서 덕 타이핑은 타입을 검사하지 않고, '~~처럼 ~~가능한지' 검사한다고 했었습니다. 그러면, dynamic은 object처럼 모든 타입을 담을 수 있고 덕 타이핑이 적용되어 있다고 했으니, 일단 매개변수가 넘어올 때, 타입검사는 하지 않는다고 봐도 될 거 같습니다. 그렇다면, 실제 Feed메서드가 실행될 때, _animal에 담긴 객체가 'Animal처럼 Feed가능한지' 검사하게 되겠죠.

그런데, 더 넘어가기 전에 질문입니다. 과연 이 검사는 누가 수행하게 되는 걸까요? 컴파일러 일까요? 위에서 "실제 Feed메서드가 실행될 때""라고 했으니깐 컴파일러는 아닌 거 같네요. 그렇다면, 프로그램 실행 시에 환경을 관리하는 런타임이 이 부분을 처리한다고 볼 수 있을 것 같네요.

위 코드를 실행하면 (1), (2), (3)번 호출 중에서 에러가 납니다. 범인은 누구일까요? 바로 Rabbit이겠죠? 이 중에서 'Animal처럼 Feed할 수 있는'타입은 Tiger랑 Duck이니까요. 덕 타이핑은 이처럼, 객체의 타입으로 판단하지 않고, 객체가 무엇을 할 수 있는지를 보고 판단합니다. 그게 정적 언어와 가장 크게 다른 점입니다. 그리고 이번에 C# 4.0에 추가된 dynamic이라는 타입은 바로 이 덕 타이핑을 지원하기 위해서 추가된 타입인 것이죠.

그런데, 여기서 한가지 의문을 가져 볼 수 있습니다. 위에서 정적 타입 검사의 장점에 대해서 설명을 드렸었는데요. 그렇다면, 덕 타이핑은 그 장점들을 해치는 것 처럼 보이기도 하는데요, 과연 그 점은 어떻게 된 걸까요? C#도 드디어 너무 복잡해지는 잘 못된 길을 걷기 시작한 걸까요?

C# 4.0에 추가된 dynamic은 C#자체에 동적 언어의 장점을 '추가'한 것입니다. C#의 명세서를 보면, C#을 간결하고, 현대적인 언어라고 명시하고 있습니다. 즉, C#은 현대적인 요구에 따라서 점점 현대적으로 발전해 나가되, 간결함을 유지하겠다는 의미입니다. 그래서 일까요? C#은 현대적인 덕 타이핑을 지원하기 위해서 dynamic이라는 타입을 추가했지만, 추가된 부분도 기존에 동작하는 부분과 최대한 유사하게 동작하도록, 그리고 기존의 기능에는 변화가 없도록 해서 간결하도록 매우 고심한 흔적이 보입니다. 즉, dynamic에 대한 필요를 느끼지 못한다면, 쓰지 않아도 전혀 상관없다는 이야기입니다.

그리고 dynamic은 단순히 덕 타이핑만을 C#에 들여온 건 아닙니다. DLR을 통한 IronPython, IronRuby등의 동적 언어와의 연동을 가능하게 함과 동시에, 기존의 COM과의 상호운용성 역시 매우 향상시켰습니다. 이런 모든 장점을 수용하고 제공하기 위해서, dynamic이라는 타입과 그에 따른 지원 매커니즘이 추가된 것이죠.

1.1.1 정적 언어와 동적 언어?

기존의 C#만으로도 사실 큰 불편함 없이 프로그래밍을 해왔던 거 같은데, 왜 이런 것들이 계속 추가되면서 C#을 복잡하게 하고, 개발자들은 공부해야 할 내용이 늘어나는 걸까요? 아래의 보고서를 한번 보시죠.

> "상사와 회의 할 때 영어로 하면 내 의견을 끝까지 다 듣고 합리적으로 대화를 하는데, 우리나라 말로 하면 상당히 권위적으로 나오는 경향이 있다. 그래서 가급적 영어로 대화를 한다"
> (LGERI 리포트 "조직 내 침묵 현상", 황인경, 박지원, 2008. 12. 3)

즉, 우리가 평소에 사용하는 언어도 그 언어가 생겨난 사회의 문화를 반영하기 때문에 위와 같이 다른 쓰임새를 가질 때가 있다는 말입니다. 동적 언어와 정적 언어를 선호하는 분들의 논쟁을 보면, 서로 만날 수 없는 평행선을 보는 것 같은 기분이 들 때가 있습니다. 하지만, 종종 C#같은 언어를 주력으로 쓰면서도, python같은 언어를 이용해서 스크립트 작업을 매우 간편하게 처리하는 사람들도 보게 됩니다. 저 역시 바탕이 C#같은 정적 언어이지만, 기존 버전으로 작업하면서 C# 4.0의 동적 프로그래밍을 활용할 수 있다면, 매우 간단하게 해결될 작업을 만나곤 합니다. 즉, 정적 언어는 여전히 가장 많이 쓰이고 유용한 방법이지만, 만능이 아니라는 겁니다. 그래서 동적 언어의 장점을 활용해서 매우 간단하게 문제를 해결할 수 있는 부분이 존재하고, 그 걸 위해서 이번 C# 4.0에 동적 프로그래밍을 지원하는 부분이 추가되었다고 볼 수 있습니다.

그리고 동적 언어와 정적 언어의 선호도는 사고방식에서 갈린다고 볼 수도 있습니다. 동적 언어를 선호하는 분들은 정적 언어의 타입안정성이 실제로는 제대로 안정성을 보장하지 못한다고 이야기 합니다. 그리고, 타입시스템에 기대기 보다 철저한 유닛테스팅을 통한 검증을 더 중요하게 생각합니다. 그래서일까요, 대부분의 동적 언어를 옹호하시는 분들은 유닛테스트를 무척 강조합니다. 동적 언어를 썼을 때 생길 거라고 하는 문제들은 테스팅을 철저히 해서 다 걸러낼 수 있다는 것이죠. 어떻게 보면 유닛테스트는 테크닉이라기 보다는 문화적인 측면이 강합니다. 개발 프로세스 속에 녹아 들어야 원활하게 이루어질 수 있다는 말인데요, 그래서 이런 차이점도 기호를 명확하게 구분하는 요인이 되는 것 같습니다.

그럼 이제 C# 4.0에서 동적 프로그래밍이 어떤 모습이고, 그 지원 메커니즘은 구체적으로 어떤 모습인지 하나씩 알아보도록 하겠습니다.

1.2 Dynamic 타입의 추가.

우선, 아래의 코드를 보시죠.

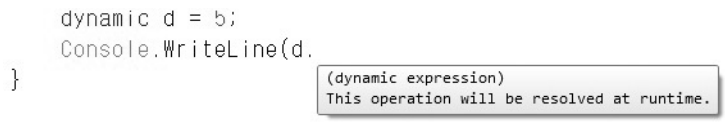
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    public class HelloDynamic
    {
        static void Main(string[] args)
        {
            dynamic d = 5;
            Console.WriteLine(d.ToString());

            d = "abcde";
            Console.WriteLine(d.ToString());
        }
    }
}
```

〈코드 13〉

dynamic이라는 타입을 처음 보시는 분들도 있을 텐데요, 우선 dynamic은 어떤 타입 이든지 될 수 있는 object같은 거라고 생각을 하시면 될 거 같습니다. 이 코드를 실행하면 다음과 같은 결과가 나옵니다.

〈그림3〉

그리고 dynamic대신에 object타입으로 d의 타입을 바꿔봐도 결과는 동일합니다. 위에서 말씀 드렸듯이 object같은 거라고 생각하라고 말씀 드린 이유가 여기에 있습니다. 하지만, 만약 비주얼스튜디오 2010에서 위의 코드를 치시다 보면, 조금 독특한 부분을 보셨을 겁니다. 아마도 "d.ToString()"이 부분을 타이핑 하실 때 였을 텐데요. 아래와 같았을 겁니다.



〈그림4〉

즉, 이 부분은 동적인 표현식이니깐, 이 코드가 직접 실행되는 런타임에서야 이게 무슨 의미인지 확인된다는 말입니다. 그럼 이 부분에 대해서 설명을 더 드리기 전에 아래 코드를 하나 더 보시죠.

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

namespace ConsoleApplication1
{
    public class FirstDynamicClass
    {
        public void Hello()
        {
            Console.WriteLine("Hello Mr.Dynamic!");
        }
    }

    public class HelloDynamic
    {
        static void Main(string[] args)
        {
            dynamic fdc = new FirstDynamicClass();
            fdc.Hello();
        }
    }
}
```

〈코드 14〉

결과는 역시 예측할 수 있듯이 아래와 같습니다.

〈그림5〉

그러면, 위에서 했듯이 dynamic을 object로 바꿔볼까요? 그러면, 아래와 같은 에러가 발생합니다.

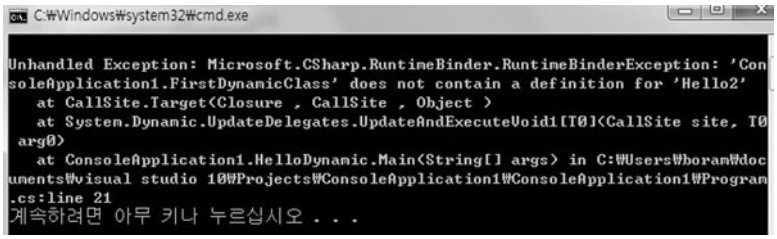
〈그림6〉

즉, object타입에는 Hello라는 메서드의 정의나 첫 번째 인자로 object를 받는 확장메서드가 없다는 말입니다. 이렇듯이, dynamic과 object는 매우 비슷하게 생겼지만 결정적인 차이가 존재하는 것이죠. 그 차이점은 바로 〈그림 4〉에서 볼 수 있습니다. 즉, 1.1절에서 설명 드렸던 덕 타이핑을 지원하느냐 못하느냐의 차이점인 거죠.

즉, object타입은 정적 타입이기 때문에 타입의 메타데이터에서 Hello라는 메서드를 찾습니다. 그런데, 〈그림 6〉의 에러메세지에서 보듯이 object타입에는 그런 메서드가 정의 되어있지 않았고, 확장 메서드 역시 찾을 수 없었습니다. 즉, object타입의 메타데이터에서 그런 정보를 찾을 수 없었다는 것이죠. 그래서 컴파일러는 메타데이터에서 찾을 수 없는 메서드를 호출하므로, 컴파일 에러를 내고 컴파일 조차 시켜주지 않는 것이죠.

반면에, 동적 타입인 dynamic은 덕 타이핑을 활용합니다. 즉, 메타데이터에서 찾을 수 있냐 없냐가 아니라, 실제로 특정타입이 특정 메서드를 가지고 있느냐 없느냐 하는 것입니다. 〈코드 14〉에서 보시듯이, 실제로 fdc라는 이름과 묶이는 타입은 FirstDynamicClass타입입니다. 그리고 그 타입에는 Hello라는 메서드가 존재합니다. 즉, 덕 타이핑의 "~~가 ~~를 할 수 있느냐"하는 테스트를 통과 하는 것이죠. 그래서 〈코드 14〉의 코드는 무사히 실행되는 것입니다.

즉, 동적 타입은 컴파일 타임에서 메타데이터를 가지지 않습니다. 그래서 '.'을 찍었을 때, 인텔리센스가 나오지 않는 것이죠. 인텔리센스는 해당 타입의 메타데이터를 활용해서 그 타입의 메서드나 변수 프로퍼티 등을 보여주는데, 메타데이터가 없으므로 보여줄게 없는 셈이죠. 그리고 마찬가지고, 특정 호출이 적절한 호출인지 아닌지도 판단할 수 없습니다. 그래서 컴파일 타임에서 아무런 에러도 발생하지 않고, 만약에 적절하지 못한 호출이 있다면, 런타임에 가서야 에러가 발생하는 것이죠. 〈코드 14〉에서 Hello가 아니라 Hello2메서드를 호출한다면 아래와 같은 예외가 발생하는 걸 볼 수 있습니다.



〈그림7〉

위와 같이 RuntimeBinderException이 발생하면서, FirstDynamicClass타입에 Hello2의 정의가 없다는 에러가 뜨게 됩니다.

그러면, 내부적으로는 어떤 점이 dynamic과 object의 차이를 만들어 낼까요? 〈코드 14〉의 FirstDynamicClass에 아래와 같은 메서드를 추가한 뒤에 리플렉터에서 한번 확인 해보겠습니다.

```
public dynamic Hello2(dynamic d)
{
    return 5;
}
```

〈코드 15〉

그러면, 각각 FirstDynamicClass와 HelloDynamic은 리플렉터에서 아래와 같이 나타납니다.

```
public class FirstDynamicClass
{
    // Methods
    public void Hello()
    {
        Console.WriteLine("Hello Mr.Dynamic!");
    }

    [return: Dynamic]
    public object Hello2([Dynamic] object d)
    {
        return 5;
    }
}

public class HelloDynamic
{
    // Methods
    private static void Main(string[] args)
    {
        object fdc = new FirstDynamicClass();

(1)     if (<Main>o__SiteContainer0.<>p__Site1 == null)
        {
            <Main>o__SiteContainer0.<>p__Site1 =
                CallSite<Action<CallSite, object>>
                  .Create(Binder.InvokeMember(
                       CSharpBinderFlags.ResultDiscarded,
                       "Hello",
                       null,
                       typeof(HelloDynamic),
                       new CSharpArgumentInfo[]
                          { CSharpArgumentInfo.Create(
                               CSharpArgumentInfoFlags.None, null)
                          }
                  ));
        }

(2)     <Main>o__SiteContainer0.<>p__Site1
            .Target.Invoke(<Main>o__SiteContainer0.<>p__Site1, fdc);
    }

    // Nested Types
    [CompilerGenerated]
    private static class <Main>o__SiteContainer0
    {
        // Fields
        public static CallSite<Action<CallSite, object>> <>p__Site1;
    }
}
```

〈코드 16〉

이 코드에서 주목하실 부분은, dynamic은 온데간데 없어지고 dynamic이 있던 자리에 object가 버티고 있다는 점입니다. 즉, dynamic과 object는 유사한 점만 가지고 있는 점은 아닌 것 같네요. 그런데, object를 사용할 때와는 매우 다른 코드들이 보입니다. 우선 FirstDynamicClass의 Hello2메서드를 보시죠.

