

Nullable value types (C# reference)

Article • 03/18/2022 • 7 minutes to read • [6 contributors](#)



In this article

[Declaration and assignment](#)

[Examination of an instance of a nullable value type](#)

[Conversion from a nullable value type to an underlying type](#)

[Lifted operators](#)

[Boxing and unboxing](#)

[How to identify a nullable value type](#)

[C# language specification](#)

[See also](#)

A *nullable value type* $T?$ represents all values of its underlying [value type](#) T and an additional [null](#) value. For example, you can assign any of the following three values to a `bool?` variable: `true`, `false`, or `null`. An underlying value type T cannot be a nullable value type itself.

Note


C# 8.0 introduces the nullable reference types feature. For more information, see [Nullable reference types](#). The nullable value types are available beginning with C# 2.

Any nullable value type is an instance of the generic [System.Nullable<T>](#) structure. You can refer to a nullable value type with an underlying type T in any of the following interchangeable forms: `Nullable<T>` or $T?$.

You typically use a nullable value type when you need to represent the undefined value of an underlying value type. For example, a Boolean, or `bool`, variable can only be either `true` or `false`. However, in some applications a variable value can be undefined or missing. For example, a database field may contain `true` or `false`, or it may contain no value at all, that is, `NULL`. You can use the `bool?` type in that scenario.

Declaration and assignment



As a value type is implicitly convertible to the corresponding nullable value type, you can assign a value to a variable of a nullable value type as you would do that for its underlying value type. You can also assign the `null` value. For example:

C#	 Copy
<pre>double? pi = 3.14; char? letter = 'a'; int m2 = 10; int? m = m2; bool? flag = null; // An array of a nullable value type: int?[] arr = new int?[10];</pre>	

The default value of a nullable value type represents `null`, that is, it's an instance whose `Nullable<T>.HasValue` property returns `false`.

Examination of an instance of a nullable value type



Beginning with C# 7.0, you can use the [is operator with a type pattern](#) to both examine an instance of a nullable value type for `null` and retrieve a value of an underlying type:

C#	 Copy	 Run
<pre>int? a = 42; if (a is int valueOfA) { Console.WriteLine(\$"a is {valueOfA}"); } else { Console.WriteLine("a does not have a value"); } // Output: // a is 42</pre>		



You always can use the following read-only properties to examine and get a value of a nullable value type variable:

- `Nullable<T>.HasValue` indicates whether an instance of a nullable value type has a value of its underlying type.
- `Nullable<T>.Value` gets the value of an underlying type if `HasValue` is `true`. If `HasValue` is `false`, the `Value` property throws an `InvalidOperationException`.

The following example uses the `HasValue` property to test whether the variable contains a value before displaying it:

C#	 Copy	 Run
<pre>int? b = 10; if (b.HasValue) { Console.WriteLine(\$"b is {b.Value}"); } else { Console.WriteLine("b does not have a value"); } // Output: // b is 10</pre>		



You can also compare a variable of a nullable value type with `null` instead of using the `HasValue` property, as the following example shows:

C#	 Copy	 Run
<pre>int? c = 7; if (c != null) { Console.WriteLine(\$"c is {c.Value}"); } else { Console.WriteLine("c does not have a value"); } // Output: // c is 7</pre>		

Conversion from a nullable value type to an underlying type


If you want to assign a value of a nullable value type to a non-nullable value type variable, you might need to specify the value to be assigned in place of `null`. Use the

[null-coalescing operator ??](#) to do that (you can also use the `Nullable<T>.GetValueOrDefault(T)` method for the same purpose):

C#	 Copy	 Run
<pre>int? a = 28; int b = a ?? -1; Console.WriteLine(\$"b is {b}"); // output: b is 28 int? c = null; int d = c ?? -1; Console.WriteLine(\$"d is {d}"); // output: d is -1</pre>		

If you want to use the [default](#) value of the underlying value type in place of `null`, use the `Nullable<T>.GetValueOrDefault()` method.

You can also explicitly cast a nullable value type to a non-nullable type, as the following example shows:


C#	 Copy
<pre>int? n = null; //int m1 = n; // Doesn't compile int n2 = (int)n; // Compiles, but throws an exception if n is null</pre>	

At run time, if the value of a nullable value type is `null`, the explicit cast throws an [InvalidOperationException](#).

A non-nullable value type `T` is implicitly convertible to the corresponding nullable value type `T?`.

Lifted operators

The predefined unary and binary [operators](#) or any overloaded operators that are supported by a value type `T` are also supported by the corresponding nullable value type `T?`. These operators, also known as *lifted operators*, produce `null` if one or both operands are `null`; otherwise, the operator uses the contained values of its operands to calculate the result. For example:

C#	 Copy
<pre>int? a = 10; int? b = null; int? c = 10;</pre>	

```
a++;          // a is 11
a = a * c;    // a is 110
a = a + b;    // a is null
```

ⓘ Note

For the `bool?` type, the predefined `&` and `|` operators don't follow the rules described in this section: the result of an operator evaluation can be non-null even if one of the operands is `null`. For more information, see the [Nullable Boolean logical operators](#) section of the [Boolean logical operators](#) article.

