C# Design Notes for Jan 21, 2015 #98

New issue

Closed MadsTorgersen opened this issue on Jan 27 2015 · 249 comments

MadsTorgersen commented on Jan 27 2015 Contributor

C# Design Meeting Notes for Jan 21, 2015

Quotes of the day:

> Live broadcast of design meetings: we could call it C#-SPAN
>
> We've made it three hours without slippery slopes coming up!

Agenda

This is the first design meeting for the version of C# coming after C# 6. We shall colloquially refer to it as C# 7. The meeting focused on setting the stage for the design process and homing in on major themes and features.

1. Design process
2. Themes
3. Features

See also Language features currently under consideration by the language design group.
#1. Design process

We have had great success sharing design notes publicly on CodePlex for the last year of C# 6 design. The ability of the community to see and respond to our thinking in real time has been much appreciated.

This time we want to increase the openness further:

- we involve the community from the beginning of the design cycle (as per these notes!)
- in addition to design notes (now issues on GitHub) we will maintain feature proposals (as checked-in Markdown documents) to reflect the current design of the feature
- we will consider publishing recordings of the design meetings themselves, or even live streaming
- we will consider adding non-Microsoft members to the design team.

Design team

The C# 7 design team currently consists of

- Anders Hejlsberg
- Mads Torgersen
- Lucian Wischik
- Matt Warren
- Neal Gafter
- Anthony D. Green
- Stephen Toub
- Kevin Pilch-Bisson
- Vance Morrison
- Immo Landwerth

Anders, as the chief language architect, has ultimate say, should that ever become necessary. Mads, as the language PM for C#, pulls together the agenda, runs the meetings and takes the notes. (Oooh, the power!)

To begin with, we meet 4 hours a week as we decide on the overall focus areas. There will not be a separate Visual Basic design meeting during this initial period, as many of the overall decisions are likely to apply to both and need to happen in concert.

Assignees

MadsTorgersen

Labels

Area-Language Design

Design Notes

Language-C#

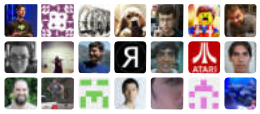
Projects

None yet

Milestone

Unknown

107 participants



and others

Feature ideas

Anyone can put a feature idea up as an *issue* on GitHub. We'll keep an eye on those, and use them as input to language design.

A way to gauge interest in a feature is to put it up on UserVoice, where there's a voting system. This is important, because the set of people who hang out in our GitHub repo are not necessarily representative of our developer base at large.

Design notes

Design notes are point-in-time documents, so we will put them up as *issues* on GitHub. For a period of time, folks can comment on them and the reactions will feed into subsequent meetings.

Owners and proposals

If the design team decides to move on with a feature idea, we'll nominate an *owner* for it, typically among the design team members, who will drive the activities related to the design of that feature: gathering feedback, making progress between meetings, etc. Most importantly, the owner will be responsible for maintaining a *proposal* document that describes the current state of that feature, cross-linking with the design notes where it was discussed.

Since the proposals will evolve over time, they should be documents in the repo, with history tracked. When the proposal is first put up, and if there are major revisions, we will probably put up an issue too, as a place to gather comments. There can also be pull requests to the proposals.

We'll play with this process and find a balance.

Other ways of increasing openness

We are very interested in other ideas, such as publishing recordings (or even live streaming?) of the design meeting themselves, and inviting non-Microsoft luminaries, e.g., from major players in the industry, onto the design team itself. We are certainly open to have "guests" (physical or virtual) when someone has insights that we want to leverage.

However, these are things we can get to over time. We are not going to do them right out of the gate.

Decisions

It's important to note that the C# design team is still in charge of the language. This is not a democratic process. We derive immense value from comments and UserVoice votes, but in the end the governance model for C# is benevolent dictatorship. We think design in a small close-knit group where membership is long-term is the right model for ensuring that C# remains tasteful, consistent, not too big and generally not "designed by committee".

If we don't agree within the design team, that is typically a sign that there are offline activities that can lead to more insight. Usually, at the end of the day, we don't need to vote or have the Language Allfather make a final call.

Prototypes

Ideally we should prototype every feature we discuss, so as to get a good feel fro the feature and allow the best possible feedback from the community. That may note be realistic, but once we have a good candidate feature, we should try to fly it.

The cost of the prototyping is an issue. This may be feature dependent: Sometimes you want a quick throwaway prototype, sometimes it's more the first version of an actual implementation.

Could be done by a member of the design team, the product team or the community.

Agenda

It's usually up to Mads to decide what's ready to discuss. Generally, if a design team member wants something on the agenda, they get it. There's no guarantee that we end up following the plan in the meeting; the published notes will just show the agenda as a summary of what was *actually* discussed.
#2. Themes

If a feature is great, we'll want to add it whether it fits in a theme or not. However, it's useful to have a number of categories that we can rally around, and that can help select features that work well together.

We discussed a number of likely themes to investigate for C# 7.

Working with data

Today's programs are connected and trade in rich, structured data: it's what's on the wire, it's what apps and services produce, manipulate and consume.

Traditional object-oriented modeling is good for many things, but in many ways it deals rather poorly with this setup: it bunches functionality strongly with the data (through encapsulation), and often relies heavily on mutation of that state. It is "behavior-centric" instead of "data-centric".

Functional programming languages are often better set up for this: data is immutable (representing *information*, not *state*), and is manipulated from the outside, using a freely growable and context-dependent set of functions, rather than a fixed set of built-in virtual methods. Let's continue being inspired by functional languages, and in particular other languages – F#, Scala, Swift – that aim to mix functional and object-oriented concepts as smoothly as possible.

Here are some possible C# features that belong under this theme:

- pattern matching
- tuples
- "denotable" anonymous types
- "records" - compact ways of describing shapes
- working with common data structures (List/Dictionary)
- extension members
- slicing
- immutability
- structural typing/shapes?

A number of these features focus on the interplay between "kinds of types" and the ways they are used. It is worth thinking of this as a matrix, that lets you think about language support for e.g. denoting the types (*type expressions*), creating values of them (*literals*) and consuming them with matching (*patterns*) :

Type	Denote	Create	Match
General	`T`	`new T()` , `new T { x = e }`	`T x` , `var x` , `*`
Primitive	`int` , `double` , `bool`	`5` , `.234` , `false`	`5` , `.234` , `false`
String	`string`	`"Hello"`	`"Hello"`
Tuple	`(T1, T2)`	`(e1, e2)`	`(P1, P2)`
Record	`{ T1 x1, T2 x2 }`	`new { x1 = e1, x2 = e2 }`	`{ x1 is P1, x2 is P2 }`
Array	`T[]`	`new T[e]` , `{ e1, e2 }`	`{ P1, P2 }` , `P1 :: P2`
List	?	?	?
Dictionary	?	?	?
...			

A lot of the matrix above is filled in with speculative syntax, just to give an idea of how it could be used.

We expect to give many of the features on the list above a lot of attention over the coming months: they have a lot of potential for synergy if they are designed together.

Performance and reliability (and interop)

C# and .NET has a heritage where it sometimes plays a bit fast and loose with both performance and reliability.

While (unlike, say, Java) it has structs and reified generics, there are still places where it is hard to get good performance. A top issue, for instance is the frequent need to copy, rather than reference. When devices are small and cloud compute cycles come with a cost, performance certainly starts to matter more than it used to.

On the reliability side, while (unlike, say, C and C++) C# is generally memory safe, there are certainly places where it is hard to control or trust exactly what is going on (e.g., destruction/finalization).

Many of these issues tend to show up in particular on the boundary to unmanaged code - i.e. when doing interop. Having coarse-grained interop isn't always an option, so the less it costs and the less risky it is to cross the boundary, the better.

Internally at Microsoft there have been research projects to investigate options here. Some of the outcomes are now ripe to feed into the design of C# itself, while others can affect the .NET Framework, result in useful Roslyn analyzers, etc.

Over the coming months we will take several of these problems and ideas and see if we can find great ways of putting them in the hands of C# developers.

Componentization

The once set-in-stone issue of how .NET programs are factored and combined is now under rapid evolution.

With generalized extension members as an exception, most work here may not fall in the language scope, but is more tooling-oriented:

- generating reference assemblies
- static linking instead of IL merge
- determinism
- NuGet support
- versioning and adaptive light-up

This is a theme that shouldn't be driven primarily from the languages, but we should be open to support at the language level.

Distribution

There may be interesting things we can do specifically to help with the distributed nature of modern computing.

- Async sequences: We introduced single-value asynchrony in C# 5, but do not yet have a satisfactory approach to asynchronous sequences or streams
- Serialization: we may no longer be into directly providing built-in serialization, but we need to make sure we make it reasonable to custom-serialize data - even when it's immutable, and without requiring costly reflection.

Also, await in catch and finally probably didn't make it into VB 14. We should add those the next time around.

Metaprogramming

Metaprogramming has been around as a theme on the radar for a long time, and arguably Roslyn is a big metaprogramming project aimed at writing programs about programs. However, at the language level we continue not to have a particularly good handle on metaprogramming.

Extention methods and partial classes both feel like features that could grow into allowing *generated* parts of source code to merge smoothly with *hand-written* parts. But if generated parts are themselves the result of language syntax - e.g. attributes in source code, then things quickly get messy from a tooling perspective. A keystroke in file A may cause different code to be generated into file B by some custom program, which in turn may change the meaning of A. Not a feedback loop we're eager to have to handle in real time at 20 ms keystroke speed!

Oftentimes the eagerness to generate source comes from it being too hard to express your concept beautifully as a library or an abstraction. Increasing the power of abstraction mechanisms in the language itself, or just the syntax for applying them, might remove a lot of the motivation for generated boilerplate code.

Features that may reduce the need for boilerplate and codegen:

- Virtual extension methods/default interface implementations
- Improvements to generic constraints, e.g.:
  - generic constructor constraints
  - delegate and enum constraints
  - operators or object shapes as constraints (or interfaces), e.g. similar to C++ concepts
- mixins or traits
- delegation

Null

With null-conditional operators such as `x?.y` C# 6 starts down a path of more null-tolerant operations. You could certainly imagine taking that further to allow e.g. awaiting or foreach'ing null, etc.

On top of that, there's a long-standing request for non-nullable reference types, where the type system helps you ensure that a value can't be null, and therefore is safe to access.

Importantly such a feature might go along well with proper safe *nullable* reference types, where you simply cannot access the members until you've checked for null. This would go great with pattern matching!

Of course that'd be a lot of new expressiveness, and we'd have to reconcile a lot of things to keep it compatible. In his blog, Eric Lippert mentions a number of reasons why non-nullable reference types would be next to impossible to fully guarantee. To be fully supported, they would also have to be known to the runtime; they couldn't just be handled by the compiler.

Of course we could try to settle for a less ambitious approach. Finding the right balance here is crucial.

Themeless in Seattle

Type providers: This is a whole different kind of language feature, currently known only from F#. We wouldn't be able to just grab F#'s model though; there'd be a whole lot of design work to get this one right!

Better better betterness: In C# we made some simplifications and generalizations to overload resolution, affectionately known as "better betterness". We could think of more ways to improve overload resolution; e.g. tie breaking on staticness or whether constraints match, instead of giving compiler errors when other candidates would work.

Scripting: The scripting dialect of C# includes features not currently allowed in C# "proper": statements and member declarations at the top level. We could consider adopting some of them.

params IEnumerable.

Binary literals and digit separators.
#3. Features

The Matrix above represents a feature set that's strongly connected, and should probably be talked about together: we can add kinds of types (e.g. tuples, records), we can add syntax for representing those types or creating instances of them, and we can add ways to match them as part of a greater pattern matching scheme.

Pattern matching

Core then is to have a pattern matching framework in the language: A way of asking if a piece of data has a particular shape, and if so, extracting pieces of it.

```
if (o is Point(var x, 5)) ...
```

There are probably at least two ways you want to use "patterns":

1. As part of an expression, where the result is a bool signaling whether the pattern matched a given value, and where variables in the pattern are in scope throughout the statement in which the pattern occurs.
2. As a case in a switch statement, where the case is picked if the pattern matches, and the variables in the pattern are in scope throughout the statements of that case.

A strong candidate syntax for the expression syntax is a generalization of the `is` expression: we consider the type in an `is` expression just a special case, and start allowing any pattern on the right hand side. Thus, the following would be valid `is` expressions:

```
if (o is Point(*, 5) p) Console.WriteLine(o.x);
if (o is Point p) Console.WriteLine(p.x);
if (p is (var x, 5) ...
```

Variable declarations in an expression would have the same scope questions as declaration expressions did.

