Learn web development

JavaScript object basics

Overview: Objects

Next

In the first article looking at JavaScript objects, we'll look at fundamental JavaScript object syntax, and revisit some JavaScript features we've already looked at earlier on in the course, reiterating the fact that many of the features you've already dealt with are in fact objects.

Prerequisites:	Basic computer literacy, a basic understanding of HTML and CSS, familiarity with JavaScript basics (see First steps and Building blocks).
Objective:	To understand the basic theory behind object-oriented programming, how this relates to JavaScript ("most things are objects"), and how to start working with JavaScript objects.

Object basics

An object is a collection of related data and/or functionality (which usually consists of several variables and functions — which are called properties and methods when they are inside objects.) Let's work through an example to show us what they look like.

To begin with, make a local copy of our oojs.html file. This contains very little — a `<script>` element for us to write our source code into, an `<input>` element for us to enter sample instructions into when the page is rendered, a few variable definitions, and a function that outputs any code entered into the input into a `<p>` element. We'll use this as a basis for exploring basic object syntax.

As with many things in JavaScript, creating an object often begins with defining and initializing a variable. Try entering the following below the JavaScript code that's already in your file, then saving and refreshing:

```
var person = {};
```

If you enter `person` into your text input and press the button, you should get the following result:

```
[object Object]
```

Congratulations, you've just created your first object. Job done! But this an empty object, so we can't really do much with it. Let's update our object to look like this:

```
var person = {
  name: ['Bob', 'Smith'],
  age: 32,
  gender: 'male',
  interests: ['music', 'skiing'],
  bio: function() {
    alert(this.name[0] + ' ' + this.name[1] + ' is ' + this.age + ' years old
  },
  greeting: function() {
    alert('Hi! I\'m ' + this.name[0] + '.');
  }
};
```

After saving and refreshing, try entering some of the following into your text input:

```
person.name[0]
person.age
person.interests[1]
person.bio()
person.greeting()
```

You have now got some data and functionality inside your object, are now able to access them with some nice simple syntax!

Note: If you are having trouble getting this to work, try comparing your code against our version — see oojs-finished.html (also see it running live). One common mistake when you are starting out with objects is to put a comma on the end of the last member — this will cause an error.

So what is going on here? Well, an object is made up of multiple members, each of which have a name (e.g. `name` and `age` above), and a value (e.g. `['Bob', 'Smith']` and 32). Each name/value pair must be

separated by a comma, and the name and value in each case are separated by a colon. The syntax always follows this pattern:

```
var objectName = {
  member1Name: member1Value,
  member2Name: member2Value,
  member3Name: member3Value
}
```

The value of an object member can be pretty much anything — in our person object we've got a string, a number, two arrays, and two functions. The first four items are data items, and are referred to as the object's **properties**. The last two items are functions that allow the object to do something with that data, and are referred to as the object's **methods**.

An object like this is referred to as an **object literal** — we've literally written out the object contents as we've come to create it. This is in contrast to objects instantiated from classes, which we'll look at later on.

It is very common to create an object using an object literal when you want to transfer a series of structured, related data items in some manner, for example sending a request to the server to be put into a database. Sending a single object is much more efficient than sending several items individually, and it is easier to work with than an array, when you want to identify individual items by name.

Dot notation

Above, you accessed the object's properties and methods using **dot notation**. The object name (person) acts as the **namespace** — it must be entered first to access anything **encapsulated** inside the object. Next you write a dot, then the item you want to access — this can be the name of a simple property, an item of an array property, or a call to one of the object's methods, for example:

```
person.age
person.interests[1]
person.bio()
```

Sub-namespaces

It is even possible to make the value of an object member another object. For example, try changing the name member from

```
name: ['Bob', 'Smith'],
```

to

```
name : {
  first: 'Bob',
  last: 'Smith'
},
```

Here we are effectively creating a **sub-namespace**. This sounds complex, but really it's not — to access these items you just need to chain the extra step onto the end with another dot. Try these:

```
person.name.first
person.name.last
```

Important: At this point you'll also need to go through your method code and change any instances of

```
name[0]
name[1]
```

to

```
name.first
name.last
```

Otherwise your methods will no longer work.

Bracket notation

There is another way to access object properties — using bracket notation. Instead of using these:

```
person.age
person.name.first
```

You can use

```
person['age']
person['name']['first']
```

This looks very similar to how you access the items in an array, and it is basically the same thing — instead of using an index number to select an item, you are using the name associated with each member's value. It is no wonder that objects are sometimes called **associative arrays** — they map strings to values in the same way that arrays map numbers to values.

