

Use Stack Overflow for Teams at work to find answers in a private and secure environment. Get your first 10 users free. [Sign up.](#)

How to check if an object is nullable?

Asked 10 years, 9 months ago Active 5 months ago Viewed 87k times



How do I check if a given object is nullable in other words how to implement the following method...

189



```
bool IsNullableValueType(object o)
{
    ...
}
```



49

EDIT: I am looking for nullable value types. I didn't have ref types in mind.

*//Note: This is just a sample. The code has been simplified
//to fit in a post.*

```
public class BoolContainer
{
    bool? myBool = true;
}

var bc = new BoolContainer();

const BindingFlags bindingFlags = BindingFlags.Public
    | BindingFlags.NonPublic
    | BindingFlags.Instance
    ;

object obj;
object o = (object)bc;

foreach (var fieldInfo in o.GetType().GetFields(bindingFlags))
{
    obj = (object)fieldInfo.GetValue(o);
}
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

So now as a work around I decided to check if o is nullable and create a nullable wrapper around obj.

c#

.net

nullable

edited Apr 7 '10 at 9:53



Josh Lee

126k

26

227

249

asked Dec 17 '08 at 14:17



Autodidact

17.7k

14

57

77

Should the code include strings as being nullable? They are a non-generic ValueType which appears to be nullable. Or are they not a ValueType? – [TamusJRoyce](#) May 2 '12 at 21:23

String is not a ValueType. It is a Reference type. – [Suncat2000](#) Mar 22 '13 at 13:09

This is a really good question! The 'Type.IsNullableType()' is kind of deceiving because it actually only checks for the type being a 'Nullable<T>', which didn't return expected results if you actually wanted to check for any types that can accept a null value (e.g. I tried to use with a.IsNullableType(), where 'a' was a 'typeof(string)' determined at runtime) – [ErrCode](#) Nov 7 '18 at 8:35

Answer is in fieldInfo.FieldType: check if FieldType is generic and generic type is of Nullable<> type. (Example: if (FieldType.IsGenericType && FieldType.GetGenericTypeDefinition() == typeof(Nullable<>))). Do not try to get obj.GetType() it will have UndelyingSystemType of Nullable<T> variable T (in your case of Boolean type, instead of Nullable<Boolean>), it's a boxing problem. – [SoLaR](#) Aug 21 at 8:46

14 Answers



252



There are two types of nullable - `Nullable<T>` and reference-type.

Jon has corrected me that it is hard to get type if boxed, but you can with generics: - so how about below. This is actually testing type `T`, but using the `obj` parameter purely for generic type inference (to make it easy to call) - it would work almost identically without the `obj` param, though.

```
static bool IsNullable<T>(T obj)
{
    if (obj == null) return true; // obvious
    Type type = typeof(T);
    if (!type.IsValueType) return true; // ref-type
    if (Nullable.GetUnderlyingType(type) != null) return true; // Nullable<T>
    return false; // value-type
}
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

but this won't work so well if you have already boxed the value to an object variable.

edited Mar 10 '12 at 23:07



Abel

43.9k

18

119

204

answered Dec 17 '08 at 14:20



Marc Gravell ♦

822k

212

2224

2623

- 7 The last line is only valid if you somehow manage to get a boxed `Nullable<T>` instead of boxing straight to `T`. It's possible, but tricky to achieve from what I remember. – [Jon Skeet](#) Dec 17 '08 at 14:22

This code was helpful for me, not because I got a boxed `Nullable<T>` but because I was writing a generic WPF converter base class and some properties are nullable, so I used `Nullable.GetUnderlyingType` to detect that case and `Activator.CreateInstance` to make a boxed nullable, (`Convert.ChangeType` doesn't handle nullables btw). – [Qwertie](#) Jun 9 '11 at 17:43 ✎

- 1 @Abel if you mean re his edit to clarify that he hadn't considered reference types, I *think* my answer predated that edit; the reader can make their own decision there, based on their own needs, I suspect (confirmed: his comment re ref-types as added at 14:42; my answer was all <= 14:34) – [Marc Gravell](#) ♦ Mar 10 '12 at 23:56 ✎