A strong candidate for the switch syntax is to simply generalize current switch statements so that

- the switch expression can be any type
- the case labels can contain patterns, not just constants
- the cases are checked in order of appearance, since they can now overlap

```
switch (o) {
case string s:
    Console.WriteLine(s);
    break;
case int i:
    Console.WriteLine($"Number {i}");
    break;
case Point(int x, int y):
    Console.WriteLine("({x},{y})");
    break;
case null:
    Console.WriteLine("<null>);
    break
}
```

Other syntaxes you can think of:

Expression-based switch: An expression form where you can have multiple cases, each producing a result value of the same type.

Unconditional deconstruction: It might be useful to separate the deconstruction functionality out from the checking, and be able to unconditionally extract parts from a value that you know the type of:

```
(var x, var y) = getPoint();
```

There is a potential issue here where the value could be null, and there's no check for it. It's probably ok to have a null reference exception in this case.

It would be a design goal to have symmetry between construction and deconstruction syntaxes.

Patterns *at least* have type testing, value comparison and deconstruction aspects to them.

There may be ways for a type to specify its deconstruction syntax.

In addition it is worth considering something along the lines of "active patterns", where a type can specify logic to determine whether a pattern applies to it or not.

Imagine positional deconstruction or active patterns could be expressed with certain methods:

```
class Point {
    public Point(int x, int y) {...}
    void  econstruct(out int x, out int y) { ... }
    static bool Match(Point p, out int x, out int y) ...
    static bool Match(  b ect  son, out int x, out int y) ...
}
```

We could imagine separate syntax for specifying this.

One pattern that does not put new requirements on the type is matching against properties/fields:

```
if (o is Point {   is var x,   is   }) ...
```

Open question: are the variables from patterns mutable?

This has a strong similarity to declaration expressions, and they could coexist, with shared scope rules.

Records

Let's not go deep on records now, but we are aware that we need to reconcile them with primary constructors, as well as with pattern matching.

Array Slices

One feature that could lead to a lot of efficiency would be the ability to have "windows" into arrays - or even onto unmanaged swaths of memory passed along through interop. The amount of copying that could be avoided in some scenarios is probably very significant.

Array slices represent an interesting design dilemma between performance and usability. There is nothing about an array slice that is functionally different from an array: You can get its length and access its elements. For all intents and purposes they are indistinguishable. So the best user experience would certainly be that slices just *are* arrays - that they share the same type. That way, all the existing code that operates on arrays can work on slices too, without modification.

Of course this would require quite a change to the runtime. The performance consequences of that could be negative even on the existing kind of arrays. As importantly, slices themselves would be more efficiently represented by a struct type, and for high-perf scenarios, having to allocate a heap object for them might be prohibitive.

One intermediate approach might be to have slices be a struct type Slice, but to let it implicitly convert to T [] in such a way that the underlying storage is still shared. That way you can use Slice for high performance slice manipulation (e.g. in recursive algorithms where you keep subdividing), but still make use of existing array-based APIs at the cost of a boxing-like conversion allocating a small object.

ref locals and ref returns

Just like the language today has ref parameters, we could allow locals and even return values to be by `ref`. This would be particularly useful for interop scenarios, but could in general help avoid copying. Essentially you could return a "safe pointer" e.g. to a slot in an array.

The runtime already fully allows this, so it would just be a matter of surfacing it in the language syntax. It may come with a significant conceptual burden, however. If a method call can return a *variable* as opposed to a *value*, does that mean you can now assign to it?:

```
m(x, y) = 5;
```

You can now imagine getter-only properties or indexers returning refs that can be assigned to. Would this be quite confusing?

There would probably need to be some pretty restrictive guidelines about how and why this is used.

readonly parameters and locals

Parameters and locals can be captured by lambdas and thereby accessed concurrently, but there's no way to protect them from shared-mutual-state issues: they can't be readonly.

In general, most parameters and many locals are never intended to be assigned to after they get their initial value. Allowing `readonly` on them would express that intent clearly.

One problem is that this feature might be an "attractive nuisance". Whereas the "right thing" to do would nearly always be to make parameters and locals readonly, it would clutter the code significantly to do so.

An idea to partly alleviate this is to allow the combination `readonly var` on a local variable to be contracted to `val` or something short like that. More generally we could try to simply think of a shorter keyword than the established `readonly` to express the readonly-ness.

Lambda capture lists

Lambda expressions can refer to enclosing variables:

```
var name =  etName();
var  uery = customers.Where(c => c.Name == name);
```

This has a number of consequences, all transparent to the developer:

- the local variable is lifted to a field in a heap-allocated object
- concurrent runs of the lambda may access and even modify the field at the same time
- because of implementation tradeoffs the content of the variable may be kept live by the GC, sometimes even after lambdas directly using them cease to exist.

For these reasons, the recently introduced lambdas in C++ offer the possibility for a lambda to explicitly specify what can be captured (and how). We could consider a similar feature, e.g.:

```
var name =  etName();
var  uery = customers.Where([name]c => c.Name == name);
```

This ensures that the lambda only captures `name` and no other variable. In a way the most useful annotation would be the empty `[]`, making sure that the lambda is never accidentally modified to capture *anything*.

One problem is that it frankly looks horrible. There are probably other syntaxes we could consider. Indeed we need to think about the possibility that we would ever add nested functions or class declarations: whatever capture specification syntax we come up with would have to also work for them.

C# always captures "by reference": the lambda can observe and effect changes to the original variable. An option with capture lists would be to allow other modes of capture, notable "by value", where the variable is copied rather than lifted:

```
var name =  etName();
var  uery = customers.Where([val name]c => c.Name == name);
```

This might not be *too* useful, as it has the same effect as introducing another local initialized to the value of the original one, and then capture *that* instead.

If we don't want capture list as a full-blown feature, we could consider allowing attributes on lambdas and then having a Roslyn analyzer check that the capture is as specified.

Method contracts

.NET already has a contract system, that allows annotation of methods with pre- and post-conditions. It grew out of the Spec# research project, and requires post-compile IL rewriting to take full effect. Because it has no language syntax, specifying the contracts can get pretty ugly.

It has often been proposed that we should add specific contract syntax:

```
public void  emove(string item)
    re uires item  = null
    ensures Count >=
{
   ...
}
```

One radical idea is for these contracts to be purely runtime enforced: they would simply turn into checks throwing exceptions (or FailFast'ing - an approach that would need further discussion, but seems very attractive).

When you think about how much code is currently occupied with arguments and result checking, this certainly seems like an attractive way to reduce code bloat and improve readability.

Furthermore, the contracts can produce metadata that can be picked up and displayed by tools.

You could imagine dedicated syntax for common cases - notably null checks. Maybe that is the way we get some non-nullability into the system?

1

MadsTorgersen added Language-C# Area-Design Notes labels on Jan 27 2015

svick commented on Jan 28 2015 — Contributor

So, array slices are basically an improved version of `rray egment<T>` ?

s-aida commented on Jan 28 2015

While Pattern matching is awesome, I'm still disappointed to know that `break;` is mandatory. I'm pretty sure you've had lots of discussions in codeplex or elsewhere (which I've never read) so no need for rationale or history behind this, but that's about the only part of C# which I have to feel is *lamely* designed.

MadsTorgersen commented on Jan 28 2015 — Contributor

@svick: Yeah I guess. With possible language support.
@shunsukeaida: this is totally a straw man syntax - we aren't even close to settling on a specific syntax. However, if we want to fit pattern matching into switch, we have to kind of blend in well. One might argue that we could just get rid of break, and make it implicit and the end of a case block. That's certainly worth considering.

MadsTorgersen was assigned by **theoy** on Jan 28 2015

theoy added this to the **Unknown** milestone on Jan 28 2015

agocke commented on Jan 28 2015 — Contributor

@MadsTorgersen Something not mentioned in relation to pattern matching: have discriminated unions been considered?

ashmind commented on Jan 28 2015

@MadsTorgersen @shunsukeaida I proposed a better switch `syntax` on CodePlex (independently from destructuring) https://roslyn.codeplex.com/discussions/565550, though it might not work with labels.

I think a better `switch` syntax is rather important whether pattern matching gets into the language or not.

ashmind commented on Jan 28 2015

> Anyone can put a feature idea up as an issue on GitHub. We'll keep an eye on those, and use them as input to language design.

Is there some way to distinguish those from e.g. Roslyn bugs? E.g. some title prefix or so? Labels are good but I can't assign them when reporting the issue I believe.

And I think it would be cool to have some minimal format for it, e.g. at least require the description of the problem, a proposed solution, some use cases using the proposed solution. BCL team seems to be doing a good job there with speclets.

Also I think some certain yes/no on proposals would be great, so that any GitHub proposal is not only considered, but either rejected, accepted for prototyping, or sent to rework due to some design constraints. Again I think BCL team is doing great with their periodic review approach.

jods4 commented on Jan 28 2015

Here's a theme suggestion.

With multi-core becoming commonplace and the nice Task and TPL apis, I see more and more concurrent programming, even if just a few (2-3) background threads.

It would be very nice to get some help from C# regarding correctness here. Immutable data structures are a big help, but for mutable data the main issue is that nothing formally indicates which thread has data ownership, or if data is shared in a read-only fashion, or if data must be accessed inside a lock. In complex programs and large teams this stuff is really hard to get right.

C# was a rockstar when it introduced `async` to help write asynchronous code. I know it's a very hard topic but I would love to see C# bring safe concurrent programming to mainstream.

daveaglick commented on Jan 28 2015 Contributor

I would love to see more native language support for metaprogramming. Mixins and/or traits would certainly help with code reuse challenges and I would personally love to see one of these make it in. While T4 templates work okay for code generation, they often feel clunky and hacky (for example, even though it's a tooling thing, the fact that VS addins are needed just to consistently evaluate them is a clue something is off). Some kind of language support or hook for code generation would be welcome. I would also love to see some sort of rudimentary support for aspects and weaving. PostSharp works well and has a nice API, but requires extra libraries and IL rewriting. Fody aspects work, but there's a really steep learning curve. I'm guessing full AOP support would be way out of scope for this language update, but incremental support or even hooks (perhaps in Roslyn) would be welcomed.

There was also mention of improved overload resolution. This has been a particular pain point of mine, especially when dealing with generics and extension methods. For example, it would be awesome if generic constraints were taken into account when determining a match. I understand the reason why they're not considered (I.e., not part of the method signature) but it would be great if there were a way to overcome this.

jods4 commented on Jan 28 2015

+1 AOP can have lots of useful applications, some of them have been requested by the community for a long time. A few examples:

- Automatic `NotifyPropertyChanged` implementation.
- Automatic check for null argument on public methods (I think the ASP.NET team does that for vNext).
- Automatic implementation of a "IsDirty" flag on model entities.
- Wrapping logging, security or error handlers around services.
  And the list goes on!

svick commented on Jan 28 2015 Contributor

@jods4 Of interest may be the fact that **@jaredpar**, one of the coauthors of the *Uniqueness and Reference Immutability for Safe Parallelism* paper, is now on the Roslyn team and said:

> a part of my job will be trying to integrate some of the goodness that came out of our research back into the language

bojanrajkovic commented on Jan 28 2015

I think `let` might be a useful convention for `readonly var`, or even `const`, though that's probably leading too much down the path of too many meanings for `const` that C/C++ fall prey to.

biboudis commented on Jan 28 2015

@MadsTorgersen What do you mean by delegation? Following theories of incomplete objects that get composed by delegation with proper late-binding semantics maybe?

apskim commented on Jan 28 2015

> Something not mentioned in relation to pattern matching: have discriminated unions been considered?

@agocke Yes, in the original proposal discriminated unions were implemented with record subclassing — similar to case classes in Scala.

@MadsTorgersen By the way, regarding the state of pattern matching. If you do decide to consider them for C# 7, will the first prototype be merged into the current Github source as a feature branch? Will the community be able to contribute to such prototypes? I think you've mentioned before that you want to simplify the PR process in a couple of months, so that external feature prototyping/bug fixing would be easier.

Romoku commented on Jan 28 2015

Non-nullable reference types might be better served with another approach. Assess the feasibility of CLR support for describing a non-null reference type. Even if C# cannot currently take advantage of non-nullable reference types there might be another .NET language that figures out how.

horaciojcfilho commented on Jan 28 2015

Not about new Visual Basic language version? Is Visual Basic coming to die or was placed in the background? I love VB and I'd love to hear about new features 😄 for it.

Romoku commented on Jan 28 2015

@HoracioFilho They mentioned that they are planning the features for both C# and VB right now.

> To begin with, we meet 4 hours a week as we decide on the overall focus areas. There will not be a separate Visual Basic design meeting during this initial period, as many of the overall decisions are likely to apply to both and need to happen in concert.

damageboy commented on Jan 28 2015

One thing HPC like devs could benefit from, is have the ability to "construct" primitive array types off of memory mapped file, like java's nio supports...

