





C# -CHAPTER8-

SOUL SEEK



목차

1. 리플렉션과 애트리뷰트
 2. Dynamic형식
- 
- 

리플렉션과 애트리뷰트

1. 리플렉션과 애트리뷰트

리플렉션

- 프로그램 실행 도중에 객체의 정보를 조사하거나, 다른 모듈에 선언된 인스턴스를 생성하거나, 기존 개체에서 형식을 가져오고 해당하는 메소드를 호출, 또는 해당 필드와 속성에 접근할 수 있는 기능
- **System.Reflection** 사용.
- 코드는 어셈블리에 의해서 기계어로 바뀌면서 해당 타입정보는 사라진다.
 - ➔ **C#**에서는 컴파일 된 실행파일에 타입에 대한 메타정보가 저장됩니다. 따라서 실행 중에도 해당 타입에 대한 정보를 불러와서 사용이 가능

리플렉션 Method

Method	반환 형식	설명
GetConstructors()	ConstructorInfo[]	해당 형식의 모든 생성자 목록을 반환
GetEvents()	EventInfo[]	해당 형식의 이벤트 목록을 반환
GetFields()	FieldInfo[]	해당 형식의 필드 목록을 반환
GetGenericArguments()	Type[]	해당 형식의 형식 매개 변수 목록을 반환
GetInterfaces()	Type[]	해당 형식이 상속하는 인터페이스 목록을 반환
GetMembers()	MemberInfo[]	해당 형식의 멤버 목록을 반환
GetMethods()	MethodInfo[]	해당 형식의 메소드 목록을 반환
GetNestedTypes()	Type[]	해당 형식의 내장 형식 목록을 반환
GetProperties()	PropertyInfo[]	해당 형식의 프로퍼티 목록을 반환

1. 리플렉션과 애트리뷰트

System.Reflection.BindingFlags 열거형을 이용해 원하는 **Field**를 **Flag** 옵션으로 찾아낼 수 있다.

```
Type type = a.GetType();
```

```
// public 인스턴스 필드 조회
```

```
var fields1 = type.GetFields(BindingFlags.Public | BindingFlags.Instance);
```

```
// 비 public 인스턴스 필드 조회
```

```
var fields2 = type.GetFields(BindingFlags.NonPublic | BindingFlags.Instance);
```

```
// public 정적 필드 조회
```

```
var field3 = type.GetFields(BindingFlags.Public | BindingFlags.Static);
```

```
// 비 public 정적 필드 조회
```

```
var field4 = type.GetFields(BindingFlags.NonPublic | BindingFlags.Static);
```

인스턴스가 있어야 호출이 가능한 **Object.GetType()**을 사용하지 않고 **Type**을 알아오는 방법.

```
Type a = typeof(int);
```

```
Console.WriteLine(a.FullName);
```

```
Type b = Type.GetType("System.Int32");
```

```
Console.WriteLine(b.FullName);
```

typeof 연산자의 매개 변수는 **int**

Type.GetType() 메소드의 매개 변수는 형식의 네임스페이스를 포함하는 전체 이름

1. 리플렉션과 애트리뷰트

리플렉션을 이용해서 객체를 생성하고 이용하기

- 리플렉션을 제대로 활용 할 수 있는 방법.
- 코드 안에서 런타임에 특정 형식의 인스턴스를 만들 수 있게 하는 것.
 - 프로그램을 조금 더 동적으로 동작할 수 있도록 구성이 가능.
- **System.Activator** 클래스의 도움을 받아서 구현 할 수 있다.

object a = Activator.CreateInstance(typeof(int));

인스턴스를 만들고자 하는 형식의 **Type**객체를 **typeof**를 이용해 매개 변수로 넘기면,
Activator.CreateInstance() 메소드는 입력 받는 형식의 인스턴스를 생성하여 반환한다.

이것을 일반화 버전으로 만들면..

