# # Background

The C# design team is currently considering ways to add support non nullable reference types. Considerations so far have centered around flow based analyis and the "two-type" or "three-type" approach to references types (i.e. T, T! and T?) and whether such a scheme is enforceable via the compiler or an analyzer. This proposal attempts to define a more specific, but similar, scheme for non nullable references for C#, in particular in combination with generic types. Due to the scale of the exercise this proposal progressively adds changes to each version of C# from version 1.0 to version 6.0 to ensure that the features introduced with each of those versions works well with non nullable references.

## ## C# 1.0

The term variable, if not further qualified, is taken to mean any of the seven categories of variables : static variables, instance variables, array elements, value parameters, reference parameters, output parameters and local variables.

### ### Effective Nullability

At a given location in the executable code of a function member, a variable or expression is assigned an effective nullability, via static flow analysis, of either effectively nullable, effectively non nullable or effectively unusable. A variable, property or return type of reference type may be annoted with the non nullable annotation (!). Effective nullability is similar to definite assignment states in that affects how and when a variable or expression may be used rather than affecting the type of the variable, property or return type.

typeof(string!) == typeof(string) // true

string! s1 = "";

string s2 = "";

s1.GetType() == s2.GetType() // true

The non nullable annotation may not be used on void as it is not a type.

Given two types T and U where there exists an implicit conversion of references of type T to references of type U then an effectively non nullable expression of type T is implictly convertable to an effectively nullable or non nullable expression of type U. There is not an implicit conversion from effectively nullable expressions of type T to effectively non nullable expressions of type U.

A variable of value type or annotated as non nullable starts in an effectively non nullable state if it is initially assigned otherwise it starts in an effectively unusable state. It is an error to use a variable that is effectively unusable.

An effectively non nullable expression may not be passed as an argument for a reference or output parameter that is effectively nullable.

The following forms of expressions are considered effectively non nullable:

* a literal, other than the null literal (including string literals)
* the result of the new operator
* the this expression
* a definitely assigned effectively non nullable variable (including via implicit boxing conversions)
* Any expression of value type
* a method invocation or property access with a return/property type that is a reference type annotated as a non nullable
* a cast expression to a non null reference (i.e. (string!)s)
* a conditional expression (i.e. ternary operator) where the 2nd and 3rd operands are effectively non nullable.
* The result of the typeof operator

All other variables and expressions not already defined as effectively non nullable or unusable as considered effectively nullable. Assigning an effectively non nullable variable or expression to a variable of value type or annotated as non nullable changes the variable being assigned to be definitely assigned and therefore effectively non nullable.

The compiler will insert a null check against an effectively nullable variable or expression when cast to an effectively non nullable expression and throw a NullReferenceExpressions if the expression evaluates to null. E.g.

string! s = (string!)null; // The compiler could potentially detect the cast of a null literal and raise a compiler error

The type used with an as or is expression may not be annotated as non nullable.

### ### Arrays

Arrays may contain non nullable elements. For example

string![] words1; // An array of non nullable elements

string[]! words2; // A non nullable array with nullable elements

string![]![]! words3; // A non nullable array of non nullable arrays of non nullable elements

An array of with elements annotated as non nullable may be assigned (including via reference or output parameters) an expression of the following forms:

* another compatible array type (i.e implicitly convertible via array covariance and elements of non null references) where each element is effectively non nullable
* an array creation expression for a zero element array
* an array creation expression that takes an array initializer in which all elements are effectively non nullable
* or the following new form of the array creation expression where the array initializer allows a single effectively non nullable expression, the default element initializer, instead of an { variable-initializer-list }. For example

string![] words1 = new string![10] "hello"; // creates a 10 element array with all elements initialised to "hello"

Parameter arrays, denoted by the params modifier, are syntactic shorthand for passing an array creation expression as the argument. As such the if the element type of the parameter array is non nullable then the arguments passed for the parameter array must also be effectively non nullable.

### ### Conditional Assignment Operator

The conditional assignment operator, ?=, will assign a value to the left operand only if the right operand evaluates to a non null value (noting that this may be used with value types. It may be preferable to compare with the default value of the type rather than null). The conditional assignment expression returns a boolean indicating whether a non null value was assigned. When the conditional assignment operator is used as an expression statement it has no affect on whether the left operand is definitely assigned or effective nullability. The conditional assignment statement may be used in the condition expression of if, while and for statements. However the boolean expression containing the conditional assignment expression must not be a sub-expression within the conditional expression. The left operand of a conditional assignment is considered definitely assigned and adopts the effective nullability as if the conditional assignment was a normal assignment within the body of the containing if, for or while statement as well as in the for iterator statement expressions statement. The definite assignment status and effective nullability of the left operand of the conditional assignment returns to the status it was immediately before the if, for or while statement once the statement execution completes.

