DISCRIMINATED UNIONS LDM UPDATE #2

Matt Warren

+ Bill, Chuck, Cyrus, Fred, Kathleen, Mads

This is a presentation to the C# Language Design Meeting (LDM) from the C# Discriminated Union Working Group.
It is a review of the discussions and investigations that the working group has been having since the last update.
The point is to catch the LDM members up and get feedback from them.

This presentation may be entirely biased by my memory as to what occurred plus a few incoherent notes.
I am expecting other WG members to clarify points and answer questions during the presentation.

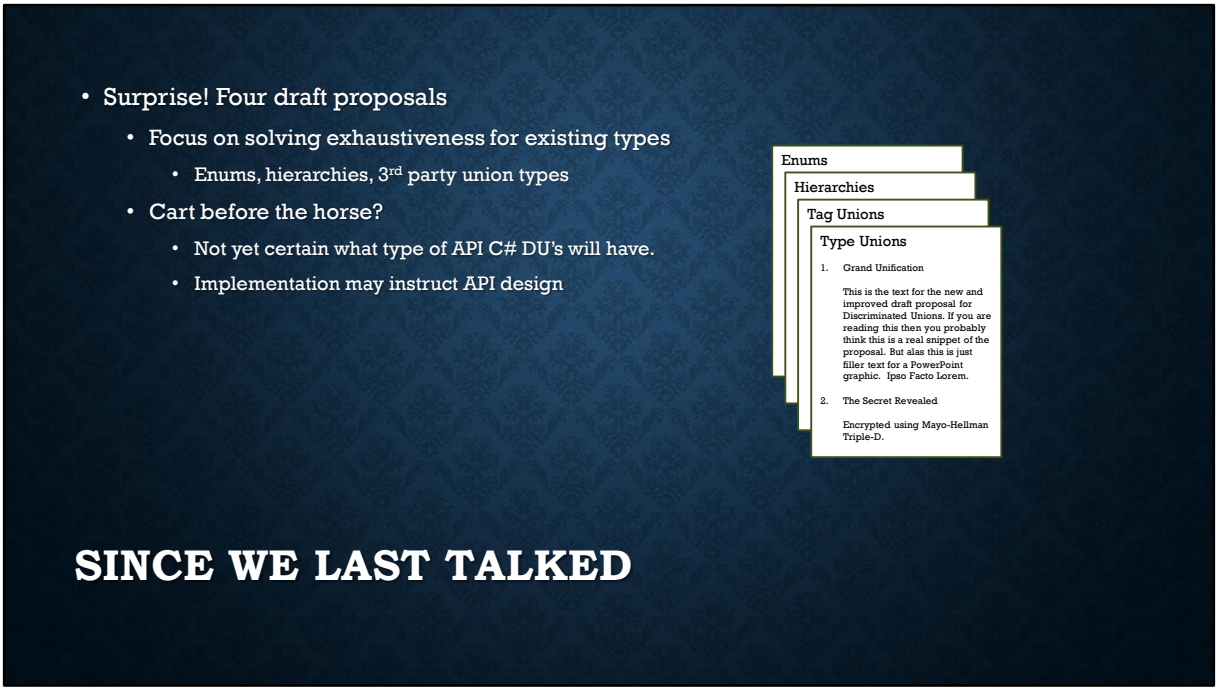
We were focused early on understanding Option, Result and OneOf or equivalent existing 3rd party types, since customers focused on them a lot when we asked about discriminated unions.

There was a creation of a "not the real syntax" pseudo syntax for declaring and referring to union types so we could avoid rat-holing on syntax issues. So, then we promptly forgot the reason why it was a pseudo syntax and kept having discussions on syntax.

