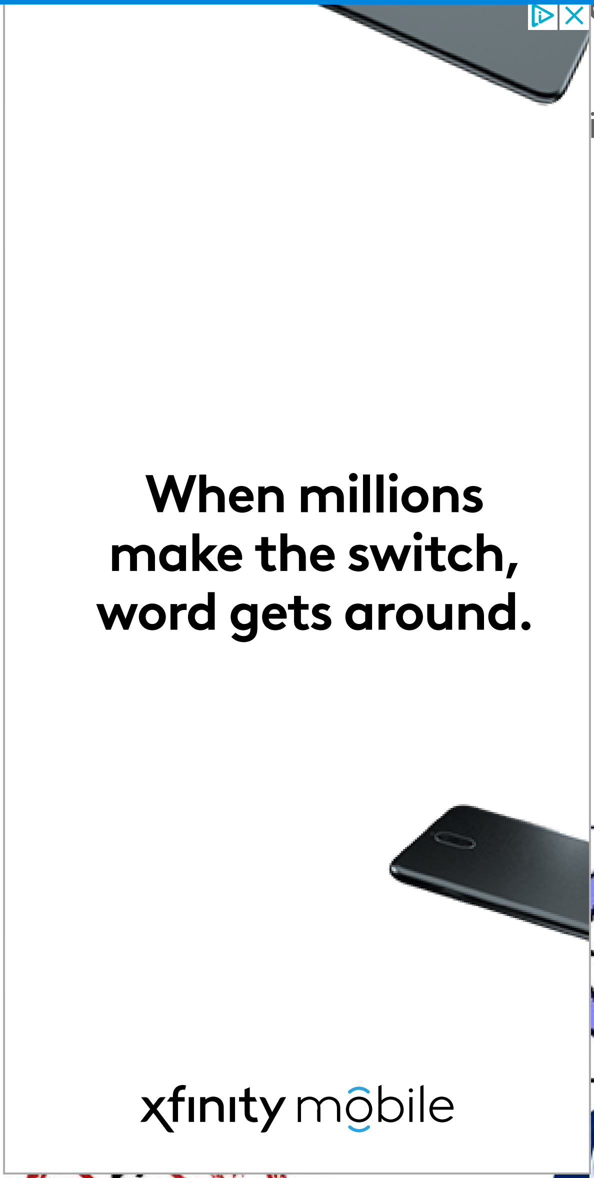
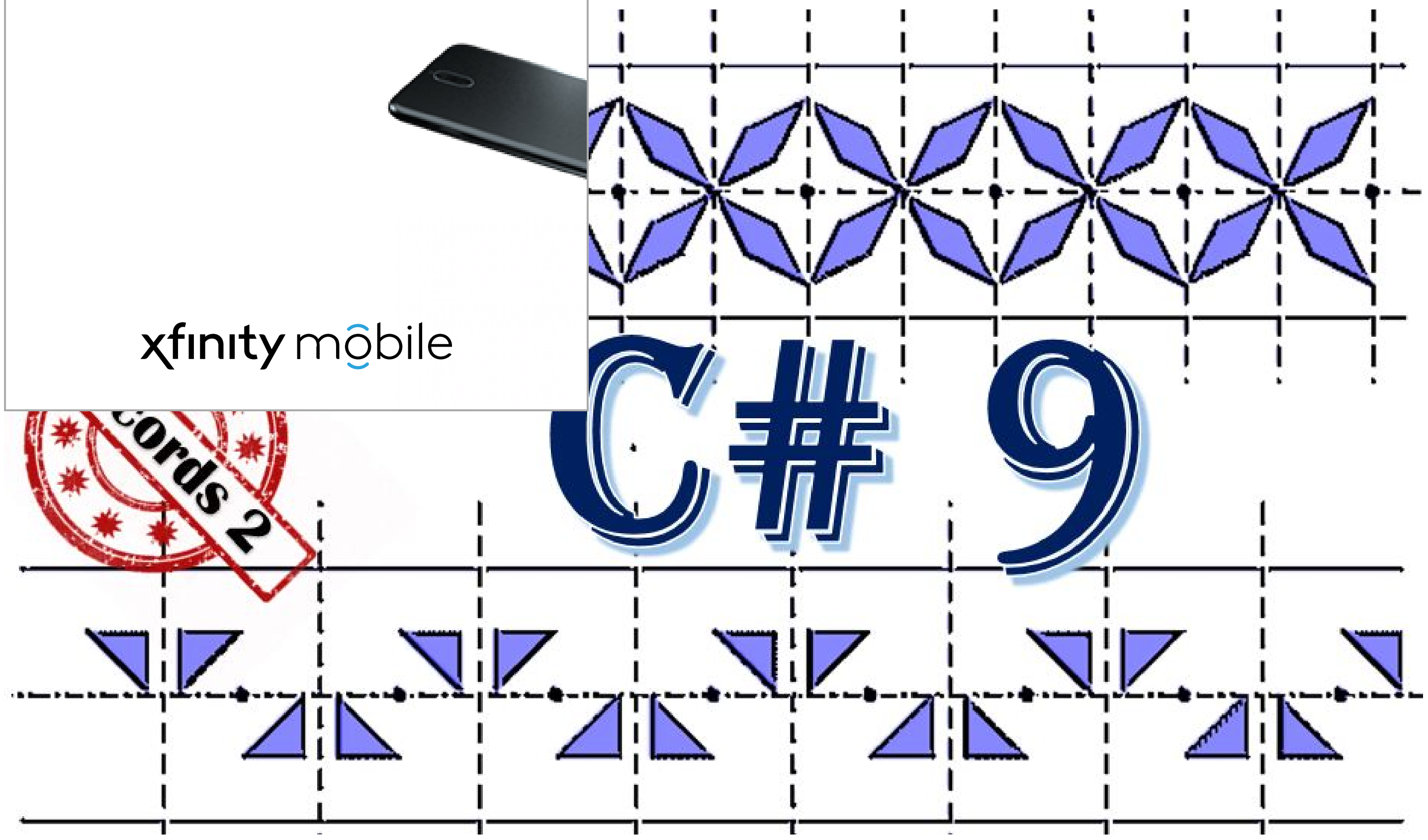
eep Dive Into

