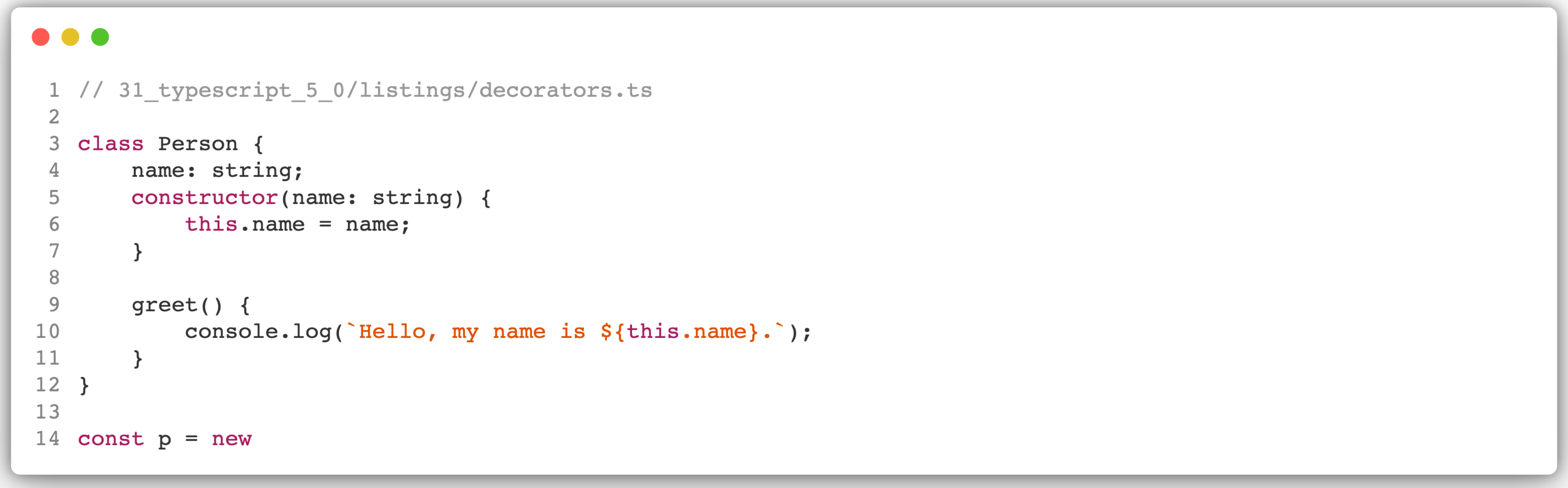
TypeScript 5.0

What's new in this major?

Decorators

Decorators are an **`upcoming ECMAScript`** feature that allow us to customize classes and their members in a **`reusable way`**.

Decorators

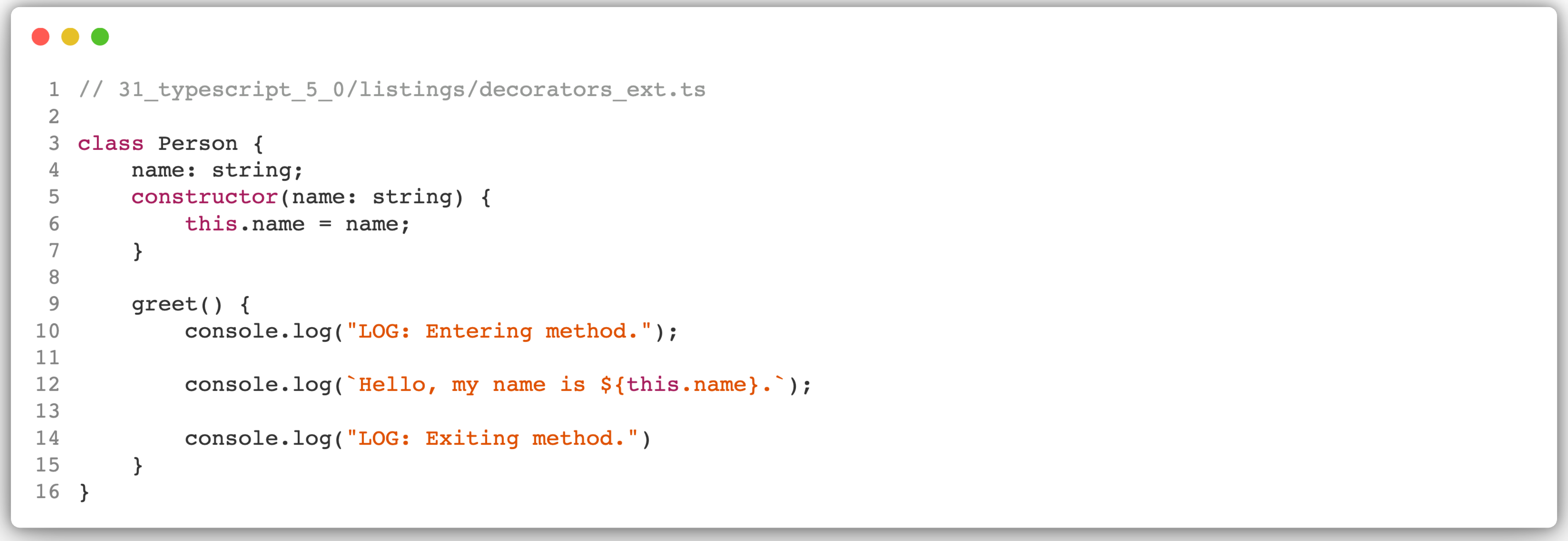


```
// 31_typescript_5_0/listings/decorators.ts

class Person {
    name: string;
    constructor(name: string) {
        this.name = name;
    }

    greet() {
        console.log(`Hello, my name is ${this.name}.`);
    }
}

const p = new
```