이 메서드는 dynamic을 하나 받고, dynamic을 리턴 하는 메서드인데요. 그 dynamic은 모두 object로 바뀌어 있지만, 메서드에 [return: Dynamic], 그리고 파라미터에 [Dynamic]같이 어트리뷰트가 붙어있습니다. 즉, 런타임에게 이 object는 dynamic으로 간주하고 처리해달라고 부탁을 하는 것 처럼 보입니다. 여기에서 유추해볼 수 있는 사실은, dynamic은 실제로 존재하는 정적 타입이긴 하지만, BCL에 실제로 존재하는 타입은 아니며, 이렇게 표시를 함으로써 dynamic에 맞는 처리를 해준다는 것입니다. 그런데, Main메서드를 보면, object에 어떠한 표시도 되어있지 않습니다. 그렇다면, Hello2와 같이 외부의 메서드를 호출하는 경우에는 어트리뷰트를 붙여서 런타임에게 알려주는데, 그냥 Main메서드에서 처럼 사용하는 경우는 어떻게 런타임에게 알려주는 걸까요? Main메서드 안의 코드를 보시면, 뭔가 굉장히 긴 코드가 적혀 있는 걸 보실 수 있습니다. 〈그림 4〉의 Main메서드를 보시면, 그저 아래와 같이 딱 두 줄인데 말이죠.

```
dynamic fdc = new FirstDynamicClass();
fdc.Hello();
```

〈코드 17〉

즉, dynamic을 처리하는 메커니즘을 사용하기 위한 코드를 생성해서 그를 통해서 런타임이 동적으로 처리하도록 한다고 생각할 수 있습니다. 실제로 C# 4.0에서는 dynamic을 처리하기 위해서 DLR의 힘을 빌립니다. DLR은 Dynamic Language Runtime의 약자로서, IronRuby나 IronPython과 같은 동적 언어를 닷넷에서 사용할 수 있도록 해주는 런타임입니다. 즉, 이 DLR의 힘을 빌려서 동적 프로그래밍을 C#에서 구현하는 것이죠.

C# 컴파일러는 dynamic과 관련된 구문을 만나면, 우선 그 자리에 동적으로 call site를 생성합니다. 위의 HelloDynamic클래스의 끝부분을 보면, [CompilerGenerated]어트리뷰트와 함께 '〈Main〉o__SiteContainer0'라는 이름이 독특한 클래스가 생성되어 있는 걸 볼 수 있습니다. 그리고 그 클래스 안에, '〈〉p__Site1'라는 정적 변수가 하나 선언되어 있고 그 변수의 타입은 'CallSite〈Action〈CallSite, object〉〉'입니다. 여기서 컴파일러가 call site를 정의했음을 확인 할 수 있습니다. 그리고 이 call site를 dynamic과 관계된 호출에서 생성하고, 생성된 call site를 통해서 동적인 호출을 실제로 수행합니다. 바로 〈코드 16〉에서 (1)로 표시된 부분이 call site를 생성하는 부분이고, (2)로 표시된 부분이 생성된 call site를 통해 동적인 호출을 실행하는 부분입니다.

그리고 call site를 생성하는 부분을 자세히 들여다 보면, Binder.InvokeMember를 통해서 동적인 멤버호출을 생성하고 있는 게 보입니다.이 Binder의 내부를 들여다 보면,

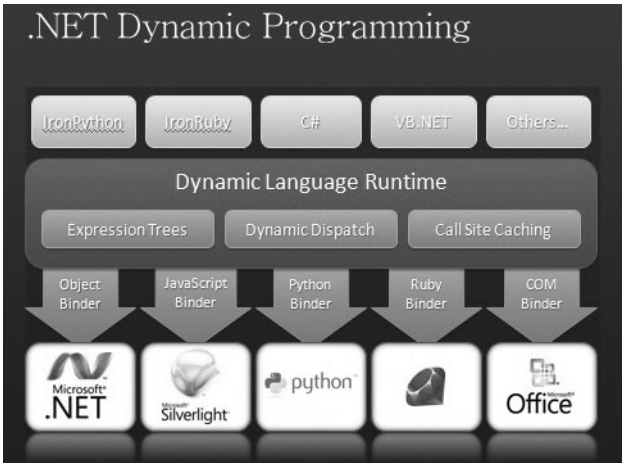
```
[EditorBrowsable(EditorBrowsableState.Never)]
public static class Binder
{
    // Methods
    public static CallSiteBinder BinaryOperation(...);
    public static CallSiteBinder Convert(...);
    public static CallSiteBinder GetIndex(...);
    public static CallSiteBinder GetMember(...);
```

```
    public static CallSiteBinder Invoke(...);
    public static CallSiteBinder InvokeConstructor(...);
    public static CallSiteBinder InvokeMember(CSharpBinderFlags flags, string name,
IEnumerable<Type> typeArguments, Type context,
IEnumerable<CSharpArgumentInfo> argumentInfo);
    public static CallSiteBinder IsEvent(...);
    public static CallSiteBinder SetIndex(...);
    public static CallSiteBinder SetMember(...);
    public static CallSiteBinder UnaryOperation(...);
}
```

〈코드 18〉

불필요하게 긴 부분은 모두 ...으로 줄이고 InvokeMember만 그대로 놓아두었습니다. 즉, call site를 적절한 호출로 연결해주는 중간 역할을 한다고 볼 수 있을 것 같습니다. MSDN에서는 네임스페이스에 대해서 "The Microsoft.CSharp.RuntimeBinder namespace provides classes and interfaces that support interoperation between Dynamic Language Runtime and C#.(C#과 DLR간의 상호운용을 지원하는 클래스와 인터페이스를 포함한다)"(http://msdn.microsoft.com/en-us/library/microsoft.csharp.runtimebinder(VS.100).aspx)정도로 설명하고 있으며, Binder클래스에 대해서도 "Infrastructure. Contains factory methods to create dynamic call site binders for CSharp(C#을 위한 동적 call site 바인더를 생성하는 팩토리 메서드를 가진 하부구조 클래스)."(위와 동일한 출처)정도로 설명을 하고 있습니다.

즉, 동적인 호출을 DLR이 어떻게 처리해야 하는지 Binder를 통해서 정의하고, 그 정의를 포함한 call site를 생성해서 DLR이 그 호출을 제대로 처리할 수 있도록 지원해 주는 것입니다. 이 바인더는 각각의 언어별로 존재하는 데요. PDC2008의 앤더스 헬스버그의 PPT를 참고하시면 아래와 같습니다.



〈그림8〉 앤더스 헬스버그의 "The Future of C#"중에서.

위 그림을 보시면, DLR의 핵심 요소로 3가지를 보실 수 있는데요. Dynamic Dispatch는 동적인 연산을 해당언어에 대해서 적절하게 바인딩 하는 부분을 말합니다. 그리고 call site 캐싱은 call site를 활용해서, 동적인 호출을 빠르게 실행할 수 있도록 캐시 메커니즘을 활용하는 것이구요. 마지막으로 Expression Trees는 기존의 C# 3.0에서 등장했던 Expression Trees의 확장판입니다. dynamic에 대해서 조금 더 설명드린후에, 이 3가지 요소에 대해서 하나씩 설명을 드려보고자 합니다.

일단 dynamic이 어떤 부분에서 쓰일 수 있으며, 그 의미는 어떻게 되는지 설명 드리겠습니다. 우선, 아래의 코드를 보시죠.

```
dynamic d = ...;

d.Foo(1, 2, 3); // (1)

d.Prop = 10; // (2)

var x = d + 10; // (3)

int y = d; // (4)

string y = (string)d; // (5)

Console.WriteLine(d); // (6)
```

〈코드 19〉

(d.Foo(); 에서 d는 Foo의 실행요청을 받는 receiver(수신자)이고, d의 타입이 dynamic이라면 d는 dynamic receiver(동적 수신자)가 됩니다.)

1. Foo메서드의 호출요청을 받은 객체의 타입이 dynamic이므로 컴파일러는 런타임에게 이 코드에서 d의 실제 런타임 시점의 타입이 뭐든지에 상관없이 "Foo"라는 메서드를 매개변수{1, 2, 3}를 적용해서 바인드해야 한다는 걸 알려줍니다.

2. 역시 1번과 마찬가지로 동적 수신자가 있으므로 컴파일러는 런타임에게 이 코드에서는 "Prop"이라는 프로퍼티같은(필드나 프로퍼티)걸 바인드해야 하고 거기에 10이라는 값을 할당해야 한다고 알려줍니다.

3. 여기서는 +연산자는 동적으로 바인드되는 연산인데요, 매개변수 중에 하나가 dynamic이기 때문이죠. 런타임은 실제 d의 런타임 시점의 타입에 대해서 일반적인 연산자 오버로딩 규칙을 따라서 적합한 연산을 찾습니다.

4. 여기서는 암시적인 형변환이 있는데요, 컴파일러는 int와 d의 실제 런타임 타입에 대한 모든 형변환을 고려해본 뒤에 d에서 int로의 형변환이 가능한지 판단하도록 런타임에게 알려줍니다. (사실은 암시적인 형변환이 수행되는 것은 아닙니다. dynamic에서 dynamic이나 object를 제외한 다른 타입으로는 암시적인 형변환이 불가능합니다. 다만, 동적 표현식은 암시적인 형변환이 가능한데요. 나중에 더 자세하게 다룰 기회가 있을 것 같습니다.)

5. 이번에는 명시적인 형변환 인데요, 컴파일러는 이 변환을 컴파일하고 런타임에게 이 형변환에대해서 검토해보도록 알려줍니다.

6. 비록 컴파일타임에서 볼 수 있는 메서드를 호출하지만, 인자가 dynamic이므로 컴파일타임에서는 오버로딩 판별을 할 수 없습니다. 그래서 어떤 Console.WriteLine을 호출할지도 역시 런타임에 결정하게 됩니다.

이중에 3번과 6번을 보면, 동적 수신자가 없더라도, 연산에 dynamic이 끼어들면 그 결과 역시 dynamic이 되므로 해당 구문은 동적으로 처리되어야 한다는 걸 보여줍니다. 실제로 아래와 같은 코드

를 보면,

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

namespace ConsoleApplication1
{
    public class HelloDynamic
    {
        static void Main(string[] args)
        {
            dynamic d = "abcd";

            Console.WriteLine(d.ToString());
        }
    }
}
```

〈코드 20〉

위 코드에서 d.ToString()에 대한 call site가 생기리라는 건 예측할 수 있습니다. d의 타입이 dynamic이기 때문이죠. 그런데, 위 코드를 리플렉터에서 보면 아래와 같습니다.

```
public class HelloDynamic
{
    // Methods
    private static void Main(string[] args)
    {
        object d = "abcd";

(3)      if (<Main>o__SiteContainer0.<>p__Site1 == null)
        {
            <Main>o__SiteContainer0.<>p__Site1 =
              CallSite<Action<CallSite, Type, object>>.Create(
                      Binder.InvokeMember(
                              CSharpBinderFlags.ResultDiscarded,
(1)                           "WriteLine", null, typeof(HelloDynamic), new
                          //이하 생략.........