List<int> list = Activator.CreateInstance<List<int>>();

1. 리플렉션과 애트리뷰트

객체의 프로퍼티에 값을 할당하는 것도 동적으로 할 수 있다.

- **Type.GetProperties()**의 반환형식인 **PropertyInfo** 객체는 **SetValue()**와 **GetValue()** 메소드를 가지고 있다. 이를 이용해 **GetValue()**를 호출하면 프로퍼티로부터 값을 읽을 수 있고, **SetValue()**를 호출하면 프로퍼티에 값을 할당할 수 있다.

Class Profile

```
{
    public string Name { get; set; }
    public string Phone { get; set; }
}

static void Main()
{
    Type type = typeof(Profile);
    Object profile = Activator.CreateInstance(type);

    PropertyInfo name = type.GetProperty( "Name" );
    PropertyInfo phone = type.GetProperty( "Phone" );

    name.SetValue(profile, "박찬호", null);
    phone.SetValue(profile, "997-5511", null);
    Console.WriteLine("{0}, {1}", name.GetValue(profile, null),
                        phone.GetValue(profile, null));
}
```

Type.GetProperties() 메소드는 그 형식의 모든 프로퍼티를 **PropertyInfo** 형식의 배열로 반환하지만, **Type.GetProperty()** 메소드는 특정 이름의 프로퍼티를 찾아 그 프로퍼티의 정보를 담은 하나의 **PropertyInfo** 객체만을 반환한다.

1. 리플렉션과 애트리뷰트

리플렉션을 이용해 메소드를 호출하는 방법

- **MethodInfo** 클래스에는 **Invoke()**라는 메소드가 있다, 이 메소드를 이용하면 동적으로 메소드를 호출하는 것이 가능하다.

class Profile

```
{  
    public string Name{ get; set; }  
    public string Phone{ get; set; }  
    public void Print()  
    {  
        Console.WriteLine("{0}, {1}", Name, Phone);  
    }  
}
```

static void Main()

```
{  
    Type type = typeof(Profile);  
    Profile profile = (Profile)Activator.CreateInstance(type);  
    profile.Name = "류현진";  
    profile.Phone = "010-1412-2222";  
  
    MethodInfo method = type.GetMethod("Print");  
    method.Invoke(profile, null);  
}
```

null 매개 변수가 오는 자리에는 **Invoke()** 메소드가 호출할 메소드의 매개 변수가 와야 한다. 여기에서는 **Profile.Print()** 메소드의 매개 변수가 없으므로 **null**을 넘기는 것이다.

1. 리플렉션과 애트리뷰트

형식 내보내기

- **C#**은 프로그램 실행 중에 새로운 형식을 만들어 낼 수 있는 기능도 제공하고 있다.
 - **System.Reflection.Emit** 네임스페이스에 있는 클래스들을 통해 이루어 진다
 - **Emit**은 프로그램이 실행 중에 만들어 낸 새 형식을 **CLR**의 메모리에 “내보낸다”는 의미

Emit 네임스페이스의 클래스

클래스	설명
AssemblyBuilder	동적 어셈블리를 정의하고 나타낸다.
ConstructorBuilder	동적으로 만든 클래스의 생성자를 정의하고 나타낸다.
CustomAttributeBuilder	사용자 정의 애트리뷰트를 만든다
EnumBuilder	열거 형식을 정의하고 나타낸다.
EventBuilder	클래스의 이벤트를 정의하고 나타낸다.
FieldBuilder	필드를 정의하고 나타낸다.
GenericTypeParameterBuilder	동적으로 정의된 형식(클래스)과 메소드를 위한 일반화 형식 매개 변수를 정의하고 생성한다.
ILGenerator	MSIL(Microsoft Intermediate Language) 명령어를 생성한다.
LocalBuilder	메소드나 생성자 내의 지역 변수를 나타낸다.
MethodBuilder	동적으로 만든 클래스의 메소드(또는 생성자)를 정의하고 나타낸다.

