

Tipos de referencia que aceptan valores NULL

Artículo • 10/05/2023

En un contexto "oblivious" que acepta valores NULL, todos los tipos de referencia aceptaban valores null. Los *tipos de referencia que aceptan valores NULL* hacen referencia a un grupo de características habilitadas en un contexto que admite un valor NULL que minimizan las probabilidades de que el código haga que el tiempo de ejecución genere [System.NullReferenceException](#). Los *tipos de referencia que aceptan valores NULL* incluyen tres características que ayudan a evitar estas excepciones, incluida la capacidad de marcar explícitamente un tipo de referencia como que *acepta valores NULL*:

- El análisis de flujo estático mejorado que determina si una variable puede estar `null` antes de desreferenciarla.
- Los atributos que anotan las API para que el análisis de flujo determine el *null-state*.
- Las anotaciones de variables que los desarrolladores usan para declarar explícitamente el *null-state* previsto para una variable.

El análisis de null-state y las anotaciones de variables están deshabilitados de forma predeterminada en los proyectos existentes, lo que significa que todos los tipos de referencia siguen aceptando valores NULL. A partir de .NET 6, están habilitados de forma predeterminada en los proyectos *nuevos*. Para obtener información sobre cómo habilitar estas características mediante la declaración de un *contexto de anotación que acepta valores NULL*, vea [Contextos que aceptan valores NULL](#).

En el resto de este artículo se describe cómo funcionan esas tres áreas de características para generar advertencias cuando el código puede estar **desreferenciando** un valor `null`. Desreferenciar una variable significa acceder a uno de sus miembros mediante el operador `.` (punto), como se muestra en el ejemplo siguiente:

C#

```
string message = "Hello, World!";  
int length = message.Length; // dereferencing "message"
```

Al desreferenciar una variable cuyo valor es `null`, el entorno de ejecución produce una excepción [System.NullReferenceException](#).

También puede explorar estos conceptos en nuestro módulo de aprendizaje sobre [Seguridad sobre la aceptación de valores NULL en C#](#).

Análisis del null-state

El *análisis de null-state* realiza el seguimiento del elemento *null-state* de las referencias. Este análisis estático emite advertencias cuando puede que el código desreferencie `null`. Puede solucionar estas advertencias para minimizar las incidencias cuando el entorno de ejecución produce una excepción [System.NullReferenceException](#). El compilador usa el análisis estático para determinar el estado *null-state* de una variable. Una variable puede ser *not-null* o *maybe-null*. El compilador determina que una variable es *not-null* de dos maneras:

1. Se ha asignado un valor a la variable que se sabe que *no es NULL*.
2. La variable se ha comprobado con `null` y no se ha modificado desde esa comprobación.

Cualquier variable que el compilador no haya determinado como *not-null* se considera *maybe-null*. El análisis proporciona advertencias en situaciones en las que puede desreferenciar accidentalmente un valor `null`. El compilador genera advertencias basadas en el estado *null-state*.

- Cuando una variable es *not-null*, esa variable se puede desreferenciar de forma segura.
- Cuando una variable es *maybe-null*, se debe comprobar esa variable para asegurarse de que no sea `null` antes de desreferenciarla.

Considere el ejemplo siguiente:

C#

```
string message = null;

// warning: dereference null.
Console.WriteLine($"The length of the message is {message.Length}");

var originalMessage = message;
message = "Hello, World!";

// No warning. Analysis determined "message" is not null.
Console.WriteLine($"The length of the message is {message.Length}");
```

```
// warning!  
Console.WriteLine(originalMessage.Length);
```

En el ejemplo anterior, el compilador determina que `message` es *maybe-null* cuando se imprime el primer mensaje. No hay ninguna advertencia para el segundo mensaje. La línea de código final genera una advertencia porque `originalMessage` podría ser NULL. En el ejemplo siguiente se muestra un uso más práctico para recorrer un árbol de nodos hasta la raíz y procesar cada nodo durante el recorrido:

C#

```
void FindRoot(Node node, Action<Node> processNode)  
{  
    for (var current = node; current != null; current = current.Parent)  
    {  
        processNode(current);  
    }  
}
```

El código anterior no genera advertencias para desreferenciar la variable `current`. El análisis estático determina que `current` nunca se desreferencie cuando es *maybe-null*. La variable `current` se comprueba con `null` antes de acceder a `current.Parent` y antes de pasar `current` a la acción `ProcessNode`. En los ejemplos anteriores se muestra cómo el compilador determina el estado *null-state* de las variables locales cuando se inicializan, se asignan o se comparan con `null`.

