Does Typescript support the ?. operator? (And, what's it called?)

209

27

Does Typescript currently (or are there plans to) support the safe navigation operator of `?.`

ie:

```
var thing = foo?.bar
// same as:
var thing = (foo) ? foo.bar : null;
```

Also, is there a more common name for this operator (it's incedibly hard to google for).

typescript

edited Mar 7 '13 at 1:36

asked Mar 7 '13 at 0:11

Marty Pitt

13.5k 31 107 186

3 @mattytommo you do have that in c#, its called the null coalescing operator and uses the ?? syntax weblogs.asp.net/scottgu/archive/2007/09/20/… – basarat Mar 7 '13 at 2:34

2 @BasaratAli Unfortunately not, coalesce is fine for `property ?? property2` , but if you tried `property.company ?? property1.company` and `property` was null, you'd get a `NullReferenceException` – mattytommo Mar 7 '13 at 8:49

1 @mattytommo Thanks I get it now '?.' actually soaks all null references in the chain. Sweet. – basarat Mar 8 '13 at 2:21

9 @mattytommo this does exist now for C#: msdn.microsoft.com/en-us/library/dn986595.aspx – Highmastdon Oct 6 '15 at 8:52

7 The Microsoft rep that visited us called it the Elvis operator as the question mark looks like Elvis' hair and a microphone he is singing into... – Zymotik Jun 21 '17 at 10:13

10 Answers

88

As far as what to call this operator in CoffeeScript, it's called the **existential operator** (specifically, the "accessor variant" of the existential operator).

From [CoffeeScript's documentation on Operators]:

> The accessor variant of the existential operator `?.` can be used to soak up null references in a chain of properties. Use it instead of the dot accessor `.` in cases where the base value may be **null** or **undefined**.

So, the **accessor variant of the existential operator** appears to be the proper way to refer to this operator; and TypeScript does not currently appear to support it (although [others have expressed a desire for this functionality]).

edited May 10 '17 at 20:16

Aleksandar
56 3 8

answered Mar 7 '13 at 0:21

Donut
80k 14 118 136

16 "accessor variant of the existential operator". Naturally. So catchy, it's near impossible to forget. :). Thanks for the extremely thorough answer. – Marty Pitt Mar 7 '13 at 0:35

@MartyPitt Sure thing! I agree, I'd love to see a) wider adoption of an operator like this (C# please!) and b) a better name (the "safe navigation" operator from your linked blog post has a nice ring to it). – Donut Mar 7 '13 at 0:38

3 github.com/Microsoft/TypeScript/issues/16 The issue was moved here. – Martin Vseticka Jul 28 '15 at 20:04

1 Angular implements this in it's templates: angular.io/guide/… – Enzoaeneas May 9 '18 at 20:23

3 In some other languages its called the "Elvis" operator – k0enf0rNL Sep 26 '18 at 8:08

119

Not as nice as a single ?, but it works:

```
var thing = foo && foo.bar || null;
```

You can use as many && as you like:

1,503 1 11 16

25 && evaluates as long as the statement is true. If it is true, it returns the last value. If it is false, it returns the first value that evaluated to false. That may be 0, null, false etc. || returns the first value that evaluates to true. – A. K-R Mar 8 '16 at 14:29

21 Doesn't work well if the bar is defined but evaluates to false (like boolean false, or zero). – mt_serg Dec 14 '17 at 8:29

62

This is defined in the ECMAScript Optional Chaining specification, so we should probably refer to *optional chaining* when we discuss this. Likely implementation:

```
const result = a?.b?.c;
```

The long and short of this one is that the TypeScript team are waiting for the ECMAScript specification to get tightened up, so their implementation can be non-breaking in the future. If they implemented something now, it would end up needing major changes if ECMAScript redefine their specification.

See Optional Chaining Specification

Where something is never going to be standard JavaScript, the TypeScript team can implement as they see fit, but for future ECMAScript additions, they want to preserve semantics even if they give early access, as they have for so many other features.

Short Cuts

So all of JavaScripts funky operators are available, including the type conversions such as...

```
var n: number = +myString; // convert to number
var b: bool = !!myString; // convert to bool
```

Manual Solution

But back to the question. I have an obtuse example of how you can do a similar thing in JavaScript (and therefore TypeScript) although

So if `foo` is `undefined` the result is `undefined` and if `foo` is defined and has a property named `bar` that has a value, the result is that value.

