Though a comparatively minor release, C# 7.3 addresses some long outstanding complaints from C# 1 and 2.

Overload Resolution

Since version 1.0 C#'s overload resolution rules had a rather questionable design. Under some circumstances it would pick two or more methods as candidates, though all but one couldn't be used. Depending on the priority of the incorrectly selected method, the compiler would then either indicate no method matched or the matches were ambiguous.

C# 7.3 moves some of the checks to happen during overload resolution, rather than after, so the false matches don't cause compiler errors. The Improved overload candidates proposal outlines these checks:

> 1. When a method group contains both instance and static members, we discard the instance members if invoked without an instance receiver or context, and discard the static members if invoked with an instance receiver. When there is no receiver, we include only static members in a static context, otherwise both static and instance members. When the receiver is ambiguously an instance or type due to a color-color situation, we include both. A static context, where an implicit this instance receiver cannot be used, includes the body of members where no this is defined, such as static members, as well as places where this cannot be used, such as field initializers and constructor-initializers.
>
> 2. When a method group contains some generic methods whose type arguments do not satisfy their constraints, these members are removed from the candidate set.
>
> 3. For a method group conversion, candidate methods whose return type doesn't match up with the delegate's return type are removed from the set.

Generic Constraints: enum, delegate, and unmanaged

Developers have been complaining about the inability to indicate a generic type must be an enum since generics were introduced in C# 2.0. This has finally been addressed, and you can now use the `enum` keyword as a generic constraint. Likewise, you can now use the keyword `delegate` as a generic constraint.

These don't necessarily work the way you might expect. If the constraint is `where T : enum` then someone can use `Foo<System.Enum>` when what you probably meant is for them to use a subclass of `System.Enum`. Nonetheless, this should cover most scenarios for enums and delegates.

The Unmanaged type constraint proposal uses the "unmanaged" keyword to indicate the generic type must be a "type which is not a reference type and doesn't contain reference type fields at any level of nesting." This is intended to be used with low level interop code where you need to "create re-usable routines across all unmanaged types". Unmanaged types include:

- The primitives sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, IntPtr, or UIntPtr.
- Any enum type.
- Pointer types.
- User defined structs only containing the above.

Attributes on Hidden Fields

While auto-implemented properties are very useful, they do have some limitations. You cannot apply attributes to the backing field, because you can’t see it. While not usually an issue, this can be problematic when dealing with serialization.

The Auto-Implemented Property Field-Targeted Attributes proposal addresses this in a straight-forward fashion. When applying an attribute to an auto-implemented property, simply prepend the `field:` modifier.

```
[Serializable]
public class Foo {

    [field: NonSerialized]
    public string MySecret { get; set; }
}
```

Tuple Comparisons (== and !=)

While the name of the proposal, Support for == and != on tuple types, summarizes this feature nicely there are some details and edge cases to be aware of. The most important is a potentially breaking change:

> If someone wrote their own ValueTuple types with an implementation of the comparison operator, it would have previously been picked up by overload resolution. But since the new tuple case comes before overload resolution, we would handle this case with tuple comparison instead of relying on the user-defined comparison.

Ideally this custom ValueTuple type would follow the same rules as the C# 7.3 compiler, but there may be subtle differences in how it handles nested tuples and dynamic types.

Expression variables in initializers

In a way this reads like an anti-feature. Rather than adding capabilities, Microsoft removed restrictions on where expression variables can be used.

> We remove the restriction preventing the declaration of expression variables (out variable declarations and declaration patterns) in a ctor-initializer. Such a declared variable is in scope throughout the body of the constructor.

> We remove the restriction preventing the declaration of expression variables (out variable declarations and declaration patterns) in a field or property initializer. Such a declared variable is in scope throughout the initializing expression.
>
> We remove the restriction preventing the declaration of expression variables (out variable declarations and declaration patterns) in a query expression clause that is translated into the body of a lambda. Such a declared variable is in scope throughout that expression of the query clause.

The original reasoning for these restrictions were simply noted as “due to lack of time”. Presumably the restrictions reduced the amount of testing that needed to be done for previous versions of C# 7.

Stack Allocated Arrays

A rarely used, but important feature of C# is the ability to allocate an array inside the stack via the `stackalloc` keyword. This may offer performance improvements over normal arrays, which are allocated on the heap and cause GC pressure.

```
int* block = stackalloc int[3] { 1, 2, 3 };
```

There is some danger in using a stack allocated array. Since it requires taking a pointer against the stack, it can only be used in an unsafe context. The CLR attempts to mitigate this by enabling buffer overrun detection, which will cause the application to “terminate as quickly as possible”.

With C# 7.3, you gain the ability to initialize the array when it is created just like you would with normal arrays. The proposal lacks details, but Microsoft was considering pre-initializing a master array which then could be rapidly copied when the function is invoked. In theory this would be faster than creating an array and then initializing it element by element.

Note stack allocated arrays are meant for scenarios where you need lots of small, briefly used arrays. They shouldn’t be used for large arrays or deeply recursive functions, as you may exceed the amount of available stack space.

Stack Allocated Spans

A safe alternative to a stack allocated array is a stack allocated span. By eliminating the pointer, you also eliminate the possibility of a buffer overrun. Which in turn means you can use this without having to mark the method as unsafe.

```
Span<int> block = stackalloc int[3] { 1, 2, 3 };
```

Note `Span<T>` requires the `System.Memory` NuGet package.

Re-assignable Ref Locals

Ref local variables can now be reassigned just like normal local variables.

For other C# 7.3 proposals see the csharplang GitHub site.