(4)      if (<Main>o__SiteContainer0.<>p__Site2 == null)
        {
            <Main>o__SiteContainer0.<>p__Site2 =
              CallSite<Func<CallSite, object, object>>.Create(
                      Binder.InvokeMember(
                              CSharpBinderFlags.None,
                              "ToString",
                              null,
                              typeof(HelloDynamic),
                              new CSharpArgumentInfo[]
                                      { CSharpArgumentInfo.Create(
                                              CSharpArgumentInfoFlags.None,
                                              null)
                                      }
                      ));
        }

        <Main>o__SiteContainer0.<>p__Site1.Target
              .Invoke(<Main>o__SiteContainer0.<>p__Site1,
```

```
(2)             typeof(Console),
                <Main>o__SiteContainer0.<>p__Site2.Target
                    .Invoke(<Main>o__SiteContainer0.<>p__Site2, d)
            );
        }

        // Nested Types
        [CompilerGenerated]
        private static class <Main>o__SiteContainer0
        {
            // Fields
            public static CallSite<Action<CallSite, Type, object>> <>p__Site1;
            public static CallSite<Func<CallSite, object, object>> <>p__Site2;
        }
    }
```

〈코드 21〉

컴파일러가 생성한 〈Main〉o__SiteContainer0클래스를 보면, call site가 두개 생성되었고, 〈〉p_Site2가 생성되는 곳을 보면, 이 call site는 (1)과 (2)에서 강조된 것 처럼, Console.WriteLine에 대한 것임을 확인할 수 있습니다. 즉, dynamic이 들어간 표현식은 그 결과의 타입도 dynamic이 된다는 걸 알 수 있습니다.

1.2.1 바인딩 과정

```
dynamic d = ......;
d.Foo(1, 2, d);
```

〈코드 22〉

〈코드 22〉와 같은 코드가 실행될 때, 대략 아래와 같은 절차를 따르게 됩니다.

1. DLR이 이 호출 이전에, 동일한 매개변수 타입(int, int, dynamic)과 동일한 액션(InvokeMember)으로 생성된 call site가 캐시에 저장되어 있는지 확인합니다. 캐시에 저장이 되어 있다면, 캐시되어있던 걸 리턴합니다.

2. 캐시에 해당되는 내용이 없다면, DLR은 호출을 받는 객체가 DynamicObject(이하 DO)인지 확인합니다. 이 객체는 스스로 어떻게 동적으로 바인딩하는지를 알고 있는 객체입니다.(COM IDispatch object, 루비나 파이썬의 객체나 DynamicObject를 상속한 닷넷 객체들 처럼). 만약 DO라면 DLR은 DO에게 해당 액션을 바인딩 해달라고 요청합니다. DO에게 바인딩을 요청한 뒤에 받는 결과는 바인딩의 결과를 나타내주는 expression tree입니다.

3. DO가 아니라면, DLR은 language binder(C#의 경우는 C# 런타임 바인더)에게 해당 액션을 바인딩해줄 것을 요청합니다. 그러면 C# 런타임 바인더가 그 액션을 바인딩하고 바인딩의 결과를 expression tree로 리턴해줍니다.

4. 2번이나 3번이 수행된 다음엔 결과로 받은 expression tree가 DLR의 캐시 속으로 통합되고 같은 형태의 요청이 다시 들어온다면 바인딩을 위한 절차를 수행하지 않고 캐시에 저장된 결과를 가지고 실행하게 됩니다.

그리고 이어서 C# runtime binder에 대해서 조금 소개해 드리면요, C# runtime binder는 무엇을 어디에 바인딩할지를 결정하는 심볼테이블을 reflection을 이용해서 생성합니다. 만약에 컴파일타임에 타입이 정해진 매개변수라면 C# 액션(InvokeMember같은)의 정보에 해당 타입으로 기록되고 런타임 시에 바인딩될때 그 매개변수는 이미 기록된 타입으로 사용될 수 있겠죠. 그런데 만약에 컴파일 타임에 dynamic으로 결정된 매개변수라면(dynamic타입의 변수나, dynamic을 리턴 하는 표현식), 런타임 바인더는 reflection을 이용해서 그 매개변수의 타입을 알아내고, 알아낸 타입을 그 매개변수의 타입으로 사용하게 됩니다.

런타임 바인더는 심볼테이블을 필요할 때 필요한 만큼 만드는데요, 위의 〈코드 22〉의 경우는 Foo라는 메서드를 호출하는데요, 런타임 바인더는 d의 실제 런타임 타입에 대해서 Foo라는 이름을 가진 멤버들을 모두 로드합니다. 그리고 런타임 바인더는 C# 컴파일러가 하는 것과 동일한 오버로딩 판별알고리즘을 수행합니다. 그리고 이 과정에서는 컴파일 시에 받는 것과 동일한 문법을 사용하며, 동일한 에러, 동일한 예외를 출력합니다. 그리고 마지막으로 이런 과정을 거친 결과로 해당 호출을 어떤 메서드와 바인딩할지 심볼테이블에 기록하고, 그 바인딩된 결과를 expression tree로 생성하고 DLR에게 리턴 해줍니다. 여기서 사용하는 expression tree는 위에서 말씀 드렸듯이 C# 3.0의 expression tree를 확장한 것입니다. 기존의 expression tree에 동적인 연산, 변수, 이름 바인딩, 흐름제어를 위한 노드 들이 추가된 거죠. 그러면, 이 expression tree에 대해서 말씀 드리겠습니다.

1.3 Expression Trees

우선, 기존의 C# 3.0에 있었던 Expression Trees에 대해서 잠깐 설명을 드리도록 하겠습니다. 사실 C# 3.0으로 개발을 해왔다고 하더라도, Expression Trees에 대해서는 잘 모르는 경우가 많기 때문입니다. 저 역시도 그랬었구요.

일단 Expression trees는 실행 가능한 코드를 데이터로 변환하는 방법을 제공해줍니다. 이렇게 데이터로 변환하게 되면 컴파일하기 전에 코드를 변경한다거나 하는 일이 매우 수월해지는데요. 예를 들면, LINQ to SQL에서는 C#코드를 SQL쿼리 문으로 변환을 해야 합니다. 이 경우에, 컴파일 된 IL코드를 쿼리로 변경하는 게 효율적일까요? 아니면, 코드를 어떤 데이터구조로 생성한 다음에 그 데이터구조에서 쿼리로 변경하는 게 효율적일까요? 당연히 후자가 훨씬 효율적입니다. 그래서 C#에서 각종 LINQ식을 SQL서버와 같이 독립된 프로세스에서 수행하는 경우를 위해서, 코드를 쉽게 다루고 변환할 수 있도록 제공되는 데이터 구조가 바로 Expression Trees입니다. 그럼 Expression Trees의 간단한 예를 들어보겠습니다.

```
Func<int, int, int> function = (a, b) => a + b;
```

〈코드 23〉

〈코드 23〉와 같은 문장은 곤충이 머리, 가슴, 배로 나뉘듯이 세 부분으로 구성됩니다.

1. 선언부 : Func〈int, int, int〉 function
2. 등호 연산자 : =
3. 람다 식 : (a, b) => a + b;

변수 function은 두 숫자를 받아서 어떻게 더하는지를 나타내는 코드를 참조하고 있습니다. 그리고 위의 람다 표현식을 메서드로 표현해본다면 대략 아래와 같은 모양이 되겠죠.

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

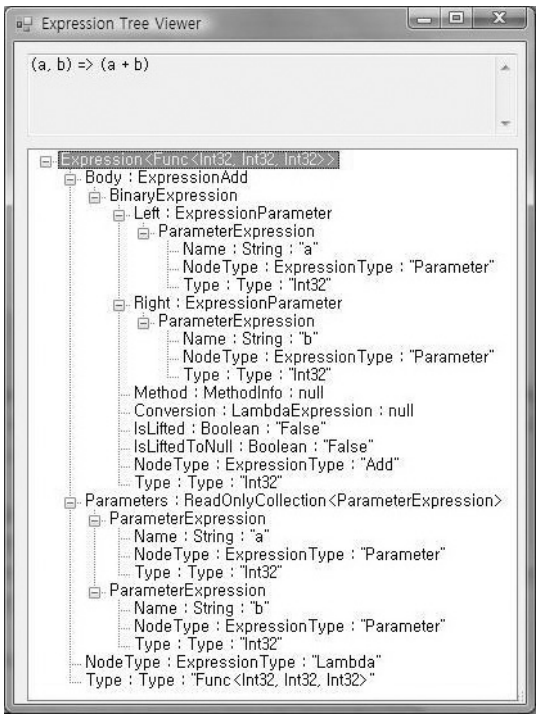
〈코드 24〉

그런데, Expression Trees는 자료구조라고 했었습니다. 그런데, 위의 람다 식은 메서드이므로 직접 실행되는 코드입니다. 그럼, 어떻게 코드를 Expression Trees로 구성할 수 있을까요? 매우 간단한 방법으로 그렇게 할 수 있습니다. 아래의 코드를 보시죠.

```
using System.Linq.Expressions;

....

Expression<Func<int, int, int>> expression = (a, b) => a + b;
```

〈코드 25〉

위와 같이 작성해주면, expression이라는 변수에 "(a, b) => a + b"라는 람다 식이 Expression Trees로 구성되어서 저장되는 것이죠. 그럼, 람다 식이 어떻게 Expression Trees로 생성된 모양이 어떤지 확인해보기 위해서 ExpressionTreeVisualizer(이하 ETV)를 사용해보도록 하겠습니다.
(ETV는 http://code.msdn.microsoft.com/csharpsamples에서 받을 수 있습니다.)



〈그림9〉

〈코드 25〉의 람다 식을 입력해보면, 〈그림 9〉과 같은 결과가 나옵니다. 위 그림에서 최상위 노드인 Expression의 하위 노드를 보면, Body, Parameters, NodeType, Type등 4개를 확인할 수 있습니다. 그 4개의 항목에 대해서 설명을 드리면 아래와 같습니다.

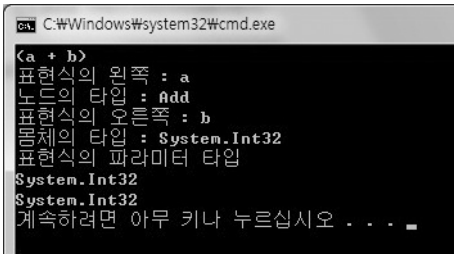
1. Body : expression의 몸체를 리턴한다.
2. Parameters : 람다식의 파라미터를 리턴한다.
3. NodeType : Expression trees의 특정노드의 ExpressionType을 리턴한다. ExpressionType은 45가지의 값을 가진 열거형타입인데, Expression trees에 속할 수 있는 모든 노드의 목록이 포함되어 있다. 예를 들면, 상수를 리턴하거나, 파라미터를 리턴한다거나, 둘 중에 뭐가 더 큰지 결정한다거나(〈,〉), 두 값을 더한다거나(+) 하는 것들이 있다.
4. Type : expression의 정적인 타입을 리턴한다. 위의 예제 같은 경우에는 Func〈int, int, int〉이다.

그럼, 예제를 통해서 위 4가지 값들을 한번 확인해 보겠습니다.

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

namespace ConsoleApplication1
{
    public class HelloDynamic
    {
        static void Main(string[] args)
        {
            Expression<Func<int, int, int>> expr = (a, b) => a + b;

            BinaryExpression body = (BinaryExpression)expr.Body;
            ParameterExpression left = (ParameterExpression)body.Left;
            ParameterExpression right = (ParameterExpression)body.Right;

            Console.WriteLine(expr.Body);
            Console.WriteLine("표현식의 왼쪽 : {0}\n노드의 타입 : {1}"
                + "\n표현식의 오른쪽 : {2}\n몸체의 타입 : {3}",
                left.Name, body.NodeType, right.Name, body.Type);

            Console.WriteLine("표현식의 파라미터 타입");
            foreach (var param in expr.Parameters)
            {
                Console.WriteLine(param.Type);
            }
        }
    }
}
```

〈코드 26〉

그리고 결과는 아래와 같습니다.



〈그림 10〉

1.3.1 Expression Trees v2

아직까지는 왜 DLR에서 Expression Trees를 개량해서, 코드를 구성하는 공통 자료구조로 사용하게 되었는지 명확하지가 않습니다. IronPython을 시작으로 DLR구현을 맡은 Jim Hugunin이 이 부분에 대해서 이야기를 했는데요, 그 사정을 이제부터 말씀 드리겠습니다.

Jim Hugunin에 따르면 애초에 DLR을 설계할 때 코드를 표현할 공통자료구조로 트리형태를 생각하고 있었다고 합니다. 그리고 타입이 없으면서 런타임에 late-bound되는 노드들로 구성된 트리로 구현을 시작했다고 합니다. 그래서 그 당시에 첫 DLR의 언어였던 IronPython에 맞게 구현된 트리에는 모든 노드가 타입이 없었으며, 타입이 있는 노드는 ConstantExpression딱 하나였다고 합니다. 상수 값이면 뭐든지 가지고 있는 노드인거죠. 그리고 자신들이 생각하고 있는 트리와 Expression Trees 사이에는 공통점이 없다고 생각했었다고 합니다. 왜냐면 정적 언어에서 쓰는 자료구조가 동적 언어에는 안 맞을 거라고 생각했기 때문이라는 군요. 그래서 독립적으로 트리를 개발해 나갔다고 합니다.

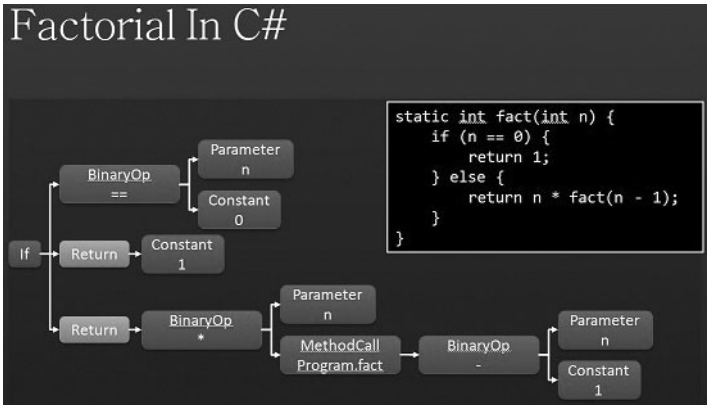
하지만, 다른 언어의 구현을 추가하면서, 기존의 DLR트리에 각 언어의 특징을 제대로 표현하기 힘들어 졌다고 하는군요. 각 언어의 특징을 지원하기 위한 방법이 필요한데, 그렇다고 해서 직접적으로 그 언어의 특징을 위한 노드를 추가할 수 는 없었다고 합니다. 아마도 DLR트리는 DLR에서 도는 언어들이 공통적으로 공유하게 되는 자료구조인데, 특정언어에만 있는 노드를 추가하면, 점점 지저분해 지기 때문이겠죠.