### ### Run-time Null Checks

When the value of an effectively non-nullable variable or expression is read, other than for an equals null check, will be guarded by a null check of the value. If the value is null a NullReferenceException will be thrown. This may be elided for member access since a null check is ready performed by the run-time. Likewise when value a value is written to a variable annotated as non nullable the new value will be checked that it is not null before writing the value and will otherwise throw a NullReferenceException. If a sequence of operations means that logically the same null check will be performed sequentially with no other operation in between then one of the null checks may be elided. A method invocation is considered an intervening operation.

Null checks will be added for reference and value parameters annotated as non nullable at the start of the containing method, however if the null check fails then an ArgumentNullException will be thrown instead of a NullReferenceException.

Static and instance variables annotated as non null references that do not have an initializer expression will no longer be considered initially or definitely assigned and therefore will start as effectively unusable. They will still be initialised to the default value (i.e. null) and that initialisation will not be subject for null write tests. All such static variables must be definitely assigned within the class static constructor otherwise it is a compiler error. All instance variables must be definitely assigned within all class instance constructors or the constructor must delegate construction to another instance constructor on the class that does. Therefore it is a compiler error to omit the default constructor for a class if it has instance variables that are not initially assigned. No inter-procedural analysis will be performed to determine if methods called from constructors definitely assign static or instance variables except for the constructor delegation already mentioned. The definite assignment must occur within the body of the relevant constructor. Static and instance fields are considered definitely assigned within methods and properties of the class. Since it is legal to invoke member functions/properties of a class within the class constructor, including virtual members, it not possible to statically check that any reference annotated as non null or expression will not be null since definite assignment of static/instance fields is not guaranteed until construction is complete. This is why static compile time checks are supplemented with aforementioned runtime null checks to capture null refences as soon as possible to prevent null propagation throughout program execution.

It is legal for a reference static/instance variable annotated as non nullable to have an initializer expression that refers to another static/instance variable annotated as non nullable within the same class or another class. While intra-assembly analysis would be possible to prevent this, analysis is not possible across assembly boundaries (particularly pre-compiled assemblies). Therfore the following class is perfectly legal but will generate runtime exceptions.

class Loki

{

object! a = b; // initially assigned and therefore effectively non nullable

object! b = a; // initially assigned and therefore effectively non nullable

object! c; // initially unassigned and therefore effectively unusable

Loki()

{

c = Deceive(); // definitely assigned and therefore effectively non nullable

}

object! Deceive() { return c; } // c is considered effectively non nullable

}

Specifically the initial assignment to a will pick up the default value of b which will be null and this will fail the null check. This is because field initialization occurs in textual order.

Method return values may be annotated as non nulluble (as in the Loki.Deceive method above). The return type must be a reference type and all return statements within the method must return an effectively non nullable expression. The compiler will insert a null check against the return value prior to returning and throw a NullReferenceException if null. As the return value of such a method is also effectively non nullable a null read check will be performed by the caller before using the return value. For example in the Loki class the Deceive method would check if c is null before returning and the constructor will check the return value of Deceive for null before assignment to c and the assignment to c will check the expression being assigned is not null. As per the elision rules stated earlier one of the latter 2 null checks may be elided however the null check at the return statement within Deceive may not be elided as it is on the other side of a method invocation.

Properties of reference types annotated as non null behave as if they were methods with the following signatures where T is the reference type of the property (indexers behave in an similar fashion):

void Set(T! value)

T! Get()

Reference variable nullability annotations will be encoded in assembly metadata as attributes on parameters, static/instance variables, method return values and properties. A new enum and attribute will be defined as follow (the names I have used below may violate some framework design guidelines but I am not sure, but these can be easily changed and are just to illustrate the concept:

public enum Nullability { Nullable = 0, NonNullable = 1 };

public class NullabilityAttribute: Attribute

{

public Nullability Nullability { get; set; }

public Nullability ArrayDepth1Nullability { get; set; }

public Nullability ArrayDepth2Nullability { get; set; }

public Nullability ArrayDepth3Nullability { get; set; }

public Nullability ArrayDepth4Nullability { get; set; }

// etc... need to decide what depth we will support arrays of arrays

}

The following pairs of examples show equivalent declarations of variables one involving annotating as non nullable and the other using attributes

object! o;