I realize this is mainly a runtime feature, rather than a c# feature, is there a "better" place to discuss those?

jvlppm commented on Jan 28 2015

Something that I missed before, and reactive extensions almost gave it to me:
Ability to combine await and yield, resulting in an asynchronous sequence.

```
async var  enerateNumbers()
{
    int i =  ;
    while (true)
    {
        await Task. elay(Time pan. rom econds(1));
        yield return i  ;
    }
}
```

```
foreach(await int item in  enerateNumbers())
{
    WriteLine(" tem:  {item}");
}
```

jonpryor commented on Jan 28 2015

`readonly var` could be abbreviated as `let`, which is already a contextual keyword within query comprehension expressions and semantically declares a `readonly` variable...

👍 1

MrDoomBringer commented on Jan 28 2015

+1 to the Contracts system. There's already a similar concept with type constraints for generics.

```
class  oo<T> where T :  Comparable { }
```

An extension to this pattern for functions would be intuitive to anyone familiar with constraints.

```
void  oo(string bar) where bar  = null, Count >   { }
```

Would separate 'ensures' and 'requires' actually be necessary? Could the statement be inferred?

👍 1

MadsTorgersen commented on Jan 28 2015 Contributor

@agocke: We are trying to come up with a general framework for pattern matching over classes. Ideally there wouldn't be a new "thing" in the language called algebraic datatypes or discriminated unions. Instead there may be compact syntax for describing hierarchies of simple value types (that would happen to be well suited for pattern matching) as a shorthand for class declarations.

@ashmind: yeah, maybe it's time to revamp switch. Worth a think.

@ashmind: I think we'll go through and apply the labels. We'll build up our processes as we go - the BCL team are ahead of us here, and we'll draw on their experiences.

@jods4: safe concurrency is an insanely hard problem. Not sure what we'd do.

@somedave: realistically I can't see us doing something about metaprogramming at the language level before we have a very clear idea of a) what would really move the needle for a lot of developers, and b) how to deal with it at a tooling level.

@jods4: AOP is notoriously hard to reason about for a developer. It's a very big hammer.

@bojanrajkovic, **@jonpryor**: `let` is certainly a candidate.

@biboudis: Full-blown delegation (with this-binding and everything) seems out of the question - that's a whole alternative mechanism to inheritance. But some languages have this nifty little implement-interface-by-redirection trick. Not sure if it's worthwhile or even sane.

@apskim: No clear decisions yet as to how we will share prototypes. For the time being, most of the team is heads down making the *upcoming* version great. This further-out blue sky stuff won't get priority until that's out the door. Yes, would be great to involve the community in prototyping.

@Romoku: I'd love to get non-nullable reference types into the CLR. It seems a tall order even there: what's inside an array of non-nullable reference type when it's first created? etc.

@HoracioFilho: Yeah, **@Romoku** has it right. It doesn't make sense to start separate efforts. Let's get the broad brushstrokes worked out and then split out to language-lawyer on the syntax. :-)

@damageboy: If I understand you correctly, yes that would be an awesome interop feature: Array-typed "windows" onto native data. Privileged (and unsafe!) methods would create them, but be able to share them with your code.

@jvlppm: Yes! When I mentioned "async sequences" this is the kind of thing I had in mind.

@MrDoomBringer: Whatever the syntax, it would certainly occupy the same "region" of a method declaration as the constraints. Maybe it makes sense to merge them.

agocke commented on Jan 28 2015 Contributor

@apskim @MadsTorgersen

Technically, you're correct, but there are actually no new features there specific to discriminated unions. :) The method of generating discriminated unions in the draft specification has one (to me very significant) weakness -- it has no specification of completeness checking in switch statements right now. One way to add this would be to add specialized support just for discriminated unions, for example coming up with a new syntax for `enum` which desugers to a discriminated union behind the scenes.

Alternatively, I would like a spec addition that guarantees that the following data structure provides completeness checking in a switch statement.

```
public abstract class  inaryTree<T>
{
  private  inaryTree() {};
  public sealed class Leaf< >() :  inaryTree< >;
  public sealed class Node< >(  item,   left,   right) :  inaryTree< >;
}
```

It's a little more heavyweight than some *de novo* syntax, but it works. Obviously I would be volunteering to implement this warning. ;)

Romoku commented on Jan 28 2015

@MadsTorgersen From what I understand of how Eric Lippert and Spec# describe non-null references types the contents of the array cannot be null when *observed* during the program execution. This means that up to the point of observance the state can be undetermined, but there **must** be a value at the point of observance.

jamesbascle commented on Jan 28 2015

@somedave Agreed on the AOP.

PostSharp is amazing, but It can be hard to keep it working right on a large team of developers, and it's pretty expensive for all the bells and whistles for a large team as well, even if only a couple of people are even writing the code.

Some language level support for generic argument and result manipulation on methods (like OnMethodBoundaryAspect) alone would be a huge step to meeting the need for metaprogramming and supplanting the wonkiness of PostSharp.

jamesbascle commented on Jan 28 2015

@MadsTorgersen
The PostSharp VisualStudio add-on seems to allow one to easily see which methods are manipulated by which aspects, both at the "This Method is manipulated by This Aspect" and "This Aspect manipulates These Methods".

We've saved ourselves literally thousands of lines of code, but it gets pretty expensive and it's a pretty obscure technical expense that is hard to justify to management types.

It would also be nice if C#/.NET were the first language/framework with anything approaching the kind of reach it has to REALLY have built-in first-class AOP facilities in a language. AspectJ and the things found in some of the dynamic languages or the LISPs are not exactly getting a lot of attention these days. I'd personally see it as a huge huge win for this new open design process from a developer perspective.

damageboy commented on Jan 28 2015

@MadsTorgersen you understood me perfectly

ufcpp commented on Jan 28 2015

How about Units of Measure (like F#)?
From the perspective of performance and convenience, I prefer to use primitive values (int ,byte, etc.), but at the same time, from the perspective of reliability, I'd like to use user-defined values. This dilemma can be solved by Units of Measure.

chrisaut commented on Jan 28 2015

+1 for Contracts from me.

The current solution seems half hearted and not really supported very well.

Non-nullness could IMO be left to contracts, IF there is a static analyzer that can enforce this.

If they are to become a language feature, common cases could be expressed much shorter.
I'm thinking something like:

```
public int >=   M(string  s, double < 1   y, MyClass c)
    re uires c. s nitiali ed
{}
```

which would be the same as

```
public int M(string s, double y)
    re uires s  = null,
    re uires y < 1  ,
    re uires c.. s nitiali ed
    ensures result >=
{}
```

However it could looks a bit messy. Thoughts?

PS: the example

```
if (o is Point(*, 5) p) Console.WriteLine(o.x);
```

should that be

```
if (o is Point(*, 5) p) Console.WriteLine(p.x);
```

or am I misunderstanding?

TuckerDowns commented on Jan 28 2015

Sorry to leave a non constructive comment here but I do have a question related to this process of specifying and designing C#7.

I am interested in watching and studying this process and caught the note about this being broadcast live. What is the best way to follow along and maybe watch these meetings?

I am an undergraduate student who just started a class on language specifications and design and it's definitely one of the most interesting topics I have studied thus far. I don't really use c# but I am really interested in studying the process of language design and that seems to be (partly or mostly) independent of the exact language being specified. I also think it's important, for my benefit, to note here that C# is obviously not a new language and so there is a lot of base work being built upon here.

Anyway, with that said I would love to follow along as closely as I can and would like some more information about how to do that.

Thanks, -TuckerD

RxDave commented on Jan 28 2015

👍 Method contracts! Yes please. Also really liked the idea of syntax for tuples and records. Pattern matching - even if just within switch case statements - would be fantastic as well.

jonschoning commented on Jan 28 2015

If the 'pattern on the right hand side' allows matching on subclasses of the data on the left of this 'is' operator, then without having the capability of exhaustive checks, (as seems from what is described so far), imho the most important benefits of pattern matching are lost.

An 'exhaustive check' would mean that there would be a compiler warning if new subclasses were added to the data being matched on that didn't have cases for that new subclass in existing pattern matching code.

Actually, I'm thinking now it may be difficult to make exhaustive checks work for concrete classes because of ambiguity with the syntax allowing destructuring of the primary constructor.

Perhaps there could exhaustive checks only when matching on abstract classes if that causes less ambiguity.

Thinking about it some more, classes are not closed, so exhaustive checks would make no sense... This really is not what I would call pattern matching without discriminated unions. This should just be called "destructuring"

Plasma commented on Jan 28 2015

Thanks **@MadsTorgersen** - a few various thoughts:

1. Introducing a "type safe basic type". Let me distinguish between:
   int customerId = 1;
   int billingId = 1;

Can we "mark" this integer (or Guid, or long, etc) of customerId (pulled from a database for example) as a "CustomerId" type?

public CustomerId : int { }

I want to prevent a method that takes:

```
public void Charge(int customer d)
```

... accidentally being passed in a billingId (also an INT).

Ideally my interface was:

```
public void Charge(Customer d customer d)
```

... and passing in a billingId would result in a compilation error.

1. +1 to AOP integration (we use Castle Dynamic proxies right now). Really useful for authorization, caching, logging, ... cross cutting concerns. I think the developer experience is understandable when the debugger correctly shows the stack trace of calls being intercepted (which is what happens with Castle Windsor).
2. Ahead-of-time compilation of an entire ASP.NET website. I don't know if this is applicable to C# spec, but a way to "100% warm up, pre-load, be ready" a .NET website would be good. Right now even under Release mode, there is JIT, which I never want on a production site.
3. Let me use commas (or spaces for universal support?) in number definitions:
   const int BigNum = 1,000,000; // No need to count the zeros
4. I use Tuple<string, int> etc a fair bit; ideally there was a super nice way to return a tuple with named fields vs Item1/Item2 without making a new class

chrisaut commented on Jan 28 2015

@plasma, regarding your "type safe basic type", what's wrong with:

```
public struct Customer d {
     public int  d { get; }
     public Customer d(int id) { this. d = id; }
       optional
    public static implicit int(Customer d c d) => c d. d;
}
```

If you need many of those, you can just create an abstract base class:

```
public abstract class  d ase {
     public int  d { get; }
     public  d ase (int id) { this. d = id; }
       optional
    public static implicit int( d ase c d) => c d. d;
}
public class Customer d :  dbase {}
```

It's not a struct anymore, but it should hardly matter.
It's a bit of typing, but with Record types and primary constructors it'll get so short it's essentially nothing.

```
public struct Customer d(int id) {
     public int  d { get; } = id;
}
```

I believe the real feature you want is ValueType (pseudo?) - inheritance, a form a compiler enforced type aliasing.
I don't think adding yet another form of "type" to the language is a good idea, it's getting kinda complicated as it is.

ryancerium commented on Jan 28 2015

Apply the `readonly` modifier to classes, methods, and parameters to help support immutable data structures. Perhaps add the `mutable` keyword for `readonly` classes the way C++ uses it.

I've wanted static methods in an interface before, but generic constructor constraints might have been the underlying problem.

Could you implement constraints at compile time using a static analyzer? Like using an unassigned variable is a compiler warning (error?) now.

Discriminated unions like Rust has.

Interfaces with default implementations like Java just added (is thinking about adding?). Is this different than traits or mixins in large way?

What if you make the lambda capture list an extended portion of the parameter list?

```
var number =  etNumber();
var name =  etName();
var  uery = customers.Where((c, ref name, value number) => c.Name == name    c.Number == number);
```

RichiCoder1 commented on Jan 28 2015

@ryancerium
:+1: for `readonly class` with `mutable` members.

> What if you make the lambda capture list an extended portion of the parameter list?
>
> ```
> var number =  etNumber();
> var name =  etName();
> var  uery = customers.Where((c, ref name, value number) => c.Name == name    c.Number == number
> ```

I'd be against this if only because of the cognative load of differentiating between captures and params. While `[number, value name](c) => ...` isn't the prettiest, it's at least a lot more immediately clear.

Fizzelen commented on Jan 28 2015

How about adding private variables inside a property, to prevent side effects from modifying the the backing field directly

public int Id
{
int _id = 0;
get { return _id; }
set { _id = value; }
}

leppie commented on Jan 28 2015 Contributor

The pattern matching construct should be an expression and not a statement like switch. My 2 cents.