- 1 Will `(obj == null)` throw an exception when `obj = 1` ? – [Qi Fan](#) Mar 27 '12 at 22:14

- 2 @JustinMorgan If `T` is a generic parameter constrained by `T : struct`, then `T` is not allowed to be `Nullable<>`, so you need no check in that case! I know the type `Nullable<>` is a struct, but in C# the constraint where `T : struct` specifically exclude nullable value-types. The spec says: "Note that although classified as a value type, a nullable type (§4.1.10) does not satisfy the value type constraint." – [Jeppe Stig Nielsen](#) Dec 3 '12 at 14:07 ✎

There is a very simple solution using method overloads

47

<http://deanchalk.com/is-it-nullable/>

excerpt:

```
public static class ValueTypeHelper
{
    public static bool IsNullable<T>(T t) { return false; }
    public static bool IsNullable<T>(T? t) where T : struct { return true; }
}
```

then

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

```

int a = 123;
int? b = null;
object c = new object();
object d = null;
int? e = 456;
var f = (int?)789;
bool result1 = ValueHelper.IsNullable(a); // false
bool result2 = ValueHelper.IsNullable(b); // true
bool result3 = ValueHelper.IsNullable(c); // false
bool result4 = ValueHelper.IsNullable(d); // false
bool result5 = ValueHelper.IsNullable(e); // true
bool result6 = ValueHelper.IsNullable(f); // true

```

edited Feb 6 '15 at 7:41

answered Nov 9 '10 at 8:50



Dean Chalk

15.9k 4 45 80

- 6 plus one for you sir for adding test cases. I've used those test cases for checking all the other answers. More people should go this extra bit. – [Marty Neal](#) Jan 7 '11 at 21:06
- 4 For what it's worth, this doesn't work in VB.NET. It results in a compiler error of "Overload resolution failed because no accessible 'IsNullable' is most specific for these arguments" in all situations where `True` would be returned. – [ckittel](#) Aug 23 '11 at 18:39
- 1 I really like this solution - and it is a shame VB cannot handle it. I tried working around with `ValueType` but ran into trouble with VB compiler being inconsistent about which overload to use based on whether it was called as a shared method or an extension, I even raised a question about this as it seems weird: stackoverflow.com/questions/12319591/... – [James Close](#) Sep 7 '12 at 14:03
- 19 You're checking the **compile-time** type, but it's already obvious (from intellisense) if the compile-time type is nullable (`System.Nullable<T>`). If you say `object g = e;` and then `ValueHelper.IsNullable(g)`, what do you expect to obtain? – [Jeppe Stig Nielsen](#) Dec 3 '12 at 14:36
- 17 I just verified; **this does not work**, as Jeppe said. If the variables are cast to `object`, it will always return false. So you cannot determine the type of an unknown object at runtime this way. The only time this works is if the type is fixed at compile-time, and in that case you do not need a runtime check at all. – [HugoRune](#) Jul 2 '13 at 13:59

▲
30

The question of "How to check if a type is nullable?" is actually "How to check if a type is `Nullable<T>`?", which can be generalized to "How to check if a type is a constructed type of some generic type?", so that it not only answers the question "Is `Nullable<int>` a `Nullable<T>`?", but also "Is `List<int>` a `List<T>`?".

Most of the provided solution use the `Nullable.GetUnderlyingType()` method, which will obviously only work with the case of `Nullable<T>`.

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

To check if a type is some form of `Nullable<>` using reflection, you first have to convert your constructed generic type, for example `Nullable<int>`, into the generic type definition, `Nullable<>`. You can do that by using the `GetGenericTypeDefinition()` method of the `Type` class. You can then compare the resulting type to `Nullable<>`:

```
Type typeToTest = typeof(Nullable<int>);
bool isNullable = typeToTest.GetGenericTypeDefinition() == typeof(Nullable<>);
// isNullable == true
```

The same can be applied to any generic type:

```
Type typeToTest = typeof(List<int>);
bool isList = typeToTest.GetGenericTypeDefinition() == typeof(List<>);
// isList == true
```