By the end of the last period of discussions it became clear that there were really two kinds of union types that we were interested in; one was a "tag union" that was like a classic FP discriminated union with tag states and associated variables where interaction is through matching and deconstruction, and another was a "type union" that just represented a constrained set of possible typed values like Typescript where interaction is often via type tests and variable assignment. Each seemed to have a set of important scenarios they could solve better than the other. There was some discussion whether one solution could subsume all scenarios, and it seemed mathematically that each could in place of the other, if you ignored edge cases, like allocations or interchangeability.

There was a lot of discussion of exhaustiveness, because it was an important to pattern matching and came as an expectation when talking about discriminated unions since that's how its always done in FP languages. Customers coming from FP languages or wanting to move C# in that direction have been telling us to give them exhaustiveness checking for enums and the like already.

When debating solutions for tag and type unions or whether one of the two should reign supreme (in case the appetite for doing both was non-existent) performance due to allocations was a primary concern. It seemed a lot easy to build solutions for tag unions that could minimize or avoid allocations. However, usually these solutions required increase footprints which lead to alternate performance issues. However, by the end of the period and review with the LDM at large, that was a consensus that tag unions were superior and they should be prioritized over type unions.

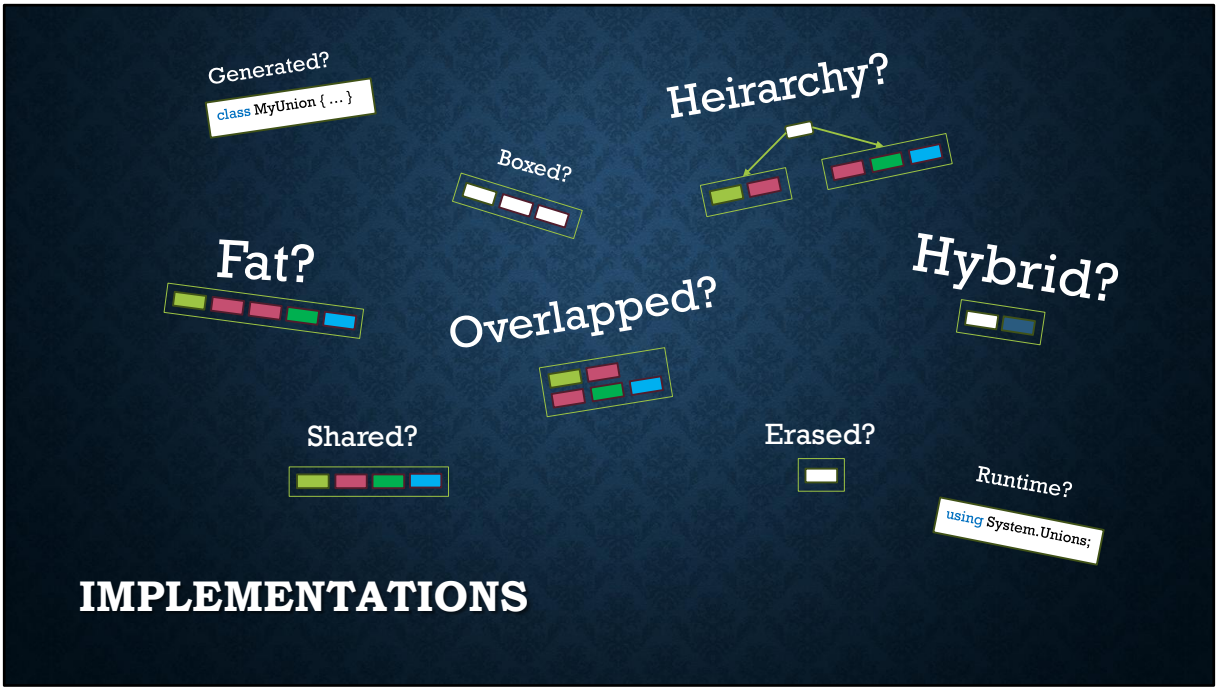


While the design team was on a break (shipping C#) I figured that I could get a jump on some of the "easy" problems and prepare some draft proposals for features that solved problems that would not rely on C# having discriminated union feature.

