FastEnum is SUPER fast !!

速さは正義

ちょっとした工夫で案外メチャクチャ速くなる！

速いコードを書くひとつの参考になれば幸いです

本当は FastEnum は存在しない方が良い

標準が速ければ最高だし、そうあってほしい

.NET Core 頑張れ


Inside FastEnum
.NET Conf in Tokyo 2019
鈴木 孝明


Inside FastEnum
.NET Conf in Tokyo 2019
鈴木 孝明

About

Name

鈴木 孝明 a.k.a @xin9le

Work

Application Engineer

Award

Microsoft MVP
for Developer Technologies

Web Site

https://xin9le.net

MVP Microsoft® Most Valuable Professional

sigma consulting

JAPAN


空港がない
陸の孤島
Remote Worker @福井

という話をしました

https://www.slideshare.net/xin9le/dotnetperformancetips-170268354

という話をしました

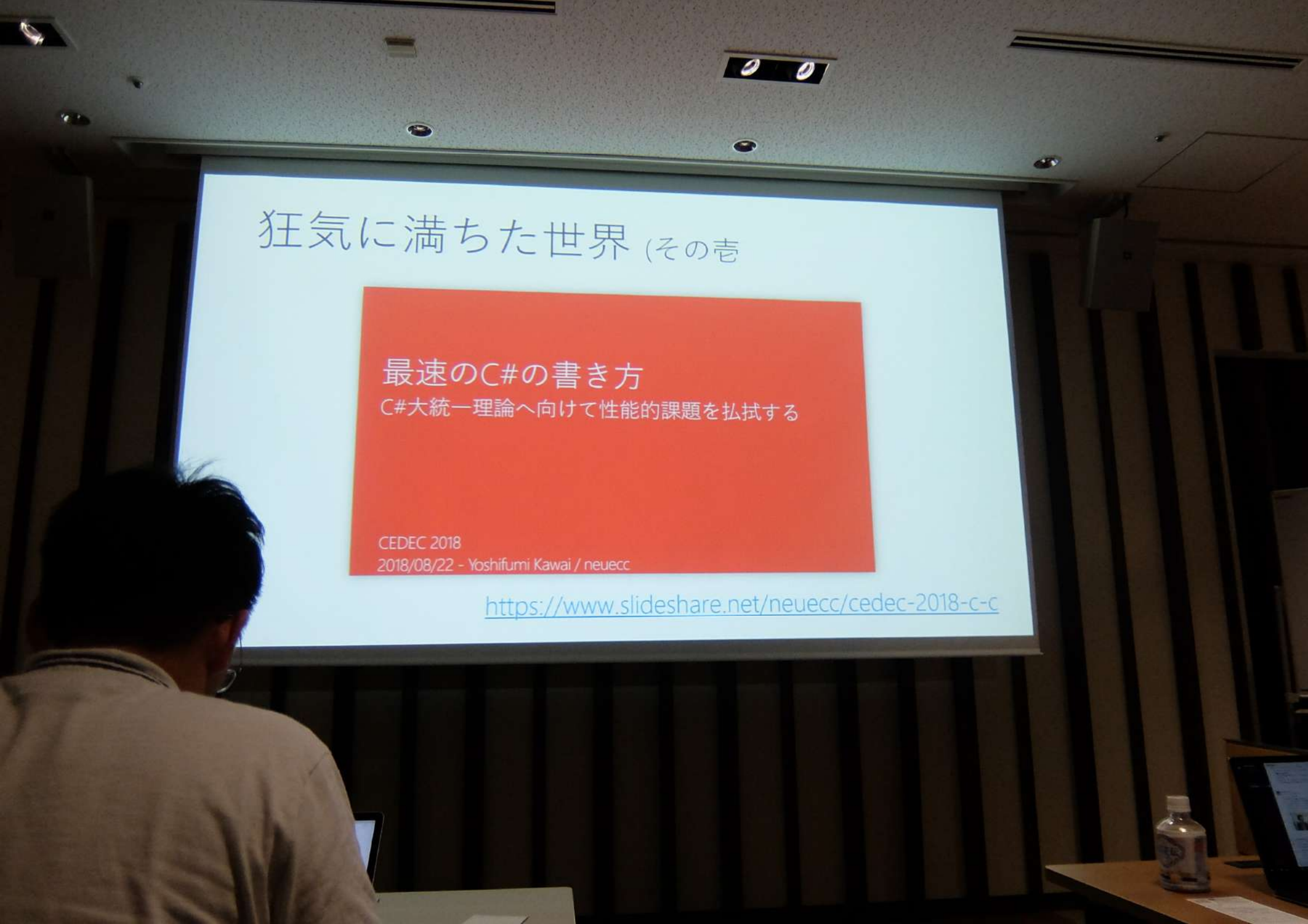
https://www.slideshare.net/xin9le/dotnetperformancetips-170268354

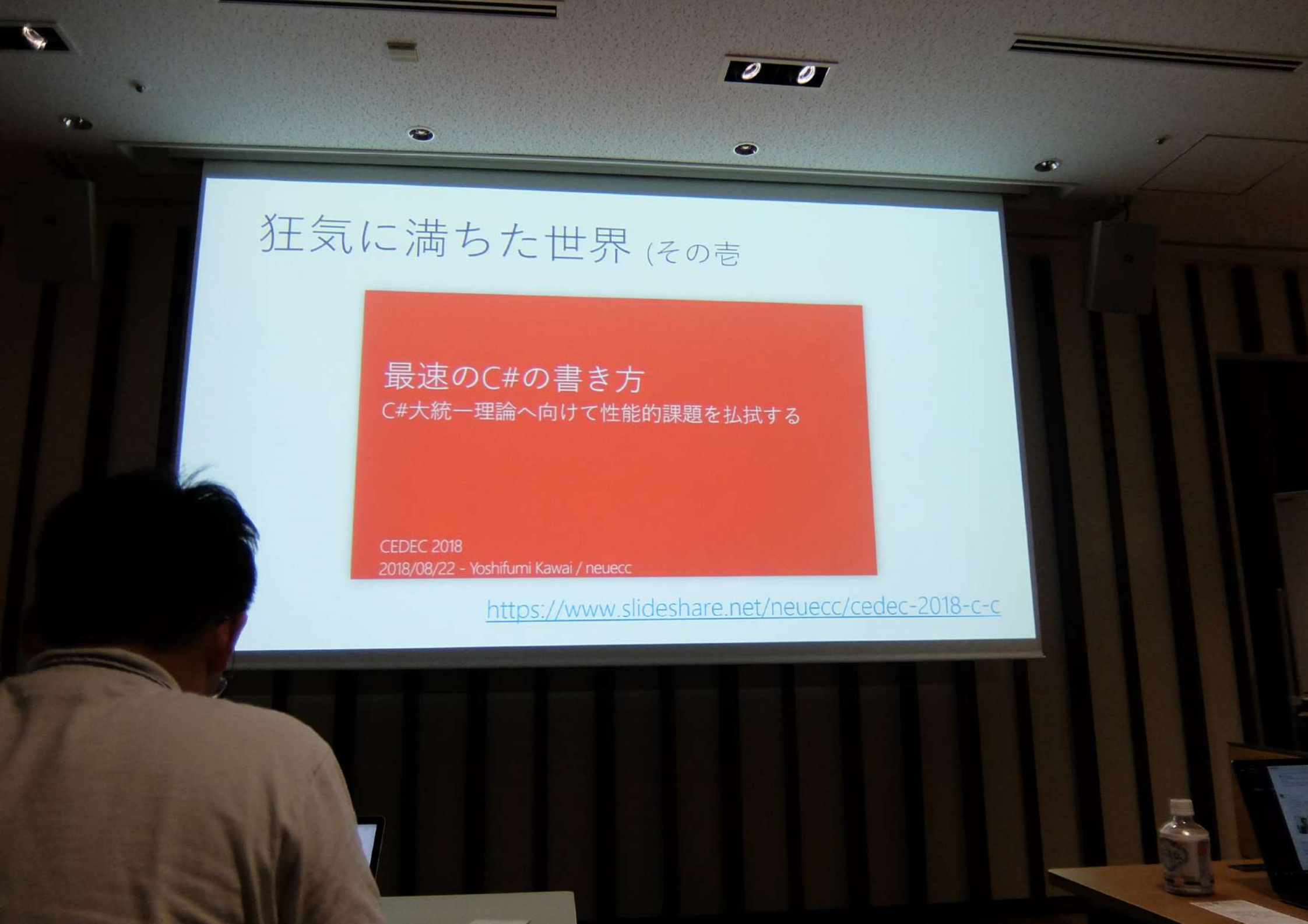
という話をしました

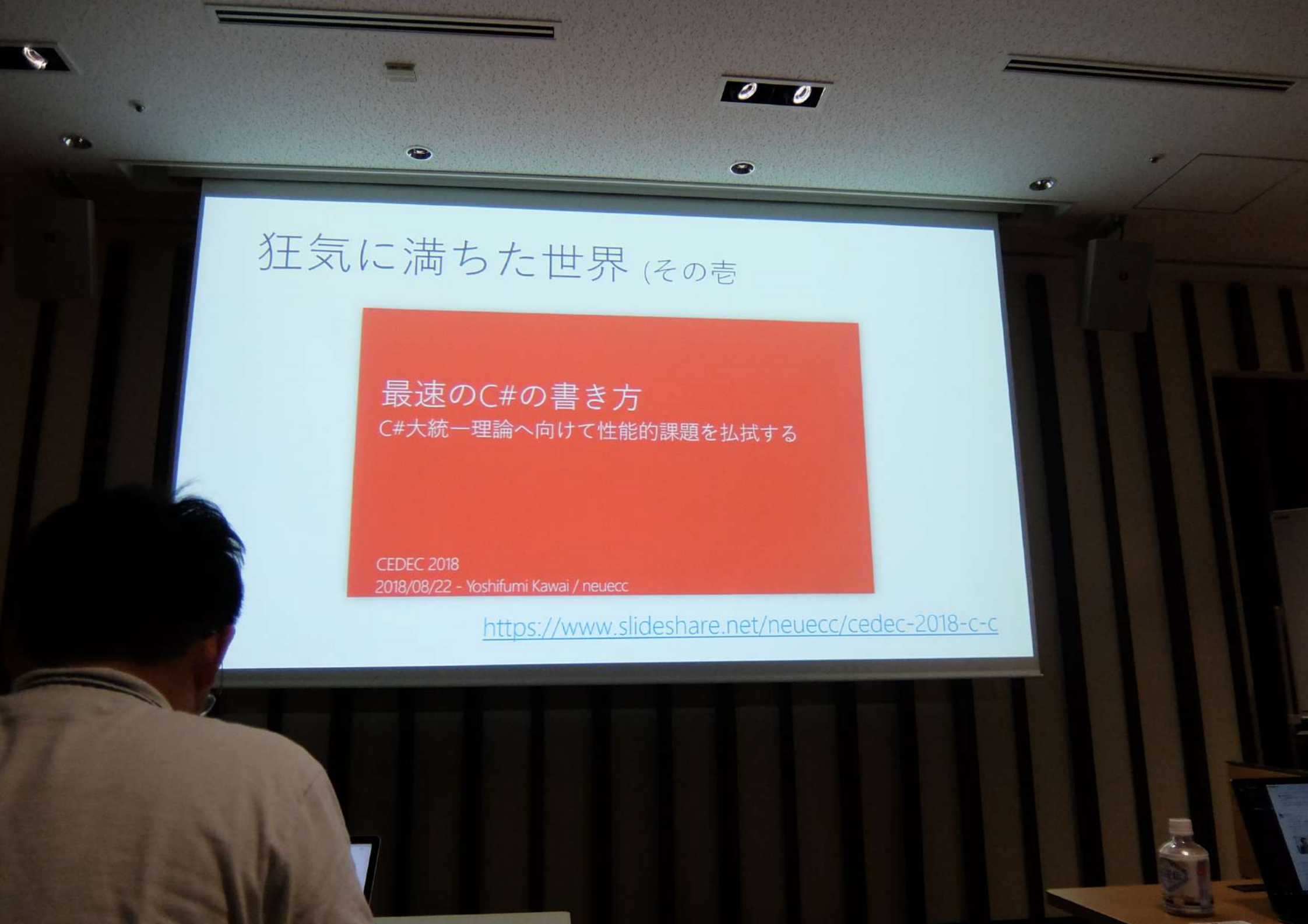
https://www.slideshare.net/xin9le/dotnetperformancetips-170268354

という話をしました

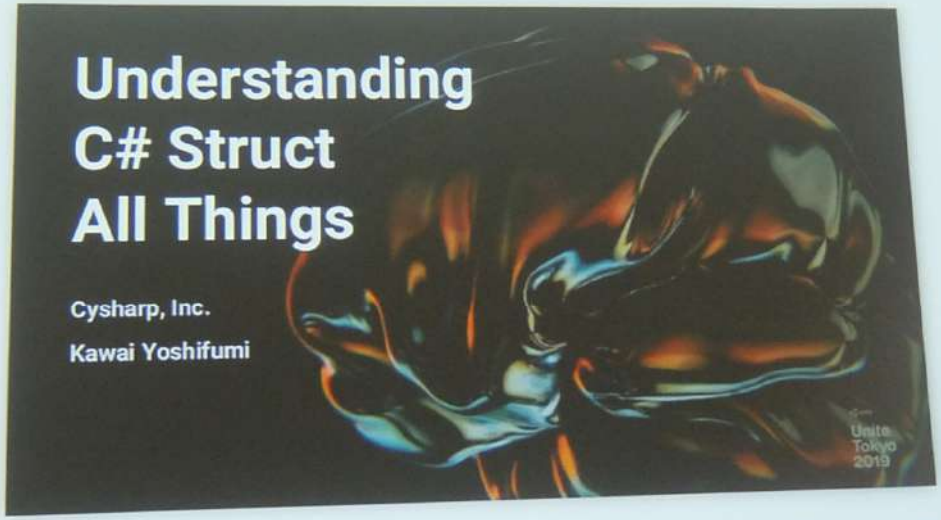
https://www.slideshare.net/xin9le/dotnetperformancetips-170268354


狂気に満ちた世界 (その壱
最速のC#の書き方
C#大統一理論へ向けて性能的課題を払拭する
CEDEC 2018
2018/08/22 - Yoshifumi Kawai / neuecc
https://www.slideshare.net/neuecc/cedec-2018-c-c


狂気に満ちた世界 (その壱
最速のC#の書き方
C#大統一理論へ向けて性能的課題を払拭する
CEDEC 2018
2018/08/22 - Yoshifumi Kawai / neuecc
https://www.slideshare.net/neuecc/cedec-2018-c-c

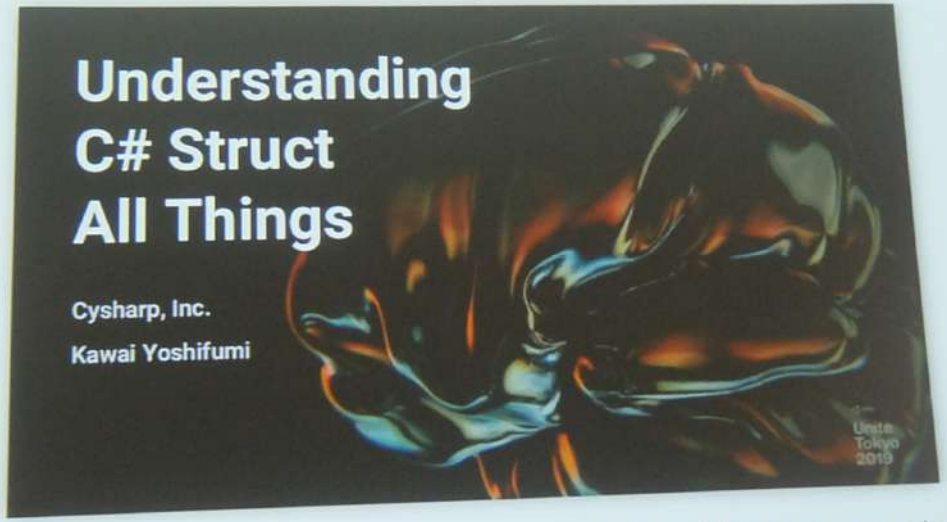

狂気に満ちた世界 (その壱
最速のC#の書き方
C#大統一理論へ向けて性能的課題を払拭する
CEDEC 2018
2018/08/22 - Yoshifumi Kawai / neuecc
https://www.slideshare.net/neuecc/cedec-2018-c-c

