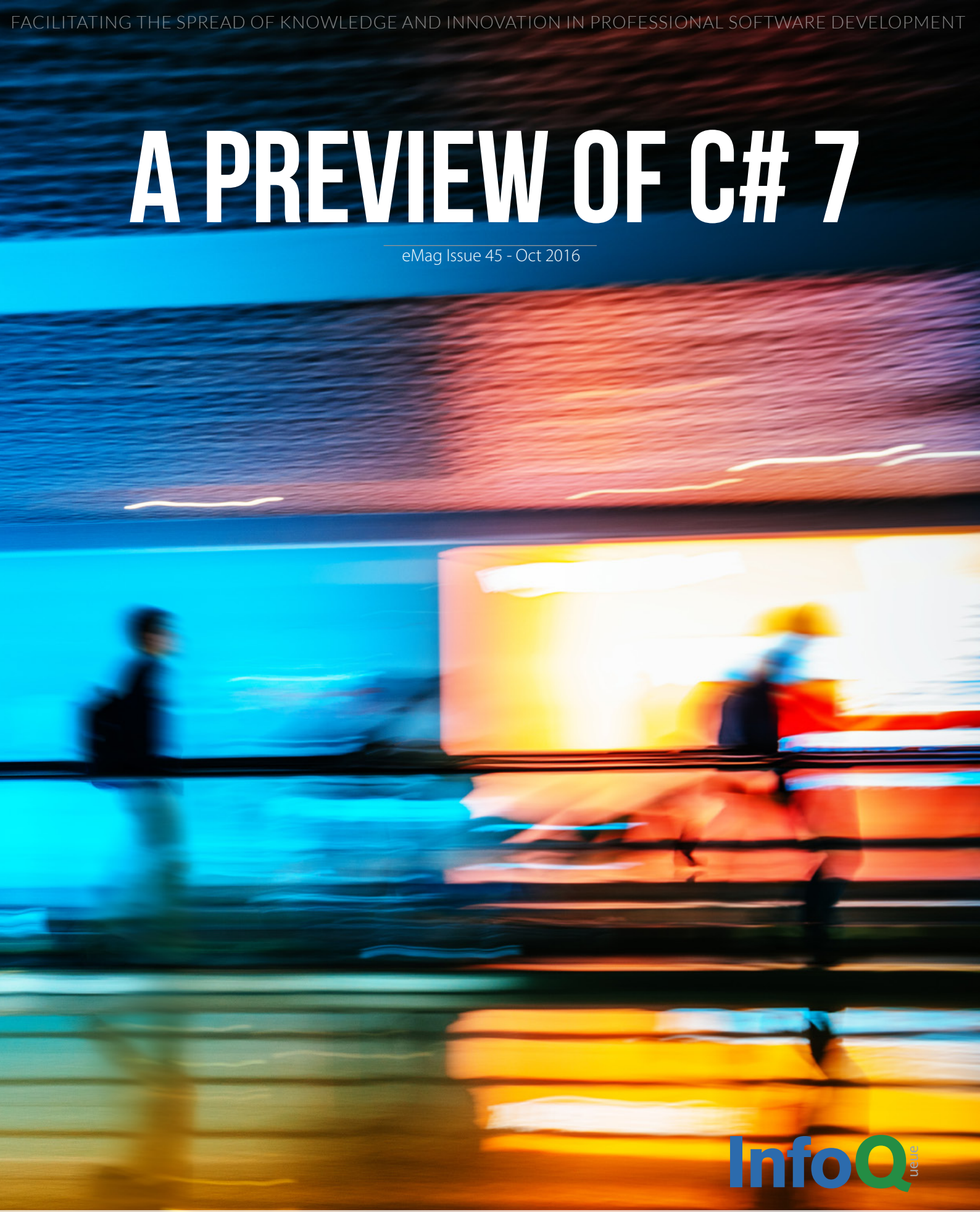
FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

A PREVIEW OF C# 7

eMag Issue 45 - Oct 2016



InfoQ queue

PRESENTATION
C# Today and Tomorrow

ARTICLE
C# 7 Features Previewed

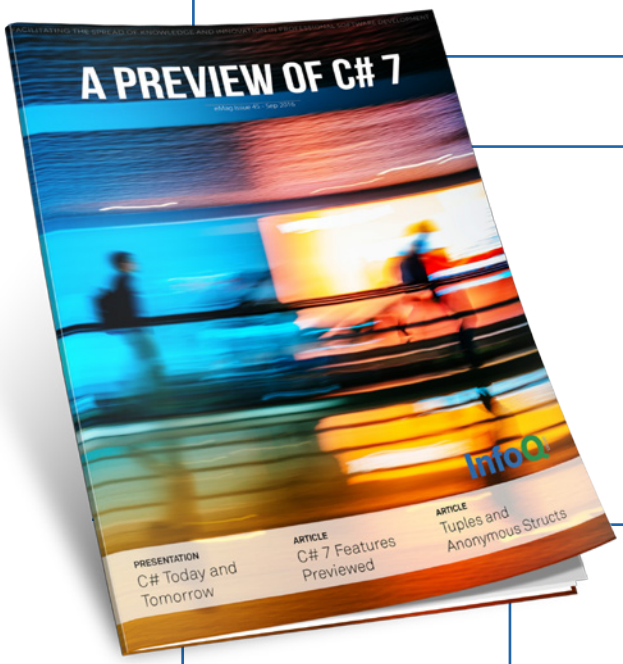
ARTICLE
Tuples and Anonymous Structs

C# Today and Tomorrow

Mads Torgersen talks about how C# is evolving, how the teams work in the open source space, and some of the future features and changes to the language (C# 7).

C# 7 Features Previewed

Over the last year we've shown you various features that were being considered for C# 7. With the 4th preview of Visual Studio 15, Microsoft has decided to demonstrate the features it has planned for the final release of C# 7.



Tuples and Anonymous Structs

The plans for C# 7 are being constantly reviewed by Microsoft, even as it nears completion. In this article, we'll be looking at some of the proposals starting with language support for tuples-- a data structure popular many other languages.

Managed Pointers

A big emphasis for many developers, especially those writing games or working on pure number crunching, is raw performance. One way to get more performance out of C# is to avoid allocating memory without having to copy structs instead. The next proposal shows how C# can expose the CLR managed pointer support to do just that.

Patterns and Practices in C# 7 Preview

C# 7 is going to be a major update with a lot of interesting new capabilities. Using the principles found in the .NET Framework Design Guidelines, we're going to take a first pass at laying down strategies for getting the most from these new features.

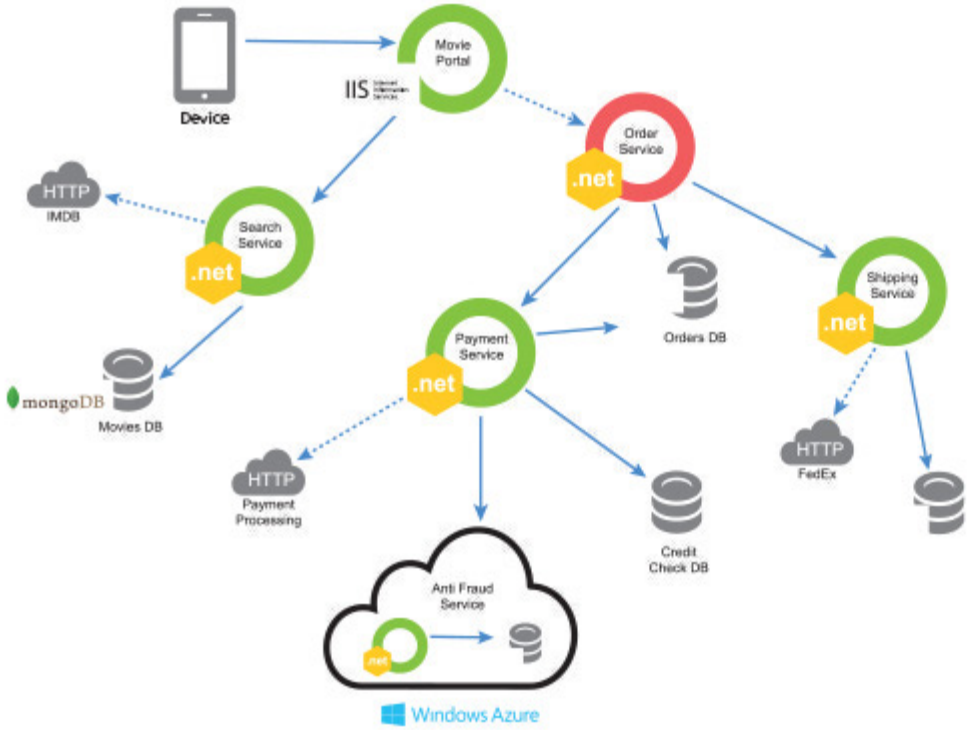
Advanced Pattern-Matching Features Removed from C# 7

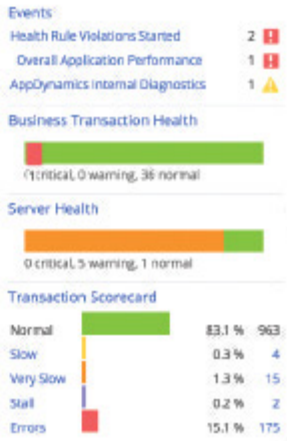
Advanced pattern matching features that were originally expected to be present in C# 7 have been recently excluded from the future branch and will not make it into the next version of the language.

APPDYNAMICS

Your .Net apps are critical, complex and distributed.

See **EVERYTHING** and release **FASTER** with AppDynamics.





.NET applications are becoming increasingly distributed and asynchronous. AppDynamics lets you monitor the performance of your applications in production or staging and drastically reduce time to root cause any problems that may occur.