1. 리플렉션과 애트리뷰트

클래스	설명
ModuleBuilder	동적 어셈블리 내의 모듈을 정의하고 나타낸다.
OpCodes	ILGenerator 클래스의 멤버를 이용한 내보내기 작업에 사용할 MSIL 명령어의 필드 표현을 제공.
ParameterBuilder	매개 변수 정보를 생성하거나 결합한다.
PropertyBuilder	형식(클래스)의 프로퍼티를 정의한다.
TypeBuilder	실행 중에 클래스를 정의하고 생성한다.

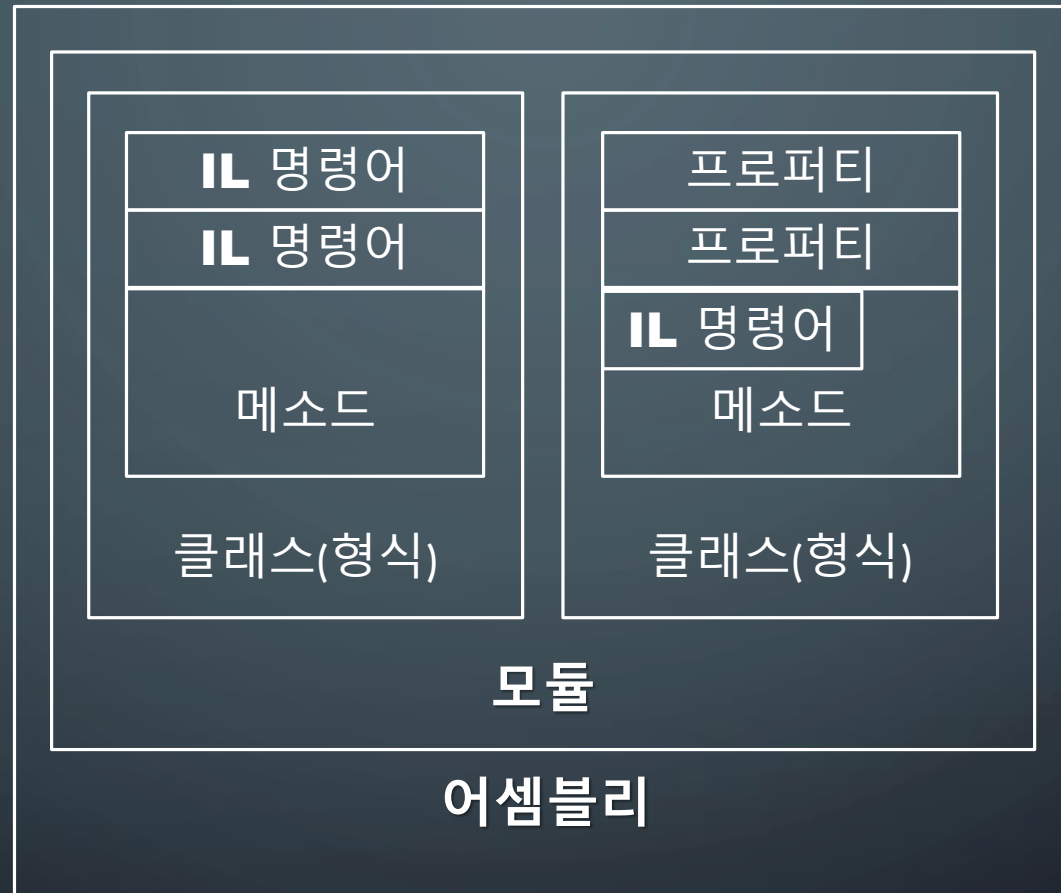
클래스 사용 요령과 순서

1. **AssemblyBuilder**를 이용해서 동적으로 어셈블리를 만든다.
2. **ModuleBuilder**를 이용해서 1에서 생성한 어셈블리 안에 모듈을 만들어 넣는다.
3. **TypeBuilder**를 2에서 생성한 모듈 안에 클래스(형식)를 만들어 넣는다.
4. 3에서 생성한 클래스 안에 매소드(**MethodBuilder** 이용)나 프로퍼티(**PropertyBuilder** 이용)를 만들어 넣는다.
5. 4에서 생성한 것이 매소드라면, **ILGenerator**를 이용해서 매소드 안에 **CPU**가 실행할 **IL** 명령어들을 넣는다.

