Possible pitfalls of using this (extension method based) shorthand

Asked 16 years, 3 months ago Modified 4 years, 11 months ago Viewed 5k times

15

C#6 Update

In C#6 `?.` is now a language feature:

```
// C#1-5
propertyValue1 = myObject != null ? myObject.StringProperty : null;

// C#6
propertyValue1 = myObject?.StringProperty;
```

The question below still applies to older versions, but if developing a new application using the new `?.` operator is far better practice.

Original Question:

I regularly want to access properties on possibly null objects:

```
string propertyValue1 = null;
if( myObject1 != null )
    propertyValue1 = myObject1.StringProperty;

int propertyValue2 = 0;
if( myObject2 != null )
    propertyValue2 = myObject2.IntProperty;
```

And so on...

I use this so often that I have a snippet for it.

You can shorten this to some extent with an inline if:

```
propertyValue1 = myObject != null ? myObject.StringProperty : null;
```

However this is a little clunky, especially if setting lots of properties or if more than one level can be null, for instance:

```
propertyValue1 = myObject != null ?
    (myObject.ObjectProp != null ? myObject.ObjectProp.StringProperty) : null :
null;
```

What I really want is `??` style syntax, which works great for directly null types:

```
int? i = SomeFunctionWhichMightReturnNull();
propertyValue2 = i ?? 0;
```

So I came up with the following:

```
public static TResult IfNotNull<T, TResult>( this T input, Func<T, TResult>
action, TResult valueIfNull )
    where T : class
{
    if ( input != null ) return action( input );
    else return valueIfNull;
}

//lets us have a null default if the type is nullable
public static TResult IfNotNull<T, TResult>( this T input, Func<T, TResult>
action )
    where T : class
    where TResult : class
{ return input.IfNotNull( action, null ); }
```

This lets me us this syntax:

```
propertyValue1 = myObject1.IfNotNull( x => x.StringProperty );
propertyValue2 = myObject2.IfNotNull( x => x.IntProperty, 0);

//or one with multiple levels
propertyValue1 = myObject.IfNotNull(
    o => o.ObjectProp.IfNotNull( p => p.StringProperty ) );
```

This simplifies these calls, but I'm not sure about checking this sort of extension method in - it does make the code a little easier to read, but at the cost of extending object. This would appear on everything, although I could put it in a specifically referenced namespace.

This example is a rather simple one, a slightly more complex one would be comparing two nullable object properties:

```
if( ( obj1 == null && obj2 == null ) ||
    ( obj1 != null && obj2 != null && obj1.Property == obj2.Property ) )
    ...

//becomes
if( obj1.NullCompare( obj2, (x,y) => x.Property == y.Property )
    ...
```

What are the pitfalls of using extensions in this way? Are other coders likely to be confused? Is this just abuse of extensions?

I guess what I really want here is a compiler/language extension:

```
propertyValue1 = myObject != null ? myObject.StringProperty : null;

//becomes
propertyValue1 = myObject?StringProperty;
```

This would make the complex case far easier:

```
propertyValue1 = myObject != null ?
    (myObject.ObjectProp != null ? myObject.ObjectProp.StringProperty) : null

//becomes
propertyValue1 = myObject?ObjectProp?StringProperty;
```

This would only work for value types, but you could return nullable equivalents:

```
int? propertyValue2 = myObject?ObjectProp?IntProperty;

//or

int propertyValue3 = myObject?ObjectProp?IntProperty ?? 0;
```

c# .net extension-methods

Share
Improve this question
Follow

edited Jan 2, 2020 at 18:52

Bradley Grainger
28.1k ● 4 ● 94 ● 111

asked Sep 23, 2008 at 19:05

Keith
155k ● 82 ● 306 ● 446

11 Answers

Sorted by: Highest score (default)

16

We independently came up with the exact same extension method name and implementation: Null-propagating extension method. So we don't think it's confusing or an abuse of extension methods.

I would write your "multiple levels" example with chaining as follows:

```
propertyValue1 = myObject.IfNotNull(o => o.ObjectProp).IfNotNull(p =>
p.StringProperty);
```

There's a now-closed bug on Microsoft Connect that proposed "?." as a new C# operator that would perform this null propagation. Mads Torgersen (from the C#

language team) briefly explained why they won't implement it.

Share

Improve this answer

Follow

edited Oct 10, 2008 at 15:04

answered Oct 10, 2008 at 14:32

Bradley Grainger

28.1k ● 4 ● 94 ● 111

Yeah, I asked Mads directly at TechEd - basically this feature keeps missing the cut, they might still add it in a future C# version. – Keith Mar 28, 2011 at 13:20

@Keith Thanks for the update! (It can't hurt to keep reminding the C# team that customers would find this useful.) – Bradley Grainger Mar 29, 2011 at 2:33

They are now seriously considering it: blogs.msdn.com/b/jerrynixon/archive/2014/02/26/... – Mark Cidade Feb 28, 2014 at 15:32

1 The status of the "Null propagation" feature is "Done". See Roslyn's Language feature implementation status – orad Jul 25, 2014 at 17:01