Several types may seem the same, but a different number of type arguments means it's a completely different type.

```
Type typeToTest = typeof(Action<DateTime, float>);
bool isAction1 = typeToTest.GetGenericTypeDefinition() == typeof(Action<>);
bool isAction2 = typeToTest.GetGenericTypeDefinition() == typeof(Action<,>);
bool isAction3 = typeToTest.GetGenericTypeDefinition() == typeof(Action<,,>);
// isAction1 == false
// isAction2 == true
// isAction3 == false
```

Since `Type` objects are instantiated once per type, you can check for reference equality between them. So if you want to check if two objects are of the same generic type definition, you can write:

```
var listOfInts = new List<int>();
var listOfStrings = new List<string>();

bool areSameGenericType =
    listOfInts.GetType().GetGenericTypeDefinition() ==
    listOfStrings.GetType().GetGenericTypeDefinition();
// areSameGenericType == true
```

If you'd like to check if an object is nullable, rather than a type, then you can use the above technique together with Marc Gravell's

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

```
static bool IsNullable<T>(T obj)
{
    if (!typeof(T).IsGenericType)
        return false;

    return typeof(T).GetGenericTypeDefinition() == typeof(Nullable<>);
}
```

edited Jan 3 '11 at 12:16

answered Jan 3 '11 at 11:58



Allon Guralnek

13.2k 5 49 90

@ AllonGuralnek There is simplified version down there in my answer. I wanted to make it as edit and as my reputation is not your level, it would be edit without my name on your answer, even so, it seems that review is always shooting me into leg, that it is adresing author even if it was not. Strange world, some people do not get definitions:). – [ipavlu](#) Nov 10 '15 at 20:01

@ipavlu: Your version is not simplified, it's in fact more complicated. I think you mean it's optimized since you cache the result. That makes it more difficult to understand. – [Allon Guralnek](#) Nov 10 '15 at 21:36

@ AllonGuralnek static generic class and static one time initialized fields, that is complicated? Dear God, I made terrible crime :). – [ipavlu](#) Nov 11 '15 at 0:48

@ipavku: Yes, because it has nothing to do with the question "How to check if an object is nullable?". I try to keep it simple and to the point, and I avoid introducing unneeded and unrelated concepts. – [Allon Guralnek](#) Nov 11 '15 at 5:36


- 1 @nawfal: If I understood you correctly, your questing my implementation in the face of the existence of `Nullable.GetUnderlyingType()` that already provided by the framework. Why not just use the method in the framework? Well, you should. It is clearer, more concise and better tested. But in my post I'm trying to teach how to use reflection to get the information you want, so that someone can apply it to any type (by replacing `typeof(Nullable<>)` with any other type). If you look at the sources of `GetUnderlyingType()` (original or decompiled), you'll see it is very similar to my code. – [Allon Guralnek](#) Jul 2 '16 at 19:05

▲ This works for me and seems simple:

26

```
static bool IsNullable<T>(T obj)
{
    return default(T) == null;
}
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

-
- 7 For what it's worth, this is also the test [used by Microsoft](#) – [canton7](#) May 6 '15 at 9:29
-
- 1 Nice... Is this not the top answer cause it came later? I find the top answer so confusing. – [Vincent Buscarello](#) Oct 15 '18 at 19:10
-
- It isn't the top answer because of up vote count. – [Erik](#) Mar 27 at 18:57
-
- 1 This should be the top answer. After days of trying different methods I randomly thought of this solution, tried it, and it seems to be working perfectly (compared to the top-rated answer) – [user3163495](#) Apr 26 at 17:27 
-
- 1 Upvote. Common guys, only 226 more votes needed! – [Etienne Charland](#) Jun 7 at 6:57
-

Well, you could use:

20 `return !(o is ValueType);`

... but an object itself isn't nullable or otherwise - a *type* is. How were you planning on using this?

answered Dec 17 '08 at 14:21



[Jon Skeet](#)

1132k 716 8199
8587

-
- 2 This threw me off a bit. e.g. `int? i = 5; typeof(i)` returns `System.Int32` instead of `Nullable<Int32>` -- `typeof(int?)` returns `Nullable<Int32>`.. where can I get some clarity on this topic? – [Gishu](#) Feb 12 '09 at 11:39
-
- 2 `typeof(i)` will give a compiler error- you can't use `typeof` with a variable. What did you actually do? – [Jon Skeet](#) Feb 12 '09 at 11:47
-
- 15 `i.GetType()` will box to `Object` first, and there's no such thing as a boxed nullable type - `Nullable<int>` gets boxed to a null reference or a boxed int. – [Jon Skeet](#) Feb 12 '09 at 14:52
-
- That way is better than `Nullable.GetUnderlyingType(type) != null` ? – [Kiquenet](#) Nov 28 '13 at 13:27
-
- @Kiquenet: We don't *have* the type here - just the value. – [Jon Skeet](#) Nov 28 '13 at 13:28
-

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

11

