TypeScript and field initializers

164

How to init a new class in `TS` in such a way (example in `C#` to show what I want):

```
// ... some code before
return new MyClass { Field1 = "ASD", Field2 = "QWE" };
// ...  some code after
```

38

SOLUTION:
Classic `JavaScript` syntax:

```
return { Field1: "ASD", Field2: "QWE" };
```

typescript

edited Feb 8 '18 at 23:53
jhpratt
3,417 10 24 33

asked Jan 3 '13 at 15:36
Nickon
3,778 9 43 94

9 The "solution" you just appended to your question is not valid TypeScript or JavaScript. But it is valuable to point out that it is the most intuitive thing to try. – Jacob Foshee Aug 12 '15 at 22:43

Sorry, my bad... – Nickon Aug 13 '15 at 8:19

1 @JacobFoshee not valid JavaScript? See my Chrome Dev Tools: i.imgur.com/vpathu6.png *(but Visual Studio Code or any other TypeScript linted would inevitably complain)* – Mars Robertson Jun 30 '16 at 9:46

4 @MichalStefanow the OP edited the question after I posted that comment. He did have `return new MyClass { Field1: "ASD", Field2: "QWE" };` – Jacob Foshee Jun 30 '16 at 20:28

68

As stated, you can already do this by using interfaces in TypeScript instead of classes:

```
interface Name {
    first: string;
    last: string;
}
class Person {
    name: Name;
    age: number;
}

var bob: Person = {
    name: {
        first: "Bob",
        last: "Smith",
    },
    age: 35,
};
```

edited Jan 17 '17 at 12:49

answered Jan 3 '13 at 15:44

Wouter de Kort

31.4k 8 60 93

65 In your example, bob is not an instance of class Person. I don't see how this is equivalent to the C# example. – Jack Wester Feb 23 '13 at 17:27

3 interface names are better to be started with capital "I" – Kiarash Oct 4 '15 at 9:36

15 1. Person is not a class and 2. @JackWester is right, bob is not an instanceof Person. Try alert(bob instanceof Person); In this code example Person is there for Type Assertion purposes only. – Jacques Oct 13 '15 at 10:42

13 I agree with Jack and Jaques, and I think its worth repeating. Your bob is of the *type* `Person`, but it is not at all an instance of `Person`. Imagine that `Person` actually would be a class with a complex constructor and a bunch of methods, this approach would fall flat on its face. It's good that a bunch of people found your approach useful, but it's not a solution to the question as it's stated and you could just as well use a class Person in your example instead of an interface, it would be the same type. – Alex Aug 4 '16 at 17:21

20 Why is this an accepted answer when it is clearly wrong? – Hendrik Wiese Oct 7 '17 at 20:14

```
    public address: string = "default"
    public age: number = 0;

    public constructor(init?:Partial<Person>) {
        Object.assign(this, init);
    }
}

let persons = [
    new Person(),
    new Person({}),
    new Person({name:"John"}),
    new Person({address:"Earth"}),
    new Person({age:20, address:"Earth", name:"John"}),
];
```

Original Answer:

My approach is to define a separate `fields` variable that you pass to the constructor. The trick is to redefine all the class fields for this initialiser as optional. When the object is created (with its defaults) you simply assign the initialiser object onto `this` ;

```
export class Person {
    public name: string = "default"
    public address: string = "default"
    public age: number = 0;

    public constructor(
        fields?: {
            name?: string,
            address?: string,
            age?: number
        }) {
        if (fields) Object.assign(this, fields);
    }
}
```

or do it manually (bit more safe):

```
if (fields) {
```

usage:

```
let persons = [
    new Person(),
    new Person({name:"Joe"}),
    new Person({
        name:"Joe",
        address:"planet Earth"
    }),
    new Person({
        age:5,
        address:"planet Earth",
        name:"Joe"
    }),
    new Person(new Person({name:"Joe"})) //shallow clone
];
```

and console output:

```
Person { name: 'default', address: 'default', age: 0 }
Person { name: 'Joe', address: 'default', age: 0 }
Person { name: 'Joe', address: 'planet Earth', age: 0 }
Person { name: 'Joe', address: 'planet Earth', age: 5 }
Person { name: 'Joe', address: 'default', age: 0 }
```

This gives you basic safety and property initialization, but its all optional and can be out-of-order. You get the class's defaults left alone if you don't pass a field.

You can also mix it with required constructor parameters too -- stick `fields` on the end.

About as close to C# style as you're going to get I think (actual field-init syntax was rejected). I'd much prefer proper field initialiser, but doesn't look like it will happen yet.

