C#의 역사

객체지향 윈도우즈 프로그래밍

2016112158 김희수

**C#의 시작**

마이크로소프트에서 일하던 Anders Hejlsberg와 그가 이끄는 팀이 1999년 1월에 C#을 만들었다. 그 당시 .NET 프레임워크와 Visual Studio 2002와 함께 출시되었으며, Java와 형태가 매우 흡사하였다. 이는 “단순하고 현대적인 범용 객체지향 언어”를 추구한 결과였다. C# 언어의 특징은 .NET 프로그램이 동작하는 .NET 플랫폼을 가장 직접적으로 반영하고, .NET 플랫폼에 강하게 의존하는 프로그래밍 언어이다. C++이 확장되어 +가 4개 붙은 모양이 #모양과 비슷하다지만 언어 자체는 C++보다 자바와 비슷하다. 객체-클래스 모델과 쓰레기 수집이 그 예가 되겠다

**C# 1.0**

C#의 출시 버전인 C# 1.0은 비동기 기능과 제네릭 기능이 없었다. LINQ 또한 없었으나 C# 1.0은 Windows플랫폼에서 Java를 대체하는 실용적인 방법이었다. C# 1.0때부터 이벤트, 델리게이트 등이 포함되어 있었으며, 클래스와 인터페이스 또한 포함되어있었다.

**※ C# 1.2**

C# 1.2에는 IEnumerator가 IDsiposable를 구현할 때 IEnumerator 인터페이스의 Dispose라고 불리우는 foreach 루프를 구현할 수 있게 되었다.

**C# 2.0**

* C# 2.0의 주요특징으로는 제네릭(Generic)과 부분 형식(Partial Type)이 있겠다. 뿐만 아니라 무명메소드(Anoymous methods), Nullable 형식, iterator, Corvariance and Contravariance등이 추가되었다. 또한, 기존 기능에 getter/setter등의 Property, 정적 클래스, 대리자 유추 등이 추가되었다.
* 제네릭을 사용할 수 있게 되면서 형식을 안전하게 유지하면서 임의의 형식에서 형식 및 메서드를 작동할 수 있게 되었다. 예를 들어 List를 사용하면 List 또는 List를 사용하고 이를 반복하는 동안 해당 문자열이나 정수에 형식이 안전한 작업을 수행할 수 있었다. 제네릭을 사용하는 것이 모든 작업에서 ArrayList에서 파생되거나 Object에서 캐스팅한 ListInt를 만드는 것보다 좋았다.
* C# 2.0부터 iterator가 도입되었다. iterator를 사용하면 열거 가능 형식의 모든 항목을 foreach 루프로 검사할 수 있다

// Generic

class MyStack<T>

{

T[] \_elements;

int pos = 0;

public MyStack()

{

\_elements = new T[100];

}

public void Push(T element)

{

\_elements[++pos] = element;

}

public T Pop()

{

return \_elements[pos--];

}

}

// 두 개의 서로 다른 타입을 갖는 스택 객체를 생성

MyStack<int> numberStack = new MyStack<int>();

MyStack<string> nameStack = new MyStack<string>();

출처: http://www.csharpstudy.com/CSharp/CSharp-generics.aspx

// iterator

int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };

foreach (int i in numbers)

{

System.Console.Write("{0} ", i);

}

// Output : 4 5 6 1 2 3 -2 -1 0

// 무명 메소드와 람다식

// 무명 메소드

MyDelegate A;

A = delegate(int a,int b)

{

return a + b;

}

// 람다식, 무명메소드에 비해 간결해졌다

A = (int a,int b) => a + b;

* iterator를 언어의 첫번째 클래스 부분에 사용하면 언어의 가독성과 코드 추론 능력이 크게 향상된다

**C# 3.0**

* C# 3.0은 2007년에 Visual Studio 2008과 함께 출시되었다. 3.0은 C#의 성장에 있어서 아주 큰 변화를 가져왔는데, 덕분에 C#은 진정으로 강력한 프로그래밍 언어로 자리매김하였다.
* 3.0의 큰 골자는 LINQ(Language-Intergrated Query)의 도입과 var 키워드의 지원이다. 쿼리식 람다식 및 익명형식이 LINQ의 생성기반이다

**※ LINQ**

* LINQ를 사용할 경우 쿼리가 클래스, 메소드, 이벤트와 같은 고급 언어 구문이 된다. 언어 키워드 및 친숙한 연산자를 사용하여 강력한 형식의 개체 컬렉션에 대해 쿼리를 작성한다

// var (익명형식)

var v = new { Amount = 108, Message = "Hello" };

// Rest the mouse pointer over v.Amount and v.Message in the following

// statement to verify that their inferred types are int and string.

