Monday: Address Book with Constructors

Now that we have a basic understanding of JavaScript objects, let's put them to work! In the next few lessons we'll build an address book application to store contact information for our friends and acquaintances. Once complete, our application will look and function something like **this (http://mgoren.github.io/address-book/address-book.html)** .

Since each contact will have multiple properties, we will use `Contact` JavaScript objects to encapsulate data. And since each contact will have the same properties (such as name, phone number, etc.) we can create a single `Contact` constructor to make many unique contacts with the same properties. This means we will have a single function that defines every property for a contact object.

Each entry in our address book (or, `Contact` object) should have a first name and last name. As such, we'll begin creating our `Contact` constructor with `firstName` and `lastName` properties, and include additional properties later on. We'll also create a prototype that retrieves a contact object's full name.

Contact Constructor

Before we begin, let's experiment in the JavaScript console:

Here's a basic `Contact` constructor:

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

Let's copy and paste this into the JavaScript console so that we can explore how this works before beginning our project:

```
> function Contact(first, last) { this.firstName = first; this.lastName = last; }
> var ada = new Contact("Ada", "Lovelace");
> ada.firstName
  "Ada"
> ada.lastName
  "Lovelace"
```

Whenever a new `Contact` object is created, it is initialized with whatever first and last names we pass into the constructor as arguments. The line `var ada = new Contact("Ada", "Lovelace");` creates a new `Contact` object by calling the `Contact` constructor and passing it the strings "Ada" and

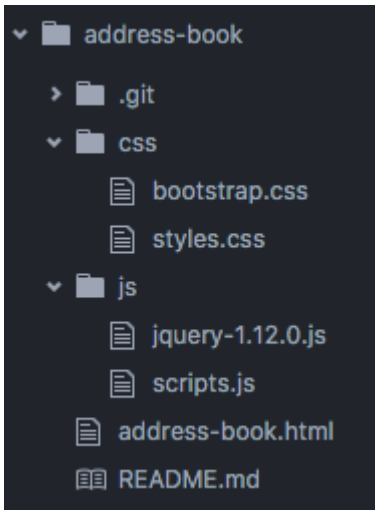
"Lovelace". The `ada.firstName` property becomes "Ada" and the `ada.lastName` property becomes "Lovelace".

The JavaScript console can also display a summary of the object, which is useful for debugging:

```
> ada
  Contact {firstName: "Ada", lastName: "Lovelace"}
```

Here, we can see that the variable `ada` holds an object that is an *instance* of the `Contact` type and that it has two properties - `firstName` with the value "Ada" and `lastName` with the value "Lovelace".

Now that we've experimented with our `Contact` constructor, let's begin building our address book. First, we'll need to set up our JavaScript project directory. The image below details all files that will be used in our application:



Note: Because we will continue to build on this same address book project over the next few lessons, example repositories will be linked throughout for reference. These are simply a snapshot of the project at the given point in development; you're not required to do anything with these repositories.

Example GitHub Address Book Repo with Initial Files (https://github.com/cngondo/address-book)

Now, let's place our `Contact` constructor into *scripts.js* :

js/scripts.js

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

User Interface Logic

Now that our JavaScript `Contact` constructor is in place, let's incorporate it into a web page through the user interface. First, we'll create a basic form that allows users to enter a contact's first and last names. Each new contact created through this form will be added to an ongoing list of *all* contacts.

We'll begin by adding this form, and a place to append newly-created contacts to:

address-book.html

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/bootstrap.css" rel="stylesheet" type="text/css">
    <link href="css/styles.css" rel="stylesheet" type="text/css">
    <script src="js/jquery-1.12.0.js"></script>
    <script src="js/scripts.js"></script>
    <title>Address book</title>
  </head>
  <body>
    <div class="container">
      <h1>Address book</h1>
       <div class="row">
        <div class="col-md-6">
          <h2>Add a contact:</h2>
          <form id="new-contact">
            <div class="form-group">
              <label for="new-first-name">First name</label>
              <input type="text" class="form-control" id="new-first-name">
            </div>
            <div class="form-group">
              <label for="new-last-name">Last name</label>
              <input type="text" class="form-control" id="new-last-name">
            </div>

