Lesson Weekend

Intermediate JavaScript (/intermediate-javascript) / Object-Oriented JavaScript (/intermediate-javascript/object-oriented-javascript) / Address Book: Finding and Deleting Contacts

Text | Cheat sheet

We can now add a `Contact` to our `AddressBook`. However, this feature isn't very helpful unless we can also retrieve a contact later. This feature is necessary for any application that uses a real database — it's just as important to be able to retrieve data from the database as it is to store it. In this lesson, we'll create an `AddressBook.prototype.findContact()` method that allows us to find a `Contact` by its `id` property.

Finding `Contact`s

This method will actually be really easy to write — and very efficient. Here it is:

js/scripts.js

```
...

AddressBook.prototype.findContact = function(id) {
  if (this.contacts[id] !== undefined) {
    return this.contacts[id];
  }
  return false;
};

...
```

- First, our new method takes an `id` as an argument. This will contain the unique ID we assigned to each `Contact` in the last lesson.
- Next, we use a conditional to check if `this.contacts[id] !== undefined`. If we try to find a property in an object that doesn't exist, JavaScript will return `undefined`. We don't want our method to return `undefined`, though. We want it to return a specific `Contact` and if that `Contact` doesn't exist, we'll return `false`. So if this conditional isn't met, that means the contact doesn't exist in the database and the method will return `false`.
- So if `this.contacts[id]` isn't `undefined`, we'll return that specific contact (`this.contacts[id]`). There's no need to iterate through the object. If our `AddressBook` mock database were an array, we'd have to loop through each of the contacts until we find a matching ID — which isn't efficient at all. Finding a value by its key in a JavaScript object means it's super fast to look up a contact — no matter how big the address book gets.

By the way, here's an interesting fact — under the hood, JavaScript stores all the properties of an object in an array. So technically, we are kind of using an array to look things up. However, JavaScript uses a very efficient **hashing algorithm** so that each key in an object corresponds to an index in an array. It does this because it's very fast to find an element in an array based on its index. This is

more advanced stuff which we cover in our Hash Tables (https://www.learnhowtoprogram.com/computer-science/bit-manipulation-and-hashing/hash-tables) computer science curriculum. We don't recommend looking at this lesson yet as it's more advanced, but later in the program, you'll have an opportunity to learn about this.

Testing it Out

Let's test out our new method. Like before, we'll copy/paste the contents of `scripts.js` into the DevTools console.

Next, enter the following in the console to populate our `AddressBook`:

```
> let addressBook = new AddressBook();
> let contact = new Contact("Ada", "Lovelace", "808-555-010
0");
> let contact2 = new Contact("Grace", "Hopper", "503-555-01
99");
> addressBook.addContact(contact);
> addressBook.addContact(contact2);
```

Then we'll confirm our code provided each `Contact` an `id` by inspecting each object's `id` property:

```
> contact.id;
1
> contact2.id;
2
```

Once again, notice our `Contact` object has an `id` property even though the constructor didn't assign this property. We could update our constructor to assign an ID right away but it's not necessary to

do so. Constructors are great for any properties that need to be created on initialization of an object — but it's also okay to add properties to an object later.

Now we can test our new `AddressBook.prototype.findContact()` method by providing it an ID number and confirming it returns the correct info. Let's try getting our second contact's information:

```
> addressBook.findContact(2);
Contact {firstName: "Grace", lastName: "Hopper", phoneNumbe
r: "503-555-0199", id: 2}
```

Woohoo! We've confirmed that `AddressBook.prototype.findContact()` successfully locates a `Contact` using the corresponding `id`.

There's something else to note here, something that's not really ideal about JavaScript objects. Remember how when we identified a contact by their first name, we had to do `this.contacts["Ada"]`, not `this.contacts[Ada]`? We don't have to do that here. You might think that makes sense because the `id` property is a number, but see what happens when we try this out in the console:

```
> addressBook.findContact("2");
Contact {firstName: "Grace", lastName: "Hopper", phoneNumbe
r: "503-555-0199", id: 2}
```

You might think this is because JavaScript is doing loose equality and converting the string into an integer, but actually it's the other way around. Object properties can be strings (or symbols) but not numbers. We can verify that the keys are actually strings by doing the following in the console:

```
> Object.keys(addressBook.contacts)[0];
"1"
> typeof Object.keys(addressBook.contacts)[0];
"string"
```

The `Object.keys()` method (a **static** method called on the object type, and not the instance) returns all the keys in an object. Remember, object keys are object property names. We are using bracket notation to look at the first key in `addressBook.contacts[0]`. You'll see it is `"1"` — a string, and not a number! We can even verify this by using `typeof` to return the type of the key, which shows that it is indeed a string.

