Lesson Weekend

Intermediate JavaScript (/intermediate-javascript) / Object-Oriented JavaScript (/intermediate-javascript/object-oriented-javascript) / Address Book: Objects Within Objects

Text Cheat sheet

In a real world application, we'd save our address book's `Contact`s in a database. However, we aren't working with databases yet. Instead, we'll create a mock database (a fake database) and store its data inside a global variable.

As we discussed in Variable Scope (https://www.learnhowtoprogram.com/introduction-to-programming/javascript-and-web-browsers/variable-scope), we want to avoid global variables wherever possible. So why are we going to use one here?

Well, one of the biggest problems with global variables is that they never fall out of scope — and their values persist throughout an application. Generally, this is a recipe for bugs. However, we *want* the values in a database to persist and be available all throughout an application. What is the point of a database if we can't retrieve data from it? That's why we're using a global variable here — to better imitate what a database actually does.

Take note that a mock database wouldn't actually be useful in the real world so we wouldn't use a global variable like this in a real world application, either. At this point, you might wonder why we don't just jump into using databases then. Well, they're pretty complicated! For now, we will stay focused on core JavaScript concepts. It will still be a while before we start working with actual databases.

Also, just because we are using a global variable to mock a database doesn't mean you should start adding global variables throughout your code. For the next several sections, here is a guideline: if your variable is meant to represent a potential database, a global variable is fine. Otherwise, avoid them if possible. Most of the projects we do throughout Intermediate JavaScript will *not* need to mock a database, so think very carefully about whether or not you need to add this functionality as you build your projects.

`AddressBook` Constructor

Much like our `Contact`s, our `AddressBook` will be a JavaScript object. But instead of containing properties like `firstName` or `lastName`, it will contain a list of `Contact` objects, similar to how the previous lesson depicted objects being saved within other objects.

To do this we'll need an `AddressBook` constructor. Let's add the following new constructor to the top of `scripts.js`:

js/scripts.js

```
function AddressBook() {
  this.contacts = {};
}

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

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

`AddressBook` objects contain a single property: An empty object called `contacts`. This is where we'll store entries in our address book. Each entry will be a `Contact` object. As we can see, we'll be storing objects within an object — all of the `Contact` objects will be stored in the `contacts` property, an object within the `AddressBook` object.

If we wanted to, we could build out our application to have many instances of `AddressBook`s, each with their own `Contact`s. We could also include an `owner` property that gives information about the owner of the `AddressBook`. Or, we could add a `lastModified` timestamp that tells us when the `AddressBook` was last modified. However, we will keep this simple with just one `contacts` property and one instance of `AddressBook`.

We'll also add comments showing where `AddressBook` and `Contact` logic will go in `scripts.js`. This will make it easier to follow along with the lessons.

js/scripts.js

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

// 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;
};
```

Adding a Method to the `AddressBook` Prototype

`AddressBook` s can only do one thing right now: store a list of contacts in key-value pairs. Let's define a few prototypes for our `AddressBook` objects to give them more functionality.

Adding `Contact` s to the `AddressBook`

We'll create a prototype method to add new `Contact` s to an `AddressBook` . This will go right below the `AddressBook` constructor:

js/scripts.js

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

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

// 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;
};
```

- Our new `AddressBook.prototype.addContact()` method takes a `Contact` object as an argument. We can tell because the parameter is named `contact`, which indicates that the method expects a `Contact` object.
- `this.contacts` is the address book property where we're storing all of our `Contact` objects. `this` represents the instance of the address book, so when we write `this.contacts`, it means we're accessing the `contacts` property of the address book instance.
- With `this.contacts[contact.firstName] = contact;`, we are creating a new key in the address book's `contacts` property, *and* assigning it a value:
    - The key `contact.firstName` will be set to the contact's first name. Here we need to use bracket notation to create the key, because `contact.firstName` is a variable.
    - The value we assign to the new key with `= contact;` is the `Contact` object that we pass into the method.

- Generally, a contact in a real database will have a unique ID to locate it. Soon, we'll refactor our code to do this. For now, we're using the `Contact` object's `firstName` property as an ID.

That's all it takes for us to add a new `Contact` object to our `AddressBook`!

Let's try it out. We can copy/paste the contents of `scripts.js` into the DevTools console, and enter each of the following five lines:

```
> 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);
```

Let's walk through what each of these lines is doing:

1. We create an `AddressBook` object.
2. We create a new `Contact` object with a `firstName` of `"Ada"`, saved to the variable name `contact`.
3. We create another new `Contact` object, this time with a `firstName` of `"Grace"`, saved to the variable name `contact2`.
4. We add the first `Contact` object to our `AddressBook`, using our new `AddressBook.prototype.addContact()` method.
5. We add the second `Contact` object to the `AddressBook` using the same new method.

Viewing `Contact`s in the `AddressBook`

If we then run the following in the console, we can see the contents of our `AddressBook`:

```
> addressBook;
AddressBook {contacts: {…}}
```

We can see that the `addressBook` is an object that contains another object called `contacts`. To access these contacts, we can do the following:

```
> addressBook.contacts;
{Ada: Contact, Grace: Contact}
```

Both of our contacts are there! But how do we access them? Well, each object has a key. The first one has a key of `Ada` while the second has a key of `Grace`. So we can access them like this:

```
> addressBook.contacts["Ada"];
Contact {firstName: "Ada", lastName: "Lovelace", phoneNumbe
r: "503-555-0100"}
```

We can do the same for `addressBook.contacts["Grace"]`.

Note that we **cannot** do the following:

```
> addressBook.contacts[Ada];
```

We'll get the following error if we do:

```
Uncaught ReferenceError: Ada is not defined
```

This is because JavaScript is reading this as a variable, not a string — and we haven't defined an `Ada` variable. Instead, if the key is a string, we need to write it as a string.

If we wanted to get even more specific information about Ada — for instance, her phone number — we can do so like this:

```
> addressBook.contacts["Ada"].phoneNumber;
"503-555-0100"
```

We just need to identify the property we want the value of — in this case, it's the `phoneNumber` property. Sometimes objects can be very deeply nested. No matter how deeply nested an object or property is, we can keep drilling down further until we retrieve it. We will cover this further in a future lesson.

In the next lesson, we'll add a property to help us assign IDs to each contact.

Example GitHub Repo for the Address Book (https://github.com/epicodus-lessons/oop-address-book-v2/tree/2_objects_within_objects)

Previous (/intermediate-javascript/object-oriented-javascript/objects-within-objects)
Next (/intermediate-javascript/object-oriented-javascript/address-book-unique-ids)

Lesson 10 of 33
Last updated March 23, 2023

disable dark mode

(http://www.epicodus.com)