Since we had focused so much discussion on exhaustiveness and that was not necessarily a DU specific feature, so I figured I would make progress there on my own. I revived the closed enum and closed class hierarchy proposals that Neal had done (rewording them mostly) and created two draft proposals for how to integrate 3rd party union types (tag and type unions) with C# pattern matching.
This is mostly how the compiler would recognize API patterns or attributes that would declare these existing types to be union types and allow them to be matched on in switch and is expressions.

However, the consensus from the other working group members was that we should understand how the union feature in C# was going to work since the API we would generate would probably relate to how the compiler would recognize 3rd party types, and that the implementation details could impact the API design, so we should do the design work first for union types in C# before proposing and implementing support for 3rd party types.

So that’s what happened.



We decided to study a bunch of different possible implementations for tag unions and type unions. Some are represented here.

Hierarchy – this is just a class hierarchy. It relates to how we might represent a tag union, convert tags and associated values into sub-types of a base type that is the union type itself.

Fat – A layout for a tag union where each variable for each tag state is given a field, so it has all the fields for all the cases and thus takes up the most footprint. And then one field for the tag itself.

Shared – A layout similar to Fat but reusing similar typed fields across different tag cases.

Overlapped – A layout similar but with better reuse of memory space, overlapping as much of each case variables as possible, given the constraints of the runtime.

Boxed – A consideration for tag unions was to have a type with N fields, where N was the number of variables for the tag with the most variables, and a field for the tag. Any struct variable would get boxed.

Erased – This technique would require erasing the language concept of union down to just an object value where all struct types included in the union would necessarily be boxed. It only applied to type unions.

Hybrid – A way to cheat boxing allocations sometimes, by pairing an object reference field and an integer bits field, that would allow small structs like primitives to not get boxed.
There was both an alternative to the boxed tag union where each of the N items was a hybrid field, and an alternative to the Erased model, where the union got erased to a hybrid value

There were variations of some of the type union designs where the type was meant to be compile time generated by the compiler and others where it was mean to be a type like OneOf that would be part of the runtime.
There were potentially interested issues with both designs.

Tag unions were only considered as compiler generated types (except for potentially specific cases like Option and Result).

We wanted to evaluate each design on a variety of factors. Some were probably self evident and did not require much to answer but by thinking it through. Others got performance tests so we could compare between them then costs of creating, assigning and accessing values, and the overhead of allocations.

Allocations – We wanted to understand which solutions would require allocations and how those allocation would impact performance.

Footprint – This is a measure of how much space the type itself consumes on the stack; its sizeof at runtime. This affects how much space it takes up inside other structures, arrays and the cost of passing an instance as a parameter.

Performance – Mostly the time cost of creation, assignment and accessing values.

Tearing – This was a big concern brought up since many solutions require struct types with multiple fields.
Tearing potentially occurs when copying structs larger than a single reference (hardware pointer size) into memory that is observable via another thread, like a mutable member in a class.
It is possible to observe the state of the struct mid copy and end up seeing some fields

populated with new data and some fields populated with old data.

Type Parameters – Would generic type parameters be possible, declared with the type or used as variable types within the unions? And how would they impact the possible layout solutions and affect performance.

Ref Types – Whether it would be possible to use ref types as tag union variable types or type union types.

External Types – Whether it would be possible to refer to types defined outside the union as part of the union.

Interop – This is mostly how difficult would it be to use the union type in other languages like VB or F# when exposed via shared APIs.

Untyped matching – Whether it was possible to match on unions stored in variables typed as object (or as a generic type parameter).

- Allocations matter
- Struct or Allocating
- Struct => Tearing (common)
- **Best struct layout: Overlapped > Shared > Fat**
- Struct bad for untyped matching (object or generic)
- Ref types => generated structs
- Interop not ideal with erasure
- **Generic APIs are slow** – ☹ Type Unions fix
- Hybrid (variant) relies on generic API (?)
- **Tag and Type unions same under the hood**

```
interface ITypeUnion
{
   bool IsType<T>();
   bool TryGet<T>(out T value);
   T Get<T>();
}
```

DISCOVERIES

As it turned out, a lot of these things were self evident.