For the [comparison operators](#) `<`, `>`, `<=`, and `>=`, if one or both operands are `null`, the result is `false`; otherwise, the contained values of operands are compared. Do not assume that because a particular comparison (for example, `<=`) returns `false`, the opposite comparison (`>`) returns `true`. The following example shows that 10 is

- neither greater than or equal to `null`
- nor less than `null`

C#

 Copy

 Run

```
int? a = 10;
Console.WriteLine($"{a} >= null is {a >= null}");
Console.WriteLine($"{a} < null is {a < null}");
Console.WriteLine($"{a} == null is {a == null}");
// Output:
// 10 >= null is False
// 10 < null is False
// 10 == null is False

int? b = null;
int? c = null;
Console.WriteLine($"null >= null is {b >= c}");
Console.WriteLine($"null == null is {b == c}");
// Output:
// null >= null is False
// null == null is True
```

For the [equality operator](#) `==`, if both operands are `null`, the result is `true`, if only one of the operands is `null`, the result is `false`; otherwise, the contained values of operands are compared.

For the [inequality operator](#) `!=`, if both operands are `null`, the result is `false`, if only one of the operands is `null`, the result is `true`; otherwise, the contained values of

operands are compared.



If there exists a [user-defined conversion](#) between two value types, the same conversion can also be used between the corresponding nullable value types.

Boxing and unboxing

An instance of a nullable value type $T?$ is [boxed](#) as follows:



- If [HasValue](#) returns `false`, the null reference is produced.
- If [HasValue](#) returns `true`, the corresponding value of the underlying value type T is boxed, not the instance of [Nullable<T>](#).

You can unbox a boxed value of a value type T to the corresponding nullable value type $T?$, as the following example shows:

C#	 Copy	 Run
<pre>int a = 41; object aBoxed = a; int? aNullable = (int?)aBoxed; Console.WriteLine(\$"Value of aNullable: {aNullable}"); object aNullableBoxed = aNullable; if (aNullableBoxed is int valueOfA) { Console.WriteLine(\$"aNullableBoxed is boxed int: {valueOfA}"); } // Output: // Value of aNullable: 41 // aNullableBoxed is boxed int: 41</pre>		

How to identify a nullable value type



The following example shows how to determine whether a [System.Type](#) instance represents a constructed nullable value type, that is, the [System.Nullable<T>](#) type with a specified type parameter T :

C#	 Copy	 Run
<pre>Console.WriteLine(\$"int? is {(IsNullable(typeof(int?)) ? "nullable" : "non-nullable")} value type"); Console.WriteLine(\$"int is {(IsNullable(typeof(int)) ? "nullable" : "non-nullable")} value type"); bool IsNullable(Type type) => Nullable.GetUnderlyingType(type) != null;</pre>		



```
// Output:  
// int? is nullable value type  
// int is non-nullable value type
```

As the example shows, you use the [typeof](#) operator to create a [System.Type](#) instance.

If you want to determine whether an instance is of a nullable value type, don't use the [Object.GetType](#) method to get a [Type](#) instance to be tested with the preceding code. When you call the [Object.GetType](#) method on an instance of a nullable value type, the instance is [boxed](#) to [Object](#). As boxing of a non-null instance of a nullable value type is equivalent to boxing of a value of the underlying type, [GetType](#) returns a [Type](#) instance that represents the underlying type of a nullable value type:

C#	 Copy	 Run
<pre>int? a = 17; Type typeOfA = a.GetType(); Console.WriteLine(typeOfA.FullName); // Output: // System.Int32</pre>		

Also, don't use the [is](#) operator to determine whether an instance is of a nullable value type. As the following example shows, you cannot distinguish types of a nullable value type instance and its underlying type instance with the [is](#) operator:

C#	 Copy	 Run
<pre>int? a = 14; if (a is int) { Console.WriteLine("int? instance is compatible with int"); } int b = 17; if (b is int?) { Console.WriteLine("int instance is compatible with int?"); } // Output: // int? instance is compatible with int // int instance is compatible with int?</pre>		

Instead use the [Nullable.GetUnderlyingType](#) from the first example and [typeof](#) operator to check if an instance is of a nullable value type.

ⓘ Note

The methods described in this section are not applicable in the case of **nullable reference types**.

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Nullable types](#)
- [Lifted operators](#)
- [Implicit nullable conversions](#)
- [Explicit nullable conversions](#)
- [Lifted conversion operators](#)

See also

- [C# reference](#)
- [What exactly does 'lifted' mean?](#)
- [System.Nullable<T>](#)
- [System.Nullable](#)
- [Nullable.GetUnderlyingType](#)
- [Nullable reference types](#)