TypeScript 4.9

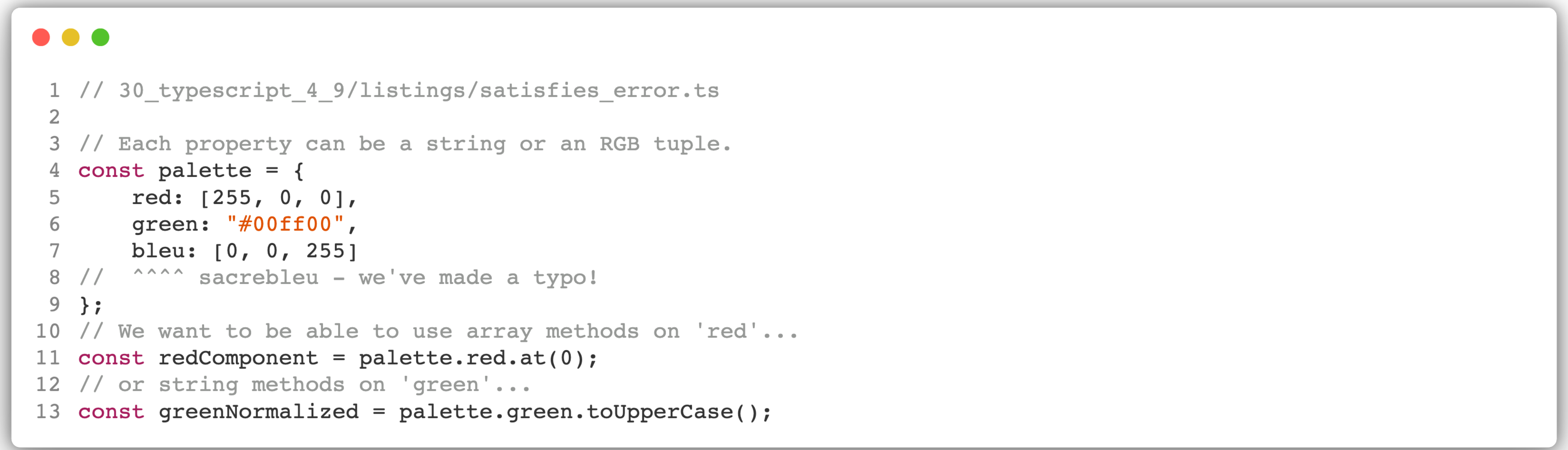
What's new?

The `satisfies` Operator

TypeScript developers are often faced with a dilemma:

we want to ensure that some **`expression`** ***`matches`*** some type, but also **`want to keep the`** ***`most specific`*** **`type`** of that expression for inference purposes.

The `satisfies` Operator



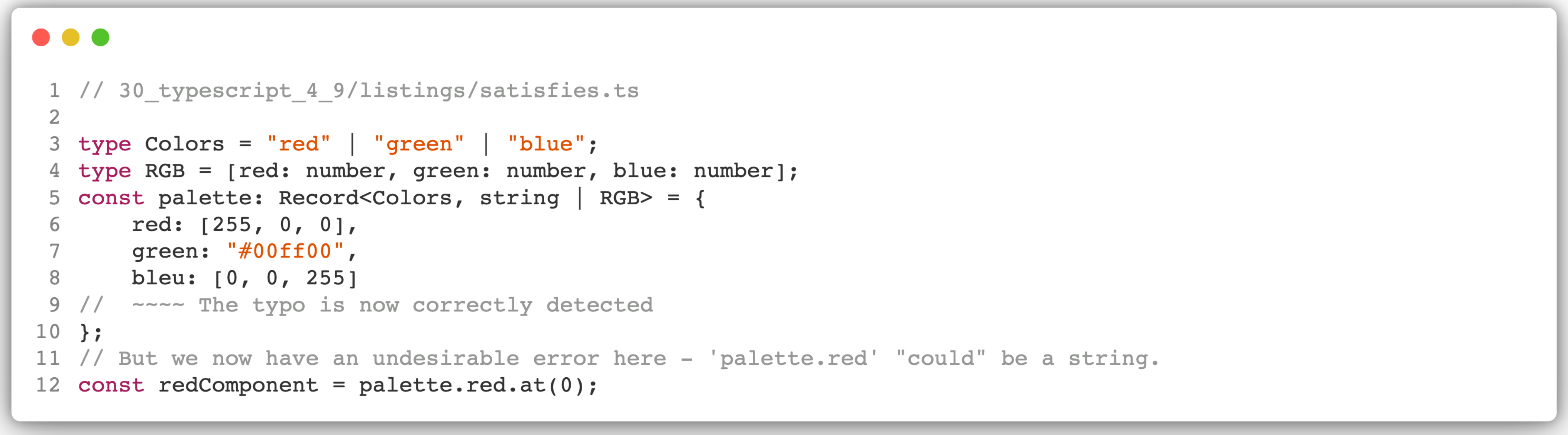
```typescript
// 30_typescript_4_9/listings/satisfies_error.ts

// Each property can be a string or an RGB tuple.
const palette = {
    red: [255, 0, 0],
    green: "#00ff00",
    bleu: [0, 0, 255]
//  ^^^^ sacrebleu - we've made a typo!
};
// We want to be able to use array methods on 'red'...
const redComponent = palette.red.at(0);
// or string methods on 'green'...
const greenNormalized = palette.green.toUpperCase();
```

