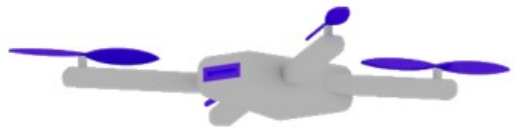

.NET Conf 2023
x Seoul

C#11 static abstract members
이해와 대비

발표 구성 및 목표

- F# interfaces-with-static-abstract-members RFC 문서를 중심으로 내용 작성됨
- C# static abstract interface 기능 이해하기
- 다른 언어의 비슷한 기능 둘러보기
- 우려되는 점을 알아보기

틀린 내용이 있다면 봐주십시오... 😿

왜 조사를 시작했는가?

> Sang Kil Cha | F# Korea Slack
>
> 이번 .NET7에서 상당히 큰 언어적인 변화가 있었습니다. static abstract member를 선언할 수 있게 된 점인데,
> F#7에서도 지원하게 되었습니다. 이게 상당히 매력적이면서도 위험한 언어적인 요소라 F#개발자들이 엄청 고심한 흔적이 엿보입니다.
> 이 부분을 읽어보시길 추천드립니다.
>
> 개인적으로는 F#이 언어적인 아름다움을 더 갖게 되어 좋네요...

- 매력적?
- 위험한?
- 아름다움?

F#7에서의 경고

F# RFC FS-1124 - Interfaces with static abstract members

지침

- 유형 분류 충동에 굴복하지 마십시오
- 최대 추상화 충동에 굴복하지 마십시오
- 애플리케이션 코드에서 IWSAM을 사용하면 귀하 또는 귀하의 팀이 나중에 해당 사용을 제거할 때 큰 위험이 따릅니다
- 구현이 안정적이고 폐쇄형이며 논쟁의 여지가 없는 유형에서만 IWSAM을 구현하십시오
- 구성 프레임워크의 기초로 IWSAM을 사용하지 마십시오

Announcing F# 7 | Static abstract members support in interfaces

이러한 단점들 때문에, F#에서

[warning FS3535] static abstract method를 포함한 interface를 선언하거나

[warning FS3536] 제네릭 형식 매개변수에 대한 제약 조건 상황 밖에서, 타입으로 사용되면

경고를 표시합니다.

이때의 감상

- 왜 이번에 별 예고도 없이 F#이 6 -> 7 버전업이 되었는가
  - 심지어 별로 추가된 기능도 없음
- static abstract members가 뭔가
  - 어디에 쓰는 건데
- 왜 이렇게 부정적인가
  - null checking operator `!!` 같은 기능이 또 나왔나?

static abstract members란?

.NET 6에서 미리보기 기능으로 들어갔으며 .NET 7에서 정식 기능으로 편입

`static virtual members in interface` 기능은 **Generic Math Support**를 위해 추가된 언어 기능 중 하나

- interface
  - **static virtual members in interfaces**
- operator
  - checked user defined operators `checked(~~~)`
  - relaxed shift operators *이제 shift류 연산자의 두번째 인수가 정수(혹은 정수로 암시적 변환)가 아니어도 됨*
  - unsigned right-shift operator

.NET 6 | Static abstract members declared in interfaces

.NET 7 | Tutorial: Explore C# 11 feature - static virtual members in interfaces

C# 11 | Generic Math Support

static abstract members 단순 예제 (1)

```
interface IFavorite
{
  static abstract string Favorite { get; }
  static virtual int SizeAtAge(int age) => age;
}

class Dog : IFavorite
{
  public static string Favorite { get => "Bones"; }
  public static int SizeAtAge(int age) => age * 2;
}
class Cat : IFavorite
{
  public static string Favorite { get => "Fish"; }
}
class Tiger : Cat, IFavorite
{
  new public static string Favorite { get => "Human"; }
  public static int SizeAtAge(int age) => age * 4;
}
```

static abstract members 단순 예제 (2)

```
1   void whatIsYourFavorite<T>(T iHaveAFavorite) where T : IFavorite
2   {
3     Console.WriteLine($"{iHaveAFavorite.GetType().Name}'s favorite is {T.Favorite}");
4     Console.WriteLine($"{iHaveAFavorite.GetType().Name}': size at age 5 :  {T.SizeAtAge(5)}");
5   }
6
7   whatIsYourFavorite(new Dog());
8   // Dog's favorite is Bones
9   // Dog': size at age 5 :  10
10
11  whatIsYourFavorite(new Cat());
12  // Cat's favorite is Fish
13  // Cat': size at age 5 :  5
14
15  whatIsYourFavorite(new Tiger());
16  // Tiger's favorite is Human
17  // Tiger': size at age 5 :  20
```

Generic Math란?