예를 들면, Python의 print문을 보면, Python의 print문은 출력에 대한 걸 어떻게 처리할지에 대해 알고 있는 static 메서드를 호출한다고 합니다. 그래서 DLR트리에 static메서드 호출을 위한 노드를 추가했다고 합니다. 그러면 print호출은 static 메서드를 호출하는 걸로 연결할 수 있으니까요.

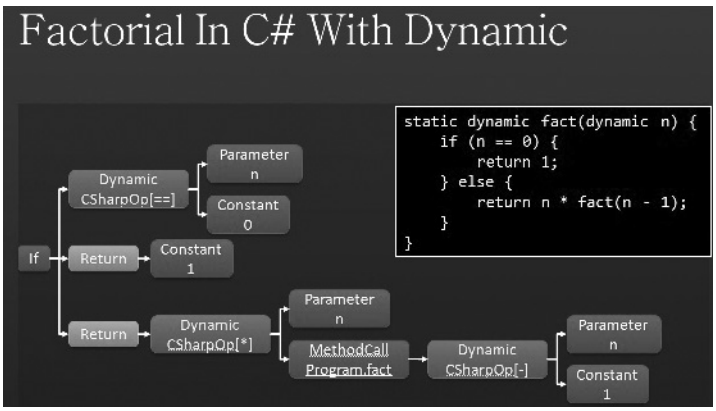
그리고 이런 작업을 하다 보니, 그 동안 개발팀이 추가해오던 노드들이 정적인 타입이 적용되는 Expression Trees에 있는 기존의 노드들과 딱 들어맞는 다는 걸 깨달았다고 합니다. 그래서 처음부터 다시 만드는 대신에 기존에 잘 쓰고 있는 걸 확장하자고 마음을 먹었고, 기존의 Expression Trees에 동적인 연산이나 변수, 이름 바인딩, 흐름제어 등을 위한 노드를 추가했다고 합니다.

그리고 DLR에서 동작하는 언어가 이런 공통된 트리형태를 사용하게 되면서 각각의 언어에 맞는 최적화를 수행하는 게 수월해졌다고 합니다. 기존의 Expression Trees를 사용하게 되면서 얻는 장점과 매우 동일한 점입니다. 그래서 각 언어의 컴파일러는 코드를 Expression Trees의 형태로 만들어서 그 트리를 DLR에게 전해준다고 합니다.

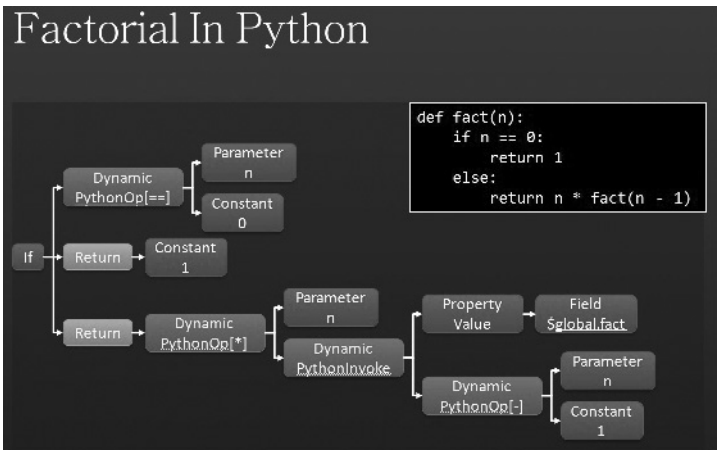
Jim Hugunin이 PDC2008에서 강연한 자료를 통해서 각 언어별로 나타나는 Expression Trees의 모양을 간단하게 보도록 하겠습니다.(아래의, 〈그림 11〉, 〈그림 12〉, 〈그림 13〉의 출처는 모두 해당 강연에 사용된 "Dynamic Languages In Microsoft .NET" ppt파일)



〈그림 11〉C# 팩토리얼을 구현한 Expression Trees



〈그림 12〉Dynamic을 사용해서 구현한 Expression Trees



〈그림 13〉Python의 Expression Trees

이렇게 Expression Trees v2는 DLR에서 매우 중요한 위치를 차지하고 있고, DLR과의 협력을 위해서 C#역시 Expression Trees v2를 핵심적으로 사용하고 있습니다.

1.4 바인딩 더 깊게 보기

이제부터는 dynamic과 관련된 각종 구문과 표현식 등이 어떻게 바인딩 되는지, 그 규칙을 좀 더 자세하게 알아보겠습니다.

```
dynamic d = 10;
C c = new C();

//위쪽 그룹
d.foo();
d.SomeProp = 10;
d[10] = 10;

//아래쪽 그룹
c.Foo(d);
C.StaticMethod(d);
c.SomeProp = d;
```

〈코드 27〉

위 그룹과 아래 그룹의 차이점은 뭘까요? 네~! Give that man a cigar!(누가 정답을 말했을 때 하는 말입니다) 위 그룹은 연산을 요청 받는 객체가 동적인 객체, 즉 동적 수신자이구요. 아래 그룹은 정적 수신자와 정적 메서드인게 바로 차이점입니다.

위 그룹은 동적인 표현식(expression)속에서 바로 동적인 행위가 일어나고, 아래그룹의 표현식은 직접적으로 동적인 표현식이 아닙니다. 각각의 연산의 매개변수로 dynamic타입이 들어가면서, 전체 표현식을 간접적이며 동적으로 만들고 있는 거죠. 이런 경우에는 컴파일러가 동적인 바인딩과 정적인 바인딩을 섞어서 수행하는데요. 예를 들어서 dynamic타입을 매개변수로 받는 오버로드가 있을 경우에(예 : void Foo(dynamic d)), 어떤 멤버집합(member set)을 오버로드 해야 할지 결정할 때(예 : 이름이 Foo에 파라미터가 한 개인 멤버집합)는 정적인 타입을 사용해서 판단할 테고, 실제로 오버로드를 판별(resolution)할 때는 매개변수의 런타임 타입을 사용할 것이기 때문이죠.

컴파일러가 dynamic타입인 표현식을 보게 되면, 그 안에 포함된 연산들을 동적 연산처럼 처리하게 됩니다. 즉, 표현식이 인덱스를 통한 접근이든 메서드 호출이든 상관없이 그 표현식의 결과로 나오는 타입은 런타임에 결정될 거라는 것입니다. 그 결과로 컴파일 타임에 동적인 표현식의 결과로 나오는 타입은 dynamic이겠죠.

이미 설명 드렸듯이 컴파일러는 이런 모든 동적인 연산들을 DLR을 통해서 dynamic call site라는 걸로 변환을 합니다. call site는 제네릭한 델리게이트를 가지고 있는 정적 필드입니다. 어떤 연산에 대한 호출을 가지고 있다가, 추후에 같은 타입의 연산이 호출되면 다시 call site를 생성할 필요 없이 정적 필드에 저장된 델리게이트를 호출해서 실행에 필요한 부하를 최대한 줄이는데 도움을 줍니다. call site가 만들어지면, 컴파일러는 그 call site에 저장된 델리게이트를 호출할 코드를 생성하구요, 거기에다가 매개변수를 넘겨줍니다.

만약에, 호출한 객체가 DynamicObject를 상속해서 스스로 동적 연산을 어떻게 처리할지 아는 객체가 아니거나, 미리 저장된 델리게이트와 타입이 안 맞아서 캐시가 불발이 나면, call site와 같이 생성된 CallSiteBinder가 호출됩니다. CallSiteBinder는 call site에 필요한 바인딩을 어떻게 처리해야 하는지 알고 있는 객체인데요, C#은 이 CallSiteBinder에서 상속한 바인더를 갖고 있습니다. 이 C# CallSiteBinder가 적절한 바인딩을 통해서 DLR의 call site에 저장될 내용을 Expression trees형태로 만들어서 리턴합니다.

1.4.1 동적 수신자의 경우와 캐시

공개된 문서를 통해 볼 수 있는 현재의 캐시 방식은 그냥 단순히 매개변수들의 타입이 일치하는지 검사하는 겁니다.

```
args0.M(arg1, arg2, ...);
```

〈코드 28〉

위와 같은 코드가 있고, 이전에 args0의 타입이 C이고, 매개변수 arg1과 arg2가 모두 int인 호출이 있었다고 해보면요, 캐시를 체크하는 코드는 대략 아래와 같습니다.

```
if (args0.GetType() == typeof(C) &&
    arg1.GetType() == typeof(int) &&
    arg2.GetType() == typeof(int) &&
    ...
    )
{
    //CallSiteBinder의 바인드 결과는 여기에 계속 통합되구요
}
    ......//캐시 검사는 좀 더 많을 수도 있구요
else
{
    //여기서 CallSiteBinder의 bind 메서드를 호출하고, 캐시를 업데이트 합니다.
}
```

〈코드 29〉

지금까지 간단하게 알아본 내용을 마무리 하려면, C# CallSiteBinder가 뭘 어떻게 하는지를 알면 되겠네요. 〈코드 27〉의 두 그룹의 연산 중에 위 그룹의 연산을 보면요, 메서드 호출, 속성 접근, 인덱서 호출 등 3가지 연산이 있습니다. 일단 모든 연산은 표준 C# 런타임 바인더를 통해서 C# playload라는 객체가 생성되고, C# 런타임 바인더가 그 payload를 데이터 객체로 사용합니다. payload는 바운드 되야 할 액션에 대한 정보를 갖고 있다고 합니다.

C# 런타임 바인더는 쉽게 작은 컴파일러라고 생각하면 되는데요, 일반적인 컴파일러가 갖고 있는 심볼 테이블이나 타입시스템, 오버로드 판별 및 타입 교체 같은 기능을 갖고 있기 때문입니다. 간단하게 d.Foo(1)을 예로 생각 해보겠습니다..

런타임 바인더가 호출되면, 현재 call site에 대한 payload와 call site에 대한 런타임 시점의 매개변수를 갖습니다. 그리고 동적 수신자를 포함해서 그 모든 런타임 매개변수와 타입을 모아서 그 타입에 대한 심볼테이블을 만듭니다. 그리고는 payload꾸러미를 풀어헤쳐서 수행하려고 하는 연산의 이름을 꺼냅니다.(Foo) 그리고 d의 실제 런타임 타입에서 리플렉션을 사용해서 Foo라는 이름을 갖는 모든 멤버를 뽑아냅니다. 그리고 그 멤버들을 모두 심볼테이블에 적어 넣죠. 말로 설명하기가 좀 부자연스러운데요, 상상력을 동원해서 설명을 드려보겠습니다.

1. d.Foo(1)에서 먼저 매개변수의 타입과 d의 타입을 갖고 와서 심볼테이블에 적어두고요.(주소는 메모리 주소가 들어가야 하지만, 그냥 적었습니다.)

주소	타입	이름
서울시	Int	익명(=1)
수원시	Dynamic	D

〈표 2〉

2. 그리고 리플렉션으로 d의 타입에서 Foo를 모두 찾아냈는데 대략 아래와 같다고 해보죠.

```
Foo(int a)
Foo(double b)
Foo(string c)
```

〈코드 30〉

3. 그리고 얘네들도 따로 심볼테이블에 집어넣으면?

-d의 멤버중에 Foo라는 동명이인들

주소	타입	이름
부산시	void	Foo(int a)
창원시	void	Foo(double b)
안양시	void	Foo(string c)

〈표 3〉

그러면, 타입을 찬찬히 들여다보면, 어떤 Foo가 호출되야 할지 명확하게 보입니다. d.Foo(1)호출에서 매개변수의 런타임타입이 int이므로 Foo(int a)가 호출이 되겠죠. 이건 제가 임의로 간단한 심볼테이블을 이용해서 설명 드리는 겁니다. 실제로는 더 복잡하겠죠.

이런 런타임 바인더를 설계할 때 세웠던 한가지 원칙은 "런타임 바인더는 정적 컴파일러가 하는 짓을 똑같은 의미로 할 수 있어야 한다."였다고 하는데요. 그래서 에러 메세지 역시 동일한 에러 메세지를 뱉어낸다고 합니다.

위의 바인딩의 결과로 바인딩이 성공적일 경우에 수행할 동작을 표현한 Expression trees가 만들어집니다. 그렇게 안 되는 경우에는 runtime binder exception을 던진다고 하네요. 결과로 만들어진 Expression trees는 DLR의 캐시에 포함되고 호출되면서 원래의 호출을 성공적으로 완료합니다.

1.4.2 정적 수신자의 경우

그러면 이제, 수신자가 정적 수신자인 경우, 메서드 오버로딩이 어떻게 이루어 지는지에 대해서 설명 드려보겠습니다.

```
public class C
{
    public void Foo(decimal x) { ... }
    public void Foo(string x) { ... }
```

```
    static void Main(string[] args)
    {
        C c = new C();
        dynamic d = 10;
        c.Foo(d);
    }
}
```

〈코드 31〉

이 코드가 실행되면 어떤 일이 벌어질까요? 직관적으로 생각해봤을 때, 지역변수c의 타입이 C라는 걸 알 수 있으니깐, C의 오버로드 2개중에 하나가 실행될 거라고 생각해볼 수 있습니다. 근데, d의 타입이 dynamic이라는 것도 위 소스코드에서 알 수 있는데요, 그렇다면 d의 실제 타입은 런타임에서 알 수 있으므로 컴파일러는 어떤 오버로드를 호출해야 하는지 판단할 수가 없습니다.