El análisis del estado NULL no realiza un seguimiento de los métodos llamados. Como resultado, los campos inicializados en un método auxiliar común llamado por constructores generarán una advertencia con la plantilla siguiente:

La propiedad "*name*" que no acepta valores NULL debe contener un valor distinto de NULL al salir del constructor.

Puede solucionar estas advertencias de una de estas dos maneras: *encadenamiento de constructores* o *atributos que aceptan valores NULL* en el método auxiliar. En el código siguiente se muestra un ejemplo de cada caso. La clase `Person` usa un constructor común al que llaman todos los demás constructores. La clase `Student` tiene un método auxiliar anotado con el atributo [System.Diagnostics.CodeAnalysis.MemberNotNullAttribute](#):

C#

```
using System.Diagnostics.CodeAnalysis;

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public Person() : this("John", "Doe") { }
}

public class Student : Person
{
    public string Major { get; set; }

    public Student(string firstName, string lastName, string major)
        : base(firstName, lastName)
    {
        SetMajor(major);
    }

    public Student(string firstName, string lastName) :
        base(firstName, lastName)
    {
        SetMajor();
    }

    public Student()
    {
        SetMajor();
    }

    [MemberNotNull(nameof(Major))]
    private void SetMajor(string? major = default)
    {
        Major = major ?? "Undeclared";
    }
}
```

⚠ Nota

Se agregaron varias mejoras para la asignación definitiva y el análisis de estado NULL en C# 10. Al actualizar a C# 10, es posible que encuentre menos advertencias que admiten un valor NULL que son falsos positivos. Puede obtener más información sobre las mejoras en la [especificación de características para las mejoras de la asignación definitiva](#).

El análisis de estado que acepta valores NULL y las advertencias que genera el compilador ayudan a evitar errores de programa mediante la anulación de la referencia a `null`. En el artículo sobre [cómo resolver advertencias que aceptan valores NULL](#) se proporcionan técnicas para corregir las advertencias que probablemente verá en el código.

Atributos en firmas de API

El análisis del estado null-state necesita sugerencias de los desarrolladores para comprender la semántica de las API. Algunas API proporcionan comprobaciones NULL y deben cambiar el estado *null-state* de una variable de *maybe-null* a *not-null*. Otras API devuelven expresiones que son *not-null* o *maybe-null* en función del estado *null-state* de los argumentos de entrada. Por ejemplo, considere el siguiente código que muestra un mensaje en mayúsculas:

C#

```
void PrintMessageUpper(string? message)
{
    if (!IsNull(message))
    {
        Console.WriteLine($"{DateTime.Now}: {message.ToUpper()}");
    }
}

bool IsNull(string? s) => s == null;
```

En función de la inspección, cualquier desarrollador consideraría este código seguro y uno que no debería generar advertencias. Sin embargo, el compilador no sabe que `IsNull` proporciona una comprobación nula y emitirá una advertencia para la instrucción `message.ToUpper()`, teniendo en cuenta `message` que es una variable *tal vez null*. Para corregirlo, podemos usar el atributo [NotNullWhen](#):

C#

```
bool IsNull([NotNullWhen(false)] string? s) => s == null;
```

Esto informa al compilador de que, si `IsNull` devuelve `false`, el parámetro `s` no es `null`. Esto permite al compilador cambiar el *estado null* de `message` a *no null* dentro del bloque `if (!IsNull(message)) {...}`. Gracias a esto, no se emitirá ninguna advertencia.

