EN

EPUB/PDF

→ The JavaScript language → Data types

18th December 2019

Methods of primitives

JavaScript allows us to work with primitives (strings, numbers, etc.) as if they were objects. They also provide methods to call as such. We will study those soon, but first we'll see how it works because, of course, primitives are not objects (and here we will make it even clearer).

Let's look at the key distinctions between primitives and objects.

A primitive

- Is a value of a primitive type.
- There are 7 primitive types: `string` , `number` , `bigint` , `boolean` , `symbol` , `null` and `undefined` .

An object

- Is capable of storing multiple values as properties.
- Can be created with `{}` , for instance: `{name: "John", age: 30}` . There are other kinds of objects in JavaScript: functions, for example, are objects.

One of the best things about objects is that we can store a function as one of its properties.

```
let john = {
  name: "John",
  sayHi: function() {
    alert("Hi buddy!");
  }
};

john.sayHi(); // Hi buddy!
```

So here we've made an object `john` with the method `sayHi` .

Many built-in objects already exist, such as those that work with dates, errors, HTML elements, etc. They have different properties and methods.

But, these features come with a cost!

Objects are "heavier" than primitives. They require additional resources to support the internal machinery.

A primitive as an object

Here's the paradox faced by the creator of JavaScript:

- There are many things one would want to do with a primitive like a string or a number. It would be great to access them as methods.
- Primitives must be as fast and lightweight as possible.

The solution looks a little bit awkward, but here it is:

1. Primitives are still primitive. A single value, as desired.
2. The language allows access to methods and properties of strings, numbers, booleans and symbols.
3. In order for that to work, a special “object wrapper” that provides the extra functionality is created, and then is destroyed.

The “object wrappers” are different for each primitive type and are called: `String`, `Number`, `Boolean` and `Symbol`. Thus, they provide different sets of methods.

For instance, there exists a string method str.toUpperCase() that returns a capitalized `str`.

Here’s how it works:

```
let str = "Hello";

alert( str.toUpperCase() ); // HELLO
```

Simple, right? Here’s what actually happens in `str.toUpperCase()`:

1. The string `str` is a primitive. So in the moment of accessing its property, a special object is created that knows the value of the string, and has useful methods, like `toUpperCase()`.
2. That method runs and returns a new string (shown by `alert`).
3. The special object is destroyed, leaving the primitive `str` alone.

So primitives can provide methods, but they still remain lightweight.

The JavaScript engine highly optimizes this process. It may even skip the creation of the extra object at all. But it must still adhere to the specification and behave as if it creates one.

A number has methods of its own, for instance, toFixed(n) rounds the number to the given precision:

```
let n = 1.23456;

alert( n.toFixed(2) ); // 1.23
```

We’ll see more specific methods in chapters Numbers and Strings.

⚠ Constructors `String/Number/Boolean` are for internal use only

Some languages like Java allow us to explicitly create "wrapper objects" for primitives using a syntax like `new Number(1)` or `new Boolean(false)`.

In JavaScript, that's also possible for historical reasons, but highly **unrecommended**. Things will go crazy in several places.

For instance:

```
alert( typeof 0 ); // "number"

alert( typeof new Number(0) ); // "object"!
```

Objects are always truthy in `if`, so here the alert will show up:

```
let zero = new Number(0);

if (zero) { // zero is true, because it's an object
  alert( "zero is truthy!?!" );
}
```

On the other hand, using the same functions `String/Number/Boolean` without `new` is a totally sane and useful thing. They convert a value to the corresponding type: to a string, a number, or a boolean (primitive).

For example, this is entirely valid:

```
let num = Number("123"); // convert a string to number
```

⚠ null/undefined have no methods

The special primitives `null` and `undefined` are exceptions. They have no corresponding "wrapper objects" and provide no methods. In a sense, they are "the most primitive".

An attempt to access a property of such value would give the error:

```
alert(null.test); // error
```

Summary

- Primitives except `null` and `undefined` provide many helpful methods. We will study those in the upcoming chapters.
- Formally, these methods work via temporary objects, but JavaScript engines are well tuned to optimize that internally, so they are not expensive to call.

Tasks

Can I add a string property?

importance: 5

Consider the following code:

```
let str = "Hello";

str.test = 5;

alert(str.test);
```

How do you think, will it work? What will be shown?

solution

Previous lesson Next lesson

Share Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)