Post ▼ Ask Question

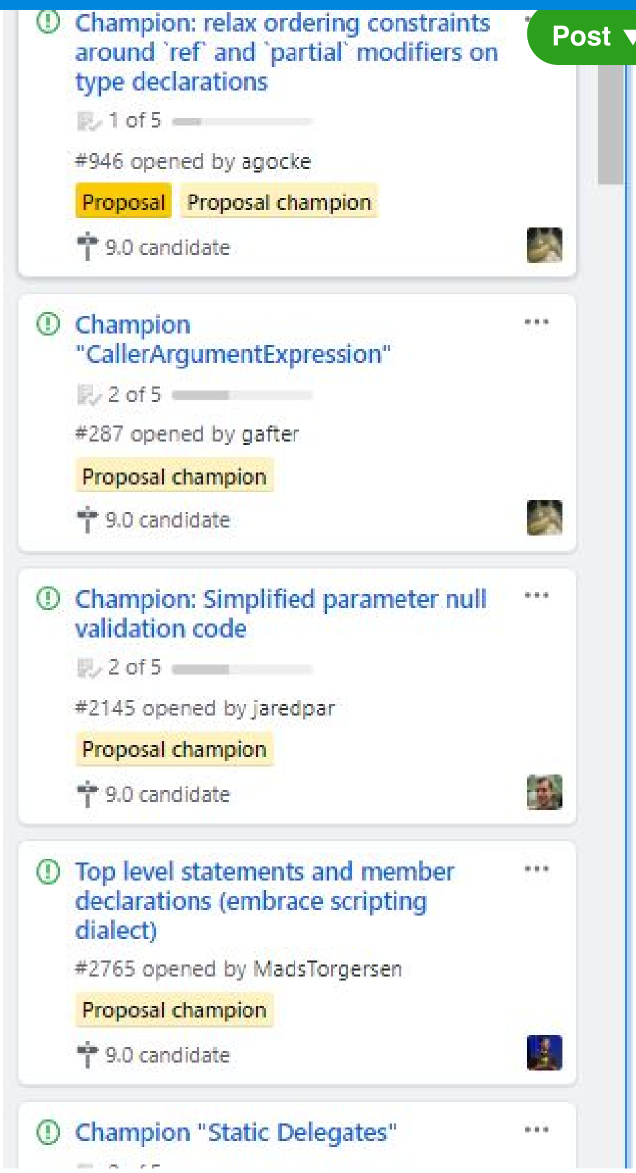
i Updated date Nov 11, 2020

176.5k 25 22





The official planned next C# release is C# 9. You can find in this link the Language Version Planning for C#·