[어셈블리] → [모듈] → [클래스] → [매소드] 또는 [프로퍼티]로 이어지는 .NET 프로그램의 계층 구조를 이해했다면 순서가 이해 될 것이다.

1. 리플렉션과 애트리뷰트

.NET 프로그램의 계층 구조.



1. 리플렉션과 애트리뷰트

앞의 순서대로 새 형식을 만들어 보자!!

AssemblyBuilder 클래스를 이용해 만들어야 하지만 **AssemblyBuilder**는 스스로 생성하는 생성자가 없기 때문에 다른 팩토리 클래스의 도움을 받아야 한다.

→ 어셈블리가 **.NET** 프로그램을 실행하면 최상위 파일 단위이지만, 메모리안에서는 프로그램을 실행하면 메모리에 프로세스가 생성되고 **AppDomain**을 만들고

AppDomain은 어셈블리를 로딩해서 실행할 것들을 메모리에 적재한다.

→ **AppDomain**은 **AssemblyBuilder**를 만들 수 있는 객체이고 **System.AppDomain** 클래스를 말한다.

먼저, **CalculatorAssembly**를 생성해보자!

AssemblyBuilder newAssembly =

CurrentDomain 프로퍼티는
현재 코드가 실행되고 있는
AppDomain을 반환한다.

**AppDomain.CurrentDomain.DefineDynamicAssembly(
new AssemblyName("CalculatorAssembly"),
AssemblyBuilderAccess.Run);**

AssemblyBuilder는 동적 모듈을 생성하는 **DefineDynamicModule()** 메소드를 갖고 있으므로 이 메소드를 호출해서 모듈을 만들면 된다.

1. 리플렉션과 애트리뷰트

다음, **Calculator**라는 모듈을 만들자.

```
ModuleBuilder newModule =  
    newAssembly.DefineDynamicModule("Calculator");
```

다음, **Sum1To100**이라는 클래스를 만들자.

→ **ModuleBuilder**의 **DefineType()** 메소드를 이용해서 클래스를 생성한다.

```
TypeBuilder newType = newModule.DefineType("Sum1To100");
```

다음, **Calculate()** 메소드를 만들자.

→ **TypeBuilder** 클래스의 **DefineMethod()** 메소드를 호출해서 만들 수 있다.

```
MethodBuilder newMethod = newType.DefineMethod  
(  
    "Calculate",  
    MethodAttribute.Public, // 한정자  
    typeof(int),           // 반환 형식  
    new Type[0] );         // 매개 변수
```

1. 리플렉션과 애트리뷰트

다음, 1 부터 100까지의 합을 구하는 구현부를 만들어보자.

→ 메소드의 껍데기까지 만들었으니, 이 안에 메소드가 실행할 **IL 명령어(코드)**를 채워 넣어야 한다.

→ **ILGenerator** 객체를 통해서 이루어 지며, **ILGenerator** 객체는 **MethodBuilder** 클래스의 **GetILGenerator()** 메소드를 통해 얻을 수 있다.

```
ILGenerator generator = newMethod.GetILGenerator();  
generator.Emit(OpCodes.Ldc_I4, 1);
```

32비트 정수(1)를 계산 스택에 넣는다.

```
for(int i = 2; i <= 100; i++)  
{
```

```
    generator.Emit(OpCodes.Ldc_I4, i);
```

32비트 정수(1)를 계산 스택에 넣는다.

```
    generator.Emit(OpCodes.Add);
```

계산 후 계산 스택에 담겨 있는 두개의 값을 꺼내서 더한 후, 그 결과를 다시 계산 스택에 넣는다

```
generator.Emit(OpCodes.Ret); //계산에 담겨 있는 값을 반환한다.
```

