Working with objects

by 56 contributors:              Show all...

« Previous Next »

JavaScript is designed on a simple object-based paradigm. An object is a collection of properties, and a property is an association between a name (or *key*) and a value. A property's value can be a function, in which case the property is known as a method. In addition to objects that are predefined in the browser, you can define your own objects. This chapter describes how to use objects, properties, functions, and methods, and how to create your own objects.

Objects overview

Objects in JavaScript, just as in many other programming languages, can be compared to objects in real life. The concept of objects in JavaScript can be understood with real life, tangible objects.

In JavaScript, an object is a standalone entity, with properties and type. Compare it with a cup, for example. A cup is an object, with properties. A cup has a color, a design, weight, a material it is made of, etc. The same way, JavaScript objects can have properties, which define their characteristics.

Objects and properties

A JavaScript object has properties associated with it. A property of an object can be explained as a variable that is attached to the object. Object properties are

basically the same as ordinary JavaScript variables, except for the attachment to objects. The properties of an object define the characteristics of the object. You access the properties of an object with a simple dot-notation:

```
objectName.propertyName
```

Like all JavaScript variables, both the object name (which could be a normal variable) and property name are case sensitive. You can define a property by assigning it a value. For example, let's create an object named `myCar` and give it properties named `make`, `model`, and `year` as follows:

```
var myCar = new Object();
myCar.make = "Ford";
myCar.model = "Mustang";
myCar.year = 1969;
```

Properties of JavaScript objects can also be accessed or set using a bracket notation (for more details see property accessors). Objects are sometimes called *associative arrays*, since each property is associated with a string value that can be used to access it. So, for example, you could access the properties of the `myCar` object as follows:

```
myCar["make"] = "Ford";
myCar["model"] = "Mustang";
myCar["year"] = 1969;
```

An object property name can be any valid JavaScript string, or anything that can be converted to a string, including the empty string. However, any property name that is not a valid JavaScript identifier (for example, a property name that has a space or a hyphen, or that starts with a number) can only be accessed using the square bracket notation. This notation is also very useful when property names are to be dynamically determined (when the property name is not determined until runtime). Examples are as follows:

```
// four variables are created and assigned in a single go,
// separated by commas
```

```
var myObj = new Object(),
    str = "myString",
    rand = Math.random(),
    obj = new Object();

myObj.type              = "Dot syntax";
myObj["date created"]   = "String with space";
myObj[str]              = "String value";
myObj[rand]             = "Random Number";
myObj[obj]              = "Object";
myObj[""]               = "Even an empty string";

console.log(myObj);
```

You can also access properties by using a string value that is stored in a variable:

```
var propertyName = "make";
myCar[propertyName] = "Ford";

propertyName = "model";
myCar[propertyName] = "Mustang";
```

You can use the bracket notation with `for...in` to iterate over all the enumerable properties of an object. To illustrate how this works, the following function displays the properties of the object when you pass the object and the object's name as arguments to the function:

```
function showProps(obj, objName) {
  var result = "";
  for (var i in obj) {
    if (obj.hasOwnProperty(i)) {
        result += objName + "." + i + " = " + obj[i] + "\n";
    }
  }
  return result;
}
```

So, the function call `showProps(myCar, "myCar")` would return the following:

```
```

```
myCar.make = Ford
myCar.model = Mustang
myCar.year = 1969
```

Enumerating all properties of an object

Starting with ECMAScript 5, there are three native ways to list/traverse object properties:

- `for...in` loops

  This method traverses all enumerable properties of an object and its prototype chain
- `Object.keys(o)`

  This method returns an array with all the own (not in the prototype chain) enumerable properties' names ("keys") of an object `o`.
- `Object.getOwnPropertyNames(o)`

  This method returns an array containing all own properties' names (enumerable or not) of an object `o`.

Before ECMAScript 5, there was no native way to list all properties of an object. However, this can be achieved with the following function:

```
function listAllProperties(o){
    var objectToInspect;
    var result = [];

    for(objectToInspect = o; objectToInspect !== null; objectToIn
        result = result.concat(Object.getOwnPropertyNames(objectT
    }

    return result;
}
```

This can be useful to reveal "hidden" properties (properties in the prototype chain which are not accessible through the object, because another property has the same name earlier in the prototype chain). Listing accessible properties only can easily be done by removing duplicates in the array.