Champion: relax ordering constraints around `ref` and `partial` modifiers on type declarations
1 of 5
#946 opened by agocke
Proposal
Proposal champion
9.0 candidate

Champion "CallerArgumentExpression"
2 of 5
#287 opened by gafter
Proposal champion
9.0 candidate

Champion: Simplified parameter null validation code
2 of 5
#2145 opened by jaredpar
Proposal champion
9.0 candidate

Top level statements and member declarations (embrace scripting dialect)
#2765 opened by MadsTorgersen
Proposal champion
9.0 candidate

Champion "Static Delegates"

As shown above, in the list, there are 27 proposals/features are planned for C# 9, but that does not mean that all of those proposals will be released in C# 9.

Which proposal will be released, and in which version? The .Net development team can only answer all these questions. They have the final decision, and they can change their minds all the time, both on proposed features and syntax/semantics of these features.

The most important features planned for C# 9, namely record types, unfortunately Discriminated Unions is moved to C# 10 plan, more pattern matching enhancements, and additional target-typing of existing constructs such as ternary and null-coalescing expressions.

In this article, I will describe Record and Discriminated Unions and the other proposals I will describe them briefly.

!!!!! Important !!!!!

Post ▼ Ask Question

still need a lot of work to move from the current strawman pro ta final designs will be.

Records

I have been waiting for a long time for this feature. Records are a lightweight type. They are nominally typed, and they might have (methods, properties, operators, etc.), and allow you to compare structural equality! Also, the record properties are read-only by default. Records can be Value Type or Reference Type. Records are useful to represents a complex data with many properties, like a database record or some model entity, DTO's, etc.

- Read-only properties => Immutable Type
- Equality implementations => Structural equality
- Pattern-matching support = is pattern, switch pattern etc.

The Proposal in GitHub here.

The most up-to-date proposal for records is here.

In my previous articles, I have described the Positional Records and this article I will talk about the Nominal Records. If you are not familiar with those terms, don't worry: I will simplify them as best I can. Basically, C# allows us to write the code in a positional or nominal style. Let us first take a look at the Object initializers.

Object initializers allows to create an object in a very flexible and readable format:

Microsoft Example:

```
// The following code:
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
// Can be created as follows
new Person
{
    FirstName = "Scott",
    LastName = "Hunter"
}
p.FirstName = “Alugili” // Works no Error!
```

"The one big limitation today is that the properties have to be mutable for object initializers to work" Init-only properties fix that!

```
public class Person
{
```

Post ▼ Ask Question

```
Person p = new Person
{
    FirstName = "Scott",
    LastName = "Hunter"
}
p.FirstName = “Alugili” // Error !
```

Note: init-only properties are also useful to make individual properties immutable.

Positional Records and the Nominal Records

C# allows you to write the code with a positional or nominal code style. Object initializer belongs to the nominal category. Until C# 8, the nominal category is restricted because it required writable properties. The introduced “init” accessor removes this limitation in C# 9.

Nominal Records

Nominal data is defined as data that is used for naming or labeling variables.

Example:

```
data class User {string Name, int Age};
var user = new User {Name = "Bassam", Age= 43};
var copyUser = user with {Name = "Thomas"};
```

Positional Records

The variables in ordinal data are listed in an ordered manner.

```
data class User(string Name, int Age);
var user = new User("Bassam", 43);
// Change the data
var copyUser = user with {Name = "Thomas"};
// Deconstruction
(string name, int age) = user;
// Pattern matching (Type Pattern)
if (user is ("Bassam", _))
    Console.Write($“My Name is {user.Name}”);
```

Structural Equality & Referential Equality

Usually, records are compared by structure and not by reference. In C# 9 we can make both approaches.

Referential Equality (Identity)

Identity: determines whether two objects share the same memory address.

Structural Equality

Means that two objects have equivalent content.

Equality: determines if two objects contain the same state.