[Nullability(Nullability=Nullability.NonNullable)]object o;

object![] o;

[Nullability(Nullability=Nullability.NonNullable, ArrayDepth1Nullability=Nullability.Nullable)]object o;

object[]! o;

[Nullability(Nullability=Nullability.Nullable, ArrayDepth1Nullability=Nullability.NonNullable)]object o;

object![]! o;

[Nullability(Nullability=Nullability.NonNullable, ArrayDepth1Nullability=Nullability.NonNullable)]object o;

object![][]![]![] o;

[Nullability(Nullability=Nullability.NonNullable, ArrayDepth2Nullability=Nullability.NonNullable, ArrayDepth3Nullability=Nullability.NonNullable)]object o;

While the following declarations are also equivalent the compiler should not encode nullability in this way

object o;

[Nullability(Nullability=Nullability.Nullable)]object o; // or

[Nullability()]object o;

### ### Versioning considerations

For virtual/abstract members of a class the nullability of parameters and return types is taken from the declaration of the virtual/abstract member. It is an error to override such a member and not annotate the nullability of parameters and return types identically to the original virtual/abstract declaration. Since intermediate base classes that may have overriden such members may not have understood or enforced nullability attributes/rules (coming from another language or a version of C# unaware of non nullability) they will most likely have not included the nullability attribute. Deriving classes should always look at the original virtual/abstract member declaration to determine nullability. This introduces a considerable source of breaking change to existing code bases where existing type virtual/abstract members are updated to be annotated as non nullable. Existing code containing classes that inherit from such base classes may no longer compile without modifications and pre-compiled assemblies will not enforce non nullability and runtime null checks will only be enforced when crossing assembly boundaries. Therefore the guidelines would be to avoid updating existing published type members and add this to new members or new types as needed or in the body of existing members only.

## ## C# 2.0

### ### Generics

The concept of effective nullability will be extended to apply to generic type parameters as well as variables. Each type parameter will be assigned an effective nullability separate from any variables of that type. Generic type parameters will erase the effective nullability of supplied type arguments of reference type. That is to say that the type parameter is effectively nullable. Type parameters can not ever be effectively unusable. Consider the following identity function.

T IdentityA (T t) { return t; }

This can apply to value types and both effectively nullable and non nullable expressions. So the following call to IdentityA will have a return type of nullable string.

String s = IndentityA("hello world");

While type parameters can never be effectively unusable, variables of those type parameter types may remain effectively unusable until it is definitely assigned and then adopts the effective nullability of the type parameter. For example if the IdentityA function was re-written as follows:

T IdentityA (T t)

{

T result; // T is effectively nullable but result is effectively unusable

result = t; // result is now effectively nullable

return result;

}

Each use of a generic type parameter, excluding the type parameter declaration, may be annotated as non nullable. For example:

T! IdentityB(T! t) { return t; }

T will still accept any type argument however the method argument t must be effectively non nullable (including non nullable value types). Note that the return type is also annotated as non nullable as well. Therefore the following call takes an effectively non nullable string and returns an effectively non nullable string:

string! s = IndentityB("hello world");

This is insufficient for a large number of generic types or methods that need to support nullable and non nullable references as it would require duplication of many types and methods for nullable and non nullable variations (similar to const and non const variations in C++). To solve this problem a new preserve nullability annotation, !?, may be added to generic type parameters and such a type parameter will be known as a preserving type parameter. For example:

T IdentityC<T!?>(T t) { return t; }

The concept of effective nullability is extended with a fourth option of effectively preserved. Within a generic type or method that declares a preserving type parameter the type parameter is effectively preserved. Any variable or expression that is of a type that is effectively preserved follows the same rules as effectively non nullable variables/expressions except the null checks associated with non nullable references are omitted.

Preserving type parameters will accept type arguments of any effective nullability, except effectively unusable,and the type parameter will be assigned the same effective nullability. Therefore given the above definition of IdentityC:

string! s1 = IdentityC("hello world"); // ok, T is inferred to be an effectively non nullable string

string s2 = null;

string! s3 = IdentityC(s2) // error, T is inferred to be effectively nullable

U IdentityD<U>(U u) { return IdentityC(u); } // U is effectively nullable and therefore so is T within IdentityC

U! IdentityE<U>(U! u) { return IdentityC(u); } // u is annotated as non nullable and therefore T is effectively non nullable within IdentityC

U IdentityF<U!?>(U u) { return IdentityC(u); } // U is effectively preserved and therefore so is T within IdentityC

The previous rules for defining the relationship between type parameters effective nullability and variable of those types are extended such that a local variable of type T, where T is a preserved type parameter, remains effectively unusable until it is definitely assigned and may only be assigned an expression with an effective nullability that is as strong or stronger then the effective nullability of the type parameter T. Once the local variable is definitely assigned it takes on the effective nullability of the type parameter T.

