JavaScript Objects in Detail

JAN. 27 2013 169

JavaScript's core—most often used and most fundamental—data type is the Object data type. JavaScript has one complex data type, the Object data type, and it has five simple data types: Number, String, Boolean, Undefined, and Null. Note that these simple (primitive) data types are immutable (cannot be changed), while objects are mutable (can be changed).

What is an Object

An object is an unordered list of primitive data types (and sometimes reference data types) that is stored as a series of name-value pairs. Each item in the list is called a *property* (functions are called *methods*).

Consider this simple object:

```
var myFirstObject = {firstName: "Richard",
favoriteAuthor: "Conrad"};
```

Think of an object as a list that contains items, and each item (a property or a method) in the list is stored by a name-value pair. The property names in the example above are firstName and favoriteAuthor. And the values are "Richard" and "Conrad."

Property names can be a string or a number, but if the property name is a number, it has to be accessed with the bracket notation.

More on bracket notation later. Here is another example of objects with numbers as the property name:

```
var ageGroup = {30: "Children", 100:"Very Old"};

console.log(ageGroup.30) // This will throw an error

// This is how you will access the value of the property 30, to get value "Children"

console.log(ageGroup["30"]); // Children


//It is best to avoid using numbers as property names.
```

As a JavaScript developer you will most often use the object data type, mostly for storing data and for creating your own custom methods and functions.

Reference Data Type and Primitive Data Types

One of the main differences between reference data type and primitive data types is reference data type's value is stored as a reference, it is not stored directly on the variable, as a value, as the primitive data types are. For example:

```
// The primitive data type String is stored as a value

var person = "Kobe";

var anotherPerson = person; // anotherPerson = the value of person

person = "Bryant"; // value of person changed


console.log(anotherPerson); // Kobe

console.log(person); // Bryant
```

It is worth noting that even though we changed *person* to "Bryant," the *anotherPerson* variable still retains the value that person had.

Compare the primitive data saved-as-value demonstrated above with the save-as-reference for objects:

```
var person = {name: "Kobe"};

var anotherPerson = person;

person.name = "Bryant";


console.log(anotherPerson.name); // Bryant

console.log(person.name); // Bryant
```

In this example, we copied the *person* object to *anotherPerson*, but because the value in person was stored as a reference and not an actual value, when we changed the person.name property to “Bryant” the anotherPerson reflected the change because it never stored an actual copy of it’s own value of the person’s properties, it only had a reference to it.

Object Data Properties Have Attributes

Each data property (object property that store data) has not only the name-value pair, but also 3 attributes (the three attributes are set to true by default):
— Configurable Attribute: Specifies whether the property can be deleted or changed.
— Enumerable: Specifies whether the property can be returned in a for/in loop.
— Writable: Specifies whether the property can be changed.

Note that ECMAScript 5 specifies accessor properties along with the data properties noted above. And the accessor properties are functions (getters and setters). We will learn more about ECMAScript 5 in an already-scheduled post slated for February 15.

Creating Objects

These are the two common ways to create objects:

1. Object Literals
   The most common and, indeed, the easiest way to create objects is with the object literal described here:

```
// This is an empty object initialized using
the object literal notation
var myBooks = {};


// This is an object with 4 items, again
using object literal
var mango = {
color: "yellow",
shape: "round",
sweetness: 8,


howSweetAmI: function () {
console.log("Hmm Hmm Good");
}
}
```

2. Object Constructor
   The second most common way to create objects is with Object constructor. A constructor is a function used for initializing new objects, and you use the new keyword to call the constructor.

```
var mango = new Object ();
mango.color = "yellow";
mango.shape= "round";
mango.sweetness = 8;

mango.howSweetAmI = function () {
console.log("Hmm Hmm Good");
}
```

While you can use some reserved word such as “for” as property names in your objects, it is wise to avoid this altogether.

Objects can contain any other data type, including Numbers, Arrays, and even other Objects.

Practical Patterns for Creating Objects

For simple objects that may only ever be used once in your application to store data, the two methods used above would suffice for creating objects.