狂気に満ちた世界 (その弐



https://learning.unity3d.jp/3305/

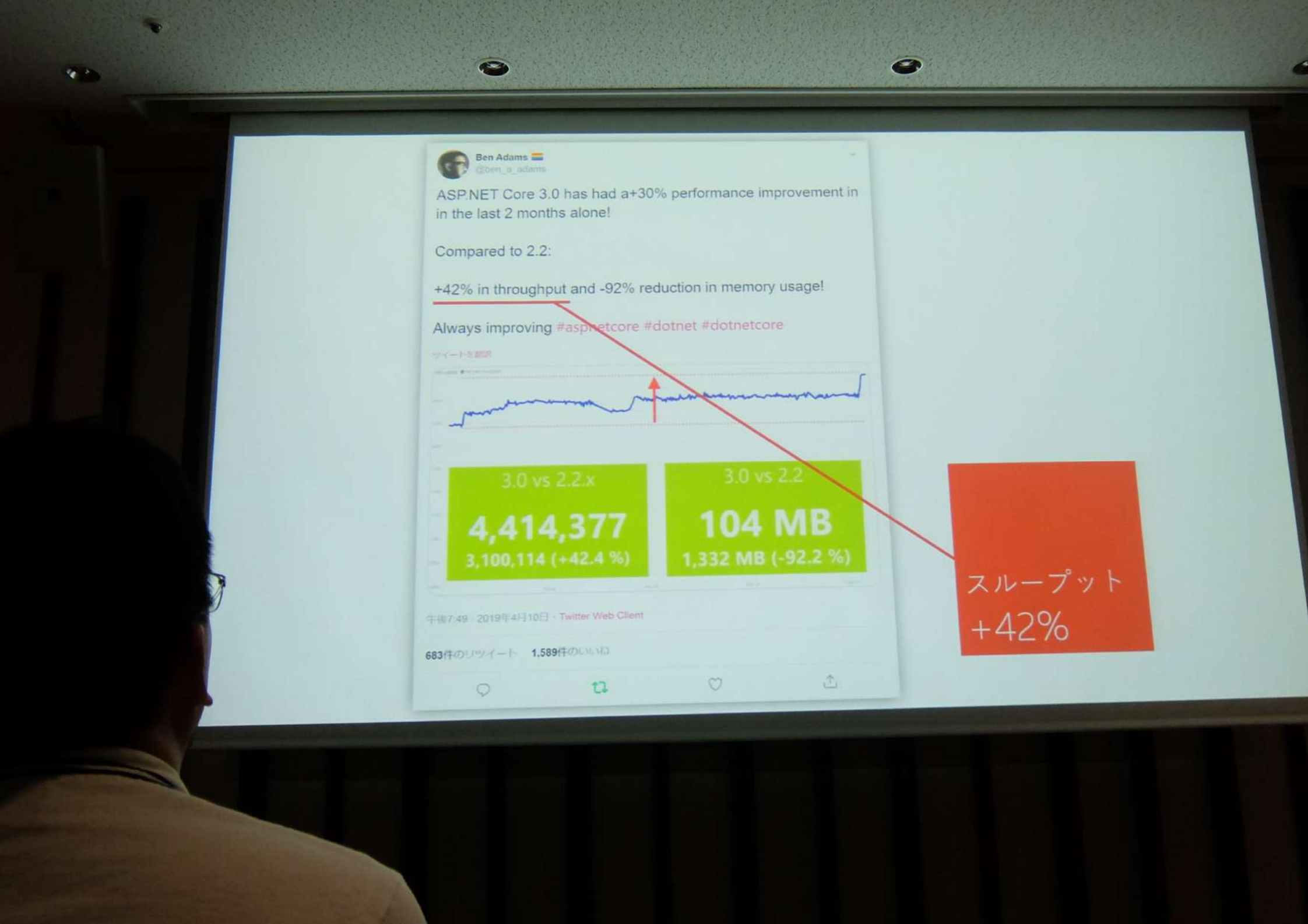
狂気に満ちた世界 (その弐

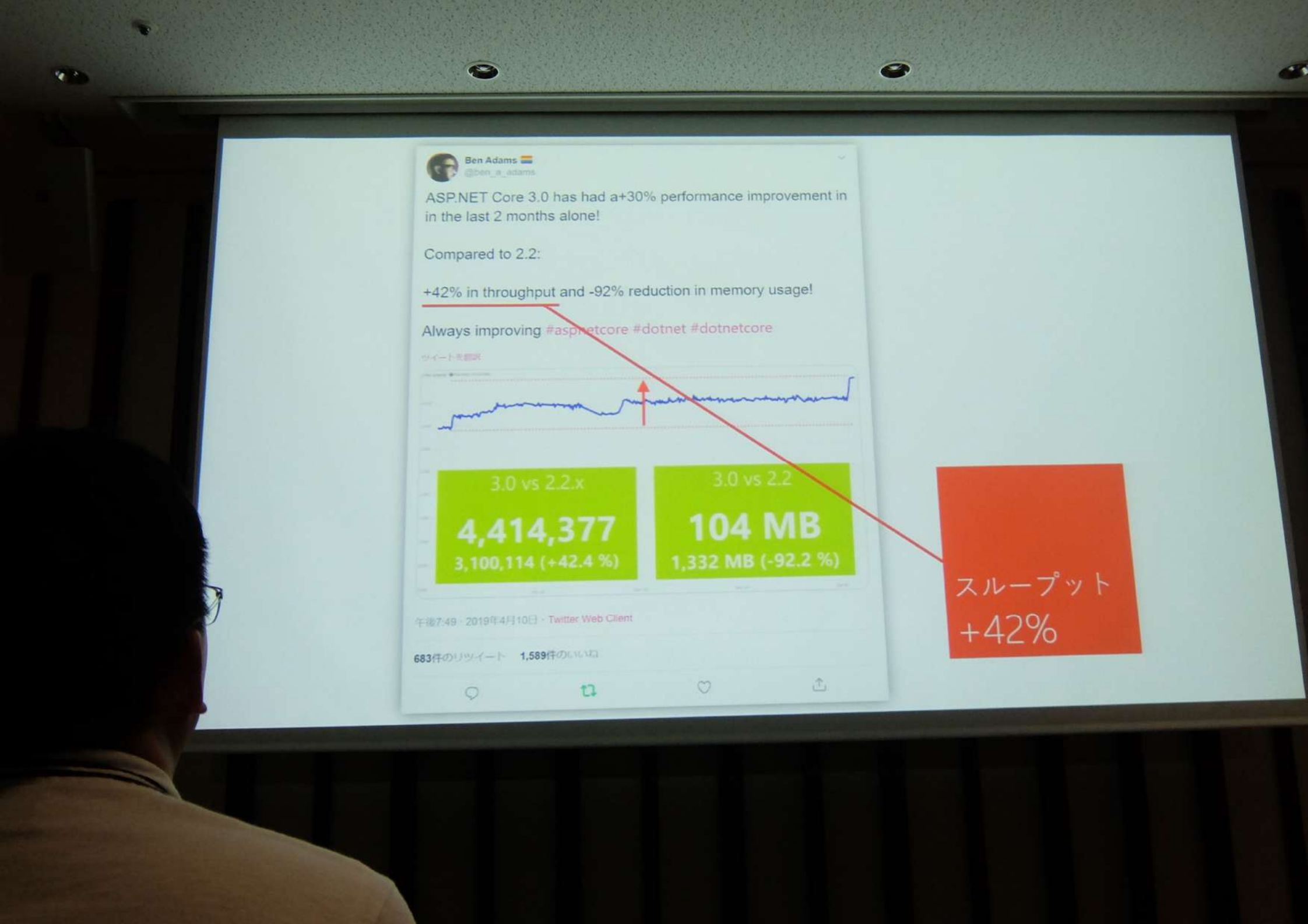


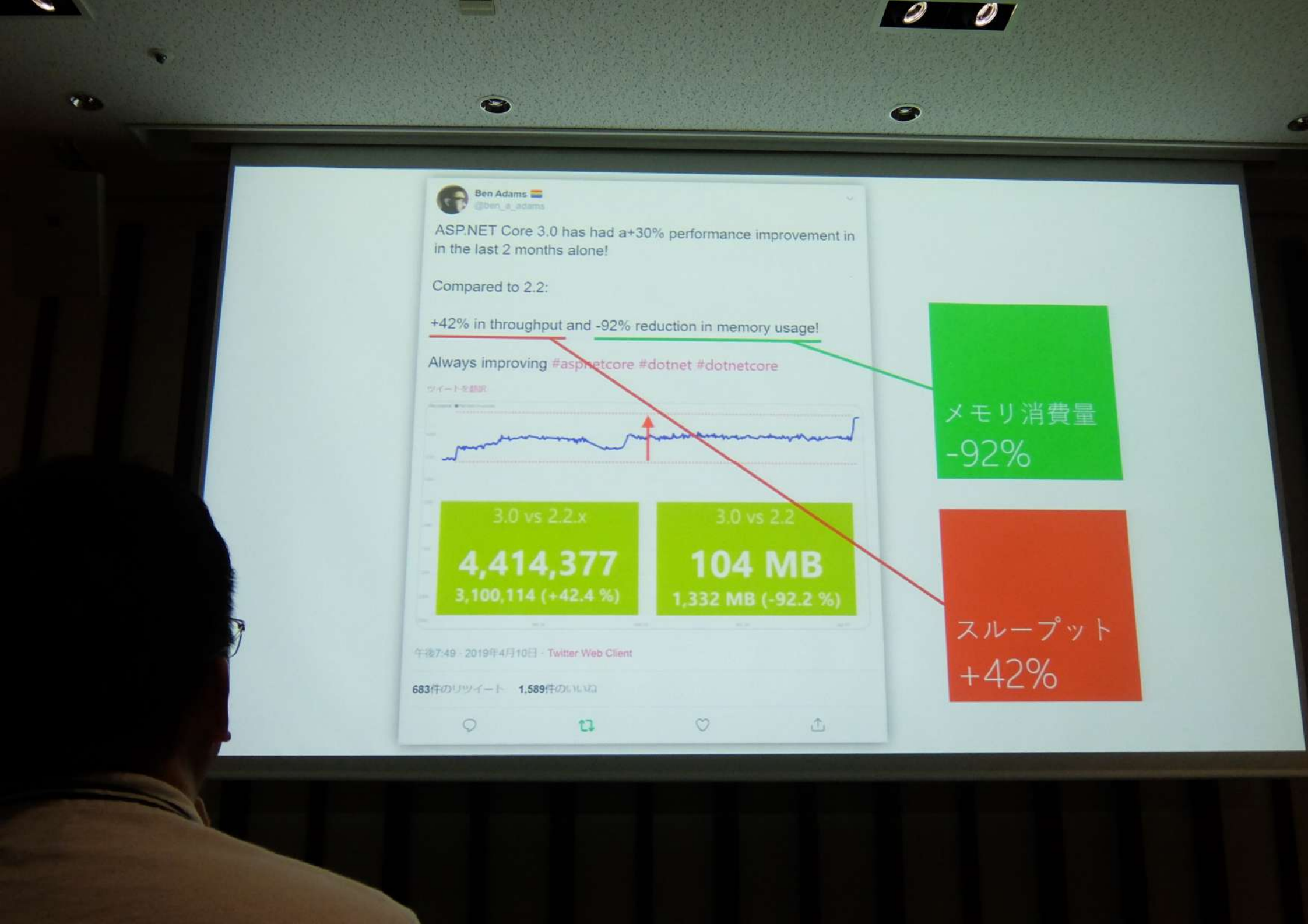
https://learning.unity3d.jp/3305/

狂気に満ちた世界 (その弐

https://learning.unity3d.jp/3305/


Ben Adams
@ben_a_adams
ASP.NET Core 3.0 has had a+30% performance improvement in in the last 2 months alone!
Compared to 2.2:
+42% in throughput and -92% reduction in memory usage!
Always improving #aspnetcore #dotnet #dotnetcore
3.0 vs 2.2.x
4,414,377
3,100,114 (+42.4 %)
3.0 vs 2.2
104 MB
1,332 MB (-92.2 %)
午後7:49 · 2019年4月10日 · Twitter Web Client
683件のリツイート 1,589件のいいね
スループット
+42%

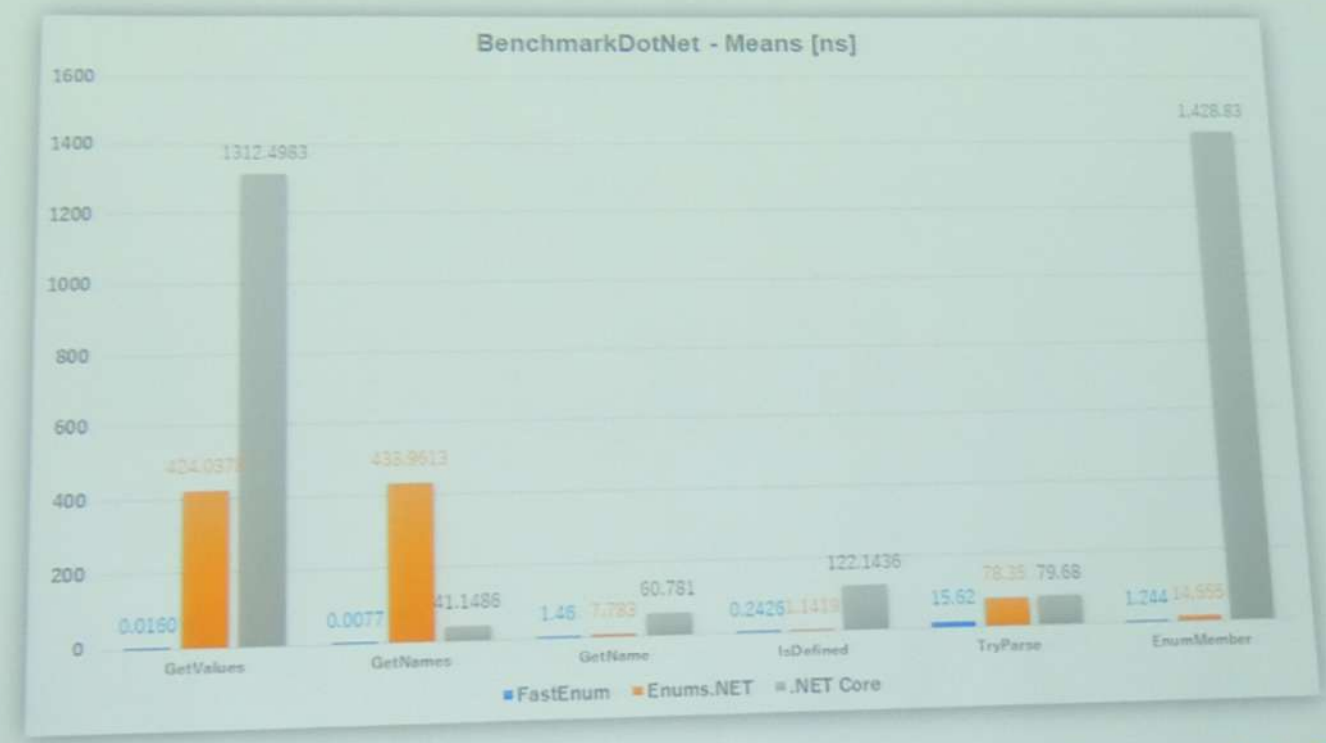

Ben Adams
ASP.NET Core 3.0 has had a+30% performance improvement in in the last 2 months alone!
Compared to 2.2:
+42% in throughput and -92% reduction in memory usage!
Always improving #aspnetcore #dotnet #dotnetcore
3.0 vs 2.2.x
4,414,377
3,100,114 (+42.4 %)
3.0 vs 2.2
104 MB
1,332 MB (-92.2 %)
午後7:49 · 2019年4月10日 · Twitter Web Client
683件のリツイート
1,589件のいいね
スループット
+42%


Ben Adams
@ben_a_adams
ASP.NET Core 3.0 has had a+30% performance improvement in in the last 2 months alone!
Compared to 2.2:
+42% in throughput and -92% reduction in memory usage!
Always improving #aspnetcore #dotnet #dotnetcore
3.0 vs 2.2.x
4,414,377
3,100,114 (+42.4 %)
3.0 vs 2.2
104 MB
1,332 MB (-92.2 %)
午後7:49 · 2019年4月10日 · Twitter Web Client
683件のリツイート 1,589件のいいね
メモリ消費量
-92%
スループット
+42%

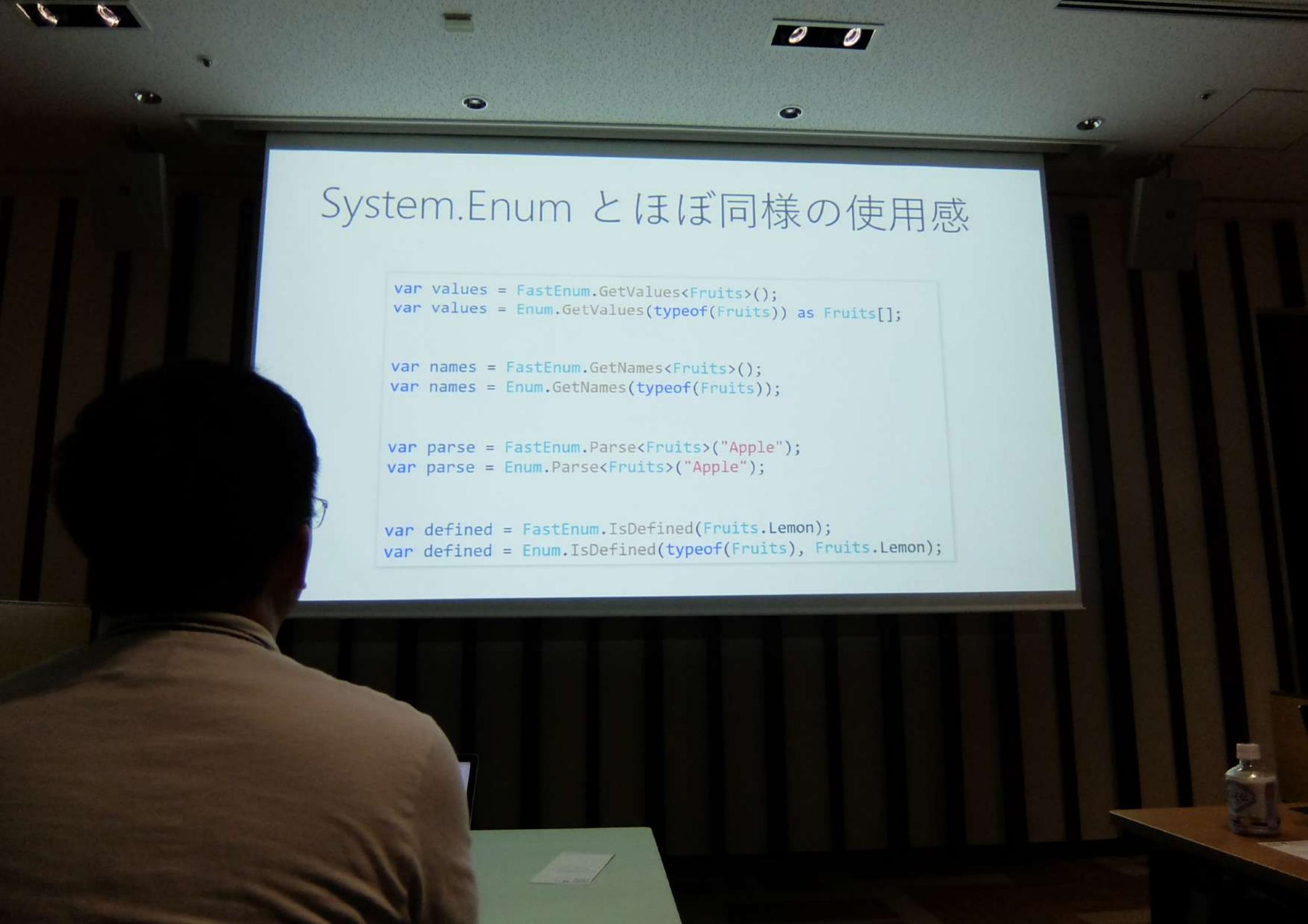

Microsoft | .NET Blog
Product Blogs
DevOps
Languages
.NET
Platform Development
Data Development
Performance Improvements in .NET Core 3.0
May 15th, 2019
Back when we were getting ready to ship .NET Core 2.0, I wrote a blog post exploring some of the many performance improvements that had gone into it. I enjoyed putting it together so much and received such a positive response to the post that I did it again for .NET Core 2.1, a version for which performance was also a significant focus. With //build last week and .NET Core 3.0's release now on
超絶長編記事も公開
in this post, we'll take a tour through some of the many improvements, big and small, that have gone into the .NET Core runtime and core libraries in order to make your apps and services leaner and faster.

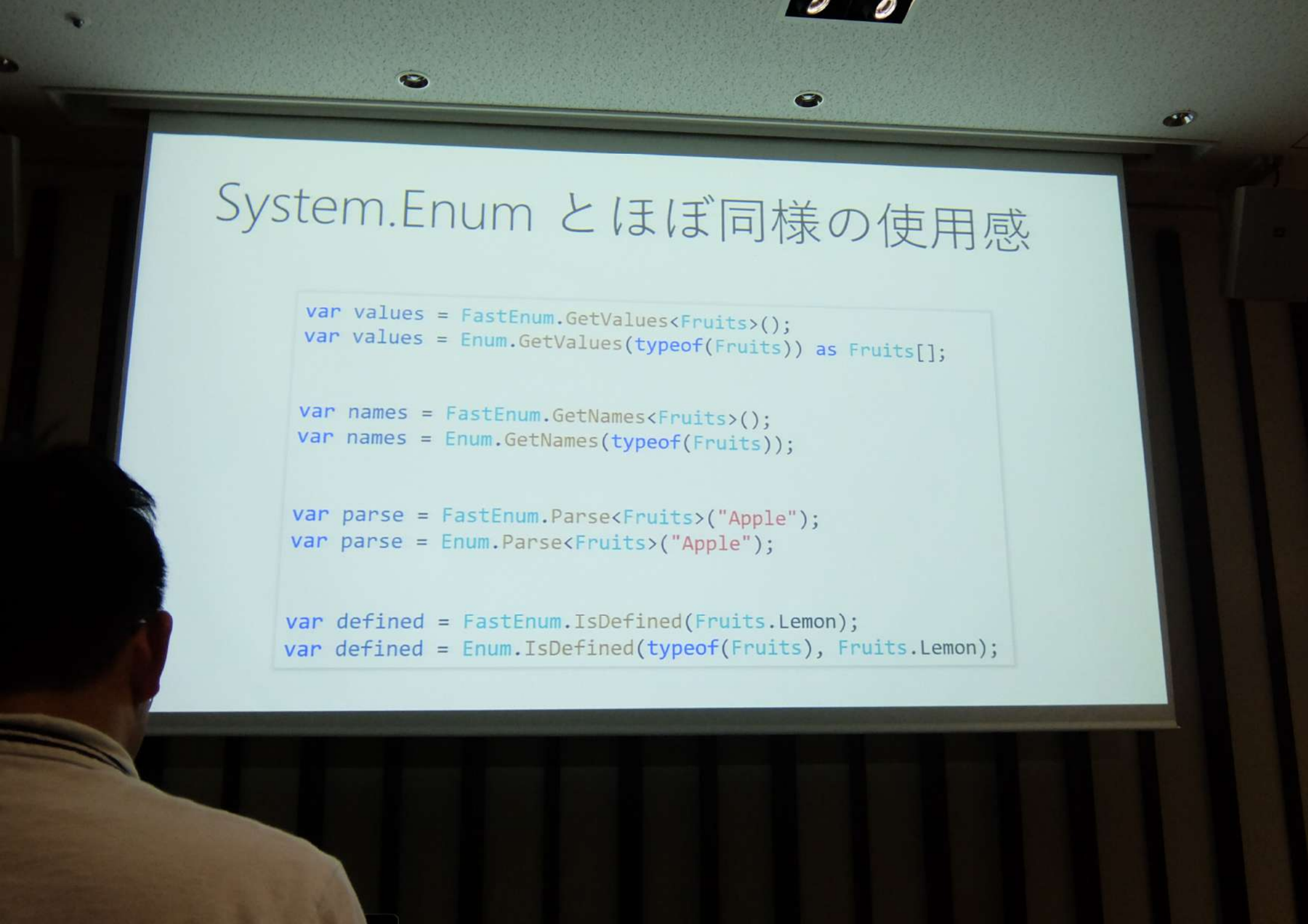

Microsoft | .NET Blog
Product Blogs
DevOps
Languages
.NET
Platform Development
Data Development
Performance Improvements in .NET Core 3.0
May 15th, 2019
Back when we were getting ready to ship .NET Core 2.0, I wrote a blog post exploring some of the many performance improvements that had gone into it. I enjoyed putting it together so much and received such a positive response to the post that I did it again for .NET Core 2.1, a version for which performance was also a significant focus. With //build last week and .NET Core 3.0's release now on
超絶長編記事も公開
into the .NET Core runtime and core libraries in order to make your apps and services leaner and faster.

