Lesson Weekend

# Intermediate JavaScript (/intermediate-javascript) / Object-Oriented JavaScript (/intermediate-javascript/object-oriented-javascript) / Constructor and Prototype Methods

Text Cheat sheet

Now that we have an understanding of JavaScript objects, constructors, and prototypes, let's put them to work! In the next few lessons, we'll build an address book app to store contact info for our friends.

**Take note that you do not have to code along with these lessons**, but you can if you want to. The first classwork of this section will prompt you to recreate this project.

Since each contact will have multiple properties, we will use `Contact` objects to encapsulate their data. And because all `Contact`s should have the same combination of properties (name, phone number, etc.) we'll create a constructor that can quickly craft many different `Contact` objects with this same structure.

We'll focus only on the business logic for now. Then we'll build code for the user interface together in upcoming lessons.

## Project Setup

First, let's create a project directory called `address-book`. It will contain a `js` subdirectory to house JavaScript logic, with a single JavaScript file called `scripts.js` inside. Like all projects, we'll also include a `README.md`.

Our project structure should look like this:

```
address-book
├── js
│   └── scripts.js
└── README.md
```

Again, we'll wait to add user interface logic (that is, the HTML and JavaScript that creates the user-facing portion of the app) until after we've written our business logic.

## Adding the `Contact` Constructor

We'll start by building a simple `Contact` constructor in `scripts.js`:

**js/scripts.js**

```
function Contact(firstName, lastName, phoneNumber) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.phoneNumber = phoneNumber;
}
```

Let's test this code in the DevTools console! Open the DevTools console on any webpage, and then copy/paste the constructor above into the console. Then, recreate our instance of `Contact` by running the following:

```
> let testContact = new Contact("Ada", "Lovelace", "808-555
-1111");
```

If we check the value of `testContact` (by typing `testContact;` into the console, and hitting *Enter*), we'll see the following response appear:

```
> testContact;
Contact {firstName: "Ada", lastName: "Lovelace", phoneNumbe
r: "503-555-1111"}
```

This means that we've successfully created a `Contact` object type with the `Contact` constructor function, and we've used it to create one `Contact` instance.

## Adding a method to the `Contact` Prototype

Next, let's create a simple prototype method to call on `Contact` objects. Let's say we want a `Contact.prototype.fullName()` method to return the contact's first and last name concatenated together.

We can define a new prototype method in our `scripts.js` file like this:

**js/scripts.js**

```
function Contact(firstName, lastName, phoneNumber) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.phoneNumber = phoneNumber;
}

Contact.prototype.fullName = function() {
  return this.firstName + " " + this.lastName;
};
```

By the way, if you still have any confusion about semicolons, let's go over this again one more time because the example above illustrates the difference with defining functions. In the first example, we begin with the `function` keyword. Functions don't have semicolons after the closing curly brace.

However, in the second function, we are assigning a function expression to a property of the `Contact.prototype`. It's like assigning a value to a variable. That means we add a semicolon to the end.

It's all a bit arbitrary, really, since JavaScript doesn't really care. As we mentioned in the Introduction to Programming course, the JavaScript interpreter will automatically insert semicolons where needed so our machines can properly read our code. However, the interpreter isn't perfect and there are some fairly obscure situations where JavaScript can get things wrong. Beginners can definitely run into these situations and it's difficult to debug if you don't know what's happening. That's why we are adding semicolons instead of having JavaScript do it automatically — because it's hard for beginners to know those weird situations when JavaScript won't do it automatically for us.

By the way, many companies (one example is Airbnb) require semicolons in their consistency standards while many others don't. While we require semicolons and consistency at Epicodus, you may not be using semicolons at a future job.

We can copy/paste the contents of `scripts.js` into the DevTools console again and create another sample `Contact` by invoking our constructor with the following code:

```
> let testContact = new Contact("Ada", "Lovelace", "503-555
-1111");
```

After that, we can call our new method in the console like this:

```
> testContact.fullName();
"Ada Lovelace"
```

And as we can see, it returns the object's `firstName` and `lastName` properties concatenated together. Now any existing or future `Contact` instances will have access to the `Contact.prototype.fullName()` method.

Now that we've created a constructor for `Contact` and a simple prototype method that can be called on `Contact` instances, let's move on to constructing the address book itself.

---

**Example GitHub Repo for the Address Book (https://github.com/epicodus-lessons/oop-address-book-v2/tree/1_address_book_constructor_and_prototype_methods)**

Previous (/intermediate-javascript/object-oriented-javascript/constructors-and-prototypes)
Next (/intermediate-javascript/object-oriented-javascript/accessing-code-from-different-branches)



disable dark mode

(http://www.epicodus.com)