Preserved type parameters will be encoded in metadata via the inclusion of a new PreserveNullability attribute applied to the type parameter. For example IdentityC would be encoded as:

T IdentityC<[PreserveNullability]T>(T t) { return t; }

The lack of runtime null checking of variables and expressions whose type is a preserved type parameter is a significant loss in the overall enforcement of non nullability. (While I have considered various options to reinstate these checks they all effectively involve custom calling conventions on top of the CLR which is undesireable). If the CLR was modified to understand preserved type parameters it could conditionally perform the null checks at runtime. However it seems unlikely a new version of the CLR would be created just to support this feature. Further implementing these checks in the CLR at a stage after compiled code using preserved type parameters already exists should be only done with care as such code could internally violate nullability requirements yet satisfy the published contracts regarding nullability. This would likely lead to NullReferenceExceptions being thrown in compiled code that did not previously throw.

The default operator may not be used with a type that is effectively preserved or non nullable. That is to say the default operator is effectively nullable.

The nullable modifier (?) may be applied to usege of a preserved type parameter outside of it's declaration. Unlike nullable value types this does not change the type but rather changes the effective nullability of the usage to be effectively nullable. For example

T? Filter<T!?>(T t) where T: class, IFoo

{

return t.Bar ? t : null;

}

It is an error to use the nullable modifier with a preserved type parameter unless the reference type constraint (i.e. class) or inheritence constraint from a reference type has been applied to the type parameter.

The nullable modifier applied to the use of a preserverved type parameter is encoded in metadata using attributes and reused the ```NullabilityAttribute```. For example the above ```Filter``` method would be encoded as follows:

[return: Nullability(Nullability=Nullability.Nullable)]

T Filter<[PreserveNullability]T>(T t) where T: class, IFoo

If a preserved type parameter A is used as a constraint for another type parameter B then B must also be annotated as preserving nullability. For example:

A IdentityG<A!?,B!?>(B b) where B : A { return b; }

Further the constraint also constrains the permitted effective nullability of type argument B to be stronger than or equal to the effective nullability of type argument A. Effectively non nullable is stronger than effectively preserved which is stronger than effectively nullable. For example

Animal! a = IdentityG<Animal!,Cat!>(new Cat()); // ok

IdentityD<Animal!,Cat>(new Cat()); // error

IdentityD<Animal,Cat!>(new Cat()); // ok

IdentityD<Animal,Cat>(new Cat()); // ok

IdentityD<T,Cat!>(new Cat()); // ok if T!?

IdentityD<T,Cat>(new Cat()); // error if T!?

When inferring the effective nullability of a preserved type parameter ```T``` for a method invocation where the type parameter is used with multiple parameters then the effective nullability will be the weakest effective nullability of the argument supplied for parameters of type ```T```. For example consider the following method and invocations:

T Bar<T!?>(T x, T y) { ... }

Bar("hello","world"); // both arguments are effectively non-nullable and therefore so is return value

string s = null;

Bar("goodbye",s); // s has the weaker effective nullability (nullable) and therefore the return value is effectively nullable

A constructed type with at least one preserving type parameters can be cast between constructed types of the same generic type with the same type arguments but with different effective nullability. For example:

class Wrapper<T!?>

{

public Wrapper(T t) { value = t; }

public T value;

}

//Then the following is legal:

(Wrapper<object!>)new Wrapper(null);

This is because the cast could be performed via casting to object first. e.g.

(Wrapper<object!>)(object)new Wrapper(null);

However in both of those cases an attempt to read the value instance variable would generate a NullReferenceException. If the CLR is updated to understand preserved type parameters and enforce null checks then the above casts should generate an InvalidCastException and the first cast example should be disallowed by the compiler.