Start your 15-day Free Trial at appdynamics.com/free-trial

JEFF MARTIN has an established career in the financial sector but follows the latest trends in the computer industry. He received his MBA from the University of Michigan and in his spare time enjoys traveling with his wife, reading, and programming. He just finished writing his first book, Visual Studio 2015 Cookbook, Second Edition. You can follow Jeff on Twitter @jeffemartin or on InfoQ.

A LETTER FROM THE EDITOR

Microsoft first released the C# programming language to the public in 2000 and since has carefully expanded the capabilities that C# offers in a measured way. The language has evolved through six releases to add everything from generics to lambda expressions to asynchronous methods and string interpolation.

As a C# developer, it is important to stay informed of the language's evolution. Understanding how and why the new language features are used is important whether you plan to incorporate them in your own projects or just read through the code of others.

In this eMag, we have curated a collection of new and previously published content that provides the reader with a solid introduction to C# 7 as it is defined today. We will start with "C# Today and Tomorrow", an informative look at the current plans for C# 7 according to Mads Torgersen. As the C# language project manager at Microsoft, Torgersen is the best possible guide to provide us with an informative look at C#'s design process.

Next, InfoQ's Jonathan Allen previews just what C# 7 is expected to include with his code-based article, "C# 7 Features Previewed". Allen continues with a more detailed look at tuples in "Tuples and Anonymous Structs" and moves on to "Managed Pointers".

The exact list of features for C# 7 is constantly evolving, and Sergio De Simone looks at the fate of advanced pattern matching in C# 7 in a short summary.

We complete this collection with Jonathan Allen's strategies for getting the most from these new features using the principles found in the .NET Framework Design Guidelines.

As De Simone's article reminds us, C# 7 is still in a pre-release state and what the final release will provide is still very much in flux. It is important to stay familiar with the current proposals so you know what to expect in the final release, and this eMag is one way to begin your journey.

C# Today and Tomorrow

By Jeff Martin

Developing a popular language is a complex undertaking. In the beginning, it is important to attract users with new features or a new way of thinking to make their work easier. Once the user base grows, though, the challenge becomes how to maintain it in such a manner to attract new users while not alienating the existing users that fueled its start. C# program manager Mads Torgersen recently presented his thoughts on this process at QCon, providing valuable insight into his thought process and future plans for C#.

So how do you evolve a language? Once you have a bunch of users out there, how do you think about what to do next with the language? On the one hand, at least when you have such a compatibility commitment that Microsoft does with C#, you have to balance between staying somewhat simple and improving aggressively so that you stay relevant to the evolving tasks of the world.

There's definitely some kind of a complexity-budget thinking that you have to do. It is important to pick the right things to massively improve for as little cost as possible in terms of giving away simplicity. Another consideration is that you want to make things better for the developers you already have versus becoming attractive to new developers who might be interested in C# while focused on a different language. The present management of C# is a balance between trying to capture interest from new users that fit new scenarios versus just bringing along the existing, trusted developer base.

Finally, how do you deal with new paradigms? In a certain sense, history is on the side of functional programming. C# is not a functional programming language but maybe it can become enough of one to work in those scenarios where functional programming is the optimal approach.

We have to stay true to the feel of C#. C# still has to be C# as it evolves. One of Torgersen's guiding themes is the idea that there should only be one code case needed to understand C#. ▶

Figure 1: Balancing C#'s evolution.

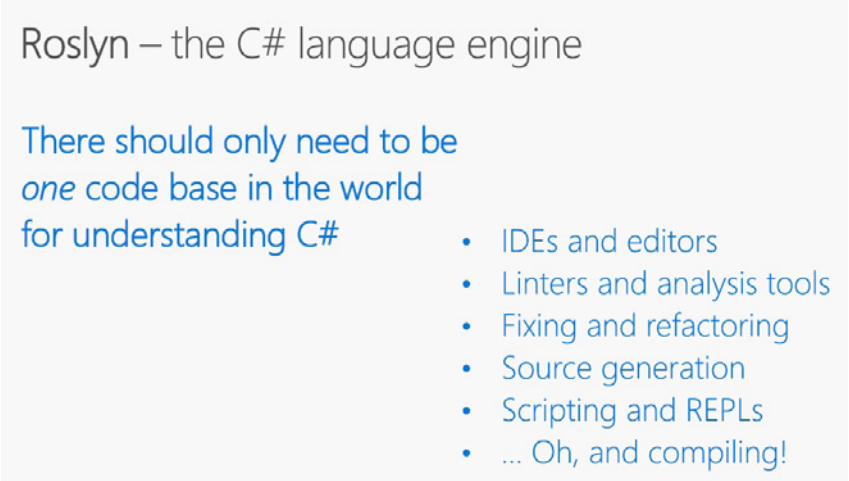


Figure 2: One code base for all.

Beyond the design and growth of the language is its place in the larger ecosystem. C#, and the .NET platform it's built on, originally targeted the traditional Win32 environment. Today, C# can run anywhere from desktops to servers to mobile devices. The arrival of .NET Core means that the C# being written today can just as easily run on Linux or Mac OS X as it can on Windows.

Since C# 7 will be released with Visual Studio "15" (the successor to Visual Studio 2015), Microsoft is developing it with a faster release cycle in mind. Torgersen notes that this means not everything originally planned will be included with C# 7; some of that will come later in C# 8.

Looking at the new language constructs

The first big new item Torgersen presented is tuples. Tuples are a useful feature for multiple return values, but they're also just as good for sticking multiple things in a data structure or to provide multiple values in a dictionary:

```
001 public static void
      tupelDemo()
002         {
003             var point =
      (x: 0, y: 0);
004             point =
      (point.x + 1, point.y
      + 3);
005
      WriteLine("My tuple: "
      + point);
006         }
```

When that method is called, it produces the following output:

```
001 My tuple: (1, 3)
```

Torgersen anticipates that some developers may have concerns regarding performance. Some developers may ask if this is going to allocate a new tuple every time around and wonder if this and that are going to be expensive. Fortunately, you don't have to worry because it won't be expensive — the tuples are struck in C#. They're not actually allocated every time around and are value types. When you pass them around, they get copied. They're not allocated on the heap. So it's completely free to

do this. This is as efficient as if we had multiple return values directly in the language just like you have multiple parameters. It's the same thing, like all values get passed on the stack.

Pattern matching is another big feature that is coming with C# 7. While the original feature set has been partially scaled back, much still remains for developers to take advantage of. What is a pattern? A pattern is sort of a declarative way to specify both a test over a given value and to extract information from it if the test is true. You can simultaneously ask questions about the value and get some extra information now.

Support of pattern matching allows developers to put a pattern in case clauses. The following code excerpt demonstrates this:

```
001     foreach (var v in values)
002                 {
003 // pattern matching with int and
     object
004                     switch (v)
005                     {
006 // pattern matching: based on integer
007                         case int i:
008                             r = (r.s +
     i, r.c + 1);
009                             break;
010 // pattern matching:  based on object
011                         case object[]
     l:
012                             var n =
     Tally(l);
013                             r = (r.s +
     n.sum, r.c + n.count);
014                             break;
015                     }
016                 }
```

Beyond C# 7

Looking ahead of C# 7, Torgersen described how the team really wishes C# had started out by distinguishing nullable and non-nullable reference types. A language like F# can do this but in C#, all reference types can be null, and these null reference exceptions can show up everywhere.

By introducing new syntax, a developer can indicate when something will be specifically null or not null. If a follow-up operation is being performed on a nullable reference type, the compiler can then object and complain to the user. The following excerpt illustrates these ideas:

```
string? n; // Nullable reference type
string  s; // Non-nullable reference type

n = null;  // Sure; it's nullable
s = null;  // Warning! Shouldn't be null!
s = n;     // Warning! Really!

WriteLine(s.Length); // Sure; it's not null
WriteLine(n.Length); // Warning! Could be null!

if (n != null) { WriteLine(n.Length); } // Sure; you checked
WriteLine(n!.Length);                   // Ok, if you insist!
```

Conclusion

Current plans call for C# 7 to include the following features:

- binary literals,
- digit separators,
- tuples,
- pattern matching (partially deferred?),
- local functions, and
- ref returns and locals.

Torgersen has already illustrated a few, as shown in this article, but we will review them all in greater detail throughout the rest of this eMag. ■

C# 7 Features Previewed

Jonathan Allen got his start working on MIS projects for a health clinic in the late 90's, bringing them up from Access and Excel to an enterprise solution by degrees. After spending five years writing automated trading systems for the financial sector he has decided to shift into high end user interface development. In his free time he enjoys studying and writing about western martial arts from the 15th thru 17th century.

Over the last year, we've seen various features that were being considered for C# 7. With the preview of Visual Studio 15, Microsoft has decided to demonstrate the features that will make it into the language's final release.

Tuple value types

.NET has a tuple type, but in the context of C#, there are a lot of problems. Because it's a reference type, you probably want to avoid using it in performance-sensitive code, as you'd have to pay a cost for garbage collection. And as the tuples are immutable, making them safer for sharing across threads, any change requires allocating a new object.

C# 7 will address this by offering a tuple as a value type. This will be a mutable type, making it more efficient when performance is essential. And as a value type, it makes a copy on assignment so there is little risk of threading issues.