The **`satisfies`** Operator

Notice that we've written **`bleu`**, whereas we probably should have written **`blue`**.

We could try to catch that bleu typo by using a type annotation on the palette, but we'd lose the information **`about each property`**.

The `satisfies` Operator



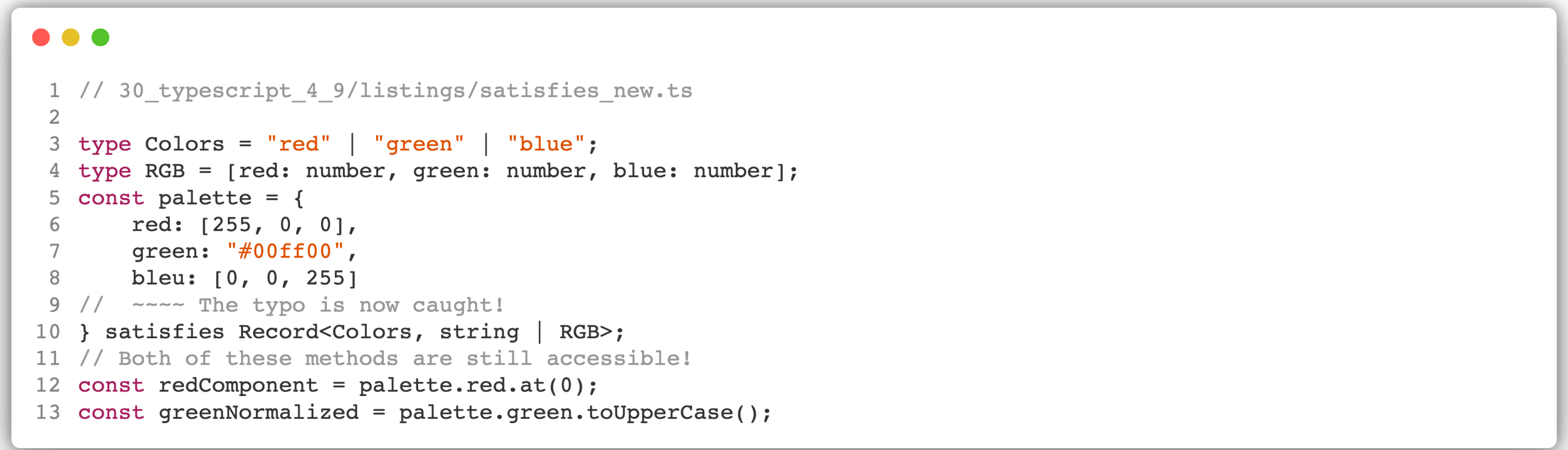
```typescript
// 30_typescript_4_9/listings/satisfies.ts

type Colors = "red" | "green" | "blue";
type RGB = [red: number, green: number, blue: number];
const palette: Record<Colors, string | RGB> = {
    red: [255, 0, 0],
    green: "#00ff00",
    bleu: [0, 0, 255]
//  ~~~~ The typo is now correctly detected
};
// But we now have an undesirable error here - 'palette.red' "could" be a string.
const redComponent = palette.red.at(0);
```

The `satisfies` Operator

The new `satisfies` operator lets us validate that the type of an expression matches some type without changing the resulting type of that expression.

