What I didn't mention
What we didn’t speak about, was that the `toString()` method can return different things, depending on which object is calling the method.
What?

In other words, the `toString()` does different things, depending on where it comes from!
Weird right.

Lets start at the beginning

MDN states that:

Every object has a `toString()` method that is automatically called when the object is to be represented as a text value or when an object is referred to in a manner in which a string is expected.

In plain English: we automatically have access to the toString() method on pretty much everything.

Ok great, but what does the toString() method **actually** return

The answer depends on where it is inherited from.

Heh?

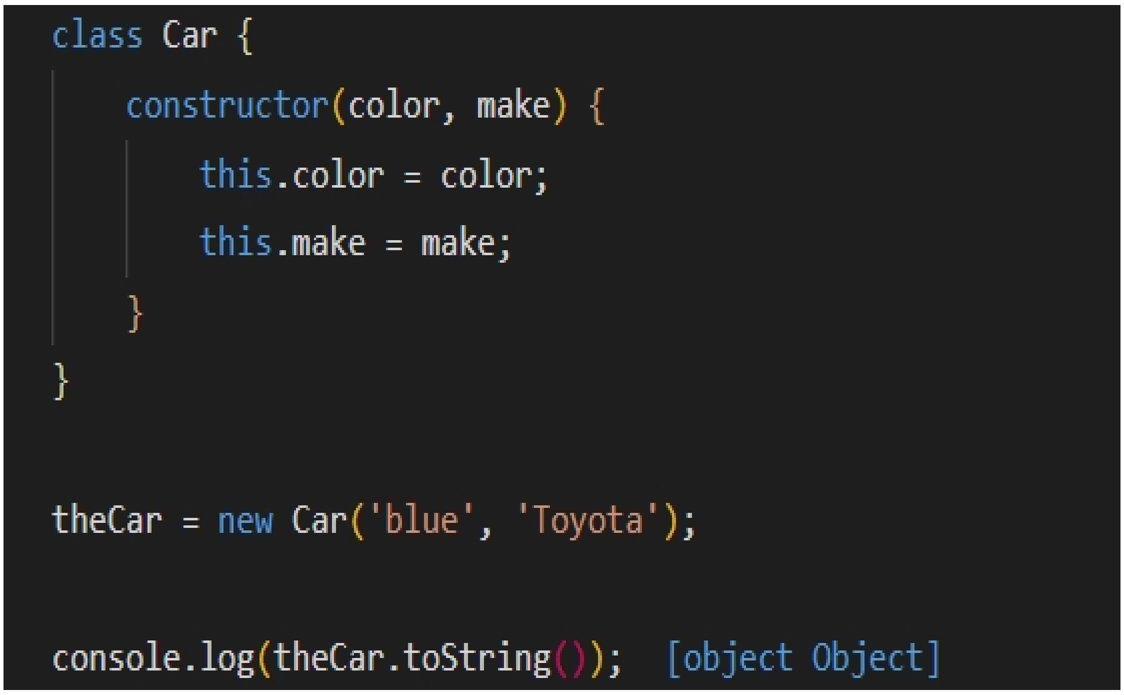
Let me explain. As MDN put it:

If this method is not overridden in a custom object, `toString()` returns "`[object type]`".

What is this ‘type’ value? It tells us what object type we’re dealing with (for example, a Date, String, Math object, etc.).

Want an example?

Lets define a Car class, and then create a car.



```
class Car {
    constructor(color, make) {
        this.color = color;
        this.make = make;
    }
}

theCar = new Car('blue', 'Toyota');

console.log(theCar.toString());  [object Object]
```

Whats interesting about the above code is that if you call the `toString()` method on this custom object, it returns the default value inherited from the Object prototype. In other words, its telling us that our object inherits from the Object prototype.
Pretty useless for our purposes right?