RichiCoder1 commented on Jan 28 2015

@Fizzelen wouldn't that just be a good old fashion property at this point? Or are you referring to doing something like:

```
public int  d
{
    int  id =  ;
    get { return  id * 2; }
    set {  id  = value   2; }
}
```

tophallen commented on Jan 28 2015

For Lambda Captures, perhaps something inline could be useful, such as:

```
var name =  etName();
var  uery = customers.Where(c => c.Name == [name]);
```

or

```
var name =  etName();
var  uery = customers.Where(c => c.Name == {name});
```

I would think something like this (perhaps not this exact syntax) could avoid the mess of having a capture list before the expression, and could allow it to capture sub properties from an object rather than the whole object as well, i.e

```
var person =  etPerson();
var  uery = customers.Where(c => c.Name == {person.Name});
```

So here, it would capture the value of `person.name` and not the whole person object.

This could also work fine with other capture options, such as by value, `{val name}` or `{val person.Name}` if this feature were desired, not sure I personally see the benefit of that either as we can create a variable to capture instead.

This might still get tricky with nested functions or nested class declarations, but if the capture was `{person.Name.To tring()}` - perhaps it would be aware and capture the property and call on it separately? or perhaps we would have to call it like so: `{person.Name}.To tring()` - and then calling `{person.Name.To tring()}` would actually capture the `To tring()` call.

RichiCoder1 commented on Jan 28 2015

@tophallen
Something like this maybe?:

```
var name =  etName();
var  uery = customers.Where(c => c.Name ==  val name );
```

Though I'm not sure what syntax would actually be happy point.

ashmind commented on Jan 28 2015

@Fizzelen I suggest just having an auto-field called `field` instead, e.g.

```
public int  d {
    get { return field; }
    set { field = value; }
}
```

tpetrina commented on Jan 28 2015

I would welcome any sort of "const"ness on method parameters. More powerful switch statement that can handle expression non-compile time constants would also help a lot.

jamesbascle commented on Jan 28 2015

@tophallen, **@RichiCoder1** IMO, the folks talking about lambda capture syntax seem to be missing the point by talking about in-lining this new syntax. We don't really get anything that way. The nice part about the way **@MadsTorgersen** originally typed it, is that it is "foward declared" before it is actually used, much like a method parameter or field or whatever, so to use it in the executing part of the lambda, you have to declare that you want to bring it into the closure before you actually use it.

All you do by inlining it into the operative part of the lambda is introduce some more complicated syntax that doesn't gain us anything. I'm not saying the original syntax is pretty, or should be used, but that is a very clear benefit of it.

s-aida commented on Jan 28 2015

As a less ambitious approach than non-nullable reference types, I think I'm okay with having a shorter approach to check that a variable neither refers to `null` nor has a value generally considered as blank (e.g. string is empty or array length is zero). `tring. sNull r mpty(foo)` is still a bit tiring even with stating using.

Maybe akin to ask if it's ever possible for `null` to call a method though.

tophallen commented on Jan 28 2015

@jamesbascle true, but the compiler could look at the expression for pattern matching, and forward declare on build, and that would allow it to not capture the whole scope onto the heap, and you could pick sub properties, or call a method to capture a specific part of a class, or to capture a state, so while the syntax might get a bit cluttered (I see what you mean though, as this would especially be annoying when you use the same captured variable multiple times (i.e. `c => c. tart ate <= [my ate]  c. nd ate >= [my ate]`), you could get the benefits of explicit capture. Forward declaring would make for less work however.

To your point perhaps the suggestion by **@ryancerium** could be tweaked to keep the readability and look something like this?

```
var name =  etName();
var  uery = customers.Where(([name], c) => c.Name == name);
```

or

```
var person =  etPerson();
var  uery = customers.Where(([person.Name],[val person. d], c) =>
                            c.Name == person.Name   c. d == person. d);
  or if we wanted captures to be one thing
var  uery = customers.Where(([person.Name, val person. d], c) =>
                            c.Name == person.Name   c. d == person. d);
```

In this way it keeps the expression cleaner, but also makes it clear that it is a captured value and not coming from the expression itself, but is to be captured in the heap for the expression to use. The issue that would be left with this method would be, should `[]` be required to ensure that you capture nothing? Or could it be specified some other way?

der-Daniel commented on Jan 28 2015

```
bservableCollection. dd ange()
```

jamesbascle commented on Jan 28 2015

@tophallen I could be wrong, but I'm pretty sure that the compiler-generated lambda helper classes only bring "this" (whatever class it is in that context) and any referenced variables, so I think it already doesn't pull in the entire calling scope, just those it needs.

mariusGundersen commented on Jan 28 2015

I'm glad you are thinking about virtual extension methods, it would be great to have dynamic binding outside of methods. For example something like

```
public string  o omething(virtual   omething s){
  return "this is the base method";
}
public string  o omething( omething mplementation s){
  return "s is  omething mplementation";
}
public string  o omething( omething lse s){
  return "s is  omething lse";
}
  ...
  omething s;
s = new  omething mplementation();
 o omething(s)  returns "s is  omething mplementation";
s = new  omething lse();
 o omething(s)  returns "s is  omething lse";
```

This is possible to do with dynamic, but dynamic isn't restrictive enough, as it will match any type. With virtual it would be possible to match only on subclasses/implementations of a type. In the above case the method marked as virtual would be the fallback in the case that a new type is introduced which doesn't have an implementation of DoSomething

mariusGundersen commented on Jan 28 2015

C# really need a stricter version of enum, one that cannot be cast from int so the compiler can do static analysis on it, for example in switch/case:

```
enum Maybe{ ep, Nope};
  ...
public bool  s ep(Maybe maybe){
  switch(maybe){
    case  ep: return true;
    case Nope: return false;
  }
}
```

The above code will not compile, since the compiler is unable to tell that all possible paths through the IsYep method has a return value. The way to fix this is to add a default case, which is really not needed. A stricter implementation of enum would have the above case compiling without error but not compiling in the following case (because not all the enum cases have been tested in the switch/case):

```
enum Color{ ed,  reen,  lue};
  ...
public bool  s ed(Color color){
  switch(color){
    case  ed: return true;
    case  reen: return false;
  }
}
```

tophallen commented on Jan 28 2015

@jamesbascle no, you are right, it was a mistake on my part - it does only capture what it think it needs.

But if say, you want to check if `c => person.Name == c.Name` - it is capturing the class containing the property `Name` (the person object) because it's really a method of the object (`get Name()`) - if we could explicitly get the backing value being the value that is referenced in the expression (i.e captured explicitly via `[person.Name]`), that's where we would see the reduced footprint. That might not be possible or ideal though. However, if we can just gain better control the lifetime and explicitly capture objects or not, that could help quite a bit. I think the biggest piece we would gain from it is explicitly preventing capture on a self-contained expression via `[]` .

It's a good feature suggestion, and there are quite a few different ways one could go about implementing it. I'm certainly not going to say mine is the best, and based on how the C# language has evolved since I've started learning and even before that, I can say fairly confidently that I trust the design team's decision on how to go about it, if it's something that is ready to be implemented in the next version at all.

jamesbascle commented on Jan 28 2015

@tophallen I think we can both agree on basically all of that. 😄

colombod commented on Jan 28 2015

You talking about things like const functions?

Dr Diego Colombo PhD

> On 28 Jan 2015, at 07:17, Toni Petrina notifications@github.com wrote:
>
> I would welcome any sort of "const"ness on method parameters. More powerful switch statement that can handle expression non-compile time constants would also help a lot.
>
> —
> Reply to this email directly or view it on GitHub.

jamesbascle commented on Jan 28 2015

@colombod I think he's talking about making mutable objects passed to functions or methods unable to be mutated within method scope. I personally think immutable type support would largely scratch that itch, but there are definitely some other cases.

It would likely have to come with some kind of performance hit, unless the logic for statically analyzing whether any of the methods called or properties accessed within the scope mutate the object in any way can be worked out at compile time.

tanghel commented on Jan 28 2015

@MadsTorgersen first of all, I'm the biggest fan of **val/let/readonly** being a marker for immutability!

We already had a discussion at the Xamarin Conference about non-nullable reference types. What do you say about the following syntax:

Declaration:

```
public void MyMethod(string  value1, ob ect  value2)
{
   ...
}
```

I can think about two approaches:

- the "!" sign means there is an ArgumentNull check by the runtime and will be used only as method parameter declaration
- the "!" sign means an addition to the type system, just like "?" is for nullable value types. For the second approach, these could be possible usages:

Guaranteed success:

```
string  string alue = "Hello world ";
ob ect  ob ect alue = new ob ect();

 nstance.MyMethod(string alue, ob ect alue);
```

Usage with null check:

```
if (string alue  = null    ob ect alue  = null)
{
     nstance.MyMethod((string )string alue, (ob ect )ob ect alue);
}
```

Usage without null check:

```
   compiler check will make sure you don t assign null
ob ect  nonNullable b ect = ob ect alue ?? new ob ect();
ob ect  nonNullable tring = string alue ?? string. mpty;

 nstance.MyMethod(nonNullable tring, nonNullable b ect);
```

By putting the argument type in the signature of the method, you enforce the programmer to transform his variables into something that cannot explicitly receive the null value. The runtime should check assignments to these data types and will disallow them.

Non-nullable method parameters will be checked by the runtime as well and could result in **ArgumentNullException** (or maybe **NullAssignmentException**) that will always occur if you assign a null value to a non-nullable reference type.

tophallen commented on Jan 28 2015

@jamesbascle Maybe then rather than bother implementing explicit capture syntax, we just implement explicit no capture syntax, like a keyword on the expression, i.e

```
var  uery = customers.Where(const c => c.Name  = null);
  or other keywords,  val  could be one or something else
var  uery = customers.Where(nocapture c => c.Name  = " omeName");
```

more similar to the async syntax for expressions, and maybe then there could be more freedom in how to implement the explicitly captured variables in an expression later, or at least, the more I think about various scenarios for it, the less happy I am with both my suggestions and the `[name](c) => c.Name == name` syntax proposed by **@MadsTorgersen** as no option really ***feels*** like the C# option.

ErikSchierboom commented on Jan 28 2015

I really like the idea of having readonly parameters and locals, the less mutability the better. Too bad that we can't make them readonly by default, as that would of course break lots of code.

Although using `val` or `let` for readonly parameters or locals would result in less "noise" than `readonly`, I would still go with the `readonly` keyword. The `readonly` approach has the advantage that it is both more explicit *and* ties into a known keyword that has the same meaning. Perhaps there could be a shortcut syntax for the `readonly` keyword, similar to how `Nullable<T>` has a `T?` shortcut.

Some syntax comparisons:

```
public void Test1(readonly string str1, readonly string str2, readonly string str3)
{
    readonly str local;
}
public void Test1(val string str1, val string str2, val string str3)
{
    val str local;
}
public void Test1(let string str1, let string str2, let string str3)
{
    let str local;
}
public void Test1(string  str1, string  str2, string  str3)
{
    str  local;
}
```

So personally I think the `readonly` keyword should be used, with some shortcut syntactic-sugar to make it easy to use.

This was referenced on Jan 28 2015

Polyglot Support reactive-streams/reactive-streams-jvm#45 Closed

Allow multiple return values in C# #102 Closed

george-polevoy commented on Jan 28 2015

Enum Types

I like the idea of typed enums, like in Java Enum Types.

This would allow for natural visitor pattern implemented directly in enum declaration instead of conditional switch, true singletons, etc.

wesnerm commented on Jan 28 2015

Proper Tail Calls
I would appreciate proper tail calls. I know that the 64-bit jitter does some of that.

Loops is a workaround for some cases, but the code potentially looks uglier.

Loops is not a workaround the lack of tail calls for mutually recursive functions.
Or, when programming a continuation-style manner, to achieve elegant backtracking and other techniques.

Tail calls is actually very helpful when calling many higher order functions.

At the least, you can support a [TailCallAttribute] attribute.

AdamSpeight2008 commented on Jan 28 2015 Contributor

Traits would be good to be have if it they could be defined like classes.

```
trait  ero        { public static T  ero() }

trait  nit        { public static T  nit() }

trait op addition { public static T [ ] (T,T) }

trait op subtract { public static T [ ] (T,T) }

trait  impleCalc
  inherits {  ero,  nit, op addition, op subtract }

trait op   ualTo    { public static T [==] (T,T) }

trait op Not  ualTo { public static T [ =] (T,T) }
```

Traits the would be an extended form of **constraited generics**
Then for example a `ummation` extension method could be generic.

```
T  um<T> ( this nums :   numerable<T> )
  where T has trait  impleCalc
{
  T total = T. ero;
  foreach( num in nums )
    total = total   num;
  return T;
}
```

Enforced at compiler and runtime (the run cost could be worth it).
It would be also applicable to **constructors**

```
has trait { new ( nt, tring) }    a constructor that takes an int and a string,
```

AdamSpeight2008 commented on Jan 28 2015 Contributor

IDEA: IsAnyOf and IsNoneOf for TypeOf expression.

leppie commented on Jan 28 2015 Contributor

> Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.