So, because JavaScript can do loose equality for us, we can write `addressBook.findContact(2)` instead of `addressBook.findContact("2")`. While this is confusing, this is just how JavaScript objects work. In the next section, we'll learn about a type of object called a Map that cleans up these issues — but for this section, we are going to stay focused on basic objects — in all their glory.

Deleting `Contact`s

Let's add one more prototype method for practice. What if we wanted to delete a `Contact` from our `AddressBook`? Here's a method to do that. We'll add it right below `addressBook.prototype.findContact()`:

js/scripts.js

```
...

AddressBook.prototype.deleteContact = function(id) {
  if (this.contacts[id] === undefined) {
    return false;
  }
  delete this.contacts[id];
  return true;
};

...
```

It's very similar to `AddressBook.prototype.findContact()`. If there's no `Contact` object at the specified `id` (`this.contacts[id] === undefined`), the method will return `false` because no such contact exists and nothing was deleted.

Otherwise, we'll use `delete` to delete the contact based on its `id` property:

```
delete this.contacts[id];
```

This will simply remove the key-value pair from the object. We can test it out by adding this method to our console code and then doing the following:

```
> let addressBook = new AddressBook();
> let contact = new Contact("Ada", "Lovelace", "503-555-010
0");
> let contact2 = new Contact("Grace", "Hopper", "503-555-01
99");
> addressBook.addContact(contact);
> addressBook.addContact(contact2);
> addressBook;
  AddressBook {contacts: {…}, currentId: 2}
  contacts: {1: Contact, 2: Contact}
  currentId: 2
  [[Prototype]]: Object
> addressBook.deleteContact(1);
> addressBook.contacts;
  AddressBook {contacts: {…}, currentId: 2}
  contacts: {2: Contact}
  currentId: 2
  [[Prototype]]: Object
```

We'll see that that `addressBook.contacts` now has only one contact. The `currentId` is still `2`, though, and `1` will *never* be used as an ID again even though the contact with that ID has been deleted. Once again, that's how a real database works.

After following along with this lesson, our entire `scripts.js` file will look like this:

js/scripts.js

```
// Business Logic for AddressBook ---------
function AddressBook() {
  this.contacts = {};
  this.currentId = 0;
}

AddressBook.prototype.addContact = function(contact) {
  contact.id = this.assignId();
  this.contacts[contact.id] = contact;
};

AddressBook.prototype.assignId = function() {
  this.currentId += 1;
  return this.currentId;
};

AddressBook.prototype.findContact = function(id) {
  if (this.contacts[id] !== undefined) {
    return this.contacts[id];
  }
  return false;
};

AddressBook.prototype.deleteContact = function(id) {
  if (this.contacts[id] === undefined) {
    return false;
  }
  delete this.contacts[id];
  return true;
};

// Business Logic for Contacts ---------
function Contact(firstName, lastName, phoneNumber) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.phoneNumber = phoneNumber;
}

Contact.prototype.fullName = function() {
```

```
  return this.firstName + " " + this.lastName;
};
```

You can also experiment with adding other methods as well. For instance, try adding an `Contact.prototype.update()` method on the `Contact` prototype. The main focus in this section is constructors and prototype methods so we encourage you to focus on business logic and increasing your understanding of these key concepts. When working with business logic, make sure to test it in the console before incorporating it into your UI logic.

Example GitHub Repo for the Address Book (https://github.com/epicodus-lessons/oop-address-book-v2/tree/4_finding_and_deleting_contacts)

Previous (/intermediate-javascript/object-oriented-javascript/address-book-unique-ids)
Next (/intermediate-javascript/object-oriented-javascript/vs-code-bracket-colorization-and-guides)

Lesson 12 of 33
Last updated March 23, 2023