The world fastest Enum (FastEnum)



https://github.com/xin9le/FastEnum


System.Enum とほぼ同様の使用感
var values = FastEnum.GetValues<Fruits>();
var values = Enum.GetValues(typeof(Fruits)) as Fruits[];
var names = FastEnum.GetNames<Fruits>();
var names = Enum.GetNames(typeof(Fruits));
var parse = FastEnum.Parse<Fruits>("Apple");
var parse = Enum.Parse<Fruits>("Apple");
var defined = FastEnum.IsDefined(Fruits.Lemon);
var defined = Enum.IsDefined(typeof(Fruits), Fruits.Lemon);


System.Enum とほぼ同様の使用感
var values = FastEnum.GetValues<Fruits>();
var values = Enum.GetValues(typeof(Fruits)) as Fruits[];
var names = FastEnum.GetNames<Fruits>();
var names = Enum.GetNames(typeof(Fruits));
var parse = FastEnum.Parse<Fruits>("Apple");
var parse = Enum.Parse<Fruits>("Apple");
var defined = FastEnum.IsDefined(Fruits.Lemon);
var defined = Enum.IsDefined(typeof(Fruits), Fruits.Lemon);

Member<T>

フィールドの値/名前などを一纏めに

値/名前のペアって案外使うのに、ペアを作る処理は地味に面倒

```
// フィールド情報

public sealed class Member<T>
    where T : struct, Enum
{
    public T Value { get; }
    public string Name { get; }
    public FieldInfo FieldInfo { get; }
    public EnumMemberAttribute EnumMemberAttribute { get; }
}
```

```
// 超高速で簡単な取得

var xs = FastEnum.GetMembers<Fruits>();
var x1 = FastEnum.GetMember(Fruits.Apple);
var x2 = Fruits.Apple.ToMember();
```

Member<T>

フィールドの値/名前などを一纏めに

値/名前のペアって案外使うのに、ペアを作る処理は地味に面倒

```
// フィールド情報

public sealed class Member<T>
    where T : struct, Enum
{
    public T Value { get; }
    public string Name { get; }
    public FieldInfo FieldInfo { get; }
    public EnumMemberAttribute EnumMemberAttribute { get; }
}
```

```
// 超高速で簡単な取得

var xs = FastEnum.GetMembers<Fruits>();
var x1 = FastEnum.GetMember(Fruits.Apple);
var x2 = Fruits.Apple.ToMember();
```

Label 属性

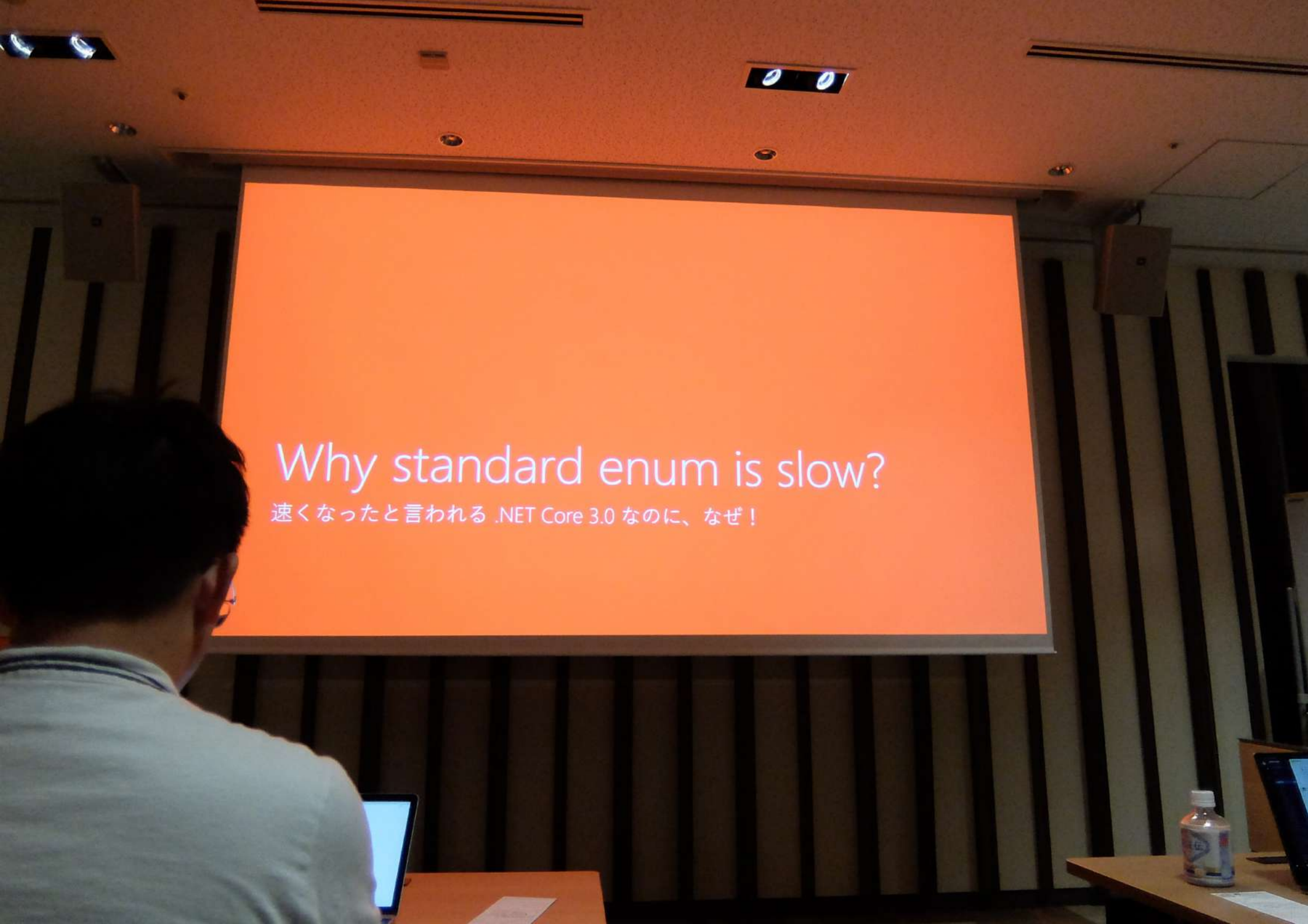
フィールドに複数の名前を付けられる

[EnumMember] だと AllowMultiple = false なので不便

Grani 社内で使われていた便利機能のひとつ

```
// こんな定義があったとして

public enum Fruits
{
    [Label("🍎", 0)]
    [Label("りんご", 1)]
    Apple,
}
```

```
// インデックス指定で取得できる

var a = Fruits.Apple.GetLabel(0); // 🍏
var b = Fruits.Apple.GetLabel(1); // りんご
```


Why standard enum is slow?
速くなったと言われる .NET Core 3.0 なのに、なぜ！

中途半端なキャッシュ

RuntimeType.GenericCache を使ってる

そこまでは良いのだけど、キャッシュの持ち方がよくない

`ulong[]` / `string[]` を持つだけで、それ以上のトリックがない

都度 Reflection している箇所も

`Enum.GetName` / `Enum.IsDefined` などなど

キャッシュとは何だったのか…

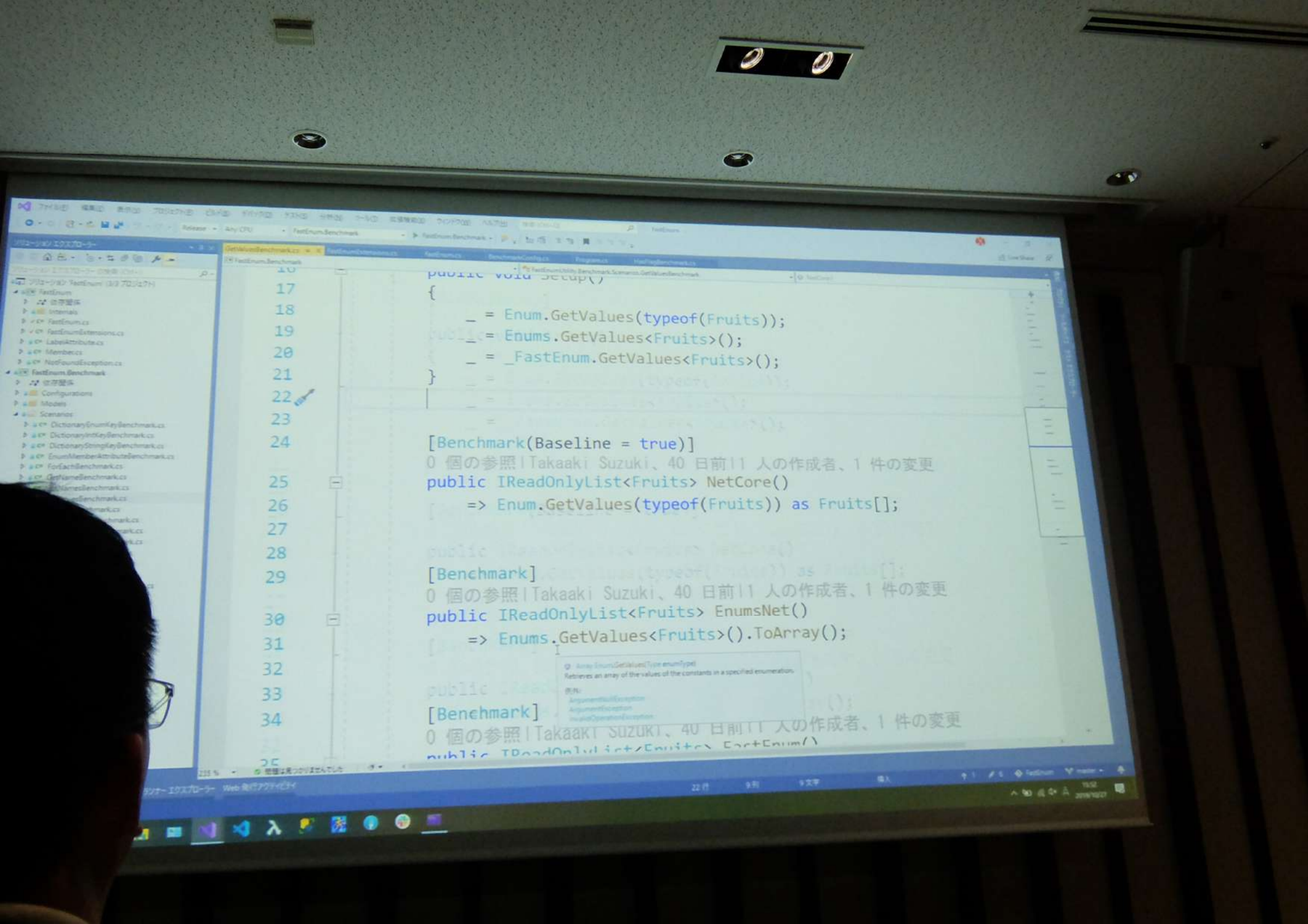
Box / Unbox の雨霰

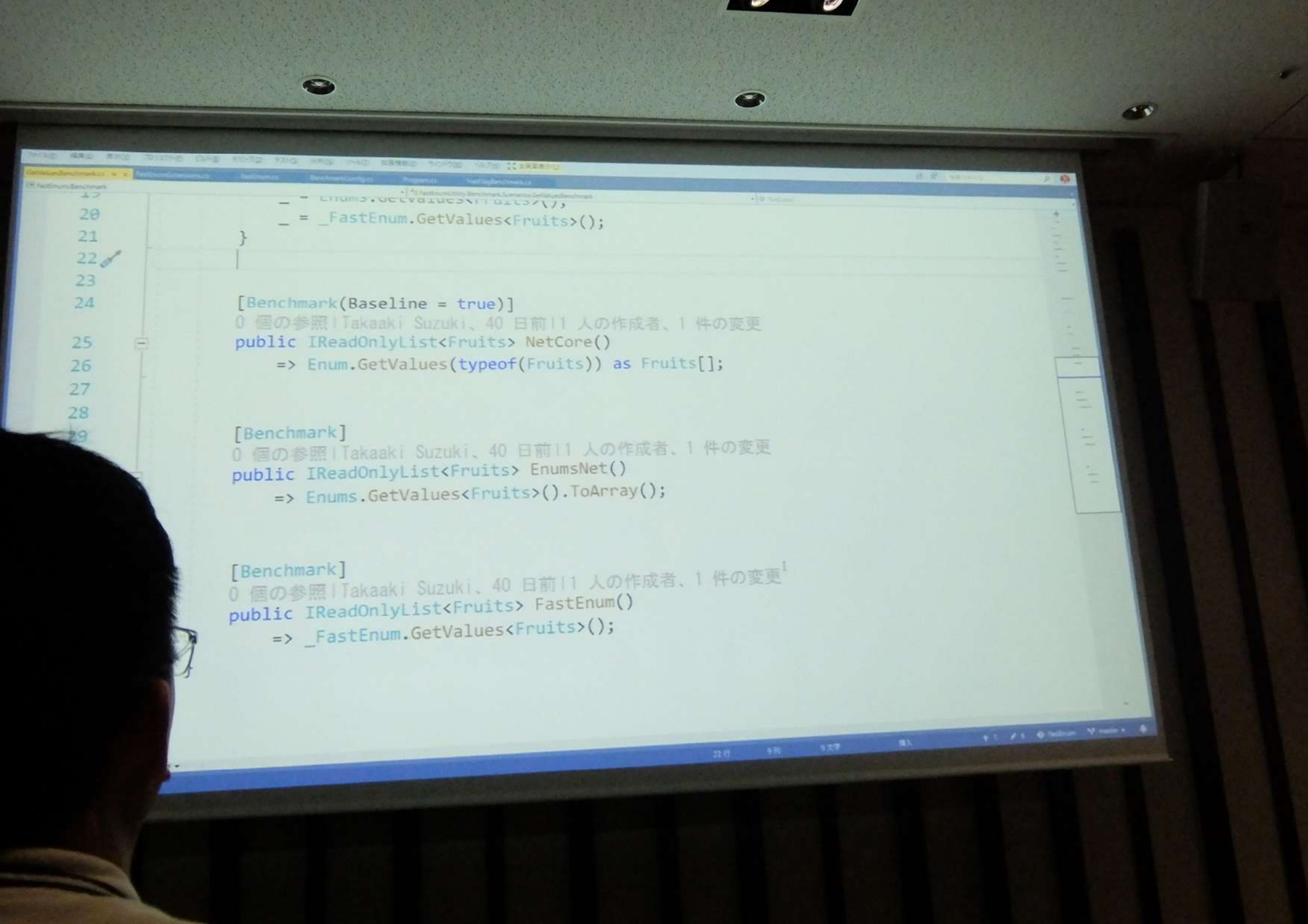
ほぼ必ず object 型を経由する

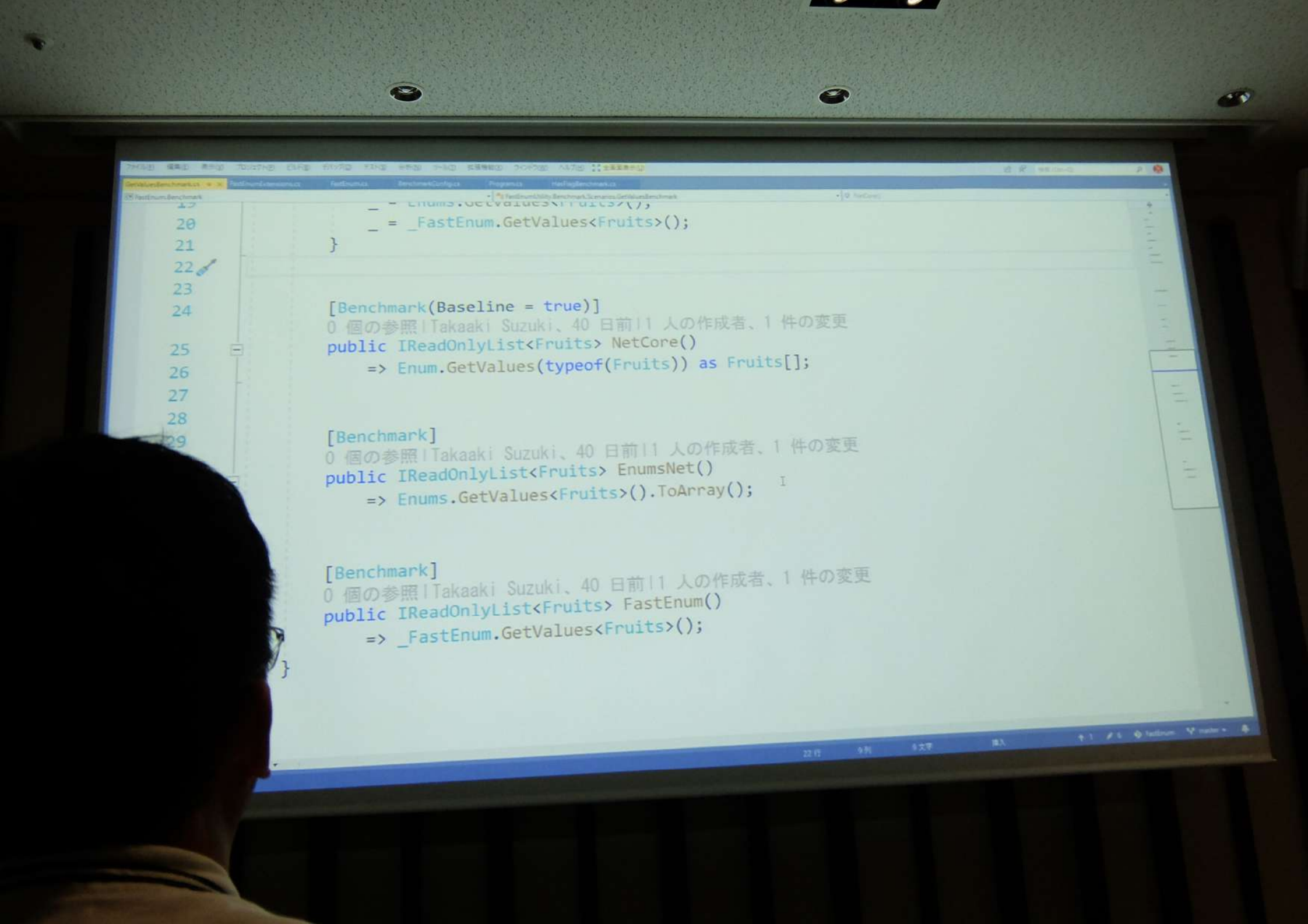
Non Generics な `Array` とか `Enum.ToObject` とかを普通に使う
`ulong` で持っていたものを有効活用しないとか超クール

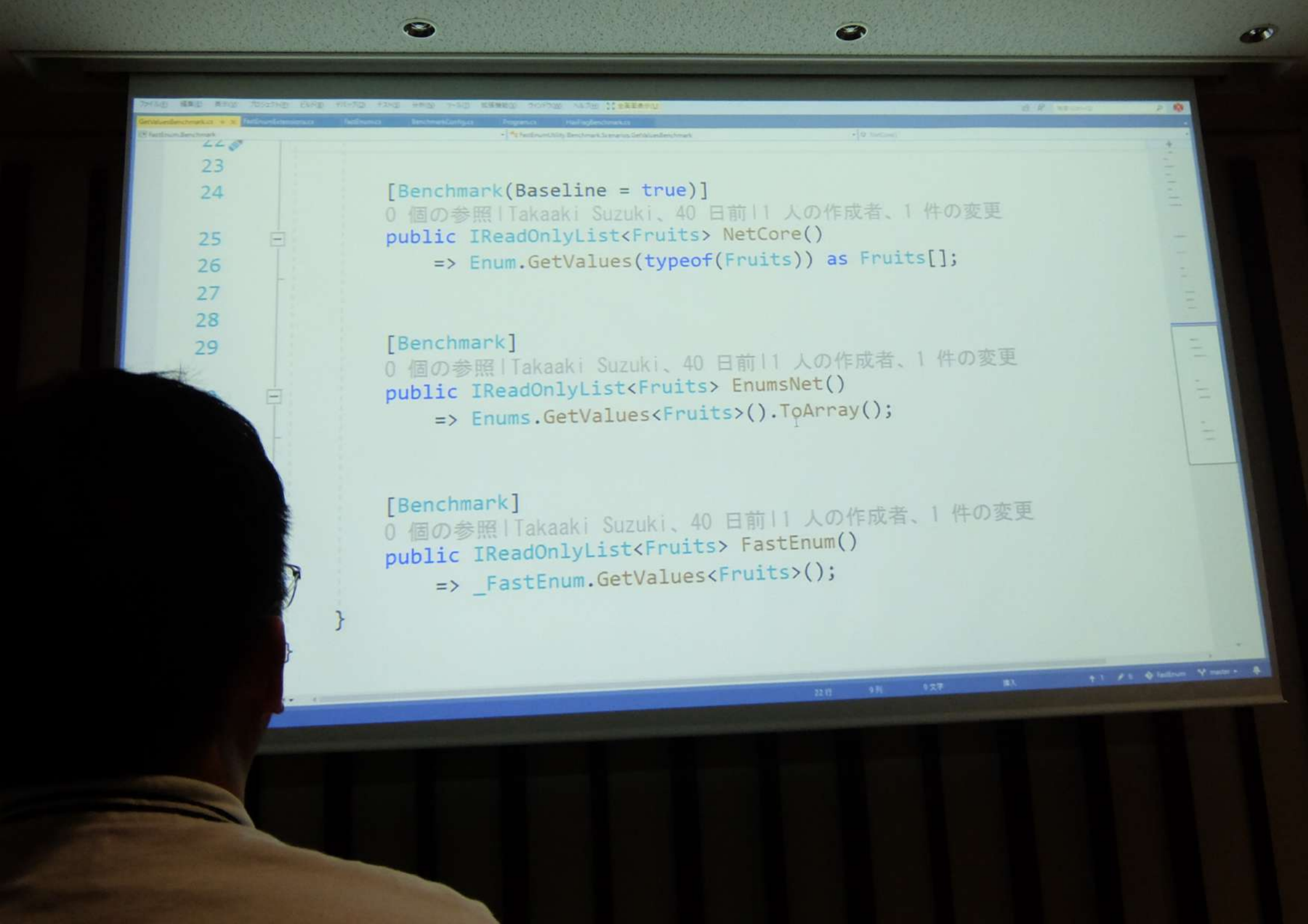
Parse<T> / TryParse<T> はまとも

`System.Enum` で Generics 版が使えるときは積極的に使う
※ .NET Core に限る
※ .NET Framework の実装は残念過ぎる