`greet` is pretty simple here, but let’s imagine it’s something way more complicated – maybe it does some async logic, it’s recursive, it has side effects, etc.

Decorators

Regardless of what kind of ball-of-mud you’re imagining, let’s say you throw in some `console.log` calls to help debug `greet`.

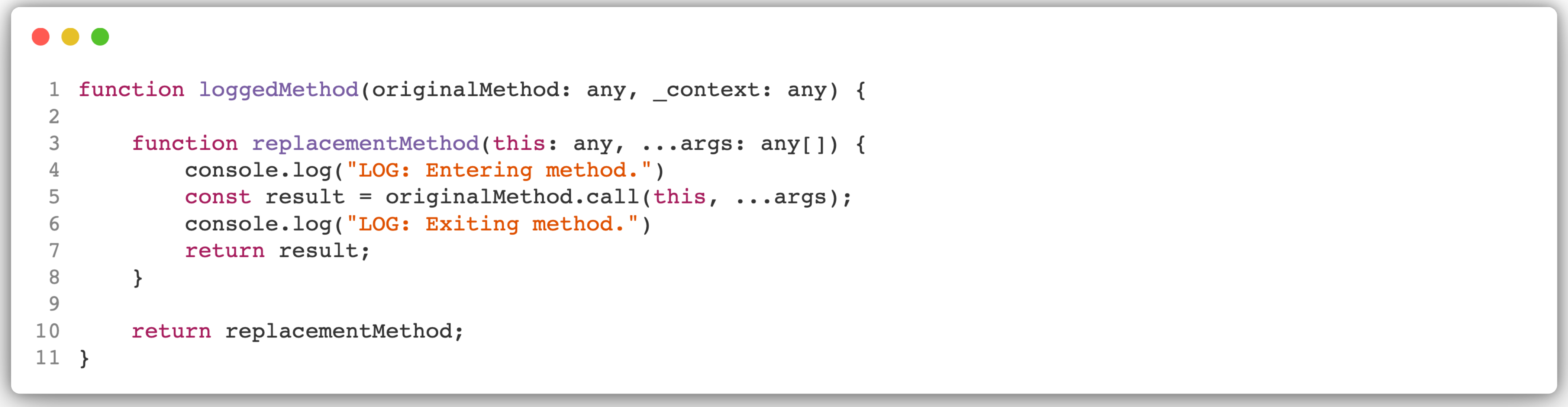


```
// 31_typescript_5_0/listings/decorators_ext.ts

class Person {
    name: string;
    constructor(name: string) {
        this.name = name;
    }

    greet() {
        console.log("LOG: Entering method.");

        console.log(`Hello, my name is ${this.name}.`);

        console.log("LOG: Exiting method.")
    }
}
```

Decorators

This pattern is fairly common. It sure would be nice if there was a way we could do this for every method!

