Attributes for null-state static analysis interpreted by the C# compiler

Article • 05/05/2023

In a nullable enabled context, the compiler performs static analysis of code to determine the *null-state* of all reference type variables:

- *not-null*: Static analysis determines that a variable has a non-null value.
- *maybe-null*: Static analysis can't determine that a variable is assigned a non-null value.

These states enable the compiler to provide warnings when you may dereference a null value, throwing a System.NullReferenceException. These attributes provide the compiler with semantic information about the *null-state* of arguments, return values, and object members based on the state of arguments and return values. The compiler provides more accurate warnings when your APIs have been properly annotated with this semantic information.

This article provides a brief description of each of the nullable reference type attributes and how to use them.

Let's start with an example. Imagine your library has the following API to retrieve a resource string. This method was originally compiled in a *nullable oblivious* context:

C#

```
bool TryGetMessage(string key, out string message)
{
    if (_messageMap.ContainsKey(key))
        message = _messageMap[key];
    else
        message = null;
    return message != null;
}
```

The preceding example follows the familiar `Try*` pattern in .NET. There are two reference parameters for this API: the `key` and the `message`. This API has the following rules relating to the *null-state* of these parameters:

- Callers shouldn't pass `null` as the argument for `key`.
- Callers can pass a variable whose value is `null` as the argument for `message`.

- If the `TryGetMessage` method returns `true`, the value of `message` isn't null. If the return value is `false,` the value of `message` is null.

The rule for `key` can be expressed succinctly: `key` should be a non-nullable reference type. The `message` parameter is more complex. It allows a variable that is `null` as the argument, but guarantees, on success, that the `out` argument isn't `null`. For these scenarios, you need a richer vocabulary to describe the expectations. The `NotNullWhen` attribute, described below describes the *null-state* for the argument used for the `message` parameter.

> **Note**
>
> Adding these attributes gives the compiler more information about the rules for your API. When calling code is compiled in a nullable enabled context, the compiler will warn callers when they violate those rules. These attributes don't enable more checks on your implementation.

Attribute	Category	Meaning
AllowNull	Precondition	A non-nullable parameter, field, or property may be null.
DisallowNull	Precondition	A nullable parameter, field, or property should never be null.
MaybeNull	Postcondition	A non-nullable parameter, field, property, or return value may be null.
NotNull	Postcondition	A nullable parameter, field, property, or return value will never be null.
MaybeNullWhen	Conditional postcondition	A non-nullable argument may be null when the method returns the specified `bool` value.
NotNullWhen	Conditional postcondition	A nullable argument won't be null when the method returns the specified `bool` value.
NotNullIfNotNull	Conditional postcondition	A return value, property, or argument isn't null if the argument for the specified parameter isn't null.
MemberNotNull	Method and property helper methods	The listed member won't be null when the method returns.
MemberNotNullWhen	Method and property helper methods	The listed member won't be null when the method returns the specified `bool` value.

Attribute	Category	Meaning
DoesNotReturn	Unreachable code	A method or property never returns. In other words, it always throws an exception.
DoesNotReturnIf	Unreachable code	This method or property never returns if the associated `bool` parameter has the specified value.

The preceding descriptions are a quick reference to what each attribute does. The following sections describe the behavior and meaning of these attributes more thoroughly.

Preconditions: `AllowNull` and `DisallowNull`

Consider a read/write property that never returns `null` because it has a reasonable default value. Callers pass `null` to the set accessor when setting it to that default value. For example, consider a messaging system that asks for a screen name in a chat room. If none is provided, the system generates a random name:

C#

```
public string ScreenName
{
    get => _screenName;
    set => _screenName = value ?? GenerateRandomScreenName();
}
private string _screenName;
```

When you compile the preceding code in a nullable oblivious context, everything is fine. Once you enable nullable reference types, the `ScreenName` property becomes a non-nullable reference. That's correct for the `get` accessor: it never returns `null`. Callers don't need to check the returned property for `null`. But now setting the property to `null` generates a warning. To support this type of code, you add the System.Diagnostics.CodeAnalysis.AllowNullAttribute attribute to the property, as shown in the following code:

C#

```
[AllowNull]
public string ScreenName
{
    get => _screenName;
    set => _screenName = value ?? GenerateRandomScreenName();
```

```
}
private string _screenName = GenerateRandomScreenName();
```

You may need to add a `using` directive for System.Diagnostics.CodeAnalysis to use this and other attributes discussed in this article. The attribute is applied to the property, not the `set` accessor. The `AllowNull` attribute specifies *pre-conditions*, and only applies to arguments. The `get` accessor has a return value, but no parameters. Therefore, the `AllowNull` attribute only applies to the `set` accessor.