First line in the Scheme language specification

AdamSpeight2008 commented on Jan 28 2015 Contributor

Keep `switch` construct as it and have a `match` construct for pattern matching.

JauernigIT commented on Jan 28 2015

+1 for Code Contracts at language level. I've ever been a fan of Code Contracts since .NET 4 and the DbC principle in general. I wrote about it in many articles.

Unfortunately, Code Contracts in their current implementation didn't spread very much in real-life projects. Imho this has several main reasons:

1. Insufficient language support.
2. Insufficient built-in tooling support in Visual Studio.

3. No StyleCop/FxCop support (afaik).
4. Limited static checking.

Code Contracts built into the language would be awesome, because some constructs have been really cumbersome (contracts on interfaces as an example). But furthermore, integration with tooling (VS, StyleCop, Pex, ...) is key to give real value. And the definition of useful contracts on all .NET core methods, because that's the requirement to let static contract checks work. And in my opinion static checking is the essential part to let code contracts succeed, including features like pure checking etc..

andrewducker commented on Jan 28 2015

Enum generic constraints would be marvellous.

"Duck typing" generic constraints would also be very useful.

asd-and-Rizzo commented on Jan 28 2015

Contracts in some way of some sort (mostly boilerplate, not static analysis).

C# 5:

```
public void  emove(string item)
{
   if (item == null) throw new  rgumentNull xception("item"));
}
```

C# 6:

```
public void  emove(string item)
{
   if (item == null) throw new  rgumentNull xception(nameof(item));
}
```

What if possible to drop `xception` after `throw` keyword(like `ttribute` is dropped while applied):

```
public void  emove(string item)
{
   if (item == null) throw new  rgumentNull(nameof(item));
}
```

What if possible to drop `new` like in some languages:

```
public void  emove(string item)
{
   if (item == null) throw  rgumentNull(nameof(item));
}
```

What if possible to use kind of `?` or `??`:

```
public void  emove(string item)
{
   item ?? throw  rgumentNull(nameof(item));
      item  == null ? throw  rgumentNull(nameof(item));
}
```

What if possible to get `throw rgumentNull` via import of static function into namespace:

```
namespace  ystem
{
public static class  xceptions
  {
    public static  rgumentNull xception  rgumentNull(string name)
    {
       return new  rgumentNull xception (name);
    }
  }
}
```

What if like `try catch finally`:

```
public void  emove(string item)
{
re uires    current Contract. ndContract lock
{
  item ?? throw  rgumentNull(nameof(item));
}
contracted
{
  body
   ...
}
ensure
{
   compiled into some form into body to prevent assignment less then  ero
 Count <   ? throw  rgument(nameof(count)," hould be greater then  ");
}
}
```

m0sa commented on Jan 28 2015

RE: Metaprogramming. Have you considered something like the ASP.NET 5 XRE's ICompileModule interface?

yetanotherchris commented on Jan 28 2015

What happened to duck typing of interfaces? I realise it's not C# specific, but I'm sure it was part of 4.6, and it would be incredibly useful for shared libraries (such as logging; an ILog interface)

m0sa commented on Jan 28 2015

@yetanotherchris assembly neutral interfaces are also an XRE-only thing

JauernigIT commented on Jan 28 2015

@andrewducker +1 for duck typing of generic constraints.

kestasjk commented on Jan 28 2015

All I have to say is: Yes please.

djelovic commented on Jan 28 2015

@MadsTorgersen Array slices are an awesome idea (the current ArraySegment is a good start), but string slices would be even more useful. If you profile allocations of a typical .NET program most of of the allocations seem to be tiny strings. So something like struct StringSlice would be awesome.

(I created one for my projects but it breaks as soon as you call an existing API that expects strings, such as double.TryParse(). So it's one of those all-or-nothing kind of things.)

Also, are you guys doing any pointer escape analysis? Implicit boxing of small objects might not be so painful if the allocation was done on the stack.

nanddonet commented on Jan 28 2015

It would be nice if you could await in getter and setter acessors of the properties using the async-await pattern. In the next release you can await inside the catch blocks, making it happen in getters and setters would be awesome!

AdamSpeight2008 commented on Jan 28 2015 Contributor

@MadsTorgersen For slicing an new operator `..` could used, to discriminate from 2D array access.

```
a[ 2 .. 5 ]     a[2] to a[3]
```

It be also applicable as a defining an range. `1 .. 1`

AdamSpeight2008 referenced this issue on Jan 28 2015

Constrained primitive and string types #104 Closed

daveaglick commented on Jan 28 2015 Contributor

@tanghel +1 for using '!' to denote non-nullable reference types similar to '?' for nullable value types. It's clean, concise, and matches up nicely with existing syntax. Of course the semantics of non-nullable reference types is another matter, but I like the syntax.

davepermen commented on Jan 28 2015

@ErikSchierboom val would be like 'readonly var', so not for explicit types. there, readonly could stay.

a.k.a.
string example = "";
readonly string example = "";

but
var example = some.Funky().Query().FromLinq();
val example = some.Funky().Query().FromLinq();

i personally never use explicit types for variables.. only var everywhere. i'd love to instead use val everywhere.

davepermen commented on Jan 28 2015

I'd like to see some nice syntax for IEnumerable to make it the default choice for people in more cases. my preferred syntax would be T*, but given the unsafe {} c++ style pointers there, that won't work.

ErikSchierboom commented on Jan 28 2015

@davepermen Ah, then it makes a lot more sense! `val` is thus an alternative for `var`. I would then prefer `let` over `val`, as the difference between `val` and `var` can be quite hard to spot.

davepermen commented on Jan 28 2015

that's true. but var is short for variable, val short for value. let is .. well.. nothing.. :)

but yes, readability might be an issue.

mariusGundersen commented on Jan 28 2015

Something that is missing is syntax for accessing the field or property of a class. This is often done in linq chains, and is usually done using an expression:

```
  here x => x.name is an expression and reflection is used to find out which property is accessed
my atabaseTable.select(x => x.Name);
  it could look like this
my atabaseTable.select( atabaseTableClass::Name);
```

`ClassName:: ieldName` would be syntaxt sugar for `typeof(ClassName)). et ield(" ieldName")`.

`select()` could be defined as

```
public static T esult select<T nput, T esult>(this T nput self,  ield nfo<T esult> field){
  return field. et alue(self);
}
```

Note that I am assuming FieldInfo is generic, which it isn't today, but should be (just like Type should be generic, like Class is in Java)

drewnoakes commented on Jan 28 2015 Contributor

Regarding lambda capture lists:

> In a way the most useful annotation would be the empty `[]`, making sure that the lambda is never accidentally modified to capture *anything*.

A simple, targeted solution to this is to have an alternative lambda 'arrow' syntax for non-capturing functions.

```
users. elect(user  > user. ctive);    " >" re uires non capture
```

Developers and IDEs can migrate existing code where possible, after which the guarantee is enforced.

It's not clear to me how copying a reference type would be enforced if a capture list allowed specifying copying. Would copy constructors have to be added to the language as well?

Giftednewt commented on Jan 28 2015 Contributor

`let` reminds me of the days when I was writing code in VB6 (fond memories), but I think I prefer it to `val` as it's significantly different enough from `var` that you could easily see whilst scanning code that the variable in question is readonly. I also think that `let` doesn't convey well enough that the variable is readonly.

The only alternatives I've thought of are `rvar` (readonly variable) or `rloc` (readonly local) or `rov` (readonly variable). Though I wouldn't consider any of those to be ideal, I think they convey the point of a readonly variable better than `let`.

ErikSchierboom commented on Jan 28 2015

Perhaps `val` would also be fine, it certainly links it to the existing `var` keyword so it has that going for it. And now that I think of it, I did not have any problem with `val` and `val` doing the same thing that is proposed in Scala.

joshuaellinger commented on Jan 28 2015

We've done one project in GoLang now and, despite some crippling design flaws, there is one area that it shines -- concurrency, ie. channels/goroutines. They have the same liberating effect on parallel processing that the property/method/events model has on component development. In other words, the syntactic sugars buys you a lot.

C# Task parallel library works as a weak substitute. You have to resort to block processing to get 1/2 the performance that GoLang gives you on scalar channels naturally. And it only works on a specific subset of problems.

We're done with GoLang due to its other flaws but we'd love to see C# copy channels/goroutines.

ps - If you think existing lambas and async/await get you the same effect, you need to use GoLang for a project to see the difference. It's not even close.

ufcpp commented on Jan 28 2015

I sometimes see Japanese programmers are confused on distinguishing val and var in Scala, even though they are professional IT developers for years. It is difficult for Japanese to make out deference between L and R.

svick commented on Jan 28 2015 Contributor

@joshuaellinger Could you explain what exactly makes channels so much better than `async` - `await`? Is is the support for asynchronous sequences?

tanghel commented on Jan 28 2015

@MadsTorgersen what about try/catch/finally aggregation?

Instead of:

```
int result;
try
{
    result =  et alue();
}
catch ( xception ex)
{
    Handle xception(ex);
} finally
{
     ispose esources();
}
```

... you could write:

```
var result = try  et alue catch Handle xception finally  ispose esources;
```

And the elongated version could look like:

```
var result = try () =>  et alue()
catch (Win32 xception ex) if (ex.Native rrorCode ==  x   42) => HandleNative xception(ex)
catch ( xception ex) => Handle xception(ex)
finally () =>  ispose esources();
```

The trick would be to enable return values on a try block, as well as support lambda expression on try/catch/finally. As long as it's a one-liner, the resulting code looks much cleaner.

soc commented on Jan 28 2015

@tanghel Yes, Scala has done this for years without issues. It's kind of weird in C# though, because `try` (and as proposed above, `switch`) would turn into an expression, but `if` would still be a statement. They either need to go the whole way or drop this proposal, imho.

igor-tkachev commented on Jan 28 2015

@Giftednewt: What about `def` instead of `let`?

tanghel commented on Jan 28 2015

@soc yes you are right, but you can use the conditional operator (**?:**) as an expression and assign it to a result variable. Changing the **if** clause to expression would most probably mean to include a **then** keyword as well. We'll see

soc commented on Jan 28 2015

@tanghel

> but you can use the conditional operator (?:) as an expression

The point is achieving some kind of consistency, not the lack of work-arounds.

> Changing the if clause to expression would most probably mean to include a then keyword as well.

I don't see how this would follow from making `if` an expression.

pauldipietro commented on Jan 28 2015

@igor-tkachev I think that as `def` is what's used in languages like Ruby and Python to define a method, it could cause some confusion. I personally would stick to `val`, or maybe something like `imm` or `imut` if you wanted to be very explicit in its purpose?

tanghel commented on Jan 28 2015

@soc you have to mark somehow that the if clause has ended and that you're expecting a result expression, that's what I meant. Or maybe I don't see a more elegant way of doing it

conditional operator

```
var result = value == 1  ? true : false;
```

hypothetical if expression

```
var result = if value == 1  then true else false;
```

fubar-coder commented on Jan 28 2015

@MadsTorgersen

> **@Romoku**: I'd love to get non-nullable reference types into the CLR. It seems a tall order even there: what's inside an array of non-nullable reference type when it's first created? etc.

Why not doing something like the D array initialization?

This would create an array with all items set to an empty string.

```
string [1  ]  a = string. mpty;
```

85 items not shown
View more

MadsTorgersen commented on Jan 29 2015 Contributor

Thanks for all the comments everyone! I'll be closing out the issue now, as the discussion is off in many directions and probably hard to productively follow for most. Many of the features discussed in the design notes are now up as individual issues, and I encourage continuing the discussion there.

It is astoundingly great to see such excitement around the next version of C# - even at this very beginning of its design cycle. The creativity and insights coming out here on GitHub are just amazing, and I'm very happy we move here. It's going to be a wonderful time going ahead! Thanks again!

Mads

MadsTorgersen closed this on Jan 29 2015

This was referenced on Jan 29 2015

- **Make use of Code Contracts where ever applicable.** dotnet/corefx#503 — Closed
- **Pattern matching based on generic type** #178 — Closed
- **C# Design Notes for Jan 28, 2015** #180 — Closed

adlangx commented on Jan 30 2015

How about an auto property with a lazy backer. That would be nice.

alisabzevari commented on Jan 31 2015

I think it would be good to have something I call it Inline type declaration:

For example think of an action in web api. You want to define a Data Access Object but defining a class and use it once (in an action) is somehow expensive:

```
public class  ample ao
{
  public int a {get; set;}
  public string b {get; set; }
}
public class  ampleController :  piController
{
  public  ample ao  et ample()
  {
     ...
```

```
  }
}
```

It can be more simpler by inline type declaration:

```
public class  ampleController :  piController
{
  public {int a, int b}  et ample()
  {
     ...
  }
}
```

jods4 commented on Feb 1 2015

@alisabzevari What I usually do if I have a web api that returns a projection (or DAO) that nothing else in C# uses, is that I declare it `ob ect` and return an anonymous type.

```
public ob ect  et ample()
{
  return new { a = 4, b = 2 };
}
```

I would agree this is not the best self-documenting code, but it works and avoids boilerplate code.

alisabzevari commented on Feb 1 2015

@jods4 I use this technique too. But the problem is we can't write strongly typed unit tests for these apis.

tophallen commented on Feb 1 2015

You don't have to, as you know that it is going to be serialized over the wire, test it as an XML object or a json object using the serializer your API uses, and validate it that way. Or test it as a dynamic object, by pulling in the Microsoft.CSharp reference.