그래서 이렇게 생각해볼 수 있습니다. 컴파일러는 호출 가능한 후보 군을 추출해서 집합을 만들고, 런타임에 실제로 오버로드 판별을 통해서 적합한 메서드를 호출한다고 말이죠. 위의 경우에는 d의 값이 10이므로, 호환이 안 되는 string보다는 decimal을 인자로 받는 오버로드가 호출될 거라고 예측해볼 수 있습니다. 그러면, 좀 더 구체적으로 이야기 해보면서 다른 경우에는 뭐가 예측이랑 다르게 돌아가는지에 대해서 이야기 해보겠습니다.

```
public class C
{
    public void Foo(decimal x) { ... }
    public void Foo(string x) { ... }
    static void Main(string[] args)
    {
        C c = new D();
        dynamic d = 10;
        c.Foo(d);
    }
}
public class D : C
{
    public void Foo(int x) {...}
}
```

〈코드 32〉

클래스 D를 C로 부터 상속했고, c를 D의 생성자로 생성한 게 차이점입니다. 런타임에 c의 타입은 D일 테고, D에는 C가 가진 오버로드보다 이 경우에 더 적합한 int형 파라미터를 가진 Foo가 있습니다. 10은 int형으로 간주될 테니, D의 Foo가 가장 적합한 선택이 되겠죠.

그런데, 결과는 조금 다릅니다. 위 코드를 작성하다 보면, 아래와 같은 상황을 보게 됩니다.

```
    static void Main(string[] args)
    {
        C c = new D();
        dynamic d = 10;
        c.foo
    }
}

public class D : C
{
    public void Foo(int x) { Console.WriteLine("Foo(int x)"); }
}
```


〈그림 14〉

즉, 분명히 오버로드는 D의 Foo를 포함해서 총 3개여야 할 텐데, 두개 밖에 안 나옵니다. 그래서 분명히 D의 Foo가 가장 좋은 선택임에도 불구하고 계속해서 C의 Foo(decimal x)가 호출이 됩니다. 하지만, c를 생성할 때 "D c = new D();"나 "dynamic c = new D()"처럼 생성하면, D의 Foo(int x)가 호출이 됩니다. 왜 그럴까요?

dynamic과 관련된 동적 바인딩을 설계할 때, 최대한 동적 바인딩이 기존의 컴파일러가 정적 바인딩에서 하던 것과 비슷하게 유지했다고 합니다. 그래서 그 결과로 dynamic이라고 명시된 매개변수나 수신자가 아니라면, 컴파일타임에 확인할 수 있는 타입을 그 변수의 타입으로 간주한다고 합니다. 즉, 위의 예제에서 c.Foo의 오버로드로 D.Foo(int x)가 포함이 안된 이유는, "C c = new D();"의 결과로 c가 런타임에 가질 타입은 D겠지만, 컴파일타임에서는 C라고 간주한다는 겁니다. 그래서 아무리 인텔리센스를 뒤져봐도 D의 Foo를 발견할 수 없으며 호출도 할 수 없는 거죠.

하지만, "D c = new D();"나 "dynamic c = new D()"처럼 생성하면, 전자의 경우는 컴파일 타임의 c의 타입을 D로 간주하므로 오버로드 3개 모두를 인텔리센스에서 확인하실 수 있구요, 후자의 경우는 동적 바인딩을 통해서 c의 런타임 타입이 D임을 알기 때문에, 오버로드 후보 군에서 Foo(int x)를 골라낼 수 있습니다. 즉, dynamic이라고 해도 기존의 룰을 최대한 존중한다는 것이죠. 이로 이런 점이 C#이 발전해 나가면서도 고수하는 점이라고 볼 수 있습니다.

1.5 Dynamic과 관련된 오버로딩과 형변환

예전에 C#팀의 멤버들이 작성한 문서를 보면, 유령 메서드라는 개념이 있었습니다. 컴파일러가 초기 바인딩인 단계에서 정적 바인딩으로 해결할 수 없는 동적 바인딩을 해야 하는 경우에 사용하는 메서드를 말하는 것이었는데요. 실체는 없지만 컴파일러 내부에서 문제해결을 위해서 사용한다고 볼 수 있습니다.

그런데 개발 도중에 그 유령메서드라는 개념과 과정이 너무 복잡하다고 해서, 단순화하는 작업을 겨쳤고, 그 결과 유령 메서드라는 개념은 살아남지 못했습니다. 그래서 원래 이 문서에도 유령메서드에 대한 언급이 있었지만, 수정을 거쳐야 했지요. 바뀐 규칙은 매우 간단하게 정리됩니다.

1.5.1 오버로드 판별

만약에 어떤 메서드를 호출할 때 Dynamic타입의 매개변수가 끼어있다면, 그 메서드에 대한 호출은 동적으로 디스패치됩니다. 그리고 런타임에서 Dynamic타입의 매개변수의 실제 값에 맞는 타입을 선택하는 것이죠. 예제를 하나 보겠습니다.

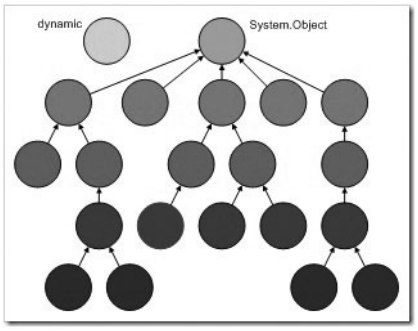
```
class C
{
    static void M(dynamic d) { Console.WriteLine("dynamic"); }
    static void M(string s) { Console.WriteLine("string"); }
    static void M(int i) { Console.WriteLine("int"); }

    static void Main(string[] args)
    {
        dynamic d = "test";
        M(d);
    }
}
```

〈코드 33〉

예제에 있는 M에 대한 호출은 런타임에 디스패치 됩니다. 그리고 M의 오버로드 중에서 어떤 메서드가 호출에 디스패치될지는 d의 실제 값에 달려있는데요, 여기서는 d의 실제 값이 String타입이기 때문에, 오버로드 메서드 중에서 'M(string)'이 선택됩니다. 그렇다면 M(dynamic)은 어떨 때 선택되는 걸까요? 파라미터에 Dynamic타입이 있는 건, Object타입과 동일한 의미를 갖습니다. 즉, 적합한 오버로드가 하나도 없을 때, 가장 마지막에 선택이 되는 것이죠.

1.5.2 Dynamic과 다른 타입간의 형변환

우선, Dynamic과 다른 타입간의 관계를 먼저 살펴보면 아래와 같습니다.



〈그림 15〉 출처 : http://blogs.msdn.com/cburrows/archive/2008/11/06/c-dynamic-part-iv.aspx

다른 타입들은 System.Object에서 시작해서 이리저리 연결되어 있지만, Dynamic은 따로 혼자 떨어져 있습니다. 이런 관계속에서 Dynamic의 형변환 룰에 대해서 알아보면 아래와 같습니다.

1. 모든 타입에서 Dynamic타입으로 암시적 형변환이 가능하다. 기본적으로 Object로 암시적 형변환이 가능한 타입이라면, Dynamic으로도 암시적 형변환이 가능하다.
2. Dynamic타입에서는 Dynamic과 Object를 제외한 다른 어떤 타입으로도 암시적 형변환이 불가능하다.
3. 하지만, 모든 '동적 표현식'은 다른 모든 타입으로 암시적 형변환이 가능하다.
4. 만약에 어떤 타입의 차이점이 Dynamic과 Object뿐이라면, 서로간에 암시적 형변환이 가능하다.

〈표 4〉 Dynamic에서 다른 타입으로 형변환 가능 여부

즉 Dynamic에서 다른 타입으로 명시적인 형변환은 가능하지만, 암시적인 형변환은 불가능 하다는 이야기입니다. 하지만 3번 을 보시면, '동적 표현식'은 암시적 형변환이 가능하다고 하는데요. 무슨 의미일까요?

```
string s = d;
```

〈코드 34〉

위와 같은 예제를 보면요, 이 형변환은 3번 규칙 때문에 성립합니다. 이 형변환은 동적 표현식에서 String타입으로의 형변환이지, Dynamic타입에서 String타입으로의 형변환이 아니기 때문이죠. 이 외에도 Return, Foreach, 프로퍼티에 값 설정하기 등의 대부분의 경우는 동일한 규칙이 적용됩니다.

그렇다면 반대로 Dynamic타입에서 String타입으로 형변환이 없어야 하는 이유는 뭘까요? 공변성(Covariance)를 통해서 컴파일러가 이 형변환을 막는 걸 확인해보겠습니다.

```
class C
{
    static void Main()
    {
        IEnumerable<dynamic> ied = null;
        IEnumerable<string> iei = null;
        var x = M(ied, iei);
    }
    static T M<T>(IEnumerable<T> x, IEnumerable<T> y) { return default(T); }
}
```

〈코드 35〉

위 예제에서 T의 타입은 Dynamic으로 결정됩니다. 그렇게 결정되는 메서드의 타입유추 과정을 살펴보면 아래와 같습니다.

1. 첫 번째 매개변수는 IEnumerable〈Dynamic〉이다. 그러므로 Dynamic은 T의 후보가 된다.
2. 두 번째 매개변수는 IEnumerable〈String〉이다. 그러므로 String은 T의 후보가 된다.
3. T의 후보군인 { Dynamic, String }을 가지고 고민하는 과정에서 둘 사이의 관계를 보는데, String에서 Dynamic으로의 형변환은 있지만, 그 반대는 성립하지 않는다. 그러므로 좀 더 일반적인 타입인 Dynamic을 고르게 되는 것이다.

〈표 5〉

위 과정의 3번 단계에서 만약 Dynamic에서 String으로 형변환이 가능한 상황이었다면, 둘 중에 어떤 타입이 더 일반적인 타입인지를 결정할 수 없게 됩니다. 그렇다면 메서드 타입유추는 둘의 모호한 관계 때문에 실패하게 되겠죠. 이런 이유때문에 Dynamic에서 다른 타입으로의 형변환이 금지되어야 하는 것이죠.

1.5.3 프로퍼티

d.Foo를 예로 들면, d는 Dynamic객체이고, Foo는 d속에 살고 있는 멤버 변수나 프로퍼티입니다. 컴파일러가 이런 구문을 만나면, 우선 Foo라는 이름을 Payload속에다가 기록합니다. 그리고 런타임에게 d의 실제 타입을 찾아서 바인드(연결) 해달라고 요청합니다.

그리고 이런 프로퍼티는 항상 3가지 경우 중 한가지경우에서 쓰이는데요. 값을 읽어오거나, 값을 대입하거나, 둘 다 하거나(+=같이)같은 경우입니다. 컴파일러가 사용된 모양을 보고, 어떻게 사용하려고 하는지도 Payload에 같이 기록합니다. 즉, 읽기만 하는 경우에는 해당 프로퍼티는 읽기전용으로 기록을 하는 식으로 말입니다.

그리고 컴파일러는 이런 접근이 필드에 접근하는 건지, 프로퍼티에 접근하는 건지 특별히 구분하지 않습니다. 그건 나중에, 런타임이 구분을 하게 됩니다. 그리고 컴파일 할 때, 이런 구문의 리턴 타입은 Dynamic으로 설정됩니다.

1.5.4 인덱서

인덱서는 두 가지로 생각해볼 수 있습니다. 첫 번째는 매개변수가 있는 프로퍼티, 두 번째는 배열이나 리스트 같은 집합의 이름을 통한 메서드 호출이 그 것인데요. Dynamic과 연관 지어서 생각할 때는 후자가 훨씬 도움이 됩니다. 메서드의 경우와 같이 인덱서도 정적으로 바운드 될 수 있지만, Dynamic 타입의 매개변수가 주어지고, 그 매개변수가 Dynamic타입을 받는 인덱서로 정적 바운드가 되지 않는 경우, 어떤 인덱서가 호출될지는 런타임에 가서 결정됩니다. 위에서 설명 드렸던 규칙대로, 런타임에 매개변수의 실제 값을 가지고 어떤 인덱서의 오버로드가 호출될지를 결정하는 것이죠. 예전에는 이런 과정에 유령메서드가 끼어들어서 복잡했지만, 무척이나 간단해진 셈이죠. 알기 쉽게 예제로 설명을 드려보겠습니다.

```
public class C
{
    public int this[int i]
    {
        get
        {
            Console.WriteLine("int i");
            return i;
        }
    }

    public int this[dynamic d]
    {
        get
        {
            Console.WriteLine("dynamic d");
            return d;
        }
    }
```

```
    static void Main(string[] args)
    {
        C c = new C();
        Console.WriteLine(c[5]);
        dynamic d = 7;
        Console.WriteLine(c[d]);
    }
}
```

〈코드 36〉

〈코드 36〉를 보시면, C에 인덱서가 두 개가 있습니다. 하나는 int를 매개변수로, 하나는 Dynamic을 매개변수로 받죠. 그리고 Main메서드 안에서 하나는 int를, 하나는 Dynamic타입의 매개변수를 넘겨주고 있습니다. 이 경우에 두 번째 인덱서는 언제 어떻게 바인드될까요? d가 Dynamic타입이기 때문에 위에서 설명 드렸던 대로 런타임에 동적으로 바인드됩니다. 그래서 d의 실제 값이 7을 받는 인덱서가 호출되는 것이죠. 그래서 출력되는 메세지를 확인해보면, 두 번다 'int i'가 출력되는 걸 확인할 수 있습니다.