Console.WriteLine(v.Amount + v.Message);

// LING

class LINQQueryExpressions

{

static void Main()

{

// Specify the data source.

int[] scores = new int[] { 97, 92, 81, 60 };

// Define the query expression.

IEnumerable<int> scoreQuery =

from score in scores

where score > 80

select score;

// Execute the query.

foreach (int i in scoreQuery)

{

Console.Write(i + " ");

}

}

}

// Output: 97 92 81

**C# 4.0**

* C#에서 도입된 것은 동적바인딩, 명명된/선택적 인수, Generic Covariant and Conttravariant, 포함된 interop 형식 등이 있다.
* Generic Covariant and Conttravariant은 제네릭을 사용하는데 있어서 더 많은 기능을 제공하지만 다소 학문적이고 프레임워크나 라이브러리를 만드는 사람에게 각광받앗다
* 명명된/선택적인수의 도입은 메소드 오버로딩을 줄여주어 편리함을 주엇다.
* 가장 중요한 것은 dynamic 키워드인데, 이 키워드는 C# 4.0부터 컴파일 시간에 컴파일러를 재정의하는 기능을 도입하였다.

//dynamic 키워드는 컴파일러에게 변수의 type을 체크하지 않도록 하고 런타임시까지는 해당 타입을 알 수없음으로 표시한다.

// 1. dynamic은 중간에 형을 변환할 수 있다.

dynamic v = 1;

// System.Int32 출력

Console.WriteLine(v.GetType());

v = "abc";

// System.String 출력

Console.WriteLine(v.GetType());

// 2. dynamic은 cast가 필요없다

object o = 10;

// 틀린표현

// (에러: Operator '+' cannot be applied to operands of type 'object' and 'int')

o = o + 20;

// 맞는 표현: object 타입은 casting이 필요하다

o = (int)o + 20;

// dynamic 타입은 casting이 필요없다.

dynamic d = 10;

d = d + 20;

출처: http://www.csharpstudy.com/CSharp/CSharp-dynamic.aspx

**C# 5.0**

* 5.0에서는 비동기 멤버, 호출자 정보 특성 등의 기능이 추가되었지만 C# 7.0의 비동기 프로그래밍을 위한 async, await모델로 옮겨졌다

**C# 6.0**

* 6.0에서는 C# 프로그래밍을 보다 생산적으로 만드는 작은 기능이 릴리즈 되었다. 예를 들어 static import, Exception Filter, Auto-property initializers, null propagator 등이다
* 또한 Catch/Finally 블록의 Await와 Getter 전용 속성의 기본값이 추가되었다
* 그리고 Roslyn 서비스형 컴파일러가 릴리즈되어 C# 컴파일러는 이제 C#으로 작성된다

**※ Auto-property initializers**

public ICollection<double> Grades { get; } = new List<double>();

**※ static import**

// using static으로 static import를 했을 경우

using static System.Math;

Pow(2, 5)

Max(30, 40)

//하지 않았을 경우

Math.Pow(2, 5)

Math.Max(30, 40)

**C# 7.0**

* 7.0의 주요 골자는 out키워드로 사용할 변수의 선언을 압축하는 것과 튜플을 통해 여러개의 반환값을 허용하는 것이다.
* 그외에도 pattern matching, local function. expanded expression bodied members, ref locals and resturns 등이 이전보다 훨씬 깔끔한 코드를 작성하도록 돕는다
* out, ref는 참조에 의한 호출을 지원하기 위한 예약어이다.

public static void Add(int i, ref int result)

{

result += i;

return ;

}

public static void Add(int i, int j, out int result) // out 매개변수는 메서드를 진행하는 동안 할당해야한다.

{

result = i + j;

return;

}

**C# 7.0 이후**

* 7.0 이후로 7.1, 7.2, 7.3 그리고 8.0에 이르기까지 많은 개선이 있어왔다.
* Readonly Members, Nullable reference type, Asynchronous streams 등등

**참고자료**

* Microsoft docs C# 버전 기록 <https://docs.microsoft.com/ko-kr/dotnet/csharp/whats-new/csharp-version-history>
* 출처가 적혀있지 않은 것들은 모두 Microsoft docs의 C#가이드에서 발췌(위와 링크 동일)