Los atributos proporcionan información detallada sobre el estado *null-state* de los argumentos, los valores devueltos y los miembros de la instancia de objeto utilizada para invocar a un miembro. Los detalles de cada atributo se pueden encontrar en el artículo de referencia del lenguaje sobre los [atributos de referencia que aceptan valores NULL](#). Todas las API del entorno de ejecución de .NET se han anotado en .NET 5. Para mejorar el análisis estático, se anotan las API para proporcionar información semántica sobre el estado *null-state* de los argumentos y los valores devueltos.

Anotaciones de variables que aceptan valores NULL

El análisis de *null-state* proporciona un análisis sólido para la mayoría de las variables. El compilador necesita más información de usted para las variables de miembro. El compilador no puede hacer suposiciones sobre el orden en el que se accede a los miembros públicos. Se puede acceder a cualquier miembro público en cualquier orden. Cualquiera de los constructores accesibles se podría usar para inicializar el objeto. Si un campo de miembro se puede establecer alguna vez en `null`, el compilador debe suponer que su *null-state* es *maybe-null* al principio de cada método.

Se usan anotaciones que pueden declarar si una variable es un **tipo de referencia que acepta valores NULL** o un **tipo de referencia que no acepta valores NULL**. Estas anotaciones hacen instrucciones importantes sobre el estado *null-state* de las variables:

- **Se supone que una referencia no debe ser NULL.** El estado predeterminado de una variable de referencia que no acepta valores NULL es *not-null*. El compilador aplica reglas que garantizan que sea seguro desreferenciar dichas variables sin comprobar primero que no se trata de un valor NULL:
 - La variable debe inicializarse como un valor distinto a NULL.
 - No se puede asignar el valor `null` a la variable. El compilador emite una advertencia cuando el código asigna una expresión *maybe-null* a una variable que no debería ser NULL.

- **Una referencia puede ser NULL.** El estado predeterminado de una variable de referencia que acepta valores NULL es *maybe-null*. El compilador aplica reglas para garantizar que haya comprobado correctamente una referencia `null`:
 - Solo se puede desreferenciar la variable si el compilador puede garantizar que el valor no sea `null`.
 - Estas variables se pueden inicializar con el valor `null` predeterminado, y se les puede asignar el valor `null` en otro código.
 - El compilador no emite advertencias cuando el código asigna una expresión *maybe-null* a una variable que podría ser NULL.

Cualquier variable de referencia que no se suponga que es `null` tiene un estado *null-state* de *not-null*. Cualquier variable de referencia que pueda ser `null` inicialmente tiene el estado *null-state* de *maybe-null*.

Un **tipo de referencia que acepta valores NULL** se anota con la misma sintaxis que los [tipos de valor que aceptan valores NULL](#): se agrega `?` junto al tipo de la variable. Por ejemplo, la siguiente declaración de variable representa una variable de cadena que acepta valores NULL, `name`:

C#

```
string? name;
```

Cualquier variable en la que `?` no se anexe al nombre de tipo es un **tipo de referencia que no acepta valores NULL**. Esto incluye todas las variables de tipo de referencia en el código existente en el momento en el que se habilita esta característica. Sin embargo, cualquier variable local con tipo implícito (declarada mediante `var`) es un **tipo de referencia que acepta valores NULL**. Como se ha mostrado en las secciones anteriores, el análisis estático determina el estado *null-state* de las variables locales para determinar si son *maybe-null*.