I put an [example on JSFiddle](JSFiddle).

This looks quite sketchy for longer examples.

```
var postCode = ((person||{}).address||{}).postcode;
```

Chain Function

If you are desperate for a shorter version while the specification is still up in the air, I use this method in some cases. It evaluates the expression and returns a default if the chain can't be satisfied or ends up null/undefined (note the `!=` is important here, we *don't* want to use `!==` as we want a bit of positive juggling here).

```
function chain<T>(exp: () => T, d: T) {
    try {
        let val = exp();
        if (val != null) {
            return val;
        }
    } catch { }
    return d;
}

let obj1: { a?: { b?: string }} = {
    a: {
        b: 'c'
    }
};

// 'c'
console.log(chain(() => obj1.a.b, 'Nothing'));

obj1 = {
    a: {}
};
```

```
console.log(chain(() => obj1.a.b, 'Nothing'));

obj1 = null;

// 'Nothing'
console.log(chain(() => obj1.a.b, 'Nothing'));
```

edited Jan 7 '18 at 15:53

answered Mar 7 '13 at 6:27

Fenton

163k 46 299 326

1 interesting but in my case `(this.loop || {}).nativeElement` saying `Property 'nativeElement' does not exist on type '{}'. any this.loop` typeof angular.io/api/core/ElementRef – Kuncevič Dec 21 '17 at 4:00

@Kuncevic - you need to either... 1) provide a compatible default in place of `{}` , or 2) use a type assertion to silence the compiler. – Fenton Dec 21 '17 at 9:51

1 (foo||{}).bar; - Jesus... – Vitalii Vasylenko Jan 7 '18 at 13:32

Assuming `foo` is an actual useful object : `(foo || {}).bar` generally isn't going to compile in typescript because `{}` won't be of the same type as `foo` . That's the problem that @VeganHunter's solution aims to avoid. – Simon_Weaver Nov 12 '18 at 5:06

1 @Simon_Weaver then (foo || {bar}).bar will let the compiler run smoothly and I think that verbosity is acceptable. – Harps Mar 7 at 8:59

30

There is an open feature request for this on github where you can voice your opinion / desire : https://github.com/Microsoft/TypeScript/issues/16

answered Feb 12 '15 at 4:53

basarat

149k 28 278 383

Edit: I have updated the answer thanks to fracz comment.

This will only tell the compiler that the value is not null or undefined. This will **not** check if the value is null or undefined.

TypeScript Non-null assertion operator

```
// Compiled with --strictNullChecks
function validateEntity(e?: Entity) {
    // Throw exception if e is null or invalid entity
}

function processEntity(e?: Entity) {
    validateEntity(e);
    let s = e!.name;  // Assert that e is non-null and access name
}
```

edited May 23 '17 at 12:03
Community ♦
1 1

answered Nov 1 '16 at 13:27
Jose A
4,023 4 26 47

4 Not the same as `?` because it *asserts* that the value is defined. `?` is expected to silently fail / evaluate to false. Anyway, good to know. – fracz Nov 3 '16 at 21:08

Oh thanks. Let me edit! – Jose A Nov 4 '16 at 22:54

1 Now that I think about it... This answer is pretty pointless, because it does not do the "safe navigation" that the C# operator does. – Jose A Nov 4 '16 at 23:06

4 This answered my question, though. I knew about ?. from c# and tried it in typescript. It didn't work, but I saw that !. existed but didn't know what it did. I wondered if it was the same, did a google search, and found my way to this question which informed me that no, they are different. – Llewey Apr 27 '17 at 13:49

6

Operator `?.` is not supported in TypeScript **version 2.0**.

