Lesson Monday

Intermediate JavaScript (/intermediate-javascript) / Object-Oriented JavaScript (/intermediate-javascript/object-oriented-javascript) / Looping Through Objects and Prototypal Inheritance

Text Cheat sheet

In order to display all of the contacts in our application, we're going to need to loop through our address book's contacts. However, since our contacts are all stored in an object, we have to do this differently than we would with looping through an array.

In this lesson, we'll learn exactly how to do that. Then, in the next lesson, we'll apply what we've learned here to actually loop through our contacts.

In the process of learning about looping through object properties, we'll also do a deeper dive into prototypal inheritance — though not too deep!

Looping Through Objects with `Object.keys()`

Let's say we have an object that stores information about the mathematician Ada Lovelace in our address book application. (Don't add any of the following code to the address book application — we are just using this as an example.) We want to take this information and convert it all into a single string which we'll display on our website.

Let's take a look at our object:

```
let mathematician = {
  firstName: "Ada",
  lastName: "Lovelace",
  profession: "Mathematician",
  funFact: "Daughter of Lord Byron",
  countryOfBirth: "England",
  yearOfBirth: 1815,
  yearOfDeath: 1852
}
```

We *could* just display each property individually (such as by doing `mathematician.firstName`) but that becomes less and less feasible the more properties our objects have — and results in more and more work.

So let's turn it into one long string instead. In Address Book: Finding and Deleting Contacts (https://www.learnhowtoprogram.com/intermediate-javascript/object-oriented-javascript/address-book-finding-and-deleting-contacts), we learned about `Object.keys()`, a method that returns an array of all the keys in an object. We can take advantage of this method to grab the keys in an array and then loop over them:

```
> const adaKeys = Object.keys(mathematician);
> let adaString = "";
> adaKeys.forEach(function(key) {
  adaString = adaString.concat(key + ": " + mathematician[k
ey] + "\n");
});
```

This isn't too bad — we create a constant called `adaKeys` that holds an array of the `mathematician` object's keys. Next we initialize an empty string called `adaString`.

Finally, we loop over our array of keys. For each key, we use `String.prototype.concat()` to add a stringified key and value along with a new line after each key-value pair.

If we print the values of `adaString` to the DevTools console, we get the following string:

```
> adaString;
'firstName: Ada\nlastName: Lovelace\nprofession: Mathematic
ian\nfunFact: Daughter of Lord Byron\ncountryOfBirth: Engla
nd\nyearOfBirth: 1815\nyearOfDeath: 1852\n'
```

Something is wrong! I don't see any new lines, only the symbol `\n` that represents a new line. Well, in order to have the new line symbols be evaluated as new lines, we need them to be processed through a `console.log()` in the DevTools or our HTML.

If we log `adaString` to the DevTools console, we get the following string:

```
> console.log(adaString);
firstName: Ada
lastName: Lovelace
profession: Mathematician
funFact: Daughter of Lord Byron
countryOfBirth: England
yearOfBirth: 1815
yearOfDeath: 1852
```

That looks better! And while this output may *look* the same as our object, we now have a string. This could be helpful if we wanted to append many different objects to the DOM, especially if they have different keys. There's no need to specify each key in our code.

You might be thinking that the keys above don't look very pretty — they are formatted like JavaScript variables, not syntactically correct English. Well, our loop could also format the keys, automatically capitalizing the first letter and then using a regular expression to identify capital letters and then add separators so there is a space between each word. Only a little bit of code would be necessary to "prettify" this string so we could easily append entire objects to the DOM. We won't demonstrate how to do this in this lesson — after all, it has nothing to do with looping — but you may want to experiment with this in your own code!

In general, using `Object.keys()` is a very effective way to loop over properties in JavaScript. In fact, it's generally the best way to do so — and it's exactly what we'll do in the next lesson.

Looping through Objects with `for...in`

Now let's take a look at some syntactic sugar that JavaScript provides for looping through objects: the `for...in` loop. **Caution:** There is an important use case where we won't want to use

`for...in`, which we will cover in a moment. Using `Object.keys()` will generally be better! Going over this gotcha will give us further insights into prototypal inheritance.