`?.` is implemented in C#6 - I've updated the question to reflect that. The answers here still apply to C#1-5 – Keith Jun 15, 2015 at 6:41

15

Here's another solution, for chained members, including extension methods:

```
public static U PropagateNulls<T,U> ( this T obj
                                     ,Expression<Func<T,U>> expr)
{  if (obj==null) return default(U);

   //uses a stack to reverse Member1(Member2(obj)) to obj.Member1.Member2
   var members = new Stack<MemberInfo>();

   bool       searchingForMembers = true;
   Expression currentExpression   = expr.Body;

   while (searchingForMembers) switch (currentExpression.NodeType)
    { case ExpressionType.Parameter: searchingForMembers = false; break;

           case ExpressionType.MemberAccess:
           { var ma= (MemberExpression) currentExpression;
             members.Push(ma.Member);
             currentExpression = ma.Expression;
           } break;

          case ExpressionType.Call:
          { var mc = (MethodCallExpression) currentExpression;
            members.Push(mc.Method);

           //only supports 1-arg static methods and 0-arg instance methods
           if (   (mc.Method.IsStatic && mc.Arguments.Count == 1)
               || (mc.Arguments.Count == 0))
            { currentExpression = mc.Method.IsStatic ? mc.Arguments[0]
                                                     : mc.Object;
              break;
            }
```

```
            throw new NotSupportedException(mc.Method+" is not supported");
          }

         default: throw new NotSupportedException
                          (currentExpression.GetType()+" not supported");
    }

     object currValue = obj;
     while(members.Count > 0)
      { var m = members.Pop();

        switch(m.MemberType)
         { case MemberTypes.Field:
             currValue = ((FieldInfo) m).GetValue(currValue);
             break;

           case MemberTypes.Method:
             var method = (MethodBase) m;
             currValue = method.IsStatic
                               ? method.Invoke(null,new[]{currValue})
                               : method.Invoke(currValue,null);
             break;

           case MemberTypes.Property:
             var method = ((PropertyInfo) m).GetGetMethod(true);
                  currValue = method.Invoke(currValue,null);
             break;

         }

        if (currValue==null) return default(U);
      }

     return (U) currValue;
}
```

Then you can do this where any can be null, or none:

```
foo.PropagateNulls(x => x.ExtensionMethod().Property.Field.Method());
```

Share

Improve this answer

Follow

edited Aug 14, 2014 at 11:03

Damian Powell

8,765 ● 7 ● 50 ● 58

answered Sep 28, 2008 at 2:00

Mark Cidade

99.8k ● 33 ● 229 ● 237

I love this idea! However, it (as well as all the other solutions here) doesn't seem to work within a LINQ statement. For instance when doing a .Select into a new anonymous type like .Select(s=> new {MyNewProperty = s.PropogateNulls(p=>p.Thing)}). This doesn't work. Still have to use the old null checking there. – Brian McCord Mar 22, 2016 at 23:24

You have to modify the code to accept static methods with two arguments, for `Enumerable.Select(src, lambda)` – Mark Cidade Mar 23, 2016 at 5:31

11

If you find yourself having to check very often if a reference to an object is null, may be you should be using the Null Object Pattern. In this pattern, instead of using null to deal with the case where you don't have an object, you implement a new class with the same interface but with methods and properties that return adequate default values.

Share
Improve this answer
Follow

edited Feb 27, 2013 at 12:58

Răzvan Flavius Panda
22.1k ● 18 ● 119 ● 175

answered Sep 23, 2008 at 19:28

ljorquera
1,080 ● 1 ● 10 ● 13

5

How is

```
propertyValue1 = myObject.IfNotNull(o => o.ObjectProp.IfNotNull( p =>
p.StringProperty ) );
```

easier to read and write than

```
if(myObject != null && myObject.ObjectProp != null)
    propertyValue1 = myObject.ObjectProp.StringProperty;
```

Jafar Husain posted a sample of using Expression Trees to check for null in a chain, Runtime macros in C# 3.

This obviously has performance implications though. Now if only we had a way to do this at compile time.

Share Improve this answer Follow

answered Sep 23, 2008 at 20:14

Robert Jeppesen
7,857 ● 3 ● 37 ● 50

1 I agree with your first comment. The trouble it seems to me is that it's not immediately obvious from the code what it is doing. It is less cluttered, but it's only easier to understand *once you know what it does*. – xan Nov 21, 2008 at 10:34

5

I just have to say that I love this hack!