From: Ali Sabzevarimailto:notifications@github.com
Sent: 1/31/2015 21:57
To: dotnet/roslynmailto:roslyn@noreply.github.com
Cc: tophallenmailto:tophallen@outlook.com
Subject: Re: [roslyn] C# Design Notes for Jan 21, 2015 (#98)

@jods4 I use this technique too. But the problem is we can't write strongly typed unit tests for these apis.

Reply to this email directly or view it on GitHub:
#98 (comment)

alisabzevari commented on Feb 1 2015

I try to consider web apis as normal classes and try to write them the way they are normal classes; Not classes that are designed specially to work with http. I think they could be used as logic layer classes in different contexts. That's why controllers in MVC6 are not required to inherit from Controller class.

On Sun Feb 01 2015 at 10:24:52 AM tophallen notifications@github.com wrote:

> You don't have to, as you know that it is going to be serialized over the wire, test it as an XML object or a json object using the serializer your API uses, and validate it that way. Or test it as a dynamic object, by pulling in the Microsoft.CSharp reference.
>
> From: Ali Sabzevarimailto:notifications@github.com
> Sent: 1/31/2015 21:57

To: dotnet/roslynmailto:roslyn@noreply.github.com
Cc: tophallenmailto:tophallen@outlook.com
Subject: Re: [roslyn] C# Design Notes for Jan 21, 2015 (#98)

@jods4 I use this technique too. But the problem is we can't write strongly typed unit tests for these apis.

Reply to this email directly or view it on GitHub:
#98 (comment)

—
Reply to this email directly or view it on GitHub
#98 (comment).

asbjornu commented on Feb 2 2015

@MadsTorgersen writes:

> ... there's a long-standing request for non-nullable reference types, where the type system helps you ensure that a value can't be null, and therefore is safe to access.

Yes! Please have a look at Ceylon and how it handles null. Simply put: All types are non-null by default. If you want `null`, you explicitly declare it with the `?` operator. Once you've done an `exist` check against that variable, it's guaranteed not to be `null`.

Now, I know this won't fit with C# and its "null by default" scheme, but being able to assert and from then on guarantee that a given variable is not null would be extremely powerful.

AdamSpeight2008 commented on Feb 2 2015 Contributor

@asbjornu I think that non-null by default would very likely break existing code.

asbjornu commented on Feb 2 2015

@AdamSpeight2008 That's why I wrote the following:

> Now, I know this won't fit with C# and its "null by default" scheme, but being able to assert and from then on guarantee that a given variable is not null would be extremely powerful.

markrendle commented on Feb 2 2015

Any predictions on whether we're likely to see a new CLR for C#7/VB14? .NET 5.0 at last?

jvlppm commented on Feb 2 2015

C# could accept an exclamation point after the type definition, indicating that the reference type is not nullable.
And this could be just a compile time check.

How would it work:

```
public static void Test(ob ect  ob ) {
    ...    code
}
```

Would compile to:

```
public static void Test(ob ect ob ) {
    if (ob  == null)
        throw new  rgumentNull xception(nameof(ob ));
    ...    code
}


ob ect  a;     uild error
ob ect  inline = new ob ect();
ob ect can eNull = new ob ect();
```

```
ob ect  would eNull = can eNull;     uild error
ob ect  cannot eNull = can eNull ;
```

The last line would be compiled to something like:

```
if (can eNull == null)
    throw new Null eference xception(nameof(can eNull));
ob ect cannot eNull = can eNull;
```

1

sharwell commented on Feb 2 2015 Member

@jvlppm One item from your examples would only be an error if it's a field.

```
class T {
  ob ect  a;    build error

  void M() {
    ob ect  x;    no build error
  }
}
```

Also note that a non-nullable type could not be used for the type of a field in a `struct`.

jvlppm commented on Feb 2 2015

@sharwell why it would not be an error? In my sample I was expecting that to be an error.
Not Nullable references must initialized.
And it should integrate well with existing code / assemblies.

galenus commented on Feb 2 2015

@jvlppm **@sharwell** actually, it shouldn't be an error, because in many situations initialization cannot be made fit into a single line. You wouldn't limit usage of non-nullable types to one-line initializables, right?

jvlppm commented on Feb 2 2015

Yes I would, because if initialization was done later, the reference would have no value until that point.

And you can always do this, in case initialization is not short:

```
ob ect  x =  et alue();

ob ect   et alue() {
    ob ect a = null;
    ...     ome initiali ation code
    return a ;
}
```

Also, inside methods, you could convert to non nullable in a single line.

```
ob ect ob ect nderConstruction = null;
...     ome initiali ation code
ob ect  notNull = ob ect nderConstruction ;
```

galenus commented on Feb 2 2015

@jvlppm your field's value may depend on constructor arguments, in which case the `et alue()` method (which should be static, BTW) will not help much.

jvlppm commented on Feb 2 2015

OK, I see.

First, don't be harsh on newcomers, the above sample have the declaration and method together for reading purposes, they are not necessarily on the same scope. The static keyword was omitted on purpose, because it is not a requirement for this feature. This should not be under discussion, since it only distracts the reader, and shifts the focus away from my humble proposal.

So ignoring primary constructors, field initialization could be delayed to constructor, requiring that they are initialized before usage / method calls.

```
class Test {
    ob ect  notNull;
    Test() {
        ob ect x = notNull;     rror, usage of notNull before initiali ation
        MethodCall();     rror, non nullable references must be initiali ed before method calls.
         taticTest(this);     rror, non nullable references must be initiali ed before this.
        notNull = Create b ect();          e uired initiali ation
        MethodCall();      , all non nullable references already initiali ed
    }
    static ob ect  Create b ect() {
        ob ect a;
        ...     nitiali ation code
        return a ;
    }
    void MethodCall() { }
    static void  taticTest(Test a) { }
}
```

I guess this way it should work, and not be so limited on initialization.

galenus commented on Feb 2 2015

@jvlppm First of all, sorry if my remarks sound harsh - that was definitely not my intention.
As of initialization problem - I believe some kind of user extendable mechanism for `default` operator overloading could help with this matter.
Let's say, you would only be allowed to use the identifier with types having the `default` operator overload or static method like `public static operator La y<MyType> default() ...` accessible in the scope of your code.

Unknown6656 commented on Feb 2 2015

I would really like something like a "dot assignment" operator, which I think should look something like this:

instead of using
```
string s = "Hello World ";
s = s.ToLower();
```
I would like to use
```
s .= ToLower() .
```
I know, this isn't much, but i really like to keep my code short in some lines and this would really help.

And an other operator I would really like to see is the following one:
Instead of using `= Math.Pow(x, y)` I would hope for something like `= x ** y .`
I know that the star-character is used for pointers, so I have also this alternative to propose: `= x  y`
(I have noticed, that there is no "Backslash-operator")

Excuse me for my bad English - I am working on it :)
Thank you very much,
Unknown6656

ggrnd0 commented on Feb 3 2015

What about javalike interface initialization? something like:

```
interface  HasPropertyName{
    string Name{get;}
}
new  HasPropertyName{Name = " lice"};
```

And additionaly

```
interface  HasPropertyName{
    string Name{get;}
}
interface  HasProperty ge{
```

```
    long  ge{get;}
}
new  HasPropertyName, HasProperty ge{Name = " lice",  ge=21};
```

So, it will

```
public var Create lice(){
    new  HasPropertyName, HasProperty ge{Name = " lice",  ge=21};
}
```

And whern we see CodeHint where will be method signature

```
public  liceType Create lice< liceType>()
    where  liceType: HasPropertyName, HasProperty ge
```

and another feature is using TypeConstraint with only interface and properties to use TypeConstraint like a type

```
public  liceType Create lice< liceType>()
    where  liceType: HasPropertyName, HasProperty ge
{
    new  liceType{Name = " lice",  ge=21};
}
```

ggrnd0 commented on Feb 3 2015

await/async...
they are too ugly...
why we need to use async and Task<?> type?
why we cannot to use await transparently?

see

```
calss  {string val;}
class  {  a;}
class C{  b;}

  load (){
    return await load rom low ource();
}
  get (){
    var b = new  {a=load ()};
      some code in get
   return b;
}
C getC(){
    var c = new C{b=get ()};
      some code in getC
    return c;
}
void main(){
    var = getC();
      some code in main
    Console.WriteLine(c.b.a.val);
}
```

When we call `await load rom low ource();` we create async task what works asynchronously untill we do not access any A instance fields.

And if loadFromSlowSource work too long `some code in get` , `some code in getC` and `some code in main` will execute parallely with `load rom low ource`

And only on `Console.WriteLine(c.b.a.val)` we will wait `load rom low ource` be done.

Is it possible in CLR/KRE?

svick commented on Feb 3 2015 Contributor

@ggrnd0

1. Code shouldn't execute in parallel automatically, unless you're absolutely sure you're not introducing race conditions. And I don't see how could the compiler be sure about that here.

2. You don't want to automatically synchronously wait on `async` code, because it negates any advantages of introducing asynchrony and because it very commonly leads to deadlocks.

ggrnd0 commented on Feb 3 2015

@svick,

1. auto add AsyncAttribute for method in compile time if it asynchronous or call any asynchronous method. Do not know what to do with condilional async method call...
2. everything leads to deadlocks, just current way too. There is error in async design, is not it?

> There are two best practices (both covered in my intro post) that avoid this situation:
> 1. In your "library" async methods, use ConfigureAwait(false) wherever possible.
> 2. Don't block on Tasks; use async all the way down.

1. it's too hard. i think there is simpler way exists.
   ThreadStatic vs DeadLocks? I do not think what use ThreadStatic with async is good idea ever.

2. i prefer it. and it will be done automatically by compiler and clr.

The another problem what i see is how to stop async?
How clr must determine when async must be stoped? In mvc any async results if they used will acessed befire ActionResult.Execute end calling.
But what about void and unused results? How wait their?

Yes this feature is too hard and complex. But i think this problems can be resolved.

And more, i think clr can know better when async needed or not and can do it independently.

I do not want to write `async Task` and `await` every there. I am too lazy =)

I found another way
`await` works like now, wait for executing async or not method.
but `async Method()` call method asynchronously. And await or property access breaks asynchronous.

In critical places when u want to exsure method executed you use `await` and do not otherwise.
if you want to start async executiong just write `async` .
This is simple? i think...

And new code example:

```
calss  {string val;}
class  {  a;}
class C{  b;}

  load (){
    return async load rom low ource();
}
  get (){
    var b = new  {a=load ()};
      some code in get
   return b;
}
C getC(){
    var c = new C{b=get ()};
      some code in getC
    return c;
}
void main(){
    var = await getC();
      some code in main
    Console.WriteLine(c.b.a.val);
}
```

When we call `async load rom low ource();` we create async task what works asynchronously untill we do not access any A instance fields or use `await` on any top level method *calling ended*.

And if loadFromSlowSource work too long `some code in get` and `some code in getC` will execute parallely with `load rom low ource`

And only on `await getC()` (or without await on `Console.WriteLine(c.b.a.val)`) we will wait `load rom low ource` be done.

gafter commented on Feb 3 2015 Member

@Ryancerium capture lists do not have anything to do with the parameters, so placing keywords in the parameters is not logical.

GPSnoopy commented on Feb 3 2015

One thing that would be really useful when writing high performance C# are generic pointers. For the moment we have to duplicate by hand (or via T4) the implementation for each type.

```
class Native rray<T> where T : unsafe struct
{
    private readonly T*  ptr;

     * ... *

    public T this[long index]
    {
        get { return *( ptr   index); }
        set { *( ptr   index) = value; }
    }
}
```

mburbea commented on Feb 3 2015

Yeah blittable would be a nice constraint but its not reasonable without some CLR reworking.

Miista commented on Feb 3 2015

All so that you don't have to write `async` and `await` ?
I would rather be explicit about when the code is going to execute in parallel.