Imagine you have an application that displays fruits and detail about each fruit. All fruits in your application have these properties: color, shape, sweetness, cost, and a showName function. It would be quite tedious and counterproductive to type the following every time you want to create a new fruit object.

```
var mangoFruit = {
color: "yellow",
sweetness: 8,
fruitName: "Mango",
nativeToLand: ["South America", "Central
America"],

showName: function () {
console.log("This is " + this.fruitName);
},
nativeTo: function () {
this.nativeToLand.forEach(function
(eachCountry) {
console.log("Grown in:" + eachCountry);
});
}
}
```

If you have 10 fruits, you will have to add the same code 10 times. And what if you had to make a change to the nativeTo function? You will have to make the change in 10 different places. Now extrapolate

this to adding objects for members on a website and suddenly you realized the manner in which we have created objects so far is not ideal objects that will have instances, particularly when developing large applications.

To solve these repetitive problems, software engineers have invented patterns (solutions for repetitive and common tasks) to make developing applications more efficient and streamlined.

Here are two common patterns for creating objects. If you have done the Learn JavaScript Properly course, you would have seen the lessons in the Code Academy used this first pattern frequently:

1. Constructor Pattern for Creating Objects

```
function Fruit (theColor, theSweetness,
theFruitName, theNativeToLand) {


this.color = theColor;

this.sweetness = theSweetness;

this.fruitName = theFruitName;

this.nativeToLand = theNativeToLand;


this.showName = function () {

console.log("This is a " + this.fruitName);

}


this.nativeTo = function () {

this.nativeToLand.forEach(function
(eachCountry) {

console.log("Grown in:" + eachCountry);

});

}




}
```

With this pattern in place, it is very easy to create all sorts of fruits. Thus:

```
var mangoFruit = new Fruit ("Yellow", 8,
"Mango", ["South America", "Central
America", "West Africa"]);

mangoFruit.showName(); // This is a Mango.

mangoFruit.nativeTo();

//Grown in:South America

// Grown in:Central America

// Grown in:West Africa
```

```
var pineappleFruit = new Fruit ("Brown", 5,
"Pineapple", ["United States"]);

pineappleFruit.showName(); // This is a
Pineapple.
```

If you had to change the fruitName function, you only had to do it in one location. The pattern encapsulates all the functionalities and characteristics of all the fruits in by making just the single Fruit function with inheritance.

Notes:
— An inherited property is defined on the object's prototype property. For example: someObject.prototype.firstName = "rich";

— An own property is defined directly on the object itself, for example:
// Let's create an object first:
var aMango = new Fruit ();
// Now we define the mangoSpice property directly on the aMango object
// Because we define the mangoSpice property directly on the aMango object, it is an own property of aMango, not an inherited property.
aMango.mangoSpice = "some value";

— To access a property of an object, we use object.property, for example:
console.log(aMango.mangoSpice); // "some value"

— To invoke a method of an object, we use object.method(), for example:
// First, lets add a method
aMango.printStuff = function () {return "Printing";}

// Now we can invoke the printStuff method:
aMango.printStuff ();

2. Prototype Pattern for Creating Objects

```
function Fruit () {


}


Fruit.prototype.color = "Yellow";

Fruit.prototype.sweetness = 7;

Fruit.prototype.fruitName = "Generic
Fruit";

Fruit.prototype.nativeToLand = "USA";


Fruit.prototype.showName = function () {

console.log("This is a " +
this.fruitName);
```

```
}


Fruit.prototype.nativeTo = function () {
console.log("Grown in:" +
this.nativeToLand);
}
```

And this is how we call the Fruit () constructor in this prototype pattern:

```
var mangoFruit = new Fruit ();
mangoFruit.showName(); //
mangoFruit.nativeTo();
// This is a Generic Fruit
// Grown in:USA
```

Further Reading

For a complete discussion on these two patterns and a thorough explanation of how each work and the disadvantages of each, read Chapter 6 of *Professional JavaScript for Web Developers*. You will also learn which pattern Zakas recommends as the best one to use (Hint: it is neither of the 2 above).