As an example, we could use `satisfies` to validate that all the properties of the palette are compatible with `string | number[]`:

The **`satisfies`** Operator



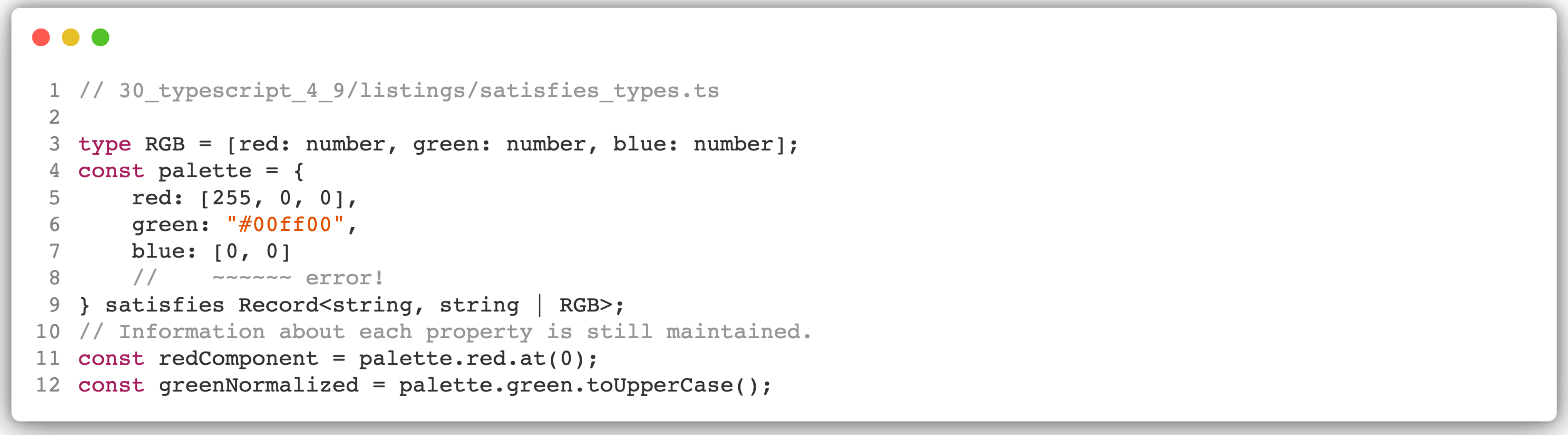
```typescript
// 30_typescript_4_9/listings/satisfies_new.ts

type Colors = "red" | "green" | "blue";
type RGB = [red: number, green: number, blue: number];
const palette = {
    red: [255, 0, 0],
    green: "#00ff00",
    bleu: [0, 0, 255]
//  ~~~~ The typo is now caught!
} satisfies Record<Colors, string | RGB>;
// Both of these methods are still accessible!
const redComponent = palette.red.at(0);
const greenNormalized = palette.green.toUpperCase();
```

The `satisfies` Operator

Maybe we don’t care if the property names match up somehow, but we do care about the types of each property.

In that case, we can also ensure that all of an object’s property values conform to some type.

The `satisfies` Operator



```typescript
// 30_typescript_4_9/listings/satisfies_types.ts

type RGB = [red: number, green: number, blue: number];
const palette = {
    red: [255, 0, 0],
    green: "#00ff00",
    blue: [0, 0]
    //    ~~~~~~ error!
} satisfies Record<string, string | RGB>;
// Information about each property is still maintained.
const redComponent = palette.red.at(0);
const greenNormalized = palette.green.toUpperCase();
```