1.5.5 형변환

위에서 설명을 드릴 때, dynamic은 다른 타입으로 암시적 형변환은 안되지만 되는 경우가 있다고 설명을 드렸었습니다. 바로 dynamic타입이 아니라 동적 표현식의 경우가 그랬었죠. 형변환의 경우는 payload가 매우 단순해집니다. 왜냐면, 컴파일러는 이미 어떤 타입으로 형변환을 하려고 하는지 알고 있기 때문이죠. 그래서 컴파일러는 그냥 payload에 형변환 하려고 하는 타입을 기록하고, 런타임 바인더에게 가능한 모든 형변환(형변환 연산자를 쓰는 경우에는 명시적 형변환도 같이)을 시도해보라고 이야기 해줍니다. 물론, dynamic타입이 아니라, 런타임에 결정될 실제 타입에서 목표 타입으로 시도해보겠죠.

형변환의 경우는 다른 모든 경우와 다르게 컴파일 하는 시점에서 dynamic이 아닌 형변환의 목표타입을 리턴합니다. 위에서 말씀 드렸듯이 이미 어떤 타입으로 형변환 하려고 하는지 알 수 있기 때문이죠.

1.5.6 연산자

연산자는 조금 다릅니다. 그냥 아무 생각 없이 훑어보면, 동적인 뭔가가 일어난다고 느끼기 힘들기 때문이죠. 그런데, d+1 같은 간단한 구문도 런타임에 바인드 되어야 합니다. 그 이유는 사용자정의 연산자가 끼어들 수 있기 때문입니다. 그래서, dynamic 매개변수를 갖는 모든 연산은 런타임에 바인드됩니다. +=나 -=같은 연산자도 포함해서 말이죠.

컴파일러는 연산자를 보면, d.Foo += 10 같이 멤버에 대입하는 연산이 있는지 혹은, d += 10 같이 변수에 대입하는 연산이 있는지 확인합니다. 그리고 그 과정에서 d를 ref를 통해 넘겨서 변경된 값이 유지되어야 하는지 확인합니다.

그리고 마지막으로 d.Foo += x 같은 구문이 있을 때, d.Foo가 바인드결과 delegate나 event타입이라면, 앞의 구문은 이벤트 수신자 추가 같은 적절한 메서드를 호출하도록 컴파일러가 연결해줍니다.

1.5.7 델리게이트 호출

델리게이트 호출은 메서드와 굉장히 유사합니다. 딱 한가지 틀린 점이 있다면, 호출되는 메서드의 이름이 명시되지 않는다는 것 뿐이죠. 그래서, 아래 예제의 두 호출은 모두 런타임에 바인드됩니다.

```
public class C
{
    static void Main(string[] args)
    {
        MyDel c = new MyDel();
        dynamic d = new MyDel();

        d();
        c(d);
    }
}
```

〈코드 37〉

첫 번째 호출은 매개변수가 없는 호출을 런타임에 바인드하게 됩니다. 런타임 바인더가 런타임에 호출의 수신자가 델리게이트 타입이 맞는지 확인하고 해당 델리게이트 시그니처와 일치하는 호출이 있는지 오버로드 판별을 통해서 찾게 됩니다.

두 번째 호출은 매개변수가 dynamic타입이기 때문에, 런타임에 바인드됩니다. 컴파일러가 컴파일시점에서 c의 타입이 델리게이트라는 걸 확인할 수 있지만, 실제 오버로드 판별은 런타임에 가서 끝나게 됩니다.

1.6 예제

이제 예제를 하나 만들어 보겠습니다. 예제에서는 앞에서 설명 드렸던, 스스로 동적인 연산을 어떻게 처리하는지 알고 있는 객체를 만들어 보겠습니다. 스스로 동적인 행동을 처리하는 객체를 만들려면, DynamicObject클래스를 상속해서 스스로 처리하기 원하는 기능을 오버라이드 해야 하는데요. 우선 DynamicObject클래스를 보시죠.

```
namespace System.Dynamic
{
    // Summary:
    //   객체가 런타임에 동적인 행위를 스스로 처리할 수 있도록 하는 간단한 클래스.
    public class DynamicObject : IDynamicMetaObjectProvider
    {

        protected DynamicObject();

        public virtual IEnumerable<string> GetDynamicMemberNames();

        public virtual DynamicMetaObject GetMetaObject(Expression parameter);

        public virtual bool TryBinaryOperation(BinaryOperationBinder binder, object arg,
out object result);

        public virtual bool TryConvert(ConvertBinder binder, out object result);

        public virtual bool TryCreateInstance(CreateInstanceBinder binder, object[]
args, out object result);
```

```
        public virtual bool TryDeleteIndex(DeleteIndexBinder binder, object[] indexes);

        public virtual bool TryDeleteMember(DeleteMemberBinder binder);

        public virtual bool TryGetIndex(GetIndexBinder binder, object[] indexes, out
object result);

        public virtual bool TryGetMember(GetMemberBinder binder, out object result);

        public virtual bool TryInvoke(InvokeBinder binder, object[] args, out object
result);

        public virtual bool TryInvokeMember(InvokeMemberBinder binder, object[] args,
out object result);

        public virtual bool TrySetIndex(SetIndexBinder binder, object[] indexes, object
value);

        public virtual bool TrySetMember(SetMemberBinder binder, object value);

        public virtual bool TryUnaryOperation(UnaryOperationBinder binder, out object
result);
    }
}
```

〈코드 38〉

형변환 부터, 인덱스, 멤버 접근 등 동적으로 처리할 수 있어야 하는 모든 행위가 정의되어 있습니다. 여기서 몇 가지만 오버라이드 해서 예제를 만들어 보겠습니다. 예제로 만들려고 하는 건 RSS리더인데요. RSS리더는 많은 분들이 아시겠지만, 아래와 같은 포맷을 읽어 들입니다.

```
<?xml version="1.0" encoding="utf-8" ?>

<rss version="2.0"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:taxo="http://purl.org/rss/1.0/modules/taxonomy/" >
  <channel>


    <title><![CDATA[wanna be의 소프트웨어 팩토리.]]></title>
    <link>http://blog.naver.com/netscout82</link>

    <image>

<url><![CDATA[http://blogfiles.naver.net/data32/2008/5/25/90/%C5%A9%B1%E2%BA%AF%C8%AF_im
g0142-netscout82.jpg]]></url>
      <title><![CDATA[wanna be의 소프트웨어 팩토리.]]></title>
      <link>http://blog.naver.com/netscout82</link>
    </image>


<description><![CDATA[&#54756;&#54756;&#54756;&#54756;&#54756;&#54756;&#54756;.]]></desc
ription>
    <language>ko</language>
    <generator>Naver Blog</generator>
    <pubDate>Thu, 19 Nov 2009 16:32:41 +0900</pubDate>


    <item>
      <author>netscout82</author>
      <category><![CDATA[낙서장 ]]></category>
```

```
      <title><![CDATA[그냥.....ㅋ]]></title>
      <link>http://blog.naver.com/netscout82/20093583105</link>
      <guid>http://blog.naver.com/netscout82/20093583105</guid>
      <description><![CDATA[&#3232;_&#3232;.... 되는게 없엉.]]></description>
      <pubDate>Mon, 16 Nov 2009 16:59:40 +0900</pubDate>
      <tag><![CDATA[]]></tag>
    </item>

.................................................
```

〈코드 39〉

우선, DynamicObject를 상속하는 부분부터 소스를 보시겠습니다.

```
using System.Collections.Generic;
using System.Linq;
using System.Dynamic;
using System.Xml.Linq;

namespace RssReader
{
    class DynamicRssReader : DynamicObject
    {
        XElement channel;

        public DynamicRssReader(string path)
        {
            XDocument doc = XDocument.Load(path);
            channel = doc.Descendants("channel").FirstOrDefault();
        }

        public DynamicRssReader(XElement node)
        {
            channel = node;
        }

        public override bool TryGetMember(GetMemberBinder binder, out object result)
        {
            result = null;

            try
            {
                IEnumerable<XElement> elements = channel.Elements(binder.Name);
                if (elements.Count() == 1)
                {
                    result = elements.FirstOrDefault().Value;
                }
                else
                {
                    result = elements;
                }

                return true;
            }
            catch
            {
                return false;
            }
        }

        public override bool TryInvokeMember(InvokeMemberBinder binder, object[] args,
out object result)
        {
```

```
            result = null;

            if (args.Length == 1)
            {
                result = channel.Elements(binder.Name).Take((int)args[0]);
                return true;
            }

            return false;
        }
    }
}
```

〈코드 40〉

위에서 보시듯이 TryGetMember와 TryInvokeMember를 오버라이드 했으므로, 이 클래스는 멤버에 접근해서 값을 가져오거나 멤버를 호출하는 부분을 동적으로 수행할 수 있다고 생각할 수 있습니다. 그리고 주목할 부분이 행위 결과를 'out'으로 선언된 변수에 할당하고, 성공이나 실패여부를 리턴 하는 것을 볼 수 있습니다. 즉, 객체가 스스로 동적인 연산을 처리할 줄 안다면, 요청이 성공했는지 실패했는지도 스스로 알려줄 수 있어야 한다는 것입니다. 위에서 작성한 클래스는 아래와 같이 사용할 수 있습니다.

```
using System;
using System.Xml.Linq;

namespace RssReader
{
    class Program
    {
        static void Main(string[] args)
        {
            dynamic rssReader = new DynamicRssReader("..\\..\\MyBlog.xml");

            Console.WriteLine("블로그 이름 : {0}", rssReader.title);
            Console.WriteLine("블로그 주소 : {0}", rssReader.link);
            Console.WriteLine("-----------------------------------------------\n");

            foreach (XElement item in rssReader.item(5))
            {
                dynamic itemReader = new DynamicRssReader(item);

                Console.WriteLine("카테고리 : {0}",itemReader.category);
                Console.WriteLine("제목 : {0}", itemReader.title);
                Console.WriteLine("글주소 : {0}", itemReader.link);
                Console.WriteLine("내용 : {0}", itemReader.description);
                Console.WriteLine("글쓴시간 : {0}", itemReader.pubDate);
                Console.WriteLine("-----------------------------------------------");
            }
        }
    }
}
```

〈코드 41〉

〈코드 41〉에서 보면, TryGetMember는 RSS의 각 item의 항목들을 가져오는데 쓰였고, TryInvokeMember는 특정항목에서 몇 개를 정해서 읽어오는 데 쓰였다는 걸 알 수 있습니다. 즉, 이런 식으로 동적으로 처리하는 객체를 만들어서 프로그래밍 하면 간결하고, 직관적으로 처리할 수 있습니다.

2. Co- Contravariance

covariance는 공변성, contravariance는 역공변성 정도로 변역 할 수 있지만, 무슨 말인지 제대로 감이 오는 단어는 아닌 것 같습니다. 왜 그럴까요? 사실 이 두 개념은 범주론(category theory)에서 이야기하는 용어이기 때문이죠. 그래서 많은 오해를 낳기도 하지요. 이번 장에서는 이 개념에 대해서 설명 드리겠습니다.

2.1 일반적인 Co- Contravariance의 개념

우선, 이 Co- Contravariance에 대한 Eric Lippert의 포스트를 인용해서, 일반적인 개념을 보기로 하겠습니다.

그러면 타입은 생각하지도 말고 정수에 대해서 한번 생각해봅시다. 정수위로 투영(projection)한다고 했을 때, 투영은 뭘 의미할까요? 투영은 하나의 정수를 취하고 또 하나의 새로운 정수를 반환하는 걸 말합니다. 즉, (z -> z + z)은 정수 하나를 받아서 그 정수를 두배 한 새로운 정수를 반환하는 투영이겠죠. 그리고 (z -> 0 - z)는 부호를 반대로 하는 투영, (z -> z * z)는 제곱을 하는 투영이겠죠.

그러면 여기서 질문입니다. 위의 투영 중에서 입력된 정수를 두 배로 하는 투영을 D라고 할 때, (x <= y) = (D(x) <= D(y)) 가 항상 성립할까요? 그렇습니다. x가 y보다 작거나 같다면, x를 두 배 한 수는 항상 y를 두배한 수보다 작거나 같습니다. 그리고 x가 y와 같다면, 당연히 두수의 두 배는 같겠죠. 즉, 투영 D는 크기의 방향성을 보존한다고 말할 수 있습니다. x < y 였다면, 투영 D를 수행한 후에요, D(x) < D(y)이기 때문이죠.

그러면, 부호를 반대로 하는 투영을 N이라고 할 때, 항상 (x <= y) = (N(x) <= N(y))일까요? 명백하게 아닙니다. 1 <= 2 이지만, 투영 N을 하고 나면 -1 <= -2 가 되어서 거짓이 되기 때문이죠. 즉, 투영 N의 경우는 원래 등호 식을 뒤집으면 항상 참인 것을 알 수 있습니다. 즉, (x <= y) = (N(y) <= N(x)) 인 것이죠. 투영 N은 크기의 방향성을 뒤집습니다.