How to Access Properties on an Object

The two primary ways of accessing properties of an object are with dot notation and bracket notation.

1. Dot Notation

```
// We have been using dot notation so far in
the examples above, here is another example
again:
var book = {title: "Ways to Go", pages: 280,
bookMark1:"Page 20"};


// To access the properties of the book
object with dot notation, you do this:
console.log ( book.title); // Ways to Go
console.log ( book.pages); // 280
```

2. Bracket Notation

```
// To access the properties of the book
object with bracket notation, you do this:
console.log ( book["title"]); //Ways to Go
console.log ( book["pages"]); // 280


//Or, in case you have the property name in
a variable:
```

```
var bookTitle = "title";

console.log ( book[bookTitle]); // Ways to
Go

console.log (book["bookMark" + 1]); // Page
20
```

Accessing a property on an object that does not exist will result in*undefined*.

Own and Inherited Properties

Objects have inherited properties and own properties. The own properties are properties that were defined on the object, while the inherited properties were inherited from the object's Prototype object.

To find out if a property exists on an object (either as an inherited or an own property), you use the in operator:

```
// Create a new school object with a property
name schoolName

var school = {schoolName:"MIT"};


// Prints true because schoolName is an own
property on the school object

console.log("schoolName" in school); // true


// Prints false because we did not define a
schoolType property on the school object, and
neither did the object inherit a schoolType
property from its prototype object
Object.prototype.

console.log("schoolType" in school); // false

// Prints true because the school object
inherited the toString method from
Object.prototype.

console.log("toString" in school); // true
```

hasOwnProperty

To find out if an object has a specific property as one of its own property, you use the hasOwnProperty method. This method is very useful because from time to time you need to enumerate an object and you want only the own properties, not the inherited ones.

```
// Create a new school object with a property
name schoolName

var school = {schoolName:"MIT"};


// Prints true because schoolName is an own
property on the school object

console.log(school.hasOwnProperty
```

```
("schoolName")); // true

// Prints false because the school object
inherited the toString method from
Object.prototype, therefore toString is not
an own property of the school object.

console.log(school.hasOwnProperty
("toString")); // false
```

Accessing and Enumerating Properties on Objects

To access the enumerable (own and inherited) properties on objects, you use the for/in loop or a general for loop.

```
// Create a new school object with 3 own
properties: schoolName, schoolAccredited, and
schoolLocation.

var school = {schoolName:"MIT",
schoolAccredited: true,
schoolLocation:"Massachusetts"};


//Use of the for/in loop to access the
properties in the school object

for (var eachItem in school) {

console.log(eachItem); // Prints schoolName,
schoolAccredited, schoolLocation


}
```

Accessing Inherited Properties

Properties inherited from Object.prototype are not enumerable, so the for/in loop does not show them. However, inherited properties that are enumerable are revealed in the for/in loop iteration.
For example:

```
//Use of the for/in loop to access the
properties in the school object

for (var eachItem in school) {

console.log(eachItem); // Prints schoolName,
schoolAccredited, schoolLocation


}


// Create a new HigherLearning function that
the school object will inherit from.

/* SIDE NOTE: As Wilson (an astute reader)
correctly pointed out in the comments below,
the educationLevel property is not actually
inherited by objects that use the
HigherLearning constructor; instead, the
educationLevel property is created as a new
property on each object that uses the
HigherLearning constructor. The reason the
```

```
property is not inherited is because we use
of the "this" keyword to define the property.
*/



function HigherLearning () {
this.educationLevel = "University";
}


// Implement inheritance with the
HigherLearning constructor
var school = new HigherLearning ();
school.schoolName = "MIT";
school.schoolAccredited = true;
school.schoolLocation = "Massachusetts";



//Use of the for/in loop to access the
properties in the school object
for (var eachItem in school) {
console.log(eachItem); // Prints
educationLevel, schoolName, schoolAccredited,
and schoolLocation
}
```

In the last example, note the educationLevel property that was defined on the HigherLearning function is listed as one of the school's properties, even though educationLevel is not an own property—it was inherited.