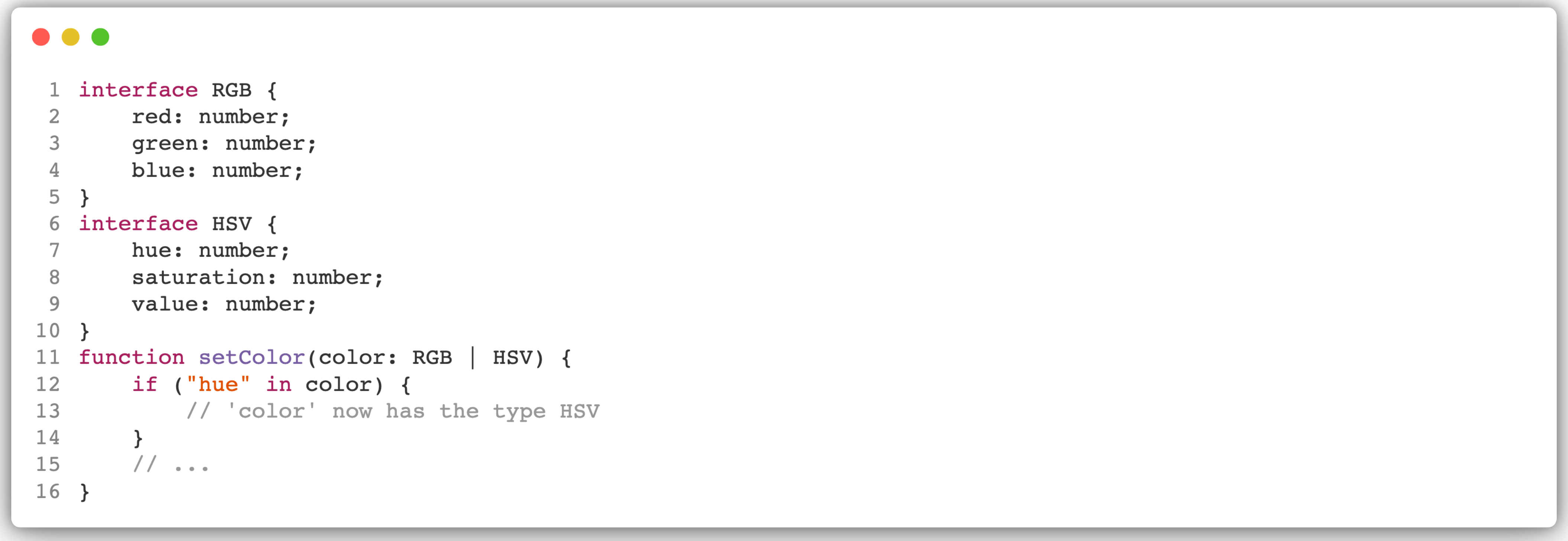
Unlisted Property Narrowing with the in Operator

As developers, we often need to deal with values that aren’t fully known at runtime.

JavaScript’s in operator can check whether a property exists on an object.

Unlisted Property Narrowing with the in Operator

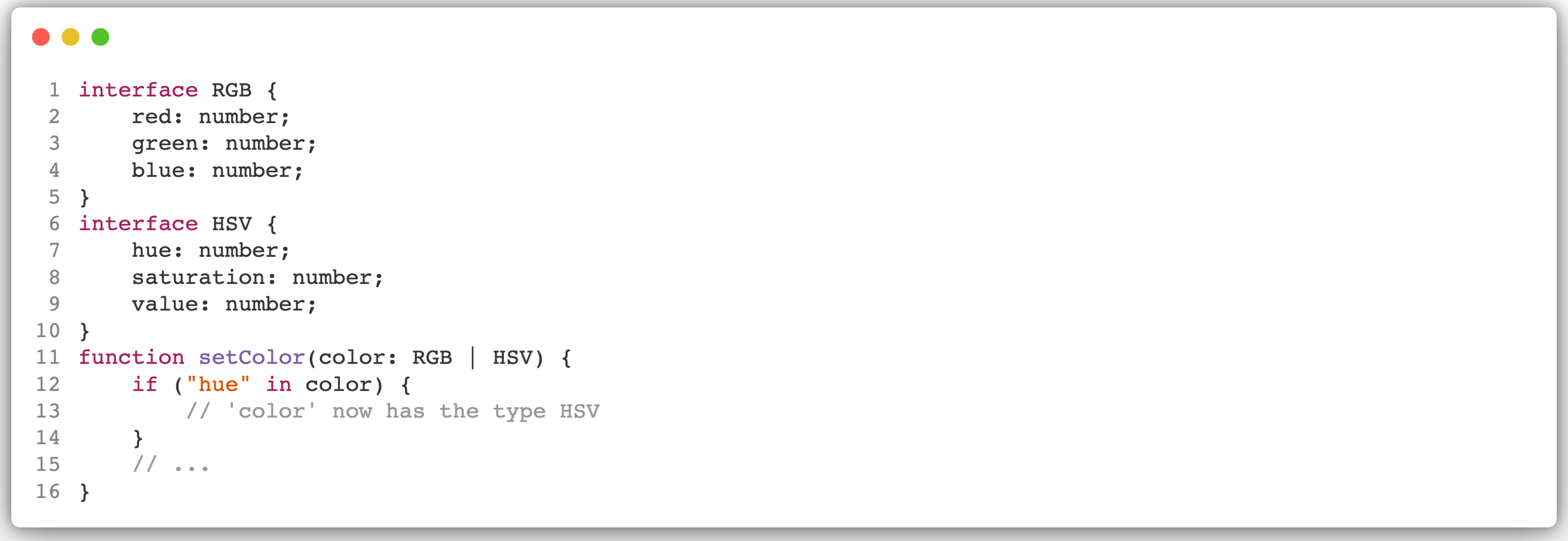
Previously, TypeScript allowed us to narrow away any types that don’t explicitly list a property.



```typescript
interface RGB {
    red: number;
    green: number;
    blue: number;
}
interface HSV {
    hue: number;
    saturation: number;
    value: number;
}
function setColor(color: RGB | HSV) {
    if ("hue" in color) {
        // 'color' now has the type HSV
    }
    // ...
}
```

