

JavaScript object basics

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

Prerequisites:	A basic understanding of HTML and CSS, familiarity with JavaScript basics (see First steps and Building blocks).
Objective:	To understand the basics of working with objects in JavaScript: creating objects, accessing and modifying object properties, and using constructors.

Object basics

An object is a collection of related data and/or functionality. These usually consist of several variables and functions (which are called properties and methods when they are inside objects). Let's work through an example to understand 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. We'll use this as a basis for exploring basic object syntax. While working with this example you should have your [developer tools JavaScript console](#) open and ready to type in some commands.

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

JS



```
const person = {};
```

Now open your browser's [JavaScript console](#), enter `person` into it, and press `Enter` / `Return`. You should get a result similar to one of the below lines:

```
[object Object]  
Object { }  
{ }
```


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

JS



```
const person = {  
  name: ["Bob", "Smith"],  
  age: 32,  
  bio: function () {  
    console.log(`${this.name[0]} ${this.name[1]} is ${this.age} years  
old.`);  
  },  
  introduceSelf: function () {  
    console.log(`Hi! I'm ${this.name[0]}.`);  
  },  
};
```

After saving and refreshing, try entering some of the following into the JavaScript console on your browser devtools:

```
JS   
person.name;  
person.name[0];  
person.age;  
person.bio();  
// "Bob Smith is 32 years old."  
person.introduceSelf();  
// "Hi! I'm Bob."
```

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

So what is going on here? Well, an object is made up of multiple members, each of which has 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:

JS



```
const 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 number, an array, and two functions. The first two 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**.

When the object's members are functions there's a simpler syntax. Instead of `bio: function ()` we can write `bio()`. Like this:

JS



```
const person = {  
  name: ["Bob", "Smith"],  
  age: 32,  
  bio() {  
    console.log(`${this.name[0]} ${this.name[1]} is ${this.age} years  
old.`);  
  },  
  introduceSelf() {  
    console.log(`Hi! I'm ${this.name[0]}.`);  
  }  
};
```

```
},  
};
```

From now on, we'll use this shorter syntax.

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 different compared 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 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:

```
JS
```



```
person.age;  
person.bio();
```

Objects as object properties

An object property can itself be an object. For example, try changing the `name` member from

JS



```
const person = {  
  name: ["Bob", "Smith"],  
};
```

to

JS



```
const person = {  
  name: {  
    first: "Bob",  
    last: "Smith",  
  },  
  // ...  
};
```

To access these items you just need to chain the extra step onto the end with another dot. Try these in the JS console:

JS



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

If you do this, you'll also need to go through your method code and change any instances of

```
JS  
name[0];  
name[1];
```

to

```
JS  
name.first;  
name.last;
```

Otherwise, your methods will no longer work.

Bracket notation

Bracket notation provides an alternative way to access object properties. Instead of using [dot notation](#) like this:

```
JS  
person.age;  
person.name.first;
```

You can instead use square brackets:

JS



```
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.

Dot notation is generally preferred over bracket notation because it is more succinct and easier to read. However there are some cases where you have to use square brackets. For example, if an object property name is held in a variable, then you can't use dot notation to access the value, but you can access the value using bracket notation.

In the example below, the `logProperty()` function can use `person[propertyName]` to retrieve the value of the property named in `propertyName`.

JS




```
const person = {  
  name: ["Bob", "Smith"],  
  age: 32,  
};  
  
function logProperty(propertyName) {  
  console.log(person[propertyName]);  
}
```



```
}  
  
logProperty("name");  
// ["Bob", "Smith"]  
logProperty("age");  
// 32
```

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 declaring the member you want to set (using dot or bracket notation), like this:

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

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

```
JS   
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 in the JS console:

JS



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

You can now test out your new members:

JS



```
person["eyes"];  
person.farewell();  
// "Bye everybody!"
```

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:

JS



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

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

JS



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

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

JS



```
const myDataName = "height";  
const myDataValue = "1.75m";  
person[myDataName] = myDataValue;
```

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

JS



```
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:

JS



```
introduceSelf() {  
  console.log(`Hi! I'm ${this.name[0]}.`);  
}
```

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?