그러면, 제곱을 하는 투영을 S라고 할 때, S는 어떨까요? -1 <= 0은 참이지만, S(-1) <= S(0)은 거짓입니다. 그리고 1 <= 2은 참이지만, S(2) <= S(1)은 거짓입니다. 즉, S는 크기의 방향성을 보존하지도 않고 뒤집지도 않는 것이죠.

이럴 때, 투영 D는 covariant하다고 말합니다. 정수간의 크기를 정의하는 관계는 보존하기 때문이죠. 그리고 투영 N은 contravariant합니다. 정수간의 크기 관계를 뒤집기 때문이죠. 그리고 투영 S는 어느 쪽도 아니므로, invariant하다고 말합니다.

그러면, 정수에 대한 생각을 접고, 참조형에 대해서 생각해보기로 합시다. 참조형에서는 크기 관계를 정의할 때, 'X의 타입이 Y타입의 변수에 저장될 수 있다면, X는 Y보다 작거나 같다'고 이야기 합니다.
(http://blogs.msdn.com/ericlippert/archive/2009/11/30/what-s-the-difference-between-covariance-and-assignment-compatibility.aspx)

이 정도면, 일반적인 개념에 대해서는 이해가 되셨으리라 생각합니다. 그러면, 타입 시스템에서 CO-Contravariance에 대해서 설명 드리겠습니다.

2.2 타입 시스템에서 Co- Contravariance의 개념

사실, 타입 시스템에서의 개념이라고 말씀을 드리지만, <=같은 크고 작음을 이야기하는 정의만 바뀔 뿐이지 기본적인 개념은 똑같습니다. 그러면, 일단 예제를 먼저 보여드리겠습니다.

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

namespace Co_ContraVariance
{
    class Animal
    {
        public bool Hungry { get; set; }

        public override string ToString()
        {
            return "Animal";
        }
    }

    class Mammal : Animal
    {
        public override string ToString()
        {
            return "Mammal";
        }
    }

    class Giraffe : Mammal
    {
        public override string ToString()
        {
            return "Giraffe";
        }
    }

    class Tiger : Mammal
    {
        public override string ToString()
        {
            return "Tiger";
        }
    }

    class Program
    {
        void FeedAnimals(IEnumerable<Animal> animals)
        {
            foreach (Animal animal in animals)
            {
                if (animal.Hungry)
                {
                    Feed(animal);
                }
            }
        }

        private void Feed(Animal animal)
        {
            Console.WriteLine(animal.ToString());
```

```
        }

        static void Main(string[] args)
        {
            List<Giraffe> giraffes = new List<Giraffe>{
                new Giraffe{ Hungry = true},
                new Giraffe{Hungry = true},
                new Giraffe{Hungry = false}
            };

            Program program = new Program();
            program.FeedAnimals(giraffes);
        }
    }
}
```

〈코드 42〉

동물에게 먹이를 주는 걸 표현해 본 프로그램입니다. 클래스를 보시면, Animal이 있고, Animal을 상속한 Mammal(포유류), 그리고 그 Mammal을 상속한 Giraffe(기린)과 Tiger입니다. 그런데, 위 코드를 vs2008에서 작성하다 보면, 아래와 같은 에러를 볼 수 있습니다.

〈그림 16〉

즉, 다른 타입이므로 메서드를 오버로드 하라는 이야기입니다. Giraffe는 Animal의 서브타입이므로 문제가 없습니다만, IEnumerable〈Giraffe〉는 IEnumerable〈Animal〉과 담고 있는 요소의 타입이 다를 뿐 어떤 관계도 없다고 보기 때문에 문제가 생기는 것입니다. 그래서 IEnumerable〈Giraffe〉에서 IEnumerable〈Animal〉로는 형변환이 안 되는 것이죠. 그래서 아래와 같은 코드를 통해 간접적으로 형변환을 해줘야 합니다.

program.FeedAnimals(giraffes.Cast〈Animal〉());

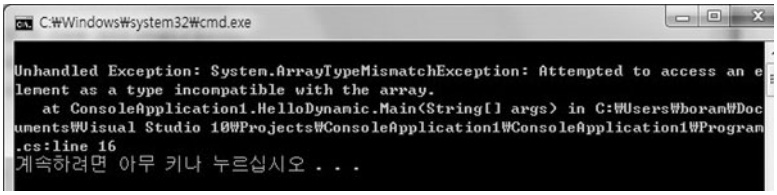
그런데, 왜 이런 경우가 위험한 걸까요? 아래의 예를 보시죠.

```
namespace ConsoleApplication1
{
    public class HelloDynamic
    {
        static void Main(string[] args)
        {
```

```
            string[] sa = new string[2];
            object[] oa = sa;
            oa[0] = "hello";
            oa[1] = 5;

            string s = (string)oa[1];
        }
    }
}
```

〈코드 43〉

위 코드는 vs2010에서도 문제없이 컴파일이 됩니다. 그런데, 실행을 해보면 아래와 같은 에러가 납니다.



〈그림 17〉

즉, 잘못된 타입을 접근하고 있다는 이야기 인데요. string배열을 object배열이 참조하고, object배열을 통해서 2번째 인덱스에 정수 값이 들어갔기 때문입니다. 현재의 배열에서는 잘못된 타입의 값을 대입하거나 접근하는 걸 막기 위해서 런타임에 동적인 타입체크를 생성해서 체크를 한다고 합니다. 즉, 컴파일 타임에는 아무 에러가 나지 않지만, 런타임에서 타입체크를 수행하니 잘 못되었다는 이야기지요. 앤더스 헬스버그도 채널9의 세션에서 이게 covariance를 구현하기 위한 배열 구현상의 단점이라고 이야기하고 있습니다. 그래서, 이런 단점을 없애기 위해서 IEnumerable에서는 이런 형변환이 금지 되어 있었습니다.

하지만, 이런 위험성이 전혀 없는 경우도 존재하는 데요, C# 4.0에서는 위험성이 없는 경우는 형변환을 허용해주겠다는 것입니다. 그럼 타입의 간의 관계를 생각해볼 때, 어떤 경우가 있는지 살펴보겠습니다.

우선 어떤 타입 T와 U가 있다고 했을 때, 다음 중 하나는 참이다.

1. T는 U보다 크다.
2. T는 U보다 작다.
3. T와 U는 같다.
4. T와 U는 서로 관련이 없다.

〈표 6〉

〈코드 42〉을 보시면, Giraffe는 Mammal의 서브타입이므로 Mammal보다 작습니다. 하지만, Giraffe는 Tiger와는 아무 관련이 없는 타입이죠. 이런 타입간의 관계에서 어떤 연산을 T와 U에 대해서 수행했을 때, 그 결과로 나온 T'와 U'가 T와 U의 관계를 그대로 유지한다면, 그 연산은 covariant하다고 이야기 할 수 있으며, 대입의 방향성과 크고 작음을 뒤집고 같음과 관계없음만 그대로 유지한다면(위 리스트에서 1,2번을 뒤집고 3,4번만 유지한다면) 이 연산은 contravariant하다고 이야기 할 수 있습니다.

그러면, 언제 위와 같은 경우가 있는지 먼저 IEnumerable을 통해서 알아보겠습니다.

```
interface IEnumerable<T>
{
     IEnumerator<T> GetEnumerator();
}

interface IEnumerator<T>
{
     bool MoveNext();
     T Current {get;}
}
```

〈코드 44〉

위 코드는 C# 3.0의 IEnumerable〈T〉의 간략한 정의입니다. 위 정의를 보시면, IEnumerable〈T〉안으로 값을 입력할 수 있는 방법은 전혀 없습니다. GetEnumerator메서드를 통해서 IEnumerator〈T〉를 얻어와도 사정은 마찬가지 입니다. 오로지 값을 읽어올 수 만 있지, 입력할 수 있는 방법은 없습니다. 즉, 〈코드 43〉와 같이 잘못된 타입의 값이 들어갈 위험성이 전혀 없다는 말이죠. 그래서 이런 경우에, covariance를 보장할 수 있게 됩니다. IEnumerable〈string〉을 IEnumerable〈object〉로 형변환을 한다고 해도, string과 object의 크기의 방향성은 그대로 유지된다는 말이죠. 그래서, 이런 covariance를 지원하기 위해서 out키워드가 추가되었습니다.

```
interface IEnumerator<out T>
{
     bool MoveNext();
     T Current {get;}
(1)  /*void Add(T x); 이렇게 T가 파라미터자리에 있으면 컴파일러가 불평한다. */
}

IEnumerable<object> x = stringlist; //이런 게 가능하다
```

〈코드 45〉

위와 같이, 언제나 크기의 방향성을 유지할 수 있는 경우 out키워드를 붙여주면, 컴파일러는 이 인터페이스에서 대해서 covariance한 형변환에 대해서 아무런 불평도 하지 않습니다. 단, 값이 들어갈 수 없다는 말은 메서드 등의 파라미터 자리에 out키워드로 선언된 T가 올 수 없다는 말입니다. 그래서 (1)과 같이 파라미터 자리에 T가 있으면, 컴파일러가 에러라고 불평을 하는 것입니다.

그러면, 반대로 contravariance의 경우를 보시죠. contravariance는 입력 받기만 하는 경우에 해당합니다.

```
public interface IComparer<T>
{
    int Compare(T x, T y);
}
```

〈코드 46〉

IComparer〈T〉를 보면, 값이 들어갈 수는 있는데, 그 값이 나올 수 있는 경우는 없습니다. 그런데, IComparer가 contravariance하다면, 기존의 크기의 방향성을 뒤집어야 하는데, 과연 그럴까요?

Animal은 Giraffe보다 더 큰 데 어떻게 IComparer〈Giraffe〉에 들어가는 걸까요? 예제를 하나 보여드리고 설명 드리겠습니다. 〈코드 42〉의 Animal관련 클래스를 그대로 사용하고, 새로 작성한 코드만 적었습니다.

```
class AnimalComparer : IComparer<Animal>
    {
        public int Compare(Animal x, Animal y)
        {
            return 1; //예제를 위해 단순화.
        }
    }

    class GiraffeComparer : IComparer<Giraffe>
    {
        public int Compare(Giraffe x, Giraffe y)
        {
            return 1; //예제를 위해 단순화.
        }
    }

.......중략.......

        static void Main(string[] args)
        {
            ........중략.......

            IComparer<Animal> animalComp = new AnimalComparer();
            IComparer<Giraffe> giraffeComp = animalComp;
            Console.WriteLine(giraffeComp.Compare(new Giraffe()
                                                  , new Giraffe()));
        }
```

〈코드 47〉

위 코드는 컴파일과 실행 역시 아무 문제가 없습니다. 그냥 상식적으로 생각했을 때 동물을 비교할 수 있다면, 기린(giraffe)을 비교할 수도 있겠죠? 사람들이 IComparer〈Giraffe〉에 비교하려고 넣는 객체는 모두 Giraffe타입이겠고 giraffeComp가 실제로 호출하게 되는 Compare메서드는 IComparer〈Animal〉을 구현한 animalComp의 메서드겠죠. 그래서 방향성이 뒤집혔지만, contravariance로 인해서 실행가능 한 코드가 됩니다.

반대로 IComparer〈Animal〉에다가 IComparer〈Giraffe〉를 대입하는 경우를 생각해보죠. 그렇다면 사람들은 animalComp를 통해서 Animal을 비교하고 싶어할 텐데, 실제로 호출될 메서드는 IComparer〈Giraffe〉의 Compare메서드 입니다. 그 메서드는 Giraffe타입을 받을 수 있으므로 Animal을 받을 수 없습니다. 그래서 이런 참조형 변환은 지원되지 않습니다.

그래서 위와 같은 경우 T에 in키워드를 붙이면, contravariance에 대해서는 컴파일러가 아무런 불평도 하지 않는 것입니다. 단, 이 경우에는 입력하는 경우만 가능해야 하므로, 리턴 타입에 in키워드가 붙은 T가 있다면, 컴파일러가 컴파일 에러를 내게 됩니다.

3. Named and Optional Parameters

이 기능은 파라미터가 길어지면, 그 순서를 맞추기 위해서 애쓰거나, 기존의 COM Interop에서 참으로 난해하게도 의미 없는 값을 전달하기 위해서 불필요한 코딩을 반복해야 했던 것과 같이 기본값으로 설정된 부분은 생략할 수도 있게 개선하기 위해서 추가되었습니다. 그리 긴 설명이 필요한 부분은 아니니, 예제를 통해서 알아보겠습니다.