To create a tuple, you can use this syntax:

```
001 var result = (5, 20);
```

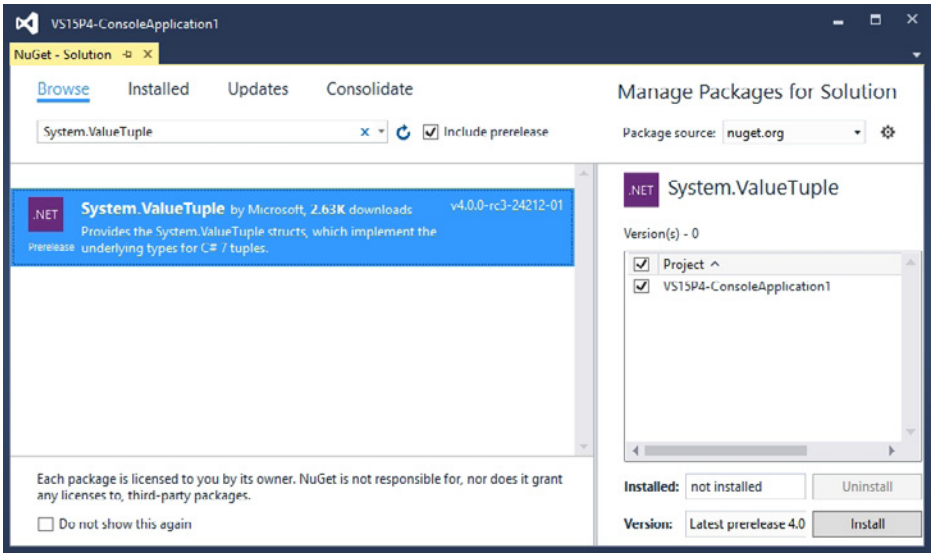
Optionally, you can name the values. This isn't necessary; it just makes the code more readable.

```
001 var result = (count: 5, sum: 20);
```

You may be thinking, "Great, but I could have written that myself." But the next bit of news is where this really matters.

Adding Tuple Support to Visual Studio 15 Preview 4

In order to experiment with tuples in your code, you will need to add the `System.ValueTuple` NuGet package to your solution. Open the NuGet package manager for your C# solution. Then, ensure nuget.org is the package source, search for System.ValueTuple with Include prerelease checked as shown in the following screenshot:



Multi-value returns

Returning two values from one function has always been a pain in C-style languages. You have to either wrap the results in some sort of structure or use output parameters. Like many functional languages, C# 7 will do the first option for you:

```
(int, int) Tally (IEnumerable<int>
     list)
```

Here we see the basic problem with generic tuples: there is no way to know what each field is for. So C# is offering a compiler trick that lets you name the results:

```
(int Count, int Sum) Tally
     (IEnumerable<int> list)
```

Note that C# isn't generating a new anonymous type. You are still getting back a tuple, but the compiler is pretending its properties are `Count` and `Sum` instead of `Item1` and `Item2`. Thus, these lines of code are equivalent:

```
var result = Tally(list);
Console.WriteLine(result.Item1);
Console.WriteLine(result.Count);
```

Tuples may be deconstructed (also called multi-assignment) from a single tuple into distinct variables. When performing deconstruction, you may do so into existing or freshly declared variables. The following code fragment demonstrates some of the possibilities with tuples:

```
 static void Main(string[] args)
       {
           Console.
WriteLine("Tuples");
           var result = (count: 5,
sum: 20);
           // var result = (5, 20);
alternate way to declare a tuple

           Console.
WriteLine(result);          //
prints (5, 20)
           Console.WriteLine(result.
count);    // prints 5
           Console.WriteLine(result.
sum);      // prints 20

           // deconstruction into
existing variables
           int count2, sum2;
           (count2, sum2) = result;
           // deconstruction into
new variables
           var (count3, sum3) =
result;
       }
```

Beyond simple utility functions, multi-value returns will be useful for writing asynchronous code, as async functions aren't allowed to use out parameters.

Pattern Matching: Decomposition

So far, we've seen just an incremental improvement over what is available in VB. The real power of pattern matching comes from decomposition, when you can tear apart an object. Consider this syntax:

```
001 if (person is Professor {Subject is
      var s, FirstName is "Scott"})
```

This does two things:

1. It creates a local variable named `s` with the value of `((Professor)person).Subject`.
2. It performs the equality check `((Professor)person).FirstName == "Scott"`.

Translated into C# 6 code, this is:

```
001 var temp = person as Professor;
002 if (temp != null && temp.FirstName ==
      "Scott")
003 {
004     var s = temp.Subject
```

Presumably, we'll be able to combine enhanced switch blocks in the final release.

Ref returns

Passing large structures by reference can be significantly faster than passing them by value, as the latter requires copying the whole structure. Likewise, returning a large structure by reference can be faster.

In languages such as C, you return a structure by reference using a pointer. This brings in the usual problems with pointers such as pointing to a piece of memory after it has been recycled for another purpose.

C# avoids this problem by using a reference, which is essentially a pointer with rules. The most important rule is that you can't return a reference to a local variable. If you'd try to do that, that variable would be on a portion of the stack that is no longer valid as soon as the function returns.

In a demonstration, C# instead returned a reference to a structure inside an array. Since it is effectively a pointer to an element in the array, the array itself can be modified. For example:

```
001 var x = ref FirstElement(myArray)
002 x = 5; //MyArray[0] now equals 5
```

The use case for this is highly performance-sensitive code. You wouldn't use it in most applications.

Binary literals

A minor addition is binary literals. The syntax is a simple prefix: for example, "5" would be "0b0101". The main use cases for this would be setting up flag-based enumerations and creating bitmasks for working with C-style interop.

```
var binary = 0b1010_1111_0000;
```

Digit Separators

Similar to binary literals is the addition of digit separators, which improve readability of number literals. Examples:

```
var hex = 0xFE_CD_BA;
var longNumber = 1_000_000_000;
```

Local functions

Local functions are functions that you define inside another function. At first glance, local functions look like slightly nicer syntax for anonymous functions. But they have some advantages:

- They don't require you to allocate a delegate to hold them. Not only does this reduce memory pressure, it also allows the compiler to inline the function.
- They don't require you to allocate an object when creating a closure. Instead, it only has access to the local variables. Again, this improves performance by reducing garbage-collection pressure.

Presumably, the second rule means that you can't create a delegate that points to a local function. Still, this offers organizational benefits over creating separate private functions to which you pass the current function's state as explicit parameters.

Partial class enhancements

The final feature demonstrated was a new way to handle partial classes. In the past, partial classes were based around the concept of generating code first. The generated code would include a set of partial methods that the developer could implement as needed to refine behavior.

With the new `replace` syntax, you can go the other way. The developer writes code in a straightforward fashion first and then the code generator comes in and rewrites it. Here is a simple example of what the developer may write:

```
001 public string FirstName {get; set;}
```

That's simple, clean, and completely wrong for a XAML-style application. Here's what the code generator will produce:

```
001 private string m_FirstName;
002 static readonly
      PropertyChangedEventArgs s_
      FirstName_EventArgs =new
003 PropertyChangedEventArgs("FirstName")
004 replace public string FirstName {
005     get {
006         return m_FirstName;
007     }
008     set {
009         if (m_FirstName == value)
010             return;
011     m_FirstName = value;
012     PropertyChanged?.Invoke(this, m_
      FirstName_EventArg);
013 }
```

By using the `replace` keyword, the generated code can literally replace the handwritten code with the missing functionality. In this example, we can even handle the tedious parts that developers often skip, such as caching `EventArgs` objects.

While the canonical example is property change notifications, this technique could be used for many aspect-oriented programming scenarios such as injecting logging, security checks, parameter validation, and other tedious boilerplate code.

To see these features in action, watch the Channel 9 video titled "The Future of C#" and the article Whats new in C# 7 (https://blogs.msdn.microsoft.com/dotnet/2016/08/24/whats-new-in-csharp-7-0/). ■

Tuples and Anonymous Structs

By Jonathan Allen

With C# 6 nearing completion, Microsoft is already planning for C# 7. While nothing is definite yet, the company is starting to categorize proposals in terms of interest and plausibility. Let's look at one of the proposals, language support for tuples.

The purpose of a tuple is to create a lightweight way to return multiple values from a function. Good tuple support eliminates the need for `out` parameters, which are usually considered to be cumbersome. Moreover, `out` parameters are incompatible with `async/await`, making them useless in many scenarios.

What about the Tuple class?

The .NET framework has a `Tuple` class since version 4 but most developers consider it useful only under limited circumstances. First of all, because `Tuple` is a class, memory has to be allocated to use it, which increases memory pressure and makes garbage-collection cycles more frequent. For it to compete with out parameters in terms of performance, it needs to be a structure.

The second issue involves API design. A return type of `Tuple<int, int>` doesn't really tell you anything. Every use of the function would require checking the documentation twice, once when writing it and again during code review. It would be far more useful if the return type were something like `Tuple<int count, int sum>`.

Anonymous structs

Consider these lines:

```
001 public (int sum, int count)
      Tally(IEnumerable<int> values) {
      ... }
002 var t = new (int sum, int count) {
      sum = 0, count = 0 };
```

Under the proposal, either line would define a new anonymous value type with `sum` and `count` properties. Note that unlike an anonymous class, the anonymous struct requires you to explicitly list the property names and types.

A benefit of using structs is that they define `Equals` and `GetHashCode` automatically — though you could argue that the default implementation isn't very efficient and the compiler should provide one instead.

Unpacking tuples

An important part of the tuple proposal is the ability to unpack tuples with a single line of code. Consider this block of code:

```
001 var t = Tally(myValues);
002 var sum = t.Sum;
003 var count = t.Count;
```

With unpacking, this simply becomes:

```
001 (var sum, var count) =
      Tally(myValues);
```

Not yet decided is whether or not you will be able to unpack a tuple without declaring new variables: in other words, omit var and use a pre-existing local variable instead.

Returning tuples

There are two proposals being considered for how tuples would be returned from a function. The first is fairly easy to understand:

```
001 return (a, b);
```

The second option has no return statement at all. Consider this example,

```
001 public (int sum, int count)
      Tally(IEnumerable<int> values)
002 {
003     sum = 0; count = 0;
004     foreach (var value in values) {
      sum += value; count++; }
005 }
```

Implicitly created local/return variables aren't a new concept. Visual Basic was originally designed that way, though it became unpopular once VB 7 introduced the return statement. It also mirrors what you would write if you were using `out` parameters. Still, not seeing a return statement would be somewhat disconcerting to many developers.

