JavaScript Succinctly

by Cody Lindley

Syncfusion | Technology Resource Portal

JavaScript Succinctly

By
Cody Lindley

Foreword by Daniel Jebaraj

Syncfusion®
Deliver innovation with ease®

Chapter 1 JavaScript Objects

Creating objects

In JavaScript, objects are king: Almost everything is an object or acts like an object. Understand objects and you will understand JavaScript. So let's examine the creation of objects in JavaScript.

An object is just a container for a collection of named values (aka properties). Before we look at any JavaScript code, let's first reason this out. Take myself, for example. Using plain language, we can express in a table, a "cody":

cody	
property	**property value**
living	True
age	33
gender	Male

The word "cody" in the table is just a label for the group of property names and corresponding values that make up exactly what a cody is. As you can see from the table, I am living, 33, and a male.

JavaScript, however, does not speak in tables. It speaks in objects, which are similar to the parts contained in the "cody" table. Translating the cody table into an actual JavaScript object would look like this:

Sample: sample01.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Create the cody object
    var cody = new Object();

    // then fill the cody object with properties (using dot notation).
    cody.living = true;
    cody.age = 33;
    cody.gender = 'male';

    console.log(cody); // Logs Object {living = true, age = 33, gender =
'male'}

</script></body></html>
```

Keep this at the forefront of your mind: objects are really just containers for properties, each of which has a name and a value. This notion of a container of properties with named values (i.e. an object) is used by JavaScript as the building blocks for expressing values in JavaScript. The

cody object is a value which I expressed as a JavaScript object by creating an object, giving the object a name, and then giving the object properties.

Up to this point, the ***cody*** object we are discussing has only static information. Since we are dealing with a programing language, we want to program our ***cody*** object to actually do something. Otherwise, all we really have is a database akin to JSON. In order to bring the ***cody*** object to life, I need to add a property *method*. Property methods perform a function. To be precise, in JavaScript, methods are properties that contain a **`Function()`** object, whose intent is to operate on the object the function is contained within.

If I were to update the ***cody*** table with a ***`getGender`*** method, in plain English it would look like this:

cody object	
property	**property value**
living	True
age	33
gender	Male
getGender	return the value of gender

Using JavaScript, the ***`getGender`*** method from the updated ***cody*** table would look like so:

Sample: sample02.html

```
<!DOCTYPE html><html lang="en"><body><script>

   var cody = new Object();
   cody.living = true;
   cody.age = 33;
   cody.gender = 'male';
   cody.getGender = function () { return cody.gender; };

   console.log(cody.getGender()); // Logs 'male'.

</script></body></html>
```

The ***`getGender`*** method, a property of the ***cody*** object, is used to return one of ***cody***'s other property values: the value "male" stored in the ***`gender`*** property. What you must realize is that without methods, our object would not do much except store static properties.

The ***cody*** object we have discussed thus far is what is known as an **`Object()`** object. We created the ***cody*** object using a blank object that was provided to us by invoking the **`Object()`** constructor function. Think of constructor functions as a template or cookie cutter for producing predefined objects. In the case of the ***cody*** object, I used the **`Object()`** constructor function to produce an empty object which I named ***cody***. Because ***cody*** is an object constructed from the **`Object()`** constructor, we call ***cody*** an **`Object()`** object. What you really need to understand, beyond the creation of a simple **`Object()`** object like ***cody***, is that the majority of values expressed in JavaScript are objects (primitive values like "foo", 5, and **`true`** are the exception but have equivalent wrapper objects).

Consider that the ***`cody`*** object created from the **`Object()`** constructor function is not really different from a string object created via the **`String()`** constructor function. To drive this fact home, examine and contrast the following code:

Sample: sample03.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myObject = new Object(); // Produces an Object() object.
    myObject['0'] = 'f';
    myObject['1'] = 'o';
    myObject['2'] = 'o';

    console.log(myObject); // Logs Object { 0="f", 1="o", 2="o"}

    var myString = new String('foo'); // Produces a String() object.

    console.log(myString); // Logs foo { 0="f", 1="o", 2="o"}

</script></body></html>
```

As it turns out, ***`myObject`*** and ***`myString`*** are both . . . objects! They both can have properties, inherit properties, and are produced from a constructor function. The ***`myString`*** variable containing the ***`'foo'`*** string value seems to be as simple as it goes, but amazingly it's got an object structure under its surface. If you examine both of the objects produced you will see that they are identical objects in substance but not in type. More importantly, I hope you begin to see that JavaScript uses objects to express values.