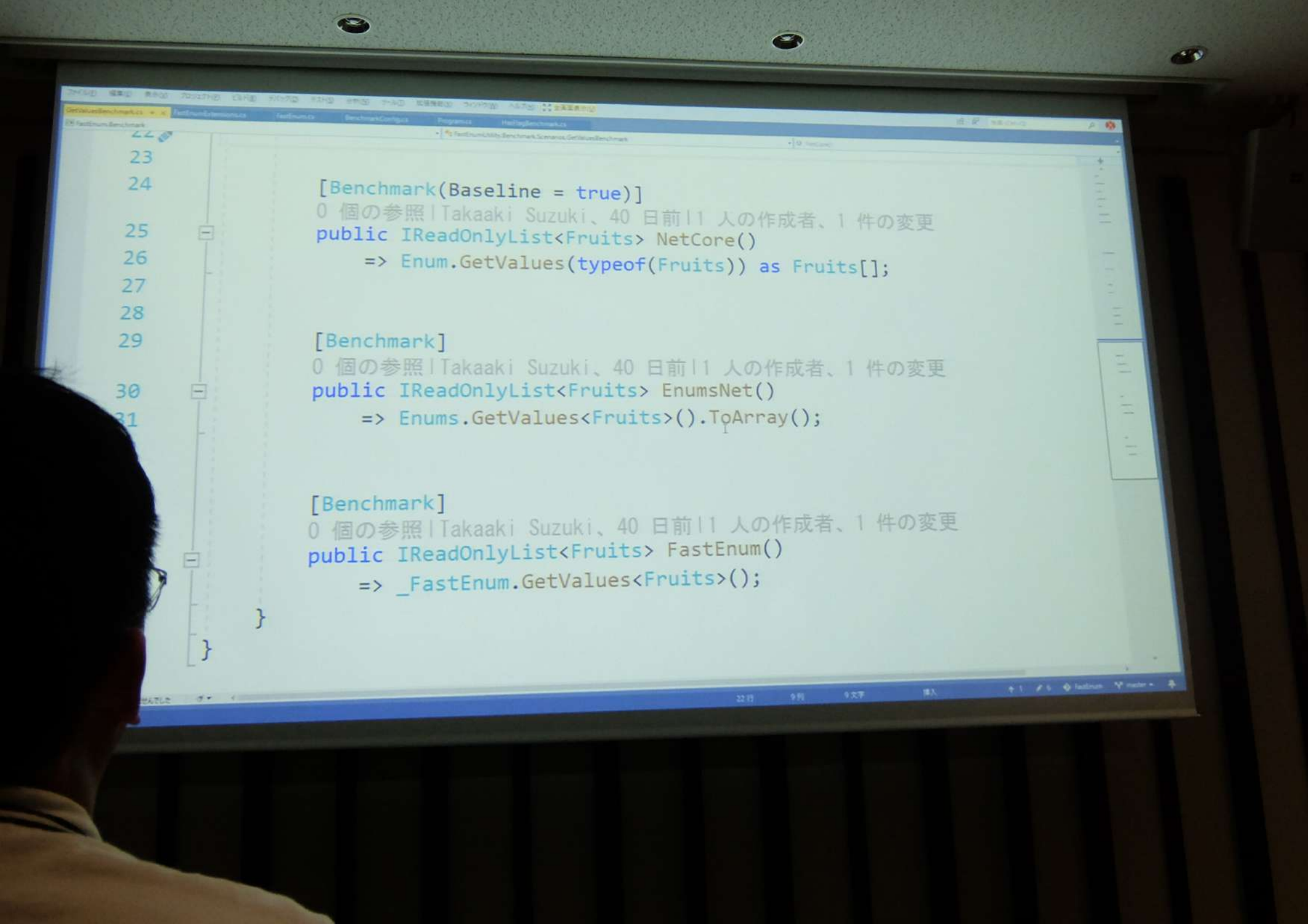

17 {
18 _ = Enum.GetValues(typeof(Fruits));
19 _ = Enums.GetValues<Fruits>();
20 _ = _FastEnum.GetValues<Fruits>();
21 }
22
23
24 [Benchmark(Baseline = true)]
0 個の参照 | Takaaki Suzuki、40 日前 | 1 人の作成者、1 件の変更
25 public IReadOnlyList<Fruits> NetCore()
26 => Enum.GetValues(typeof(Fruits)) as Fruits[];
27
28
29 [Benchmark]
0 個の参照 | Takaaki Suzuki、40 日前 | 1 人の作成者、1 件の変更
30 public IReadOnlyList<Fruits> EnumsNet()
31 => Enums.GetValues<Fruits>().ToArray();
32
33
34 [Benchmark]
0 個の参照 | Takaaki Suzuki、40 日前 | 1 人の作成者、1 件の変更


20 _ = _FastEnum.GetValues<Fruits>();
21 }
22
23
24 [Benchmark(Baseline = true)]
0 個の参照 | Takaaki Suzuki、40 日前 | 1 人の作成者、1 件の変更
25 public IReadOnlyList<Fruits> NetCore()
26 => Enum.GetValues(typeof(Fruits)) as Fruits[];
27
28
29 [Benchmark]
0 個の参照 | Takaaki Suzuki、40 日前 | 1 人の作成者、1 件の変更
public IReadOnlyList<Fruits> EnumsNet()
=> Enums.GetValues<Fruits>().ToArray();
[Benchmark]
0 個の参照 | Takaaki Suzuki、40 日前 | 1 人の作成者、1 件の変更
public IReadOnlyList<Fruits> FastEnum()
=> _FastEnum.GetValues<Fruits>();

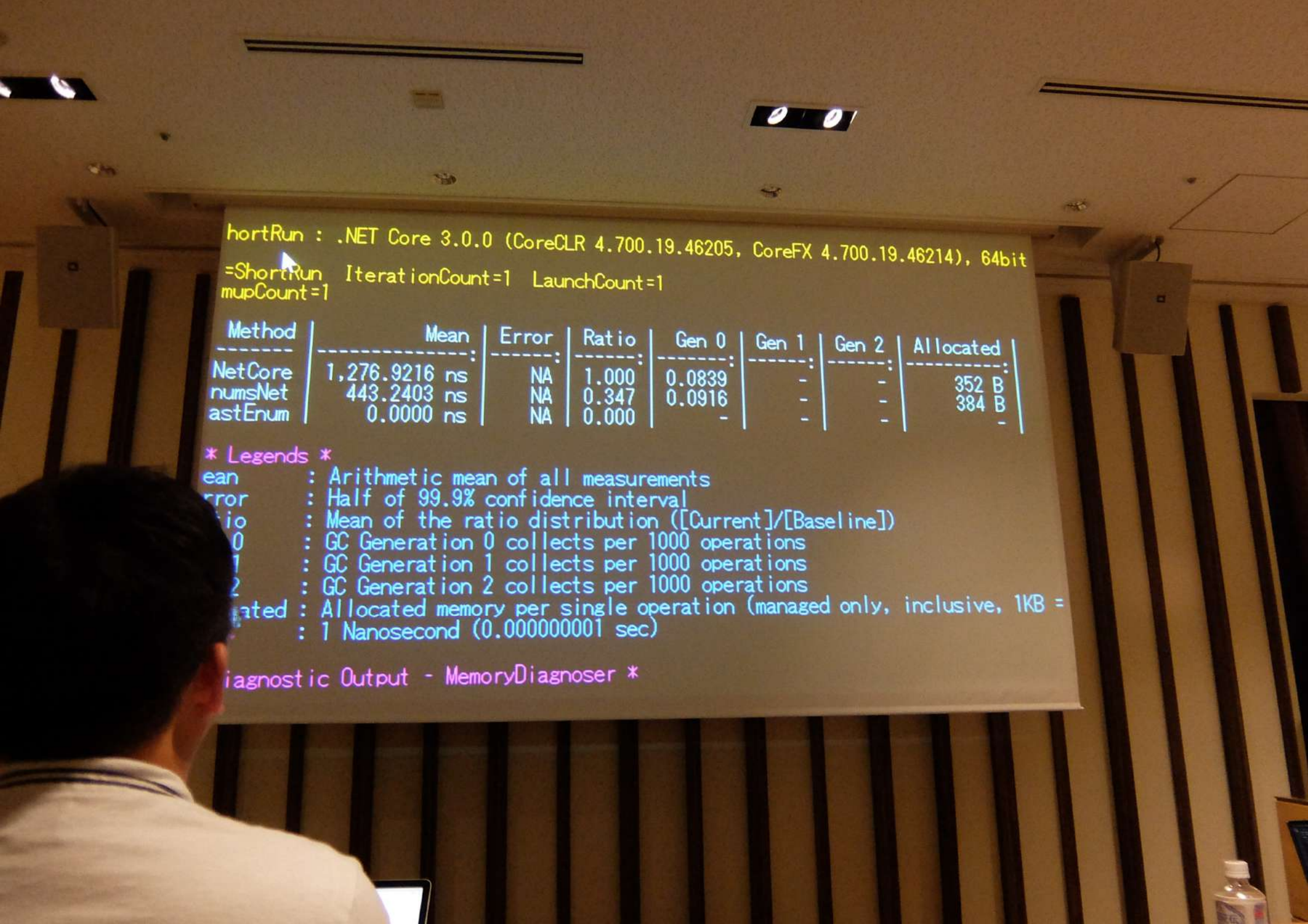

20 _ = _FastEnum.GetValues<Fruits>();
21 }
22
23
24 [Benchmark(Baseline = true)]
0 個の参照 | Takaaki Suzuki、40 日前 | 1 人の作成者、1 件の変更
25 public IReadOnlyList<Fruits> NetCore()
26 => Enum.GetValues(typeof(Fruits)) as Fruits[];
27
28
29 [Benchmark]
0 個の参照 | Takaaki Suzuki、40 日前 | 1 人の作成者、1 件の変更
public IReadOnlyList<Fruits> EnumsNet()
=> Enums.GetValues<Fruits>().ToArray();
[Benchmark]
0 個の参照 | Takaaki Suzuki、40 日前 | 1 人の作成者、1 件の変更
public IReadOnlyList<Fruits> FastEnum()
=> _FastEnum.GetValues<Fruits>();
}

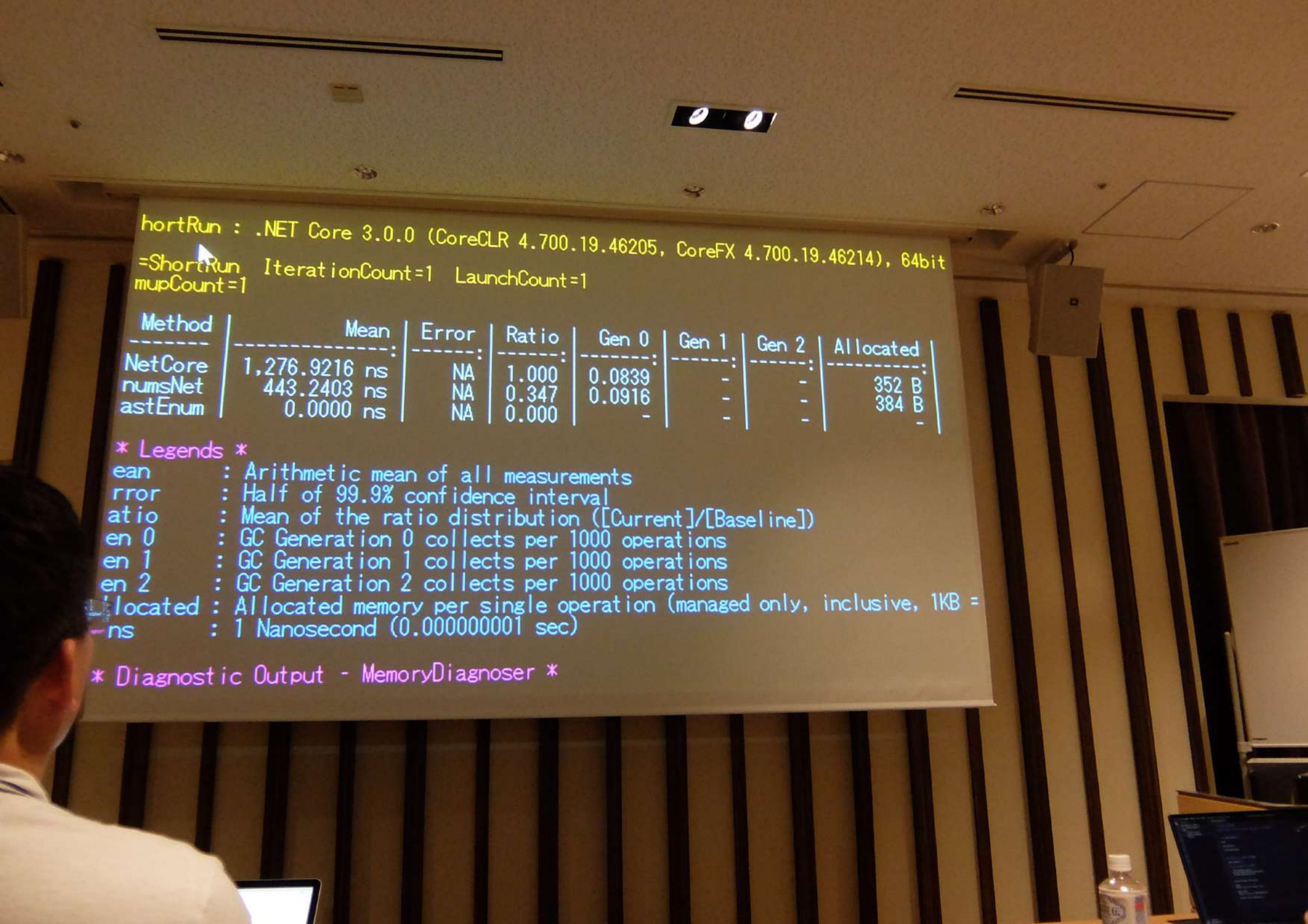


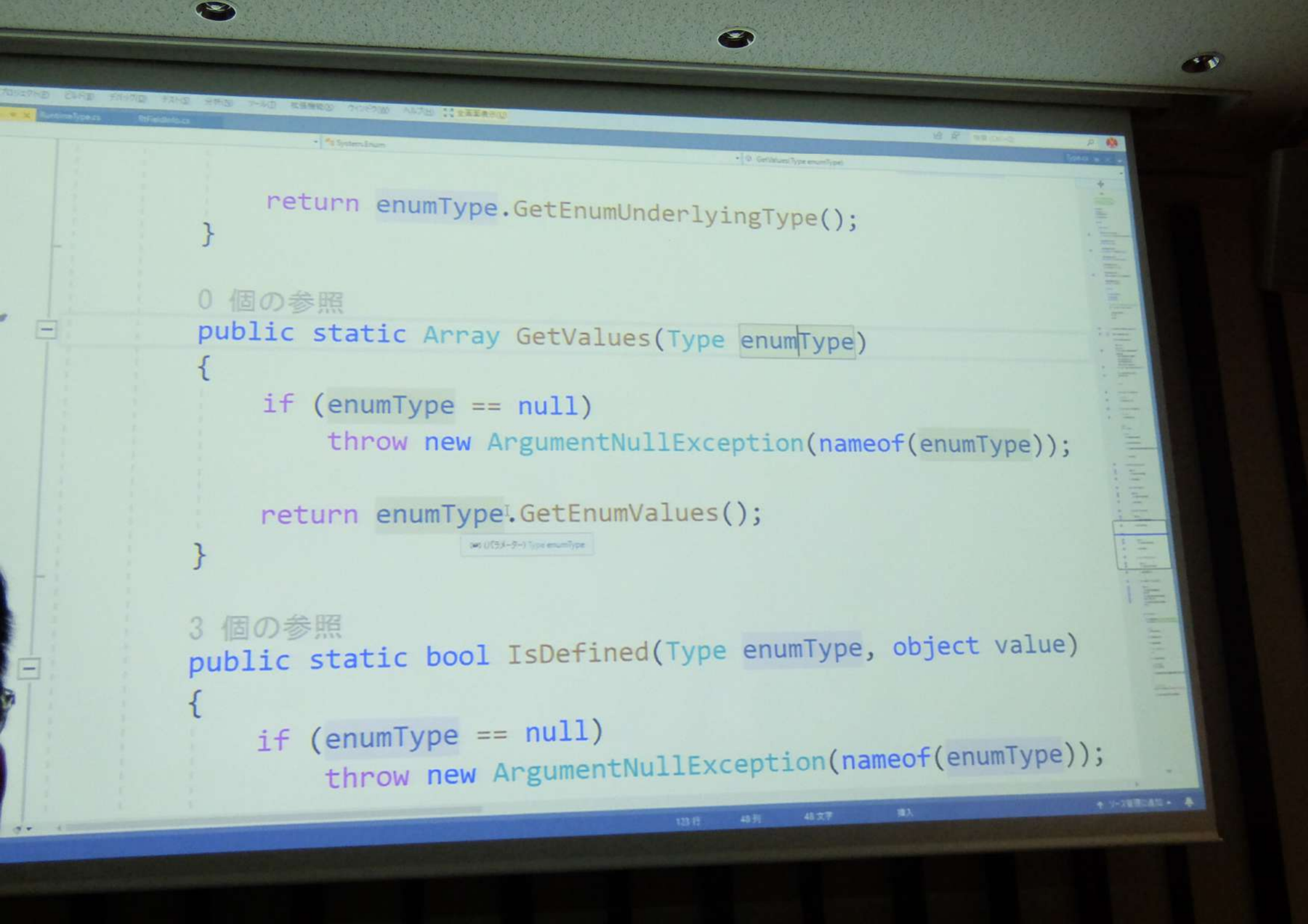
```
23
24      [Benchmark(Baseline = true)]
        0 個の参照 | Takaaki Suzuki、40 日前 | 1 人の作成者、1 件の変更
25      public IReadOnlyList<Fruits> NetCore()
26          => Enum.GetValues(typeof(Fruits)) as Fruits[];
27
28
29      [Benchmark]
        0 個の参照 | Takaaki Suzuki、40 日前 | 1 人の作成者、1 件の変更
        public IReadOnlyList<Fruits> EnumsNet()
            => Enums.GetValues<Fruits>().ToArray();


        [Benchmark]
        0 個の参照 | Takaaki Suzuki、40 日前 | 1 人の作成者、1 件の変更
        public IReadOnlyList<Fruits> FastEnum()
            => _FastEnum.GetValues<Fruits>();
}
```



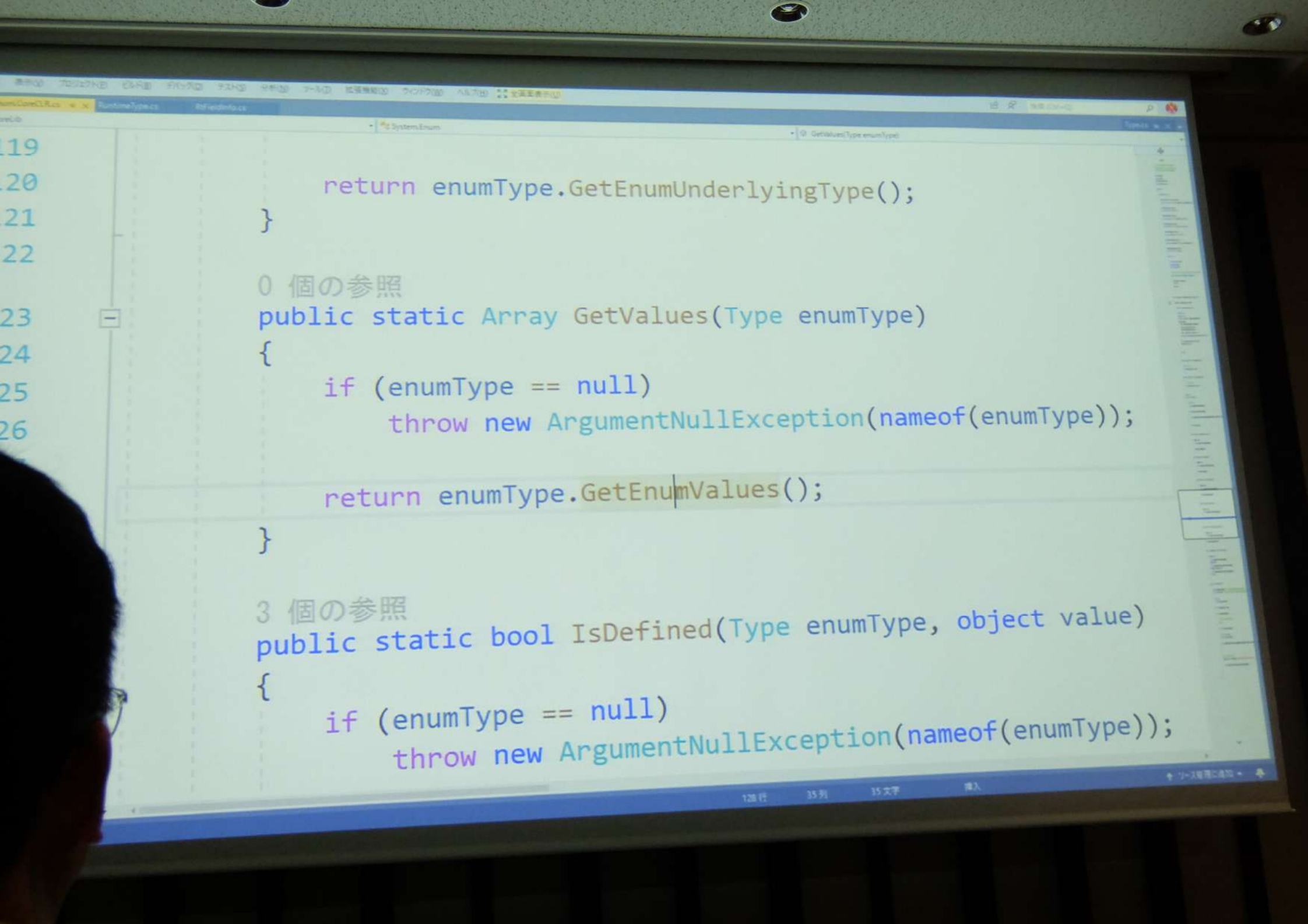
```
23
24          [Benchmark(Baseline = true)]
            0 個の参照 | Takaaki Suzuki、40 日前 | 1 人の作成者、1 件の変更
25          public IReadOnlyList<Fruits> NetCore()
26              => Enum.GetValues(typeof(Fruits)) as Fruits[];
27
28
29          [Benchmark]
            0 個の参照 | Takaaki Suzuki、40 日前 | 1 人の作成者、1 件の変更
30          public IReadOnlyList<Fruits> EnumsNet()
31              => Enums.GetValues<Fruits>().ToArray();


            [Benchmark]
            0 個の参照 | Takaaki Suzuki、40 日前 | 1 人の作成者、1 件の変更
            public IReadOnlyList<Fruits> FastEnum()
                => _FastEnum.GetValues<Fruits>();
        }
    }
```