Creating new objects

JavaScript has a number of predefined objects. In addition, you can create your own objects. You can create an object using an object initializer. Alternatively, you can first create a constructor function and then instantiate an object using that function and the `new` operator.

Using object initializers

In addition to creating objects using a constructor function, you can create objects using an object initializer. Using object initializers is sometimes referred to as creating objects with literal notation. "Object initializer" is consistent with the terminology used by C++.

The syntax for an object using an object initializer is:

```
var obj = { property_1:   value_1,   // property_# may be an iden
            2:            value_2,   // or a number...
            // ...,
            "property n": value_n }; // or a string
```

where `obj` is the name of the new object, each `property_i` is an identifier (either a name, a number, or a string literal), and each `value_i` is an expression whose value is assigned to the `property_i`. The `obj` and assignment is optional; if you do not need to refer to this object elsewhere, you do not need to assign it to a variable. (Note that you may need to wrap the object literal in parentheses if the object appears where a statement is expected, so as not to have the literal be confused with a block statement.)

Object initializers are expressions, and each object initializer results in a new object being created whenever the statement in which it appears is executed. Identical object initializers create distinct objects that will not compare to each other as equal. Objects are created as if a call to `new Object()` were made; that is, objects made from object literal expressions are instances of `Object`.

The following statement creates an object and assigns it to the variable `x` if and only if the expression `cond` is true:

```
if (cond) var x = {greeting: "hi there"};
```

The following example creates `myHonda` with three properties. Note that the `engine` property is also an object with its own properties.

```
var myHonda = {color: "red", wheels: 4, engine: {cylinders: 4, si
```

You can also use object initializers to create arrays. See array literals.

Using a constructor function

Alternatively, you can create an object with these two steps:

1. Define the object type by writing a constructor function. There is a strong convention, with good reason, to use a capital initial letter.
2. Create an instance of the object with `new`.

To define an object type, create a function for the object type that specifies its name, properties, and methods. For example, suppose you want to create an object type for cars. You want this type of object to be called `car`, and you want it to have properties for make, model, and year. To do this, you would write the following function:

```
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}
```

Notice the use of `this` to assign values to the object's properties based on the values passed to the function.

Now you can create an object called `mycar` as follows:

```
var mycar = new Car("Eagle", "Talon TSi", 1993);
```

This statement creates `mycar` and assigns it the specified values for its properties. Then the value of `mycar.make` is the string "Eagle", `mycar.year` is the integer 1993, and so on.

You can create any number of `car` objects by calls to `new`. For example,

```
var kenscar = new Car("Nissan", "300ZX", 1992);
var vpgscar = new Car("Mazda", "Miata", 1990);
```

An object can have a property that is itself another object. For example, suppose you define an object called `person` as follows:

```
function Person(name, age, sex) {
  this.name = name;
  this.age = age;
  this.sex = sex;
}
```

and then instantiate two new `person` objects as follows:

```
var rand = new Person("Rand McKinnon", 33, "M");
var ken = new Person("Ken Jones", 39, "M");
```

Then, you can rewrite the definition of `car` to include an `owner` property that takes a `person` object, as follows:

```
function Car(make, model, year, owner) {
  this.make = make;
  this.model = model;
  this.year = year;
  this.owner = owner;
}
```

To instantiate the new objects, you then use the following:

```
var car1 = new Car("Eagle", "Talon TSi", 1993, rand);
var car2 = new Car("Nissan", "300ZX", 1992, ken);
```

Notice that instead of passing a literal string or integer value when creating the new objects, the above statements pass the objects `rand` and `ken` as the arguments for the owners. Then if you want to find out the name of the owner of car2, you can access the following property:

```
car2.owner
```

Note that you can always add a property to a previously defined object. For example, the statement

```
car1.color = "black";
```

adds a property `color` to car1, and assigns it a value of "black." However, this does not affect any other objects. To add the new property to all objects of the same type, you have to add the property to the definition of the `car` object type.

Using the `Object.create` method

Objects can also be created using the `Object.create()` method. This method can be very useful, because it allows you to choose the prototype object for the object you want to create, without having to define a constructor function.

```
// Animal properties and method encapsulation
var Animal = {
  type: "Invertebrates", // Default value of properties
  displayType : function(){  // Method which will display type of
    console.log(this.type);
  }
}

// Create new animal type called animal1
```

```
var animal1 = Object.create(Animal);
animal1.displayType(); // Output:Invertebrates

// Create new animal type called Fishes
var fish = Object.create(Animal);
fish.type = "Fishes";
fish.displayType(); // Output:Fishes
```

Inheritance

All objects in JavaScript inherit from at least one other object. The object being inherited from is known as the prototype, and the inherited properties can be found in the `prototype` object of the constructor. See Inheritance and the prototype chain for more information.