Generic Math

```
1   static T Add<T>(T left, T right) where T : INumber<T>
2   {
3       return left + right;
4   }
```

- 수학적 연산을 지원하는 파라미터의 타입을 제네릭하게 선언하고
- 위 제약 안에서 연산자를 통한 표현식 지원

Generic Math의 static abstract members 필요성

> ... 연산자는 반드시 `static`으로 선언되어야 했기 때문에 ...

Operator overloading (C# reference): *연산자 선언은 public와 static 한정자를 가져야한다*

> ... 이 기능(static abstract member)을 통해 연산자가 number-like 인터페이스 안에 (abstract로) 선언될 수 있었다 ...

Generic Math 기능이 원하는 표현 방식

1. 연산자를 통한 수학적 표현
2. 피연산자는 현재 스코프에서는 일반 인터페이스로 타입 제약된 상태로 형식이 매우 자유로움

을 위해 인터페이스에 연산자를 선언하고 싶었구나!

연산자와 관계없는 Generic Math

with `static abstract members`

```
void doNumericThings<T1, T2>(T1 t1, T2 t2)
  where T1 : INumber<T1>
  where T2 : INumber<T2>
{
  var t1sOne = T1.One;
  var t2sZero = T2.Zero;
}
```

without `static abstract members` with `reflection`

```
void doNumericThings<T1, T2>(T1 t1, T2 t2)
  where T1 : INumber<T1>
  where T2 : INumber<T2>
{
  // System.Numerics.INumberBase<System.Int32>.One
  var t1sOne = t1.GetType().GetRuntimeProperties().Where(property => property.Name.Contains("One")).First().GetValue(null);
  // System.Numerics.INumberBase<System.Int16>.Zero
  var t2sZero = t2.GetType().GetRuntimeProperties().Where(property => property.Name.Contains("Zero")).First().GetValue(null)
}
```

Generic Math가 가져올 변화

> 이러한 인터페이스를 사용할 수 있다는 것은 제네릭 형식 또는 메서드의 **형식 매개 변수를 "숫자와 유사(number-like)"하도록** 제한할 수 있음

- 예를 들자면 숫자 인자 타입을 `int`, `float`이 아니라 `INumber`(number-like)로 선언할 수 있음

> 이러한 혁신을 통해 수학적 연산을 일반적으로, 즉 작업 중인 **정확한 유형을 알 필요 없이** 수행할 수 있습니다.
> 라이브러리 작성자는 "중복" 오버로드를 제거하여 코드 베이스를 단순화할 수 있기 때문에...
> 다른 개발자는 사용하는 API가 더 많은 형식을 지원하기 시작할 수 있으므로 간접적으로 이점을 얻을 수 있습니다.

- 작업자는 최대 공통 인터페이스로 작업하여 코드량을 줄이면서도
- 사용자는 해당 인터페이스를 지원하는 타입들을 더 많이 사용할 수 있음

Java: Static Method in Interface

근데 이름만 같고 목적과 사용법이 매우 다르다

- Java 8부터 인터페이스에 기본 메소드와 정적 메소드를 작성할 수 있음
- 인터페이스 내의 정적 메소드는 반드시 구현을 가지고 있어야 함
- 제네릭 메소드의 타입 파라미터로 클래스의 정적 메소드를 호출할 수는 없음
- 기능을 제공하되, 인스턴스화 될 수 없게끔 하는 제약을 제공 (기존에는 final + private constructor)
- Java에는 연산자 오버로딩이 없음

Static method in Interface in Java

Static and Default Methods in Interfaces in Java

What is the purpose of a static method in interface from Java 8?

Scala: method, implicit, trait

연산자가 정적 메소드일 필요가 없다는 것이 큰 차이점, 그리고 implicit이 scala에서 코드 축약의 핵심?