Example:

```
data class User (string name, int age);
public static void Main(){
User user = new User( "Bassam", 43);
User user2 = new User( "Bassam", 43);
if (Equals(user, user2))
{
    Console.WriteLine("Structural Equality !");
}
if (ReferenceEquals(user, user2))
{
    Console.WriteLine("!!!! The code will not execute!!!!");
}
```

Output:
Structural Equality!

Besides, Records support inheritance, which makes Records in C# unique and varies from the most functional programming languages and F#.

Inheritance in Records Example

```
abstract data class User {string name, int age};
data class SuperUser:User {bool IsAdmin};
var user = new SuperUser(FirstName = "Bassam", Age= 43, IsAdmin true)
```

Inheritance - Records mutation and “with” expression

```
// The runtime- type is preserved after the coping, consider the belo
abstract data class User {string name, int age};
data class SuperUser:User {bool IsAdmin};
var user = new SuperUser(Name = "Bassam", Age= 43,IsAdmin true);
var copyUser = user with {Name = "Thomas"};
Console.WriteLine(copyUser.GetType().Name); // Output: SuperUser
```

Nominal Records in Depth

```
data class Point3D
{
    int X,
    int Y,
    int Z,
}
```

Equivalent to:

The proposed solution is a new modifier, init, that can be applied to properties and fields,

```
class Point3D
{
   int X { get; init;}
   int Y { get; init;}
   int Z { get; init;}
   ...
   ...
}
```

Creating Record:

```
void DoSomething()
{
  var point3D = new Point3D()
  {
    X = 1,
    Y = 1,
    Z =1
  };
}
```

Record from the old proposal (Positional)

Example, the following record with a primary constructor

```
data class Point3D(int X, int Y, int Z);
```

Would be equivalent to:

```
public class Demo
{
  public void CreatePoint()
  {
    var p = new Point3D(1.0, 1.0, 1.0);
  }
}
```

Would be equivalent to:

```
  public int Y { get; }
  public int Z { get; }
  public Point(int x, int y, int z)
  {
    X = x;
    Y = y;
    Z = z;
  }
  public void Deconstruct(out int X, out int Y, out int Z)
  {
    X = this.X;
    Y = this.Y;
    Z = this.Z;
  }
}
```

The final generation of the above would be:

```
class Point3D
{
  public initonly int X { get; }
  public initonly int Y { get; }
  public initonly int Y { get; }
  public Point3D(int x, int y, int z)
  {
    X = x;
    Y = y;
    Y = z;
  }

  protected Point3D(Point3D other)
  : this(other.X, other.Y, other.Z)
  { }

  [WithConstructor]
  public virtual Point With() => new Point(this);

  public void Deconstruct(out int X, out int Y, out int Z)

  {
  X = this.X;
  Y = this.Y;
  Z = this.Z;
  }
  // Generated equality
}
```

Using Records and the With-expression

Records proposal is introduced with the new proposed feature "with-expression". In programming, an immutable object is an object whose state cannot be modified after it is

Post ▼ Ask Question

```
public class Demo
{
  public void DoIt()
  {
    var point3D = new Point3D() { X = 1, Y = 1, Z =1  };
    Console.WriteLine(point3D);
  }
}
var newPoint3D = point3D with {X = 42};
```

The created new point (newPoint3D) just like the existing one (point3D), but with the value of X changed to 42.

This proposal is working very well with pattern matching.

Records in F#

Copy from F# MSDN example, type Point3D = {X: float; Y: float; Z: float}

```
let evaluatePoint (point: Point3D) =
  match point with
        | { X = 0.0; Y = 0.0; Z = 0.0 } -
> printfn "Point is at the origin."
        | { X = xVal; Y = 0.0; Z = 0.0 } -
> printfn "Point is on the x-axis.  Value is %f." xVal
        | { X = 0.0; Y = yVal; Z = 0.0 } -
> printfn "Point is on the y-axis. Value is %f." yVal
        | { X = 0.0; Y = 0.0; Z = zVal } -
> printfn "Point is on the z-axis. Value is %f." zVal
        | { X = xVal; Y = yVal; Z = zVal } -
> printfn "Point is at (%f, %f, %f)." xVal yVal zVal

evaluatePoint { X = 0.0; Y = 0.0; Z = 0.0 }
evaluatePoint { X = 100.0; Y = 0.0; Z = 0.0 }
evaluatePoint { X = 10.0; Y = 0.0; Z = -1.0 }
```

The output of this code is as follows.
Point is at the origin.
Point is on the x-axis. Value is 100.000000.
Point is at (10.000000, 0.000000, -1.000000).