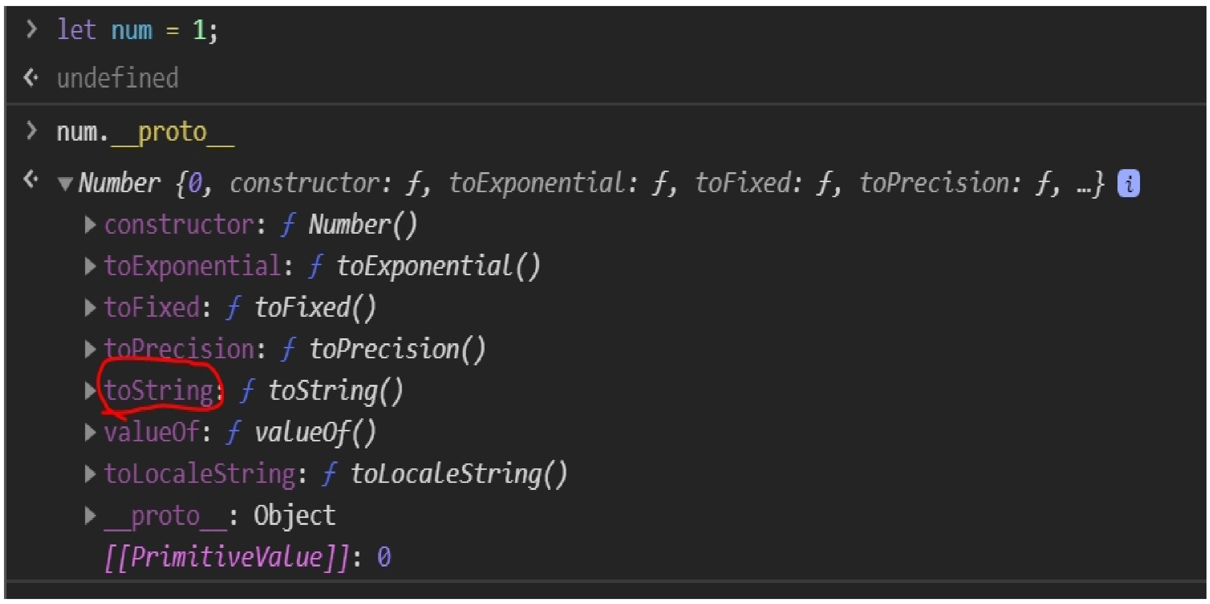
We are doomed … or are we?

There's an exception

By default, the `toString()` method is inherited by every object descended from Object. And unless the toString() method is not overridden in a custom object, `toString()` returns "`[object type]`".
The keywords here are **unless its overridden** .
Remember our example. We used the `Math.random()` method to create a random Number.
This means that we have returned to us a Number object.

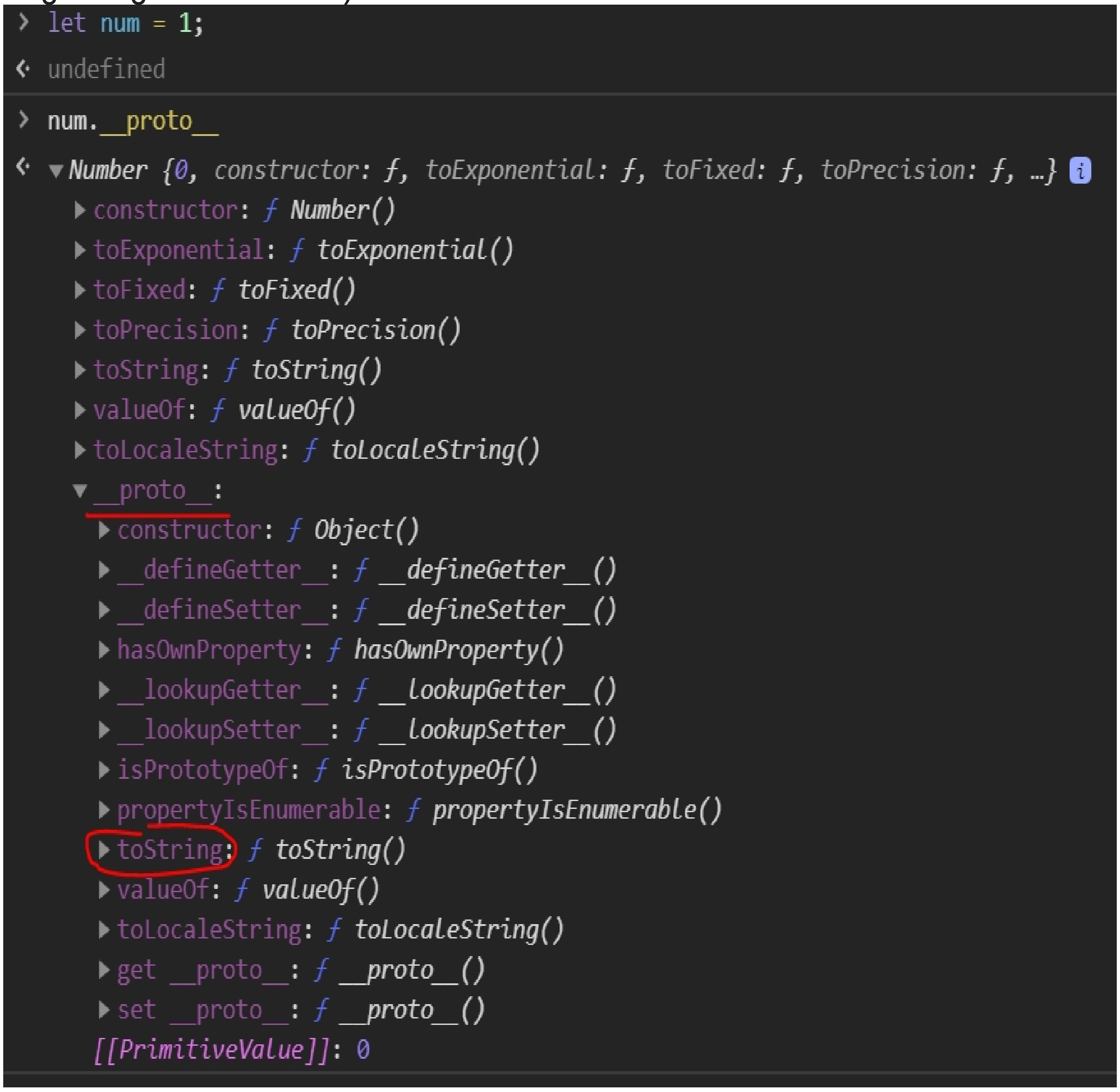
And the good news?