```
namespace ConsoleApplication1
{
    public class HelloDynamic
    {
        public static int Add(int x, int y=8, int z=3)
        {
            return x + y + z;
        }

        static void Main(string[] args)
        {
            Console.WriteLine(Add(2, 6, 9));//2+6+9 (1)
            Console.WriteLine(Add(2, 6));//2+6+3 (2)
            Console.WriteLine(Add(2));//2+8+3 (3)
            Console.WriteLine(Add(2, z:6, y:9));//2+9+6 (4)
            Console.WriteLine(Add(y:2, x:9));//9+2+3 (5)
            Console.WriteLine(Add(9, z:1));//9+8+1 (6)

        }
    }
}
```

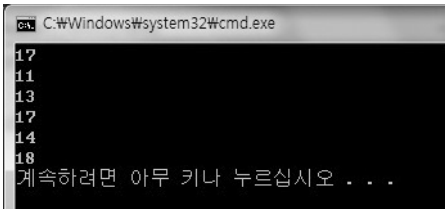
〈코드 48〉

Add메서드를 보시면, 조금 생소한 부분이 있습니다. 'int y=8, int z=3'이 그 부분인데요. 이건, "y와 z 파라미터에 아무런 값이 들어오지 않으면, 기본값으로 8과 3을 설정하겠다"하는 표시입니다. 그럼, 호출하는 부분을 한번 보죠.

(1)번은 아주 평범한 호출입니다. x, y, z에 각각 값을 직접 입력해줬으니까요. 그리고 (2)번은 보면, z의 값을 생략하고 x, y값만 넘겨주고 있습니다. 그래서 z는 기본값으로 설정된 3이 사용되었구요. (3)번을 보시면, x값만 넘겨주면서 y, z는 기본값이 사용되었습니다.

하지만, 한가지 중요한 점은 (x, y, z)와 같은 경우, y값만 생략할 수는 없다는 점입니다. 즉 Add(2,4)와 같이 호출을 했다면, 생략되는 값은 z이며, y값은 생략할 수 없다는 말입니다. 그러면, y값은 영영 생략할 수 없는 걸까요? 이런 경우를 위해서, 파라미터를 지정해서 값을 넘겨줄 수 있도록 하고 있습니다. 이렇게 이름을 지정하면, 파라미터의 순서를 바꿀 수도 있고, 중간의 파라미터만 생략할 수도 있습니다.

즉, (4)번을 보면, z와 y의 위치를 바꿨습니다. 파라미터가 길어질수록, 파라미터를 순서대로 입력하는 게 곤란해지고는 합니다. 그럴 때 아주 유용하게 쓰일 수 있겠죠. 그리고 (5)번을 보면, z를 생략하고 x와 y의 순서를 바꿨습니다. 그리고 (6)번을 보면, y를 생략한 채 x와 z값만 넘겨주고 있는 걸 볼 수 있습니다. 결과를 확인해보면 아래와 같습니다.



〈그림 18〉

그리고 Named and Optional Parameters가 COM프로그래밍에서 매우 효율적으로 쓰입니다. 이 부분에 대해서 PDC 2008에서 앤더스 헬스버그가 강연할 때 썼던 슬라이드 한 장을 보여드리겠습니다.

〈그림 19〉 출처 : 앤더스 헬스버그의 "The Future of C#"중에서.

즉, COM프로그래밍을 위해서 의미 없는 값을 반복해서 넣어야 했던 것에 비해서, 앞으로는 그림 아래쪽처럼 깔끔하게, 필요한 값만 넘겨주면 그걸로 끝나는 셈이죠.

마치면서

이렇게 퍼스트 룩 C# 4.0이 마무리 되었습니다. C# 4.0의 새로운 부분을 보면서 어떤 느낌이 드셨나요? 저 개인적으로는 매우 이해하기 어려운 부분도 많았습니다. 서로 다른 환경에서 나타난 언어는 그 언어가 발생한 환경을 제대로 이해해야만 제대로 언어의 철학과 쓰임새를 이해할 수 있지만, 저 역시 배경이 정적 언어이고, 실력의 부족으로 제대로 이해하지 못한 부분이 많기 때문입니다. 그리고 그 동안 다른 세계의 이야기라고 생각했던 동적 프로그래밍이 C#같은 메이저 언어에 통합되면서, 이제 우리들의 이야기가 되겠구나 하고 느꼈습니다.

새로운 변화가 있을 때마다, 참 적응하기가 쉽지 않은데요. 아마도 그건, 제가 미래를 제대로 내다볼만한 내공도 없고 시야도 좁기 때문이라고 생각합니다. 그래서 이런 변화가 너무도 갑작스럽게 느껴지기도 하는 것이죠. 하지만, 이제 변화는 이미 현실로 다가왔습니다. 이런 변화를 적절하게 소화해서 자신의 장기로 삼을 수도 있고, 개발 현장에서 매우 편리한 해결의 도구로 사용할 수도 있습니다. 그리고 거부감으로 손도 대지 않고 기존의 방식을 고수할 수도 있습니다. 신중한 것은 좋지만, 신중함이 지나쳐서 고집이 된다면 그 결과는 희망적이지 않을 것입니다. 이 퍼스트 룩 C# 4.0이 새로운 가능성을 받아들이고자 하는 분들에게 작은 도움이 되었으면 하고 바라면서, 글을 이만 줄이겠습니다.

ps.
참고자료를 더 보시고 싶으신 분들은, VSTS2010팀 블로그에서 제가 작성한 Welcome to C# 시리즈나 MSDN POPCON 블로그에 있는 웹 캐스트 'C#하나 알려주면 안 잡아먹지'시리즈(1: C#의 과거, 2 : 기술은 날 기다려주지 않는군.., 3 : 돈 나오면 10원에 10대씩.., 4 : 오리 얘기 한번만 더 ㅋ, 5 : 짜잘한 예제들, 6 : C#도 뱀처럼, 루비처럼, 7 : 예제 하나더..., 마지막 : 못다한 이야기), 그리고 Techday 2009의 'C# 4.0 with Dynamic'를 보실 수 있습니다. 그리고 가장 최근의 Techday 2010 Spring에서 진행한 Dynamic(for example, these four examples)도 보실 수 있습니다.

저자 : 강보람(a.k.a 워너비) Visual Studio 2010 팀 블로그 멤버

새로운 거라면 무조건 좋아하고 보는 청년이지만, 내공의 부족으로 늘 황새를 쫓는 뱁새의 기분이다. 우연의 일치인지, 실제로도 솟다리이다. TechDays Korea 2009, 2010에 참여했으며, VSTS 2010 팀 블로그에서 활동중이다.(www.vsts2010.net) 그리고 '워너비의 소프트웨어 팩토리'(http://blog.naver.com/netscout82)에서 소프트웨어와 전혀 상관없는 이야기를 더 많이 쓰고 있다.

참고자료

1. Hackers and Painters, Paul Graham, O'Reilly
2. Programming Ruby, 데이비드 토머스, 앤디 헌트, 차드파울러, 인사이트
3. http://code.msdn.microsoft.com/Project/Download/FileDownload.aspx?ProjectName=csharpfuture&DownloadId=3550
4. Pro LINQ: Language Integrated Query in C# 2008, Joseph C. Rattz, Jr. , APRESS.
5. http://blogs.msdn.com/ericlippert/archive/2007/10/16/covariance-and-contravariance-in-c-part-one.aspx
6. http://blogs.msdn.com/ericlippert/archive/2007/10/17/covariance-and-contravariance-in-c-part-two-array-covariance.aspx
7. http://blogs.msdn.com/ericlippert/archive/2007/10/19/covariance-and-contravariance-in-c-part-three-member-group-conversion-variance.aspx
8. http://blogs.msdn.com/ericlippert/archive/2007/10/22/covariance-and-contravariance-in-c-part-four-real-delegate-variance.aspx
9. http://blogs.msdn.com/ericlippert/archive/2007/10/26/covariance-and-contravariance-in-c-part-five-interface-variance.aspx
10. New features in C# 4.0, Mads Torgersen
11. http://blogs.msdn.com/samng/archive/2008/10/29/dynamic-in-c.aspx
12. http://blogs.msdn.com/cburrows/archive/2008/10/27/c-dynamic.aspx
13. http://channel9.msdn.com/pdc2008/TL10/
14. http://en.wikipedia.org/wiki/Symbol_table
15. http://blogs.msdn.com/charlie/archive/2008/01/31/expression-tree-basics.aspx
16. http://blogs.msdn.com/hugunin/archive/2007/05/15/dlr-trees-part-1.aspx
17. http://blogs.msdn.com/samng/archive/2008/11/02/dynamic-in-c-ii-basics.aspx
18. http://blogs.msdn.com/samng/archive/2008/11/06/dynamic-in-c-iii-a-slight-twist.aspx
19. http://blogs.msdn.com/samng/archive/2008/11/09/dynamic-in-c-iv-the-phantom-method.aspx
20. http://blogs.msdn.com/cburrows/archive/2008/11/06/c-dynamic-part-iv.aspx
21. http://blogs.msdn.com/cburrows/archive/2008/11/11/c-dynamic-part-v.aspx
22. http://blogs.msdn.com/samng/archive/2008/12/11/dynamic-in-c-v-indexers-operators-and-more.aspx
23. http://blogs.msdn.com/cburrows/archive/2010/04/01/errata-dynamic-conversions-and-overload-resolution.aspx

기타 참고자료는 인용문 등에 표시되어 있습니다.

Visual Studio 2010 에디션 별 기능 비교

디버깅과 진단				
IntelliTrace™ (Historical Debugger)	●			
Static Code 분석	●	●		
Code Metrics	●	●		
프로파일링	●	●		
테스팅				
Unit 테스팅	●	●	●	
Code Coverage	●	●		
Test Impact 분석	●	●		
Coded UI 테스트	●	●		
Web Performance 테스팅	●			
Load 테스팅[1]	●			
Microsoft® Test Manager 2010	●			●
Test Case 관리[2]	●			●
매뉴얼 테스트 실행	●			●
매뉴얼 테스트 Record 및 Playback	●			●
Lab Management Configuration[3]	●			●
데이터베이스 개발				
데이터베이스 구축	●	●		
데이터베이스 변동 관리[2]	●	●		
데이터베이스 Unit 테스팅	●	●		
데이터베이스 테스트 데이터 생성	●	●		
개발 플랫폼 지원				
Windows 개발	●	●	●	
Web 개발	●	●	●	
Office 및 SharePoint 개발	●	●	●	
Cloud 개발	●	●	●	
개발환경 사용자 지정	●	●	●	
아키텍쳐와 모델링				
Architecture Explorer	●			
UML® 2.0 Compliant 다이어그램 (Activity, Use Case, Sequence, Class, Component)	●			
Layer 다이어그램과 의존도 분석	●			
Read-only 다이어그램 (UML, Layer, DGML Graphs)		●		
Lab 관리				
가상환경 셋업 및 제거[3]	●			●
Provision 환경 from template[3]	●			●
Checkpoint 환경[3]	●			●
Team Foundation Server				
버전관리[2]	●	●	●	●
Work Item 추적[2]	●	●	●	●
빌드자동화[2]	●	●	●	●
Team 협업 포탈[2]	●	●	●	●
리포팅과 Business Intelligence[2]	●	●	●	●
Agile Planning 워크북[2]	●	●	●	●
테스트케이스 관리[2]	●	●	●	●
Microsoft® Visual Studio® Team Explorer 2010	●	●	●	●

MSDN Subscription
혜택 비교

MSDN Subscription 혜택 : 상용 Software 제공				
Microsoft® Visual Studio® Team Foundation Server 2010	●	●	●	●
Microsoft® Visual Studio®Team Foundation Server 2010 CAL	1	1	1	1
Microsoft® Expression Studio 3	●	●		
Microsoft Office Professional Plus 2010, Project Professional 2010, Visio Premium 2010	●	●		
MSDN Subscription 혜택 : 개발 및 테스트용 Software 제공				
Windows® Azure™	●†	●††	●†††	
Windows (모든 클라이언트와 서버 OS)	●	●	●	●
Microsoft® SQL Server®	●	●	●	●
Toolkits, Software Development Kits, Driver Development Kits	●	●	●	●
Microsoft® Office	●	●		
Microsoft Dynamics®	●	●		
기타 모든 서버 제품군	●	●		
Windows Embedded OS	●	●		
MSDN Subscription 혜택 : 기술 지원 및 리소스 제공				
유료 기술지원 인시던트	4개	4개	4개	2개
MSDN Forums 최우선 기술 지원	●	●	●	●
유료 Microsoft® e-learning collections	2 Collection	2 Collection	1 Collection	1 Collection
MSDN 매거진	●	●	●	●
MSDN Flash 뉴스레터	●	●	●	●
MSDN Online Concierge 기술 지원	●	●	●	●

† Azure 혜택 : 1개월에 250 compute hrs, 7.5GB storage, 3GB SQL Server 데이터베이스 용량, 1개월에 1M .NET 메시지

†† Azure 혜택 : 1개월에 100 compute hrs, 5GB storage, 2GB SQL Server 데이터베이스 용량, 1개월에 500k .NET 메시지

††† Azure 혜택 : 1개월에 50 compute hrs, 1개월에 3GB storage, 1GB SQL Server 데이터베이스 용량, 1개월에 300k .NET 메시지

1. 하나 또는 이상의 Microsoft® Visual Studio® Load Test Virtual User Pack 2010이 필요할 수 있습니다.
2. Team Foundation Server와 Team Foundation Server CAL이 필요합니다.
3. Microsoft® Visual Studio® Team Lab Management 2010이 필요합니다.
4. 사용자당 라이선스로서 애플리케이션 기획, 개발, 테스팅과 데모용으로 무제한 사용이 가능합니다.

UML is a registered trademark of Object Management Group, Inc.
Windows is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries.

Microsoft®

한국마이크로소프트(유)

서울특별시 강남구 대치동 892번지 포스코 센터 서관 5층 (우) 135-777
고객지원센터 : 1577-9700, 제품 홈페이지 : http://www.microsoft.com/visualstudio, 개발자 포털 : http://msdn.microsoft.com