En algunas ocasiones, debe invalidar una advertencia si sabe que una variable no es NULL pero el compilador determina que su *null-state* es *maybe-null*. Use el [operador null-forgiving](#) ! después de un nombre de variable para forzar que *null-state* sea *not-null*. Por ejemplo, si sabe que la variable `name` no es `null`, pero el compilador genera una advertencia, puede escribir el código siguiente para invalidar el análisis del compilador:

C#

```
name!.Length;
```

Los tipos de referencia que aceptan valores NULL y los tipos de valor que aceptan valores NULL proporcionan un concepto semántico similar: una variable puede representar un valor u objeto, o esa variable puede ser `null`. Sin embargo, los tipos de referencia que aceptan valores NULL y los tipos de valor que aceptan valores NULL se implementan de forma diferente: los tipos de valor que aceptan valores NULL se implementan mediante [System.Nullable<T>](#) y los tipos de referencia que aceptan valores NULL se implementan mediante atributos leídos por el compilador. Por ejemplo, `string?` y `string` se representan mediante el mismo tipo: [System.String](#). Sin embargo, `int?` y `int` se representan mediante [System.Nullable<System.Int32>](#) y [System.Int32](#), respectivamente.

Los tipos de referencia no nulas son una característica de tiempo de compilación. Esto significa que es posible que los autores de llamadas ignoren las advertencias, utilizando intencionadamente `null` como argumento de un método que espera una referencia no nula. Los autores de bibliotecas deben incluir comprobaciones en tiempo de ejecución con valores de argumento NULL. [ArgumentNullException.ThrowIfNull](#) es la opción preferida para comparar un parámetro con NULL en tiempo de ejecución.

Importante

La habilitación de anotaciones nulas puede cambiar la forma en que Entity Framework Core determina si se requiere un miembro de datos. Puede obtener más información en el artículo sobre [aspectos básicos de Entity Framework: Trabajar con tipos de referencia nula](#).

Genéricos

Los genéricos requieren reglas detalladas para controlar `T?` para cualquier parámetro de tipo `T`. Las reglas se detallan necesariamente debido al historial y a la implementación diferente para un tipo de valor que acepta valores NULL y un tipo de referencia que acepta valores NULL. [Los tipos de valor que aceptan valores NULL](#) se implementan mediante la estructura [System.Nullable<T>](#). [Los tipos de referencia que aceptan valores NULL](#) se implementan como anotaciones de tipo que proporcionan reglas semánticas al compilador.

- Si el argumento de tipo de T es un tipo de referencia, $T?$ hace referencia al tipo de referencia que acepta valores NULL correspondiente. Por ejemplo, si T es un elemento `string`, entonces $T?$ es un elemento `string?`.
- Si el argumento de tipo de T es un tipo de valor, $T?$ hace referencia al mismo tipo de valor, T . Por ejemplo, si T es un elemento `int`, $T?$ también es un elemento `int`.
- Si el argumento de tipo de T es un tipo de referencia que acepta valores NULL, $T?$ hace referencia a ese mismo tipo de referencia que acepta valores NULL. Por ejemplo, si T es un elemento `string?`, entonces $T?$ también es un elemento `string?`.
- Si el argumento de tipo de T es un tipo de valor que acepta valores NULL, $T?$ hace referencia a ese mismo tipo de valor que acepta valores NULL. Por ejemplo, si T es un elemento `int?`, entonces $T?$ también es un elemento `int?`.

Para los valores devueltos, $T?$ es equivalente a `[MaybeNull]T`; para los valores de argumento, $T?$ es equivalente a `[AllowNull]T`. Para obtener más información, consulte el artículo sobre [Atributos para el análisis de estado NULL](#) en la referencia del lenguaje.

Puede especificar un comportamiento diferente mediante [restricciones](#):

- La restricción `class` significa que T debe ser un tipo de referencia que no acepta valores NULL (por ejemplo, `string`). El compilador genera una advertencia si se usa un tipo de referencia que acepta valores NULL, como `string?` para T .
- La restricción `class?` significa que T debe ser un tipo de referencia, ya sea un tipo de referencia que no acepta valores NULL (`string`) o un tipo de referencia que acepta valores NULL (por ejemplo, `string?`). Cuando el parámetro de tipo es un tipo de referencia que acepta valores NULL, como `string?`, una expresión de $T?$ hace referencia a ese mismo tipo de referencia que acepta valores NULL, como `string?`.
- La restricción `notnull` significa que T debe ser un tipo de referencia que no acepta valores NULL o un tipo de valor que no acepta valores NULL. Si usa un tipo de referencia que acepta valores NULL o un tipo de valor que acepta valores NULL para el parámetro de tipo, el compilador genera una advertencia. Además, cuando T es un tipo de valor, el valor devuelto es ese tipo de valor, no el tipo de valor que acepta valores NULL correspondiente.