So I use the following function:

```
export function o<T>(someObject: T, defaultValue: T = {} as T) : T {
```

the usage looks like this:

```
o(o(o(test).prop1).prop2
```

plus, you can set a default value:

```
o(o(o(o(test).prop1).prop2, "none")
```

It works really well with IntelliSense in Visual Studio.

edited Apr 27 '18 at 1:52

answered Jan 17 '17 at 1:50

VeganHunter
2,009 2 11 11

1 This is exactly what I was looking for! It works in typescript 2.1.6. – Rajab Shakirov Mar 25 '17 at 16:50

3 or you could call it `elvis<T>` ;-) – Simon_Weaver Nov 12 '18 at 5:03

1 Simon_Weaver, I call it "sad clown" :o(– VeganHunter Nov 13 '18 at 6:18

1

We created this util method while working on Phonetradr which can give you type-safe access to deep properties with Typescript:

```
/**
 * Type-safe access of deep property of an object
 *
 * @param obj                   Object to get deep property
 * @param unsafeDataOperation   Function that returns the deep property
 * @param valueIfFail           Value to return in case if there is no such property
 */
export function getInSafe<O,T>(obj: O, unsafeDataOperation: (x: O) => T, valueIfFail?:
any) : T {
    try {
        return unsafeDataOperation(obj)
```

```
getInSafe(sellTicket, x => x.phoneDetails.imeiNumber, '');

//Example from above
getInSafe(foo, x => x.bar.check, null);
```

Run code snippet Expand snippet

answered Jul 28 '17 at 15:15

phidias
368 2 6

Cool!! Is there any caveats? I have a wrapper class with about 20 getters to write, every one of them has the following type of return - and all fields have to be null checked `return this.entry.fields.featuredImage.fields.file.url;` – Drenai Jan 14 '18 at 11:44

The only caveat might possibly be a performance impact, but I'm not qualified to speak to how the various JITers handle this. – Ray Suelzer Feb 17 '18 at 3:43

1

I don't generally recommend this approach (watch out for performance concerns), but you can use the spread operator to shallow clone an object, which you can then access the property on.

```
const person = { personId: 123, firstName: 'Simon' };
const firstName = { ...person }.firstName;
```

This works because the type of 'firstName' is 'propagated' through.

I'll use this most frequently when I have a `find(...)` expression that can return null and I need a single property from it:

```
// this would cause an error (this ID doesn't exist)
const people = [person];
const firstName2 = people.find(p => p.personId == 999).firstName;

// this works - but copies every property over so raises performance concerns
```

Simon_Weaver
77k 64 479 543

0

As answered before, it's currently still being considered but it has been dead in the water for a few years by now.

Building on the existing answers, here's the most concise *manual* version I can think of:

jsfiddle

```
function val<T>(valueSupplier: () => T): T {
  try { return valueSupplier(); } catch (err) { return undefined; }
}

let obj1: { a?: { b?: string }} = { a: { b: 'c' } };
console.log(val(() => obj1.a.b)); // 'c'

obj1 = { a: {} };
console.log(val(() => obj1.a.b)); // undefined
console.log(val(() => obj1.a.b) || 'Nothing'); // 'Nothing'

obj1 = {};
console.log(val(() => obj1.a.b) || 'Nothing'); // 'Nothing'

obj1 = null;
console.log(val(() => obj1.a.b) || 'Nothing'); // 'Nothing'
```

It simply silently fails on missing property errors. It falls back to the standard syntax for determining default value, which can be omitted completely as well.

Although this works for simple cases, if you need more complex stuff such as calling a function and then access a property on the result, then any other errors are swallowed as well. Bad design.

In the above case, an optimized version of the other answer posted here is the better option:

jsfiddle

```
let obj1: { a?: { b?: string }} = { a: { b: 'c' } };
console.log(o(o(o(obj1).a)).b); // 'c'

obj1 = { a: {} };
console.log(o(o(o(obj1).a)).b); // undefined
console.log(o(o(o(obj1).a)).b || 'Nothing'); // 'Nothing'

obj1 = {};
console.log(o(o(o(obj1).a)).b || 'Nothing'); // 'Nothing'

obj1 = null;
console.log(o(o(o(obj1).a)).b || 'Nothing'); // 'Nothing'
```

A more complex example:

```
o(foo(), []).map((n) => n.id)
```

You can also go the other way and use something like Lodash' `_.get()`. It is concise, but the compiler won't be able to judge the validity of the properties used:

```
console.log(_.get(obj1, 'a.b.c'));
```

edited Apr 19 '18 at 20:19

answered Apr 19 '18 at 19:56

Benny Bottema
5,520 8 51 70

0

`_.get(obj, 'address.street.name')` works great for JavaScript where you have no types. But for TypeScript *we need* the real Elvis operator!

answered Apr 20 at 2:17

Angelos Pikoulas
774 10 21