# TypeScript 2.7

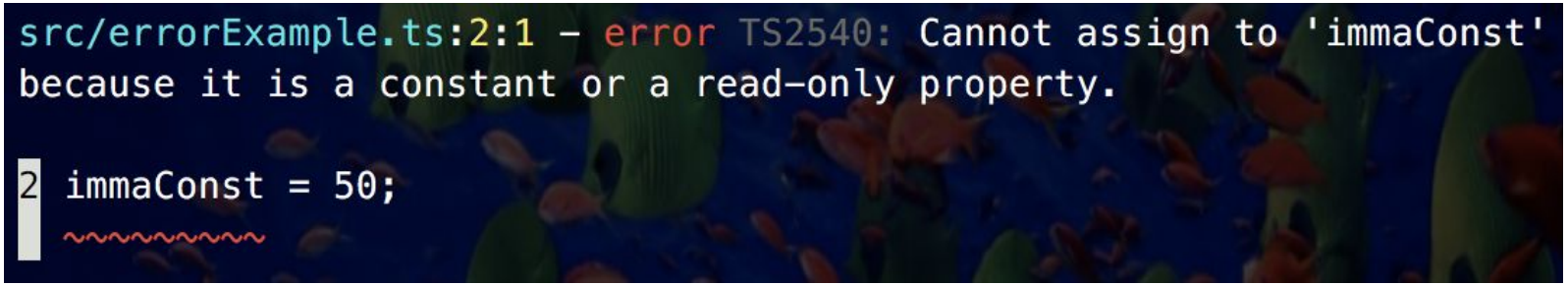
# Prettier Errors

Easy!

# tsc --pretty

---

**TS 2.7:**




**TS 2.6:**

https://github.com/JKillian/new-in-TS2.7/blob/master/src/errorExample.ts

# Numeric Separators

Easy!

```
const costPerBitcoin = 1500000000
```

```
const costPerBitcoin = 1_500_000_000
```

# ‘in’ Type Guards

Medium.

https://github.com/JKillian/new-in-TS2.7/blob/master/src/inTypeguard.ts

# Object Literal Type Inference

Medium.

https://github.com/JKillian/new-in-TS2.7/blob/master/src/objectLiteralType.ts

--strictPropertyInitialization

Medium.

https://github.com/JKillian/new-in-TS2.7/blob/master/src/strictPropertyInitialization.ts

https://github.com/JKillian/new-in-TS2.7/blob/master/src/definiteAssignment.ts

# Fixed Length Tuples

Medium.

https://github.com/JKillian/new-in-TS2.7/blob/master/src/tupleLength.ts

unique symbol

**Hard**

# What is a symbol?

---

> The **symbol** data type is highly specialized in purpose, and remarkable for its lack of versatility

> A **symbol** value may be used as an identifier for object properties; this is the data type's only purpose.

# What is a symbol?

---

```
1  Symbol("foo") !== Symbol("foo")
2  const foo = Symbol()
3  const bar = Symbol()
4  typeof foo === "symbol"
5  typeof bar === "symbol"
6  let obj = {}
7  obj[foo] = "foo"
8  obj[bar] = "bar"
9  JSON.stringify(obj) // {}
10 Object.keys(obj) // []
11 Object.getOwnPropertyNames(obj) // []
12 Object.getOwnPropertySymbols(obj) // [ Symbol(), Symbol() ]
```

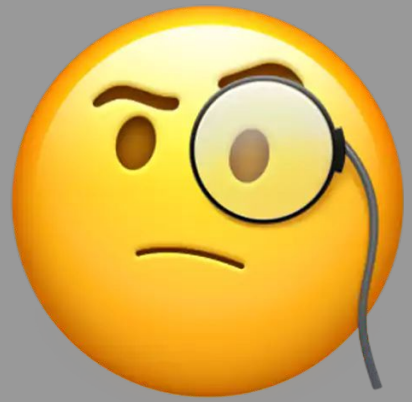
https://github.com/JKillian/new-in-TS2.7/blob/master/src/uniqueSymbol.ts

# Const-named Properties

**Hard**

https://github.com/JKillian/new-in-TS2.7/blob/master/src/constantProperties.ts

# Improved Class Type-Narrowing



**Hard**

# Improved Class Type-Narrowing

---

- Structurally identical, but distinct, class types are now preserved in union types (instead of eliminating all but one).
- Union type subtype reduction only removes a class type if it is a subclass of *and* derives from another class type in the union.
- Type checking of the instanceof operator is now based on whether the type of the left operand *derives from* the type indicated by the right operand (as opposed to a structural subtype check).

# Improved Class Type-Narrowing

```
class A {}
class B extends A {}
class C extends A {}
class D extends A { c: string }
class E extends D {}
```

```
let a1 = [new A(), new B(), new C(), new D(), new E()];  // A[]
let a2 = [new B(), new C(), new D(), new E()];  // (B | C | D)[] (previously B[])

function f1(x: B | C | D) {
    if (x instanceof B) {
        x;  // B (previously B | D)
    }
    else if (x instanceof C) {
        x;  // C
    }
    else {
        x;  // D (previously never)
    }
}
```

# --esModuleInterop

Messy!

# --esModuleInterop

TypeScript 2.7 updates CommonJS/AMD/UMD module emit to synthesize namespace records based on the presence of an `__esModule` indicator under `--esModuleInterop`. The change brings the generated output from TypeScript closer to that generated by Babel.

Previously CommonJS/AMD/UMD modules were treated in the same way as ES6 modules, resulting in a couple of problems. Namely:

- TypeScript treats a namespace import (i.e. `import * as foo from "foo"`) for a CommonJS/AMD/UMD module as equivalent to `const foo = require("foo")`. Things are simple here, but they don't work out if the primary object being imported is a primitive or a class or a function. ECMAScript spec stipulates that a namespace record is a plain object, and that a namespace import (`foo` in the example above) is not callable, though allowed by TypeScript
- Similarly a default import (i.e. `import d from "foo"`) for a CommonJS/AMD/UMD module as equivalent to `const d = require("foo").default`. Most of the CommonJS/AMD/UMD modules available today do not have a `default` export, making this import pattern practically unusable to import non-ES modules (i.e. CommonJS/AMD/UMD). For instance `import fs from "fs"` or `import express from "express"` are not allowed.

Under the new `--esModuleInterop` these two issues should be addressed:

- A namespace import (i.e. `import * as foo from "foo"`) is now correctly flagged as uncallabale. Calling it will result in an error.
- Default imports to CommonJS/AMD/UMD are now allowed (e.g. `import fs from "fs"`), and should work as expected.

> Note: The new behavior is added under a flag to avoid unwarranted breaks to existing code bases. We highly recommend applying it both to new and existing projects. For existing projects, namespace imports (`import * as express from "express"; express();`) will need to be converted to default imports (`import express from "express"; express();`).

# Thanks!

https://github.com/JKillian | jasonkillian.com

Credit and thanks to: MDN for their Symbol docs; the TypeScript team for their great docs on TS 2.7