Estas restricciones ayudan a proporcionar más información al compilador sobre cómo se usará T . Esto ayuda cuando los desarrolladores eligen el tipo para T y proporciona un mejor análisis de *null-state* cuando se usa una instancia del tipo genérico.

Contextos que aceptan valores NULL

Las nuevas características que protegen contra la generación de un elemento [System.NullReferenceException](#) pueden ser perjudiciales cuando están activadas en un código base existente:

- Todas las variables de referencia con tipo explícito se interpretan como tipos de referencia que no aceptan valores NULL.
- El significado de la restricción `class` en genéricos cambió para significar un tipo de referencia que no acepta valores NULL.
- Se generan nuevas advertencias debido a estas nuevas reglas.

Debe elegir expresamente usar estas características en los proyectos existentes. Esto proporciona una ruta de migración y conserva la compatibilidad con versiones anteriores. Los contextos que aceptan valores NULL permiten un control preciso sobre cómo interpreta el compilador las variables de tipos de referencia. El **contexto de anotación que acepta valores NULL** determina el comportamiento del compilador. Hay cuatro valores para el **contexto de anotación que acepta valores NULL**:

- *disable*: el código es "oblivious" que admite un valor NULL.
 - Las advertencias que aceptan valores NULL están deshabilitadas.
 - Todas las variables de tipo de referencia son tipos de referencia que aceptan valores NULL.
 - El uso del sufijo `?` para declarar un tipo de referencia que acepta valores NULL genera una advertencia.
 - Puede usar el operador que permite un valor NULL, `!`, pero no tiene ningún efecto.
- *enable*: el compilador permite todo el análisis de referencias nulas y todas las características del lenguaje.
 - Todas las nuevas advertencias que aceptan valores NULL están habilitadas.
 - Puede usar el sufijo `?` para declarar un tipo de referencia que acepta valores NULL.
 - Todas las demás variables de tipo de referencia son tipos de referencia que no aceptan valores NULL.
 - El operador que permite valores NULL elimina las advertencias de una posible asignación a `null`.
- *warnings*: el compilador realiza todos los análisis de valores NULL y emite advertencias cuando el código pueda desreferenciar `null`.
 - Todas las nuevas advertencias que aceptan valores NULL están habilitadas.

- El uso del sufijo `?` para declarar un tipo de referencia que acepta valores NULL genera una advertencia.
- Todas las variables de tipo de referencia pueden ser NULL. Sin embargo, los miembros tienen el estado *null-state* de *not-null* en la llave de apertura de todos los métodos, a menos que se declaren con el sufijo `?`.
- Puede usar el operador que permite valores NULL, `!`.
- *annotations*: el compilador no realiza análisis de valores NULL ni emite advertencias cuando el código pueda desreferenciar `null`.
 - Todas las nuevas advertencias que aceptan valores NULL están deshabilitadas.
 - Puede usar el sufijo `?` para declarar un tipo de referencia que acepta valores NULL.
 - Todas las demás variables de tipo de referencia son tipos de referencia que no aceptan valores NULL.
 - Puede usar el operador que permite un valor NULL, `!`, pero no tiene ningún efecto.

Tanto el contexto de anotación que acepta valores NULL como el contexto de advertencia que acepta valores NULL pueden establecerse en un proyecto con el elemento `<Nullable>` del archivo `.csproj`. Este elemento configura la forma en la que el compilador interpreta la nulabilidad de los tipos y las advertencias que se generan. En la tabla siguiente se muestran los valores permitidos y se resumen los contextos que dichos valores especifican.