The Number object overrides the `toString()` method of the Object object.
In fact, if we console out a number, we can see it has its own `toString()` method:



```
> let num = 1;
< undefined
> num.__proto__
< ▼Number {0, constructor: ƒ, toExponential: ƒ, toFixed: ƒ, toPrecision: ƒ, …} i
    ▶constructor: ƒ Number()
    ▶toExponential: ƒ toExponential()
    ▶toFixed: ƒ toFixed()
    ▶toPrecision: ƒ toPrecision()
    ▶toString: ƒ toString()
    ▶valueOf: ƒ valueOf()
    ▶toLocaleString: ƒ toLocaleString()
    ▶__proto__: Object
     [[PrimitiveValue]]: 0
```

This is different to the one inherited by the Object object. If we look at the `__proto__` of the Number object, we see it also has the `toString()` method (this is the same one we spoke about at the beginning of this article).



```
> let num = 1;
< undefined
> num.__proto__
< ▼Number {0, constructor: ƒ, toExponential: ƒ, toFixed: ƒ, toPrecision: ƒ, …} i
    ▶constructor: ƒ Number()
    ▶toExponential: ƒ toExponential()
    ▶toFixed: ƒ toFixed()
    ▶toPrecision: ƒ toPrecision()
    ▶toString: ƒ toString()
    ▶valueOf: ƒ valueOf()
    ▶toLocaleString: ƒ toLocaleString()
    ▼__proto__:
      ▶constructor: ƒ Object()
      ▶__defineGetter__: ƒ __defineGetter__()
      ▶__defineSetter__: ƒ __defineSetter__()
      ▶hasOwnProperty: ƒ hasOwnProperty()
      ▶__lookupGetter__: ƒ __lookupGetter__()
      ▶__lookupSetter__: ƒ __lookupSetter__()
      ▶isPrototypeOf: ƒ isPrototypeOf()
      ▶propertyIsEnumerable: ƒ propertyIsEnumerable()
      ▶toString: ƒ toString()
      ▶valueOf: ƒ valueOf()
      ▶toLocaleString: ƒ toLocaleString()
      ▶get __proto__: ƒ __proto__()
      ▶set __proto__: ƒ __proto__()
     [[PrimitiveValue]]: 0
```

Whats my point?

We generated a random **number**.

This means that the number returned to us does *not* inherit Object.prototype.toString(). For Number objects, the toString() method returns something different.
It returns a string representation of the object in the specified *radix*.
Don't stress. The radix is just a fancy word for how many unique digits the numerical system has. For example, a radix of 2 would represent a binary system because the binary system only has 2 digits. A radix of 10 would represent the Decimal system because the Decimal system uses ten digits from 0 through to 9.
We used a base of 16 as the hexadecimal numerical system has 16 different types of digits.

This is why we used base 16 in our `toString()` method.
Hope this clarifies things up for you.