1. 리플렉션과 애트리뷰트

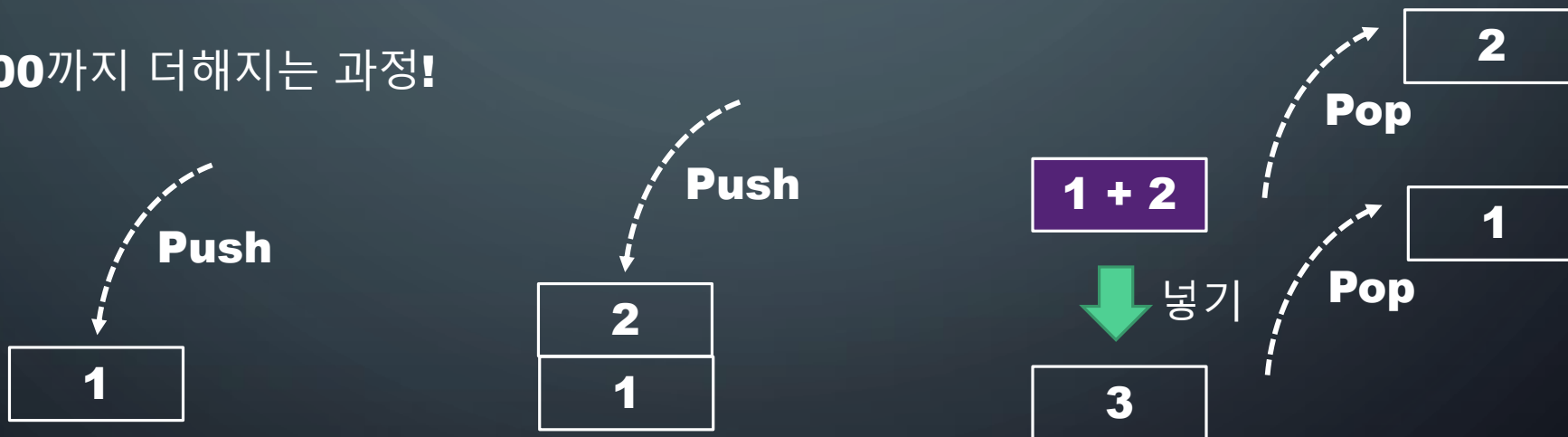
다음, **Sum1To100** 클래스를 **CLR**에 제출.

```
newType.CreateType();
```

다음, 동적할당을 하고 사용해 보자.

```
object sum1To100 = Activator.CreateInstance(newType);  
MethodInfo Calculate = sum1To100.GetType().GetMethod("Calculate");  
Console.WriteLine(Calculate.Invoke(sum1To100, null));
```

1에서 **100**까지 더해지는 과정!



1과 **2**가 스택에 쌓이고 둘이상이 쌓였으니 **1**과 **2**를 빼내서 합을 구하고 **3**이 들어와서 **1 + 2**를 한 값과 다시 둘이상이 쌓였으므로 **3 + 3**을 합니다. 이런 식으로 **10**까지 계산하는 것이다.

1. 리플렉션과 애트리뷰트

애트리뷰트

- 코드에 대한 부가 정보를 기록하고 읽을 수 있는 기능.
- 주석을 이용할 수 있지만 주석과 다르게 런타임 상황에서 클래스나 구조체, 메소드, 프로퍼티 등에 데이터를 기록해두면 이 정보를 **C#** 컴파일러나 **C#**으로 작성된 프로그램이 이 정보를 읽어 사용할 수 있다.
- 코드에 대한 정보는 **Notice**로 남길 때, 유용하다.
→이외에 활용처가 무궁무진 하다. 알아 두면 유용하다.

메타데이터

- 코드에 대한 정보, 데이터가 있는데 이를 메타데이터라 부른다.
- 애트리뷰트나 리플렉션을 통해 얻는 정보들도 **C#**코드의 메타데이터라고 할 수 있다.
- **Unity**에서 각 파일들의 정보를 가지고 있는 메타파일이라는 것이 존재한다.

