

Lesson

Weekend

# Intermediate JavaScript (/intermediate-javascript)

## / Object-Oriented JavaScript (/intermediate-javascript/object-oriented-javascript)

### / Address Book: Unique IDs

Text

Cheat sheet

As mentioned in the last lesson, a real world application would use a database. In a database, each individual `Contact` would have a unique ID. This allows us to identify records by a unique ID instead of something like their name, which isn't guaranteed to be unique. Right now, we are identifying each of our `Contact`s by their `firstName` property. However, there are multiple problems with that approach. An ID needs to be unique — but what if you have two `Contact`s that each have the same `firstName` property? In our case, it would be better if each `Contact` has a numerical ID — and we'll want to ensure that each ID is different — just like in a real database.

## Unique Object IDs

Let's update our code so each `Contact` is assigned an ID as soon as it's created.

We'll update several functions to make this work. First, let's update the `AddressBook` constructor to instantiate new `AddressBook`s with a `currentId` property:

**js/scripts.js**

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

Now each time a new `AddressBook` is created, it will have a `currentId` property that begins at `0`. What is the point of doing this? Well, our IDs are going to increase sequentially. But how does our `AddressBook` know the lowest available ID that hasn't been used yet? It would be terribly inefficient to have to look through *all* the stored `contacts` to see which IDs aren't being used — and it wouldn't even be effective because it wouldn't account for the IDs of deleted `contacts`. In a real-world database, an ID is *never* reused — even if the `contact` or other entry it's referring to is deleted.

So we'll use `this.currentId` to track the next available ID number that we can issue to a new `contact`. This means that our `AddressBook` also needs to handle assigning IDs and incrementing `this.currentId` by 1. We'll do this by defining another prototype method called `AddressBook.prototype.assignId()`:

**js/scripts.js**

```
// Business Logic for AddressBook -----  
  
...  
  
AddressBook.prototype.assignId = function() {  
    this.currentId += 1;  
    return this.currentId;  
};  
  
...
```

This new method will increment the `this.currentId` property on the `AddressBook` object by 1 and return the updated value. This mimics a database by creating sequentially incrementing ID values which are never repeated so they are always unique.

Finally, we need to call this new `AddressBook.prototype.assignId()` method whenever we add a new `Contact` to the `AddressBook`. We already have a method called `AddressBook.prototype.addContact()` that adds contacts to our mock database. We just need to update it to use our new sequential IDs as keys instead of using the `firstName` property.

### js/scripts.js

```
// Business Logic for AddressBook -----  
  
...  
  
AddressBook.prototype.addContact = function(contact) {  
    contact.id = this.assignId();  
    this.contacts[contact.id] = contact;  
};  
  
...
```

Let's take a closer look at the updates here.

First take note that the keyword `this` refers to the `AddressBook` instance that we are calling the

`AddressBook.prototype.addContact()` method on. Anytime we are in a prototype method and we refer to `this`, we are referencing the object instance that the method is called on. The same is true for constructor functions. Anytime we are in a constructor function and we reference `this`, we are referring to the object instance that the constructor function creates.

When we call `AddressBook.prototype.addContact(contact)`, we are passing in a `Contact` object that has `firstName`, `lastName`, and `phoneNumber` properties. It doesn't have an `id` property yet.

With `contact.id = this.assignId();`, we create a new `id` property on the `Contact` object that we are passing into `AddressBook.prototype.addContact()`, and we set its value to the result of calling the `AddressBook.prototype.assignId()` method. The `AddressBook.prototype.assignId()` method increments `this.currentId` by 1 and returns a number representing the current ID, and this number is what is assigned as the value of the newly created `id` property for the `Contact` object.

Technically, we don't need a separate method for assigning IDs — but it's always a good idea to keep our code separate. The `AddressBook.prototype.assignId()` has only one job — figure out the next sequential ID and return it. It doesn't need to know anything about contacts. If we later decided to use a different system for assigning IDs, we'd only need to change the code in one place. This principle to follow when writing code is called **separation of concerns**.

Another important thing to emphasize is that we can add new properties to an object whenever we want. It doesn't have to happen in an object's constructor. For instance, there's no mention of an `id` property in the `Contact` constructor. We could *technically* do something like this in the `Contact` constructor — `this.id =`

undefined. It's not very clean, though, and it's completely unnecessary to do so. Instead, we can easily add this new property later — and that's exactly what we do in the `AddressBook.prototype.addContact()` function.

Finally, we're also using a `Contact`'s new `id` property as a key when we add the `Contact` to `AddressBook`:

```
this.contacts[contact.id] = contact;
```

In this line of code, `this` refers to the `AddressBook` we've created. Our instance of `AddressBook` has a `contacts` property which itself contains an object that stores the key-value pairs of all our `Contact`s. In the line of code above, we are creating a key in the `contacts` object which corresponds to the new `Contact`'s ID. The value associated with the key is the `Contact` object itself.

This is a big improvement over using the `firstName` property as a key.

So to recap, we now assign a sequential ID to every new `Contact` object. Once an ID is assigned to the `Contact` object, the `Contact` is added to the address book's `contacts` property, which is our mock database. `Contact` objects are stored via key-value pairs in the mock database, where the key is equal to the `Contact`'s ID and the value is the `Contact` object itself.


The updated `scripts.js` file should now 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;  
};  
  
// 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;  
};
```

We can now create a `Contact` and add it to our `AddressBook` mock database! When we do so, it will have a unique ID. It doesn't have the complexity or efficiency of a real database, but soon we'll be able to use our `AddressBook` to retrieve specific contacts just as we would with an actual database.

---

 **Example GitHub Repo for the Address Book**  
([https://github.com/epicodus-lessons/oop-address-book-v2/tree/3\\_unique\\_ids](https://github.com/epicodus-lessons/oop-address-book-v2/tree/3_unique_ids))

[Previous \(/intermediate-javascript/object-oriented-javascript/address-book-objects-within-objects\)](#)

[Next \(/intermediate-javascript/object-oriented-javascript/address-book-finding-and-deleting-contacts\)](#)

Lesson 11 of 33

Last updated March 23, 2023

[disable dark mode](#)



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