Other issues

Tuple support is a complex topic. While this article covers the day-to-day aspects, many details will have to be resolved from the compiler-writer and advanced-user perspectives.

Should tuples be mutable? This could be useful from a performance or convenience standpoint, but may make the code more error prone, especially when dealing with multithreading.

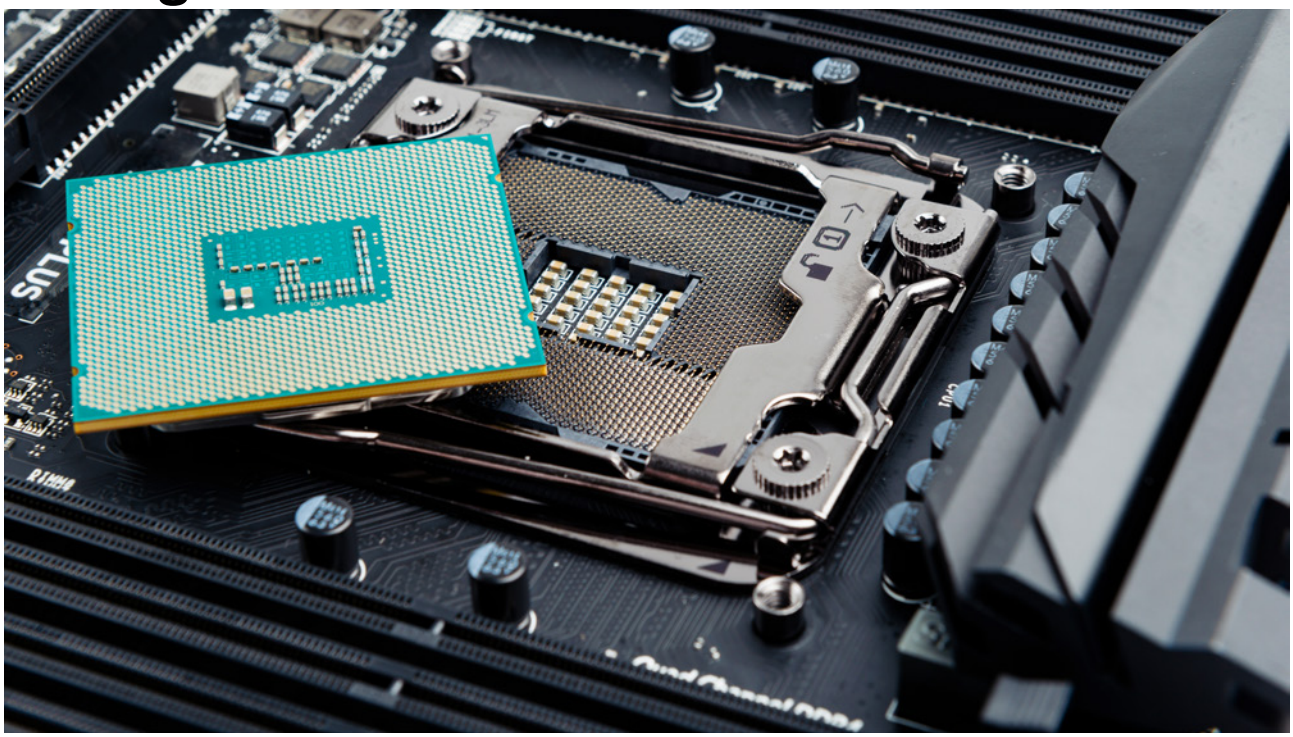
Should tuples be unified across assemblies? Anonymous types are not unified, but unlike anonymous types, these will be exposed as part of an API.

Can tuples be converted into other tuples? Superficially, they could, if they have the same type structure but different property names. Or the same property names, but wider property types.

If you pass a tuple of two values to a function that takes two parameters, will the tuple be automatically unpacked (splatted)? Conversely, can you "unsplat" a pair of arguments into one tuple parameter?

Much remains to be fleshed out. ■

Managed Pointers



By Jonathan Allen

A big emphasis for many developers, especially those writing games or working on pure number crunching, is raw performance. For them, nothing is more problematic than memory allocation. While allocation itself is cheap, too many allocations add to memory pressure and cause more frequent garbage collection cycles.

Heap allocated memory can also cause problems for the cache. If you have a list or array of reference types, the actual data is stored separately from the array, which means you may have to waste separate cache lines for the array and the objects referenced by the array. And if those objects were created at the same time, they may be scattered widely enough to need even more cache lines. This scattering of related data is known as poor locality.

Using value types ("structs" in C# parlance) can dramatically reduce the number of allocations and improve locality. However, there are limits to what you can reasonably do with structures. Because structs are designed to copy on assignment, you have to keep them small or risk a serious performance penalty that negates the reason for using them in the first place.

One way to reduce unnecessary copying is by passing value types to functions using a managed pointer. Currently, the only way to create a managed pointer in C# is by using a `ref` keyword as part of a parameter. While this addresses some performance scenarios, the CLR is capable of doing a lot more with managed pointers.

The "Ref Returns and Locals" proposal opens up two more options to C# programmers.

Ref local

Assuming that a is a local variable of type `int`, the proposal would allow you to create a ref local with this syntax:

```
001 ref int x = a;
```

Like a `ref` parameter, the ref local effectively becomes an alias for the indicated local variable, eliminating the need to make a copy. You can also use it to get a pointer to an array element or a field in another object.

```
001 ref int y = b[2];
002 ref int z = c.d;
```

In CLR terms, a ref local is called a "TypedReference". A TypedReference contains both the pointer to a location and information on what type of data may be stored at the location.

As a rule, a TypedReference is always a parameter or local variable. This is necessary because the CLR does not allow items on the heap to point inside other items on the heap. Nor may you return a TypedReference, as that would make it possible to return a reference to a local value that would of course no longer exist once the function exits.

Ref return

The second part of the proposal would allow you to return references from a function. This would allow for scenarios such as this:

```
001 public static ref TValue
      Choose<TValue>(
002       Func<bool> condition, ref
      TValue left, ref TValue right)
003 {
004       return condition() ? ref left
      : ref right;
005 }
006 Matrix3D left = […], right = […];
007 Choose(chooser, ref left, ref
      right).M20 = 1.0;
```

With this new syntax, there are no copies made to the struct anywhere in the sample. Instead, it is always creating and passing managed pointers around.

Unlike ref local, implementing this feature may require altering the CLR standard. As mentioned before, returning a TypedReference is normally not allowed. Technically speaking, you can do it — but it is considered to be not type-safe and thus "unverifiable". Using unverified code is not allowed in restricted security settings as it introduces the risk for serious bugs that are normally seen only in C/C++.

To mitigate this risk, the proposal states that you can only return a reference to something on the heap or an already existing ref/out parameter. Or in other words, the compiler would verify that you couldn't possibly return a reference to local variable. ■

Managed pointers can be used to avoid copying and the performance impact that entails.

Advanced Pattern-Matching Features Removed from C# 7

Sergio de Simone is a software engineer. Sergio has been working as a software engineer for over fifteen years across a range of different projects and companies, including such different work environments as Siemens, HP, and small startups. For the last few years, his focus has been on development for mobile platforms and related technologies. He is currently working for BigML, Inc., where he leads iOS and OS X development.

Advanced pattern-matching features that were originally expected to be present in C# 7 excluded in spring 2016 from the future branch and will not make it into the next version of the language.

The change of scope for C# 7 pattern matching has already materialized in Roslyn's GitHub repo. In particular, issue #10866 ("Split the features/patterns branch into two branches for subfeatures in/out C# 7") and pull request #10888 ("Remove evidence of advanced pattern-matching features for C# 7") thoroughly describe what this change is about.

As InfoQ reported in April 2016, pattern matching was going to be one the most appealing new features in C# 7, especially for programmers coming from a F# or Haskell background. Specifically, new pattern-matching features were expected to enhance `case` blocks by allowing for switching based on the type or range of a variable (e.g. `case int x:` or `case int x when x > 0`) and to add support for destructuring, which would allow developers to kind of tear apart an object into some of its components when it met given conditions while also creating local variables to refer to those components. An example of this is provided by the syntax `if (person is Professor {Subject is var s, FirstName is "Scott"})`.

Now, according to Roslyn issue #10866, both the syntaxes "`expression is Type identifier`" and "`case Pattern when expression`" for a few basic pattern forms have been moved to the future branch for inclusion in C# 7. The remaining fea-

tures have been left in the patterns/features branch, which hosts features "that might be delivered in a later release".

This means that the more advanced kinds of pattern matching, explained effectively by Reddit poster wreckedadvent, will not be available in C# 7, including:

- recursive pattern forms such as positional patterns (e.g., `p is Person("Mickey", *)`, property patterns (e.g., `p is Person {FirstName is "Mickey"}`), tuple patterns, wildcard `*`, etc.;
- the `let` keyword, supplying immutable vars (e.g., `let x = e2 when e2 else stmt;`), as opposed to mutable `var;`
- pattern matching based on user-defined code such as a user-defined `is` operator; and
- the `match` expression that would allow you to write:

```
001         var result = ...
002         let message = result
      match (
003           case Success<string>
      success: success.Result
004           case Failure err:
      err.Message
005           case *: "Unknown!"
006         );
```

There have been a few reactions within the community of C# developers. Those more keen on functional programming have expressed their disappointment about the lack of a feature that would have made C# more functional. Other developers, however, seem unconcerned or are glad that C# evolution is being managed in a disciplined and controlled way. ■

Pattern matching has great appeal, but its removal demonstrates C# 7's constant evolution.