Object's Prototype Attribute and Prototype Property

The prototype attribute and prototype property of an object are critically important concepts to understand in JavaScript. Read my post JavaScript Prototype in Plain, Detailed Language for more.

Deleting Properties of an Object

To delete a property from an object, you use the delete operator. You cannot delete properties that were inherited, nor can you delete properties with their attributes set to configurable. You must delete the inherited properties on the prototype object (where the properties were defined). Also, you cannot delete properties of the global object, which were declared with the var keyword.

The delete operator returns true if the delete was successful. And surprisingly, it also returns true if the property to delete was nonexistent or the property could not be deleted (such as non-configurable or not owned by the object).

These examples illustrate:

```
var christmasList = {mike:"Book", jason:"sweate

delete christmasList.mike; // deletes the mike
property


for (var people in christmasList) {

console.log(people);

}

// Prints only jason

// The mike property was deleted


delete christmasList.toString; // returns true,
toString not deleted because it is an inherited


// Here we call the toString method and it works
fine—wasn't deleted

christmasList.toString(); //"[object Object]"


// You can delete a property of an instance if t
property is an own property of that instance. Fo
example, we can delete the educationLevel proper
from the school's object we created above becaus
educationLevel property is defined on the instan
used the "this" keyword to define the property w
declare the HigherLearning function. We did not
the educationLevel property on the HigherLearnin
function's prototype.


console.log(school.hasOwnProperty("educationLeve
true

// educationLevel is an own property on school,
can delete it

delete school.educationLevel; true


// The educationLevel property was deleted from
school instance

console.log(school.educationLevel); undefined


// But the educationLevel property is still on t
HigherLearning function

var newSchool = new HigherLearning ();

console.log(newSchool.educationLevel); // Univer


// If we had defined a property on the HigherLea
function's prototype, such as this educationLeve
property:

HigherLearning.prototype.educationLevel2 = "Univ
2";


// Then the educationLevel2 property on the inst
```

```
of HigherLearning would not be own property.


// The educationLevel2 property is not an own pi
on the school instance

console.log(school.hasOwnProperty("educationLeve
false

console.log(school.educationLevel2); // Universi


// Let's try to delete the inherited educationLe
property

delete school.educationLevel2; true (always retu
true, as noted earlier)


// The inherited educationLevel2 property was nc
deleted

console.log(school.educationLevel2); University
```

Serialize and Deserialize Objects

To transfer your objects via HTTP or to otherwise convert it to a string, you will need to serialize it (convert it to a string); you can use the JSON.stringify function to serialize your objects. Note that prior to ECMAScript 5, you had to use a popular json2 library (by Douglas Crockford) to get the JSON.stringify function. It is now standardized in ECMAScript 5.

To Deserialize your object (convert it to an object from a string), you use the JSON.parse function from the same json2 library. This function too has been standardized by ECMAScript 5.

JSON.stringify Examples:

```
var christmasList = {mike:"Book", jason:"sweate
chelsea:"iPad" }

JSON.stringify (christmasList);

// Prints this string:

// "
{"mike":"Book","jason":"sweater","chels":"iPad"


// To print a stringified object with formattinc
add "null" and "4" as parameters:

JSON.stringify (christmasList, null, 4);

// "{

"mike": "Book",

"jason": "sweater",

"chels": "iPad"

}"
```

```
// JSON.parse Examples \\

// The following is a JSON string, so we cannot
access the properties with dot notation (like
christmasListStr.mike)

var christmasListStr =
'{"mike":"Book","jason":"sweater","chels":"iPad'


// Let's convert it to an object

var christmasListObj = JSON.parse
(christmasListStr);


// Now that it is an object, we use dot notatio

console.log(christmasListObj.mike); // Book
```

For more detailed coverage of JavaScript Objects, including ECMAScript 5 additions for dealing with objects, read chapter 6 of JavaScript: The Definitive Guide 6th Edition.