Required Members

Article • 03/17/2023

> **Note**
>
> This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent **language design meeting (LDM) notes**. Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the **specifications**.

Summary

This proposal adds a way of specifying that a property or field is required to be set during object initialization, forcing the instance creator to provide an initial value for the member in an object initializer at the creation site.

Motivation

Object hierarchies today require a lot of boilerplate to carry data across all levels of the hierarchy. Let's look at a simple hierarchy involving a `Person` as might be defined in C# 8:

C#

```
class Person
{
    public string FirstName { get; }
    public string MiddleName { get; }
    public string LastName { get; }

    public Person(string firstName, string lastName, string? middle-
Name = null)
    {
        FirstName = firstName;
        LastName = lastName;
        MiddleName = middleName ?? string.Empty;
    }
}
```

```
class Student : Person
{
    public int ID { get; }
    public Student(int id, string firstName, string lastName, string?
middleName = null)
        : base(firstName, lastName, middleName)
    {
        ID = id;
    }
}
```

There's lots of repetition going on here:

1. At the root of the hierarchy, the type of each property had to be repeated twice, and the name had to be repeated four times.
2. At the derived level, the type of each inherited property had to be repeated once, and the name had to be repeated twice.

This is a simple hierarchy with 3 properties and 1 level of inheritance, but many real-world examples of these types of hierarchies go many levels deeper, accumulating larger and larger numbers of properties to pass along as they do so. Roslyn is one such codebase, for example, in the various tree types that make our CSTs and ASTs. This nesting is tedious enough that we have code generators to generate the constructors and definitions of these types, and many customers take similar approaches to the problem. C# 9 introduces records, which for some scenarios can make this better:

C#

```
record Person(string FirstName, string LastName, string MiddleName =
"");
record Student(int ID, string FirstName, string LastName, string
MiddleName = "") : Person(FirstName, LastName, MiddleName);
```

`record`s eliminate the first source of duplication, but the second source of duplication remains unchanged: unfortunately, this is the source of duplication that grows as the hierarchy grows, and is the most painful part of the duplication to fix up after making a change in the hierarchy as it required chasing the hierarchy through all of its locations, possibly even across projects and potentially breaking consumers.

As a workaround to avoid this duplication, we have long seen consumers embracing object initializers as a way of avoiding writing constructors. Prior to C# 9, however, this had 2 major downsides:

1. The object hierarchy has to be fully mutable, with `set` accessors on every property.

2. There is no way to ensure that every instantation of an object from the graph sets every member.

C# 9 again addressed the first issue here, by introducing the `init` accessor: with it, these properties can be set on object creation/initialization, but not subsequently. However, we again still have the second issue: properties in C# have been optional since C# 1.0. Nullable reference types, introduced in C# 8.0, addressed part of this issue: if a constructor does not initialize a non-nullable reference-type property, then the user is warned about it. However, this doesn't solve the problem: the user here wants to not repeat large parts of their type in the constructor, they want to pass the *requirement* to set properties on to their consumers. It also doesn't provide any warnings about `ID` from `Student`, as that is a value type. These scenarios are extremely common in database model ORMs, such as EF Core, which need to have a public parameterless constructor but then drive nullability of the rows based on the nullability of the properties.

This proposal seeks to address these concerns by introducing a new feature to C#: required members. Required members will be required to be initialized by consumers, rather than by the type author, with various customizations to allow flexibility for multiple constructors and other scenarios.

Detailed Design

`class`, `struct`, and `record` types gain the ability to declare a *required_member_list*. This list is the list of all the properties and fields of a type that are considered *required*, and must be initialized during the construction and initialization of an instance of the type. Types inherit these lists from their base types automatically, providing a seamless experience that removes boilerplate and repetitive code.