Patterns and Practices in C# 7 Preview

Jonathan Allen got his start working on MIS projects for a health clinic in the late 90's, bringing them up from Access and Excel to an enterprise solution by degrees. After spending five years writing automated trading systems for the financial sector, he became a consultant on a variety of projects including the UI for a robotic warehouse, the middle tier for cancer research software, and the big data needs of a major real estate insurance company. In his free time he enjoys studying and writing about martial arts from the 16th century.

C# 7 is going to be a major update with a lot of interesting new capabilities. And while there are plenty of articles on what you can do with it, there's not quite as many on what you should do with it. Using the principles found in the .NET Framework Design Guidelines, we're going to take a first pass at laying down strategies for getting the most from these new features.

Tuple Returns

In normal C# programming, returning multiple values from one function can be quite tedious. Output parameters are an option, but only if you are exposing an asynchronous method. `Tuple<T>` is verbose, allocates memory, and doesn't have descriptive names for its fields. Custom structs are faster than Tuples, but litter the code with lots of single-use types. And finally, anonymous types combined with `dynamic` are very slow and lack static type checks.

All of these problems are solved with C#'s new tuple return syntax. Here is an example of the basic syntax:

```
001 public (string, string)
      LookupName(long id) // tuple return
      type
002 {
003     return ("John", "Doe"); // tuple
      literal
004 }
005 var names = LookupName(0);
006 var firstName = names.Item1;
007 var lastName = names.Item2;
```

The actual return type of this function is `ValueTuple<string, string>`. As the name suggests, this is a lightweight struct resembling the `Tuple<T>` class. This solves the type bloat issue, but leaves us with the same lack of descriptive names `Tuple<T>` suffers from.

```
001 public (string First, string Last)
      LookupName(long id)
002 var names = LookupName(0);
003 var firstName = names.First;
004 var lastName = names.Last;
```

The return type is still `ValueTuple<string, string>`, but now the compiler adds a TupleElementNames attribute to the function. This allows code that consumes the function to use the descriptive names instead of Item1/Item2.

WARNING: The TupleElementNames attribute is only honored by compilers. If you use reflection on the return type, you will only see the naked `ValueTuple<T>` struct. Because the attribute is on the function itself by the time you get a result, that information is lost.

The compiler maintains the illusion of extra types as long as it can. For example, consider these declarations:

```
001 var a = LookupName(0);
002 (string First, string Last) b =
      LookupName(0);
003 ValueTuple<string, string> c =
      LookupName(0);
004 (string make, string model) d =
      LookupName(0);
```

From the compiler's perspective, `a` is a `(string First, string Last)` just like `b`. Since `c` is explicitly declared as a `ValueTuple<string, string>`, there is no `c.First` property.

Example `d` shows where this design breaks down and causes you to lose a measure of type safety. It is really easy to accidentally rename fields, allowing you to assign one tuple into a different tuple that happens to have the same shape. Again, this is because the compiler doesn't really see `(string First, string Last)` and `(string make, string model)` as different types.

ValueTuple is Mutable

An interesting note about ValueTuple is that it is mutable. Mads Torgersen explains why:

The reasons why mutable structs are often bad, don't apply to tuples.

If you write a mutable struct in the usual encapsulated way, with private state and public, mutator properties and methods, then you are in for some bad surprises. The reason is that whenever those structs are held in a readonly variable, the mutators will silently work on a copy of the struct!

Tuples, however, simply have public, mutable fields. By design there are no mutators, and hence no risk of the above phenomenon.

Also, again because they are structs, they are copied whenever they are passed around. They aren't directly shared between threads, and don't suffer the risks of "shared mutable state" either. This is in contrast to the System.Tuple family of types, which are classes and therefore need to be immutable to be thread safe.

Note he said "fields", not "properties". This may cause problems with reflection-based libraries that consume the results of a tuple-returning function.

Guidelines for Tuple Returns

- CONSIDER using tuple returns instead of out parameters when the list of fields is small and will never change.
- DO use PascalCase for descriptive names in the return tuple. This makes the tuple fields look like properties on normal classes and structs.
- DO use `var` when reading a tuple return without deconstructing it. This avoids accidentally mislabeling fields.
- AVOID returning value tuples with a total size of more than 16 bytes. Note, reference variables always count as 4 bytes on a 32-bit OS and 8 bytes on a 64-bit OS.
- AVOID returning value tuples if reflection is expected to be used on the returned value.
- DO NOT use tuple returns on public APIs if there is a chance additional fields will need to be returned in future versions. Adding fields to a tuple return is a breaking change.

Deconstructing Multi-Value Returns

Going back to our LookupName example, it seems somewhat annoying to create a `names` variable that will only be used momentarily before it is replaced by separate locals. C# 7 also addresses this using what it calls "deconstruction". The syntax has several variants: ►

```
001 (string first, string last) =
     LookupName(0);
002 (var first, var last) = LookupName(0);
003 var (first, last) = LookupName(0);
004 (first, last) = LookupName(0);
```

In the last line of the above example, it is assumed the variables first and last were previously declared.

Deconstructors

Though similar in name to "destructor", a deconstructor has nothing to do with destroying an object. Just as a constructor combines separate values into one object, a deconstructor takes one object and separates it. A deconstructor allows any class to offer the deconstruction syntax described above. Let's consider the Rectangle class. It has this constructor:

```
001 public Rectangle(int x, int y, int
     width, int height)
```

When you call `ToString` on a new instance you get, "{X=0,Y=0,Width=0,Height=0}". The combination of these two facts tells us what order to present the fields in our custom deconstruction method.

```
001 public void Deconstruct(out int x,
     out int y, out int width, out int
     height)
002 {
003     x = X;
004     y = Y;
005     width = Width;
006     height = Height;
007 }
008
009 var (x, y, width, height) =
     myRectangle;
010 Console.WriteLine(x);
011 Console.WriteLine(y);
012 Console.WriteLine(width);
013 Console.WriteLine(height);
```

You may be wondering why output parameters are used instead of a return tuple. Part of the reason may be performance, as this reduces the amount of copying that needs to occur. But the main reason cited by Microsoft is it opens the door for overloading Deconstruct.

Continuing our case study, we note Rectangle has a second constructor:

```
001 public Rectangle(Point location, Size
     size);
```

We answer this with a matching deconstruct method:

```
001 public void Deconstruct(out Point
     location, out Size size);
002 var (location, size) = myRectangle;
```

This works so long as each deconstruct method has a different number of parameters. Even if you explicitly list out the types, the compiler won't be able to determine which Deconstruct method to use.

In terms of API design, structs would usually benefit from deconstruction. Classes, especially models or DTOs such as Customer and Employee, probably shouldn't have a deconstruct method. There is no way to resolve questions such as "Should it be (firstName, lastName, phoneNumber, email) or (firstName, lastName, email, phoneNumber)?" in a way that will make everyone happy.

Guidelines for Deconstructors

- CONSIDER using deconstruction when reading tuple return values, but be aware of mislabeling mistakes.
- DO provide a custom deconstruct method for structs.
- DO match the field order in a class's constructor, `ToString` override, and Deconstruct method.
- CONSIDER providing secondary deconstruct methods if the struct has multiple constructors.
- DO NOT expose Deconstruct methods on classes when it isn't obvious what order the fields should appear in.
- DO NOT expose multiple Deconstruct methods with the same number of parameters.

Out variables

C# 7 offers two new syntax options for calling functions with "out" parameters. You can now declare variables in function calls.

```
001 if (int.TryParse(s, out var i))
002 {
003     Console.WriteLine(i);
004 }
```

The other option is to ignore the output parameter entirely using a "wildcard".

```
001 if (int.TryParse(s, out *))
002 {
003     Console.WriteLine("success");
004 }
```

There is a lot of debate about the wildcard syntax. Many people don't like reusing the multiplication operator and would rather see a keyword such as "void" or "ignore". Others would like to use an under-

score (_), which is common in functional programming languages.

While the wildcards can be convenient, they imply a design flaw in the API. Under most circumstances, it would be better to simply offer an overload that omits the out parameters when they would otherwise normally be ignored.

Guidelines for Out Variables

- CONSIDER providing a tuple return alternative to out parameters.
- AVOID using out or ref parameters. [See Framework Design Guidelines]
- CONSIDER providing overloads that omit the out parameters so wildcards are not needed.

Local Functions and Iterators

Local functions are an interesting construct. At first glance they appear to be a slightly cleaner syntax for creating anonymous functions. Here you can see the differences.

```
001 public DateTime Max_Anonymous_
      Function(IList<DateTime> values)
002 {
003     Func<DateTime, DateTime, DateTime>
      MaxDate = (left, right) =>
004     {
005         return (left > right) ? left :
      right;
006     };
007
008     var result = values.First();
009     foreach (var item in values.
      Skip(1))
010         result = MaxDate(result,
      item);
011     return result;
012 }
013
014 public DateTime Max_Local_
      Function(IList<DateTime> values)
015 {
016     DateTime MaxDate(DateTime left,
      DateTime right)
017     {
018         return (left > right) ? left :
      right;
019     }
020
021     var result = values.First();
022     foreach (var item in values.
      Skip(1))
023         result = MaxDate(result,
      item);
024     return result;
025 }
```

However, once you start digging into them some interesting properties emerge.

Anonymous Functions vs. Local Functions

When you create a normal anonymous function, it always creates a matching hidden class to store the function. An instance of this class is created and stored in a static field on the same hidden class. Thus, once created there is no further overhead.

Local functions are different in that no hidden class is needed. Instead, the function is represented as a static function in the same class as its parent function.