The preceding example demonstrates what to look for when adding the `AllowNull` attribute on an argument:

1. The general contract for that variable is that it shouldn't be `null`, so you want a non-nullable reference type.
2. There are scenarios for a caller to pass `null` as the argument, though they aren't the most common usage.

Most often you'll need this attribute for properties, or `in`, `out`, and `ref` arguments. The `AllowNull` attribute is the best choice when a variable is typically non-null, but you need to allow `null` as a precondition.

Contrast that with scenarios for using `DisallowNull`: You use this attribute to specify that an argument of a nullable reference type shouldn't be `null`. Consider a property where `null` is the default value, but clients can only set it to a non-null value. Consider the following code:

C#

```
public string ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new ArgumentNullException(name-
of(value), "Cannot set to null");
}
string _comment;
```

The preceding code is the best way to express your design that the `ReviewComment` could be `null`, but can't be set to `null`. Once this code is nullable aware, you can express this concept more clearly to callers using the System.Diagnostics.CodeAnalysis.DisallowNullAttribute:

C#

```
[DisallowNull]
public string? ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new ArgumentNullException(name-
of(value), "Cannot set to null");
}
string? _comment;
```

In a nullable context, the `ReviewComment` `get` accessor could return the default value of `null`. The compiler warns that it must be checked before access. Furthermore, it warns callers that, even though it could be `null`, callers shouldn't explicitly set it to `null`. The `DisallowNull` attribute also specifies a *pre-condition*, it doesn't affect the `get` accessor. You use the `DisallowNull` attribute when you observe these characteristics about:

1. The variable could be `null` in core scenarios, often when first instantiated.
2. The variable shouldn't be explicitly set to `null`.

These situations are common in code that was originally *null oblivious*. It may be that object properties are set in two distinct initialization operations. It may be that some properties are set only after some asynchronous work has completed.

The `AllowNull` and `DisallowNull` attributes enable you to specify that preconditions on variables may not match the nullable annotations on those variables. These provide more detail about the characteristics of your API. This additional information helps callers use your API correctly. Remember you specify preconditions using the following attributes:

- AllowNull: A non-nullable argument may be null.
- DisallowNull: A nullable argument should never be null.

Postconditions: `MaybeNull` and `NotNull`

Suppose you have a method with the following signature:

C#

```
public Customer FindCustomer(string lastName, string firstName)
```

You've likely written a method like this to return `null` when the name sought wasn't found. The `null` clearly indicates that the record wasn't found. In this example, you'd

likely change the return type from `Customer` to `Customer?`. Declaring the return value as a nullable reference type specifies the intent of this API clearly:

C#

```
public Customer? FindCustomer(string lastName, string firstName)
```

For reasons covered under Generics nullability that technique may not produce the static analysis that matches your API. You may have a generic method that follows a similar pattern:

C#

```
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)
```

The method returns `null` when the sought item isn't found. You can clarify that the method returns `null` when an item isn't found by adding the `MaybeNull` annotation to the method return:

C#

```
[return: MaybeNull]
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)
```

The preceding code informs callers that the return value *may* actually be null. It also informs the compiler that the method may return a `null` expression even though the type is non-nullable. When you have a generic method that returns an instance of its type parameter, `T`, you can express that it never returns `null` by using the `NotNull` attribute.

You can also specify that a return value or an argument isn't null even though the type is a nullable reference type. The following method is a helper method that throws if its first argument is `null`:

C#

```
public static void ThrowWhenNull(object value, string valueExpression
= "")
{
    if (value is null) throw new ArgumentNullException(nameof(value),
valueExpression);
}
```

You could call this routine as follows:

C#

```
public static void LogMessage(string? message)
{
    ThrowWhenNull(message, $"{nameof(message)} must not be null");

    Console.WriteLine(message.Length);
}
```

After enabling null reference types, you want to ensure that the preceding code compiles without warnings. When the method returns, the `value` parameter is guaranteed to be not null. However, it's acceptable to call `ThrowWhenNull` with a null reference. You can make `value` a nullable reference type, and add the `NotNull` post-condition to the parameter declaration:

C#

```
public static void ThrowWhenNull([NotNull] object? value, string val-
ueExpression = "")
{
    _ = value ?? throw new ArgumentNullException(nameof(value), val-
ueExpression);
    // other logic elided
```

The preceding code expresses the existing contract clearly: Callers can pass a variable with the `null` value, but the argument is guaranteed to never be null if the method returns without throwing an exception.