`required` modifier

We add `'required'` to the list of modifiers in *field_modifier* and *property_modifier*. The *required_member_list* of a type is composed of all the members that have had `required` applied to them. Thus, the `Person` type from earlier now looks like this:

C#

```
public class Person
{
    // The default constructor requires that FirstName and LastName
be set at construction time
    public required string FirstName { get; init; }
```

```
    public string MiddleName { get; init; } = "";
    public required string LastName { get; init; }
}
```

All constructors on a type that has a *required_member_list* automatically advertise a *contract* that consumers of the type must initialize all of the properties in the list. It is an error for a constructor to advertise a contract that requires a member that is not at least as accessible as the constructor itself. For example:

C#

```
public class C
{
    public required int Prop { get; protected init; }

    // Advertises that Prop is required. This is fine, because the
constructor is just as accessible as the property initer.
    protected C() {}

    // Error: ctor C(object) is more accessible than required proper-
ty Prop.init.
    public C(object otherArg) {}
}
```

`required` is only valid in `class`, `struct`, and `record` types. It is not valid in `interface` types. `required` cannot be combined with the following modifiers:

- `fixed`
- `ref readonly`
- `ref`
- `const`
- `static`

`required` is not allowed to be applied to indexers.

The compiler will issue a warning when `Obsolete` is applied to a required member of a type and:

1. The type is not marked `Obsolete`, or
2. Any constructor not attributed with `SetsRequiredMembersAttribute` is not marked `Obsolete`.

`SetsRequiredMembersAttribute`

All constructors in a type with required members, or whose base type specifies required members, must have those members set by a consumer when that constructor is called. In order to exempt constructors from this requirement, a constructor can be attributed with `SetsRequiredMembersAttribute`, which removes these requirements. The constructor body is not validated to ensure that it definitely sets the required members of the type.

`SetsRequiredMembersAttribute` removes *all* requirements from a constructor, and those requirements are not checked for validity in any way. NB: this is the escape hatch if inheriting from a type with an invalid required members list is necessary: mark the constructor of that type with `SetsRequiredMembersAttribute`, and no errors will be reported.

If a constructor `C` chains to a `base` or `this` constructor that is attributed with `SetsRequiredMembersAttribute`, `C` must also be attributed with `SetsRequiredMembersAttribute`.

For record types, we will emit `SetsRequiredMembersAttribute` on the synthesized copy constructor of a record if the record type or any of its base types have required members.

NB: An earlier version of this proposal had a larger metalanguage around initialization, allowing adding and removing individual required members from a constructor, as well as validation that the constructor was setting all required members. This was deemed too complex for the initial release, and removed. We can look at adding more complex contracts and modifications as a later feature.

Enforcement

For every constructor `Ci` in type `T` with required members `R`, consumers calling `Ci` must do one of:

- Set all members of `R` in an *object_initializer* on the *object_creation_expression*,
- Or set all members of `R` via the *named_argument_list* section of an *attribute_target*.

unless `Ci` is attributed with `SetsRequiredMembers`.

If the current context does not permit an *object_initializer* or is not an *attribute_target*, and `Ci` is not attributed with `SetsRequiredMembers`, then it is an error to call `Ci`.

`new()` constraint

A type with a parameterless constructor that advertises a *contract* is not allowed to be substituted for a type parameter constrained to `new()`, as there is no way for the generic instantiation to ensure that the requirements are satisfied.

`struct` `default`s

Required members are not enforced on instances of `struct` types created with `default` or `default(StructType)`. They are enforced for `struct` instances created with `new StructType()`, even when `StructType` has no parameterless constructor and the default struct constructor is used.

Accessibility

It is an error to mark a member required if the member cannot be set in any context where the containing type is visible.

- If the member is a field, it cannot be `readonly`.
- If the member is a property, it must have a setter or initer at least as accessible as the member's containing type.

This means the following cases are not allowed:

C#

```
interface I
{
    int Prop1 { get; }
}
public class Base
{
    public virtual int Prop2 { get; set; }

    protected required int _field; // Error: _field is not at least
as visible as Base. Open question below about the protected construc-
tor scenario

    public required readonly int _field2; // Error: required fields
cannot be readonly
    protected Base() { }

    protected class Inner
    {
        protected required int PropInner { get; set; } // Error:
PropInner cannot be set inside Base or Derived
    }
}
public class Derived : Base, I
```

```
{
    required int I.Prop1 { get; } // Error: explicit interface imple-
mentions cannot be required as they cannot be set in an object ini-
tializer

    public required override int Prop2 { get; set; } // Error: this
property is hidden by Derived.Prop2 and cannot be set in an object
initializer
    public new int Prop2 { get; }

    public required int Prop3 { get; } // Error: Required member must
have a setter or initer

    public required int Prop4 { get; internal set; } // Error:
Required member setter must be at least as visible as the constructor
of Derived
}
```

It is an error to hide a `required` member, as that member can no longer be set by a consumer.