Allocations did matter as much as they ever do. So, if you were using a union type in a perf sensitive scenario, unions causing allocations via boxing or type instantiations would be noticeable.

Any solution not based on a struct union type would likely be inducing allocations.

Any solution based on a struct union type would be subject to tearing, but tearing already happens today and people deal with it by either avoiding mutation or mutating fields with large structs or avoiding using those types with multi-threading.
It is still an issue, since most customers might not even know it's a problem.

The best struct layout was the denser overlapped solutions, since it had the smallest footprint but did not seem to impact any of the measured costs.

Untyped matching was not possible with any solution that was struct based, due to the overhead of checking all pattern matching cases where the source is otherwise typed as object.
The object-erased solution did not have this problem, since it is already a boxed

object.

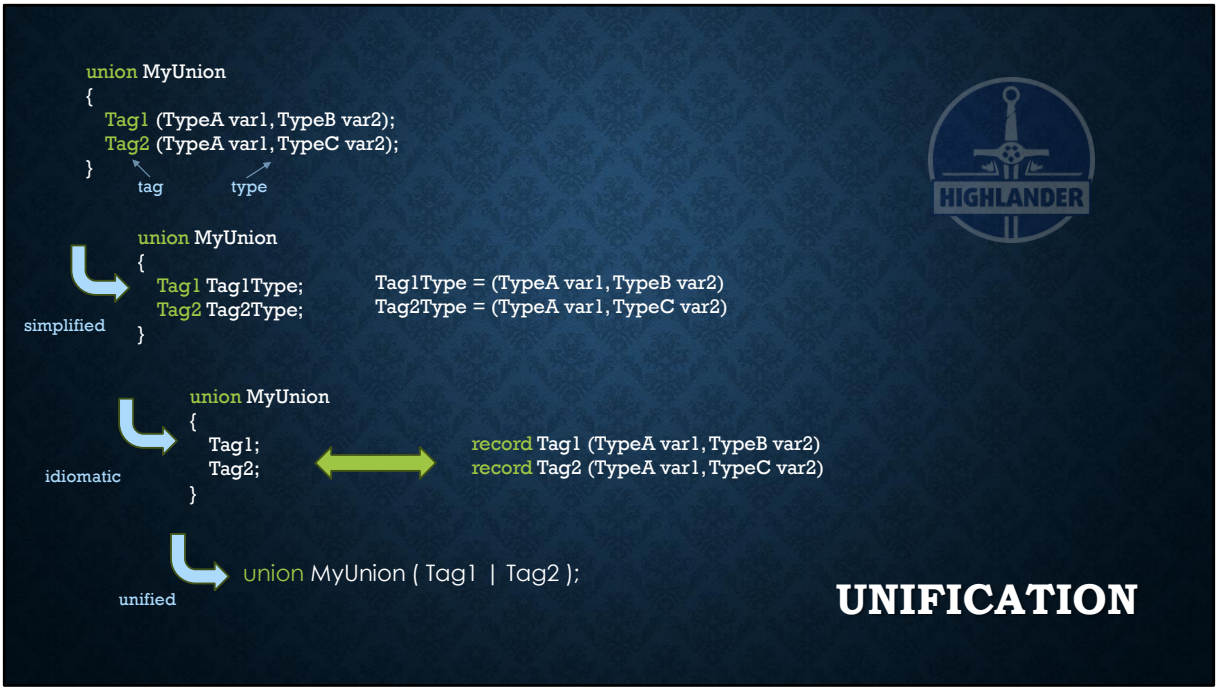
Ref types were only possible in when the union types were being generated by the compiler and as structs, so they could be ref structs themselves.

