Typescript interface default values

Asked 3 years, 7 months ago Active 2 months ago Viewed 121k times

82

9

I have the following interface in TypeScript:

```
interface IX {
    a: string,
    b: any,
    c: AnotherType
}
```

I declare a variable of that type and I initialize all the properties

```
let x: IX = {
    a: 'abc',
    b: null,
    c: null
}
```

Then I assign real values to them in an init function later

```
x.a = 'xyz'
x.b = 123
x.c = new AnotherType()
```

But I don't like having to specify a bunch of default null values for each property when declaring the object when they're going to just be set later to real values. Can I tell the interface to default the properties I don't supply to null? What would let me do this:

```
let x: IX = {
    a: 'abc'
}
```

without getting a compiler error. Right now it tells me typescript

edited Dec 13 '17 at 13:45
Mark Cooper
3,989 5 44 79

asked Jan 28 '16 at 23:47
d512
16.4k 11 60 69

I've added docs for you : basarat.gitbooks.io/typescript/content/docs/tips/… – basarat Jan 29 '16 at 0:32

1 IMO, the answer stackoverflow.com/a/35074490/129196 shouldn't be the approach to take. If you can have an object in a state without having all its properties initialized and still be valid, then you should declare those properties as optional as indicated by this answer: stackoverflow.com/a/43226857/129196. Otherwise we will lose the sole purpose of using typescript (for type safety). – Charles Prakash Dasari Feb 13 '18 at 2:26

6 Answers

71

> Can I tell the interface to default the properties I don't supply to null? What would let me do this

No. But by default they are `undefined` which is mostly just fine. You can use the following pattern, i.e have a type assertion at the point of creation:

```
let x: IX = {} as any;

x.a = 'xyz'
x.b = 123
x.c = new AnotherType()
```

I have this and other patterns documented here : https://basarat.gitbooks.io/typescript/content/docs/tips/lazyObjectLiteralInitialization.html

edited Dec 13 '17 at 13:45
Mark Cooper
3,989 5 44 79

answered Jan 28 '16 at 23:58
basarat
154k 29 291 395

11 Using `any` undermines the purpose of *Type*Script. There are other answers without this drawback. – Jack Miller Feb 1 at 7:35

could any one help me about similar question but using generics. Here at this question – TAB Feb 11 at 10:30

1 Odd that basarat would go with the 'any' example when, in the link provided, he offers a much better option with 'let foo = {} as Foo;' ('Foo" being an Interface) – Neurothustra Feb 28 at 13:52

52

You can't set default values in an interface, but you can accomplish what you want to do by using Optional Properties (compare paragraph #3):

https://www.typescriptlang.org/docs/handbook/interfaces.html

Simply change the interface to:

```
interface IX {
    a: string,
    b?: any,
    c?: AnotherType
}
```

You can then do:

```
let x: IX = {
    a: 'abc'
}
```

And use your init function to assign default values to `x.b` and `x.c` if those properies are not set.

answered Apr 5 '17 at 9:16
Timar
629 4 3

2 In the question it was asked to initialize `x.b` and `x.c` with `null` . When writing `let x = {a: 'abc'}` then `x.b` and `x.c` are `undefined` , so this answer doesn't fully meet the requirements, although it's a smart quick fix. – Benny Neugebauer Mar 17 '18 at 11:26

1 @BennyNeugebauer The accepted answer has the same flaw. This is the best answer – tel Feb 20 at 22:32

17

You can implement the interface with a class, then you can deal with initializing the members in the constructor:

```
class IXClass implements IX {
    a: string;
    b: any;
    c: AnotherType;

    constructor(obj: IX);
    constructor(a: string, b: any, c: AnotherType);
    constructor() {
        if (arguments.length == 1) {
            this.a = arguments[0].a;
            this.b = arguments[0].b;
            this.c = arguments[0].c;
        } else {
            this.a = arguments[0];
            this.b = arguments[1];
            this.c = arguments[2];
        }
    }
}
```

Another approach is to use a factory function:

```
function ixFactory(a: string, b: any, c: AnotherType): IX {
    return {
        a: a,
        b: b,
        c: c
    }
}
```

Then you can simply:

```
var ix: IX = null;
...

ix = new IXClass(...);
// or
ix = ixFactory(...);
```

Nitzan Tomer
82.9k 25 198 210

8

While @Timar's answer works perfectly for `null` default values (what was asked for), here another easy solution which allows other default values: Define an option interface as well as an according constant containing the defaults; in the constructor use the spread operator to set the `options` member variable

```
interface IXOptions {
    a?: string,
    b?: any,
    c?: number
}

const XDefaults: IXOptions = {
    a: "default",
    b: null,
    c: 1
}

export class ClassX {
    private options: IXOptions;

    constructor(XOptions: IXOptions) {
        this.options = { ...XDefaults, ...XOptions };
    }

    public printOptions(): void {
        console.log(this.options.a);
        console.log(this.options.b);
        console.log(this.options.c);
    }
}
```

Now you can use the class like this:

```
const x = new ClassX({ a: "set" });
x.printOptions();
```

Output:

```
set
null
1
```

edited Mar 4 at 18:58
oluckyman
1,583 1 18 30

answered Feb 1 at 7:27
Jack Miller
2,223 1 25 34

What's the point of `this.options = this.options;` line? – Orkhan Alikhanov Mar 4 at 15:48

1 Ups! Good catch! I think I added it to avoid TS complaining that `options` is unused before I added method `printOptions()`. You can safely remove that line. – Jack Miller Mar 4 at 16:34

2

You can use the `Partial` mapped type as explained in the documentation: https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-1.html

In your example, you'll have:

```
interface IX {
    a: string;
    b: any;
    c: AnotherType;
}

let x: Partial<IX> = {
    a: 'abc'
}
```

answered Jul 5 at 22:39
F. Bauer
81 4

I stumbled on this while looking for a better way than what I had arrived at. Having read the answers and trying them out I thought it was worth posting what I was doing as the other answers didn't feel as succinct for me. It was important for me to only have to write a short

Using a custom generic deepCopy function:

```
deepCopy = <T extends {}>(input: any): T => {
  return JSON.parse(JSON.stringify(input));
};
```

Define your interface

```
interface IX {
    a: string;
    b: any;
    c: AnotherType;
}
```

... and define the defaults in a separate const.

```
const XDef : IX = {
    a: '',
    b: null,
    c: null,
};
```

Then init like this:

```
let x : IX = deepCopy(XDef);
```

That's all that's needed..

.. however ..

If you want to **custom initialise any root element** you can modify the deepCopy function to accept custom default values. The function becomes:

```
deepCopyAssign = <T extends {}>(input: any, rootOverwrites?: any): T => {
  return JSON.parse(JSON.stringify({ ...input, ...rootOverwrites }));
};
```

```
let x : IX = deepCopyAssign(XDef, { a:'customInitValue' } );
```

Any other preferred way of deep copy would work. If only a shallow copy is needed then Object.assign would suffice, forgoing the need for the utility `deepCopy` or `deepCopyAssign` function.

```
let x : IX = object.assign({}, XDef, { a:'customInitValue' });
```

Known Issues

- It will not deep assign in this guise but it's not too difficult to modify `deepCopyAssign` to iterate and check types before assigning.
- Functions and references will be lost by the parse/stringify process. I don't need those for my task and neither did the OP.
- Custom init values are not hinted by the IDE or type checked when executed.

edited May 19 at 6:41

answered May 18 at 23:15

Moss Palmer
990 1 11 27