This is where decorators come in. We can write a function called **`loggedMethod`** that looks like the following:



```
function loggedMethod(originalMethod: any, _context: any) {

    function replacementMethod(this: any, ...args: any[]) {
        console.log("LOG: Entering method.")
        const result = originalMethod.call(this, ...args);
        console.log("LOG: Exiting method.")
        return result;
    }

    return replacementMethod;
}
```

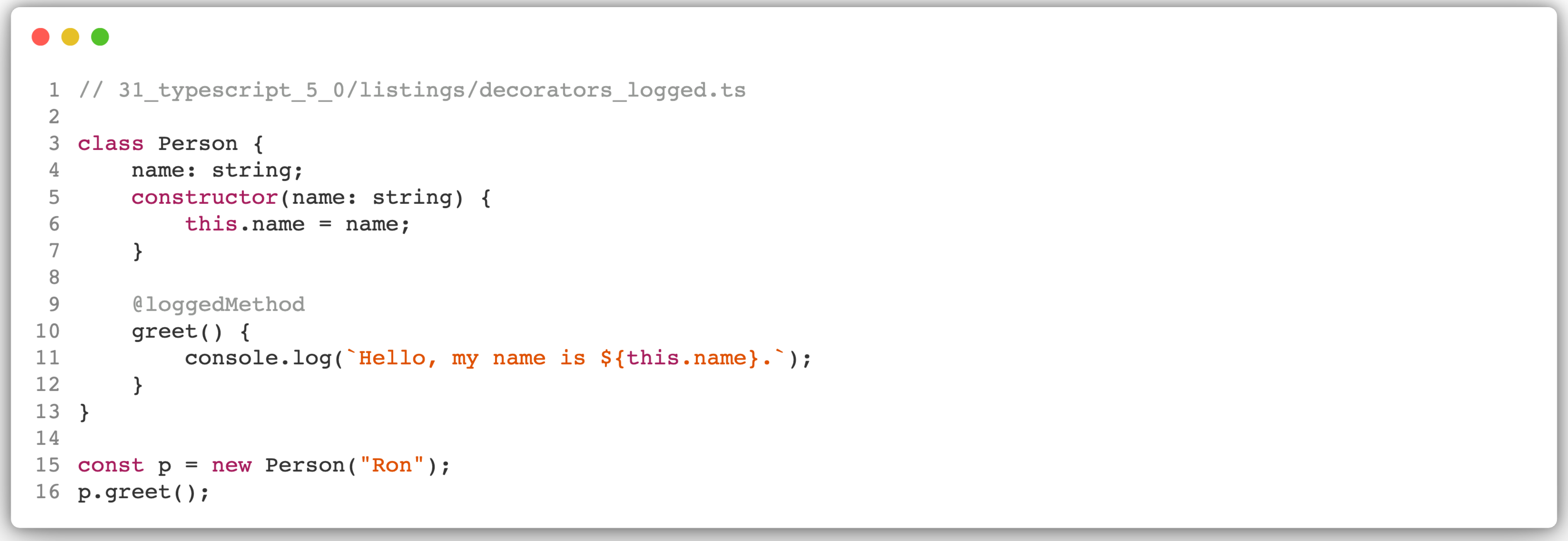
Decorators

"What's the deal with all of these **`anyS`**? What is this, **`anyScript`**!?"

Just be patient – we're keeping things simple for now so that we can focus on what this function is doing.

Decorators

Now we can use **`loggedMethod`** to *decorate* the method `greet`:

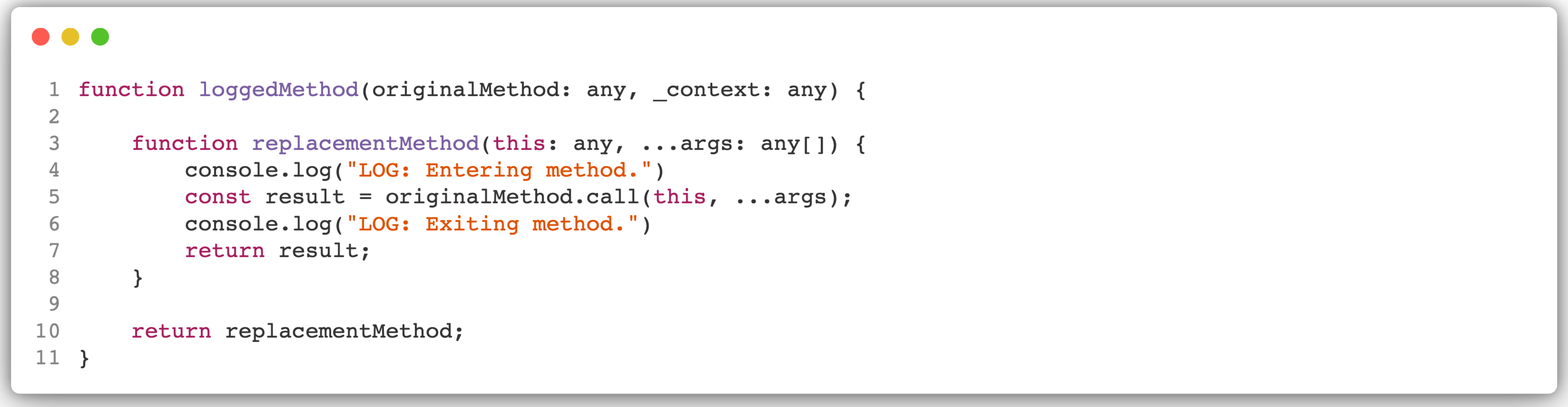


```typescript
// 31_typescript_5_0/listings/decorators_logged.ts

class Person {
    name: string;
    constructor(name: string) {
        this.name = name;
    }

    @loggedMethod
    greet() {
        console.log(`Hello, my name is ${this.name}.`);
    }
}

const p = new Person("Ron");
p.greet();
```