Interop was not a problem with most solutions, since they had simple API's that could be used in other languages.
Except for the Erased solution, since it lost the knowledge of the type membership and would require other languages to learn how that information was encoded in the API's they were using.
Still, it would be still simple to use erased objects or hybrid values in other languages, but the programmer might not be aware of the closed set of types that were expected and may end up supplying an invalid value to a parameter.

Generic API calls were too slow. This was surprising and mattered a lot to type union API design, which had a set of methods matching the example in the slide. These calls could be an order of magnitude slower that their non-generic counterparts in many of the tag union solutions.
This momentarily seemed like a bit win for the tag-unions only camp. However, it was easy to see that the type unions could also specify a non-generic variation of the API, by including custom accessor for each known type and exposing a state value like an int or enum.
Fixed, the API for tag unions and type unions started looking suspiciously similar.

The hybrid type also relied on generic methods to access values. This problem compounded for solutions that used hybrid fields in types that also had generic accessor. These have been a lot of work to optimize hybrid speed and remove generic accessor requirements. There are known solutions for this.

The implementations of compiler generated type unions turned out to be identical to the implementations for tag unions when the types in the type union are struct types that can be deconstructed, like tuples or record structs) and their identity does not need to be maintained.
The type unions that were comprised of only reference types degenerated to a simple object field in a struct wrapper.



Since much of the type union's public API would now look like the tag union's API for performance (to avoid generic accessors when possible), and since the actual implementation layout of the two would end up being the same for the cases where the type union's types are deconstructable struct types like value tuples and record structs, it is possible not only mathematically/logically for type unions to subsume tag unions, it is also mechanically possible, for cases when the union type is being compiler generated.

It is now no longer necessary to have two concepts for discriminated unions, and just focus on type unions as the kinds of unions that C# would describe.

You can see it fundamentally by starting with a declared tag union (in C# pseudo syntax) and deconstructing it.

The tag union starts as a declaration of one or more tags, each associated with zero or more variables each with a specific name and type.
The next step shows us that we can simplify this relationship by associating each tag with a single value that may be a type like a tuple or it may be nothing (have no fields). If we then upgrade the type to a C# record (giving the tuple a name) it is no longer necessary to carry the tag anymore, since it is part of the type.

Associating variables with a name is what a type does in OOP languages like C#. Which makes a union of named records an idiomatic solution to unions in C#.
From there we transition into ‘other’ type union pseudo syntax, and we see that a C# union of tag types is simply a C# type union, and the only difference is where the tag-type is declared (as part of the union declaration itself) or external to it.

- Priority
  - Option, Result and OneOf
    - Runtime 'blessed' (monadic?)
  - More like typescript, less like FP
- Not Priority
  - Custom unions (yet OneOf?)
  - Allocations and performance
  - Ref types
- Not mentioned
  - Exhaustiveness

MORE CUSTOMER INSIGHT

During this current design period we had more interactions with customers and got more feedback on discriminated unions.

It was interesting to hear that still the priority for customers (at least who we talk to) is primarily the Option, Result and OneOf types that can be found in 3rd party libraries. This may be biased by that being what they know and use now, but also that those familiar with FP languages use these types regularly.
Mostly the sentiment from customers has been that Microsoft 'bless' a set of these types in the runtime so they don't need to rely on agreeing on any library, though that has never really been a problem for other libraries.
A bit of it may be due to FP languages like F# have additional monadic behaviors for types like Option and Result.
We have been focused so far on making sure that C# unions would be able to describe types like Option and Result, but not any additional language behaviors for them.

There seems to be a consensus with those we talk to that C# unions should look and feel more like Typescript union types than class FP discriminated unions. We are not yet hearing the same kind so concerns from Customers that we have ourselves.

When it comes to features that are not priorities (maybe still desirable but not the

most important) are custom unions. Customers don't think they will be declaring their own unions. This may be due to them conflating Option and Result as "Discriminated Unions" and they imagine not needing to declare any other. It may also be due to expecting the existence of a OneOf type, that in abstract does subsume all type union types and thus all tag union types. It may be due to thinking there will be Typescript style anonymous type unions, and they won't need to declare specific named ones and use simply (A | B).