Indexing object properties

You can refer to a property of an object either by its property name or by its ordinal index. If you initially define a property by its name, you must always refer to it by its name, and if you initially define a property by an index, you must always refer to it by its index.

This restriction applies when you create an object and its properties with a constructor function (as we did previously with the `Car` object type) and when you define individual properties explicitly (for example, `myCar.color = "red"`). If you initially define an object property with an index, such as `myCar[5] = "25 mpg"`, you can subsequently refer to the property only as `myCar[5]`.

The exception to this rule is objects reflected from HTML, such as the `forms` array. You can always refer to objects in these arrays by either their ordinal number (based on where they appear in the document) or their name (if defined). For example, if the second `<FORM>` tag in a document has a `NAME` attribute of "myForm", you can refer to the form as `document.forms[1]` or `document.forms["myForm"]` or `document.myForm`.

Defining properties for an object type

You can add a property to a previously defined object type by using the `prototype`

property. This defines a property that is shared by all objects of the specified type, rather than by just one instance of the object. The following code adds a `color` property to all objects of type `car`, and then assigns a value to the `color` property of the object `car1`.

```
Car.prototype.color = null;
car1.color = "black";
```

See the `prototype` property of the `Function` object in the JavaScript reference for more information.

Defining methods

A *method* is a function associated with an object, or, simply put, a method is a property of an object that is a function. Methods are defined the way normal functions are defined, except that they have to be assigned as the property of an object. See also method definitions for more details. An example is:

```
objectName.methodname = function_name;

var myObj = {
  myMethod: function(params) {
    // ...do something
  }
};
```

where `objectName` is an existing object, `methodname` is the name you are assigning to the method, and `function_name` is the name of the function.

You can then call the method in the context of the object as follows:

```
object.methodname(params);
```

You can define methods for an object type by including a method definition in the object constructor function. For example, you could define a function that would format and display the properties of the previously-defined `car` objects; for

example,

```
function displayCar() {
  var result = "A Beautiful " + this.year + " " + this.make
    + " " + this.model;
  pretty_print(result);
}
```

where `pretty_print` is a function to display a horizontal rule and a string. Notice the use of `this` to refer to the object to which the method belongs.

You can make this function a method of `car` by adding the statement

```
this.displayCar = displayCar;
```

to the object definition. So, the full definition of `car` would now look like

```
function Car(make, model, year, owner) {
  this.make = make;
  this.model = model;
  this.year = year;
  this.owner = owner;
  this.displayCar = displayCar;
}
```

Then you can call the `displayCar` method for each of the objects as follows:

```
car1.displayCar();
car2.displayCar();
```

Using `this` for object references

JavaScript has a special keyword, `this`, that you can use within a method to refer to the current object. For example, suppose you have a function called `validate` that validates an object's `value` property, given the object and the high and low

values:

```
function validate(obj, lowval, hival) {
  if ((obj.value < lowval) || (obj.value > hival))
    alert("Invalid Value!");
}
```

Then, you could call `validate` in each form element's `onchange` event handler, using `this` to pass it the element, as in the following example:

```
<input type="text" name="age" size="3"
  onChange="validate(this, 18, 99)">
```

In general, `this` refers to the calling object in a method.

When combined with the `form` property, `this` can refer to the current object's parent form. In the following example, the form `myForm` contains a `Text` object and a button. When the user clicks the button, the value of the `Text` object is set to the form's name. The button's `onclick` event handler uses `this.form` to refer to the parent form, `myForm`.

```
<form name="myForm">
<p><label>Form name:<input type="text" name="text1" value="Beluga
<p><input name="button1" type="button" value="Show Form Name"
     onclick="this.form.text1.value = this.form.name">
</p>
</form>
```

Defining getters and setters

A getter is a method that gets the value of a specific property. A setter is a method that sets the value of a specific property. You can define getters and setters on any predefined core object or user-defined object that supports the addition of new properties. The syntax for defining getters and setters uses the object literal syntax.