Unlisted Property Narrowing with the in Operator

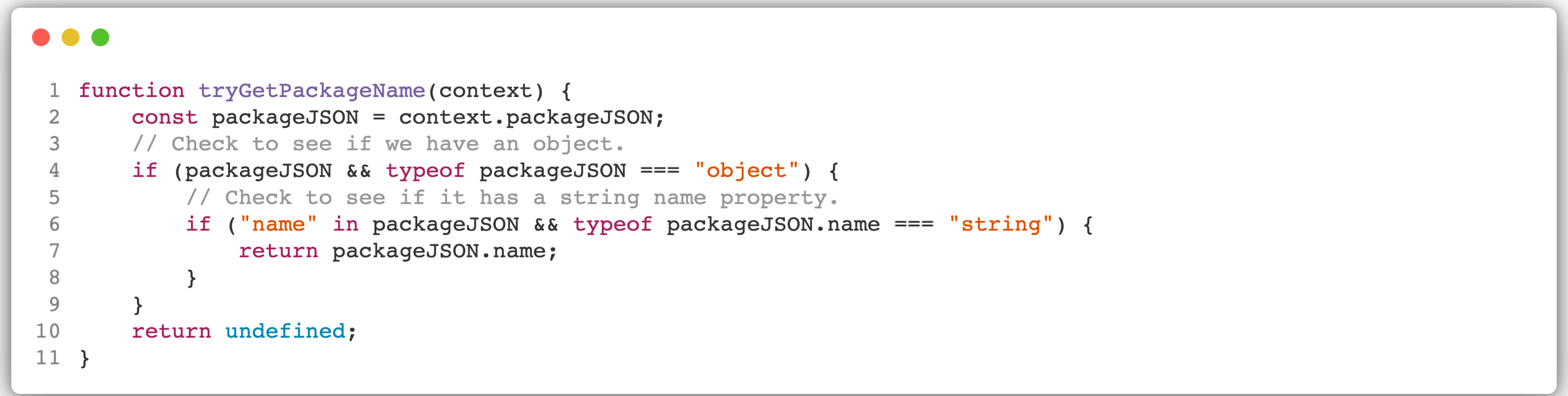
Here, the type RGB didn't list the hue and got narrowed away, and leaving us with the type HSV.



```typescript
interface RGB {
    red: number;
    green: number;
    blue: number;
}
interface HSV {
    hue: number;
    saturation: number;
    value: number;
}
function setColor(color: RGB | HSV) {
    if ("hue" in color) {
        // 'color' now has the type HSV
    }
    // ...
}
```

Unlisted Property Narrowing with the in Operator

But what about examples where no type listed a given property? In those cases, the language didn’t help us much. Let’s take the following example in JavaScript:



```javascript
function tryGetPackageName(context) {
    const packageJSON = context.packageJSON;
    // Check to see if we have an object.
    if (packageJSON && typeof packageJSON === "object") {
        // Check to see if it has a string name property.
        if ("name" in packageJSON && typeof packageJSON.name === "string") {
            return packageJSON.name;
        }
    }
    return undefined;
}
```

