

Lesson

Weekend

Introduction to Programming

(/introduction-to-programming)

/ Arrays and Looping (/introduction-to-programming/arrays-and-looping)

/ The Basics of Prototypes

Text

When we look at Mozilla's documentation for JavaScript methods, we'll often see the term **prototype**. We first learned about the term "prototype" in the last course section when we learned how to use MDN documentation for JavaScript (<https://www.learnhowtoprogram.com/introduction-to-programming/javascript-and-web-browsers/using-mdn-documentation-for-javascript>). Since we'll be learning how to use new methods on JavaScript arrays, now's a good time to review what prototype means.

What We Know So Far About Prototypes

Let's review what we learned in the last course section:

Remember, methods always belong to a specific data type, so it's important for us to always name the data type that a method belongs to, like in the example above. On MDN and elsewhere, the official syntax for the name of any JavaScript method is the following:

```
// This is not real code! This is just to highlight the  
official syntax for naming methods!  
dataType.prototype.methodName()
```

What's important about this syntax is that it always includes the data type that the method belongs to. Let's apply this syntax to the string methods we've learned about so far:

- `String.prototype.toUpperCase()`
- `String.prototype.toLowerCase()`
- `String.prototype.concat()`
- `String.prototype.charAt()`

And to the number methods we've learned about:

- `Number.prototype.toString()`
- `Number.prototype.toFixed()`

So what does the term `prototype` mean? For now, all we need to know is when we see `prototype` in the name of a method, it indicates that we're referencing a built-in (JavaScript-created) method that belongs to a specific object type, and any instance of that object type. In other words, we're not actually using (calling) the method — we're just referencing it by its official name.

For example, with `String.prototype.concat()`, we're referencing the `concat` method belonging to strings only. We're also indicating that this exact method belongs to all strings, meaning that `String.prototype.concat()` can be called on any string.

Going forward, we'll use the `dataType.prototype.methodName()` syntax when we're referencing JavaScript methods in order to clearly indicate which data type the method belongs to.

All of the above remains true, however it doesn't describe what a prototype really is in JavaScript. So, let's discuss exactly that! This will be a general overview — we'll also cover prototypes in greater depth in the next section on object-oriented JavaScript.

The Basics of Prototypes

According to the Mozilla documentation on object prototypes (https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes):

Prototypes are the mechanism by which JavaScript objects inherit features from one another.

This is really the key thing we need to know for now. Prototypes are just a way for an object to inherit functionality from another object in JavaScript.

All Objects Have Access to Prototypes

In the JavaScript Data Types (<https://www.learnhowtoprogram.com/introduction-to-programming/javascript-and-web-browsers/javascript-data-types>) lesson in the last section, we discussed how everything is either a primitive or an object. We've worked with just about all of the primitive types (other than symbols and bigint), including strings, numbers, and booleans. Since everything that isn't a primitive is an object, that means things like functions (and arrays!) are also a type of object.

We also learned that JavaScript implicitly turns certain primitives into objects. Why? In order to give them more complex functionality, including the ability to have properties and methods. The "Primitive" reference page (https://developer.mozilla.org/en-US/docs/Glossary/Primitive#primitive_wrapper_objects_in_javascript) on MDN describes this:

Except for `null` and `undefined`, all primitive values have object equivalents that wrap around the primitive values.

You don't ever need to worry about understanding JavaScript's process of creating wrapper objects since it's fairly esoteric, but the point here is just to clarify that we work with objects all of the time in JavaScript, and these objects all have access to prototypes, the mechanism that allows objects to inherit functionality from another object.

Examples

Prototypes is still an abstract concept, so let's look at examples to better understand prototypes. Let's use the DevTools console to create three different strings:

```
> const myName = "Remy";  
> const myFavFlower = "Petunias";  
> const myFavColor = "Green";
```

Now let's call the `String.prototype.toUpperCase()` method on each of these strings:

```
> myName.toUpperCase();  
"REMY"  
> myFavFlower.toUpperCase();  
"PETUNIAS"  
> myFavColor.toUpperCase();  
"GREEN"
```

As we may expect, all of our strings have been uppercased. So what's so special about this? Well, we can call `toUpperCase` on any (and every) string thanks to prototypes.

