C# Warning waves

Article • 09/29/2022

New warnings and errors may be introduced in each release of the C# compiler. When new warnings could be reported on existing code, those warnings are introduced under an opt-in system referred to as a *warning wave*. The opt-in system means that you shouldn't see new warnings on existing code without taking action to enable them. Warning waves are enabled using the **AnalysisLevel** element in your project file. When `<TreatWarningsAsErrors>true</TreatWarningsAsErrors>` is specified, enabled warning wave warnings generate errors. Warning wave 5 diagnostics were added in C# 9. Warning wave 6 diagnostics were added in C# 10. Warning wave 7 diagnostics were added in C# 11.

CS8981 - The type name only contains lower-cased ascii characters.

Warning wave 7

Any new keywords added for C# will be all lower-case ASCII characters. This warning ensures that none of your types conflict with future keywords. The following code produces CS8981:

C#

```csharp
public class lowercasename
{
}
```

You can address this warning by renaming the type to include at least one non-lower case ASCII character, such as an upper case character, a digit, or an underscore.

CS8826 - Partial method declarations have signature differences.

Warning wave 6

This warning corrects some inconsistencies in reporting differences between partial method signatures. The compiler always reported an error when the partial method signatures created different CLR signatures. Now, the compiler reports CS8826 when the signatures are syntactically different C#. Consider the following partial class:

C#

```
public partial class PartialType
{
    public partial void M1(int x);

    public partial T M2<T>(string s) where T : struct;

    public partial void M3(string s);


    public partial void M4(object o);
    public partial void M5(dynamic o);
    public partial void M6(string? s);
}
```

The following partial class implementation generates several examples of CS8626:

C#

```
public partial class PartialType
{
    // Different parameter names:
    public partial void M1(int y) { }

    // Different type parameter names:
    public partial TResult M2<TResult>(string s) where TResult :
struct => default;

    // Relaxed nullability
    public partial void M3(string? s) { }


    // Mixing object and dynamic
    public partial void M4(dynamic o) { }

    // Mixing object and dynamic
    public partial void M5(object o) { }

    // Note: This generates CS8611 (nullability mismatch) not CS8826
    public partial void M6(string s) { }
}
```

ⓘ Note

If the implementation of a method uses a non-nullable reference type when the other declaration accepts nullable reference types, CS8611 is generated instead of CS8826.

To fix any instance of these warnings, ensure the two signatures match.

CS7023 - A static type is used in an 'is' or 'as' expression.

Warning wave 5

The `is` and `as` expressions always return `false` for a static type because you can't create instances of a static type. The following code produces CS7023:

C#

```csharp
static class StaticClass
{
    public static void Thing() { }
}

void M(object o)
{
    // warning: cannot use a static type in 'is' or 'as'
    if (o is StaticClass)
    {
        Console.WriteLine("Can't happen");
    }
    else
    {
        Console.WriteLine("o is not an instance of a static class");
    }
}
```

The compiler reports this warning because the type test can never succeed. To correct this warning, remove the test and remove any code executed only if the test succeeded. In the preceding example, the `else` clause is always executed. The method body could be replaced by that single line:

C#

```csharp
Console.WriteLine("o is not an instance of a static class");
```

CS8073 - The result of the expression is always 'false' (or 'true').

Warning wave 5

The `==` and `!=` operators always return `false` (or `true`) when comparing an instance of a `struct` type to `null`. The following code demonstrates this warning. Assume `S` is a `struct` that has defined `operator ==` and `operator !=`:

```csharp
class Program
{
    public static void M(S s)
    {
        if (s == null) { } // CS8073: The result of the expression is always 'false'
        if (s != null) { } // CS8073: The result of the expression is always 'true'
    }
}

struct S
{
    public static bool operator ==(S s1, S s2) => s1.Equals(s2);
    public static bool operator !=(S s1, S s2) => !s1.Equals(s2);
    public override bool Equals(object? other)
    {
        // Implementation elided
        return false;
    }
    public override int GetHashCode() => 0;

    // Other details elided...
}
```

To fix this error, remove the null check and code that would execute if the object is `null`.

CS8848 - Operator 'from' can't be used here due to precedence. Use parentheses to disambiguate.

Warning wave 5

The following examples demonstrate this warning. The expression binds incorrectly because of the precedence of the operators.