Decorators

This pattern is fairly common. It sure would be nice if there was a way we could do this for every method!

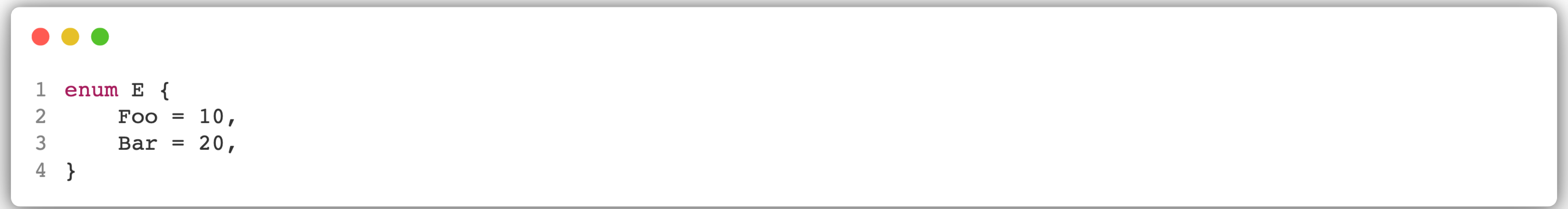
This is where decorators come in. We can write a function called **`loggedMethod`** that looks like the following:



```
function loggedMethod(originalMethod: any, _context: any) {

    function replacementMethod(this: any, ...args: any[]) {
        console.log("LOG: Entering method.")
        const result = originalMethod.call(this, ...args);
        console.log("LOG: Exiting method.")
        return result;
    }

    return replacementMethod;
}
```

All enums Are Union enums

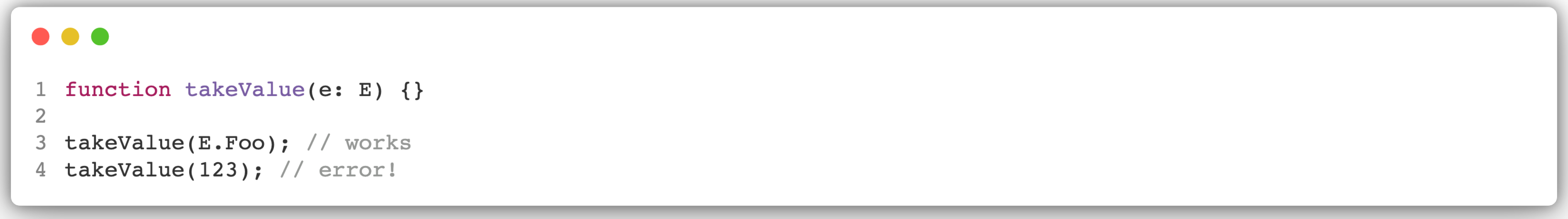
When TypeScript originally introduced enums, they were nothing more than a set of numeric constants with the same type.



```
enum E {
    Foo = 10,
    Bar = 20,
}
```