Also, allocations and the performance cost of allocations causes by union types does not seem to be on customer radar. Though this often never is until it is, and it would still be better for us to have a solution for this ready.
Being able to include ref types in a union did not seem to be an issue.

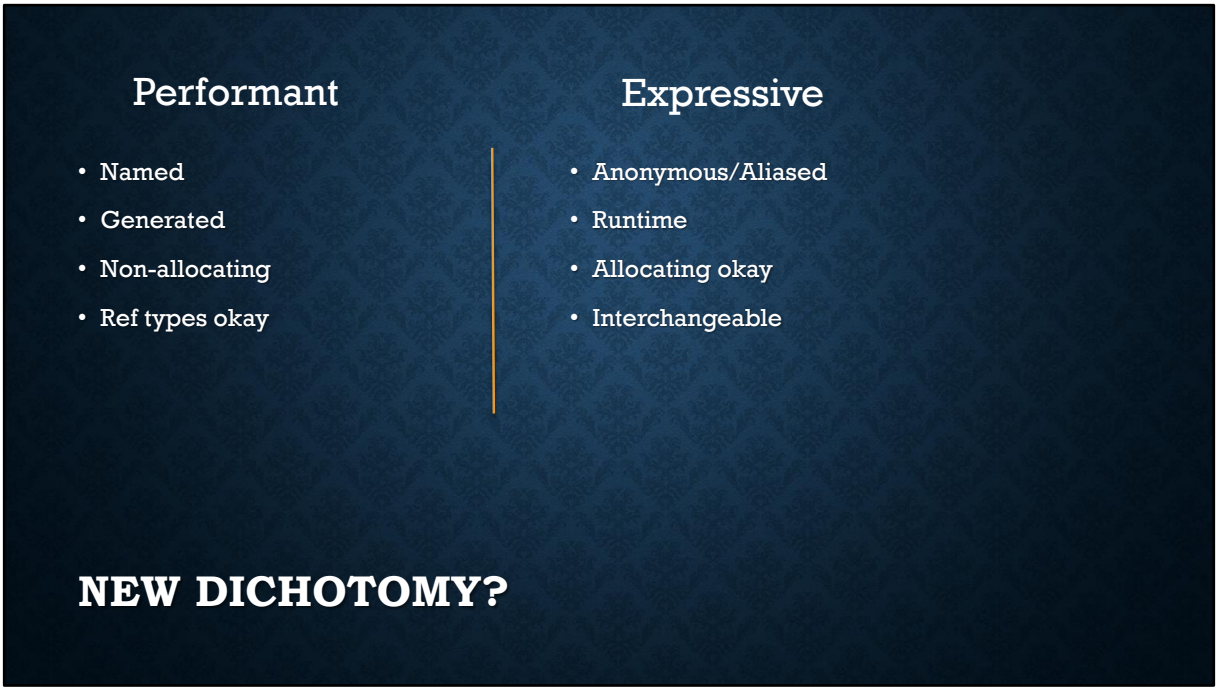
Interesting too was that exhaustiveness checking for unions never seemed to come up, even though it does come up when talking about enums and switch statements (though not using the term exhaustiveness).
It is likely, that it is still an issue and a priority, since an enum is also a degenerate form of a union, but customers are awash in other thoughts and wishes for unions and this has just not come up yet.

- Interchangeability
  - Assignability - (A|B) ab = ba
  - Coercion - (A|B|C) abc = ab
  - Equivalence - List<A|B> == List<B|A>
  - Want **OneOf** but interchangeable
  - Generated types are not interchangeable
  - Erased types are easily interchangeable

AND MORE …

When the focus shifts to Typescript like type unions existing in C#, there is strong desire to have them behave like Typescript unions.

