Experiment: making TypeScript immutable-by-default

by Evan Hahn, posted Nov 18, 2025

I like programming languages where variables are immutable by default. For example, in Rust, `let` declares an immutable variable and `let mut` declares a mutable one. I've long wanted this in other languages, like TypeScript, which is mutable by default—the opposite of what I want!

I wondered: **is it possible to make TypeScript values immutable by default?**

My goal was to do this purely with TypeScript, without changing TypeScript itself. That meant no lint rules or other tools. I chose this because I wanted this solution to be as "pure" as possible…and it also sounded more fun.

I spent an evening trying to do this. **I failed but made progress! I made arrays and `Record`s immutable by default, but I couldn't get it working for regular objects.** If you figure out how to do this completely, please contact me—I must know!

Step 1: obliterate the built-in libraries

TypeScript has built-in type definitions for JavaScript APIs like `Array` and `Date` and `String`. If you've ever changed the `target` or `lib` options in your TSConfig, you've tweaked which of these definitions are included. For example, you might add the "ES2024" library if you're targeting a newer runtime.

My goal was to swap the built-in libraries with an immutable-by-default replacement.

The first step was to stop using any of the built-in libraries. I set the noLib flag in my TSConfig, like this:

```
{
  "compilerOptions": {
    "noLib": true
  }
}
```

Then I wrote a very simple script and put it in `test.ts`:

```
console.log("Hello world!");
```

When I ran `tsc`, it gave a bunch of errors:

```
Cannot find global type 'Array'.
Cannot find global type 'Boolean'.
Cannot find global type 'Function'.
Cannot find global type 'IArguments'.
Cannot find global type 'Number'.
Cannot find global type 'Object'.
Cannot find global type 'RegExp'.
Cannot find global type 'String'.
```

Progress! I had successfully obliterated any default TypeScript libraries, which I could tell because it couldn't find core types like `String` or `Boolean`.

Time to write the replacement.

Step 2: a skeleton standard library

This project was a prototype. Therefore, I started with a minimal solution that would type-check. I didn't need it to be good!

I created `lib.d.ts` and put the following inside:

```
// In lib.d.ts:
declare var console: any;

interface Boolean {}
interface Function {}
interface IArguments {}
interface Number {}
interface RegExp {}
interface String {}
interface Object {}

// TODO: We'll update this soon.
interface Array<T> {}
```

Now, when I ran `tsc`, I got no errors! I'd defined all the built-in types that TypeScript needs, and a dummy `console` object.

As you can see, this solution is impractical for production. For one, none of these interfaces have any properties! `"foo".toUpperCase()` isn't defined, for example. That's okay because this is only a prototype. A production-ready version would need to define all of those things—tedious, but should be straightforward.

Step 3: making arrays immutable

I decided to tackle this with a test-driven development style. I'd write some code that I want to type-check, watch it *fail* to type-check, then fix it.

I updated `test.ts` to contain the following:

```
// In test.ts:
const arr = [1, 2, 3];

// Non-mutation should be allowed.
console.log(arr[1]);
console.log(arr.map((n) => n + 1));

// @ts-expect-error Mutation should not be allowed.
arr[0] = 9;
// @ts-expect-error Mutation should not be allowed.
arr.push(4);
```

This tests three things:

1. Creating arrays with array literals is possible.
2. Non-mutating operations, like `arr[1]` and `arr.map()`, are allowed.
3. Operations that mutate the array, like `arr[1] = 9`, are disallowed.

When I ran `tsc`, I saw two errors:

- `arr[0] = 9` is allowed. There's an unused `@ts-expect-error` there.
- `arr.map` doesn't exist.

So I updated the `Array` type in `lib.d.ts` with the following:

```
// In lib.d.ts:
interface Array<T> {
  readonly [n: number]: T;

  map<U>(
    callbackfn: (value: T, index: number, array: readonly T[]) => U,
    thisArg?: any
  ): U[];
}
```

The property accessor—the `readonly [n: number]: T` line—tells TypeScript that you can access array properties by numeric index, but they're read-only. That should make `arr[1]` possible but `arr[1] = 9` impossible.