When overriding a `required` member, the `required` keyword must be included on the method signature. This is done so that if we ever want to allow unrequiring a property with an override in the future, we have design space to do so.

Overrides are allowed to mark a member `required` where it was not `required` in the base type. A member so-marked is added to the required members list of the derived type.

Types are allowed to override required virtual properties. This means that if the base virtual property has storage, and the derived type tries to access the base implementation of that property, they could observe uninitialized storage. NB: This is a general C# anti-pattern, and we don't think that this proposal should attempt to address it.

Effect on nullable analysis

Members that are marked `required` are not required to be initialized to a valid nullable state at the end of a constructor. All `required` members from this type and any base types are considered by nullable analysis to be default at the beginning of any constructor in that type, unless chaining to a `this` or `base` constructor that is attributed with `SetsRequiredMembersAttribute`.

Nullable analysis will warn about all `required` members from the current and base types that do not have a valid nullable state at the end of a constructor attributed with

`SetsRequiredMembersAttribute`.

C#

```
#nullable enable
public class Base
{
    public required string Prop1 { get; set; }

    public Base() {}

    [SetsRequiredMembers]
    public Base(int unused) { Prop1 = ""; }
}
public class Derived : Base
{
    public required string Prop2 { get; set; }

    [SetsRequiredMembers]
    public Derived() : base()
    {
    } // Warning: Prop1 and Prop2 are possibly null.

    [SetsRequiredMembers]
    public Derived(int unused) : base()
    {
        Prop1.ToString(); // Warning: possibly null dereference
        Prop2.ToString(); // Warning: possibly null dereference
    }

    [SetsRequiredMembers]
    public Derived(int unused, int unused2) : this()
    {
        Prop1.ToString(); // Ok
        Prop2.ToString(); // Ok
    }

    [SetsRequiredMembers]
    public Derived(int unused1, int unused2, int unused3) : base(un-
used1)
    {
        Prop1.ToString(); // Ok
        Prop2.ToString(); // Warning: possibly null dereference
    }
}
```

Metadata Representation

The following 2 attributes are known to the C# compiler and required for this feature to function:

C#

```
namespace System.Runtime.CompilerServices
{
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct
| AttributeTargets.Field | AttributeTargets.Property, AllowMultiple =
false, Inherited = false)]
    public sealed class RequiredMemberAttribute : Attribute
    {
        public RequiredMemberAttribute() {}
    }
}

namespace System.Diagnostics.CodeAnalysis
{
    [AttributeUsage(AttributeTargets.Constructor, AllowMultiple =
false, Inherited = false)]
    public sealed class SetsRequiredMembersAttribute : Attribute
    {
        public SetsRequiredMembersAttribute() {}
    }
}
```

It is an error to manually apply `RequiredMemberAttribute` to a type.

Any member that is marked `required` has a `RequiredMemberAttribute` applied to it. In addition, any type that defines such members is marked with `RequiredMemberAttribute`, as a marker to indicate that there are required members in this type. Note that if type `B` derives from `A`, and `A` defines `required` members but `B` does not add any new or override any existing `required` members, `B` will not be marked with a `RequiredMemberAttribute`. To fully determine whether there are any required members in `B`, checking the full inheritance hierarchy is necessary.

Any constructor in a type with `required` members that does not have `SetsRequiredMembersAttribute` applied to it is marked with two attributes:

1. `System.Runtime.CompilerServices.CompilerFeatureRequiredAttribute` with the feature name `"RequiredMembers"`.
2. `System.ObsoleteAttribute` with the string `"Types with required members are not supported in this version of your compiler"`, and the attribute is marked as an error, to prevent any older compilers from using these constructors.