Unlisted Property Narrowing with the in Operator

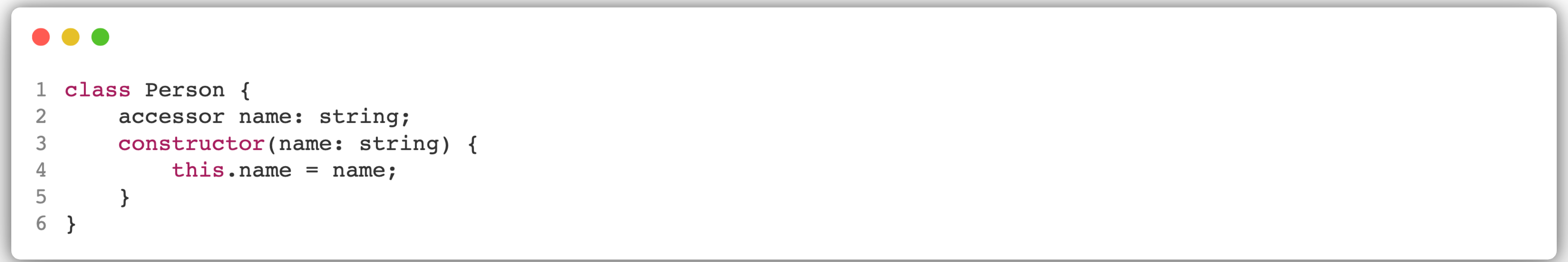
TypeScript 4.9 makes the **`in`**-operator more powerful when narrowing types that don’t list the property.

Instead of leaving them as-is, the language will intersect their types with **`Record<"property-key-being-checked", unknown>`**.

Auto-Accessors in Classes

TypeScript 4.9 supports an upcoming feature in ECMAScript called auto-accessors.

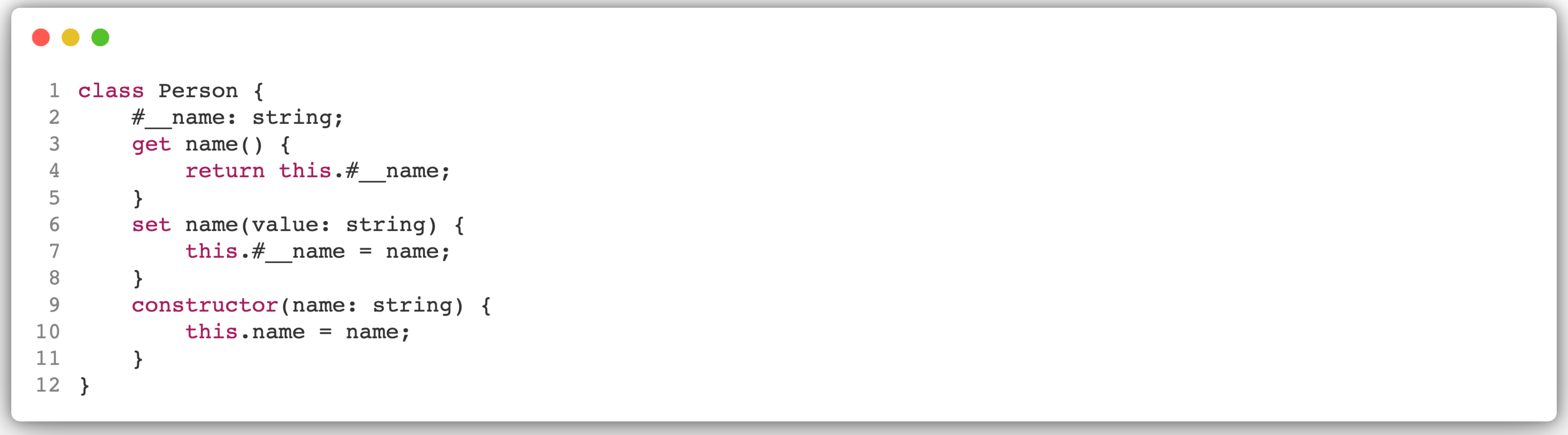
Auto-accessors are declared just like properties on classes, except that they're declared with the accessor keyword.



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

Auto-Accessors in Classes

Under the covers, these auto-accessors “de-sugar” to a get and set accessor with an unreachable private property.