```

public bool IsNullable(object obj)
{
    Type t = obj.GetType();
    return t.IsGenericType
        && t.GetGenericTypeDefinition() == typeof(Nullable<>);
}

```

answered Mar 15 '12 at 2:10



CARLOS LOTH

3,352 2 32 40

+1. Excellent solution for boxed null-able types. I haven't tested this specifically yet. So if anyone else can verify, it would be appreciated. – [TamusJRoyce](#) May 2 '12 at 21:20

I have already tested it. I had to created a kind of `Nullable` type, but with different semantics. In my situation I should support `null` as a valid value and also support no value at all. So a created an `Optional` type. As it was necessary to support `null` values, I also had to implement code for handling `Nullable` values as part of my implementation. That is where this code came from. – [CARLOS LOTH](#) May 8 '12 at 13:52

9 I think this solution is wrong. Passing a `Nullable` value type as an argument to a method expecting a parameter of type `object` should cause boxing to occur. `Nullable` is a value type and the result of boxing conversion is a reference type. There are no boxed nullables. I believe this method always returns false? – [Mishax](#) Nov 17 '12 at 8:34

1 Any test about it like another answers ? – [Kiquenet](#) Nov 28 '13 at 13:36

5 It doesn't work because of boxing value. It will always return `FALSE`. – [N Rocking](#) Apr 4 '14 at 16:21



There are two issues here: 1) testing to see whether a `Type` is nullable; and 2) testing to see whether an object represents a nullable `Type`.

10

For issue 1 (testing a `Type`), here's a solution I've used in my own systems: [TypeIsNullable-check solution](#)

For issue 2 (testing an object), Dean Chalk's solution above works for value types, but it doesn't work for reference types, since using the `<T>` overload always returns false. Since reference types are inherently nullable, testing a reference type should always return true. Please see the note [About "nullability"] below for an explanation of these semantics. Thus, here's my modification to Dean's approach:

```

public static bool IsObjectNullable<T>(T obj)
{
    return obj == null || obj.GetType().IsValueType;
}

```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).


```

        // Since the object passed is a ValueType, and it is not null, it cannot be a
        nullable object
        return false;
    }

    public static bool IsObjectNullable<T>(T? obj) where T : struct
    {
        // Always return true, since the object-type passed is guaranteed by the
        compiler to always be nullable
        return true;
    }

```

And here's my modification to the client-test code for the above solution:

```

int a = 123;
int? b = null;
object c = new object();
object d = null;
int? e = 456;
var f = (int?)789;
string g = "something";

bool isnullable = IsObjectNullable(a); // false
isnullable = IsObjectNullable(b); // true
isnullable = IsObjectNullable(c); // true
isnullable = IsObjectNullable(d); // true
isnullable = IsObjectNullable(e); // true
isnullable = IsObjectNullable(f); // true
isnullable = IsObjectNullable(g); // true

```

The reason I've modified Dean's approach in `IsObjectNullable<T>(T t)` is that his original approach always returned false for a reference type. Since a method like `IsObjectNullable` should be able to handle reference-type values and since all reference types are inherently nullable, then if either a reference type or a null is passed, the method should always return true.

The above two methods could be replaced with the following single method and achieve the same output:

```

public static bool IsObjectNullable<T>(T obj)
{
    Type argType = typeof(T);
    if (!argType.IsValueType || obj == null)

```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

However, the problem with this last, single-method approach is that performance suffers when a `Nullable<T>` parameter is used. It takes much more processor time to execute the last line of this single method than it does to allow the compiler to choose the second method overload shown previously when a `Nullable<T>`-type parameter is used in the `IsObjectNullable` call. Therefore, the optimum solution is to use the two-method approach illustrated here.

CAVEAT: This method works reliably only if called using the original object reference or an exact copy, as shown in the examples. However, if a nullable object is boxed to another `Type` (such as `object`, etc.) instead of remaining in its original `Nullable<>` form, this method will not work reliably. If the code calling this method is not using the original, unboxed object reference or an exact copy, it cannot reliably determine the object's nullability using this method.

In most coding scenarios, to determine nullability one must instead rely on testing the original object's `Type`, not its reference (e.g., code must have access to the object's original `Type` to determine nullability). In these more common cases, `IsTypeNullable` (see link) is a reliable method of determining nullability.

P.S. - About "nullability"

I should repeat a statement about nullability I made in a separate post, which applies directly to properly addressing this topic. That is, I believe the focus of the discussion here should not be how to check to see if an object is a generic `Nullable` type, but rather whether one can assign a value of null to an object of its type. In other words, I think we should be determining whether an object type is nullable, not whether it is `Nullable`. The difference is in semantics, namely the practical reasons for determining nullability, which is usually all that matters.

In a system using objects with types possibly unknown until run-time (web services, remote calls, databases, feeds, etc.), a common requirement is to determine whether a null can be assigned to the object, or whether the object might contain a null. Performing such operations on non-nullable types will likely produce errors, usually exceptions, which are very expensive both in terms of performance and coding requirements. To take the highly-preferred approach of proactively avoiding such problems, it is necessary to determine whether an object of an arbitrary `Type` is capable of containing a null; i.e., whether it is generally 'nullable'.

In a very practical and typical sense, nullability in .NET terms does not at all necessarily imply that an object's `Type` is a form of `Nullable`. In many cases in fact, objects have reference types, can contain a null value, and thus are all nullable; none of these have a `Nullable` type. Therefore, for practical purposes in most scenarios, testing should be done for the general concept of nullability, vs. the implementation-dependent concept of `Nullable`. So we should not be hung up by focusing solely on the .NET `Nullable` type but rather incorporate our understanding of its requirements and behavior in the process of focusing on the general, practical concept of nullability.

edited May 23 '17 at 11:47

answered Oct 14 '11 at 0:27

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).



8

The simplest solution I came up with is to implement Microsoft's solution ([How to: Identify a Nullable Type \(C# Programming Guide\)](#)) as an extension method:



```
public static bool IsNullable(this Type type)
{
    return Nullable.GetUnderlyingType(type) != null;
}
```

This can then be called like so:

```
bool isNullable = typeof(int).IsNullable();
```

This also seems a logical way to access `IsNullable()` because it fits in with all of the other `IsXxxx()` methods of the `Type` class.

edited Apr 1 at 8:09

answered Dec 22 '16 at 11:00



[sclarke81](#)

1,386 13 20

1 Didn't you want to use "==" instead of "!=" ? – [vkelman](#) Mar 29 at 17:02

Good spot @vkelman Instead of making that change I've updated the answer to use the current suggestion from Microsoft as this has changed since I wrote this. – [sclarke81](#) Apr 1 at 8:11

Yes, I saw that Microsoft changed its code. – [vkelman](#) Apr 2 at 14:37



6

Be carefull, when boxing a nullable type (`Nullable<int>` or `int?` for instance) :



```
int? nullValue = null;
object boxedNullValue = (object)nullValue;
Debug.Assert(boxedNullValue == null);

int? value = 10;
object boxedValue = (object)value;
Debug.Assert(boxedValue.GetType() == typeof(int))
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

edited Dec 17 '08 at 16:42

answered Dec 17 '08 at 16:10



thinkbeforecoding

6,065 1 25 31

3

Maybe a little bit off topic, but still some interesting information. I find a lot of people that use `Nullable.GetUnderlyingType() != null` to identity if a type is nullable. This obviously works, but Microsoft advises the following `type.IsGenericType &&`

`type.GetGenericTypeDefinition() == typeof(Nullable<>)` (see <http://msdn.microsoft.com/en-us/library/ms366789.aspx>).