The `map` method definition is copied from the TypeScript source code with no changes (other than some auto-formatting). That should make it possible to call `arr.map()`.

Notice that I did *not* define push. We shouldn't be calling that on an immutable array!

I ran `tsc` again and…success! No errors! We now have immutable arrays!

At this stage, I've shown that **it's possible to configure TypeScript to make all arrays immutable with no extra annotations**. No need for `readonly string[]` or

ReadonlyArray<number>! In other words, we have some immutability by default.

This code, like everything in this post, is simplistic. There are lots of other array methods, like filter() and join() and forEach()! If this were made production-ready, I'd make sure to define all the read-only array methods.

But for now, I was ready to move on to mutable arrays.

Step 4: mutable arrays

I prefer immutability, but I want to be able to define a mutable array sometimes. So I made another test case:

```
// In test.ts:
const arr = [1, 2, 3] as MutableArray<number>;
arr[0] = 9;
arr.push(4);
```

Notice that this requires a little extra work to make the array mutable. In other words, it's not the default.

TypeScript complained that it can't find MutableArray, so I defined it:

```
// In lib.d.ts:
interface MutableArray<T> extends Array<T> {
  [n: number]: T;
  push(...items: T[]): number;
}
```

And again, type-checks passed!

Now, I had mutable and immutable arrays, with immutability as the default. Again, this is simplistic, but good enough for this proof-of-concept!

This was exciting to me. It was possible to configure TypeScript to be immutable by default, for arrays at least. I didn't have to fork the language or use any other tools.

Could I make more things immutable?

Step 5: the same for Record

I wanted to see if I could go beyond arrays. My next target was the Record type, which is a TypeScript utility type. So I defined another pair of test cases similar to the ones I made for arrays:

```
// In test.ts:

// Immutable records
const obj1: Record<string, string> = { foo: "bar" };
console.log(obj1.foo);
// @ts-expect-error Mutation should not be allowed.
obj1.foo = "baz";

// Mutable records
const obj2: MutableRecord<string, string> = { foo: "bar" };
obj2.foo = "baz";
```

TypeScript complained that it couldn't find Record or `MutableRecord`. It also complained about an unused `@ts-expect-error`, which meant that mutation was allowed.

I rolled up my sleeves and fixed those errors like this:

```
// In lib.d.ts:
declare type PropertyKey = string | number | symbol;
type Record<KeyT extends PropertyKey, ValueT> = {
  readonly [key in KeyT]: ValueT;
};
type MutableRecord<KeyT extends PropertyKey, ValueT> = {
  [key in KeyT]: ValueT;
};
```

Now, we have Record, which is an immutable key-value pair, and the mutable version too. Just like arrays!

You can imagine extending this idea to other built-in types, like `Set` and `Map`. I think it’d be pretty easy to do this the same way I did arrays and records. I’ll leave that as an exercise to the reader.

Failed step 6: plain objects

My final test was to make regular objects (not records or arrays) immutable. Unfortunately for me, I could not figure this out.

Here’s the test case I wrote:

```
// In test.ts:
const obj = { foo: "bar" };
console.log(obj.foo);
// @ts-expect-error Mutation should not be allowed.
obj.foo = "baz";
```

This stumped me. No matter what I did, I could not write a type that would disallow this mutation. I tried modifying the `Object` type every way I could think of, but came up short!

There are ways to annotate `obj` to make it immutable, but that’s not in the spirit of my goal. I want it to be immutable by default!

Alas, this is where I gave up.

Can you figure this out?

I wanted to make TypeScript immutable by default. I was able to do this with arrays, `Record`s, and other types like `Map` and `Set`. Unfortunately, I couldn't make it work for plain object definitions like `obj = { foo: "bar" }`.

There's probably a way to enforce this with lint rules, either by disallowing mutation operations or by requiring `Readonly` annotations everywhere. I'd like to see what that looks like.

If *you* figure out how to make TypeScript immutable by default *with no other tools*, I would love to know, and I'll update my post. I hope my failed attempt will lead someone else to something successful.

Again, please contact me if you figure this out, or have any other thoughts.