애트리뷰트 사용 형식

- 설명하고자 하는 코드 요소 앞에 **[]**의 괄호 안에 애트리뷰트의 이름을 넣으면 된다.

[애트리뷰트 이름(애트리뷰트 매개 변수)]

```
public void MyMethod()
```

```
{  
}
```


1. 리플렉션과 애트리뷰트

보통 이런 **Notice**를 남기면 오류 목록창에 나타나게 된다.

```
Class MyClass
```

```
{
```

```
    [Obsolete("OldMethod는 폐기 되었습니다. NewMethod를 사용하세요.")]
```

```
    public void OldMethod()
```

```
    {
```

```
        Console.WriteLine("I'm old");
```

```
    }
```

```
    public void NewMethod()
```

```
    {
```

```
        Console.WirteLine("I'm new");
```

```
    }
```

```
}
```

1. 리플렉션과 애트리뷰트

호출자 정보 애트리뷰트

- 메소드의 매개 변수에 사용되며, 메소드의 호출자 이름, 호출자 메소드가 정의되어 있는 소스파일 경로, 소스파일 내의 행 번호까지 알 수 있다.
- 응용해서 프로그램의 이벤트 로그나 화면에 출력하면 그 이벤트가 어떤 코드에서 일어났는지를 알 수 있다.

애트리뷰트	설명
CallerMemberNameAttribute	현재 메소드를 호출한 메소드 또는 프로퍼티의 이름을 나타낸다.
CallerFilePathAttribute	현재 메소드가 호출된 소스 파일 경로를 나타낸다. 이 때 경로는 소스 코드를 컴파일할 때의 전체 경로를 나타낸다.
CallerLineNumberAttribute	현재 메소드가 호출된 소스 파일 내의 행 번호를 나타낸다.

```
public static class Trace
{
    public static void WriteLine("string message",
                                [CallerFilePath] string file = "",
                                [CallerLineNumber] int line = 0,
                                [CallerMemberName] string member = "")
    {
        Console.WriteLine("{0}(Line:{1}) {2}: {3}", file, line, member, message);
    }
}
```

1. 리플렉션과 애트리뷰트

직접 만들어서 사용하는 애트리뷰트

- **Obsolete** 말고도 다양한 애트리뷰트가 있다.
- **System.Attribute**를 상속하는 것으로 애트리뷰트를 만들 수 있다.

```
class History : System.Attribute
```

```
{
```

```
    private string programmer;
```

```
    public double version
```

```
{
```

```
        get;
```

```
        set;
```

```
}
```

```
    public string Changes
```

```
{
```

```
        get;
```

```
        set;
```

```
}
```

```
}
```

```
    public History(string programmer)
```

```
{
```

```
        this.programmer = programmer;
```

```
        Version = 1.0;
```

```
        Changes = "First release";
```

```
}
```

```
    public string Programmer
```

```
{
```

```
        get { return programmer; }
```

```
}
```

1. 리플렉션과 애트리뷰트

History 클래스를 사용해 보자.

```
[History("Sean", Version = 0.1, Changes = "2019-11-04 Created class stub")]
```

```
// [History("Sean", Version = 0.2, Changes = "2019-12-04 Created class stub")]
```

```
class MyClass
{
    public void Func()
    {
        Console.WriteLine("Func()");
    }
}
```

다중으로 넣고 싶지만 이
상태에서는 불가능하다.

정보에 대한 구문을 다중으로 넣고 싶지만 이 상태에서는 불가능하다.

다중 구문을 넣고 싶다면 **System.AttributeUsage**라는 애트리뷰트의 도움을 받아야 한다.

```
[System.AttributeUsage(System.AttributeTargets.Class, AllowMultiple = true)]
```

```
class History : System.Attribute
{
}
```

첫 번째 매개변수로 오는 타겟의 형식은 다양하다.