You specify unconditional postconditions using the following attributes:

- MaybeNull: A non-nullable return value may be null.
- NotNull: A nullable return value will never be null.

Conditional post-conditions: `NotNullWhen`, `MaybeNullWhen`, and `NotNullIfNotNull`

You're likely familiar with the `string` method String.IsNullOrEmpty(String). This method returns `true` when the argument is null or an empty string. It's a form of null-check: Callers don't need to null-check the argument if the method returns `false`. To make a method like this nullable aware, you'd set the argument to a nullable reference type, and add the `NotNullWhen` attribute:

C#

```
bool IsNullOrEmpty([NotNullWhen(false)] string? value)
```

That informs the compiler that any code where the return value is `false` doesn't need null checks. The addition of the attribute informs the compiler's static analysis that `IsNullOrEmpty` performs the necessary null check: when it returns `false`, the argument isn't `null`.

C#

```
string? userInput = GetUserInput();
if (!string.IsNullOrEmpty(userInput))
{
    int messageLength = userInput.Length; // no null check needed.
}
// null check needed on userInput here.
```

> **Note**
>
> The preceding example is only valid in C# 11 and later. Starting with C# 11, the **nameof expression** can reference parameter and type parameter names when used in an attribute applied to a method. In C# 10 and earlier, you need to use a string literal instead of the `nameof` expression.

The String.IsNullOrEmpty(String) method will be annotated as shown above for .NET Core 3.0. You may have similar methods in your codebase that check the state of objects for null values. The compiler won't recognize custom null check methods, and you'll need to add the annotations yourself. When you add the attribute, the compiler's static analysis knows when the tested variable has been null checked.

Another use for these attributes is the `Try*` pattern. The postconditions for `ref` and `out` arguments are communicated through the return value. Consider this method shown earlier (in a nullable disabled context):

C#

```
bool TryGetMessage(string key, out string message)
{
    if (_messageMap.ContainsKey(key))
        message = _messageMap[key];
    else
        message = null;
    return message != null;
}
```

The preceding method follows a typical .NET idiom: the return value indicates if `message` was set to the found value or, if no message is found, to the default value. If the method returns `true`, the value of `message` isn't null; otherwise, the method sets `message` to null.

In a nullable enabled context, you can communicate that idiom using the `NotNullWhen` attribute. When you annotate parameters for nullable reference types, make `message` a `string?` and add an attribute:

C#

```
bool TryGetMessage(string key, [NotNullWhen(true)] out string? mes-
sage)
{
    if (_messageMap.ContainsKey(key))
        message = _messageMap[key];
    else
        message = null;
    return message is not null;
}
```

In the preceding example, the value of `message` is known to be not null when `TryGetMessage` returns `true`. You should annotate similar methods in your codebase in the same way: the arguments could equal `null`, and are known to be not null when the method returns `true`.

There's one final attribute you may also need. Sometimes the null state of a return value depends on the null state of one or more arguments. These methods will return a non-null value whenever certain arguments aren't `null`. To correctly annotate these methods, you use the `NotNullIfNotNull` attribute. Consider the following method:

C#

```
string GetTopLevelDomainFromFullUrl(string url)
```

If the `url` argument isn't null, the output isn't `null`. Once nullable references are enabled, you need to add more annotations if your API may accept a null argument. You could annotate the return type as shown in the following code:

C#

```
string? GetTopLevelDomainFromFullUrl(string? url)
```

That also works, but will often force callers to implement extra `null` checks. The contract is that the return value would be `null` only when the argument `url` is `null`. To express that contract, you would annotate this method as shown in the following code:

C#

```
[return: NotNullIfNotNull(nameof(url))]
string? GetTopLevelDomainFromFullUrl(string? url)
```

The previous example uses the nameof operator for the parameter `url`. That feature is available in C# 11. Before C# 11, you'll need to type the name of the parameter as a string. The return value and the argument have both been annotated with the `?` indicating that either could be `null`. The attribute further clarifies that the return value won't be null when the `url` argument isn't `null`.