Interchangeability is seen as very important. This an umbrella term we assigned to a category features concerned with the assignability and equivalence of union types. It is very important that if we have anonymous type unions like (A|B) that a list of (A|B)'s be substitutable with a list of (B|A)'s. Just changing the order should not matter. This is easy for Typescript since all such unions are erased to JavaScript objects, but much more difficult for C# types that can be unique and different types at runtime. Note, this is also a problem for existing OneOf types, and it may not be solvable without erasure.



We've solved the differences between tag and type unions, theoretically and mechanically, but we still have a problem.

There still may need to be more that one kind of C# union, since use cases still imply different solutions, even if all the solutions are for type unions.

We still have at least two broad camps that imply different solutions.

The first is labeled "Performant" and it is primarily concerned with avoiding allocations. This requires solutions that are compiler generated with custom implementations and layouts determined by the types involved.
Since they are compiler generated, they can possibly support ref types, often used in other performance sensitive non-allocating scenarios.

The second is labeled "Expressive" and is primarily concerned with being interchangeable. This requires solutions that likely erased to object or other runtime type (hybrid), allocating is an okay tradeoff and are usually anonymous or aliased only as having a true runtime name would require have a separate runtime type and would interfere with interchangeability.

- C# unions let you describe variables that can hold different kinds of values, similar to a variable typed as object or the base type of a hierarchy, except…
  - Type Safe (strongly typed parameters)
  - Mix and Match (types from different hierarchies; Animal | Automobile)
  - No Defaults (exhaustiveness)
  - Performant (when needed; non-allocating + ref types)
  - Concise (like records, no ceremony)

ELEVATOR PITCH?

We did an exercise and tried envision how we might describe discriminated unions to general C# developers.
There were some different takes, I have not provided them all.
This is mine merged with some of the points from others.

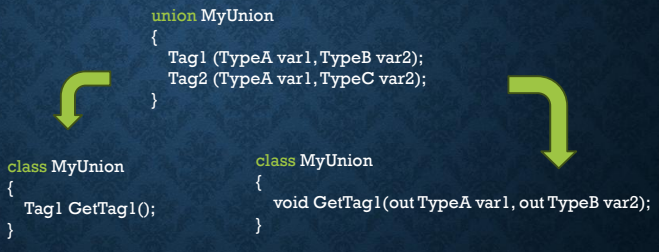
The point of this is to initiate discussion on how we talk about the feature to customers that may be knew to the concept.
How do we express the strengths and use cases?
What are the quick easy ways to explain the benefits?

- Should we try to solve exhaustiveness for existing types separately from language support for DU?
  - A) Yes, get this done now
  - B) No, do it as one package
  - C) No, customers are not really interested in exhaustiveness
  - D) Not sure

LDM QUESTION #1

This is a poke at the LDM, asking to allow for the four earlier proposals I did to move forward, so it can possibly release before the rest of DU.
In a sense, do we know enough yet? Can these move forward separately?

- Should DU in C# be idiomatic to C# and use types or match classic FP style?
  - A) Idiomatic C#; let the types flow
  - B) FP classic; deconstruction all the way
  - C) Both
  - D) Neither
  - E) Not sure



LDM QUESTION #2

The gist here is a question about how we want to think about DU in C# going forward. Do we want to maintain the split we had before with both tag-unions and type-unions? The graphic just shows one part of the difference.
A model described as a union of types would have accessors that return those types, while a model described as a union of tags and associate variables may only have accessors that enable deconstruction.

- Should there be more than one kind of Discriminated Union in C#?
  - A) Yes, Performant + Expressive
  - B) Yes, Performant Tag Union + Performant Type Union + Expressive Type Union
  - C) No, Performant only
  - D) No, Expressive only
  - E) No, Performant Tag Union only
  - F) No, Performant Type Union only
  - G) not sure
  - H) I am so confused

LDM QUESTION #3

This is testing the waters to see if opinions have changed.
We left of last time with a strong contingent focused primarily on having tag unions as the primary or only kind of unions (mostly due to ability to make them non-allocating if necessary).
Probably going to get votes all over the board, but hoping it stimulates conversations.