Record types are implemented by the compiler, which means you have to meet all of those criteria and can't get them wrong.

So not only do they save a lot of boilerplate, they eliminate an entire class of potential bugs.

Post ▼ Ask Question

Examples for other languages that support both constructors and records.

F#

```
type Greeter(name: string) = member this.SayHi() = printfn "Hi, %s" n
```

Scala

```
class Greeter(name: String)
{
  def SayHi() = println("Hi, " + name)
}
```

Equality

Records are compared by structure and by reference

Example

```
void DoSomething()
{
    var point3D1 = new Point3D()
    {
        X = 1,
        Y = 1,
        Z =1
    };

    var point3D2= new Point3D()
    {
        X = 1,
        Y = 1,
        Z =1
    };

    var compareRecords = point3D1 == point3D2; // true, operator over
}
```

Discriminated Unions (C# 10)

The term Discriminated Unions (disjoint union) is borrowed from mathematics. A simple example to understand the term.

Below the Venn diagram is shown(disjoint union) by two non-overlapping closed regions and said inclusions are shown by showing one closed curve lying entirely within another.

Thus, A = {1, 2, 3} and B = {5, 7, 9} are disjoint sets; but the sets C = {3, 5, 7} and D = {7, 9, 11} are not disjoint; for, 7 is the common element of A and B.

Two sets A and B are said to be disjoint, if A ∩ B = ϕ. If A ∩ B ≠ ϕ, then A and B are said to be intersecting or overlapping sets.

Source: https://www.math-only-math.com/disjoint-of-sets-using-Venn-diagram.html#gallery[pageGallery]/4/

Example 2

Source: https://elmprogramming.com/type-system.html

Discriminated union (disjoint union) is also widely used in the programming languages (especially in FP), which used to sum the existing data types.

Discriminated unions in C# 10

It offers a way to define types that may hold any number of different data types. Their functionality is similar to F# discriminated union.

It used for values that can be enumerated. For example, Do you like C#Corner? Answer Yes or No? It can't be anything else other than those two specific values(Yes or No).
You can imagine it like enums in C#.

The official proposal here.

Discriminated unions are useful for heterogeneous data; data that can have individual cases, with valid and error cases; data that vary in type from one instance to another. Besides, it offers an alternative for small object hierarchies.

F# discriminated union example.

```
type Person = {firstname:string; lastname:string}  // define a
record type
type ByteOrBool = Y of byte | B of bool

type MixedType =
    | P of Person         // use the record type defined above
    | U of ByteOrBool     // use the union type defined above

let unionRecord = MixedType.P({firstname="Bassam"; lastname= "Alugili
let unionType1 =  MixedType.U( B true);   // Boolean type
let unionType2 =  MixedType.U( Y 86uy);   // Byte type
```

Using C# records, this could potentially be expressed in C#, using a form of union definition syntax, as,

```
// Define a record type
public class Person
{
  public initonly string Firstname { get; }
  public initonly string Lastname { get; }
};

enum class ByteOrBool { byte Y; bool B;} // Just for demo the syntax
```

We need a time that we need is a type that represents all possible integers PLUS all possible Booleans.

In other words, ByteOrBool is the sum type. In our case, the new type is the “sum” of the byte type plus the boolean type. And as in F#, a sum type is called a “discriminated union” type.

```
enum class MixedType
{
  Person P;
  ByteOrBool U;
}
```

Constructing a union instance,

```
// Note: the “new” might be not needed!
var person = new Person()
```

Post ▼ Ask Question

```
};

var unionRecord = new MixedType.P(person); // Record C# 9
var unionType1 = new MixedType.U( B true); // Boolean type
var unionType2 = new MixedType.U( Y 86uy); // Byte type
```

Usage of discriminated unions

With Pattern matching

Using the "subtype" names directly and the upcoming match expression. These below examples and are just for demo to make a better understanding for the proposal.
Exception handling like in Java,

```
try
{
  …
  …
}
catch (CommunicationException | SystemException ex)
{
    // Handle the CommunicationException and SystemException here
}
```