1. 리플렉션과 애트리뷰트

System.AttributeTargets의 종류

Targets	설명	Targets	설명
All	모든 요소	Constructor	생성자
Assembly	어셈블리	Delegate	델리게이트
Module	모듈	Enum	열거형
Interface	인터페이스	Event	이벤트
Class	클래스	Field	필드
Struct	구조체	Property	프로퍼티
ClassMembers	클래스의 모든 멤버	Method	메소드
Parameter	메소드의 매개 변수	ReturnValue	메소드의 반환 값

타겟을 사용할 때는 결합형으로도 사용할 수 있다.

```
[System.AttributeUsage(System.AttributeTargets.Class  
                        | System.AttributeTarget.Method,  
                        AllowMultiple = true)]
```

```
class History : System.Attribute
```

```
{
```

```
    //...
```

```
}
```

DYNAMIC형식

2. DYNAMIC형식

Dynamic

int나 **string** 처럼 또 하나의 형식.

다른 형식들은 컴파일할 때 검사가 이루어지지만 **Dynamic** 형식은 실행할 때 이루어 진다.

Class MyClass

```
{  
    public void FuncAAA() { // Do Nothing };  
}
```

FuncBBB()라는 매소드는 **MyClass**에 정의되어 있지 않으므로 컴파일에서 에러가 발생

class MainApp

```
{  
    static void Main(string[] args)  
    {  
        MyClass obj = new MyClass();  
        obj.FuncAAA();  
        obj.FuncBBB();  
    }  
}
```

dynamic 형식으로 선언된 **obj**는 일단 컴파일러의 형식 검사는 피해간다.

class MainApp

```
{  
    static void Main(string[] args)  
    {  
        dynamic obj = new MyClass();  
        obj.FuncAAA();  
        obj.FuncBBB();  
    }  
}
```

2. DYNAMIC형식

- 컴파일러에서 이미 강력한 검사를 통해 위험요소를 체크하는 부분을 일부로 회피할 필요가 있을까? **dynamic**을 쓴다고 해도 이미 실행과정에서 오류를 발생시킬 것이며, 그렇다면 왜 이것이 존재하는 지는 더 알아볼 필요가 있다.
- 객체 지향프로그래밍에서는 **C#**의 어떤 형식이 오리라는 클래스와 같은 것으로 인정받으려면 상속을 받아야만 가능하다.
- 하지만 구지 클래스명이 달라도 멤버 형식과 멤버 메소드가 같은 행동을 한다면 같은 것으로 판단하는 것이 맞지 않냐 라는 생각에서 나온 방법이다.

class Duck

```
{  
    public void Walk()  
    {Console.WriteLine("Duck.Walk")}  
  
    public void Swim()  
    {Console.WriteLine("Duck.Swim")}  
  
    public void Quack()  
    {Console.WriteLine("Duck.Quack")}  
}
```

class Robot

```
{  
    public void Walk()  
    {Console.WriteLine("Duck.Walk")}  
  
    public void Swim()  
    {Console.WriteLine("Duck.Swim")}  
  
    public void Quack()  
    {Console.WriteLine("Duck.Quack")}  
}
```

비록, 클래스명은 다르지만 역할이 같으니까 같은 동작을 해서 똑같이 사용할 수 있다!

2. DYNAMIC형식

- 인터페이스와 상속을 이용하면 더 깔끔하고 멋지게 만들 수 있다. 하지만, 상속과 인터페이스 설계는 추상화에 대한 연습과 경험이 많아야 예측을 하고 사용 할 수 있는 것이다. 인터페이스의 경우 잦은 변경이 발생한다면 사용한 파생 클래스 모두에게 수정이 필요해 지는 상황이 된다.
- **dynamic**을 사용하는 것과 **Interface**나 상속을 이용하는 것은 필수가 아닌 선택이고 취향인 것이다.