- Should DU feature be factored into multiple releasable parts?
  - A) Yes, let Jared figure out when the pieces ship
  - B) No, it only makes sense as one big release
  - C) Not sure

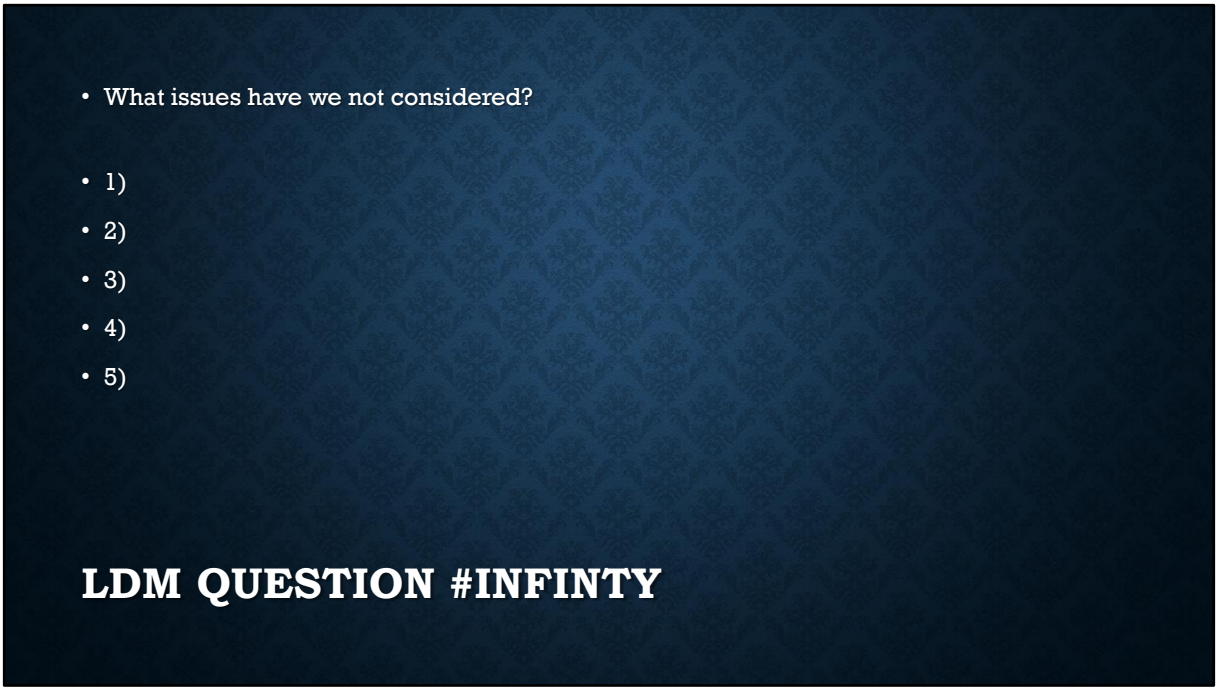
LDM QUESTION #4

This may seem like the same questions as #1 but focused on just the DU feature itself. Basically, should we be designing it to be releasable in chunks over time or one masterpiece.
Actual release decisions happen elsewhere, this is mostly to give designers to express their take on practicality verses perfection.

- Customers focused on Option/Result. Should we go beyond DU and include other common (monadic) behaviors for these specific types?
  - A) Yes, I'm all in
  - B) Maybe later; that goes beyond DU.
  - C) No, that's not C#.
  - D) Not sure

LDM QUESTION #5

I'm not sure what to expect here.
I don't think anyone else has been considering these additional behaviors, or the consequence of having these specific types in the runtime, as opposed to DU's in general.



The open-ended slide added here so LDM members can ask questions, but we never get to because they've already asked questions during the previous slides and we've run out of time.

END OF PRESENTATION