**Methods of primitives**

JavaScript allows us to work with primitives (strings, numbers etc) as if they were objects.

They also provide methods to call and 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 more clear).

Let’s look at the key distinction between primitives and objects.

A primitive

**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, e.g. functions 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. But as properties and methods are very useful in programming, JavaScript engines try to optimize them to reduce the additional burden.

**[A primitive as an object](https://javascript.info/primitives-methods" \l "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. When this happens, a special “object wrapper” is created that provides the extra functionality, 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 method [str.toUpperCase()](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/String/toUpperCase) that returns a capitalized string.

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)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/toFixed) 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](https://javascript.info/number) and [Strings](https://javascript.info/string).

**Constructors String/Number/Boolean are for internal use only**

Some languages like Java allow us to create “wrapper objects” for primitives explicitly 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 1 ); // "number"

alert( typeof new Number(1) ); // "object"!

And because what follows, zero, is an object, 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](https://javascript.info/primitives-methods" \l "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.