As type constraint

```
public class GenericClass<T> where T : T1 | T2 | T3
```

Generic class can be one of those types T1 or T2 or T3

Typed heterogeneous collections

```
var crazyCollectionFP = new List<int|double|string>
{1, 2.3, "bassam"};
```

Examples from the proposal,

Resultant type of combination of variables/values/expressions of different types through? :, ?? or switch expression combinators.

```
var result = x switch { true => "Successful", false => 0 };
```

The type of result here will be- string|int

If multiple overloads of some method have same implementations, Union type can do the job,

```
void logInput(int input) => Console.WriteLine($"The input is {inpu
void logInput(long input) => Console.WriteLine($"The input is {inp
```

Post ▼ Ask Question

Can be changed to

```
void logInput(int|long|float input) => Console.WriteLine($"The input
```

Maybe as return types,

```
public int|Exception Method() // returning exception instead of throw

public class None {}

public typealias Option<T> = T | None; // Option type

public typealias Result<T> = T | Exception; // Result type
```

More Examples here.

Enhancing the Common Type Specification

The proposal here.

Target typed null coalescing ('??') expression

It is about allowing an implicit conversion from the null coalescing expression.

Example

```
void M(List<int> list, uint? u)
{
  IEnumerable<int> x = list ?? (IEnumerable<int>)new[] { 1, 2 }; // C
  var l = u ?? -1u; // C# 8
}

void M(List<int> list, uint? u)
{
  IEnumerable<int> x = list ?? new[] { 1, 2 }; // C# 9

  var l = u ?? -1; // C# 9
}
```

Target-typed implicit array creation expression

Introducing the “new()” expression.

The official proposal Examples,

Post ▼ Ask Question

```
  new KeyValuePair<string, string>("Bar", b
}
```

Can be simplified to

```
IEnumerable<KeyValuePair<string, string>> Headers = new KeyValuePair<
[]
{
  new("Foo", foo),
  new("Bar", bar),
}
```

But you still need to repeat the type following the field/property initializer. The closest you can get is something like:

```
IEnumerable<KeyValuePair<string, string>> Headers = new[]
{
  new KeyValuePair<string, string>("Foo", foo),
  new("Bar", bar),
}
```

For the sake of completeness, I'd suggest to also make new[] a target-typed expression.

```
IEnumerable<KeyValuePair<string, string>> Headers = new[]
{
  new("Foo", foo),
  new("Bar", bar),
}
```

Target-typed new-expressions

“var” infers the left side, and this feature allows us to infer the right side.

Example

```
Point p = new (x, y);
ConcurrentDictionary> x = new();
Mads example: Point[] ps = { new (1, 4), new (3,-2), new (9, 5) }; //
```

Caller Expression Attribute (Not C# 9)

Allows the caller to ‘stringify’ the expressions passed in at a call site. The constructor of the attribute will take a string argument specifying the name of the argument to stringify.

Example

Post ▼ Ask Question

```
            [CallerArgumentExpression("argument                 Express
            [CallerArgumentExpression("low")] string lowExpression = null
            [CallerArgumentExpression("high")] string highExpression = nu
            if (argument < low) {
                throw new ArgumentOutOfRangeException(paramName: argument
            }
            if (argument > high) {
                throw new ArgumentOutOfRangeException(paramName: argument
 {argumentExpression} ({argument}) cannot be greater than {highExpress
            }
        }
        public static void NotNull < T > (T argument,
            [CallerArgumentExpression("argument")] string argumentExpress
        where T: class {
            if (argument == null) throw new ArgumentNullException(paramNa
        }
 }

 // CallerArgumentExpression: convert the expressions to a string!
 Verify.NotNull(array); // paramName: "array"

 // paramName: "index"
 // Error message by wrong Index:       "index (-1) cannot be less tha

 // "index (6) cannot be greater than array.Length - 1 (5)."
 Verify.InRange(index, 0, array.Length - 1);
```

Default in deconstruction

Allows the following syntax (int i, string s) = default; and (i, s) = default;.

Example

```
(int x, string y) = (default, default); // C# 7
(int x, string y) = default; // C# 9
```

Relax ordering of ref and partial modifiers

Allows the partial keyword before ref in the class definition.

Example

```
public ref partial struct {} // C# 7
public partial ref struct {} // C# 9
```

Parameter null-checking

Post ▼ Ask Question

Last meeting notes here.