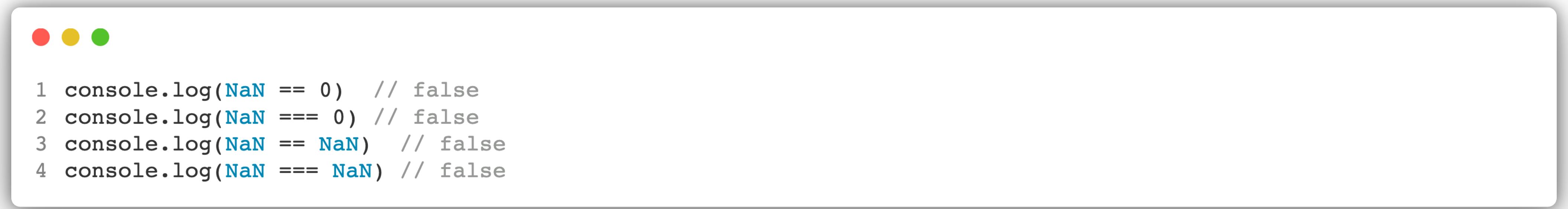


```
class Person {
    #__name: string;
    get name() {
        return this.#__name;
    }
    set name(value: string) {
        this.#__name = name;
    }
    constructor(name: string) {
        this.name = name;
    }
}
```

Checks For Equality on NaN

A major gotcha for JavaScript developers is checking against the value NaN using the built-in equality operators. For some background, NaN is a unique numeric value for “Not a Number.”

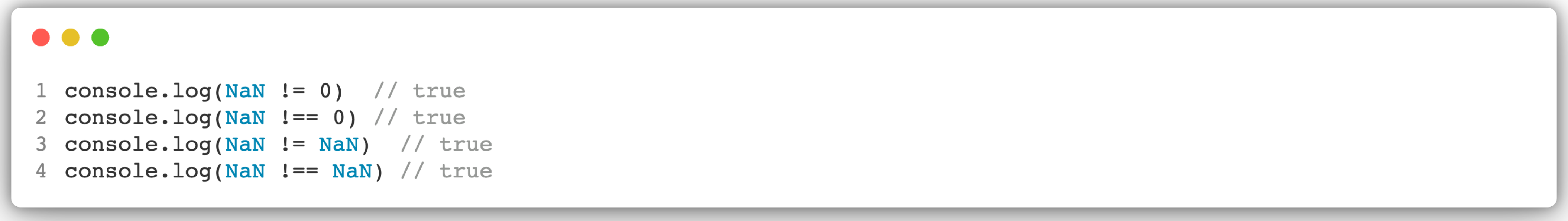
Nothing is ever equal to NaN - even NaN!



```
console.log(NaN == 0)  // false
console.log(NaN === 0) // false
console.log(NaN == NaN)  // false
console.log(NaN === NaN) // false
```

Checks For Equality on NaN

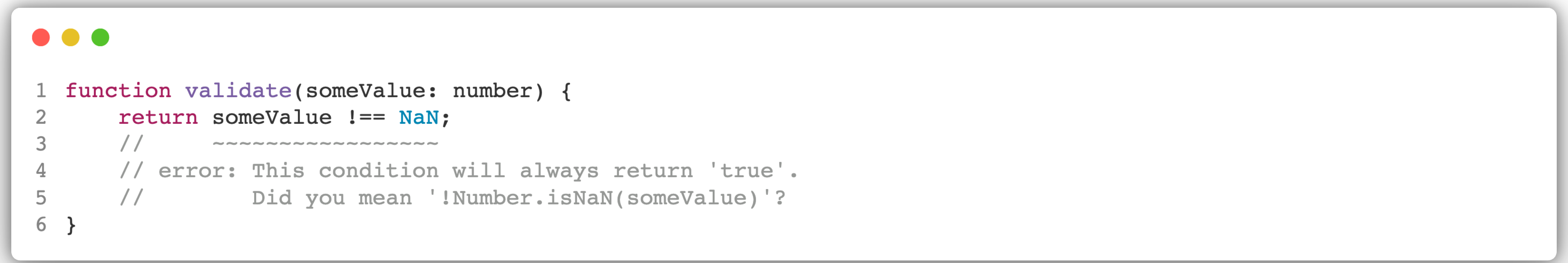
But at least symmetrically everything is always not-equal to NaN.



```
console.log(NaN != 0)  // true
console.log(NaN !== 0) // true
console.log(NaN != NaN)  // true
console.log(NaN !== NaN) // true
```

Checks For Equality on NaN

TypeScript now errors on direct comparisons against NaN, and will suggest using some variation of Number.isNaN instead.



```
function validate(someValue: number) {
    return someValue !== NaN;
    //     ~~~~~~~~~~~~~~~~~
    // error: This condition will always return 'true'.
    //        Did you mean '!Number.isNaN(someValue)'?
}
```

End

That was all for this chapter