Contents

Nullable types (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Nullable types are instances of the System.Nullable<T> struct. Nullable types can represent all the values of an underlying type `T`, and an additional null value. The underlying type `T` can be any non-nullable value type. `T` cannot be a reference type.

For example, you can assign `null` or any integer value from Int32.MinValue to Int32.MaxValue to a `Nullable<int>` and true, false, or `null` to a `Nullable<bool>`.

You use a nullable type when you need to represent the undefined value of an underlying type. A Boolean variable can have only two values: true and false. There is no "undefined" value. In many programming applications, most notably database interactions, a variable value can be undefined or missing. For example, a field in a database may contain the values true or false, or it may contain no value at all. You use a `Nullable<bool>` type in that case.

Nullable types have the following characteristics:

- Nullable types represent value-type variables that can be assigned the `null` value. You cannot create a nullable type based on a reference type. (Reference types already support the `null` value.)
- The syntax `T?` is shorthand for `Nullable<T>`. The two forms are interchangeable.
- Assign a value to a nullable type just as you would for an underlying value type: `int? x = 10;` or `double? d = 4.108;`. You also can assign the `null` value: `int? x = null;`.
- Use the Nullable<T>.HasValue and Nullable<T>.Value readonly properties to test for null and retrieve the value, as shown in the following example: `if (x.HasValue) y = x.Value;`
  - The HasValue property returns `true` if the variable contains a value, or `false` if it's `null`.
  - The Value property returns a value if HasValue returns `true`. Otherwise, an InvalidOperationException is thrown.
- You can also use the `==` and `!=` operators with a nullable type, as shown in the following example: `if (x != null) y = x.Value;`. If `a` and `b` are both null, `a == b` evaluates to `true`.
- Beginning with C# 7.0, you can use pattern matching to both examine and get a value of a nullable type: `if (x is int valueOfX) y = valueOfX;`.
- The default value of `T?` is an instance whose HasValue property returns `false`.
- Use the GetValueOrDefault() method to return either the assigned value, or the default value of the underlying value type if the value of the nullable type is `null`.
- Use the GetValueOrDefault(T) method to return either the assigned value, or the provided default value if the value of the nullable type is `null`.
- Use the null-coalescing operator, `??`, to assign a value to an underlying type based on a value of the nullable type: `int? x = null; int y = x ?? -1;`. In the example, since `x` is null, the result value of `y` is `-1`.
- If a user-defined conversion is defined between two data types, the same conversion can also be used with the nullable versions of these data types.
- Nested nullable types are not allowed. The following line doesn't compile: `Nullable<Nullable<int>> n;`

For more information, see the Using nullable types and How to: Identify a nullable type topics.

See also

- System.Nullable<T>
- System.Nullable
- ?? Operator
- C# Programming Guide
- C# Guide
- C# Reference
- Nullable Value Types (Visual Basic)

Using nullable types (C# Programming Guide)

1/23/2019 • 5 minutes to read • Edit Online

Nullable types are types that represent all the values of an underlying value type `T`, and an additional null value. For more information, see the Nullable types topic.

You can refer to a nullable type in any of the following forms: `Nullable<T>` or `T?`. These two forms are interchangeable.

Declaration and assignment

As a value type can be implicitly converted to the corresponding nullable type, you assign a value to a nullable type as you would for its underlying value type. You also can assign the `null` value. For example:

```
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

// Array of nullable type:
int?[] arr = new int?[10];
```

Examination of a nullable type value

Use the following readonly properties to examine an instance of a nullable type for null and retrieve a value of an underlying type:

- Nullable<T>.HasValue indicates whether an instance of a nullable 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 code in the following example uses the `HasValue` property to test whether the variable contains a value before displaying it:

```
int? x = 10;
if (x.HasValue)
{
    Console.WriteLine($"x is {x.Value}");
}
else
{
    Console.WriteLine("x does not have a value");
}
```

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

```
int? y = 7;
if (y != null)
{
    Console.WriteLine($"y is {y.Value}");
}
else
{
    Console.WriteLine("y does not have a value");
}
```

Beginning with C# 7.0, you can use pattern matching to both examine and get a value of a nullable type:

```
int? z = 42;
if (z is int valueOfZ)
{
    Console.WriteLine($"z is {valueOfZ}");
}
else
{
    Console.WriteLine("z does not have a value");
}
```

Conversion from a nullable type to an underlying type

If you need to assign a nullable type value to a non-nullable type, use the null-coalescing operator `??` to specify the value to be assigned if a nullable type value is null (you also can use the Nullable<T>.GetValueOrDefault(T) method to do that):

```
int? c = null;

// d = c, if c is not null, d = -1 if c is null.
int d = c ?? -1;
Console.WriteLine($"d is {d}");
```

Use the Nullable<T>.GetValueOrDefault() method if the value to be used when a nullable type value is null should be the default value of the underlying value type.