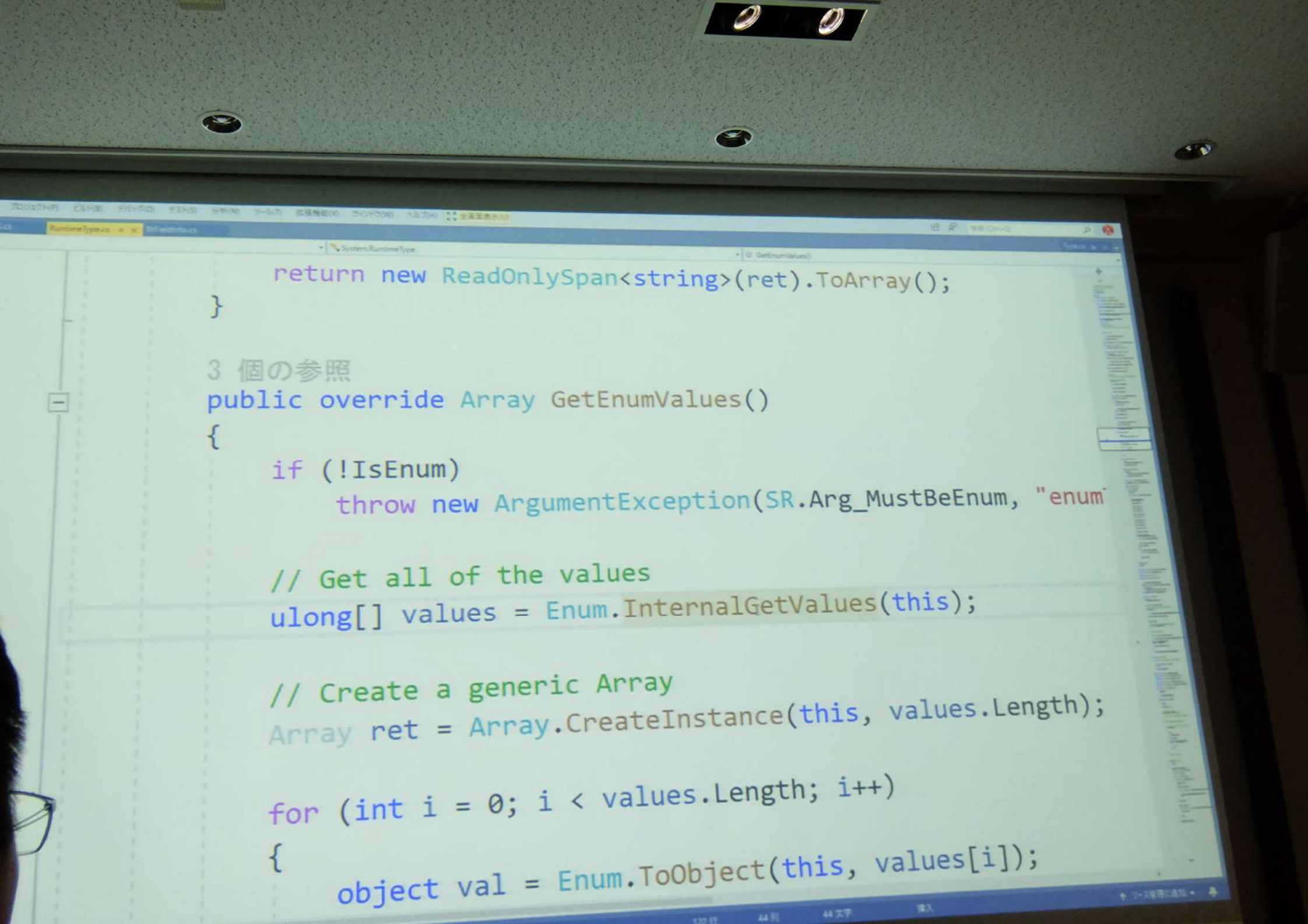

hortRun : .NET Core 3.0.0 (CoreCLR 4.700.19.46205, CoreFX 4.700.19.46214), 64bit
=ShortRun IterationCount=1 LaunchCount=1
mupCount=1
Method | Mean | Error | Ratio | Gen 0 | Gen 1 | Gen 2 | Allocated
NetCore | 1,276.9216 ns | NA | 1.000 | 0.0839 | - | - | 352 B
numsNet | 443.2403 ns | NA | 0.347 | 0.0916 | - | - | 384 B
astEnum | 0.0000 ns | NA | 0.000 | - | - | - | -
* Legends *
ean : Arithmetic mean of all measurements
rror : Half of 99.9% confidence interval
io : Mean of the ratio distribution ([Current]/[Baseline])
0 : GC Generation 0 collects per 1000 operations
1 : GC Generation 1 collects per 1000 operations
2 : GC Generation 2 collects per 1000 operations
ated : Allocated memory per single operation (managed only, inclusive, 1KB =
: 1 Nanosecond (0.000000001 sec)
iagnostic Output - MemoryDiagnoser *

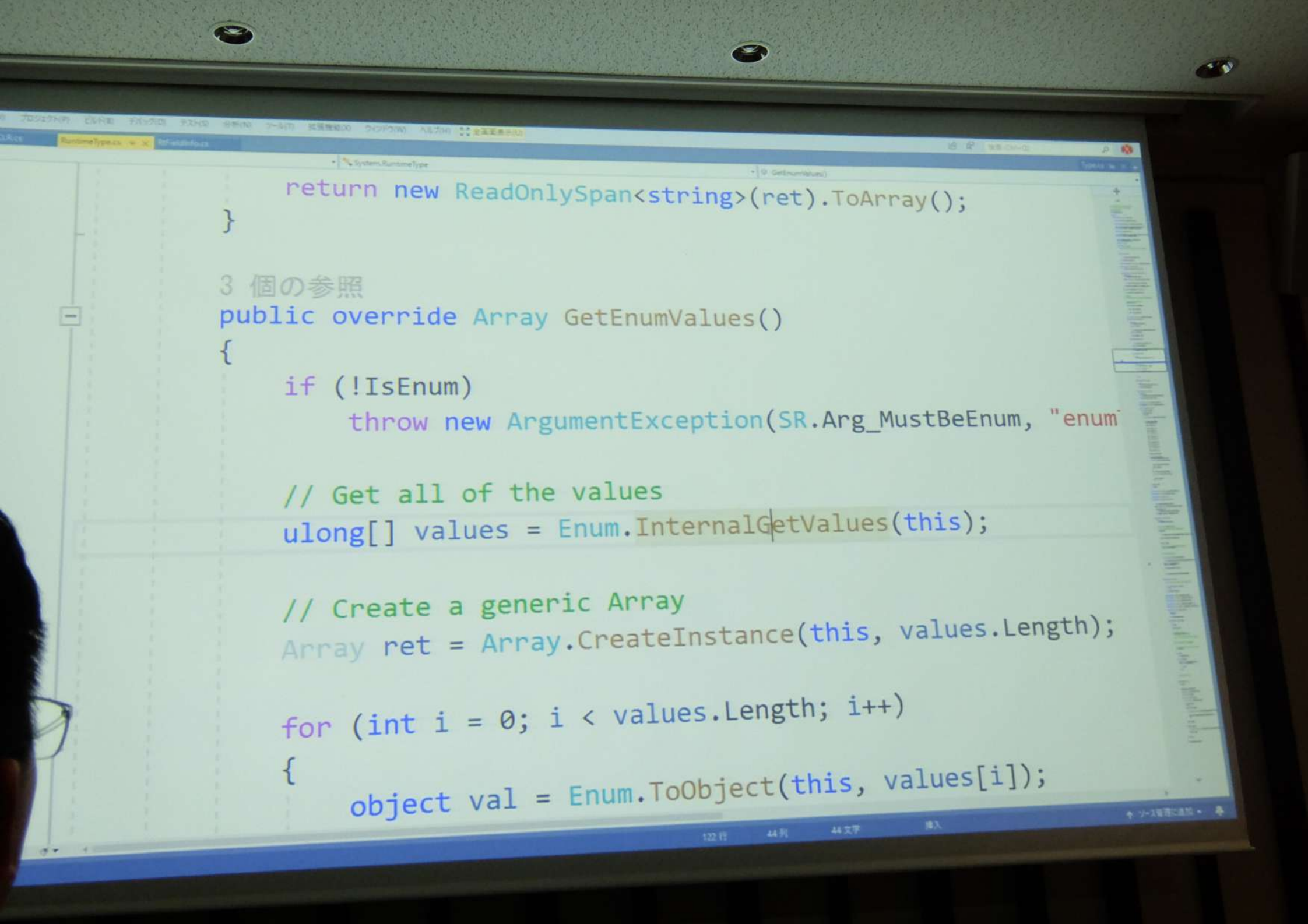

hortRun : .NET Core 3.0.0 (CoreCLR 4.700.19.46205, CoreFX 4.700.19.46214), 64bit
=ShortRun IterationCount=1 LaunchCount=1
mupCount=1
Method | Mean | Error | Ratio | Gen 0 | Gen 1 | Gen 2 | Allocated
NetCore | 1,276.9216 ns | NA | 1.000 | 0.0839 | - | - | 352 B
numsNet | 443.2403 ns | NA | 0.347 | 0.0916 | - | - | 384 B
astEnum | 0.0000 ns | NA | 0.000 | - | - | - | -
* Legends *
ean : Arithmetic mean of all measurements
rror : Half of 99.9% confidence interval
atio : Mean of the ratio distribution ([Current]/[Baseline])
en 0 : GC Generation 0 collects per 1000 operations
en 1 : GC Generation 1 collects per 1000 operations
en 2 : GC Generation 2 collects per 1000 operations
llocated : Allocated memory per single operation (managed only, inclusive, 1KB =
-ns : 1 Nanosecond (0.000000001 sec)
* Diagnostic Output - MemoryDiagnoser *

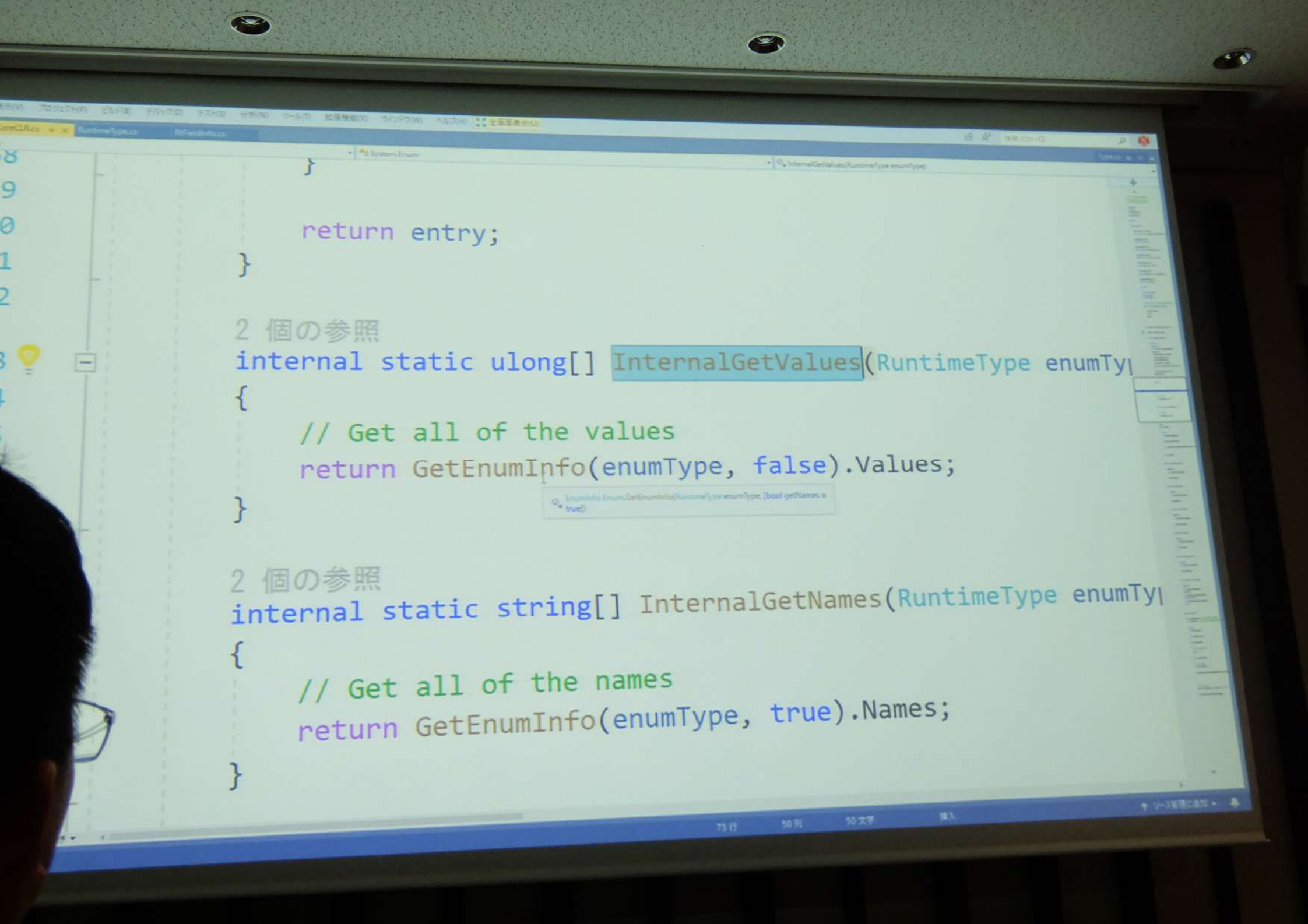


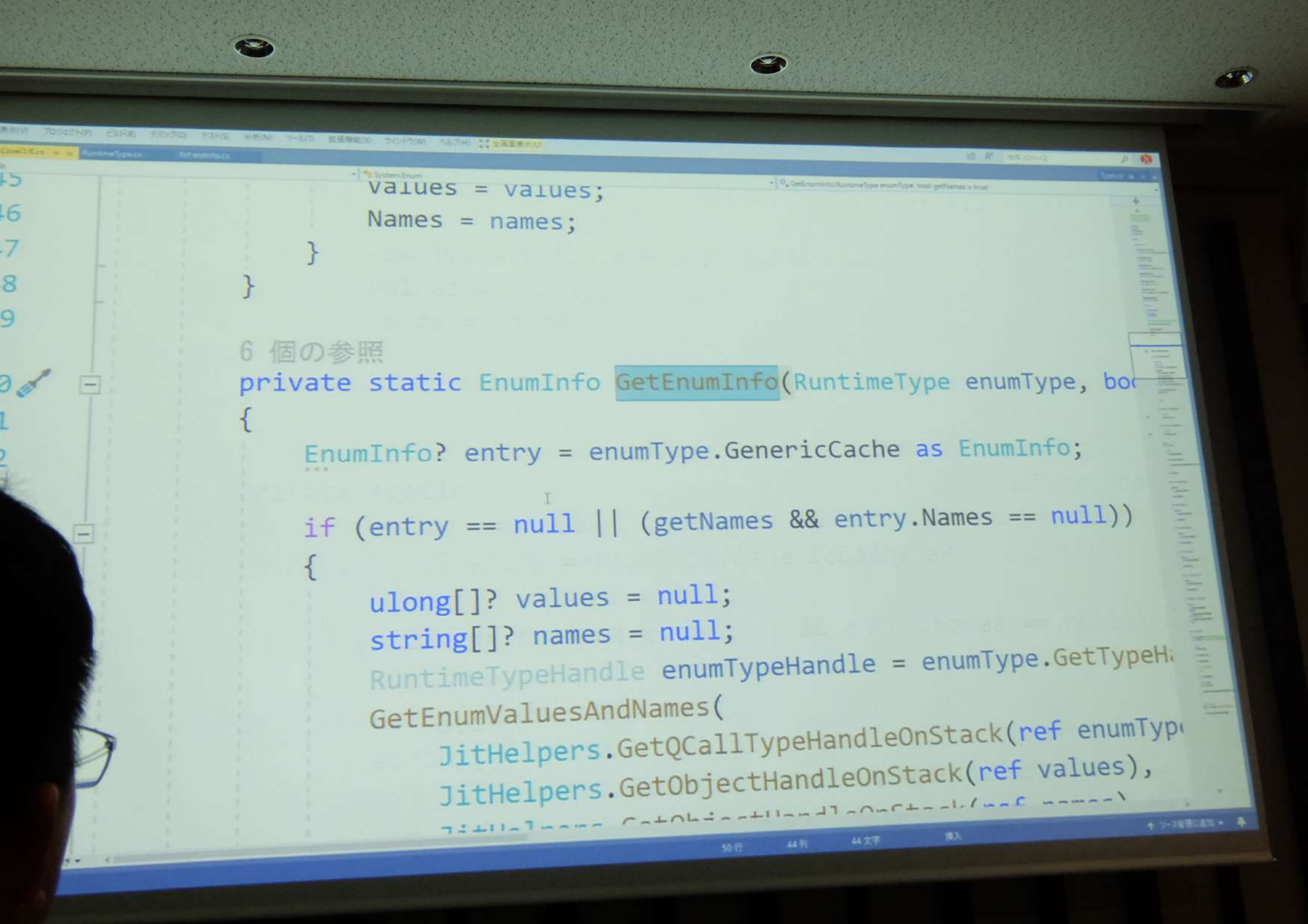
```
        return enumType.GetEnumUnderlyingType();
    }

    0 個の参照
    public static Array GetValues(Type enumType)
    {
        if (enumType == null)
            throw new ArgumentNullException(nameof(enumType));

        return enumType.GetEnumValues();
    }

    3 個の参照
    public static bool IsDefined(Type enumType, object value)
    {
        if (enumType == null)
            throw new ArgumentNullException(nameof(enumType));
```


119
120
121
122
123
124
125
126
return enumType.GetEnumUnderlyingType();
}
0 個の参照
public static Array GetValues(Type enumType)
{
if (enumType == null)
throw new ArgumentNullException(nameof(enumType));
return enumType.GetEnumValues();
}
3 個の参照
public static bool IsDefined(Type enumType, object value)
{
if (enumType == null)
throw new ArgumentNullException(nameof(enumType));

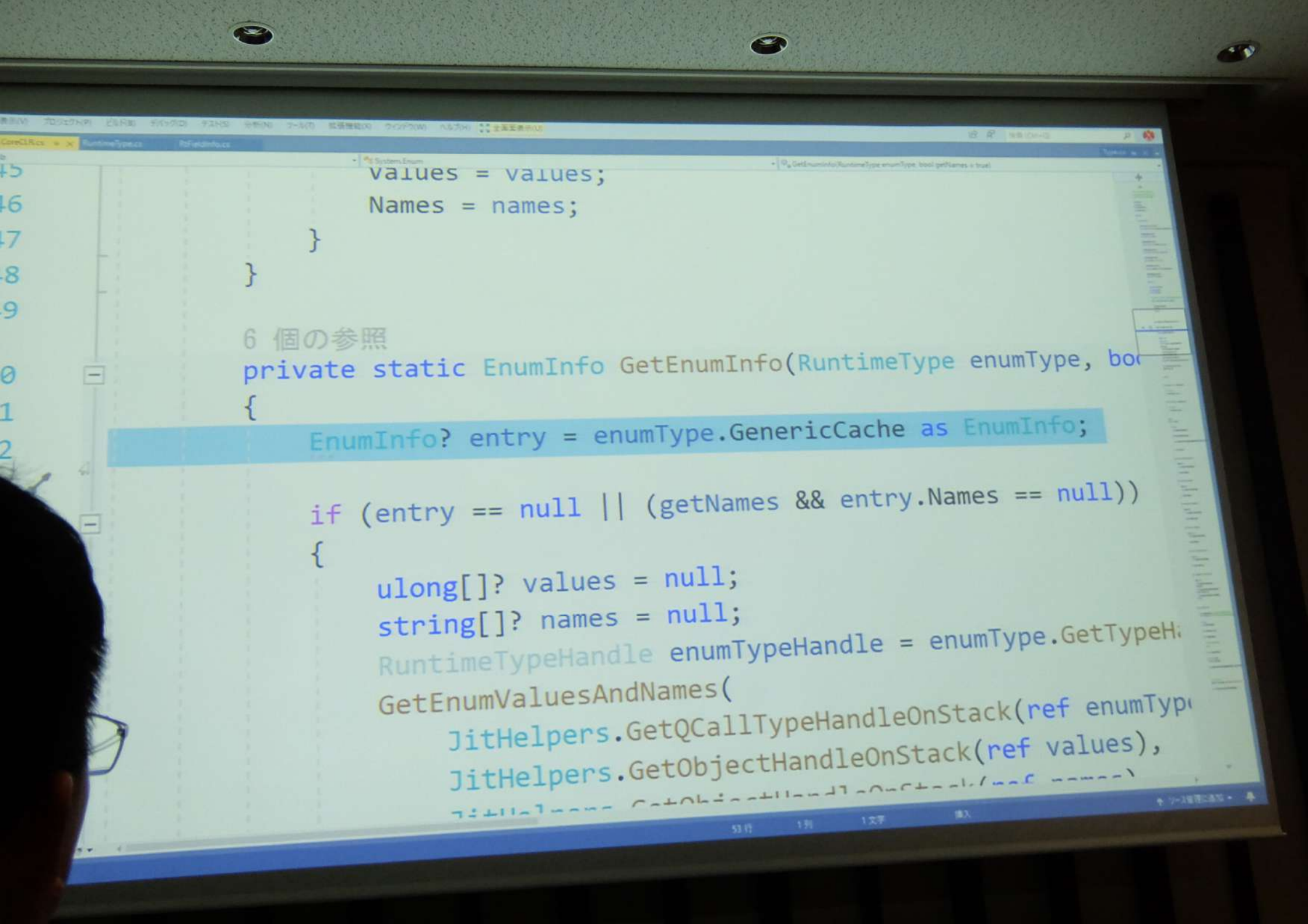

return new ReadOnlySpan<string>(ret).ToArray();
}
3 個の参照
public override Array GetEnumValues()
{
if (!IsEnum)
throw new ArgumentException(SR.Arg_MustBeEnum, "enum
// Get all of the values
ulong[] values = Enum.InternalGetValues(this);
// Create a generic Array
Array ret = Array.CreateInstance(this, values.Length);
for (int i = 0; i < values.Length; i++)
{
object val = Enum.ToObject(this, values[i]);