            <button type="submit" class="btn">Add</button>
          </form>

          <h2>Contacts:</h2>
          <ul id="contacts">

          </ul>
        </div>
      </div>
    </div>
  </body>
</html>
```

Here, the empty `<ul>` with the `id` of `"contacts"` is where each new contact will be appended.

Next, we need some JavaScript in our jQuery callback that will collect the user input from the form, and assign it to variables. We'll use the input to create new `Contact` objects with our constructor. Then, we'll append the new objects to our list for display to the user.

js/scripts.js

```
// business logic
function Contact(first, last) {
  this.firstName = first;
  this.lastName = last;
}

// user interface logic
$(document).ready(function() {
  $("form#new-contact").submit(function(event) {
    event.preventDefault();

    var inputtedFirstName = $("input#new-first-name").val();
    var inputtedLastName = $("input#new-last-name").val();

    var newContact = new Contact(inputtedFirstName, inputtedLastName);

    $("ul#contacts").append("<li><span class='contact'>" + newContact.firstName + "</span></li>"
);

    $("input#new-first-name").val("");
    $("input#new-last-name").val("");
  });
});
```

Now, if we fire up our web page, each time we add a contact it is appended to the contact list. The line `var newContact = new Contact(inputtedFirstName, inputtedLastName)` creates a new `Contact` object by calling the `Contact` constructor and passing it the inputted data. The variable `newContact` now refers to that newly created contact object, which knows its own `firstName` and `lastName`.

What we've developed so far *could* be accomplished without objects and just jQuery. Let's add another feature to our page that better illustrates how objects make a difference in managing and displaying data. When a user clicks on a contact in the list, they will see the contact's first and last names in a `div` to the right of our form.

First, we'll update our HTML to add the `<div>` where the contact details will be shown. (This will be a second column, so it should go inside the *row* div but after the closing of the existing *col-md-6* div.):

address-book.html

```
<div class="col-md-6">
  <div id="show-contact">
    <h2></h2>

    <p>First name: <span class="first-name"></span></p>
    <p>Last name: <span class="last-name"></span></p>
  </div>
</div>
```

We'll hide the "show" area at first, and add a class to make some elements look clickable:

css/styles.css

```
#show-contact {
  display: none;
}

.contact {
  cursor: pointer;
  color: #0088cc;
}

.contact:hover {
  text-decoration: underline;
}
```

Inside our form submit callback, after the code that appends the new contact to the list, we'll add this JavaScript to show the contact information when it is clicked:

js/scripts.js

```
$(".contact").last().click(function() {
  $("#show-contact").show();
  $("#show-contact h2").text(newContact.firstName);
  $(".first-name").text(newContact.firstName);
  $(".last-name").text(newContact.lastName);
});
```

If we didn't add `last()` to `$(".contact")`, each time a new contact was added, every element with the `Contact` class would show the information of the most recently added contact on click. By adding `last()`, we only bind the event to most recently-inserted contact.

Can you imagine trying to keep track of everything on this page using just jQuery? Whew!

Business Logic - Prototype Method

Instead of listing only the first name in our list of contacts, let's create a `fullName` method on the `Contact` prototype so we can easily see the full name of each person in our address book. We don't need to add a new `fullName` property, because each contact *already* includes a `firstName` and `lastName` property. We just need to add a prototype method that returns the full name based on those two properties.

We'll add the following prototype method to the business logic section of our scripts:

js/scripts.js

```
// business logic
function Contact(first, last) {
  this.firstName = first;
```

```
  this.lastName = last;
}

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

As you can see, this method is meant to be called upon a `Contact` object, and simply returns the `Contact`'s `firstName` and `lastName` properties concatenated together.

Remember, a prototype method is a method meant to be called on a specific type of object. For instance, the `Contact` objects in our address book. `fullName()` is a prototype method because it's specifically meant to return the full name of a `Contact` object. Therefore, we define it in a special way that denotes the type of object this method is meant for when we state `Contact.prototype.fullName = function()`.

User Interface Logic to Use Prototype Method

Let's also update our user interface logic to use our new `fullName()` method.

This line of code:

js/scripts.js

```
...
  $("ul#contacts").append("<li><span class='contact'>" + newContact.firstName + "</span></li>");
...
```

Should be changed to reflect the following:

js/scripts.js

```
...
  $("ul#contacts").append("<li><span class='contact'>" + newContact.fullName() + "</span></li
>");
...
```

This will append the contact's full name, as returned from our new `fullName()` prototype to our list of contacts, instead of just their first name. Perfect! In upcoming lessons we'll continue to build our address book application by adding address properties to `Contact` objects.