Søren Palmund

> Den 03/02/2015 kl. 17.41 skrev ggrnd0 notifications@github.com:
>
> **@svick**,
>
> 1. auto add AsyncAttribute for method in compile time if it call
> 2. everything leads to deadlocks, just current way too. There is error in async design, is not it?
>
> There are two best practices (both covered in my intro post) that avoid this situation:
>
> 1. In your “library” async methods, use ConfigureAwait(false) wherever possible.
> 2. Don’t block on Tasks; use async all the way down.
>
> 1. it's too hard. i think there is simpler way exists.
>   ThreadStatic vs DeadLocks? I do not think what use ThreadStatic with async is good idea ever.
> 2. i prefer it. and it will be done automatically by compiler and clr.
>
> The another problem what i see is how to stop async?
> How clr must determine when async must be stoped? In mvc any async results if they used will acessed befire ActionResult.Execute end calling.
> But what about void and unused results? How wait their?
>
> Yes this feature is too hard and complex. But i think this problems can be resolved.
>
> And more, i think clr can know better when async needed or not and can do it independently.
>
> I do not want to write async Task and await every there. I am too lazy =)
>
> I found another way
> await works like now, wait for executing async or not method.
> but async Method() call method asynchronously. And await or property access breaks asynchronous.
>
> In critical places when u want to exsure method executed you use await and do not otherwise.
> if you want to start async executiong just write async.
> This is simple? i think...
>
> And new code example:

calss A{string val;}
class B{A a;}
class C{B b;}

A loadA(){
return async loadFromSlowSource();
}
B getB(){
var b = new B{a=loadA()};
//some code in getB
return b;
}
C getC(){
var c = new C{b=getB()};
//some code in getC
return c;
}
void main(){
var = await getC();
//some code in main
Console.WriteLine(c.b.a.val);
}
When we call async loadFromSlowSource(); we create async task what works asynchronously untill we do not access any A instance fields or use await on any top level method calling ended.

And if loadFromSlowSource work too long some code in getB and some code in getC will execute parallely with loadFromSlowSource

And only on await getC() (or without await onConsole.WriteLine(c.b.a.val)) we will wait loadFromSlowSource be done.

—
Reply to this email directly or view it on GitHub.

ggrnd0 commented on Feb 3 2015

@Miista, yes. I do not want to write async/await/Task everywhere.
First, you must migrate legacy code for asynchronous.
And second, you must allways write async/await/Task.
It is tediously....

If there any way to do it implicit it will. But if noone i am sad =)

Miista commented on Feb 3 2015

But if you do it your way it isn't clear if the code will run asynchronously or not.

Søren Palmund

> Den 03/02/2015 kl. 18.55 skrev ggrnd0 notifications@github.com:
>
> **@Miista**, yes. I do not want to write async/await/Task anywhere.
> First, you must migrate legacy code for asynchronous and you must allways write async/await/Task.
> It is tediously....
>
> If there any way to do it implicit it will. But if noone i am sad =)
>
> —
> Reply to this email directly or view it on GitHub.

ggrnd0 commented on Feb 3 2015

@Miista, yes...
But i think there are more chanses when it independent then not.
If you want ensure async just write `async`.
If you want ensure value is materialized just write `await`.

Write what you want to get or nothing if it independent.
So code will be asynchronous if it is inside used libraries and is not otherwise.

I do not want to know is implementation of IRepository{obect Get(id)} asynchronous or not.
I just want object as fast as it possible.

I do not want to rewrite hundreds of methods.
I do not know is it possible to do asynchronous implementation of method.

Of course there is crutch `Task. rom esult()`, but it ugly too. it is redundant code....

ggrnd0 commented on Feb 3 2015

And, for easement, `async` and `await` blocks for multiple method calls.
Like:

```
void method(){
    async{
        asyncTask1();
        asyncTask2();
        asyncTask3();
          asyncTask1, asyncTask2, asyncTask3 call asynchronously
    }
      asyncTask1, asyncTask2, asyncTask3 still in asynchronously calling
    await{
        asyncTask4();
        asyncTask5();
        asyncTask ();
          asyncTask4, asyncTask5, asyncTask  call asynchronously
    }
      asyncTask4, asyncTask5, asyncTask  ended
      asyncTask1, asyncTask2, asyncTask3 still in asynchronously calling
}

await method();
  asyncTask1, asyncTask2, asyncTask3 ended
```

People can make a mistake but machine must not.
I think what all of this (async await) is possible and just await for it =)

ggrnd0 commented on Feb 3 2015

Note for #98 (comment)
Another way is add information about method return type.
Write this:

```
var Method(){
    return new {Name = " lice",  ge = 21};
}
```

CodeHint will be:

```
T Method() where T is {string Name, int  ge}
```

So you can access properties of T:

```
var girl = Method();
Console.WriteLine($"{girl.Name} is {girl. ge} years old.");
```

Or something similar....

sirinath commented on Feb 4 2015

Code contacts can be:

1. Runtime
2. Static / compile time

and should have a way to specify then the contract is enforced.

sirinath commented on Feb 4 2015

Also is it possible to have the C# syntax as a library as in Racket so this can be imported and even modified and extended in certain scopes.

Neil65 referenced this issue on Feb 4 2015

Proposal for non-nullable references (and safe nullable references) #227 Open

AdamSpeight2008 commented on Feb 4 2015 Contributor

@sirinath See #129

mirhagk referenced this issue on Feb 6 2015

Proposal: Nested local functions and type declarations #259 Closed

ggrnd0 commented on Feb 8 2015

Using generic types without specify Type-parameters

For interface `numerable<T>` compiler will identify `numerable` like `numerable<ob ect>`

Also, missed Type-params can be determine from execution context.

If compiler cannot determine count and/or type-params compiler will receive error message.

If it is too hard for compiler than another way:

Type aliases

We can create type aliases by `using Console = ystem.Console;` now.

I offer for declaring aliases in code:

```
public alias numerable = numerable<ob ect>;
```

Interface `numerable` will be showed in CodeHint and will be replaced for `numerable<ob ect>` by compiler.

ggrnd0 commented on Feb 8 2015

Interface default methods

Each default method will accessible like explicit implemented interface method.

There can be colisions when `interface C` inherits multiple interfaces with same default methods.
I think they both must be ignored by compiler, and `interface C` will not inherits default implementation.

Or more, interfaces will not inherit default implementations from parent interfaces.
But they will inherit signatures of default methods.

Extention methods do something similar, but requires namespace importing...

Miista commented on Feb 8 2015

Don't you mean traits?

Søren Palmund

> Den 08/02/2015 kl. 12.14 skrev ggrnd0 notifications@github.com:
>
> Interface default methods.
>
> Each default method will accessible like explicit implemented interface method.
>
> There can be colisions when interface C inherits multiple interfaces with default methods.
> I think they both must be ignored by compiler, and interface C will not inherits default implementation.

> Or more, interfaces will not inherit default implementations from parent interfaces.
> But they will inherit signatures of default methods.
>
> For simple cases default interface methods can be powerfully then Extention methods becouse they do not require adding import of namespace.
>
> —
> Reply to this email directly or view it on GitHub.

ggrnd0 commented on Feb 8 2015

I forgot for traits.
Traits is better than interface default methods.

ggrnd0 commented on Feb 9 2015

`catch` and `finally` blocks after `using`

ggrnd0 commented on Feb 9 2015

Make all types to be assembly neutral!!!

jods4 commented on Feb 9 2015

`try {} fault {}` would be a small but nice addition.
It exists in MSIL and although C# doesn't expose it directly, it sometimes already generates it (e.g. inside iterators).

It's not something that is insanely useful but I have wished it existed a few times. It also makes C# more feature-complete with regard to `try` blocks, now that it has gained exception filters.

MadsTorgersen referenced this issue on Feb 10 2015
Proposal: Language support for Tuples #347 Open

ggrnd0 commented on Feb 10 2015

current method pointer. for using `nameof(thisMethod)` or doing reqursive calls `thisMethod(params)`

VSVeras commented on Feb 10 2015

How nice it would use enumerators equal to JAVA 8.

leppie commented on Mar 3 2015 Contributor

I know this is closed, but box and unbox user-defined operators on value types would be nice.

Use case: Would allow user to control interning without having to call another method.

ggrnd0 commented on Mar 10 2015

@leppie , is it https://msdn.microsoft.com/en-us/library/aa288476(v=vs.71).aspx ?

leppie commented on Mar 10 2015 Contributor

@ggrnd0: I guess it could/would be implemented using the same syntax (I would prefer it being more explicit), but currently you cannot implement user-defined conversions to and from `ob ect` . It is not allowed.

ggrnd0 commented on Mar 10 2015

@leppie , i forgot it... there would be an exclusion for value types. +1 for it.

ggrnd0 commented on Mar 10 2015

Another small and strange thing:
Make System.Int32 same like int: you cannot use System.Int32 for `enum` .
But it does not mater if `enum` will be reference type like java.

gafter commented on Mar 11 2015 Member

@ggrnd0 You can use System.Int32 for an enum base type in C# 6.

ggrnd0 commented on Mar 11 2015

-

bgorven commented on Mar 12 2015

Rather than NonNull, an approach which implies significant compile-time checking against nullable values, both on the part of the compiler and the user, consider Default.

What is Default?

Well, I'm glad you asked. Default is a wrapper type that can hold any value of type T, even null, but can always provide a non-null value when asked. That is, `var a = b` is semantically identical to `b. alue = b. alue ?? b. efault; var a = b. alue` . Thus the only caveat is that wherever a variable or field of type Default (T! from here on) is declared, it must be initialized with a value of type T!.

The power of Default, though, over any other approach, is the freedom it gives the programmer in deciding that default value. In a single function, you could declare three `string` s, one with a default value of "true", one with "false", and one with "FILE_NOT_FOUND".

To create that initial T!, we have a static helper method Default.Create, much like Tuple.Create, that provides several options: `T efault.Create(T t)` , which must be statically checked to only take an immediately constructed value or a string constant (trivially: `efault.Create(new T())` passes, all else fail), `T efault.Create<T>() where T : new()` or, `T efault.Create<T>( unc<T > factory)` .

Methods can then annotate their input parameters with T! for free. Values of type T can be returned or assigned to out parameters of type T! if and only if that variable is definitely assigned AND is never assigned a nullable value (except from the left side of `??`). The compiler then auto-wraps that value into a Default where it is both the current and default value.

Objects can not be instantiated with Default as a generic parameter, rather they must be instantiated with a user-supplied value-type implementing IDefault. Ideally, methods on the Default class would return such types, facilitating type inference in many situations. .Net's handling of struct generics allows the `b. alue = b. alue ?? b. efault; var a = b. alue` sugar to still work flawlessly even when b came from a generic container. Unfortunately, Default does break variance, as does Nullable.

Further sugar could allow class authors to specify a no-arg constructor as default (suggested syntax: `C() : default { }; string(char c = , int n = ) : default` , etc.) allowing T! of that type to be instantiated without going through the `efault` class' helpers.

One last concern is what happens if someone decides to `throw new Null eference xception()` inside a constructor or IDefault getter, but I suppose that's not a problem the type system could ever solve.

In closing, the advantages of this scheme are:

1. Conceptually simple, closely matching the precedent set by Nullable
2. Guaranteed safe: no nulls need ever be dereferenced.
3. Low friction:
   a) plain old nullable references can easily be assigned to T!
   b) you can even inspect the T! to determine whether it holds a concrete value, or the default.
4. Static checking is restricted to two places: variable/field definitions, and factory-method bodies. Both trivial.

gafter added the `Area-Language Design` label on Mar 19 2015

stimpy77 commented on Mar 24 2015

Since you're discussing C# design, can you please look into an `bort isposable` or somesuch that the compiler can pick up and clean up? To my knowledge this is a problem that has been going on since WCF premiered, where WCF clients can be in an open state or in a faulted state.

http://www.jondavis.net/techblog/post/2007/05/23/Windows-Communication-Framework-%28WCF%29-Beware-the-fake-IDisposable-implementation-!!.aspx

Perhaps a WCF client proxy could implement IAbortDisposable or IDisposeAbortable or something, and the syntax sugar to make use of it might be something like ...

```
using (var proxy = new My erviceProxy()) {
       do stuff, and then always automatically . ispose() or . bort()
} finally {
    if (proxy. s aulted) {
          do cleanup before the CL  automatically invokes . bort()
    }
}
```

This way .Dispose() always works but if it's faulted then there is an opportunity to clean up or something. I'm just brainstorming syntax ideas, but to manually do the following instead of `using(..) {}` ..

```
    if (this. tate == Communication tate.Closing
    this. tate == Communication tate.Closed
    this. tate == Communication tate. aulted)
{
    this. bort();
}
else
{
    this.Close();
}
```

.. is a nightmare.

svick commented on Mar 24 2015 Contributor

@stimpy77 To me, that sounds as if WCF should be fixed, instead of creating a language feature to fix it from the outside.