I hadn't realized that extension methods don't imply a null check, but it totally makes sense. As James pointed out, The extension method call itself is not any more expensive than a normal method, however if you are doing a ton of this, then it does make sense to follow the Null Object Pattern, that ljorquera suggested. Or to use a null object and ?? together.

```
class Class1
{
    public static readonly Class1 Empty = new Class1();
.
.
x = (obj1 ?? Class1.Empty).X;
```

Share Improve this answer Follow

answered Sep 23, 2008 at 20:15

Stefan Rusek

4,767 ● 2 ● 30 ● 25

Extension methods by themselves aren't more expensive, however, this particular extension method is more expensive, because it requires a lambda/anonymous method. Lambdas get compiled down to class allocations in the background. Thus, it's more expensive because it requires an allocation. – Judah Gabriel Himango Jan 29, 2009 at 23:06

1 @JudahHimango Lamdas are only complied to class allocations if they capture a variable (becoming a closure). If they don't, they get turned into a compiler generated static method... You can verify this by looking at a compiled DLL with something like dotPeek (jetbrains.com/decompiler) – John Gibb Feb 24, 2014 at 17:50

1

> it does make the code a little easier to read, but at the cost of extending object. This would appear on everything,

Note that you are not actually extending anything (except theoretically).

```
propertyValue2 = myObject2.IfNotNull( x => x.IntProperty, 0);
```

will generate IL code exactly as if it were written:

```
ExtentionClass::IfNotNull(myObject2,  x => x.IntProperty, 0);
```

There is no "overhead" added to the objects to support this.

Share Improve this answer Follow

answered Sep 23, 2008 at 20:04

James Curran

103k ● 37 ● 185 ● 262

To reader not in the know it looks like you're calling a method on a null reference. If you want this, I'd suggest putting it in a utility class rather than using an extension

1

method:

```
propertyValue1 = Util.IfNotNull(myObject1, x => x.StringProperty );
propertyValue2 = Util.IfNotNull(myObject2, x => x.IntProperty, 0);
```

The "Util." grates, but is IMO the lesser syntactic evil.

Also, if you developing this as part of a team, then gently ask what others think and do. Consistency across a codebase for frequently used patterns is important.

Share Improve this answer Follow

answered Sep 23, 2008 at 20:24

mancaus

3,001 ● 1 ● 21 ● 18

1

While extension methods generally cause misunderstandings when called from null instances, I think the intent is *pretty straightforward* in this case.

```
string x = null;
int len = x.IfNotNull(y => y.Length, 0);
```

I would want to be sure this static method works on Value Types that can be null, such as int?

Edit: compiler says that neither of these are valid:

```
    public void Test()
    {
        int? x = null;
        int a = x.IfNotNull(z => z.Value + 1, 3);
        int b = x.IfNotNull(z => z.Value + 1);
    }
```

Other than that, go for it.

Share

Improve this answer

Follow

edited Sep 24, 2008 at 10:15

answered Sep 23, 2008 at 23:39

Amy B

110k ● 21 ● 139 ● 190

That's why there's two overloads - one that needs a default with no constraint on the result type, and one that doesn't need a default but constrains the result to reference types.
– Keith Sep 24, 2008 at 8:00

That's because int? is actually compiled to Nullable<int>, which is actually a struct. It's only compiler magic that lets you compare it to a null (it correctly fails the where TResult : class

constraint). I may need to add another overload specific to Nullable<T> – Keith Sep 26, 2008 at 10:53

On second thoughts - Nullable<int> only has two properties: HasValue an Value - both used in the ?? syntax. This shouldn't be needed for int? – Keith Oct 6, 2008 at 18:55

1

Not an answer to the exact question asked, but there is *Null-Conditional Operator* in C# 6.0. I can argue it will be a poor choice to use the option in OP since C# 6.0 :)

So your expression is simpler,

```
string propertyValue = myObject?.StringProperty;
```

In case `myObject` is null it returns null. In case the property is a value type you have to use equivalent nullable type, like,

```
int? propertyValue = myObject?.IntProperty;
```

Or otherwise you can coalesce with null coalescing operator to give a default value in case of null. For eg,

```
int propertyValue = myObject?.IntProperty ?? 0;
```

`?.` is not the only syntax available. For indexed properties you can use `?[..]`. For eg,

```
string propertyValue = myObject?[index]; //returns null in case myObject is
null
```

One surprising behaviour of the `?.` operator is that it can intelligently bypass subsequent `.Member` calls if object happens to be null. One such example is given in the link:

```
var result = value?.Substring(0, Math.Min(value.Length,
length)).PadRight(length);
```

In this case `result` is null if `value` is null and `value.Length` expression wouldn't result in `NullReferenceException`.

Share

Improve this answer

edited Jan 7, 2015 at 15:29

answered Jan 7, 2015 at 15:17

nawfal

Follow 73k 58 336 376

0

Personally, even after all your explanation, I can't remember how the heck this works:

```
if( obj1.NullCompare( obj2, (x,y) => x.Property == y.Property )
```

This could be because I have no C# experience; however, I could read and understand everything else in your code. I prefer to keep code language agnostic (esp. for trivial things) so that tomorrow, another developer could change it to a whole new language without too much information about the existing language.

Share Improve this answer Follow

answered Sep 23, 2008 at 19:13

Swati
52.6k 4 40 54

0

Here is another solution using myObject.NullSafe(x=>x.SomeProperty.NullSafe(x=>x.SomeMethod)), explained at http://www.epitka.blogspot.com/

Share Improve this answer Follow

answered May 28, 2009 at 15:21

epitka
17.6k 21 89 146

Thanks, but that's a lot more code to do the same thing. Also your Maybe class is very similar to the framework's Nullable<T> and by using Invoke you add an unnecessary performance hit. Still - it's nice to see an alternate take on the same problem. – Keith May 28, 2009 at 22:18