For comparison, If you use the casting approach, your initialiser object must have ALL the fields for the type you are casting to, plus don't get any class specific functions (or derivations) created by the class itself.

edited Dec 7 '16 at 22:51 answered Jun 7 '16 at 14:38

type/compile safety between the params and properties, though. – Fiddles Aug 16 '16 at 1:36

1 @user1817787 you'd probably be better off defining anything that has a default as optional in the class itself, but assign a default. Then don't use `Partial<>` , just `Person` - that will require you to pass an object that has the required fields. that said, see here for ideas (See Pick) that limit Type Mapping to certain fields. – Meirion Hughes Dec 13 '16 at 18:08

2 This really should be the answer. Its the best solution to this issue. – Ascherer Feb 10 '17 at 5:41

5 that is GOLDEN piece of code `public constructor(init?:Partial<Person>) { Object.assign(this, init); }` – Kuncevič Jul 25 '17 at 4:29

1 If Object.assign is showing an error like it was for me, please see this SO answer: stackoverflow.com/a/38860354/2621693 – Jim Yarbro Nov 11 '18 at 10:12

16

Below is a solution that combines a shorter application of `Object.assign` to more closely model the original `C#` pattern.

But first, lets review the techniques offered so far, which include:

1. Copy constructors that accept an object and apply that to `Object.assign`
2. A clever `Partial<T>` trick within the copy constructor
3. Use of "casting" against a POJO
4. Leveraging `Object.create` instead of `Object.assign`

Of course, each have their pros/cons. Modifying a target class to create a copy constructor may not always be an option. And "casting" loses any functions associated with the target type. `Object.create` seems less appealing since it requires a rather verbose property descriptor map.

Shortest, General-Purpose Answer

So, here's yet another approach that is somewhat simpler, maintains the type definition and associated function prototypes, and more closely models the intended `C#` pattern:

```
const john = Object.assign( new Person(), {
    name: "John",
```

That's it. The only addition over the `C#` pattern is `Object.assign` along with 2 parenthesis and a comma. Check out the working example below to confirm it maintains the type's function prototypes. No constructors required, and no clever tricks.

Working Example

This example shows how to initialize an object using an approximation of a `C#` field initializer:

```
class Person {
    name: string = '';
    address: string = '';
    age: number = 0;

    aboutMe() {
        return `Hi, I'm ${this.name}, aged ${this.age} and from ${this.address}`;
    }
}

// typescript field initializer (maintains "type" definition)
const john = Object.assign( new Person(), {
    name: "John",
    age: 29,
    address: "Earth"
});

// initialized object maintains aboutMe() function prototype
console.log( john.aboutMe() );
```

Run code snippet Expand snippet

edited Oct 25 '17 at 1:52

answered Oct 21 '17 at 3:38

Jonathan B.
928 7 15

I really like this one. Nice, thanks. – PRMan Apr 13 at 0:01

16

```
var anInstance: AClass = <AClass> {
    Property1: "Value",
    Property2: "Value",
    PropertyBoolean: true,
    PropertyNumber: 1
};
```

Edit:

WARNING If the class has methods, the instance of your class will not get them. If AClass has a constructor, it will not be executed. If you use instanceof AClass, you will get false.

In conclusion, you should used interface and not class. The most common use is for the domain model declared as Plain Old Objects. Indeed, for domain model you should better use interface instead of class. Interfaces are use at compilation time for type checking and unlike classes, interfaces are completely removed during compilation.

```
interface IModel {
   Property1: string;
   Property2: string;
   PropertyBoolean: boolean;
   PropertyNumber: number;
}

var anObject: IModel = {
     Property1: "Value",
     Property2: "Value",
     PropertyBoolean: true,
     PropertyNumber: 1
 };
```

edited Mar 13 at 21:17

answered Nov 17 '15 at 8:17

rdhainaut

1,048 9 22

18 If `AClass` contained methods, `anInstance` wouldn't get them. – Monsignor Apr 25 '16 at 9:20

2 Also if AClass has a constructor, it will not be executed. – Michael Erickson Aug 14 '16 at 19:31

12

I suggest an approach that does not require Typescript 2.1:

```
class Person {
    public name: string;
    public address?: string;
    public age: number;

    public constructor(init:Person) {
        Object.assign(this, init);
    }

    public someFunc() {
        // todo
    }
}

let person = new Person(<Person>{ age:20, name:"John" });
person.someFunc();
```

key points:

- Typescript 2.1 not required, `Partial<T>` not required
- It supports functions (in comparison with simple type assertion which does not support functions)

edited Mar 7 '17 at 4:52

answered Mar 6 '17 at 5:59

VeganHunter
2,019 2 11 11

1 It doesn't respect mandatory fields: `new Person(<Person>{});` (because casting) and to be clear too; using Partial<T> supports functions. Ultimately, if you have required fields (plus prototype functions) you'll need to do: `init: { name: string, address?: string, age: number }` and drop the cast. – Meirion Hughes Mar 6 '17 at 21:44

Thank you Meirion. I corrected the answer. – VeganHunter Mar 7 '17 at 4:56

Also when we get conditional type mapping you'll be able to map just the functions to partials, and keep the properties as is. :) – Meirion Hughes Mar 7 '17 at 7:24