Example

```
// Before C# 1..7.x
void DoSomething(string txt)
{
    if (txt is null)
    {
       throw new ArgumentNullException(nameof(txt));
     }
  …
}

// Candidate for C# 9
void DoSomething (string txt!)
{
  …
}
```

Skip locals init

Allow specifying System.Runtime.CompilerServices.SkipLocalsInitAttribute as a way to tell the compiler to not emit localsinit flag. SkipLocalsInitiAttribute is added to CoreCLR.
The end result of this will be that the locals may not be zero-initialized by the JIT, which is, in most cases, unobservable in C#.

In addition to that stackalloc data will not be zero-initialized. That is observable but also is the most motivating scenario.

Lambda discard parameters

Allow the lambda to have multiple declarations of the parameters named _. In this case, the parameters are "discards" and are not usable inside the lambda.

Examples

```
Func zero = (_,_) => 0;
(_,_) => 1, (int, string) => 1, void local(int , int);
```

Attributes on local functions

The idea is to permit attributes to be part of the declaration of a local function.

We should also test other attributes:"

[DoesNotReturn]
[DoesNotReturnIf(bool)]
[Disallow/Allow/Maybe/NotNull]
[Maybe/NotNullWhen(bool)]
[Obsolete]

Basic Example,

```
static void Main(string[] args)
{
  static bool LocalFunc([NotNull] data)
  {
    return true;
  }
}
```

The main use case for this feature,

Another example of using it with EnumeratorCancellation on the CancellationToken parameter of a local function implementing an async iterator, which is common when implementing query operators.

```
public static IAsyncEnumerable Where(this IAsyncEnumerable source, Fu
{
  if (source == null)
      throw new ArgumentNullException(nameof(source));

  if (predicate == null)
      throw new ArgumentNullException(nameof(predicate));

  return Core();

  async IAsyncEnumerable<T> Core([EnumeratorCancellation] Cancellatio
  {
       await foreach (var item in source.WithCancellation(token))
       {
        if (predicate(item))
        {
            yield return item;
        }
       }
  }
}
```

Advanced Example here.

Introduces a new set of native types (nint, nuint) the 'n' for native. The design of the new data types is planned to allow a one C# source file to use 32 naturally- or 64-bit storage depending on the host platform type and the compilation settings.

Example

The native type is depending on the OS,

```
nint nativeInt = 55; //take 4 bytes when I compile in 32 Bit host.
nint nativeInt = 55; //take 8 bytes when I compile in 64 Bit host wit
```

Function pointers

I remember the term function pointer from C/C++. FP is a variable that stores the address of a function that can later be called through that function pointer. Function pointers can be invoked and passed arguments just as in a normal function call.

The proposal here.

One of the new C# candidate features is called Function Pointers. The C# function pointer allows for the declaration of function pointers using the func* syntax. It is similar to the syntax used by delegate declarations.

Example

```
unsafe class Example
{
  void Example(Action<int> a, delegate*<int, void> f)
  {
    a(42);
    f(42);
  }
}
```

Enhancing Pattern Matching and More C# 9 stuff you can read in the following C# Preview article.

https://www.c-sharpcorner.com/article/c-sharp-9-preview/

Summary

You have read about the candidates in C# 9 Feature Status, and I have demonstrated them to you.

version.

Import: a lot of things are still in discussions. The proposed features and syntax/semantics might be changed, or the feature itself might be changed or removed. Only .NET developers can decide which features will be released in C# 9 and when it will be released. In the next article, I will continue with the proposals.

When the C# 9 will be released, I will make for you a cheat sheet as in C# 7 and C# 8. You can follow me on GitHub or my home page.

- C# 8
- C# 7.x

C# | C# 9 | C# 9 candidates | C# 9 features | C# proposals

Next Recommended Article
Deep Copy of Object in C#

Bassam Alugili TOP 500

Bassam Alugili is Senior Software Specialist and databases expert at STRATEC SE. STRATEC is a world-leading partner for fully automated analyzer systems, software for laboratory data management, and smart consumables.

https://stackoverflow.com/users/771040/bassam-alugili

364 715.6k 1

View Previous Comments

C# Corner ecome a member Login

Post ▼ Ask Question

Type your comment here and press Enter Key (Minimum 10 characters)

Thank you Michel, I will forward your comments to the editor!
Bassam Alugili
364 5.7k 715.6k
Aug 22, 2020
0 0 Reply