C#

```csharp
bool b = true;
var source = new Src();
b = true;
source = new Src();
var a = b && from c in source select c;
Console.WriteLine(a);

var indexes = new Src2();
int[] array = { 1, 2, 3, 4, 5, 6, 7 };
var range = array[0..from c in indexes select c];
```

To fix this error, put parentheses around the query expression:

C#

```csharp
bool b = true;
var source = new Src();
b = true;
source = new Src();
var a = b && (from c in source select c);
Console.WriteLine(a);

var indexes = new Src2();
int[] array = { 1, 2, 3, 4, 5, 6, 7 };
var range = array[0..(from c in indexes select c)];
```

Members must be fully assigned, use of unassigned variable (CS8880, CS8881, CS8882, CS8883, CS8884, CS8885, CS8886, CS8887)

Warning wave 5

Several warnings improve the definite assignment analysis for `struct` types declared in imported assemblies. All these new warnings are generated when a struct in an imported assembly includes an inaccessible field (usually a `private` field) of a reference type, as shown in the following example:

C#

```
public struct Struct
{
    private string data = String.Empty;
    public Struct() { }
}
```

The following examples show the warnings generated from the improved definite assignment analysis:

- CS8880: Auto-implemented property 'Property' must be fully assigned before control is returned to the caller.
- CS8881: Field 'field' must be fully assigned before control is returned to the caller.
- CS8882: The out parameter 'parameter' must be assigned to before control leaves the current method.
- CS8883: Use of possibly unassigned auto-implemented property 'Property'.
- CS8884: Use of possibly unassigned field 'Field'
- CS8885: The 'this' object can't be used before all its fields have been assigned.
- CS8886: Use of unassigned output parameter 'parameterName'.
- CS8887: Use of unassigned local variable 'variableName'

```C#
public struct DefiniteAssignmentWarnings
{
    // CS8880
    public Struct Property { get; }
    // CS8881
    private Struct field;

    // CS8882
    public void Method(out Struct s)
    {

    }

    public DefiniteAssignmentWarnings(int dummy)
    {
        // CS8883
        Struct v2 = Property;
        // CS8884
        Struct v3 = field;
        // CS8885:
        DefiniteAssignmentWarnings p2 = this;
    }

    public static void Method2(out Struct s1)
    {
        // CS8886
        var s2 = s1;
        s1 = default;
    }

    public static void UseLocalStruct()
    {
        Struct r1;
        var r2 = r1;
    }
}
```

You can fix any of these warnings by initializing or assigning the imported struct to its default value:

```C#
public struct DefiniteAssignmentNoWarnings
{
    // CS8880
    public Struct Property { get; } = default;
    // CS8881
    private Struct field = default;

    // CS8882
    public void Method(out Struct s)
    {
        s = default;
    }

    public DefiniteAssignmentNoWarnings(int dummy)
    {
        // CS8883
        Struct v2 = Property;
        // CS8884
        Struct v3 = field;
        // CS8885:
        DefiniteAssignmentNoWarnings p2 = this;
    }

    public static void Method2(out Struct s1)
    {
        // CS8886
        s1 = default;
        var s2 = s1;
    }

    public static void UseLocalStruct()
    {
        Struct r1 = default;
        var r2 = r1;
    }
}
```

CS8892 - Method will not be used as an entry point because a synchronous entry point 'method' was found.

Warning wave 5

This warning is generated on all async entry point candidates when you have multiple valid entry points, including one or more synchronous entry point.

The following example generates CS8892:

C#

```csharp
public static void Main()
{
    RunProgram();
}

// CS8892
public static async Task Main(string[] args)
{
    await RunProgramAsync();
}
```

Note

The compiler always uses the synchronous entry point. In case there are multiple synchronous entry points, you get a compiler error.

To fix this warning, remove or rename the asynchronous entry point.

CS8897 - Static types can't be used as parameters

Warning wave 5

Members of an interface can't declare parameters whose type is a static class. The following code demonstrates both CS8897 and CS8898:

C#

```csharp
public static class Utilities
{
    // elided
}

public interface IUtility
{
    // CS8897
    public void SetUtility(Utilities u);

    // CS8898
    public Utilities GetUtility();
}
```

To fix this warning, change the parameter type or remove the method.

CS8898 - static types can't be used as return types

Warning wave 5

Members of an interface can't declare a return type that is a static class. The following code demonstrates both CS8897 and CS8898:

```C#
public static class Utilities
{
    // elided
}

public interface IUtility
{
    // CS8897
    public void SetUtility(Utilities u);

    // CS8898
    public Utilities GetUtility();
}
```

To fix this warning, change the return type or remove the method.