You specify conditional postconditions using these attributes:

- MaybeNullWhen: A non-nullable argument may be null when the method returns the specified `bool` value.
- NotNullWhen: A nullable argument won't be null when the method returns the specified `bool` value.
- NotNullIfNotNull: A return value isn't null if the argument for the specified parameter isn't null.

Helper methods: `MemberNotNull` and `MemberNotNullWhen`

These attributes specify your intent when you've refactored common code from constructors into helper methods. The C# compiler analyzes constructors and field initializers to make sure that all non-nullable reference fields have been initialized before each constructor returns. However, the C# compiler doesn't track field assignments through all helper methods. The compiler issues warning `CS8618` when fields aren't initialized directly in the constructor, but rather in a helper method. You add the MemberNotNullAttribute to a method declaration and specify the fields that are initialized to a non-null value in the method. For example, consider the following example:

C#

```
public class Container
{
    private string _uniqueIdentifier; // must be initialized.
    private string? _optionalMessage;

    public Container()
    {
        Helper();
    }

    public Container(string message)
    {
        Helper();
        _optionalMessage = message;
    }

    [MemberNotNull(nameof(_uniqueIdentifier))]
    private void Helper()
    {
        _uniqueIdentifier = DateTime.Now.Ticks.ToString();
    }
}
```

You can specify multiple field names as arguments to the `MemberNotNull` attribute constructor.

The MemberNotNullWhenAttribute has a `bool` argument. You use `MemberNotNullWhen` in situations where your helper method returns a `bool` indicating whether your helper method initialized fields.

Stop nullable analysis when called method throws

Some methods, typically exception helpers or other utility methods, always exit by throwing an exception. Or, a helper may throw an exception based on the value of a Boolean argument.

In the first case, you can add the DoesNotReturnAttribute attribute to the method declaration. The compiler's *null-state* analysis doesn't check any code in a method that follows a call to a method annotated with `DoesNotReturn`. Consider this method:

C#

```
[DoesNotReturn]
private void FailFast()
{
```

```
    throw new InvalidOperationException();
}

public void SetState(object containedField)
{
    if (containedField is null)
    {
        FailFast();
    }

    // containedField can't be null:
    _field = containedField;
}
```

The compiler doesn't issue any warnings after the call to `FailFast`.

In the second case, you add the System.Diagnostics.CodeAnalysis.DoesNotReturnIfAttribute attribute to a Boolean parameter of the method. You can modify the previous example as follows:

C#

```
private void FailFastIf([DoesNotReturnIf(true)] bool isNull)
{
    if (isNull)
    {
        throw new InvalidOperationException();
    }
}

public void SetFieldState(object? containedField)
{
    FailFastIf(containedField == null);
    // No warning: containedField can't be null here:
    _field = containedField;
}
```

When the value of the argument matches the value of the `DoesNotReturnIf` constructor, the compiler doesn't perform any *null-state* analysis after that method.

Summary

Adding nullable reference types provides an initial vocabulary to describe your APIs expectations for variables that could be `null`. The attributes provide a richer vocabulary to describe the null state of variables as preconditions and postconditions. These attributes more clearly describe your expectations and provide a better experience for the developers using your APIs.

As you update libraries for a nullable context, add these attributes to guide users of your APIs to the correct usage. These attributes help you fully describe the null-state of arguments and return values.

- AllowNull: A non-nullable field, parameter, or property may be null.
- DisallowNull: A nullable field, parameter, or property should never be null.
- MaybeNull: A non-nullable field, parameter, property, or return value may be null.
- NotNull: A nullable field, parameter, property, or return value will never be null.
- MaybeNullWhen: A non-nullable argument may be null when the method returns the specified `bool` value.
- NotNullWhen: A nullable argument won't be null when the method returns the specified `bool` value.
- NotNullIfNotNull: A parameter, property, or return value isn't null if the argument for the specified parameter isn't null.
- DoesNotReturn: A method or property never returns. In other words, it always throws an exception.
- DoesNotReturnIf: This method or property never returns if the associated `bool` parameter has the specified value.