You can explicitly cast a nullable type to a non-nullable type. For example:

```
int? n = null;

//int m1 = n;    // Doesn't compile.
int n2 = (int)n; // Compiles, but throws an exception if n is null.
```

At run time, if the value of a nullable type is null, the explicit cast throws an InvalidOperationException.

A non-nullable value type is implicitly converted to the corresponding nullable type.

Operators

The predefined unary and binary operators and any user-defined operators that exist for value types may also be used by nullable types. These operators produce a null value if one or both operands are null; otherwise, the operator uses the contained values to calculate the result. For example:

```
int? a = 10;
int? b = null;
int? c = 10;

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

For the relational operators (`<` , `>` , `<=` , `>=`), if one or both operands are null, the result is `false` . 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.

```
int? num1 = 10;
int? num2 = null;
if (num1 >= num2)
{
    Console.WriteLine("num1 is greater than or equal to num2");
}
else
{
    Console.WriteLine("num1 >= num2 is false (but num1 < num2 also is false)");
}

if (num1 < num2)
{
    Console.WriteLine("num1 is less than num2");
}
else
{
    Console.WriteLine("num1 < num2 is false (but num1 >= num2 also is false)");
}

if (num1 != num2)
{
    Console.WriteLine("num1 != num2 is true!");
}

num1 = null;
if (num1 == num2)
{
    Console.WriteLine("num1 == num2 is true if the value of each is null");
}
// Output:
// num1 >= num2 is false (but num1 < num2 also is false)
// num1 < num2 is false (but num1 >= num2 also is false)
// num1 != num2 is true!
// num1 == num2 is true if the value of each is null
```

The above example also shows that an equality comparison of two nullable types that are both null evaluates to `true` .

Boxing and unboxing

A nullable value type is boxed by the following rules:

- If HasValue returns `false` , the null reference is produced.
- If HasValue returns `true` , a value of the underlying value type `T` is boxed, not the instance of Nullable<T>.

You can unbox the boxed value type to the corresponding nullable type, as the following example shows:

```
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
```

The bool? type

The `bool?` nullable type can contain three different values: true, false, and null. The `bool?` type is like the Boolean variable type that is used in SQL. To ensure that the results produced by the `&` and `|` operators are consistent with the three-valued Boolean type in SQL, the following predefined operators are provided:

- `bool? operator &(bool? x, bool? y)`
- `bool? operator |(bool? x, bool? y)`

The semantics of these operators is defined by the following table:

X	Y	X&Y	X|Y
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	null	true
null	false	false	null
null	null	null	null

Note that these two operators don't follow the rules described in the Operators section: the result of an operator evaluation can be non-null even if one of the operands is null.

See also

- Nullable types
- C# Programming Guide
- What exactly does 'lifted' mean?

How to: Identify a nullable type (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

The following example shows how to determine whether a System.Type instance represents a closed generic nullable type, that is, the System.Nullable<T> type with a specified type parameter `T`:

```
Console.WriteLine($"int? is {(IsNullable(typeof(int?)) ? "nullable" : "non nullable")} type");
Console.WriteLine($"int is {(IsNullable(typeof(int)) ? "nullable" : "non nullable")} type");

bool IsNullable(Type type) => Nullable.GetUnderlyingType(type) != null;

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

As the example shows, you use the typeof operator to create a System.Type object.

If you want to determine whether an instance is of a nullable 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 type, the instance is boxed to Object. As boxing of a non-null instance of a nullable type is equivalent to boxing of a value of the underlying type, GetType returns a Type object that represents the underlying type of a nullable type:

```
int? a = 17;
Type typeOfA = a.GetType();
Console.WriteLine(typeOfA.FullName);
// Output:
// System.Int32
```

Don't use the is operator to determine whether an instance is of a nullable type. As the following example shows, you cannot distinguish types of instances of a nullable type and its underlying type with using the `is` operator:

```
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?
```

You can use the code presented in the following example to determine whether an instance is of a nullable type:

```
int? a = 14;
int b = 17;
if (IsOfNullableType(a) && !IsOfNullableType(b))
{
    Console.WriteLine("int? a is of a nullable type, while int b -- not");
}

bool IsOfNullableType<T>(T o)
{
    var type = typeof(T);
    return Nullable.GetUnderlyingType(type) != null;
}

// Output:
// int? a is of a nullable type, while int b -- not
```

See also

- Nullable types
- Using nullable types
- GetUnderlyingType