- Scala는 인스턴스 메소드가 연산자로서 동작
- 타입별로 암시적 변환 코드를 구현하여 인자의 타입을 맞출 수도 있고 (C#도 마찬가지)
- 제네릭, 트레이트, 암시적 파라미터를 통해
  - 기존 타입에 대한 서브 타이핑 없이 기능을 확장하고
  - 깔끔하고 제네릭한 코드로 타입별 기능을 사용할 수 있다.
  - 타입 클래스 패턴이라 부르며, Haskell의 TypeClass에서 파생

TOUR OF SCALA | Operator

Implicit Conversions in Scala

SCALA 3 — BOOK | Type Class

Type Classes. Scala의 Implicit 마법의 결정체

Type Classes in Scala and Haskell

F#: Statically Resolved Type Parameters

컴파일 시간에 실제 타입이 정해지는 타입 파라미터, 제네릭은 런타임 시점에서 결정

```
fsi> let inline add a b = a + b ;;
val inline add:
  a: ^a -> b: ^b -> 'c when (^a or ^b) : (static member (+) : ^a * ^b -> 'c)
```

```
fsi> add 4 2 ;;
val it: int = 6

fsi> add -2.0  4.7 ;;
val it: float = 2.7

fsi> add "123" "가나다" ;;
val it: string = "123가나다"
```

- 타입 파라미터가 특정 멤버를 가지게 제약할 수 있음
- C++의 Function Template과 비슷하지만 인라인 함수에만 쓸 수 있음

Statically Resolved Type Parameters

[C++] 함수 템플릿(Function Template)과 템플릿 함수(Template Function)

`double` in .net6.0 vs .net7.0

.net6.0

```
struct Double :
  IComparable,
  IComparable<Double>,
  IConvertible,
  IEquatable<Double>,
  ISpanFormattable,
  IFormattable
{
  ...
}
```

.net7.0

```
struct Double :
  IComparable,
  IComparable<Double>,
  IConvertible,
  IEquatable<Double>,
  ISpanFormattable,
  IFormattable,

  IParsable<Double>,
  ISpanParsable<Double>,
  IAdditionOperators<Double, Double, Double>,
  IAdditiveIdentity<Double, Double>,
  IBinaryFloatingPointIeee754<Double>,
  IBinaryNumber<Double>,
  IBitwiseOperators<Double, Double, Double>,
  IComparisonOperators<Double, Double, bool>,
  IEqualityOperators<Double, Double, bool>,
  IDecrementOperators<Double>,
  IDivisionOperators<Double, Double, Double>,
  IIncrementOperators<Double>,
  IModulusOperators<Double, Double, Double>,
  IMultiplicativeIdentity<Double, Double>,
```

🔥 1. 최대 추상화 충동을 유발

- static abstract members와 일반 수학은 공통적으로
  - 더 많은 추상화를 통해
  - 더 적은 코드로 더 넓은 범위를 커버하여
  - 재사용성의 증가

- 재사용성의 증가는
  - 매우 매력있고
  - 합당해보이나

- 실제로는
  - 재사용되는 양은 극히 적으며
  - 굉장한 시간의 낭비이며
  - 라이브러리 사용자로 하여금 학습과 사용에 복잡성을 높인다

🔥 2. 마이크로 인터페이스의 확산과 후속 요구

- 일반 수학의 표현법과 활용을 다른 분야에서 적용하길 바랄 것이고
  - 그 분야(라이브러리, 프레임워크)들은 최대 추상화를 구현하기 위한 리소스를 소모해야할 것

- 인터페이스는 점점 세분화될 것이며
  - 나누어진 인터페이스를 이해하는 것은 사용자 개개인의 시간과 노력으로 지불됨

🔥 3. 끊나지 않는 **적합한** 일반화 지점 찾기

- 추상화의 정도는 절대로 적합한 지점을 찾을 수 없으며
- 항상 비생산적인 논쟁을 불러일으킬 것이며
- 소프트웨어 엔지니어링의 다른 합리적인 목표가 무시될 위험이 있다

A. 타입 제약이 아닌 타입으로 사용

하면 안됨

```
// The type 'T' cannot be used as type parameter 'TSelf' in the generic type or method 'INumber<TSelf>'.
// There is no boxing conversion or type parameter conversion from 'T' to 'System.Numerics.INumber<T>'.
// csharp(CS0314)

// Operator '+' cannot be applied to operands of type 'INumber<T>' and 'INumber<T>'
// csharp(CS0019)

public static INumber<T> Add<T>(INumber<T> left, INumber<T> right) => left + right; // 💥error
```

```
interface INumber<TSelf> : ... IAdditionOperators<TSelf, TSelf, TSelf> ... where TSelf : INumber<TSelf>
{}

interface IAdditionOperators<TSelf, TOther, TResult> where TSelf : IAdditionOperators<TSelf, TOther, TResult>?
{
  static abstract TResult operator +(TSelf left, TOther right);
}
```