### ### Effective Nullability of Type Parameters for Virtual Members

Virtual members declared in a generic type that are inherited or overridden in an inherited type determine the effective nullability of any type parameters used in the signature or body of the member by considering any erasure or preservation of effective nullability of type parameters via supplied type arguments to types in the inheritence hierarchy between the type declaring the virtual member and the type that overrides it. For example:

// T is always effectively nullable for members declared in A

class A<T>

{

virtual T IdentityA(T t) { return t; }

}

// T is effectively preserved on new members in B and for members inherited from A, T is effectively nullable

class B<T!?> : A<T>

{

// ok since T is effectively nullable

override T IdentityA(T t) { return default(T); }

// error since T is effectively preserved but the return value of IdentityA is effectively nullable

virtual T IdentityB(T t) { return IdentityA(t); }

}

class C<T>: B<T>

{

// ok since T is effectively nullable and therefore for members declared in B, T is also effectively nullable

override T IdentityB(T t) { return IdentityA(t); }

}

Type Parameters to generic interface or delegate declarations are taken to be implicitly preserved and do require the annotated as preserving nullability. So for example IEnumerable<T> is implicitly taken to mean IEnumerable<T!?> and likewise Func<T> is taken to mean Func<T!?>

As a further example consider ```List<T>``` which implements ```IEnumerable<T>```. The type parameter T is effectively nullable for ```List<T>``` and therefore supplied as a type argument to ```IEnumerable<T>``` (where T is implicitly preserved due to IEnumerable being an interface) then T is effectively nullable for members of IEnumerable which matches the existing semantics for using ```IEnumerable<T>``` from ```List<T>```. Consider a new type declared as follows:

class Producer<T> : IEnumerable<T!>

Within the Producer class the members of IEnumerable use of type parameter T are considered effectively non nullable.

### ### Miscellaneous changes

A coalescing operator expression is effectively non nullable if either operand is effectively non nullable. Method groups, relating to static members, and anonymous methods subject to implicit conversion to a compatible delegate type are also effectively non nullable. A similar rule applies to method groups relating to an instance expression but only where the instance expression is also effectively non nullable.

Anonymous method and method group conversions will apply the new rules for implicit conversions for identifying the valid candidate method or types to create a valid conversion. Anonymous methods that do not include an anonymous method signature may not be converted to a delegate type that has any parameters annotated as non nullable or parameters with a type that is a preserved type parameter.

The conditional assignment operator may be used with nullable value types.

## ## C# 3.0

Implicitly typed local variables have the same effective nullability as the initializer expression for the variable. An implicitly typed iteration variable in a foreach statement has the same effective nullability as the element type of the collection being iterated. (Alternatively we could permit ! or ? on the var declaration to make the effective nullability of the implicitly typed variable explicit).

Implicitly typed lambda parameters obtain their effective nullability from the effective nullability of the parameters of the delegate type the lambda is converted to.

The effective nullability of properties on anonymous types is the same as that of the initializer expression for that property. (Alternatively a property could have ! or ? appended to it to make the nullability explicit. e.g. ```new { Name? = "John" }```)

The effective nullability of the element type of an implicitly typed array is the weakest effective nullability of any of the elements of the initializer expression.

## ## C# 4.0

As dynamic is a reference type and is equivalent to object then variables of dynamic type may be annotated as non nullable. As with other reference types a nullable reference may be cast to a non nullable dynamic reference.

Where the target variable of an implicit dynamic conversion is annotated as non nullable or it's type is a preserved type parameter then the dynamic expression to be converted must be effective non-nullable or preserved respectively.

Variance conversions for interfaces and delegates work as expected with the existing implicit reference conversions for effective nullability conversions.

Optional Parameters annotated as non nullable or with a type that is a preserved type parameter may only be assigned an effectively non nullable or preserved expression respectively.

### ## C# 5.0

The addition of async/await does not appear to require any special treatment. It would be advisable to update ```Task``` to ```Task<!?>``` so that async methods and awaitable expressions can return effectively non null references. However this is a library change only and the existing return value inference rules should work with anonymous async lambdas for example.

### ## C# 6.0

The nameof operator currently can not be used with nullable types (e.g ```nameof(Point?)```) and therefore it is an error to apply the nameof operator to a type annotated as non nullable (e.g. ```nameof(String!)```).

When the result of a member access would normally be effectively preserved or stronger and the member is accessed via a null-conditional operator then the result becomes effectively nullable.

The null-conditional operators maybe used with effectively non nullable or preserved expressions however the expression is implicitly converted to be effectively nullable first and therefore will not apply null checks when reading the value. For example:

string! s = "hello world";

string s2;

s2 = s?.ToString();// is equivalent to the following

s2 = ((string)s)?.ToString();// is equivalent to the preceding