The following illustrates how getters and setters could work for a user-defined object `o`.

```
var o = {
  a: 7,
  get b() {
    return this.a + 1;
  },
  set c(x) {
    this.a = x / 2
  }
};

console.log(o.a); // 7
console.log(o.b); // 8
o.c = 50;
console.log(o.a); // 25
```

The `o` object's properties are:

- `o.a` — a number
- `o.b` — a getter that returns `o.a` plus 1
- `o.c` — a setter that sets the value of `o.a` to half of the value `o.c` is being set to

Please note that function names of getters and setters defined in an object literal using "[gs]et *property*()" (as opposed to `__define[GS]etter__`) are not the names of the getters themselves, even though the `[gs]et` *`propertyName`*`(){ }` syntax may mislead you to think otherwise. To name a function in a getter or setter using the "[gs]et *property*()" syntax, define an explicitly named function programmatically using `Object.defineProperty` (or the `Object.prototype.__defineGetter__` legacy fallback).

The following code illustrates how getters and setters can extend the `Date` prototype to add a `year` property to all instances of the predefined `Date` class. It uses the `Date` class's existing `getFullYear` and `setFullYear` methods to support the `year` property's getter and setter.

These statements define a getter and setter for the year property:

```
var d = Date.prototype;
Object.defineProperty(d, "year", {
  get: function() {return this.getFullYear() },
  set: function(y) { this.setFullYear(y) }
});
```

These statements use the getter and setter in a `Date` object:

```
var now = new Date;
console.log(now.year); // 2000
now.year = 2001; // 987617605170
console.log(now);
// Wed Apr 18 11:13:25 GMT-0700 (Pacific Daylight Time) 2001
```

In principle, getters and setters can be either

- defined using object initializers, or
- added later to any object at any time using a getter or setter adding method.

When defining getters and setters using object initializers all you need to do is to prefix a getter method with `get` and a setter method with `set`. Of course, the getter method must not expect a parameter, while the setter method expects exactly one parameter (the new value to set). For instance:

```
var o = {
  a: 7,
  get b() { return this.a + 1; },
  set c(x) { this.a = x / 2; }
};
```

Getters and setters can also be added to an object at any time after creation using the `Object.defineProperties` method. This method's first parameter is the object on which you want to define the getter or setter. The second parameter is

an object whose property names are the getter or setter names, and whose property values are objects for defining the getter or setter functions. Here's an example that defines the same getter and setter used in the previous example:

```
var o = { a:0 }

Object.defineProperties(o, {
    "b": { get: function () { return this.a + 1; } },
    "c": { set: function (x) { this.a = x / 2; } }
});

o.c = 10 // Runs the setter, which assigns 10 / 2 (5) to the 'a'
console.log(o.b) // Runs the getter, which yields a + 1 or 6
```

Which of the two forms to choose depends on your programming style and task at hand. If you already go for the object initializer when defining a prototype you will probably most of the time choose the first form. This form is more compact and natural. However, if you need to add getters and setters later — because you did not write the prototype or particular object — then the second form is the only possible form. The second form probably best represents the dynamic nature of JavaScript — but it can make the code hard to read and understand.

Deleting properties

You can remove a non-inherited property by using the `delete` operator. The following code shows how to remove a property.

```
// Creates a new object, myobj, with two properties, a and b.
var myobj = new Object;
myobj.a = 5;
myobj.b = 12;

// Removes the a property, leaving myobj with only the b property
delete myobj.a;
console.log ("a" in myobj) // yields "false"
```

You can also use `delete` to delete a global variable if the `var` keyword was not used to declare the variable:

```
g = 17;
delete g;
```

Comparing Objects

In JavaScript objects are a reference type. Two distinct objects are never equal, even if they have the same properties. Only comparing the same object reference with itself yields true.

```
// Two variables, two distict objects with the same properties
var fruit = {name: "apple"};
var fruitbear = {name: "apple"};

fruit == fruitbear // return false
fruit === fruitbear // return false
```

```
// Two variables, a single object
var fruit = {name: "apple"};
var fruitbear = fruit;  // assign fruit object reference to fruit

// here fruit and fruitbear are pointing to same object
fruit == fruitbear // return true
fruit === fruitbear // return true
```

For more information about comparison operators, see Comparison operators.

See also

- To dive deeper, read about the details of javaScript's objects model.
- To learn about ECMAScript6 classes (a new way to create objects), read the JavaScript classes chapter.

« Previous Next »