This is really great but wanted to mention a possible typo .. one of your class point3d has XYY properties instead of XYZ
Michel Comeau
1992 2 0
Aug 21, 2020
0 0 Reply

Good article, thanks
Aslan Kystaubayev
1990 4 0
May 23, 2020
1 1 Reply

You are welcome. This week I will publish the next part.
Bassam Alugili
364 5.7k 715.6k
May 26, 2020
0

Wow Great !!
Raju Paladiya
1431 566 6.8k
Feb 11, 2020
1 1 Reply

Thank you Raju!
Bassam Alugili
364 5.7k 715.6k
Feb 11, 2020
1

These changes will make c# easier. Good article.
Jefferson S. Motta
535 3.8k 166.3k
Feb 04, 2020
1 1 Reply

Thank you and I will update it in the next few days. keep in touch.
Bassam Alugili
364 5.7k 715.6k
Feb 04, 2020
0

Wow nicely explained
Ankur Mistry
38 32.9k 3.7m
Jan 01, 2020
1 1 Reply

Ankur Mistry thank you!
Bassam Alugili
364 5.7k 715.6k
Jan 09, 2020
0

Thank you Zhenyu Liu to open this good discussion. I will post it here to help the other develope
https://www.reddit.com/r/csharp/comments/eaiyif/some_questions_about_the_proposed_c_9_fe

C# Corner Become a member Login

Post ▼ Ask Question

HN about the article. some of the comments are really good: https://news.ycombinator.com/item?id=21719949

Bassam Alugili Dec 11, 2019

• 364 • 5.7k • 715.6k 0 2 Reply

Wow that many comments.

Mahesh Chand Dec 11, 2019

• Admin • 331.8k • 179.9m 1

Thank you! And in https://www.reddit.com/r/csharp, the article was trending up; after a few hours, it got more than 44 upvotes. But it was removed from the community administrator. The problem is that some developers using the article to attack Micorosft C# development team, and the C# compiler team does not accept the critical opinions. Maybe we can talk to Mads Torgersen in the MVPs summit so that he can help us to make C# communities working together and not against each other!
https://www.reddit.com/r/csharp/comments/e6vdgo/deep_dive_into_c_9/

Bassam Alugili Dec 12, 2019

• 364 • 5.7k • 715.6k 0

Wow, awesome

Zhenyu Liu Dec 09, 2019

• 1958 • 36 • 0 1 1 Reply

Thank you Zhenyu I am happy to hear that from you!

Bassam Alugili Dec 10, 2019

• 364 • 5.7k • 715.6k 1

Thank you Mahesh :)

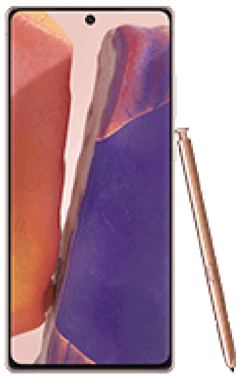
Bassam Alugili Dec 06, 2019

• 364 • 5.7k • 715.6k 0 0 Reply

C# Corner

Become a member Login

Post ▼ Ask Question



FEATURED ARTICLES

CRUD Operation With Image Upload In ASP.NET Core 5 MVC

The "Full-Stack" Developer Is A Myth In 2020

What Is Power BI

What is .NET?

What Is React And Why React Is So Popular

View All

C# Corner

Become a member Login

Post ▼ Ask Question

TRENDING UP

01 How To Create An Application Using ReactJS And Redux

02 Implement Global Exception Handling In ASP.NET Core Application

03 Prediction Using Supervised ML (Prediction Of Marks)

04 Azure Data Explorer - Kusto Query - Transform Rows To Columns

05 Building A Dashboard With ASP.NET Core And DotVVM

06 Azure Data Explorer - Working With Kusto Case Sensitivity

07 Rockin' The Code World with dotNetDave ft. Jeremy Likness - Show 5

08 What is .NET?

09 What Is React And Why React Is So Popular

10 Producer Consumer Pattern In C#

View All

C# Corner

Become a member Login

Post ▼ Ask Question

About Us Contact Us Privacy Policy Terms Media Kit Sitemap Report a Bug FAQ Partners

C# Tutorials Common Interview Questions Stories Consultants Ideas Certifications