Notes

You might find it odd to see the string value ***`'foo'`*** in object form because typically a string is represented in JavaScript as a primitive value (e.g., **`var myString = 'foo';`**). I specifically used a string object value here to highlight that anything can be an object, including values that we might not typically think of as an object (e.g., string, number, Boolean). Also, I think this helps explain why some say that everything in JavaScript can be an object.

JavaScript bakes the **`String()`** and **`Object()`** constructor functions into the language itself to make the creation of a **`String()`** object and **`Object()`** object trivial. But you, as a coder of the JavaScript language, can also create equally powerful constructor functions. In the following sample, I demonstrate this by defining a non-native custom ***`Person()`*** constructor function so that I can create people from it.

Sample: sample04.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Define Person constructor function in order to create custom Person()
objects later.
```

```
    var Person = function (living, age, gender) {
        this.living = living;
        this.age = age;
        this.gender = gender;
        this.getGender = function () { return this.gender; };
    };

    // Instantiate a Person object and store it in the cody variable.
    var cody = new Person(true, 33, 'male');

    console.log(cody);

    /* The String() constructor function that follows, having been defined by
JavaScript, has the same pattern. Because the string constructor is native to
JavaScript, all we have to do to get a string instance is instantiate it. But
the pattern is the same whether we use native constructors like String() or
user-defined constructors like Person(). */

    // Instantiate a String object stored in the myString variable.
    var myString = new String('foo');

    console.log(myString);

</script></body></html>
```

The user-defined *`Person()`* constructor function can produce *`Person`* objects, just as the native **`String()`** constructor function can produce string objects. The *`Person()`* constructor is no less capable, and is no more or less malleable, than the native **`String()`** constructor or any of the native constructors found in JavaScript.

Remember how the *`cody`* object we first looked at was produced from an **`Object()`**. It's important to note that the **`Object()`** constructor function and the new *`Person()`* constructor shown in the previous code example can give us identical outcomes. Both can produce an identical object with the same properties and property methods. Examine the two sections of code that follow, showing that *`codyA`* and *`codyB`* have the same object values even though they are produced in different ways.

Sample: sample05.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Create a codyA object using the Object() constructor.

    var codyA = new Object();
    codyA.living = true;
    codyA.age = 33;
    codyA.gender = 'male';
    codyA.getGender = function () { return codyA.gender; };

    console.log(codyA); // Logs Object {living=true, age=33, gender="male",
...}

    /* The same cody object is created below, but instead of using the native
Object() constructor to create a one-off cody, we first define our own
Person() constructor that can create a cody object (and any other Person
object we like) and then instantiate it with "new". */

    var Person = function (living, age, gender) {
        this.living = living;
        this.age = age;
        this.gender = gender;
        this.getGender = function () { return this.gender; };
    };

    var codyB = new Person(true, 33, 'male');

    console.log(codyB); // Logs Object {living=true, age=33, gender="male",
...}

</script></body></html>
```

The main difference between the ***codyA*** and ***codyB*** objects is not found in the object itself, but in the constructor functions used to produce the objects. The ***codyA*** object was produced using an instance of the **`Object()`** constructor. The ***`Person()`*** constructor produced ***codyB***, but can also be used as a powerful, centrally defined object "factory" to be used for creating more ***`Person()`*** objects. Crafting your own constructors for producing custom objects also sets up prototypal inheritance for ***`Person()`*** instances.

Both solutions resulted in the same complex object being created. It's these two patterns that are most commonly used for constructing objects.

JavaScript is really just a language that is prepackaged with a few native object constructors used to produce complex objects which express a very specific type of value (e.g., numbers, strings, functions, objects, arrays, etc.), as well as the raw materials via **`Function()`** objects for crafting user-defined object constructors (e.g., ***`Person()`***). The end result—no matter the pattern for creating the object—is typically the creation of a complex object.

Understanding the creation, nature, and usage of objects and their primitive equivalents is the focus of the rest of this book.