COM과 .NET 사이의 상호 운용성을 위한 Dynamic 형식

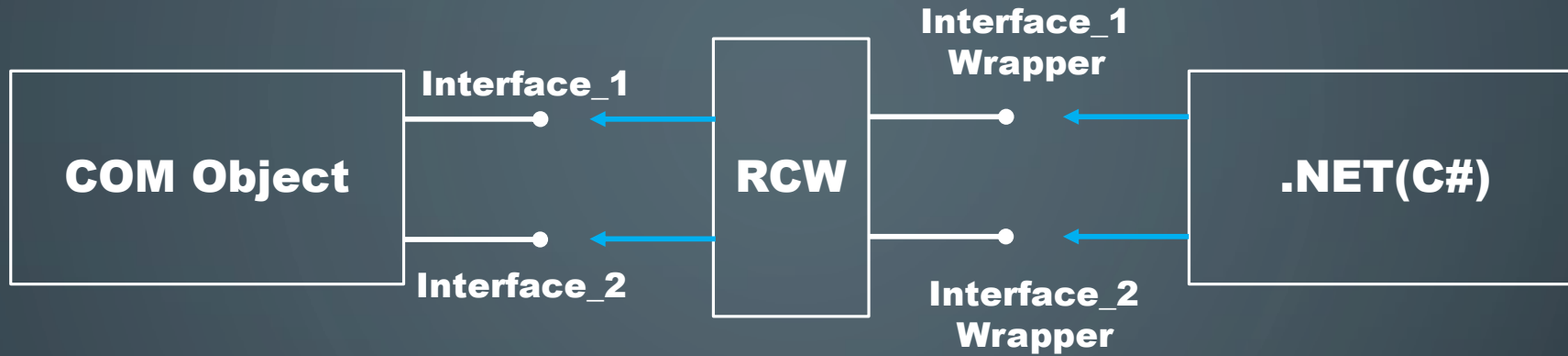
COM이란?

- **Componet Object Model**의 약자 – 마이크로소프트의 소프트웨어 컴퍼넌트 규격을 말한다.
- **OLE, ActiveX, COM+**와 같은 파생 규격들이 모두 **COM**을 바탕으로 만들어졌다.
- **.NET**의 영향으로 사용하지 않을 수도 있지만 이미 정립된 소프트웨어들에 대한 지원이 필요하기 때문에 **.NET**에서 **COM**을 여전히 지원하고 있다. 대표적으로 **MS Office**.

C#은 어떻게 지원을 하고 있는 것인가?

- **.NET** 언어들은 **RCW(Runtime Callable Wrapper)**를 통해 **COM** 컴퍼넌트를 사용할 수 있다.
- **RCW**는 **.NET Framework**가 제공하는 **Type Library Importer(tlbimp.exe)**를 이용해서 만들 수 있는데, 비주얼 스튜디오를 이용해서 **COM** 객체를 프로젝트 참조에 추가하면 **IDE**가 자동으로 **tlbimp.exe**를 호출해서 **RCW**를 만들어 준다.
- **RCW**는 **COM**에 대한 프록시 역할을 함으로써 **C#** 코드에서 **.NET** 클래스 라이브를 사용하듯 **COM API**를 사용할 수 있게 해준다.

2. DYNAMIC형식



1. **COM**은 메소드가 결과를 반환할 때 실제 형식이 아닌 **object**형식으로 반환한다. 이 때문에 **C#**코드에서는 이 결과를 실제 형식으로 변환해줘야 하는 번거로움이 있다.
2. **COM**은 오버로딩을 지원하지 않는다. 대신 메소드의 선택적 매개 변수의 기본값 매개 변수를 지원한다. **C# 4.0**이상의 버전이 되기 전까지는 선택적 매개 변수와 기본값 매개 변수를 지원하지 못했기 때문에 번거로움이 많았다.
3. 다행히 **4.0**이상의 버전으로 오면서 문제가 해결되었고 우리는 적극 적으로 **COM API**를 활용 할 수 있다.

Excel COM 컴포넌트를 사용하여 예제를 만들어 보면 쉽게 알 수 있다.