I looked at this from a performance side of view. The conclusion of the test (one million attempts) below is that when a type is a nullable, the Microsoft option delivers the best performance.

Nullable.GetUnderlyingType(): 1335ms (3 times slower)

GetGenericTypeDefinition() == typeof(Nullable<>): 500ms

I know that we are talking about a small amount of time, but everybody loves to tweak the milliseconds :-)! So if your boss wants you to reduce some milliseconds then this is your saviour...

```
/// <summary>Method for testing the performance of several options to determine if a
type is nullable</summary>
[TestMethod]
public void IdentityNullablePerformanceTest()
{
    int attempts = 1000000;

    Type nullableType = typeof(Nullable<int>);

    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();
    for (int attemptIndex = 0; attemptIndex < attempts; attemptIndex++)
    {
        Assert.IsTrue(Nullable.GetUnderlyingType(nullableType) != null, "Expected to be
a nullable");
    }

    Console.WriteLine("Nullable.GetUnderlyingType(): {0} ms",
stopwatch.ElapsedMilliseconds);
}
```

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

```

        Assert.IsTrue(nullableType.IsGenericType &&
nullableType.GetGenericTypeDefinition() == typeof(Nullable<>), "Expected to be a
nullable");
    }

    Console.WriteLine("GetGenericTypeDefinition() == typeof(Nullable<>): {0} ms",
stopwatch.ElapsedMilliseconds);
    stopwatch.Stop();
}

```

answered Jul 4 '14 at 15:32



Roel van Megen

181 1 4

- 1 Hi, there is probably one issue with measuring time, the Assert can affect results. Have you tested without Assert? Also Console.WriteLine should be outside metered area. +1 for an attempt to quantify performance issues:). – [ipavlu](#) Nov 11 '15 at 1:19

@ipavlu Console.WriteLine is indeed outside metered area ;) – [nawfal](#) Jul 2 '16 at 12:31

Roel, as ipavlu has mentioned, Assert should be outside the loop. Secondly, you should also test it against non-nullable as well to test for false cases. I did a similar test (2 nubles and 4 non-nullable) and I get ~2 seconds for GetUnderlyingType and ~1 second for GetGenericTypeDefinition, ie, GetGenericTypeDefinition is twice faster (not thrice). – [nawfal](#) Jul 2 '16 at 12:38

Did another round with 2 nullable and 2 non-nullable - this time GetUnderlyingType was 2.5 times slower. With only non-nullable - this time both are neck and neck. – [nawfal](#) Jul 2 '16 at 13:03

But more importantly, GetUnderlyingType is useful when you have to check for nullability & get underlying type if it is nullable. This is very useful and you see patterns often like Activator.CreateInstance(Nullable.GetUnderlyingType(type) ?? type) . It is like as keyword, checks for the cast as well as does it & return result. If you want to get the underlying type of nullable back then doing a GetGenericTypeDefinition check and then getting generic type will be a bad idea. Also GetUnderlyingType is much more readable & memorable. I wouldnt mind it if I am doing it only ~1000 times. – [nawfal](#) Jul 2 '16 at 13:04

▲ This version:

0

- caching results is faster,
- does not require unnecessary variables, like Method(T obj)
- NOT COMPLICATED :),

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

```

public static class IsNullable<T>
{
    private static readonly Type type = typeof(T);
    private static readonly bool is_nullable = type.IsGenericType &&
type.GetGenericTypeDefinition() == typeof(Nullable<>);
    public static bool Result { get { return is_nullable; } }
}

bool is_nullable = IsNullable<int?>.Result;

```

edited Nov 11 '15 at 1:13

answered Nov 10 '15 at 20:05



ipavlu

1,243 9 20

I think you answered your-self with that static declaration 'is_nullable'. Tip: declare objects with int? (object a = (int?)8;) and see what happens. – SoLaR Aug 21 at 8:16



0



Here is what I came up with, as everything else seemed to fail - at least on the *PLC - Portable Class Library / .NET Core* with \geq C# 6

Solution: Extend static methods for any Type T and `Nullable<T>` and use the fact that the static extension method, matching the underlying type is going to be invoked and takes precedence over the generic T extension-method.