We don't use a `modreq` here because it is a goal to maintain binary compat: if the last `required` property was removed from a type, the compiler would no longer synthesize this `modreq`, which is a binary-breaking change and all consumers would need to be recompiled. A compiler that understands `required` members will ignore this obsolete

attribute. Note that members can come from base types as well: even if there are no new `required` members in the current type, if any base type has `required` members, this `Obsolete` attribute will be generated. If the constructor already has an `Obsolete` attribute, no additional `Obsolete` attribute will be generated.

We use both `ObsoleteAttribute` and `CompilerFeatureRequiredAttribute` because the latter is new this release, and older compilers don't understand it. In the future, we may be able to drop the `ObsoleteAttribute` and/or not use it to protect new features, but for now we need both for full protection.

To build the full list of `required` members `R` for a given type `T`, including all base types, the following algorithm is run:

1. For every `Tb`, starting with `T` and working through the base type chain until `object` is reached.
2. If `Tb` is marked with `RequiredMemberAttribute`, then all members of `Tb` marked with `RequiredMemberAttribute` are gathered into `Rb`
   a. For every `Ri` in `Rb`, if `Ri` is overridden by any member of `R`, it is skipped.
   b. Otherwise, if any `Ri` is hidden by a member of `R`, then the lookup of required members fails and no further steps are taken. Calling any constructor of `T` not attributed with `SetsRequiredMembers` issues an error.
   c. Otherwise, `Ri` is added to `R`.

Open Questions

Nested member initializers

What will the enforcement mechanisms for nested member initializers be? Will they be disallowed entirely?

C#

```
class Range
{
    public required Location Start { get; init; }
    public required Location End { get; init; }
}

class Location
{
    public required int Column { get; init; }
    public required int Line { get; init; }
}
```

```
_ = new Range { Start = { Column = 0, Line = 0 }, End = { Column = 1,
Line = 0 } } // Would this be allowed if Location is a struct type?
_ = new Range { Start = new Location { Column = 0, Line = 0 }, End =
new Location { Column = 1, Line = 0 } } // Or would this form be nec-
essary instead?
```

Discussed Questions

Level of enforcement for `init` clauses

Do we strictly enforce that members specified in a `init` clause without an initializer must initialize all members? It seems likely that we do, otherwise we create an easy pit-of-failure. However, we also run the risk of reintroducing the same problems we solved with `MemberNotNull` in C# 9. If we want to strictly enforce this, we will likely need a way for a helper method to indicate that it sets a member. Some possible syntaxes we've discussed for this:

- Allow `init` methods. These methods are only allowed to be called from a constructor or from another `init` method, and can access `this` as if it's in the constructor (ie, set `readonly` and `init` fields/properties). This can be combined with `init` clauses on such methods. A `init` clause would be considered satisfied if the member in the clause is definitely assigned in the body of the method/constructor. Calling a method with a `init` clause that includes a member counts as assigning to that member. If we do decided that this is a route we want to pursue, now or in the future, it seems likely that we should not use `init` as the keyword for the init clause on a constructor, as that would be confusing.
- Allow the `!` operator to suppress the warning/error explicitly. If initializing a member in a complicated way (such as in a shared method), the user can add a `!` to the init clause to indicate the compiler should not check for initialization.

Conclusion: After discussion we like the idea of the `!` operator. It allows the user to be intentional about more complicated scenarios while also not creating a large design hole around init methods and annotating every method as setting members X or Y. `!` was chosen because we already use it for suppressing nullable warnings, and using it to tell the compiler "I'm smarter than you" in another place is a natural extension of the syntax form.

Required interface members

This proposal does not allow interfaces to mark members as required. This protects us from having to figure out complex scenarios around `new()` and interface constraints in generics right now, and is directly related to both factories and generic construction. In order to ensure that we have design space in this area, we forbid `required` in interfaces, and forbid types with *required_member_lists* from being substituted for type parameters constrained to `new()`. When we want to take a broader look at generic construction scenarios with factories, we can revisit this issue.