Here's an example of a `for...in` loop:

```
> let adaString = "";
> for (const key in mathematician) {
  adaString = adaString.concat(key + ": " + mathematician[k
ey] + "\n");
}
> console.log(adaString);
firstName: Ada
lastName: Lovelace
profession: Mathematician
funFact: Daughter of Lord Byron
countryOfBirth: England
yearOfBirth: 1815
yearOfDeath: 1852
```

The only thing we had to change here is the following syntax: `for (const key in mathematician)`. The code inside the curly brackets `{ }` remains the same, and so does the output when we log the value of `adaString`.

As we can see, this is a special kind of `for` loop. We first specify a variable name — here we call it `key` but we could call it `property` or something else. Then, we specify the object we are iterating over, which is `mathematician`. In pseudocode, the syntax of the `for...in` loop looks like so:

```
for (const property in object) {
  // execute code for each property in the object
}
```

This all seems great, right? Well, now for the gotcha — and it's a big one. Let's see what happens if we use `for...in` with a contact created using our `Contact` prototype. First, add this code to your DevTools console:

```
> function Contact(firstName, lastName, phoneNumber) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.phoneNumber = phoneNumber;
}
> Contact.prototype.fullName = function() {
  return this.firstName + " " + this.lastName;
};
> let newContact = new Contact("Ada", "Lovelace", "111-111-
1111");
```

Now try looping over the contact object, looking at each `key`:

```
> for (const key in newContact) {
  console.log(newContact[key]);
}
Ada
Lovelace
111-111-1111
ƒ () {
  return this.firstName + " " + this.lastName;
}
```

And what do we get? Well, the first three values make sense — but why is the following listed as a property?

```
ƒ () {
  return this.firstName + " " + this.lastName;
}
```

This is a method that's listed in our `Contact` object! The `Contact.prototype.fullName()` method to be precise. Now is a good time to review prototypes and prototypal inheritance, along with exploring more about `for...in` loops.

A Review of Prototypes

Let's take a closer look at the `newContact` object in the console:

In the above image, we see that in addition to the three properties we created, `Contact` also has a `__proto__` property. If you are following along now in your DevTools console, note that this property may also be labeled as `[[Prototype]]` instead of `__proto__`. As we learned in a previous lesson, if we want to access an object's prototype, we do so by accessing the `__proto__` property like so:

```
> newContact.__proto__;
fullName: ƒ ()
constructor: ƒ Contact(firstName, lastName, phoneNumber)
[[Prototype]]: Object
```

Both `fullName` and `constructor` are properties of this `__proto__` object. This is how our humble `newContact` object uses prototypal inheritance to get access to the `Contact` constructor and the `Contact.prototype.fullName()` method.

In turn, we can see that `__proto__` *also* has a `__proto__` property, which contains the functionality of basic objects in JavaScript. In a previous lesson, we learned that this chain of `__proto__` objects makes **prototypal inheritance** possible in JavaScript. Let's review it now. Say we call this method on our `newContact` object:

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

When we call `newContact.fullName()`, JavaScript will first look at our `newContact` object to see if that method is attached to it. If it's not, it will look in the `__proto__` property to see if the method is there. If it isn't, it will look at that `__proto__`'s `__proto__` — until it finds the method.

And if it doesn't find the method, it will return `Uncaught TypeError: [functionName] is not a function` — where `[functionName]` is the name of the function it couldn't find.

Well, this is the problem with `for...in` — it doesn't just iterate over properties of an object — it iterates over all the **enumerable** properties of the object as well as enumerable properties in the prototype chain. (Enumerable (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Enumerability_and_ownership_of_properties) just means that the property has an internal enumerable flag set to `true`.)

In the case of our `newContact` object, that means `for...in` also enumerates over the properties of the `Contact` object type — specifically `Contact.prototype.fullName()`, which we do not want!

This is actually a pretty annoying behavior — and it's too bad that `for...in` loops do this.

We can fix the issue by doing the following:

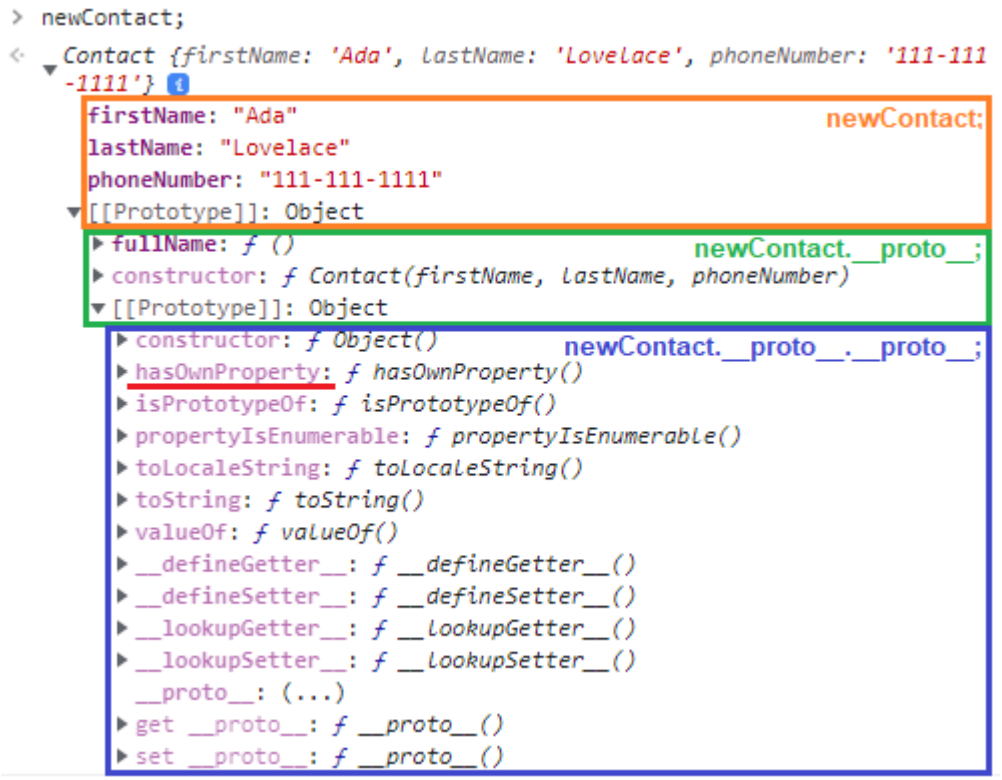
```
> let newContact = new Contact("Ada", "Lovelace", "111-111-
1111");
> for (const key in newContact) {
  if (newContact.hasOwnProperty(key)) {
    console.log(newContact[key]);
  }
}
Ada
Lovelace
111-111-1111
```

The `Object.prototype.hasOwnProperty()` method returns a boolean. If a property belongs directly to an object (as `firstName` belongs to our `newContact` object), it will return `true`. If the property doesn't belong to the object (as is the case of `Contact.prototype.fullName()`, which belongs to the `Contact` object type), it will return `false`.

By the way, notice that the method `Object.prototype.hasOwnProperty()` belongs to the `Object` type (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object) and not `Contact`, or `newContact` for that matter. We can call `Object.prototype.hasOwnProperty()` on `newContact`, because `newContact` has access to this method through prototypal inheritance.

For every object in JavaScript, the `Object` type ends the chain of prototypes. This means that every object in JavaScript inherits functionality from the `Object` type through prototypal inheritance. In order for `newContact` to access `Object.prototype.hasOwnProperty()`, it has to go to its own `__proto__` property, and then to the `__proto__`'s `__proto__` property. The following image shows this, highlighting which properties are returned from which object:

- The orange box highlights the properties belonging to `newContact;`.
- The green box highlights the properties belonging to `Contact`, accessed through `newContact.__proto__;`.
- The blue box highlights the properties belonging to `Object`, accessed through `newContact.__proto__.__proto__;`.
- The red underline highlights the `Object.prototype.hasOwnProperty()` method.



Well, so much for the syntactic sugar of a `for...in` loop. While we can verify that properties actually belong to objects, it's probably just better to iterate using `Object.keys()` instead. In fact, the Mozilla documentation mostly recommends `for...in` loops for debugging. Check out the documentation on `for...in` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in) for more information.

However, this dive into `for...in` hopefully provides a good reminder about how prototypal inheritance works. If you are feeling especially brave, you might even want to read more about Inheritance and the prototype chain (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain). At this point, it's enough to have a very basic understanding of how JavaScript objects inherit from other objects.

In this lesson, we've learned a few ways to iterate over properties in an object. We've also reviewed how prototypal inheritance works, a key and often very confusing topic for developers. In the next lesson, we'll use what we've learned to actually loop through the contacts in our address book application.

Previous (/intermediate-javascript/object-oriented-javascript/address-book-user-interface)
Next (/intermediate-javascript/object-oriented-javascript/address-book-adding-interactivity)

Lesson 18 of 33
Last updated March 23, 2023