For T :

```

public static partial class ObjectExtension
{
    public static bool IsNullable<T>(this T self)
    {
        return false;
    }
}

```

and for `Nullable<T>`

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

```

    {
        return true;
    }
}

```

Using Reflection and `type.IsGenericType` ... did not work on my current set of .NET Runtimes. Nor did the [MSDN Documentation](#) help.

```
if (type.IsGenericType && type.GetGenericTypeDefinition() == typeof(Nullable<>)) {...}
```

In part because the Reflection API has been changed quite significantly in .NET Core.

edited Aug 25 '16 at 7:19

answered Aug 25 '16 at 6:27



Lorenz Lo Sauer

16.7k 11 68 83



0



I think the ones using Microsoft's suggested testing against `IsGenericType` are good, but in the code for `GetUnderlyingType`, Microsoft uses an additional test to make sure you didn't pass in the generic type definition `Nullable<>`:

```

public static bool IsNullableType(this Type nullableType) =>
    // instantiated generic type only
    nullableType.IsGenericType &&
    !nullableType.IsGenericTypeDefinition &&
    Object.ReferenceEquals(nullableType.GetGenericTypeDefinition(), typeof(Nullable<>));

```

answered Jan 21 at 20:43



NetMage

16.5k 1 22 37



-1



a simple way to do this:

```

public static bool IsNullable(this Type type)
{
    if (type.IsValueType) return Activator.CreateInstance(type) == null;
}

```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

these are my unit tests and all passed

```
IsNullble_String_ShouldReturn_True  
IsNullble_Boolean_ShouldReturn_False  
IsNullble_Enum_ShouldReturn_Fasle  
IsNullble_Nullable_ShouldReturn_True  
IsNullble_Class_ShouldReturn_True  
IsNullble_Decimal_ShouldReturn_False  
IsNullble_Byte_ShouldReturn_False  
IsNullble_KeyValuePair_ShouldReturn_False
```

actual unit tests

```
[TestMethod]  
public void IsNullble_String_ShouldReturn_True()  
{  
    var typ = typeof(string);  
    var result = typ.IsNullable();  
    Assert.IsTrue(result);  
}  
  
[TestMethod]  
public void IsNullble_Boolean_ShouldReturn_False()  
{  
    var typ = typeof(bool);  
    var result = typ.IsNullable();  
    Assert.IsFalse(result);  
}  
  
[TestMethod]  
public void IsNullble_Enum_ShouldReturn_Fasle()  
{  
    var typ = typeof(System.GenericUriParserOptions);  
    var result = typ.IsNullable();  
    Assert.IsFalse(result);  
}  
  
[TestMethod]  
public void IsNullble_Nullable_ShouldReturn_True()  
{  
    var typ = typeof(Nullable<bool>);  
    var result = typ.IsNullable();  
    Assert.IsTrue(result);  
}
```

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.


```
public void IsNullable_Class_ShouldReturn_True()
{
    var typ = typeof(TestPerson);
    var result = typ.IsNullable();
    Assert.IsTrue(result);
}

[TestMethod]
public void IsNullable_Decimal_ShouldReturn_False()
{
    var typ = typeof(decimal);
    var result = typ.IsNullable();
    Assert.IsFalse(result);
}

[TestMethod]
public void IsNullable_Byte_ShouldReturn_False()
{
    var typ = typeof(byte);
    var result = typ.IsNullable();
    Assert.IsFalse(result);
}

[TestMethod]
public void IsNullable_KeyValuePair_ShouldReturn_False()
{
    var typ = typeof(KeyValuePair<string, string>);
    var result = typ.IsNullable();
    Assert.IsFalse(result);
}
```

answered Aug 18 '15 at 3:31



VJPPaz

451 3 12

protected by [Brian Mains](#) Jul 10 '14 at 17:17

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 [reputation](#) on this site (the [association bonus](#) does not count).

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