Setting object members

So far we've only looked at retrieving (or **getting**) object members — you can also **set** (update) the value of object members by simply declaring the member you want to set (using dot or bracket notation), like this:

```
person.age = 45;
person['name']['last'] = 'Cratchit';
```

Try entering these lines, and then getting the members again to see how they've changed:

```
person.age
person['name']['last']
```

Setting members doesn't just stop at updating the values of existing properties and methods; you can also create completely new members. Try these:

```
person['eyes'] = 'hazel';
person.farewell = function() { alert("Bye everybody!"); }
```

You can now test out your new members:

```
person['eyes']
person.farewell()
```

One useful aspect of bracket notation is that it can be used to set not only member values dynamically, but member names too. Let's say we wanted users to be able to store custom value types in their people data, by typing the member name and value into two text inputs? We could get those values like this:

```
var myDataName = nameInput.value;
var myDataValue = nameValue.value;
```

we could then add this new member name and value to the `person` object like this:

```
person[myDataName] = myDataValue;
```

To test this, try adding the following lines into your code, just below the closing curly brace of the `person` object:

```
var myDataName = 'height';
var myDataValue = '1.75m';
person[myDataName] = myDataValue;
```

Now try saving and refreshing, and entering the following into your text input:

```
person.height
```

Adding a property to an object using the method above isn't possible with dot notation, which can only accept a literal member name, not a variable value pointing to a name.

What is "this"?

You may have noticed something slightly strange in our methods. Look at this one for example:

```
greeting: function() {
  alert('Hi! I\'m ' + this.name.first + '.');
}
```

You are probably wondering what "this" is. The `this` keyword refers to the current object the code is being written inside — so in this case `this` is equivalent to `person`. So why not just write `person` instead? As you'll see in the Object-oriented JavaScript for beginners article when we start creating constructors, etc., `this` is very useful — it will always ensure that the correct values are used when a member's context changes (e.g. two different `person` object instances may have different names, but will want to use their own name when saying their greeting).

Let's illustrate what we mean with a simplified pair of person objects::

```
var person1 = {
  name: 'Chris',
  greeting: function() {
```

```
      alert('Hi! I\'m ' + this.name + '.');
    }
  }

  var person2 = {
    name: 'Brian',
    greeting: function() {
      alert('Hi! I\'m ' + this.name + '.');
    }
  }
```

In this case, `person1.greeting()` will output "Hi! I'm Chris."; `person2.greeting()` on the other hand will output "Hi! I'm Brian.", even though the method's code is exactly the same in each case. As we said earlier, `this` is equal to the object the code is inside — this isn't hugely useful when you are writing out object literals by hand, but it really comes into its own when you are dynamically generating objects (for example using constructors). It will all become clearer later on.

You've been using objects all along

As you've been going through these examples, you have probably been thinking that the dot notation you've been using is very familiar. That's because you've been using it throughout the course! Every time we've been working through an example that uses a built-in browser API or JavaScript object, we've been using objects, because such features are built using exactly the same kind of object structures that we've been looking at here, albeit more complex ones than our own custom examples.

So when you used string methods like:

```
1 myString.split(',');
```

You were using a method available on an instance of the `String` class. Every time you create a string in your code, that string is automatically created as an instance of `String`, and therefore has several common methods/properties available on it.

When you accessed the document object model using lines like this:

```
1 var myDiv = document.createElement('div');
2 var myVideo = document.querySelector('video');
```

You were using methods available on an instance of the `Document` class. For each webpage loaded, an instance of `Document` is created, called `document`, which represents the entire page's structure, content, and other features such as its URL. Again, this means that it has several common

methods/properties available on it.

The same is true of pretty much any other built-in object/API you've been using — `Array`, `Math`, etc.

Note that built in Objects/APIs don't always create object instances automatically. As an example, the Notifications API — which allows modern browsers to fire system notifications — requires you to instantiate a new object instance using the constructor for each notification you want to fire. Try entering the following into your JavaScript console:

```
var myNotification = new Notification('Hello!');
```

Again, we'll look at constructors in a later article.

Note: It is useful to think about the way objects communicate as **message passing** — when an object needs other to perform some kind of action often it will send a message to another object via one of its methods, and wait for a response, which we know as a return value.

Summary

Congratulations, you've reached the end of our first JS objects article — you should now have a good idea of how to work with objects in JavaScript — including creating your own simple objects. You should also appreciate that objects are very useful as structures for storing related data and functionality — if you tried to keep track of all the properties and methods in our `person` object as separate variables and functions, it would be inefficient and frustrating, and we'd run the risk of clashing with other variables and functions that have the same names. Objects let us keep the information safely locked away in their own package, out of harm's way.

In the next article we'll start to look at object-oriented programming (OOP) theory, and how such techniques can be used in JavaScript.

↑ Overview: Objects Next →