[Expandir tabla](#)

Context	Advertencias de desreferenciación	Advertencias de asignación	Tipos de referencia	Sufijo <code>?</code>	Operador <code>!</code>
disable	Disabled	Disabled	Todos aceptan valores NULL.	Genera una advertencia.	No tiene ningún efecto.
enable	habilitado	habilitado	No acepta valores NULL a menos que se declare con <code>?</code> .	Declara un tipo que acepta valores NULL.	Suprime las advertencias relativas a una posible asignación <code>null</code> .
warnings	habilitado	No aplicable	Todos aceptan valores NULL, pero los miembros se consideran <i>no</i>	Genera una advertencia.	Suprime las advertencias relativas a una posible

Context	Advertencias de desreferenciación	Advertencias de asignación	Tipos de referencia	Sufijo ?	Operador !
			NULL en la llave de apertura de los métodos.		asignación null.
annotations	Disabled	Disabled	No acepta valores NULL a menos que se declare con ?.	Declara un tipo que acepta valores NULL.	No tiene ningún efecto.

Las variables de tipo de referencia en el código compilado en un contexto *deshabilitado* son *nullable-oblivious*. Puede asignar un valor `null` literal o una variable *maybe-null* a una variable que es *nullable-oblivious*. Sin embargo, el estado predeterminado de una variable *nullable-oblivious* es *not-null*.

Puede elegir qué configuración es la mejor para el proyecto:

- Elija *disable* para los proyectos heredados que no quiere actualizar en función de diagnósticos o nuevas características.
- Elija *warnings* para determinar dónde el código puede producir [System.NullReferenceException](#). Puede solucionar esas advertencias antes de modificar el código para habilitar tipos de referencia que no aceptan valores NULL.
- Elija *annotations* para expresar la intención de diseño antes de habilitar las advertencias.
- Elija *enable* para nuevos proyectos y proyectos activos en los que quiera protegerse de excepciones de referencia nula.

Ejemplo:

XML
<code><Nullable>enable</Nullable></code>

También puede usar directivas para establecer los mismos contextos en cualquier lugar del código fuente: Son muy útiles cuando se migra un código base grande.

- `#nullable enable`: establece el contexto de anotación nula y el contexto de advertencia nula en **enable**.
- `#nullable disable`: establece el contexto de anotación nula y el contexto de advertencia nula en **disable**.
- `#nullable restore`: restaura el contexto de anotación que acepta valores NULL y el contexto de advertencia que acepta valores NULL según la configuración del proyecto.
- `#nullable disable warnings`: establezca el contexto de advertencia nula en **disable**.
- `#nullable enable warnings`: establece el contexto de advertencia nula en **enable**.
- `#nullable restore warnings`: restaura el contexto de advertencia que acepta valores NULL según la configuración del proyecto.
- `#nullable disable annotations`: establece el contexto de anotación nula en **disable**.
- `#nullable enable annotations`: establece el contexto de anotación nula en **enable**.
- `#nullable restore annotations`: restaura el contexto de advertencia de anotación según la configuración del proyecto.

Para cualquier línea de código, puede establecer cualquiera de las siguientes combinaciones:

 Expandir tabla

Contexto de advertencia	Contexto de anotación	Uso
proyecto predeterminado	proyecto predeterminado	Valor predeterminado
enable	disable	Corrección de advertencias de análisis
enable	proyecto predeterminado	Corrección de advertencias de análisis
proyecto predeterminado	enable	Adición de anotaciones de tipo
enable	enable	Código ya migrado
disable	enable	Anotación de código antes de corregir advertencias
disable	disable	Adición de código heredado al proyecto migrado

Contexto de advertencia	Contexto de anotación	Uso
proyecto predeterminado	disable	Raramente
disable	proyecto predeterminado	Raramente

Esas nueve combinaciones proporcionan un control preciso sobre los diagnósticos que el compilador emite para el código. Puede habilitar más características en cualquier área que esté actualizando sin ver advertencias adicionales que aún no está listo para abordar.