Closures

If your anonymous or local function refers to a variable in the containing function, it is called a "closure" because it closes over or captures the local function. Here is an example,

```
001 public DateTime Max_Local_
      Function(IList<DateTime> values)
002 {
003     int callCount = 0;
004
005     DateTime MaxDate(DateTime left,
      DateTime right)
006     {
007         callCount++; <--The variable
      callCount is being closed over.
008         return (left > right) ? left :
      right;
009     }
010
011     var result = values.First();
012     foreach (var item in values.
      Skip(1))
013         result = MaxDate(result,
      item);
014     return result;
015 }
```

For anonymous functions, this requires a new instance of the hidden class each time the containing function is called. This ensures each call to the function has its own copy of the data that is shared between the parent and anonymous function.

The downside of this design is that each call to the anonymous function requires instantiating a new object. This can make it expensive to use, as it puts pressure on the garbage collector.

With a local function, a hidden struct is created instead of a hidden class. This allows it to continue ▶

storing pre-call data while eliminating the need to instantiate a separate object. Similar to the anonymous function, the local function is physically stored in the hidden struct.

Delegates

When creating an anonymous or local function, you'll often want to package it in a delegate so that you can use it in an event handler or LINQ expression.

Anonymous functions are, by definition, anonymous. So in order to use them, you always need to store them in a variable or argument as a delegate.

Delegates cannot point to structs (unless they are boxed, which has weird semantics). So if you create a delegate that points to a local function, the compiler creates a hidden class instead of a hidden struct. And if that local function is a closure, a new instance of the hidden class is created each time the parent function is called.

Iterators

In C#, functions that use yield return to expose an `IEnumerable<T>` cannot immediately validate its parameters. Instead, the parameter validation doesn't occur until `MoveNext` is called on the anonymous enumerator that was returned.

This isn't a problem in VB because it supports anonymous iterators. Here is an example from MSDN:

```
001 Public Function GetSequence(low As
      Integer, high As Integer) _
002 As IEnumerable
003     ' Validate the arguments.
004     If low < 1 Then Throw New
      ArgumentException("low is too low")
005     If high > 140 Then Throw New
      ArgumentException("high is too
      high")
006
007     ' Return an anonymous iterator
      function.
008     Dim iterateSequence = Iterator
      Function() As IEnumerable
009                               For
      index = low To high
010
      Yield index
011                               Next
012                           End
      Function
013     Return iterateSequence()
014 End Function
```

In the current version of C#, GetSequence and its iterator need to be entirely separate functions. With C# 7, these can be combined through the use of a local function.

```
001 public IEnumerable<int>
      GetSequence(int low, int high)
002 {
003     if (low < 1)
004         throw new
      ArgumentException("low is too
      low");
005     if (high > 140)
006         throw new
      ArgumentException("high is too
      high");
007
008     IEnumerable<int> Iterator()
009     {
010         for (int i = low; i <= high;
      i++)
011             yield return i;
012     }
013
014     return Iterator();
015 }
```

Iterators require building a state machine, so they behave like closures returned as a delegate in terms of hidden classes.

Guidelines for Anonymous and Local Functions

- DO use local functions instead of anonymous functions when a delegate is not needed, especially when a closure is involved.
- DO use local iterators when returning an `IEnumerator` when parameters need to be validated.
- CONSIDER placing local functions at the very beginning or end of a function to visually separate them from their parent function.
- AVOID using closures with delegates in performance sensitive code. This applies to both anonymous and local functions.

Ref Returns, Locals, and Properties

Structs have some interesting performance characteristics. Since they are stored in line with their parent data structure, they don't have the object header overhead of normal classes. This means you can pack them very densely in arrays with little or no wasted space. Besides reducing your overall memory overhead, this gives you great locality, making your CPU's tiny cache much more efficient. This is why people working on high performance applications love structs.

But if your struct is too large, you have to be really careful about making unnecessary copies. Microsoft's guideline for this is 16 bytes, which is enough for 2 doubles or 4 integers. That's not much, though sometimes you can stretch it using bit-fields.

You also have to be extremely careful with mutable structs. It is really easy to accidentally make changes to a copy of the struct when you were intending to modify the original.

Ref Locals

One way around this is to use smart pointers so that you never need to make a copy. Here is some performance sensitive code from an ORM I've been working on:

```
001 for (var i = 0; i < m_Entries.Length;
      i++)
002 {
003     if (string.Equals(m_Entries[i].
      Details.ClrName, item.Key,
      StringComparison.OrdinalIgnoreCase)
004         || string.Equals(m_
      Entries[i].Details.SqlName,
      item.Key, StringComparison.
      OrdinalIgnoreCase))
005     {
006         var value = item.Value ??
      DBNull.Value;
007
008         if (value == DBNull.Value)
009         {
010             if
      (!ignoreNullProperties)
011                 parts.Add($"{m_
      Entries[i].Details.QuotedSqlName}
      IS NULL");
012         }
013         else
014         {
015             m_Entries[i].
      ParameterValue = value;
016             m_Entries[i].UseParameter
      = true;
017             parts.Add($"{m_
      Entries[i].Details.QuotedSqlName}
      = {m_Entries[i].Details.
      SqlVariableName}");
018         }
019
020         found = true;
021         keyFound = true;
022         break;
023     }
024 }
```

The first thing you'll note is it doesn't use `for-each`. To avoid the copy, it has to use the old style for loop. And even then, all reads and writes are performed directly against the value in the `m_Entries` array.

With C# 7's ref locals, you could significantly reduce the clutter without changing the semantics.

```
001 for (var i = 0; i < m_Entries.Length;
      i++)
002 {
003     ref Entry entry = ref m_
      Entries[i]; //create a reference
004     if (string.Equals(entry.
      Details.ClrName, item.Key,
      StringComparison.OrdinalIgnoreCase)
005         || string.Equals(entry.
      Details.SqlName, item.Key,
      StringComparison.OrdinalIgnoreCase))
006     {
007         var value = item.Value ??
      DBNull.Value;
008
009         if (value == DBNull.Value)
010         {
011             if
      (!ignoreNullProperties)
012                 parts.Add($"{entry.
      Details.QuotedSqlName} IS NULL");
013         }
014         else
015         {
016             entry.ParameterValue =
      value;
017             entry.UseParameter =
      true;
018             parts.Add($"{entry.
      Details.QuotedSqlName} = {entry.
      Details.SqlVariableName}");
019         }
020
021         found = true;
022         keyFound = true;
023         break;
024     }
025 }
```

This works because a "ref local" is really a safe pointer. We say it is "safe" because the compiler won't allow you to point to anything ephemeral such as the result of normal function.

And in case you are wondering, "`ref var entry = ref m_Entries[i];`" is valid syntax. You cannot, however, have it unbalanced. Either `ref` is used for both the declaration and the expression or neither use it. ►

Ref Returns

Complementing this feature is the ref return. This allows you create copy-free function. Continuing our example, we can pull out the search behavior into its own static function.

```
001 static ref Entry FindColumn(Entry[]
     entries, string searchKey)
002 {
003     for (var i = 0; i < entries.
     Length; i++)
004     {
005         ref Entry entry = ref
     entries[i]; //create a reference
006         if (string.Equals(entry.
     Details.ClrName, searchKey,
     StringComparison.OrdinalIgnoreCase)
007             || string.
     Equals(entry.Details.SqlName,
     searchKey, StringComparison.
     OrdinalIgnoreCase))
008         {
009             return ref entry;
010         }
011     }
012     throw new Exception("Column not
     found");
013 }
```

In this example we returned a reference to an array element. You can also return references to fields on objects, ref properties (see below), and ref parameters.

```
001 ref int Echo(ref int input)
002 {
003     return ref input;
004 }
005 ref int Echo2(ref Foo input)
006 {
007     return ref Foo.Field;
008 }
```

An interesting feature of ref returns is the caller can choose whether or not to use it. Both of the following lines are equally valid:

```
001 Entry copy = FindColumn(m_Entries,
     "FirstName");
002 ref Entry reference = ref
     FindColumn(m_Entries, "FirstName");
```

Ref Returns and Properties

You can create a ref return style property, but only if the property is read only. For example,

```
001 public ref int Test { get { return
     ref m_Test; } }
```

For immutable structs, this pattern seems like a no brainer. There's no extra cost to the consumer, who can choose to read it as either a ref or normal value as they see fit.

For mutable structs, things get interesting. First of all, this fixes the old problem of accidentally trying to modify a struct returned by a property, only to have the modification lost to the ether. Consider this class:

```
001 public class Shape
002 {
003     Rectangle m_Size;
004     public Rectangle Size { get {
     return m_Size; } }
005 }
006 var s = new Shape();
007 s.Size.Width = 5;
```

In C# 1, the size wouldn't be changed. In C# 6, it would be a compiler error. In C# 7, we just add `ref` and everything works.

```
001     public ref Rectangle Size { get {
     return ref m_Size; } }
```

At first glance it looks like this will prevent you from overriding the whole size at once. But as it turns out, you can still write code such as:

```
001 var rect = new Rectangle(0, 0, 10,
     20);
002 s.Size = rect;
```

Even though the property is "read-only", this works exactly as expected. One just has to understand one isn't getting back a Rectangle, but a pointer to a location that holds Rectangles.

Now we've got a problem. Our immutable struct is no longer immutable. Even though individual fields cannot be altered, the whole value can be replaced via the ref property. C# will warn you about this by disallowing this syntax:

```
001 readonly int m_LineThickness;
002 public ref int LineThickness { get {
     return ref m_LineThickness; } }
```

Since there is no such thing as a read-only ref return, you can't create a reference to a read-only field.

