! (null-forgiving) operator (C# reference)

Article • 12/02/2022

The unary postfix `!` operator is the null-forgiving, or null-suppression, operator. In an enabled nullable annotation context, you use the null-forgiving operator to suppress all nullable warnings for the preceding expression. The unary prefix `!` operator is the logical negation operator. The null-forgiving operator has no effect at run time. It only affects the compiler's static flow analysis by changing the null state of the expression. At run time, expression `x!` evaluates to the result of the underlying expression `x`.

For more information about the nullable reference types feature, see Nullable reference types.

Examples

One of the use cases of the null-forgiving operator is in testing the argument validation logic. For example, consider the following class:

C#

```
#nullable enable
public class Person
{
    public Person(string name) => Name = name ?? throw new
ArgumentNullException(nameof(name));

    public string Name { get; }
}
```

Using the MSTest test framework, you can create the following test for the validation logic in the constructor:

C#

```
[TestMethod, ExpectedException(typeof(ArgumentNullException))]
public void NullNameShouldThrowTest()
{
    var person = new Person(null!);
}
```

Without the null-forgiving operator, the compiler generates the following warning for the preceding code: `Warning CS8625: Cannot convert null literal to non-`

`nullable reference type`. By using the null-forgiving operator, you inform the compiler that passing `null` is expected and shouldn't be warned about.

You can also use the null-forgiving operator when you definitely know that an expression can't be `null` but the compiler doesn't manage to recognize that. In the following example, if the `IsValid` method returns `true`, its argument isn't `null` and you can safely dereference it:

C#

```
public static void Main()
{
    Person? p = Find("John");
    if (IsValid(p))
    {
        Console.WriteLine($"Found {p!.Name}");
    }
}

public static bool IsValid(Person? person)
    => person is not null && person.Name is not null;
```

Without the null-forgiving operator, the compiler generates the following warning for the `p.Name` code: `Warning CS8602: Dereference of a possibly null reference`.

If you can modify the `IsValid` method, you can use the NotNullWhen attribute to inform the compiler that an argument of the `IsValid` method can't be `null` when the method returns `true`:

C#

```
public static void Main()
{
    Person? p = Find("John");
    if (IsValid(p))
    {
        Console.WriteLine($"Found {p.Name}");
    }
}

public static bool IsValid([NotNullWhen(true)] Person? person)
    => person is not null && person.Name is not null;
```

In the preceding example, you don't need to use the null-forgiving operator because the compiler has enough information to find out that `p` can't be `null` inside the `if` statement. For more information about the attributes that allow you to provide

additional information about the null state of a variable, see Upgrade APIs with attributes to define null expectations.

C# language specification

For more information, see The null-forgiving operator section of the draft of the nullable reference types specification.

See also

- Remove unnecessary suppression operator (style rule IDE0080)
- C# operators and expressions
- Tutorial: Design with nullable reference types