Well, when you only have to create a single object literal, it's not so useful. But if you create more than one, `this` enables you to use the same method definition for every object you create.

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

```
JS
const person1 = {
  name: "Chris",
  introduceSelf() {
    console.log(`Hi! I'm ${this.name}.`);
  },
};

const person2 = {
  name: "Deepti",
  introduceSelf() {
    console.log(`Hi! I'm ${this.name}.`);
  },
};
```

In this case, `person1.introduceSelf()` outputs "Hi! I'm Chris."; `person2.introduceSelf()` on the other hand outputs "Hi! I'm Deepti.", even though the method's code is exactly the same in each case. This isn't hugely useful when you are writing out object literals by hand, but it will


be essential when we start using **constructors** to create more than one object from a single object definition, and that's the subject of the next section.

Introducing constructors

Using object literals is fine when you only need to create one object, but if you have to create more than one, as in the previous section, they're seriously inadequate. We have to write out the same code for every object we create, and if we want to change some properties of the object - like adding a `height` property - then we have to remember to update every object.

We would like a way to define the "shape" of an object — the set of methods and the properties it can have — and then create as many objects as we like, just updating the values for the properties that are different.

The first version of this is just a function:

```
JS   
  
function createPerson(name) {  
  const obj = {};  
  obj.name = name;  
  obj.introduceSelf = function () {  
    console.log(`Hi! I'm ${this.name}.`);  
  };  
  return obj;  
}
```

This function creates and returns a new object each time we call it. The object will have two members:

- a property `name`
- a method `introduceSelf()`.

Note that `createPerson()` takes a parameter `name` to set the value of the `name` property, but the value of the `introduceSelf()` method will be the same for all objects created using this function. This is a very common pattern for creating objects.

Now we can create as many objects as we like, reusing the definition:

```
JS
const salva = createPerson("Salva");
salva.introduceSelf();
// "Hi! I'm Salva."

const frankie = createPerson("Frankie");
frankie.introduceSelf();
// "Hi! I'm Frankie."
```

This works fine but is a bit long-winded: we have to create an empty object, initialize it, and return it. A better way is to use a **constructor**. A constructor is just a function called using the `new` keyword. When you call a constructor, it will:

- create a new object
- bind `this` to the new object, so you can refer to `this` in your constructor code
- run the code in the constructor
- return the new object.

Constructors, by convention, start with a capital letter and are named for the type of object they create. So we could rewrite our example like this:

JS



```
function Person(name) {  
  this.name = name;  
  this.introduceSelf = function () {  
    console.log(`Hi! I'm ${this.name}.`);  
  };  
}
```

To call `Person()` as a constructor, we use `new`:

JS



```
const salva = new Person("Salva");  
salva.introduceSelf();  
// "Hi! I'm Salva."  
  
const frankie = new Person("Frankie");  
frankie.introduceSelf();  
// "Hi! I'm Frankie."
```

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 in our own basic custom examples.

So when you used string methods like:

JS



```
myString.split(",");
```

You were using a method available on a `String` object. 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 and properties available on it.

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

JS



```
const myDiv = document.createElement("div");  
const myVideo = document.querySelector("video");
```


You were using methods available on a `Document` object. 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 and properties available on it.

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

Note that built in objects and 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:

JS



```
const myNotification = new Notification("Hello!");
```

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see [Test your skills: Object basics](#).

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 look at **prototypes**, which is the fundamental way that JavaScript lets an object inherit properties from other objects.

Help improve MDN

Was this page helpful to you?



[Learn how to contribute.](#)

This page was last modified on Feb 29, 2024 by [MDN contributors](#).