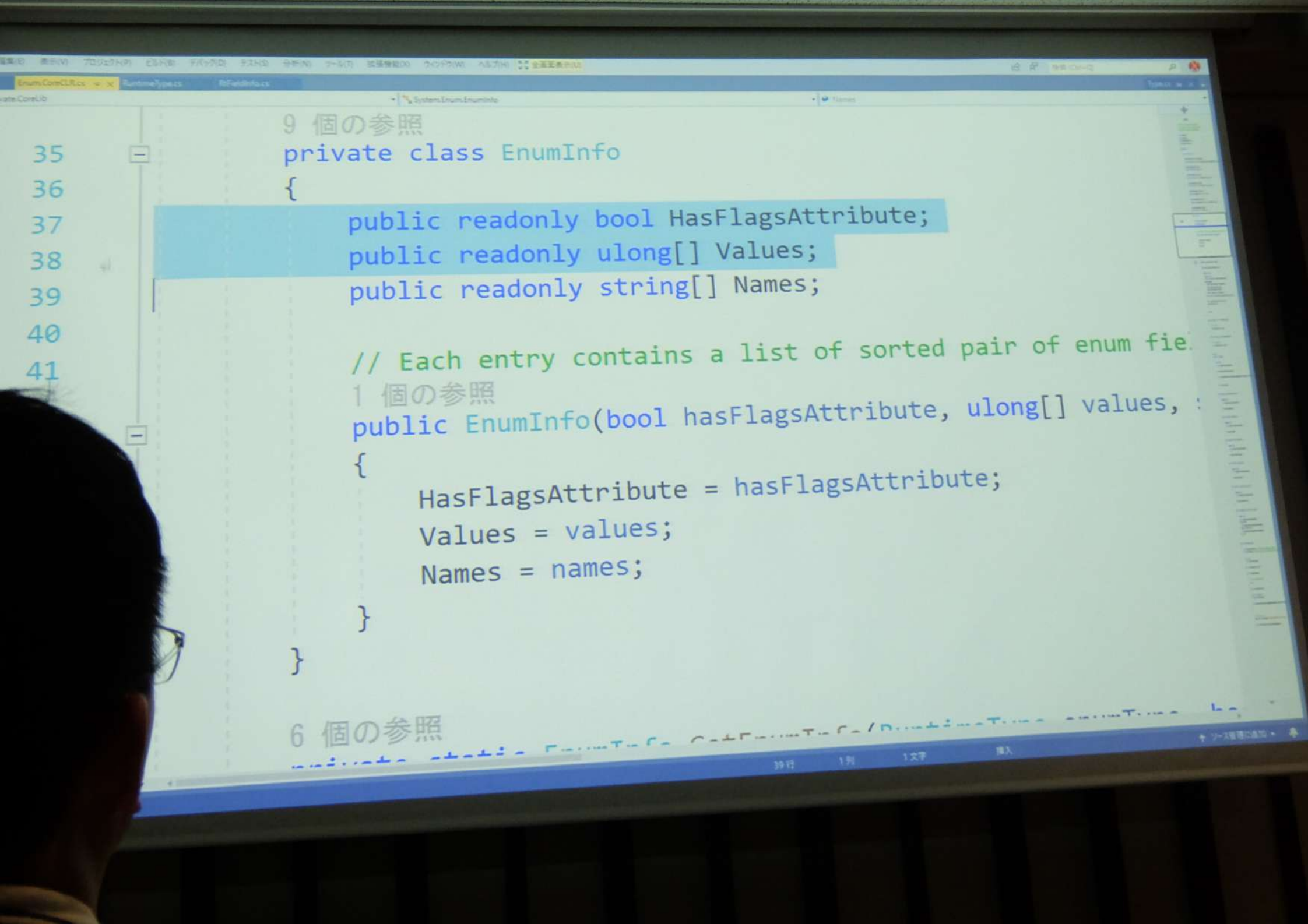

return new ReadOnlySpan<string>(ret).ToArray();
}
3 個の参照
public override Array GetEnumValues()
{
if (!IsEnum)
throw new ArgumentException(SR.Arg_MustBeEnum, "enum
// Get all of the values
ulong[] values = Enum.InternalGetValues(this);
// Create a generic Array
Array ret = Array.CreateInstance(this, values.Length);
for (int i = 0; i < values.Length; i++)
{
object val = Enum.ToObject(this, values[i]);

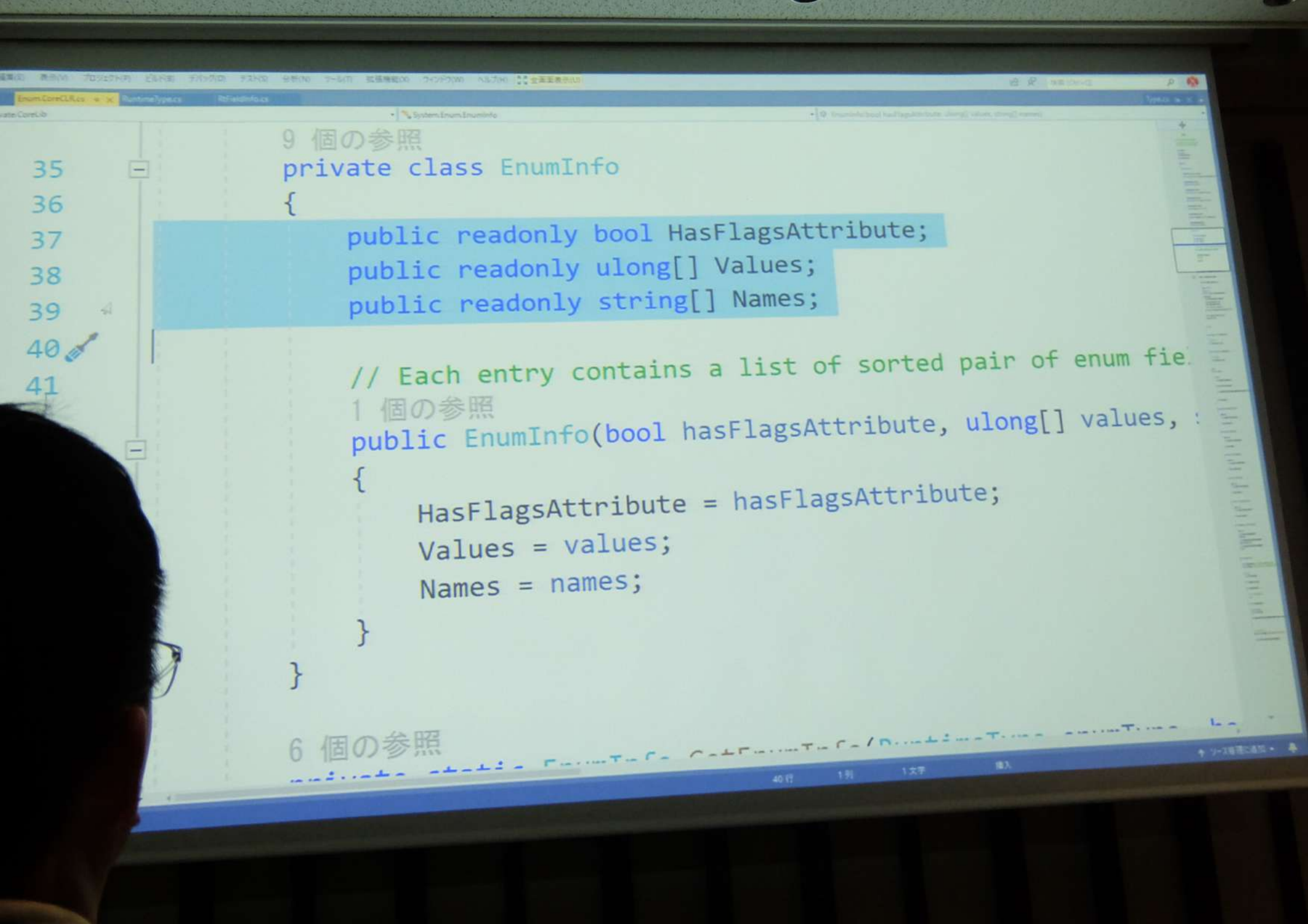

}
return entry;
}
2 個の参照
internal static ulong[] InternalGetValues(RuntimeType enumTy
{
// Get all of the values
return GetEnumInfo(enumType, false).Values;
}
2 個の参照
internal static string[] InternalGetNames(RuntimeType enumTy
{
// Get all of the names
return GetEnumInfo(enumType, true).Names;
}

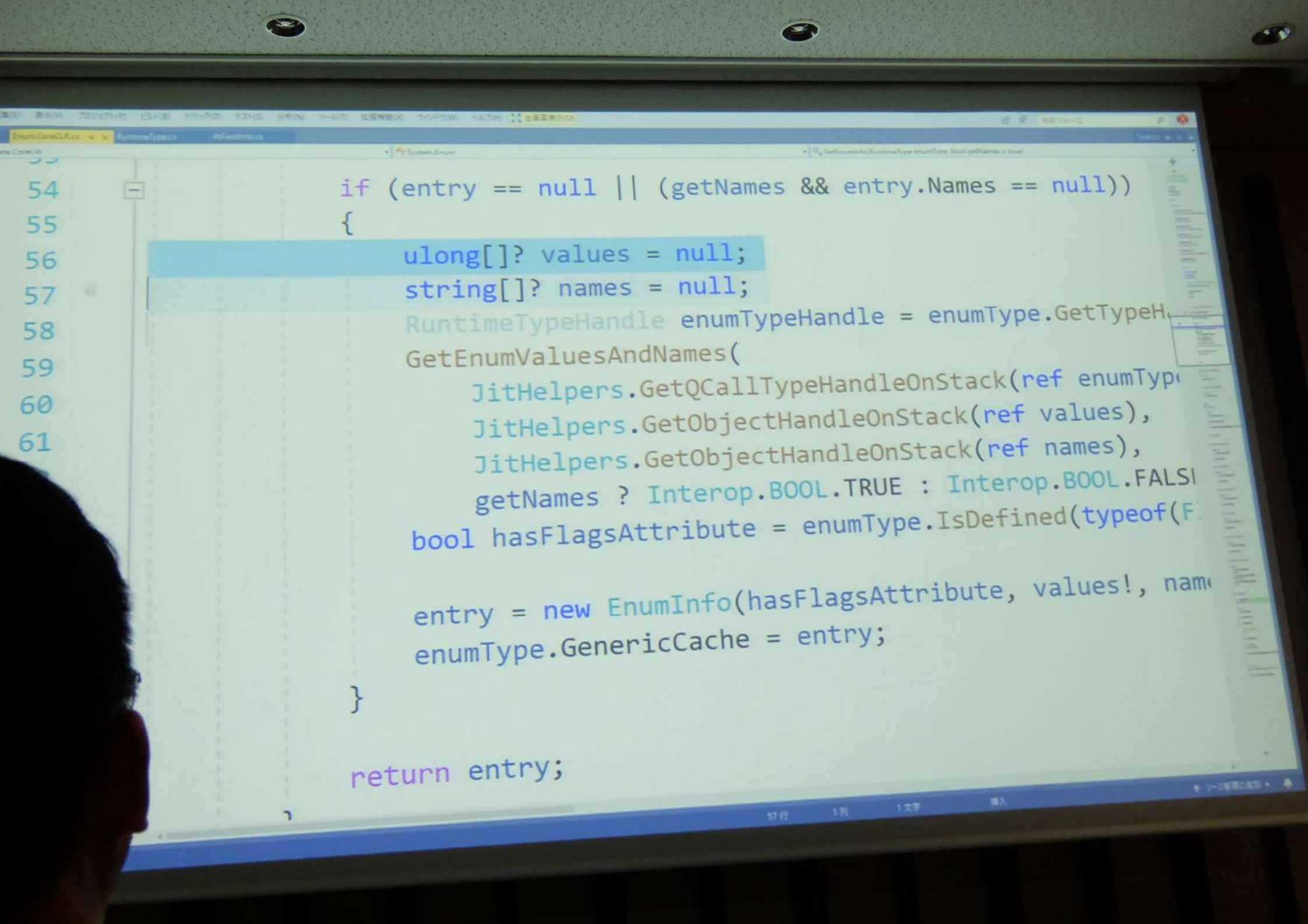


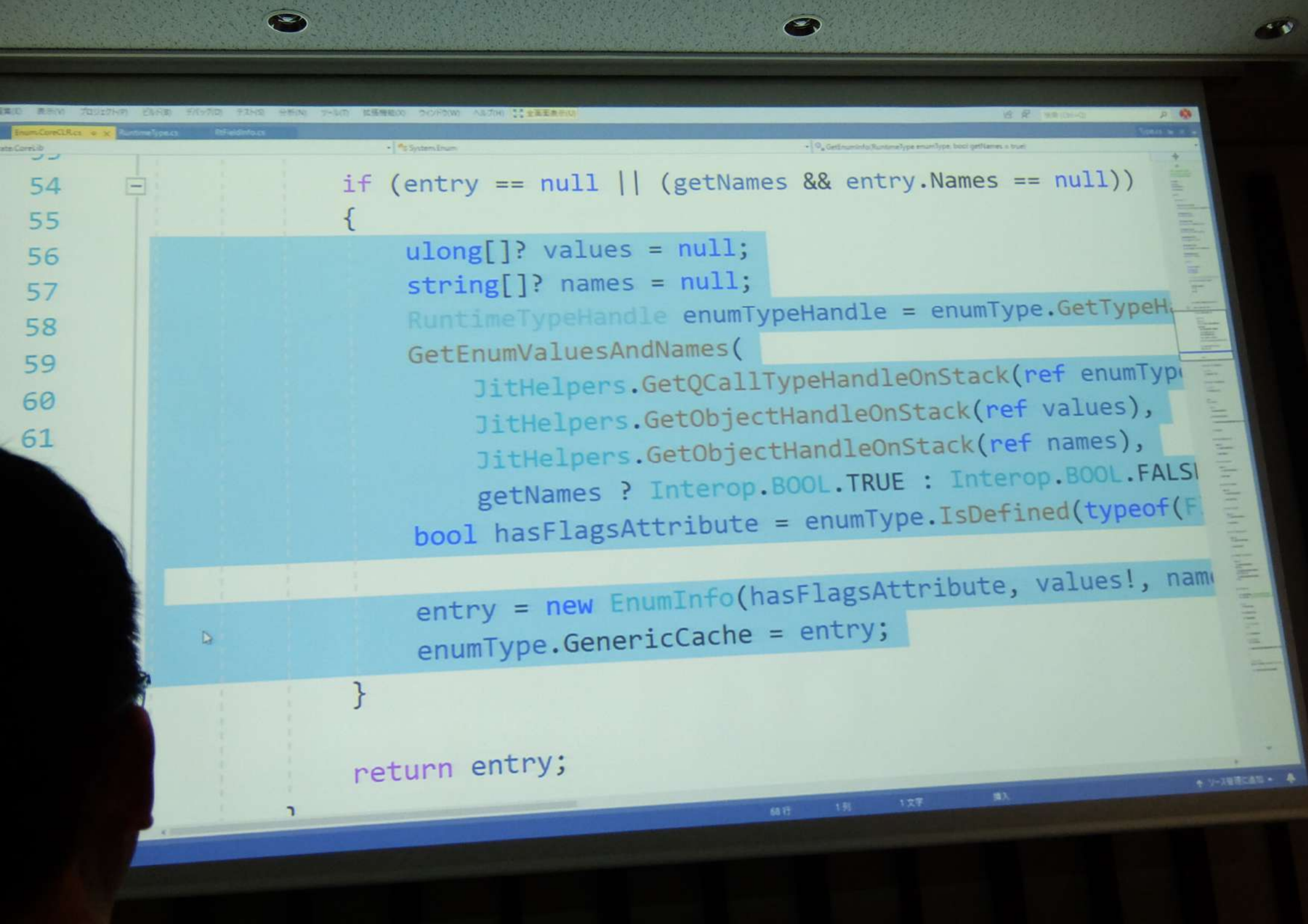
```
            values = values;
            Names = names;
        }
    }

    6 個の参照
    private static EnumInfo GetEnumInfo(RuntimeType enumType, bo
    {
        EnumInfo? entry = enumType.GenericCache as EnumInfo;

        if (entry == null || (getNames && entry.Names == null))
        {
            ulong[]? values = null;
            string[]? names = null;
            RuntimeTypeHandle enumTypeHandle = enumType.GetTypeH
            GetEnumValuesAndNames(
                JitHelpers.GetQCallTypeHandleOnStack(ref enumTyp
                JitHelpers.GetObjectHandleOnStack(ref values),
```

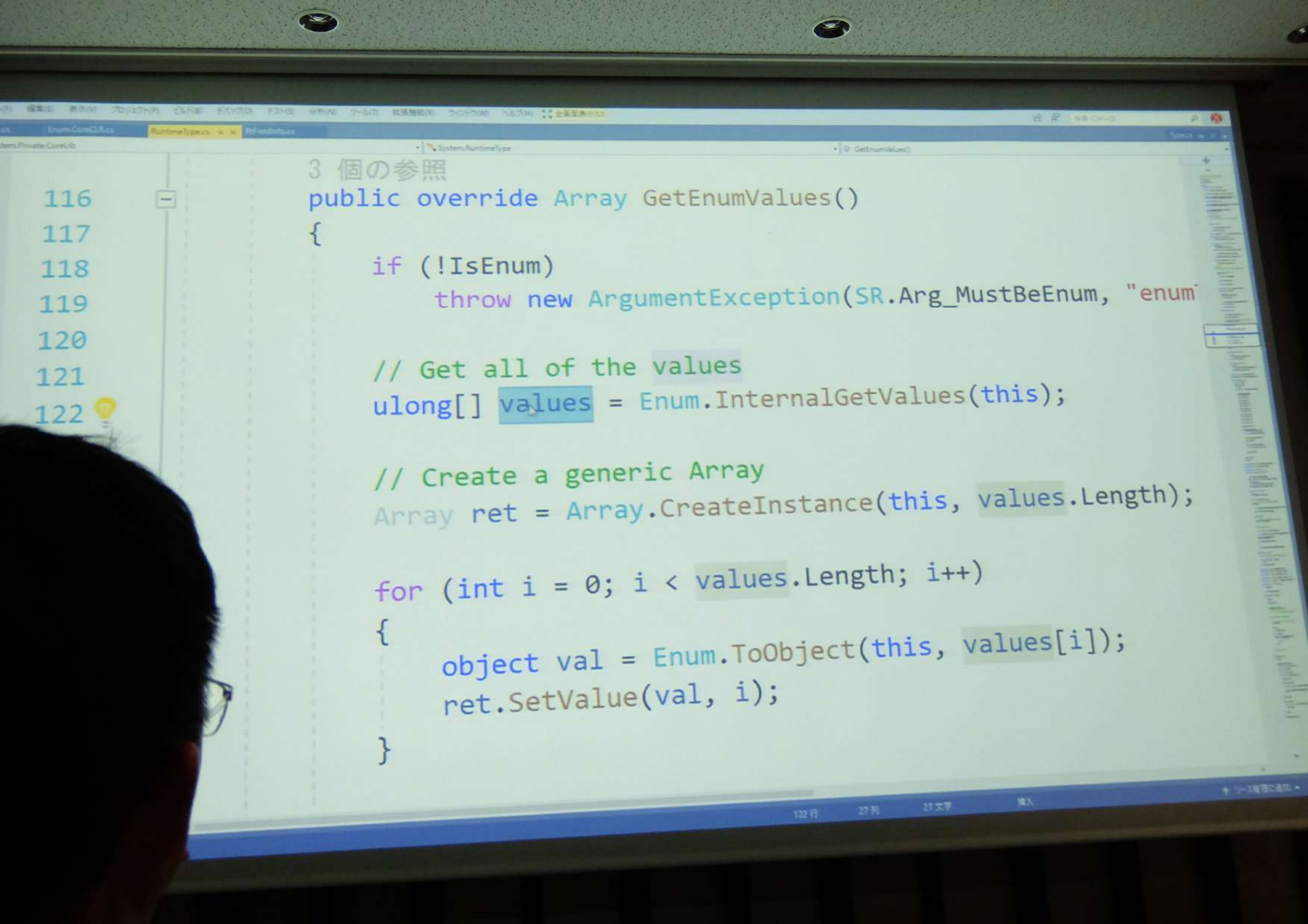

values = values;
Names = names;
}
}
6 個の参照
private static EnumInfo GetEnumInfo(RuntimeType enumType, bo
{
EnumInfo? entry = enumType.GenericCache as EnumInfo;
if (entry == null || (getNames && entry.Names == null))
{
ulong[]? values = null;
string[]? names = null;
RuntimeTypeHandle enumTypeHandle = enumType.GetTypeH
GetEnumValuesAndNames(
JitHelpers.GetQCallTypeHandleOnStack(ref enumTyp
JitHelpers.GetObjectHandleOnStack(ref values),