All `enums` Are Union `enums`

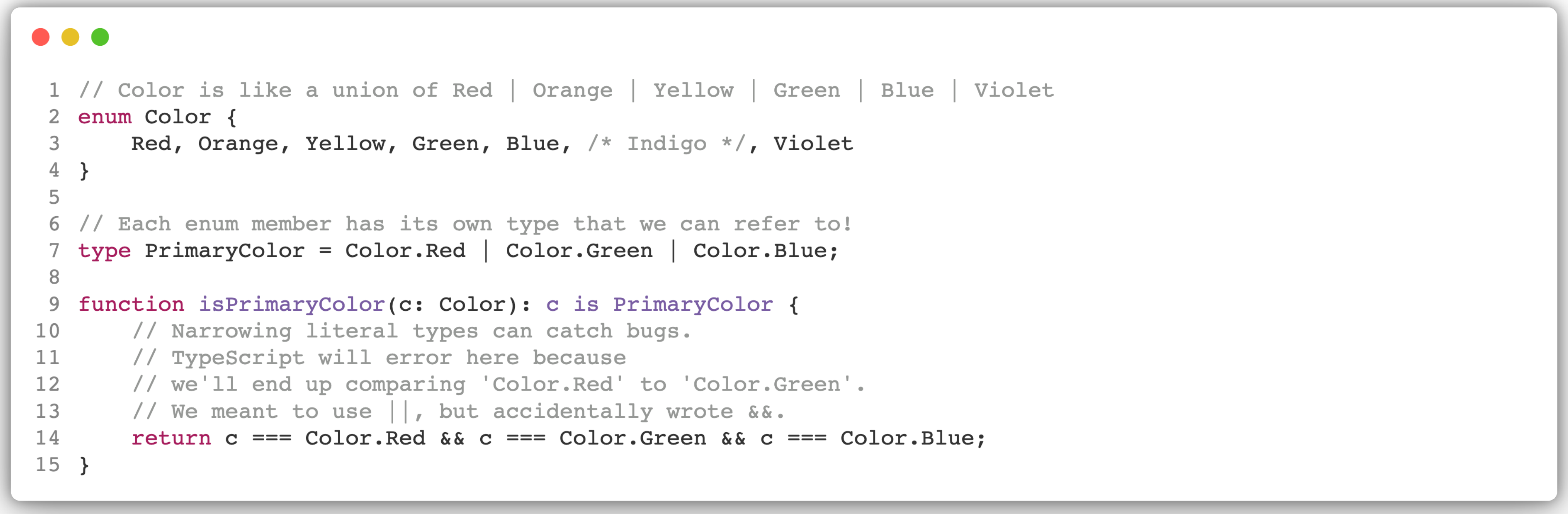
The only thing special about **`E.Foo`** and **`E.Bar`** was that they were assignable to anything expecting the type **`E`**. Other than that, they were pretty much just `numbers`.



```
function takeValue(e: E) {}

takeValue(E.Foo); // works
takeValue(123); // error!
```

All enums Are Union enums

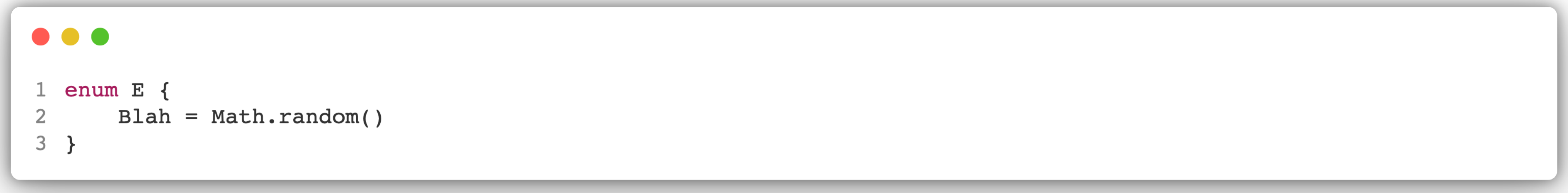
It wasn’t until TypeScript 2.0 introduced enum literal types that enums got a bit more special. Enum literal types gave each enum member its own type, and turned the enum itself into a *union* of each member type.



```
// Color is like a union of Red | Orange | Yellow | Green | Blue | Violet
enum Color {
    Red, Orange, Yellow, Green, Blue, /* Indigo */, Violet
}

// Each enum member has its own type that we can refer to!
type PrimaryColor = Color.Red | Color.Green | Color.Blue;

function isPrimaryColor(c: Color): c is PrimaryColor {
    // Narrowing literal types can catch bugs.
    // TypeScript will error here because
    // we'll end up comparing 'Color.Red' to 'Color.Green'.
    // We meant to use ||, but accidentally wrote &&.
    return c === Color.Red && c === Color.Green && c === Color.Blue;
}
```

All enums Are Union enums

One issue with giving **each enum member its own type** was that those types were in some part associated with the actual value of the member. In some cases it’s not possible to compute that value – for instance, an enum member could be initialized by a function call.



```
enum E {
    Blah = Math.random()
}
```

All `enums` Are Union `enums`

TypeScript 5.0 manages to **`make all enums into union enums by creating a unique type for each computed member`**. That means that all enums can now be narrowed and have their members referenced as types as well.

End

That was all for this chapter