- `INumber`의 `T`가 가져야하는 타입 제약 `TSelf: INumber<TSelf>` 을 만족하지 못하기 때문
- 모르면 맞는 규칙 추가 *(`TSelf`는 어디서 나온건데)*

A. 타입 제약이 아닌 타입으로 사용하면 안됨

```
1   public static INumber<T> Add<T>(T left, T right) where T : INumber<T> => left + right; // ⭕ work!
```

```
1   void doNumericThings(IFavorite t1, IFavorite t2)
2   {
3     // A static virtual or abstract interface member can be accessed only on a type parameter.
4     // csharp(CS8926)
5     var sizeAtAge = IFavorite.SizeAtAge(2); // 💥error
6   }
7
8   void doNumericThings<T>(T t1, T t2) where T : IFavorite
9   {
10    var sizeatage = T.SizeAtAge(2); // ⭕ work!
11  }
```

- 복잡한 인터페이스 선언이 아니더라도 타입 파라미터를 통해서만 접근이 가능하도록 설정되어 있음

B. 고차함수가 더 간단하고 일반적일 수 있음

```fsharp
type ISomeFunctionality<'T when 'T :> ISomeFunctionality<'T>> =
    static abstract DoSomething: 'T -> 'T

let SomeGenericThing<'T when 'T :> ISomeFunctionality<'T>> (arg: 'T) =
    //...
    'T.DoSomething(arg)
    //...

type MyType1 =
    interface ISomeFunctionality<MyType1> with
        static member DoSomething(x) = ...

type MyType2 =
    static member DoSomethingElse(x) = ...

SomeGenericThing<MyType1> arg1
SomeGenericThing<MyType2> arg2 // oh no, MyType2 doesn't have the interface! Stuck!
```

- 정적멤버인터페이스를 구현한 타입 안에서의 일반화

B. 고차함수가 더 간단하고 일반적일 수 있음

```
fsi> let SomeGenericThing doSomething arg =
       doSomething arg

val SomeGenericThing: doSomeThing: ('a -> 'b) -> arg: 'a -> 'b
```

```
type MyType1 =
    static member DoSomething(x) = ...

type MyType2 =
    static member DoSomethingElse(x) = ...

SomeGenericThing MyType1.DoSomething arg1
SomeGenericThing MyType2.DoSomethingElse arg2
```

- 함수 전달이 더 짧은 구현이며 더 일반적임
- 추상화 메소드를 구현한 수가 10개 미만이라면, 고차함수를 사용하라

C. 닫힌 연산에만 사용하세요

- 정적 메소드가 가지는 한계
  - 계산에 영향을 주는 정보는 파라미터 안에만 존재
    - 암시적인 컨텍스트 (인스턴스 멤버)
    - 전역 상태를 쉽게 사용할 수 없음
  - 특정 정보와 함께 인스턴스화 될 수 없음 (`ParserV1`, `ParserV2`)

- 한계로 인한 협소한 적용 범위
  - 연산이 내부적으로 닫혀 있으며
  - 구현 내용에 반박의 여지가 *(미래에도)* 없는
  - 영역에서만 사용해야한다

요구사항의 변경이 존재하거나 컨텍스트에 의존해야하는 도메인 영역에서는 절대 사용하지 말 것

RFC의 지침

- static abstract members가 가지는 고유의 제약을 이해하세요
- 유형 분류 & 최대 추상화 충동에 지지마세요
- 논쟁과 변경의 여지가 없는 형식에만 사용하세요
- 정적 인터페이스와 비정적 인터페이스를 섞어쓰지마세요

들은 생각

- 작성하기도, 쓰기도, 쉽지 않은 기능
- 함께 일하기

  1. 이견이 없는 일반 수학 라이브러리를 만들 수 있을까?
  2. 만들어진 라이브러리(복잡성의 증가)를 팀이 이해하고 활용해줄 수 있을까?

- .NET7은 LTS가 아닌데 .NET8가 나올 시점에는 커뮤니티에서 어떻게 사용되고 있을까
- Java랑 C# 볼수록 안 비슷하다...

▶ 이젠 AOT 컴파일인가?

▶ 닷넷에도 타입 논쟁이?

남은 의문

- IWSAM implementations are static
  - 제네릭을 사용하지만 정적인 구현?