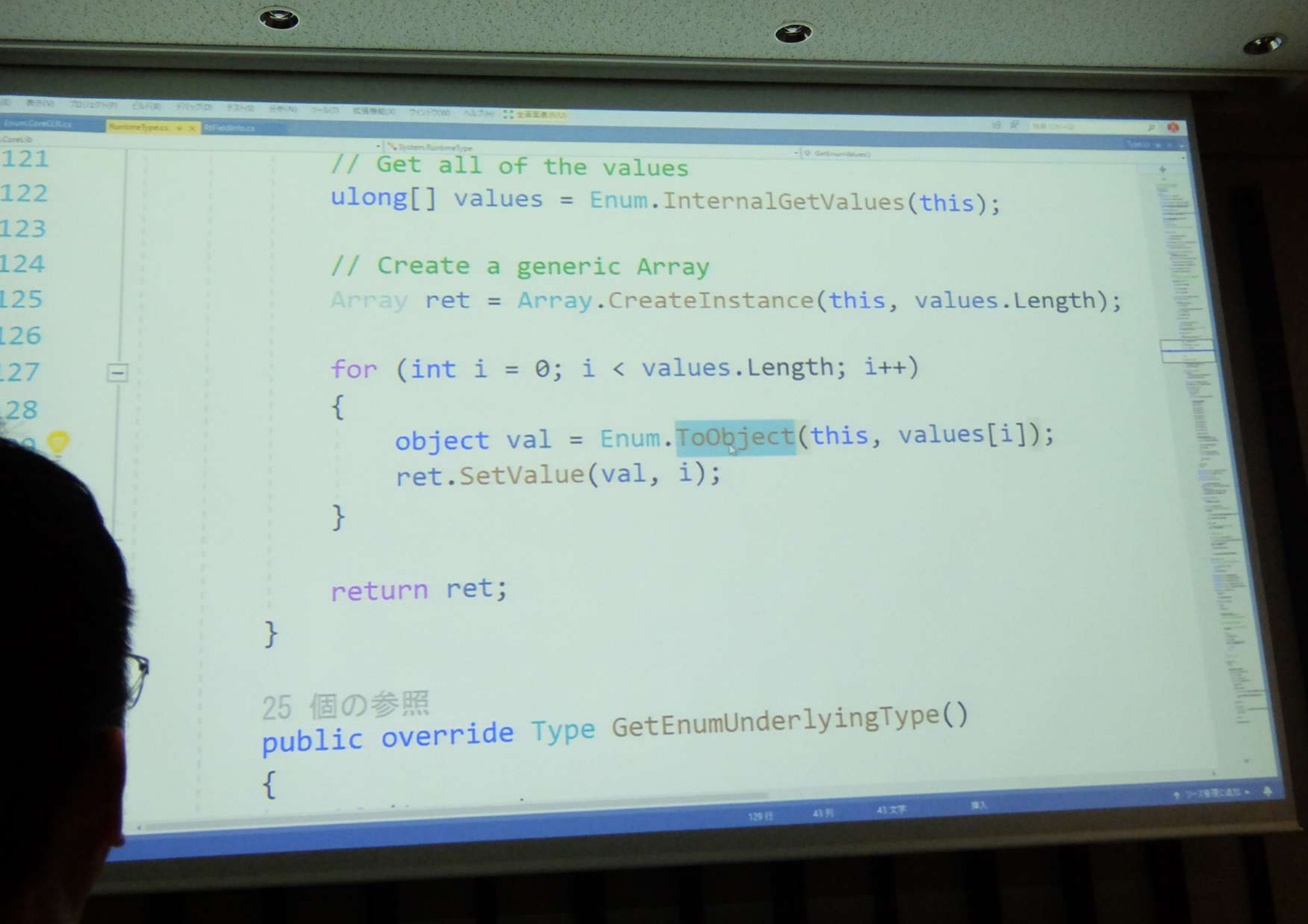

9 個の参照
private class EnumInfo
{
public readonly bool HasFlagsAttribute;
public readonly ulong[] Values;
public readonly string[] Names;
// Each entry contains a list of sorted pair of enum fie
1 個の参照
public EnumInfo(bool hasFlagsAttribute, ulong[] values,
{
HasFlagsAttribute = hasFlagsAttribute;
Values = values;
Names = names;
}
}
6 個の参照


9 個の参照
private class EnumInfo
{
public readonly bool HasFlagsAttribute;
public readonly ulong[] Values;
public readonly string[] Names;
// Each entry contains a list of sorted pair of enum fie
1 個の参照
public EnumInfo(bool hasFlagsAttribute, ulong[] values,
{
HasFlagsAttribute = hasFlagsAttribute;
Values = values;
Names = names;
}
}
6 個の参照


54 if (entry == null || (getNames && entry.Names == null))
55 {
56 ulong[]? values = null;
57 string[]? names = null;
58 RuntimeTypeHandle enumTypeHandle = enumType.GetTypeH
59 GetEnumValuesAndNames(
60 JitHelpers.GetQCallTypeHandleOnStack(ref enumTyp
61 JitHelpers.GetObjectHandleOnStack(ref values),
JitHelpers.GetObjectHandleOnStack(ref names),
getNames ? Interop.BOOL.TRUE : Interop.BOOL.FALSI
bool hasFlagsAttribute = enumType.IsDefined(typeof(F
entry = new EnumInfo(hasFlagsAttribute, values!, nam
enumType.GenericCache = entry;
}
return entry;


54 if (entry == null || (getNames && entry.Names == null))
55 {
56 ulong[]? values = null;
57 string[]? names = null;
58 RuntimeTypeHandle enumTypeHandle = enumType.GetTypeH
59 GetEnumValuesAndNames(
60 JitHelpers.GetQCallTypeHandleOnStack(ref enumTyp
61 JitHelpers.GetObjectHandleOnStack(ref values),
JitHelpers.GetObjectHandleOnStack(ref names),
getNames ? Interop.BOOL.TRUE : Interop.BOOL.FALS
bool hasFlagsAttribute = enumType.IsDefined(typeof(F
entry = new EnumInfo(hasFlagsAttribute, values!, nam
enumType.GenericCache = entry;
}
return entry;


3 個の参照
public override Array GetEnumValues()
{
if (!IsEnum)
throw new ArgumentException(SR.Arg_MustBeEnum, "enum
// Get all of the values
ulong[] values = Enum.InternalGetValues(this);
// Create a generic Array
Array ret = Array.CreateInstance(this, values.Length);
for (int i = 0; i < values.Length; i++)
{
object val = Enum.ToObject(this, values[i]);
ret.SetValue(val, i);
}


// Get all of the values
ulong[] values = Enum.InternalGetValues(this);
// Create a generic Array
Array ret = Array.CreateInstance(this, values.Length);
for (int i = 0; i < values.Length; i++)
{
object val = Enum.ToObject(this, values[i]);
ret.SetValue(val, i);
}
return ret;
}
25 個の参照
public override Type GetEnumUnderlyingType()
{


Why FastEnum is fast?
高速化のアプローチとちょっとした工夫

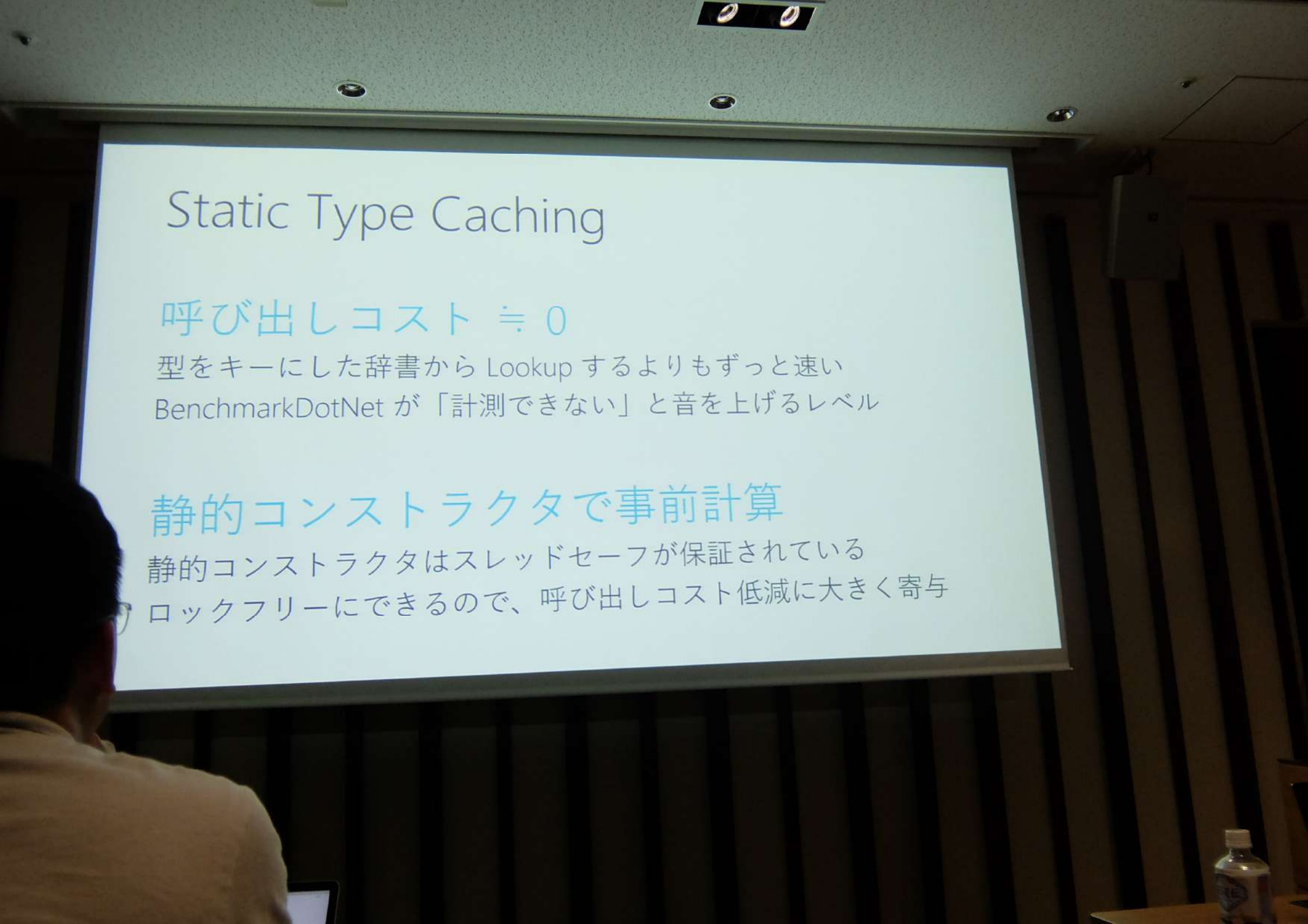
Static Type Caching

```
public static class FastEnum
{
    public static IReadOnlyList<T> GetValues<T>()
        where T : struct, Enum
        => Cache<T>.Values;  // キャッシュを直接参照

    // 静的 Generics 型のフィールドにキャッシュを持つ
    private static class Cache<T>
        where T : struct, Enum
    {
        public static readonly T[] Values;
        static Cache()
            => Values = (T[])Enum.GetValues(typeof(T));
    }
}
```

Static Type Caching

```
public static class FastEnum
{
    public static IReadOnlyList<T> GetValues<T>()
        where T : struct, Enum
        => Cache<T>.Values;  // キャッシュを直接参照

    // 静的 Generics 型のフィールドにキャッシュを持つ
    private static class Cache<T>
        where T : struct, Enum
    {
        public static readonly T[] Values;
        static Cache()
            => Values = (T[])Enum.GetValues(typeof(T));
    }
}
```

T 型ごとに
別の型扱い


Static Type Caching
呼び出しコスト ≒ 0
型をキーにした辞書から Lookup するよりもずっと速い
BenchmarkDotNet が「計測できない」と音を上げるレベル
静的コンストラクタで事前計算
静的コンストラクタはスレッドセーフが保証されている
ロックフリーにできるので、呼び出しコスト低減に大きく寄与

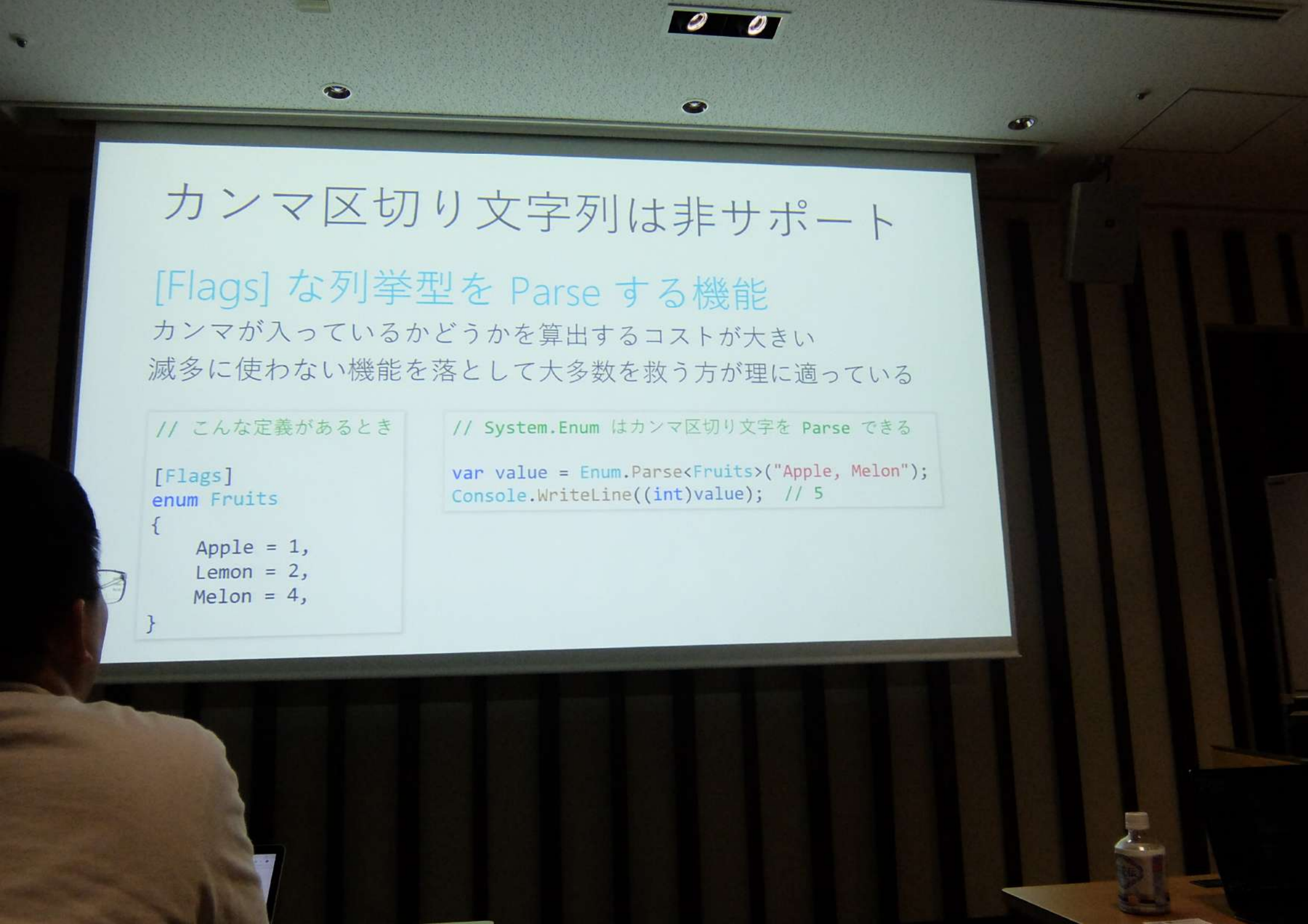
Generics API のみをサポート

Non Generics API は box 化を避けられない

Generics API だけでも大抵のケースに耐えられる

```
// FastEnum (一切 Box 化しない)

var a = FastEnum.GetValues<Fruits>();
var b = FastEnum.GetName(Fruits.Banana);
var c = FastEnum.Parse<Fruits>("Apple");
var d = FastEnum.IsDefined(Fruits.Lemon);
```

```
// .NET Core (Box 化する)

var a = Enum.GetValues(typeof(Fruits));
var b = Enum.GetName(typeof(Fruits), Fruits.Banana);
var c = (Fruits)Enum.Parse(typeof(Fruits), "Apple");
var d = Enum.IsDefined(typeof(Fruits), Fruits.Lemon);
```


カンマ区切り文字列は非サポート
[Flags] な列挙型を Parse する機能
カンマが入っているかどうかを算出するコストが大きい
滅多に使わない機能を落として大多数を救う方が理に適っている
// こんな定義があるとき
[Flags]
enum Fruits
{
Apple = 1,
Lemon = 2,
Melon = 4,
}
// System.Enum はカンマ区切り文字を Parse できる
var value = Enum.Parse<Fruits>("Apple, Melon");
Console.WriteLine((int)value); // 5

IsDefined<T>(value); の最適化

列挙型の値の連続性を利用

値が連続しているなら最大/最小範囲に入っているかを比較
辞書の ContainsKey をするより範囲チェックの方が何倍も高速

```
// 連続した値の列挙型

enum Fruits
{
    Apple = 1,
    Lemon,  // 2
    Melon,  // 3
    Banana, // 4
}
```

```
// こんな感じのイメージで判定すると速い
// 連続しているかどうかは事前に判定してキャッシュ

public static bool IsDefined<T>(T value)
    where T : struct, Enum
    => IsContinuous
    ? MinValue <= value && value <= MaxValue
    : Values.ContainsKey(value);
```

Parse<T>("value/name"); の最適化

"123", "Apple" をどう判別/解析するか

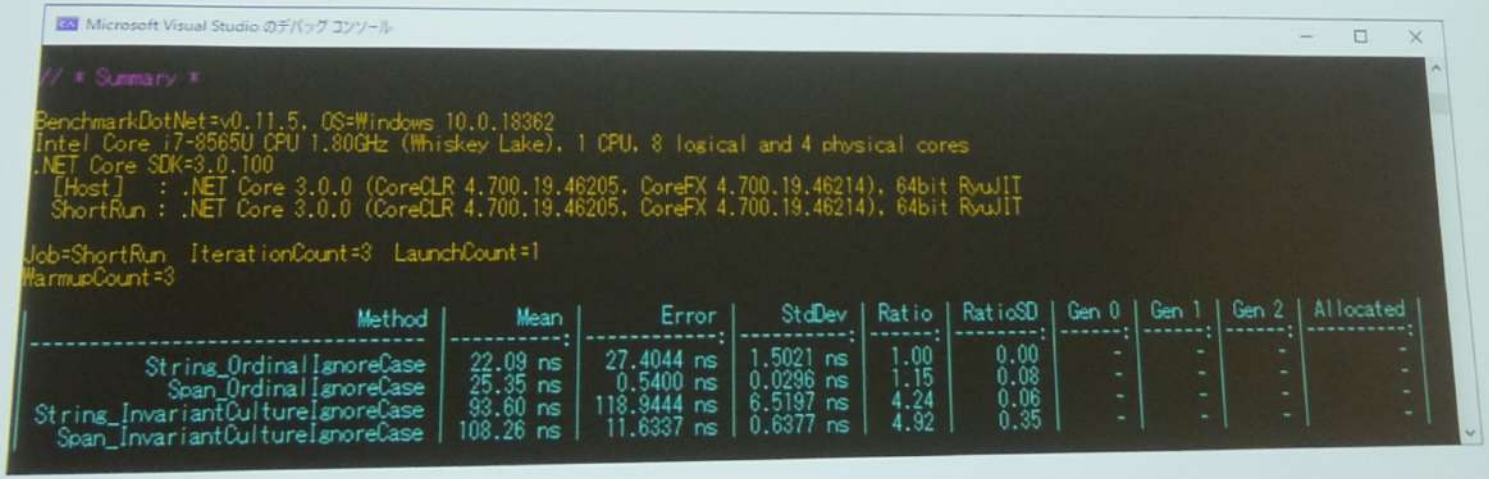
「1 文字目が数字か +/-」の場合は値で解析する

```
public static bool TryParse<T>(string value, out T result)
    where T : struct, Enum
{
    // フィールド名は数字/特殊記号で始められない仕様を利用
    return IsNumeric(value[0])
        ? TryParseByValue(value, out result)
        : TryParseByName(value, out result);

    static bool IsNumeric(char c)
        => char.IsDigit(c) || c == '-' || c == '+';
}
```

大小無視の文字列比較

string.Equals(OrdinalIgnoreCase) が最速

StringComparison.InvariantCultureIgnoreCase より４倍速い



温もりティ溢れる手作り辞書

キーの特殊化で EqualityComparer を廃止

GetHashCode(); と Equals(); を直呼び出しにするハック

http://engineering.grani.jp/entry/2017/07/28/145035

```
// 正味たった 2 行だけの違い
// 最終的にこの差がベンチマークに効いてくる

var hash = EqualityComparer<int>.Default.GetHashCode(key);
var hash = key.GetHashCode();  // faster

if (EqualityComparer<TKey>.Default.Equals(next.Key, key))
if (next.Key == key)  // faster
```