Ref Returns and Indexers

Probably the biggest limitation of ref returns and locals is it requires a fixed point to reference. Consider this line:

```
001 ref int x = ref myList[0];
```

This won't work because a list, unlike an array, makes a copy of the struct when you read its value. Below is the actual implementation of List<T> from Reference Source.

```
001 public T this[int index] {
002     get {
003         // Following trick can reduce
      the range check by one
004         if ((uint) index >= (uint)_
      size) {
005             ThrowHelper.
      ThrowArgumentOutOfRangeException();
006         }
007         Contract.EndContractBlock();
008         return _items[index]; <--
      return makes a copy
009     }
```

This also affects `ImmutableArray<T>` and normal arrays when accessed via the `IList<T>` interface. However, you could create your own version of `List<T>` that defines its index as a ref return.

```
001 public ref T this[int index] {
002     get {
003         // Following trick can reduce
      the range check by one
004         if ((uint) index >= (uint)_
      size) {
005             ThrowHelper.
      ThrowArgumentOutOfRangeException();
006         }
007         Contract.EndContractBlock();
008         return ref _items[index]; <--
      return ref makes a reference
009     }
```

If you do this, you'll need to explicitly implement the `IList<T>` and `IReadOnlyList<T>` interfaces. This is because ref returns have a different signature than normal returns and thus don't satisfy the interface's requirements.

Since indexers are actually just specialized properties, they have the same limitations as ref properties; meaning you can't explicitly define setters and the indexer is writable.

Guidelines for Ref Returns, Locals, and Properties

- CONSIDER using ref returns instead of index values in functions that work with arrays.
- CONSIDER using ref returns instead of normal returns for indexers on custom collection classes that hold structs.
- DO expose properties containing mutable structs as ref properties.
- DO NOT expose properties containing immutable structs as ref properties.
- DO NOT expose ref properties on immutable or read-only classes.
- DO NOT expose ref indexers on immutable or read-only collection classes.

ValueTask and Generalized Async Return Types

When the Task class was created, its primary role was to simplify multi-threaded programming. It created a channel that let you push long running operations into the thread pool and read back the results at a later date on your UI thread. And when using fork-join style concurrency, it performed admirably.

With the introduction of async/await in .NET 4.5, some of its flaws started to show. As we reported in 2011 (see Task Parallel Library Improvements in .NET 4.5), creating a Task object took longer than was acceptable and thus the internals had to be reworked. This resulted in a "a 49 to 55% reduction in the time it takes to create a Task<Int32> and a 52% reduction in size".

That's a good step, but Task still allocates memory. So when you are using it in a tight loop such as seen below, a lot of garbage can be produced.

```
001 while (await stream.ReadAsync(buffer,
      offset, count) != 0)
002 {
003     //process buffer
004 }
```

And as been said many times before, the key to high performance C# code is in reducing memory allocations and the subsequent GC cycle. Joe Duffy of Microsoft wrote in Asynchronous Everything:

First, remember, Midori was an entire OS written to use garbage collected memory. We learned some key lessons that were necessary for this to perform adequately. But I'd say the prime directive was to avoid superfluous allocations like the plague. Even short-lived ones. There is a mantra that permeated .NET in the early days: Gen0 collections are free. Unfortunately, this shaped a lot of .NET's library code, ▶

and is utter hogwash. Gen0 collections introduce pauses, dirty the cache, and introduce beat frequency issues in a highly concurrent system.

The real solution here is to create a struct-based task to use instead of the heap-allocated version. This was actually created under the name ValueTask<T> and was published in the System.Threading.Tasks.Extensions library. And because `await` already works on anything that exposes the right method, you can use it today.

Manually Exposing ValueTask<T>

The basic use case for `ValueTask<T>` is when you expect the result to be synchronous most of the time and you want to eliminate unnecessary memory allocations. To start with, let's say you have a traditional task-based asynchronous method.

```
001 public async Task<Customer>
      ReadFromDBAsync(string key)
```

Then we wrap it in a caching method:

```
001 public ValueTask<Customer>
      ReadFromCacheAsync(string key)
002 {
003     Customer result;
004     if (_Cache.TryGetValue(key, out
      result))
005         return new
      ValueTask<Customer>(result); //no
      allocation
006
007     else
008         return new
      ValueTask<Customer>(ReadFromCache
      Async_Inner(key));
009 }
```

And add a helper method to build the async state machine.

```
001 async Task<Customer>
      ReadFromCacheAsync_Inner(string
      key)
002 {
003     var result = await
      ReadFromDBAsync(key);
004     _Cache[key] = result;
005     return result;
006 }
```

With this in place, consumers can call ReadFromCacheAsync with exactly the same syntax as ReadFromDBAsync;

```
001 async Task Test()
002 {
003     var a = await
      ReadFromCacheAsync("aaa");
004     var b = await
      ReadFromCacheAsync("bbb");
005 }
```

Generalized Async

While the above pattern is not difficult, this is rather tedious to implement. And as we know, the more tedious the code is to write, the more likely it is to contain simple mistakes. So the current proposal for C# 7 is to offer generalized async returns.

Under the current design, you can only use the async keyword with methods that return `Task`, `Task<T>`, or `void`. When complete, generalized async returns will extend that capability to anything "tasklike". Something is considered to be tasklike if it has an `AsyncBuilder` attribute. This indicates the helper class used to create the tasklike object.

In the feature design notes, Microsoft estimates maybe five people will actually create tasklike classes that gain general acceptance. Everyone else will most likely use one of those five. Here is our above example using the new syntax:

```
001 public async ValueTask<Customer>
      ReadFromCacheAsync(string key)
002 {
003     Customer result;
004     if (_Cache.TryGetValue(key, out
      result))
005         return result; //no
      allocation
006     else
007         return await
      ReadFromDBAsync(key); //unwraps the
      Task and re-wrap in ValueTask
008 }
```

As you can see, we've eliminated the helper method and, other than the return type, it looks just like any other async method.

When to Use ValueTask<T>

So should you use `ValueTask<T>` instead of `Task<T>`? Not necessarily. It can be a little hard to find, so we'll quote the documentation:

Methods may return an instance of this value type when it's likely that the result of their operations will

be available synchronously and when the method is expected to be invoked so frequently that the cost of allocating a new Task<TResult> for each call will be prohibitive.

There are tradeoffs to using a ValueTask<TResult> instead of a Task<TResult>. For example, while a ValueTask<TResult> can help avoid an allocation in the case where the successful result is available synchronously, it also contains two fields whereas a Task<TResult> as a reference type is a single field. This means that a method call ends up returning two fields worth of data instead of one, which is more data to copy. It also means that if a method that returns one of these is awaited within an async method, the state machine for that async method will be larger due to needing to store the struct that's two fields instead of a single reference.

Further, for uses other than consuming the result of an asynchronous operation via await, ValueTask<TResult> can lead to a more convoluted programming model, which can in turn actually lead to more allocations. For example, consider a method that could return either a Task<TResult> with a cached task as a common result or a ValueTask<TResult>. If the consumer of the result wants to use it as a Task<TResult>, such as to use with in methods like Task.WhenAll and Task.WhenAny, the ValueTask<TResult> would first need to be converted into a Task<TResult> using ValueTask<TResult>.AsTask, which leads to an allocation that would have been avoided if a cached Task<TResult> had been used in the first place.

As such, the default choice for any asynchronous method should be to return a Task or Task<TResult>. Only if performance analysis proves it worthwhile should a ValueTask<TResult> be used instead of Task<TResult>. There is no non-generic version of ValueTask<TResult> as the Task.CompletedTask property may be used to hand back a successfully completed singleton in the case where a Task-returning method completes synchronously and successfully.

This is a rather long passage, so we've summarized it in our guidelines below.

Guidelines for ValueTask<T>

- CONSIDER using ValueTask<T> in performance sensitive code when results will usually be returned synchronously.
- CONSIDER using ValueTask<T> when memory pressure is an issue and Tasks cannot be cached.
- AVOID exposing ValueTask<T> in public APIs unless there are significant performance implications.
- DO NOT use ValueTask<T> when calls to Task.WhenAll or WhenAny are expected.

Expression Bodied Members

An expression bodied member allows one to eliminate the brackets for simple functions. This takes what is normally a four-line function and reduces it to a single line. For example:

```
001 public override string ToString()
002 {
003     return FirstName + " " +
      LastName;
004 }
005 public override string ToString() =>
      FirstName + " " + LastName;
```

Care must be taken to not go too far with this. For example, let's say you need to avoid the leading space when the first name is empty. You could write:

```
001 public override string ToString() =>
      !string.IsNullOrEmpty(FirstName)
      ? FirstName + " " + LastName :
      LastName;
```

But then you might want to check for a missing last name.

```
001 public override string ToString() =>
      !string.IsNullOrEmpty(FirstName)
      ? FirstName + " " + LastName :
      (!string.IsNullOrEmpty(LastName) ?
      LastName : "No Name");
```

As you can see, one can get carried away quite quickly when using this feature. So while you can do a lot by chaining together multiple conditional or null-coalescing operators, you should exhibit restraint.

Expression Bodied Properties

New in C# 6 are expression bodied properties. They are useful when working with MVVM style models that use a Get/Set method for handling things such as property notifications.

Here is the C# 6 code:

```
001 public string FirstName
002 {
003     get { return Get<string>(); }
004     set { Set(value); }
005 }
```

And the C# 7 alternative:

```
001 public string FirstName
002 {
003     get => Get<string>();
004     set => Set(value);
005 }
```

While the line count hasn't gone down, much of the line-noise is gone. And with something as small and repetitive as a property, every little bit helps.

For more information on how Get/Set works in these examples, see "CallerMemberName" in the news report titled C#, VB.NET To Get Windows Runtime Support, Asynchronous Methods.

Expression Bodied Constructors

Also new to C# 7 are expression bodied constructors. Here is an example:

```
001 class Person
002 {
003     public Person(string name) =>
      Name = name;
004     public string Name { get; }
005 }
```

The use here is very limited. It really only works if you have zero or one parameters. As soon as you add a second parameter that needs to be assigned to a field/property, you have to switch to a traditional constructor. You also can't initialize other fields, hook up event handlers, etc. (Parameter validation is possible, see "Throw Expressions" below.)