A better question to ask is: how does each string know how to uppercase itself? Well, they have to be taught somehow. Just as a baby inherits certain traits when it is born, JavaScript objects inherit methods and properties when they are created. They do so through **prototypal inheritance**, a concept that can be confusing even to more experienced developers. For that reason, it's not necessary to dig into it too deeply right now.

The big advantage of this naming convention for methods is that we can clearly distinguish which object a method belongs to. This allows for multiple methods of the same name to exist in JavaScript. Be careful, though! Even though two methods share the same name `concat`, they are not the same!

For instance, if we look up JavaScript's `concat` method on MDN, we'll actually discover that there are two different `concat` methods:

- `String.prototype.concat()` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/concat)
- `Array.prototype.concat()` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/concat)

When we refer to `String.prototype.concat()`, we are referring to a `concat()` method that all strings inherit. When we say `Array.prototype.concat()`, we are referring to a `concat()` method that all arrays inherit. Even though these methods have the same name, they are actually different methods. Sure, they do essentially the same thing: they put things together. When called on strings, they put two strings together. When called on arrays, they put two arrays together. However, we can only call one on strings and the other on arrays. Fortunately our browser's JavaScript interpreter sorts this all out for us! We just need to do our best to make sure that we're calling the right method on the right object.

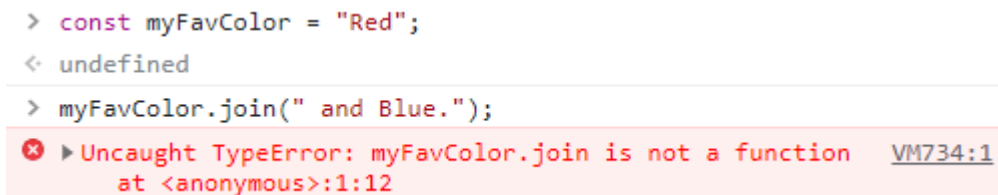
So let's reiterate what we do need to know. We'll see `prototype` in a JS method name when we are looking at documentation, like `String.prototype.concat()`. We'll also continue to use this same syntax when naming JS methods in the curriculum, which will help

us differentiate methods that have the same name. When we see a method like `Array.prototype.concat()` this indicates to us that the `concat` method in question can be applied to any and all arrays.

Note: Web APIs, like `document`, `Element`, and others also use prototypal inheritance, but we don't need to worry about that now. Methods belonging to Web API objects don't follow the naming convention `dataType.prototype.methodName()` where `dataType` is the name of the object and `methodName` is the name of the method. As an example, we **do not** list the `Element.removeAttribute()` (<https://developer.mozilla.org/en-US/docs/Web/API/Element/removeAttribute>) method as `Element.prototype.removeAttribute()`.

A Common Error

Check out the following image that shows a common error message: `Uncaught TypeError: {name of method in code} is not a function`.



```
> const myFavColor = "Red";  
< undefined  
> myFavColor.join(" and Blue.");  
✖ ▶ Uncaught TypeError: myFavColor.join is not a function VM734:1  
   at <anonymous>:1:12
```

If you get a `TypeError` error message similar to the above example, this usually means the object you are calling the method on doesn't have a definition for that method. In other words, you've called a method that doesn't exist for that object type. In the example, I've called the `join` method on a string, but this method actually belongs to arrays: `Array.prototype.join()`.

More on MDN

If you *really* have to look into prototypal inheritance more to get a grasp on all of this, see the Mozilla documentation on Object prototypes (<https://developer.mozilla.org/en->

US/docs/Learn/JavaScript/Objects/Object_prototypes). This is an advanced topic — and at this point, we recommend steering clear of it. Many experienced developers find it confusing, too.

[Previous \(/introduction-to-programming/arrays-and-looping/additional-pair-programming-tips\)](#)

[Next \(/introduction-to-programming/arrays-and-looping/static-versus-instance-with-built-in-js-objects\)](#)

Lesson 3 of 50

Last updated February 28, 2023

[disable dark mode](#)



© 2023 Epicodus (<http://www.epicodus.com/>), Inc.