And, how is your `bort isposable` any better than normal `isposable`, whose implementation of `ispose()` looks something like the following?

```
if (this. s aulted)
    this. bort();
else
    this.Close();
```

asbjornu commented on Mar 25 2015

@stimpy77, I agree with **@svick**. The fact that WCF is so complex that it doesn't work with `using (...) {}` is not the problem of the C# language, it's the problem of WCF and should be fixed in WCF.

richardtallent commented on Apr 3 2015

Code contracts would be an improvement, but they treat the symptom, not the disease, as it were.

Say your method only accepts whole numbers, but uses an `int` parameter, which can theoretically be negative. Sure, putting a contract on the parameter to ensure it is >= 0 provides some protection, but it just moves the problem, so now the callee either has to do the same checks or defaulting before calling the method, or it has to handle the exceptions.

In the end, code contracts don't address the underlying issue -- *your parameter has the wrong logical type*. You're using `int` when you really want a far more limited type of either *zero or a positive integer*. Using an unsigned integer would be better, but has its own foibles, so we hold our nose and pretend that a signed type for a never-negative parameter is our best option.

So what's really going on is that your code contract is creating, logically, an anonymous type with restrictions not present in the base type. But by making it anonymous, it's not reusable, either between methods *or between the method and the caller*.

A better approach, IMHO, is to use the type system. E.g., if I want only whole numbers, I create a type that aliases int but does not allow negative numbers to be assigned to it:

```
public contract Whole nt : int where >=  ;
public contract Positive nt : int where >= 1 ?? 1;
public contract NN tring : string where  null ??  tring. mpty;

public class Customer {
    public Whole nt  ge { get; set; }
    public NN tring Name { get; set; }
    public Positive nt Num isits { get; set; }
}
```

This moves the responsibility back where it belongs: where the invalid value assignment occurs, not when some hapless method gets the bad value as an argument. It also encourages reuse, and provides the option of an *alternative default* rather than an exception when a bad value is assigned or the default value of the underlying type would be invalid.

For backward compatibility, it should always be possible to implicitly cast one of these contract types to their underlying type. This would allow, for example, `Collection<T>.Count` to return a `Whole nt` but callers can still assign the result directly to Int32.

Using the keyword `contract` above is just an example -- perhaps extending the concept of an Interface would be better.

adamralph commented on Apr 3 2015 Contributor

@richardtallent you make some very good points which should be taken note of. I guess the anti-pattern you've sniffed out there is a flavour of PrimitiveObsession.

asbjornu commented on Apr 3 2015

Brilliant ideas, **@richardtallent**! I'd love to see something like that in C# one day.

adamralph commented on Apr 4 2015 Contributor

Of course, we do already have a type system which allows us to encapsulate such things 😉

The 'contracts' demonstrated above are a shorthand for value types. Whilst the syntactic sugar may drastically reduce LOC, and I welcome such improvements, we shouldn't shy away from practising this encapsulation today just because the convenient shorthand doesn't yet exist.

We should also take care not to travel only the half the distance away from anti-patterns like primitive obsession:

```
public contract  ge: int where >=  ;

public class Customer
{
    public  ge  ge { get; set; }
}
```

richardtallent commented on Apr 4 2015

Adam, you're right, it is a form of PrimitiveObsession. I use structs to resolve this in my own code and it works brilliantly, but syntactic sugar around to accomplish the same thing more expressively would be a welcome change.

I agree about not "traveling half the distance," but I think there should be room for allowing contracts to be composed:

```
public contract Whole nt : int where >=  ;
public contract Positive nt : Whole nt where  =  ;
public contract Customer isitCount : Positive nt;
```

This is straightforward (but terribly wordy) to do with existing structs, and it's more DRY-friendly, since the same value constraints tend to pop up regularly.

I toyed around in my head the possibility of polymorphic contracts (e.g., `contract Positive nt : Whole nt, Not ero nt`) to allow even more flexible reuse, but I think I've convinced myself it would not be sufficiently better than strict composition to be worth the effort.

If Microsoft does grant us something like this, it would be great if they updated various BCL methods to use these more restrictive value types. For example, `List<T>. tem` could take WholeInt32 rather than plain old Int32. Maybe that would cause compatibility issues, even with implicit conversion to and from Int32. Still, it would be nice to move that direction.

MovGP0 commented on Apr 22 2015

I strongly prefer for `let` instead of `readonly` or `val` for readonly primitives (and types).

It is used in F#, so using `let` would be more consistent between different languages.

MovGP0 commented on Apr 22 2015

The idea of **@richardtallent** to define types might be a good idea. Howeven, contracts are also used ro ensure pre- and postconditions and not just type checking. So it might only be considered additionally.

jvlppm commented on Apr 22 2015

Why don't we have a method returning `var` (meaning `auto`)? because its not a variable?
We also have `let` for declarations inside linq queries.
What keyword would a method return for specifing `auto`?

I think this is becoming a mess.

jvlppm commented on Apr 22 2015

Support for something like this?

```
class Test {
    string attribute { get; }

    public Test(int input, string test) {
        int value = input   1;    static context.
        base(value);
        attribute = test;
    }
}
```

MovGP0 commented on Apr 22 2015

For implementing pattern matching, I do not like to use a `switch` statement, since it does not return a value:

```
int result = match animal {
    case is  og: 1;
    case is Cat: 2;
    default:  ;
}
```

MovGP0 commented on Apr 22 2015

@jvlppm it would be almost pointless to have a method that returns `var`.

Granted, the return type could be interfered by the compiler, which would ease refactorings. but it might also increase the programmer error rate by beeing less explicit.

I believe it is very important to have a language that decreases the potential error rate by design, which is why non-nullable reference types, readonly values, contracts, pattern matching, and even units/dimensions are a good idea.

MovGP0 commented on Apr 22 2015

@ggrnd0 the `catch` / `finally` block after a `using` block is not a very good idea, since having both in the same method violates SRP. Using an aspect to do the exception handling is a much better idea.

ig-sinicyn referenced this issue on Apr 23 2015

[Req] Public api for compile-time codegen? #2205 Closed

AdamSpeight2008 commented on Apr 24 2015 Contributor

Method Contracts should be generalized out to a traits system (#129).

MovGP0 commented on Apr 24 2015

@AdamSpeight2008 I agree that traits are a powerful concept.

However, I am unsure if that can do things like checking the order in which methods are called at compile time, as you can do with contracts.

MovGP0 commented on Apr 24 2015

@richardtallent i would use the `value` keyword in your example:

```
public contract Whole nt : int where value >=  ;
public contract Positive nt : int where value >= 1;
public contract Non mpty tring : string where value  = null    value  =  tring. mpty;
```

richardtallent commented on Apr 24 2015

> On Apr 24, 2015, at 12:02 AM, Johann Dirry notifications@github.com wrote:
>
> **@richardtallent** i would use the value keyword in your example:
>
> public contract WholeInt : int where value >= 0;
> public contract PositiveInt : int where value >= 1;
> public contract NonEmptyString : string where value != null && value != String.Empty;

Excellent idea. It also makes it easier to support conditions that need dotted notation, like the following:

public contract QuadrantIPoint : Point where value.X >=0 && value.Y >= 0;

public contract SSN : string where (value != null) && (value.Length==9) && value.All(c => c >= '0' && c <= '9');

Regarding your comment earlier about pre/post conditions, I agree. This concept would handle many common pre/post conditions, but since the contract is defined outside the method taking or returning it, it would have limited visibility and would not be able to, for example, compare its value to other arguments of the method. So it's not a complete replacement for the functionality of the current contract system, just a way to cleanly deal with classes of contracts that really just boil down to a more constrained version of another type.

--Richard

AlexeyRaga commented on Apr 28 2015

Abstraction over generics, anyone +1?

I mean, I can abstract over types with generics, e.g. `List<T>` , but for some reason I cannot abstract one level above e.g. `My ind< < >>` where `My ind` takes not a proper type, but a generic as a parameter which you would construct like, say, `My ind<List>` or `My ind<Task>`

This would be very and opens soooo many doors (in functional programming direction mainly, but not only). For example, it would finally be possible to generalise over `numerable<T>` and `bservable<T>` and to have functionality that can accept either of them.

Or it gives us the ability to write functions that preserve shape, like "you give me a list - I return you a list, you give me a task - I return you a task, you give me an observable - I return you an observable".

AlexeyRaga commented on Apr 28 2015

Sum types!
Someone has mentioned discriminated unions, from my POV it is much bigger than any contracts, especially if we have pattern matching.

AlenPelin commented on Aug 29 2015

Apologize if I'm writing to the wrong thread, please direct me if I should post the idea somewhere else or open separate issue for that.

From my experience I noticed same typical situation that I write over and over:

```
foreach (var language ersions in language ata)
{
  var language = language ersions. ey;
  var versions = language ersions. alue;
  foreach (var version ields in versions)
  {
    var versionNumber = version ields. ey;
    var fields = version ields. alue;
    Process(language, versionNumber, fields);
  }
}
```

It would be really handy if there is special support of IDictionary<TKey, TValue> in C# that can simplify the code and make it more expressive. Something like that:

```
foreach (var language, versions in language ata)
{
  foreach (var versionNumber, fields in versions)
  {
    Process(language, versionNumber, fields);
  }
}
```

Does anybody think it is a good idea?

alrz commented on Aug 29 2015 Contributor

@AlenPelin if `ey aluePair<,>` could be treated as a tuple, since `ictionary<,>` implements `numerable<>` you should get this out-of-the-box.

```
foreach ((var language, var versions) in language ata)
{
  foreach ((var versionNumber, var fields) in versions)
  {
    Process(language, versionNumber, fields);
  }
}
```

AlenPelin commented on Aug 29 2015

@alrz thanks, I will check that.
UPD: I reviewed the tuple concept and you seem to be right, just need to wait for tuples to be implemented first ^_^ which is quite complicated thought.

gafter commented on Aug 31 2015 Member

@alrz The `foreach` loop does not specify that it does decomposition, so even if we add a decomposition operator/assignment to the language, you would not get any special support in `foreach` without additional language support. Specifically, the syntax you wrote provided a type for the iteration variable `(var language, var versions)` but then you omitted the name of the variable - thus it would be a syntax error.

AlenPelin commented on Aug 31 2015

Thanks **@gafter** for your comments.
Well, as soon as `foreach` needs special support for either `ey aluePair`, or `tuples` and `ey aluePair` needs tuple representation, I believe first option is much easier to implement. Anyway, the question is still open: whether we really need that sugar or not.

alrz commented on Aug 31 2015 — Contributor

@gafter of course currently it is a syntax error! this is based on my assumption — that pattern matching, tuples, record types, etc are a group of features (in a functional language though) that always come together, otherwise it doesn't make much sense. If we have special syntax for them it should support decomposition everywhere not just in assignments. including

```
   switch statement
void  (ob ect o) {
    switch(o) {
    case (int item1, int item2): ...
    }
}

   method parameters
void  ((int item1, int item2)) { ... }
```

and aforementioned scenarios — which are supported in F# plus a powerful type inference, you don't even need to mention types as I did here.

and I didn't quite get what you meant by

> you omitted the name of the variable

of course I did because I'm not intrested in the tuple itself, but just the members. However, in Haskell you can have both at the same time with as-patterns e.g. `whole tuple (item1,item2)` hence it wouldn't be an unreasonable feature.

alrz referenced this issue on Aug 31 2015

Discussion: Scope of pattern variables, and tuple decomposition #4781 — Closed

gafter commented on Sep 1 2015 — Member

@alrz We already have one set of method overload resolution/dispatch rules. It is too late to replace them with a set that would have made sense if we had started with pattern matching. For example, if we were to do pattern matching for method dispatch as many functional languages do, we would select the overload based on the dynamic type of parameters rather than just upon the static type. Since that would change the behavior of existing code (`int x` is syntactically a pattern and also a parameter declaration), we cannot compatibly make that change.

This was referenced on Sep 29 2015

Contracts dart-lang/sdk#3049 — Open

[Proposal] enable code generating extensions to the compiler #5561 — Open

Incorporate feedback comments raised while open sourcing System.IO.IsolatedStorage dotnet/corefx#4975 — Merged

Proposal: Declaration of anonymous type #8404 — Open

gafter commented on Apr 25 2016 — Member

Design notes have been archived at https://github.com/dotnet/roslyn/blob/future/docs/designNotes/2015-01-21%20C%23%20Design%20Meeting.md but discussion can continue here.

This was referenced on Jul 21 2016

Proposal: Provide generalized language features in lieu of tuples #12654 — Closed

Proposal: C# language support of "weak" keyword #13950 — Open

Proposal: user-defined null/default check (non-defaultable value types / nullable-like types) #15108 — Closed

Proposal: Non-defaultable value types dotnet/csharplang#146 — Open