So our advice is to simply ignore this feature. It is going to make your single-parameter constructors look different from all of your other constructors while offering only a very small reduction in code size.

Expression Bodied Destructors

In an effort to make C# more consistent, destructors are allowed to an expression bodied member just like methods and constructors.

For those who have forgotten, a destructor in C# is really an override of the finalizer method on `System.Object`. Considering how significantly it changes how the GC treats the class, I would prefer you write this:

```
001 protected override void Finalize()
002 {
003     ReleaseResources();
004 }
```

This way it is very obvious that you have a finalizer. Unfortunately, that's not an option in C#. Instead you are required to use the destructor syntax.

```
001 ~UnmanagedResource()
002 {
003     ReleaseResources();
004 }
```

One problem with this is it looks a lot like a constructor, and thus can be easily overlooked. Another is that it mimics the destructor syntax in C++, which has completely different semantics. But that ship has sailed, so let's move on to the new syntax.

```
001 ~UnmanagedResource() =>
      ReleaseResources();
```

Now we have a single, easily missed line that brings the object into the finalizer queue lifecycle. This isn't like a trivial property or `ToString` method, this is something really important that needs to be visible. So again I advise that you don't use it.

Guidelines for Expression Bodied Members

- DO use expression bodied members for simple properties.
- DO use expression bodied members for methods that just call other overloads of the same method.
- CONSIDER using expression bodied members for trivial methods.
- DO NOT use more than one conditional (a ? b : c) or null-coalescing (x ?? y) operator in an expression bodied member.
- DO NOT use expression bodied members for constructors and finalizers.

Throw Expressions

Superficially, programming languages can generally be divided into two styles:

- Everything is an expression
- Statements, declarations, and expressions are separate concepts

Ruby is an instance of the former, where even declarations are expressions. By contrast, Visual Basic represents the latter, with a strong distinction between statements and expressions. For example, there is a completely different syntax for "if" when it stands alone and when it appears as part of a larger expression.

C# is mostly in the second camp, but due to its C heritage it does allow you to treat assignment statements as if they were expressions. This allows you to write code such as:

```
001 while ((current = stream.ReadByte())
      != -1)
002 {
003     //do work;
004 }
```

For the first time, C# 7 will be allowing a non-assignment statement to be used as an expression. Without any changes to the syntax, you can now place a "throw" statement anywhere that's expecting a normal expression. Here are some examples from Mads Torgersen's press release:

```
001 class Person
002 {
003     public string Name { get; }
004
005     public Person(string name)
      => Name = name ?? throw new
      ArgumentNullException("name");
006
007     public string GetFirstName()
008     {
009         var parts = Name.Split(" ");
010         return (parts.Length
      > 0) ? parts[0] : throw new
      InvalidOperationException("No
      name!");
011     }
012
013     public string
      GetLastName() => throw new
      NotImplementedException();
014 }
```

In each of these examples, it is pretty obvious what's going on. But what if we move the throws expression?

```
001 return (parts.Length == 0) ? throw
      new InvalidOperationException("No
      name!") : parts[0];
```

Now it isn't quite so clear. While the left and right clauses are related, the middle clause has nothing to do with them. Seen pictorially, the first version has the "happy path" on the left and the error path on the right. The second version has the error path splitting the happy path in half, breaking the flow of the whole line. (Image 1)

Let's look at another example. Here we are including a function call in the mix.

```
001 void Save(IList<Customer> customers,
      User currentUser)
002 {
003     if (customers == null ||
      customers.Count == 0) throw new
      ArgumentException("No customers to
      save");
004
005     _Database.SaveEach("dbo.
      Customer", customers, currentUser);
006 }
007
008 void Save(IList<Customer> customers,
      User currentUser)
009 {
010     _Database.SaveEach("dbo.
      Customer", (customers == null ||
      customers.Count == 0) ? customers
      : throw new ArgumentException("No
      customers to save"), currentUser);
011 }
```

Already we can see the length alone is problematic (though long lines are not unheard of with LINQ). But to get a better idea of how one reads the code, we'll color the conditional orange, the function call blue, the function arguments gold, and the error path red. (Image 2)

Again, you can see context keeps bouncing around with the parameters found in three separate places.

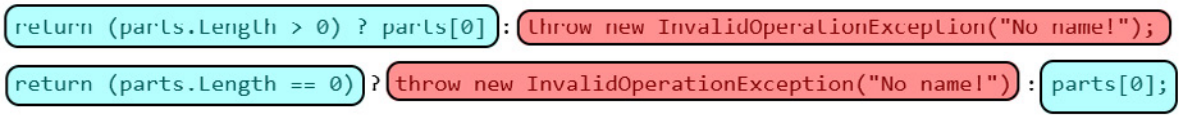


Image 1

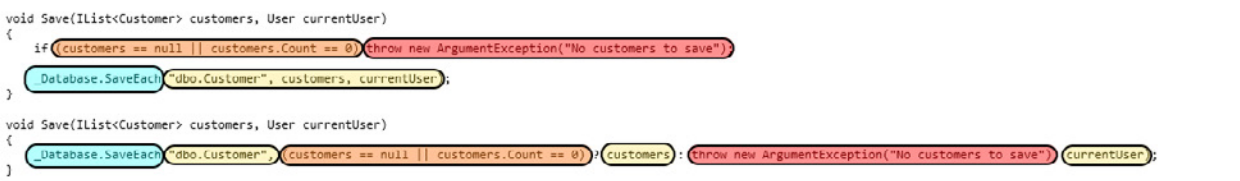


Image 2

Guidelines for Throw Expressions

- CONSIDER placing throw expressions on the right side of conditional (a ? b : c) and null-coalescing (x ?? y) operators in assignments/return statements.
- AVOID placing throw expressions on the middle slot of a conditional operator.
- DO NOT place throw expressions inside a function's parameter list.

For more information on how exceptions affect API design, see Designing with Exceptions in .NET.

Pattern Matching and Enhanced Switch Blocks

Pattern matching, which among other things enhances switch blocks, doesn't have any impact on API design. So while it certainly can make working with heterogeneous collections easier, it is still better to use shared interfaces and polymorphism when possible.

That said, there are some implementation details one should be aware of. Consider this example from the announcement in August:

```
001 switch(shape)
002 {
003     case Circle c:
004         WriteLine($"circle with radius
      {c.Radius}");
005         break;
006     case Rectangle s when (s.Length ==
      s.Height):
007         WriteLine($"{s.Length} x
      {s.Height} square");
008         break;
009     case Rectangle r:
010         WriteLine($"{r.Length} x
      {r.Height} rectangle");
011         break;
012     default:
013         WriteLine("<unknown shape>");
014         break;
015     case null:
016         throw new
      ArgumentNullException(nameof(shape));
017 }
```

Previously, the order in which case expressions occurred didn't matter. In C# 7, like Visual Basic, switch statements are evaluated almost strictly in order. This allows for `when` expressions.

The practical effect of this is you want your most common cases to be first in the switch block, just as you would in a series of if-else-if blocks. Likewise, if any check is particularly expensive to make then it should be near the bottom so it is executed only when necessary.

The exception to the strict ordering rule is the default case. It is always processed last, regardless of where it actually appears in the order. This can make the code harder to understand, so I recommend always placing the default case last.

Pattern Matching Expressions

While switch blocks will probably be the most common use for pattern matching in C#; that is not the only place they can appear. Any Boolean expression evaluated at runtime can include a pattern expression.

Here is an example that determines if the variable 'o' is a string, and if so tries to parse it as an integer.

```
001 if (o is string s && int.TryParse(s,
      out var i))
002 {
003     Console.WriteLine(i);
004 }
```

Note how a new variable named 's' is created by the pattern expression, then reused later by `TryParse`. This technique can be chained together for even more complex expressions:

```
001 if ((o is int i) || (o is string s &&
      int.TryParse(s, out i)))
002 {
003     Console.WriteLine(i);
004 }
```

For the sake of comparison, here's what the above code would typically look like in C# 6.

```
001 if (o is int)
002 {
003     Console.WriteLine((int)o);
004 }
005 else if (o is string && int.
      TryParse((string) o, out i))
006 {
007     Console.WriteLine(i);
008 }
```

It is too soon to tell if the new pattern matching code is more efficient the older style, but it can potentially eliminate some of the redundant type checks. ■

PREVIOUS ISSUES

44

Cloud Lock-In

Technology choices are made, and because of a variety of reasons--such as multi-year licensing cost, tightly coupled links to mission-critical systems, long-standing vendor relationships--you feel "locked into" those choices. In this InfoQ emag, we explore the topic of cloud lock-in from multiple angles and look for the best ways to approach it.

Exploring Container Technology in the Real World

43

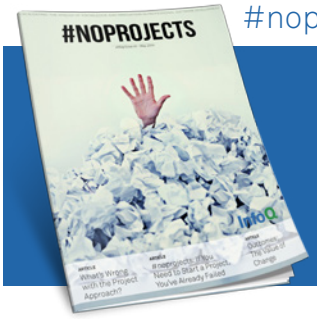
The creation of many competing, complementary and supporting container technologies has followed in the wake of Docker, and this has led to much hype and some disillusion around this space. This eMag aims to cut through some of this confusion and explain the essence of containers, their current use cases, and future potential.

Java Agents and Bytecode

42

In this eMag we have curated articles on bytecode manipulation, including how to manipulate bytecode using three important frameworks: Javassist, ASM, and ByteBuddy, as well as several higher level use cases where developers will benefit from understanding bytecode.

#noprojects



41

#NoProjects – a number of authors have challenged the idea of the project as a delivery mechanism for information technology product development. The two measures of success and goals of project management and product development don't align and the project mindset is even considered to be an inhibitor against product excellence. This emag presents some alternative approaches.