温もりティ溢れる手作り辞書

キーの特殊化で EqualityComparer を廃止

GetHashCode(); と Equals(); を直呼び出しにするハック

http://engineering.grani.jp/entry/2017/07/28/145035

```
// 正味たった 2 行だけの違い
// 最終的にこの差がベンチマークに効いてくる

var hash = EqualityComparer<int>.Default.GetHashCode(key);
var hash = key.GetHashCode();  // faster

if (EqualityComparer<TKey>.Default.Equals(next.Key, key))
if (next.Key == key)  // faster
```

列挙型を等値/大小で比較する

```
// 具象型の場合はできるけれど...
public static void Compare(Fruits left, Fruits right)
{
    var a = left == right;  // OK
    var b = left <= right;  // OK
}
```

```
// Enum 制約をしていても Generics 型だとできない！
public static void Compare<T>(T left, T right)
    where T : struct, Enum
{
    var a = left == right;  // コンパイルエラー
    var b = left <= right;  // コンパイルエラー
    var c = EqualityComparer<T>.Default.Equals(left, right);  // OK
}
```


列挙型を等値/大小で比較する
// 具象型の場合はできるけれど...
public static void Compare(Fruits left, Fruits right)
{
var a = left == right; // OK
var b = left <= right; // OK
}
// Enum 制約をしていても Generics 型だとできない！
lic static void Compare<T>(T left, T right)
here T : struct, Enum
a = left == right; // コンパイルエラー
b = left <= right; // コンパイルエラー
c = EqualityComparer<T>.Default.Equals(left, right); // OK
なんとか
解決したい

Unsafe.As を使った強制型変換

System.Runtime.CompilerServices.Unsafe

unsafe オプションを使わずに unsafe なことをする悪い子😈

```
public static void Compare<T>(T left, T right)
    where T : struct, Enum
{
    // ref を使った型変換 = 同一メモリ領域を別の型として参照
    ref var l = ref Unsafe.As<T, int>(ref left);
    ref var r = ref Unsafe.As<T, int>(ref right);

    var a = l == r;  // OK
    var b = l <= r;  // OK
}
```

Unsafe.As のちょっとズルい使い方

UnderlyingType の TryParse に委譲

同一メモリ領域を参照した強制型変換なので、できちゃう😈

```
internal sealed class Int32Operation<T> : IUnderlyingOperation<T>
    where T : struct, Enum
{
    public bool TryParse(string text, out T result)
    {
        // x に書き込めば result にも書き込まれたことになる
        result = default;
        ref var x = ref Unsafe.As<T, int>(ref result);
        return int.TryParse(text, out x);
    }
}
```

Enum.HasFlag の脅威的性能

FastEnum が唯一勝てなかったメソッド

引数が Enum なので呼び出すだけで box 化して超遅そうだが…
自作したものより 200 倍以上高速でアロケーションもない

```
// 特殊な最適化がかかってる (たぶん

[Intrinsic]
public bool HasFlag(Enum flag)
{}
```

Method	Mean	Error	StdDev	Median	Ratio	RatioSD	Gen 0	Gen 1	Gen 2	Allocated
NetCore	0.0125 ns	0.3945 ns	0.0216 ns	0.0000 ns	?	?	-	-	-	-
FastEnum	2.2857 ns	3.0054 ns	0.1647 ns	2.3766 ns	?	?	-	-	-	-

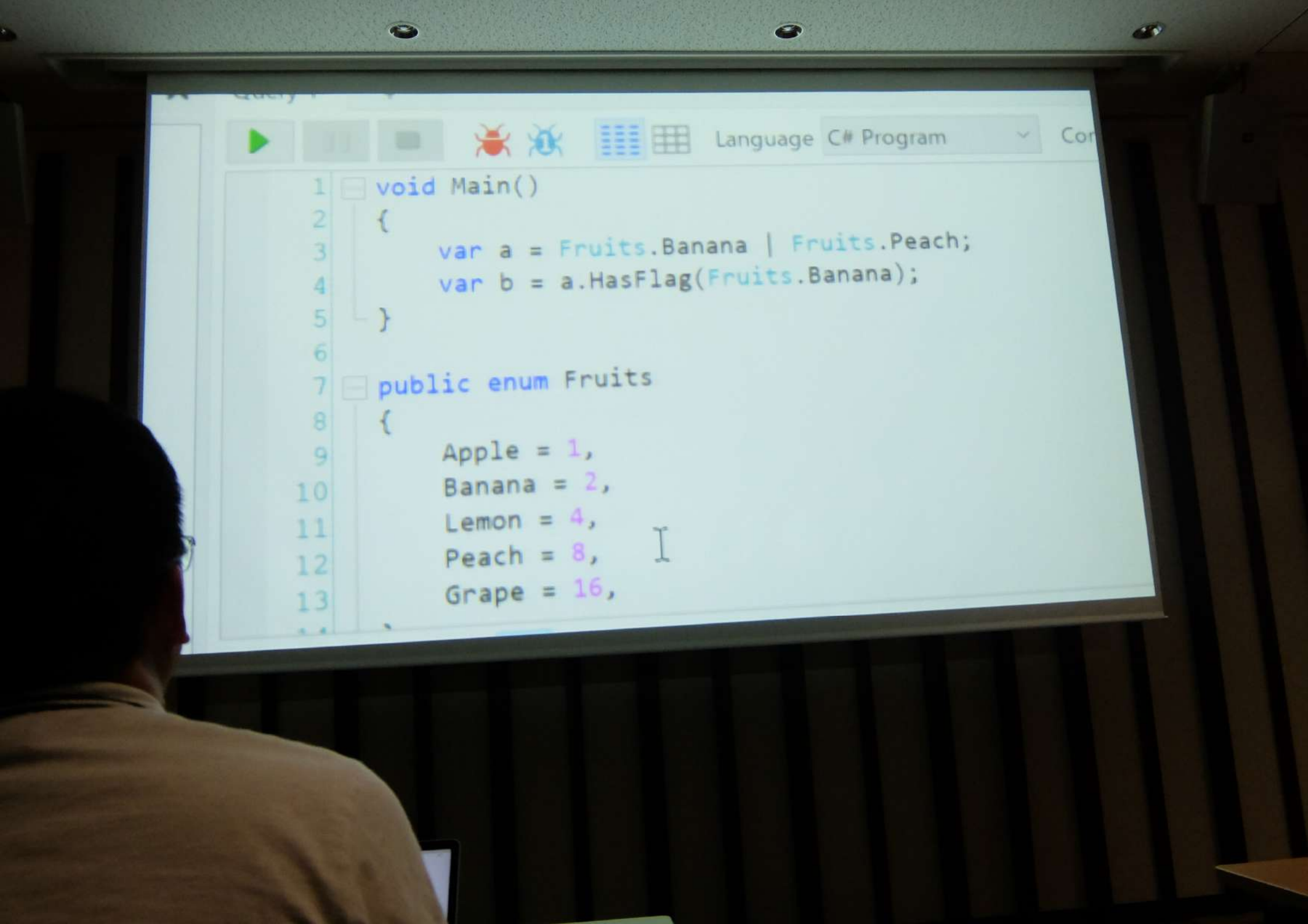
Enum.HasFlag の脅威的性能

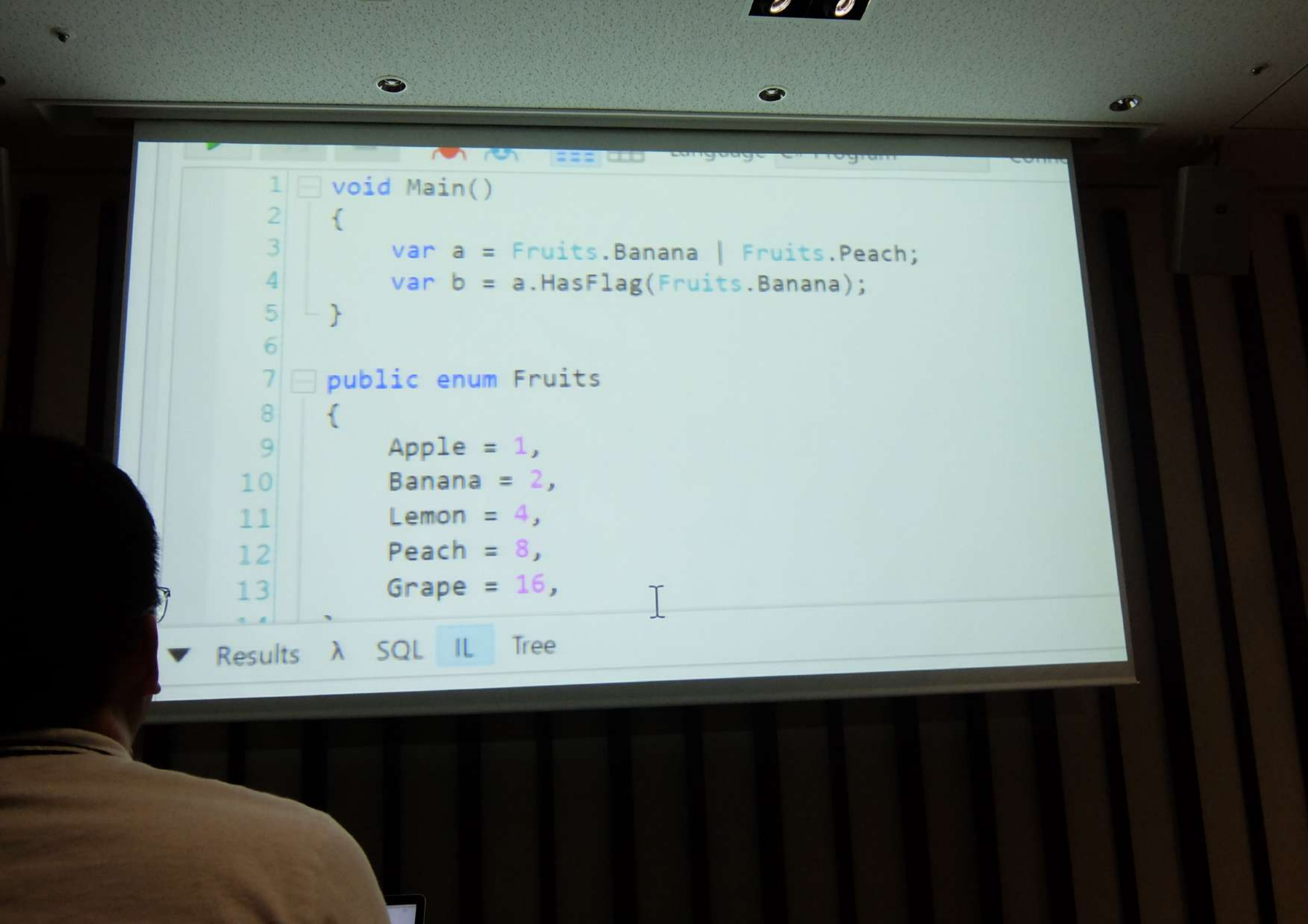
FastEnum が唯一勝てなかったメソッド

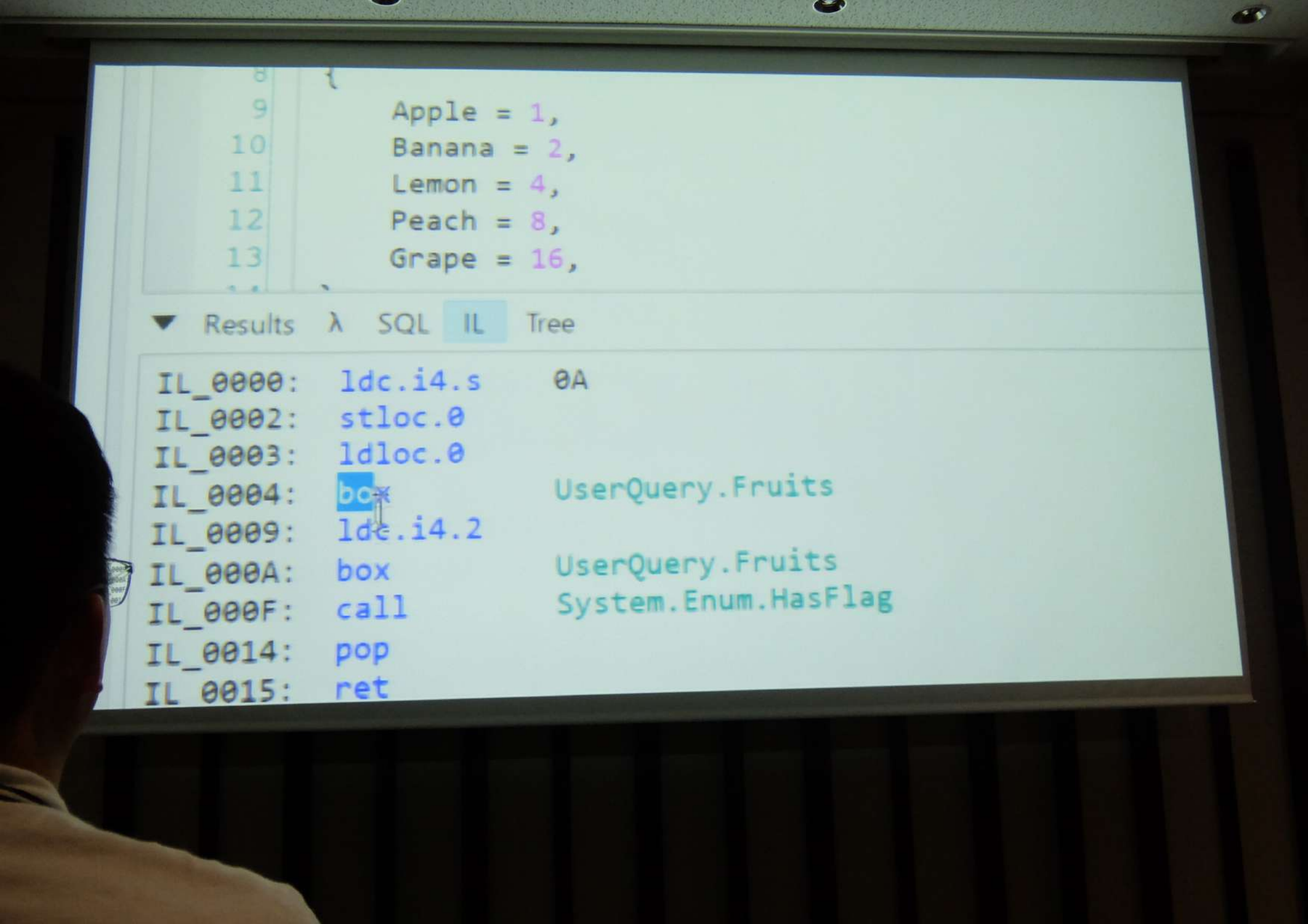
引数が `Enum` なので呼び出すだけで box 化して超遅そうだが…
自作したものより 200 倍以上高速でアロケーションもない

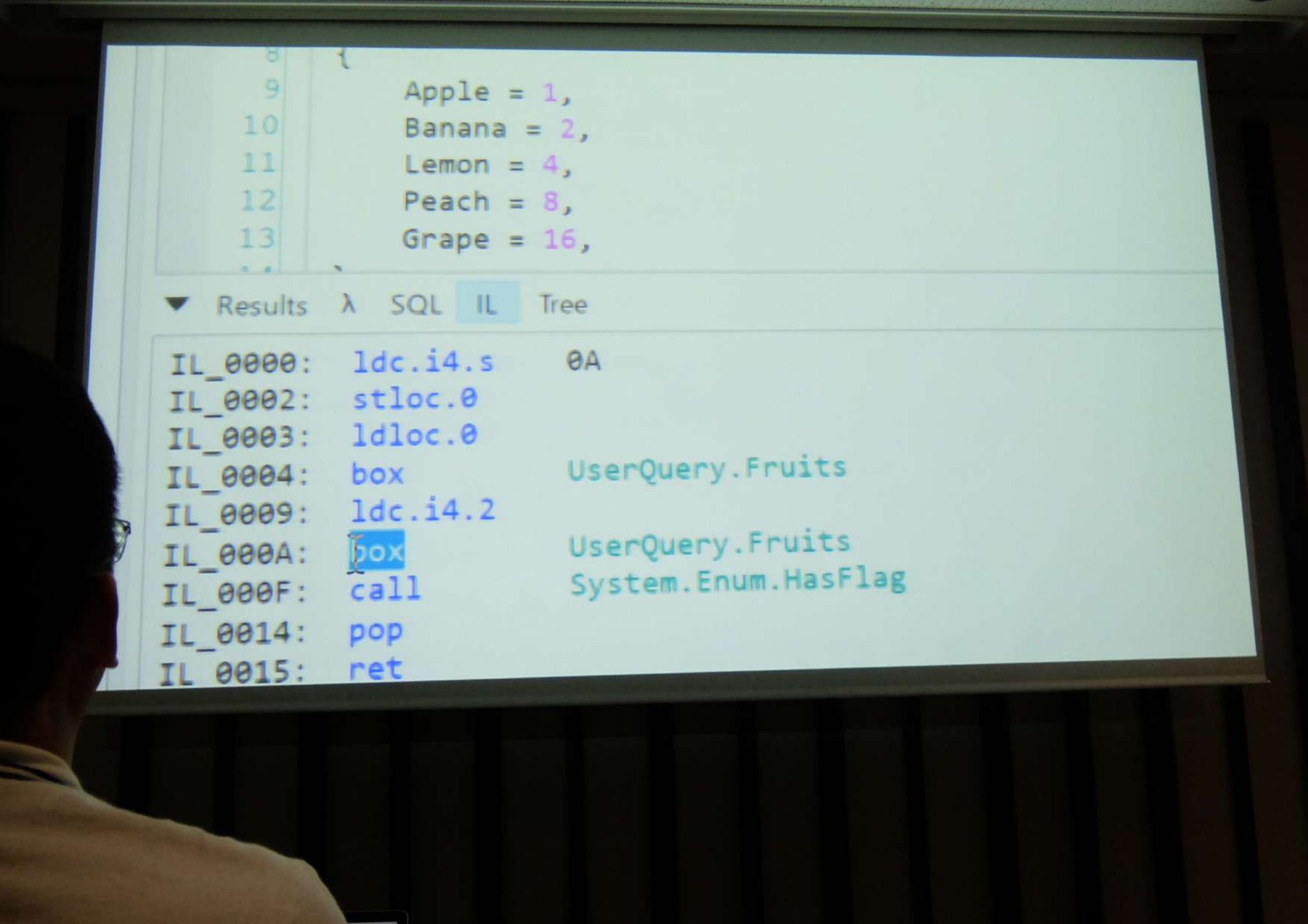
```
// 特殊な最適化がかかってる（たぶん

[Intrinsic]
public bool HasFlag(Enum flag)
{}
```

Method	Mean	Error	StdDev	Median	Ratio	RatioSD	Gen 0	Gen 1	Gen 2	Allocated
NetCore	0.0125 ns	0.3945 ns	0.0216 ns	0.0000 ns	?	?	-	-	-	-
FastEnum	2.2857 ns	3.0054 ns	0.1647 ns	2.3766 ns	?	?	-	-	-	-


Language C# Program
void Main()
{
var a = Fruits.Banana | Fruits.Peach;
var b = a.HasFlag(Fruits.Banana);
}
public enum Fruits
{
Apple = 1,
Banana = 2,
Lemon = 4,
Peach = 8,
Grape = 16,


1 void Main()
2 {
3 var a = Fruits.Banana | Fruits.Peach;
4 var b = a.HasFlag(Fruits.Banana);
5 }
6
7 public enum Fruits
8 {
9 Apple = 1,
10 Banana = 2,
11 Lemon = 4,
12 Peach = 8,
13 Grape = 16,
Results λ SQL IL Tree


{
Apple = 1,
Banana = 2,
Lemon = 4,
Peach = 8,
Grape = 16,
}
Results
λ
SQL
IL
Tree
IL_0000: ldc.i4.s 0A
IL_0002: stloc.0
IL_0003: ldloc.0
IL_0004: box UserQuery.Fruits
IL_0009: ldc.i4.2
IL_000A: box UserQuery.Fruits
IL_000F: call System.Enum.HasFlag
IL_0014: pop
IL_0015: ret


{
Apple = 1,
Banana = 2,
Lemon = 4,
Peach = 8,
Grape = 16,
Results
λ
SQL
IL
Tree
IL_0000: ldc.i4.s 0A
IL_0002: stloc.0
IL_0003: ldloc.0
IL_0004: box UserQuery.Fruits
IL_0009: ldc.i4.2
IL_000A: box UserQuery.Fruits
IL_000F: call System.Enum.HasFlag
IL_0014: pop
IL_0015: ret

ReadOnlyArray

ReadOnlyCollection は foreach が超遅い

Struct Enumerator を利用した配列の読み取り専用 wrapper で解決

https://ufcpp.net/blog/2018/12/howtoenumerate/

```
public sealed class ReadOnlyArray<T> : IReadOnlyList<T>
{
    private readonly T[] source;
    public ReadOnlyArray(T[] source) => this.source = source;

    public Enumerator GetEnumerator() => new Enumerator(this.source);
    public struct Enumerator : IEnumerator<T> { /* 省略 */ }
    // 以下略...
}
```

ReadOnlyArray

ReadOnlyCollection は foreach が超遅い

Struct Enumerator を利用した配列の読み取り専用 wrapper で解決

https://ufcpp.net/blog/2018/12/howtoenumerate/

```
public sealed class ReadOnlyArray<T> : IReadOnlyList<T>
{
    private readonly T[] source;
    public ReadOnlyArray(T[] source) => this.source = source;

    public Enumerator GetEnumerator() => new Enumerator(this.source);
    public struct Enumerator : IEnumerator<T> { /* 省略 */ }
    // 以下略...
}
```