Importante

El contexto global que admite un valor NULL no se aplica a los archivos de código generado. En cualquier estrategia, el contexto que admite un valor NULL está *deshabilitado* para cualquier archivo de código fuente marcado como generado. Esto significa que las API de los archivos generados no se anotan. Hay cuatro maneras de marcar un archivo como generado:

1. En el archivo `.editorconfig`, especifique `generated_code = true` en una sección que se aplique a ese archivo.
2. Coloque `<auto-generated>` o `<auto-generated/>` en un comentario en la parte superior del archivo. Puede estar en cualquier línea de ese comentario, pero el bloque de comentario debe ser el primer elemento del archivo.
3. Inicie el nombre de archivo con *TemporaryGeneratedFile_*
4. Finalice el nombre de archivo con *.designer.cs*, *.generated.cs*, *.g.cs* o *.g.i.cs*.

Los generadores pueden optar por usar la directiva de preprocesador **#nullable**.

De forma predeterminada, los contextos de advertencias y anotaciones que aceptan valores NULL están **deshabilitados**. Esto implica que el código existente se compila sin cambios y sin generar ninguna advertencia nueva. A partir de .NET 6, los proyectos nuevos incluyen el elemento `<Nullable>enable</Nullable>` en todas las plantillas de proyecto.

Estas opciones proporcionan dos estrategias distintas para [actualizar un código base existente](#) para usar tipos de referencia que aceptan valores NULL.

Problemas conocidos

Las matrices y estructuras que contienen tipos de referencia son dificultades conocidas en las referencias que aceptan valores NULL y el análisis estático que determina la seguridad de los valores NULL. En ambas situaciones, se puede inicializar una referencia que no acepta valores NULL en `null` sin generar advertencias.

Estructuras

Una estructura que contiene tipos de referencia que no aceptan valores NULL permite asignarle `default` sin ninguna advertencia. Considere el ejemplo siguiente:

C#

```
using System;

#nullable enable

public struct Student
{
    public string FirstName;
    public string? MiddleName;
    public string LastName;
}

public static class Program
{
    public static void PrintStudent(Student student)
    {
        Console.WriteLine($"First name: {student.FirstName.ToUpper()}");
        Console.WriteLine($"Middle name: {student.MiddleName?.ToUpper()}");
        Console.WriteLine($"Last name: {student.LastName.ToUpper()}");
    }

    public static void Main() => PrintStudent(default);
}
```

En el ejemplo anterior, no hay ninguna advertencia en `PrintStudent(default)` mientras que los tipos de referencia que no aceptan valores NULL `FirstName` y `LastName` son NULL.

Otro caso más común es cuando se trata de estructuras genéricas. Considere el ejemplo siguiente:

C#

```
#nullable enable

public struct Foo<T>
{
    public T Bar { get; set; }
}

public static class Program
{
    public static void Main()
    {
        string s = default(Foo<string>).Bar;
    }
}
```

En el ejemplo anterior, la propiedad `Bar` será `null` en tiempo de ejecución y se asigna a una cadena que no acepta valores NULL sin ninguna advertencia.

Matrices

Las matrices también son un problema conocido en los tipos de referencia que aceptan valores NULL. Considere el ejemplo siguiente, que no genera ninguna advertencia:

```
C#

using System;

#nullable enable

public static class Program
{
    public static void Main()
    {
        string[] values = new string[10];
        string s = values[0];
        Console.WriteLine(s.ToUpper());
    }
}
```

En el ejemplo anterior, la declaración de la matriz muestra que contiene cadenas que no aceptan valores NULL, mientras que todos sus elementos se inicializan en `null`. Después, a la variable `s` se le asigna un valor `null` (el primer elemento de la matriz). Por último, se desreferencia la variable `s`, lo que genera una excepción en tiempo de ejecución.

Vea también

- [Tipos de referencia que aceptan valores NULL: propuesta](#)
- [Borrador de especificación de tipos de referencia que aceptan valores NULL](#)
- [Anotaciones de parámetros de tipo sin restricciones](#)
- [Tutorial de introducción a las referencias que no aceptan valores NULL](#)
- [Nullable](#) (opción del compilador de C#)


Colaborar con nosotros en GitHub


El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)