You could have the properties on the class and then assign them the usual way. And obviously they may or may not be required, so that's something else too. It's just that this is such nice syntactic sugar.

```
class MyClass{
    constructor(public Field1:string = "", public Field2:string = "")
    {
        // other constructor stuff
    }
}

var myClass = new MyClass("ASD", "QWE");
alert(myClass.Field1); // voila! statement completion on these properties
```

answered Jan 9 '13 at 1:15

Ralph Lavelle
4,525 4 29 43

3 I asked about field initializers... You describe constructors... – Nickon Jan 9 '13 at 10:26

6 My deepest apologies. But you didn't actually "ask about field initializers", so it's natural to assume that you might be interested in alternative ways of newing up a class in TS. You might give a tiny bit more information in your question if you are so ready to downvote. – Ralph Lavelle Jan 9 '13 at 22:56

+1 constructor is the way to go where possible; but in cases where you have a lot of fields and want to only initialise some of them, I think my answer makes things easier. – Meirion Hughes Jun 8 '16 at 8:27

1 If you have many fields, initializing an object like this would be pretty unwieldy as you would be passing the constructor a wall of nameless values. That isn't to say this method has no merit; it would be best used for simple objects with few fields. Most publications say that around four or five parameters in a member's signature is the upper limit. Just pointing this out because I found a link to this solution on someone's blog while searching for TS initializers. I have objects with 20+ fields that need to be initialized for unit tests. – Dustin Cleveland Oct 8 '17 at 2:35

10

In some scenarios it may be acceptable to use `Object.create` . The Mozilla reference includes a polyfill if you need back-compatibility or want to roll your own initializer function.

Applied to your example:

Useful Scenarios

- Unit Tests
- Inline declaration

In my case I found this useful in unit tests for two reasons:

1. When testing expectations I often want to create a slim object as an expectation
2. Unit test frameworks (like Jasmine) may compare the object prototype (`__proto__`) and fail the test. For example:

```
var actual = new MyClass();
actual.field1 = "ASD";
expect({ field1: "ASD" }).toEqual(actual); // fails
```

The output of the unit test failure will not yield a clue about what is mismatched.

3. In unit tests I can be selective about what browsers I support

Finally, the solution proposed at http://typescript.codeplex.com/workitem/334 does not support inline json-style declaration. For example, the following does not compile:

```
var o = {
  m: MyClass: { Field1:"ASD" }
};
```

edited Oct 21 '17 at 6:14

Jonathan B.
928 7 15

answered Aug 11 '15 at 20:52

Jacob Foshee
1,939 22 44

2

I wanted a solution that would have the following:

- All the data objects are required and must be filled by the constructor.

```
export class Person {
  id!: number;
  firstName!: string;
  lastName!: string;

  getFullName() {
    return `${this.firstName} ${this.lastName}`;
  }

  constructor(data: OnlyData<Person>) {
    Object.assign(this, data);
  }
}

const person = new Person({ id: 5, firstName: "John", lastName: "Doe" });
person.getFullName();
```

All the properties in the constructor are mandatory and may not be omitted without a compiler error.

It is dependant on the `OnlyData` that filters out `getFullName()` out of the required properties and it is defined like so:

```
// based on : https://medium.com/dailyjs/typescript-create-a-condition-based-subset-
types-9d902cea5b8c
type FilterFlags<Base, Condition> = { [Key in keyof Base]: Base[Key] extends Condition ?
never : Key };
type AllowedNames<Base, Condition> = FilterFlags<Base, Condition>[keyof Base];
type SubType<Base, Condition> = Pick<Base, AllowedNames<Base, Condition>>;
type OnlyData<T> = SubType<T, (_: any) => any>;
```

Current limitations of this way:

- Requires TypeScript 2.8
- Classes with getters/setters

answered Apr 23 at 18:51

VitalyB

7,380 6 55 83

1

```
return <MyClass>{ Field1: "ASD", Field2: "QWE" };
```

answered Aug 7 '16 at 4:05

Peter Pompeii

1,101 9 12

6 Unfortunately, (1) this is not type-casting, but a compile-time type-*assertion*, (2) the question asked *"how to init a new **class**"* (emphasis mine), and this approach will fail to accomplish that. It would be certainly nice if TypeScript had this feature, but unfortunately, that's not the case. – John Weisz Jan 16 '17 at 17:04

-1

If you're using an old version of typescript < 2.1 then you can use similar to the following which is basically casting of any to typed object:

```
const typedProduct = <Product>{
                    code: <string>product.sku
                };
```

> NOTE: Using this method is only good for data models as it will remove all the methods in the object. It's basically casting any object to a typed object

edited Jun 22 '18 at 11:51

andrew.fox

3,810 4 35 56

answered Jun 1 '18 at 10:34

Mohsen Tabareh

438 4 7