Syntax questions

- Is `init` the right word? `init` as a postfix modifier on the constructor might interfere if we ever want to reuse it for factories and also enable `init` methods with a prefix modifier. Other possibilities:
  - `set`
- Is `required` the right modifier for specifying that all members are initialized? Others suggested:
  - `default`
  - `all`
  - With a ! to indicate complex logic
- Should we require a separator between the `base`/`this` and the `init`?
  - `:` separator
  - ',' separator
- Is `required` the right modifier? Other alternatives that have been suggested:
  - `req`
  - `require`
  - `mustinit`
  - `must`
  - `explicit`

Conclusion: We have removed the `init` constructor clause for now, and are proceeding with `required` as the property modifier.

Init clause restrictions

Should we allow access to `this` in the init clause? If we want the assignment in `init` to be a shorthand for assigning the member in the constructor itself, it seems like we should.

Additionally, does it create a new scope, like `base()` does, or does it share the same scope as the method body? This is particularly important for things like local functions, which the init clause may want to access, or for name shadowing, if an init expression introduces a variable via `out` parameter.

Conclusion: `init` clause has been removed.

Accessibility requirements and `init`

In versions of this proposal with the `init` clause, we talked about being able to have the following scenario:

C#

```
public class Base
{
    protected required int _field;

    protected Base() {} // Contract required that _field is set
}
public class Derived : Base
{
    public Derived() : init(_field = 1) // Contract is fulfilled and
_field is removed from the required members list
    {
    }
}
```

However, we have removed the `init` clause from the proposal at this point, so we need to decide whether to allow this scenario in a limited fashion. The options we have are:

1. Disallow the scenario. This is the most conservative approach, and the rules in the Accessibility are currently written with this assumption in mind. The rule is that any member that is required must be at least as visible as its containing type.
2. Require that all constructors are either:
   a. No more visible than the least-visible required member.
   b. Have the `SetsRequiredMembersAttribute` applied to the constructor. These would ensure that anyone who can see a constructor can either set all the things it exports, or there is nothing to set. This could be useful for types that are only ever created via static `Create` methods or similar builders, but the utility seems overall limited.
3. Readd a way to remove specific parts of the contract to the proposal, as discussed in LDM previously.

Conclusion: Option 1, all required members must be at least as visible as their containing type.

Override rules

The current spec says that the `required` keyword needs to be copied over and that overrides can make a member *more* required, but not less. Is that what we want to do? Allowing removal of requirements needs more contract modification abilities than we are currently proposing.

Conclusion: Adding `required` on override is allowed. If the overridden member is `required`, the overridding member must also be `required`.

Alternative metadata representation

We could also take a different approach to metadata representation, taking a page from extension methods. We could put a `RequiredMemberAttribute` on the type to indicate that the type contains required members, and then put a `RequiredMemberAttribute` on each member that is required. This would simplify the lookup sequence (no need to do member lookup, just look for members with the attribute).

Conclusion: Alternative approved.

Metadata Representation

The Metadata Representation needs to be approved. We additionally need to decide whether these attributes should be included in the BCL.

1. For `RequiredMemberAttribute`, this attribute is more akin to the general embedded attributes we use for nullable/nint/tuple member names, and will not be manually applied by the user in C#. It's possible that other languages might want to manually apply this attribute, however.
2. `SetsRequiredMembersAttribute`, on the other hand, is directly used by consumers, and thus should likely be in the BCL.

If we go with the alternative representation in the previous section, that might change the calculus on `RequiredMemberAttribute`: instead of being similar to the general embedded attributes for `nint`/nullable/tuple member names, it's closer to `System.Runtime.CompilerServices.ExtensionAttribute`, which has been in the framework since extension methods shipped.

Conclusion: We will put both attributes in the BCL.

Warning vs Error

Should not setting a required member be a warning or an error? It is certainly possible to trick the system, via `Activator.CreateInstance(typeof(C))` or similar, which means we may not be able to fully guarantee all properties are always set. We also allow suppression of the diagnostics at the constructor-site by using the `!`, which we generally do not allow for errors. However, the feature is similar to readonly fields or init properties, in that we hard error if users attempt to set such a member after initialization, but they can be circumvented by reflection.

Conclusion: Errors.