W3Schools OOP JavaScript

Basics of JavaScript Objects

Real life objects, properties, and methods

In real life, a car is an *object*.

A car has *properties* like weight and color, and *methods* like start and stop.

```
// properties
car.name = 'Fiat';
car.model = '500';
car.weight = '850 kg';
car.color = 'white';

// methods
car.start();
car.drive();
car.brake();
car.stop();
```

All cars have the same properties, but the property values differ from car to car.

All cars have the same methods, but the methods are performed at different times.

JavaScript objects

We have already learned that JavaScript variables are containers for data values.

This code assigns a *simple value* (Fiat) to a *variable* named **car**:

```
var car = 'Fiat';
```

Objects are variables too. But objects can contain many values.

This code assigns *many values* (Fiat, 500, white) to a *variable* named **car**:

```
var car = {type: 'Fiat', model: '500', color: 'white'};
```

The values are written as *name:value pairs* (name and value separated by a colon). JavaScript objects are containers for *named values*.

Object properties

The name:value pairs in JavaScript objects are called *properties*.

```
var person = {firstName: 'John', lastName: 'Doe', age: 50, eyeColor: 'blue'};
```

Property	Property Value
firstName	John
lastName	Doe
age	50
eyeColor	blue

Object methods

Methods are *actions* that can be performed on objects.

Methods are stored in properties as *function definitions*.

Property	Property Value
firstName	John
lastName	Doe
age	50
eyeColor	blue
fullName	**function()** *{return this.firstName+this.lastName;}*

JavaScript objects are containers for named values called properties or methods.

Object definition

You define (and create) a JavaScript object with an object literal:

```
var person = {firstName: 'John', lastName: 'Doe',
             age: 50, eyeColor: 'blue'};
```

Spaces and line breaks are not important. An object definition can span multiple lines:

```
var person = {
    firstName:'John',
    lastName:'Doe',
    age:50,
    eyeColor:'blue'
};
```

Accessing object properties

You can access object properties in two ways:

```
objectName.propertyName
```

or

```
objectName["propertyName"]
```

Examples

```
person.lastName;
person['lastName'];
```

Accessing object methods

You can access an object method with the following syntax:

```
objectName.methodName()
```

Example

```
name = person.fullName();
```

If you access the **fullName** method *without ()*, it will return the *function definition*:

```
name = person.fullName;
```

A method is actually a function definition stored as a property value.

Do not declare strings, numbers, or booleans as objects!!

When a JavaScript variable is declared with the keyword **new**, this variable is created as an object:

```
var x = new String(); // Declares x as a String object
var y = new Number(); // Declares y as a Number object
var z = new Boolean(); // Declares z as a Boolean object
```

Avoid **String**, **Number**, and **Boolean** objects. They complicate your code and slow down execution speed.

Object Definitions

JavaScript Objects

In JavaScript, objects are king. If you understand objects, you understand JavaScript.

In JavaScript, *almost* everything is an object.

- Booleans can be objects (or primitive data treated as objects)
- Numbers can be objects (or primitive data treated as objects)
- Strings can be objects (or primitive data treated as objects)
- Dates are *always* objects
- Maths are *always* objects
- Regular expressions are *always* objects
- Arrays are *always* objects
- Functions are *always* objects
- Objects are objects

In JavaScript all values, except primitive values, are objects. Primitive values are strings ('John Doe'), numbers (3.14), true, false, null, and undefined.

Objects are variables containing variables

JavaScript variables can contain single values:

```
var person = 'John Doe';
```

Objects are variables too. But an object can contain many values. Again, tehse values are stored as *key:value* pairs. A JavaScript object is a *collection of named values*. Objects written as name:value pairs are similar to:

- Associative arrays in PHP
- Dictionaries in Python
- Hash tables in C
- Hash maps in Java
- Hashes in Ruby and Perl

Object methods

Methods are *actions* that can be performed on objects. Object properties can be both primitive values, other objects, and functions. An *object method* is an object property containing a *function definition*. JavaScript objects are containers for named values, called *properties* and *methods*.

Creating a JavaScript object

With JavaScript, you can define and create your own objects.

There are different ways to create new objects:

- Define and create a single object, using an *object literal*.
- Define and creaet a single object with the keyword **new**.
- Define an *object constructor*, and then create objects of the constructed type.
- In **ECMAScript 5**, an object can also be created with the function **Object.create()**.

Using an object literal

This is the easiest way to create a JavaScript object.

Using an *object literal*, you both define and create an object in one statement. An object literal is a list of name:value pairs (such as **age:50;**) inside curly braces **{}**. The following example creates a new JavaScript object with four properties:

```
var person = {firstName:'John', lastName:'Doe', age:50, eyeColor:'blue'};
```

Using the JavaScript keyword new

This example will also create the aforementioned obejct:

```
var person = new Object();
person.firstName = 'John';
// etc.
```

The two examples above do exactly the same thing. There is no need to use **new Object()**. For simplicity, readability, and execution speed, use the first one (the *object literal* method).

Using an object constructor

The examples above are limited in many situations. They only create a single object. Sometimes we like to have an *object type* that can be used to create many objects of one type. The standard way to create an object type is to use an *object constructor function*:

```
function person(first, last, age, eye) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eye;
}

var myFather = new person('John', 'Doe', 50, 'blue');
var myMother = new person('Sally', 'Rally', 48, 'green');
```

The above function **person()** is an object constructor. Once you have an object constructor, you can create new objects of the same type:

```
var myFather = new person('John', 'Doe', 50, 'blue');
```

```
// etc
```

The this keyword

- In JavaScript, the thing called **this** is the object that *owns* the JavaScript code.
- The value of **this**, when used in a function, is the object that *owns* the function.
- The value of **this**, when used in an object, is the object itself.
- The **this** keyword in an object constructor does not have a value. It is only a substitute for the new object.
- The value of **this** will become the new object when the constructor is used to create an object.
- Note that **this** is not a *variable*. It is a *keyword*. You cannot change the value of **this**.

Built-in JavaScript constructors

JavaScript has built-in constructors for native objects.

Example

```
var x1 = new Object();          // A new Object object
var x2 = new String();          // A new String object
var x3 = new Number();          // A new Number object
var x4 = new Boolean();         // A new Boolean object
var x5 = new Array();           // A new Array object
var x6 = new RegExp();          // A new RegExp object
var x7 = new Function();        // A new Function object
var x8 = new Date();            // A new Date object
```

The **Math()** object is not on the list. **Math** is a global object. The **new** keyword cannot be used on **Math**.

Avoiding complex objects when you can

As you can see, JavaScript has object versions of the primitive data types **String**, **Number**, and **Boolean**. There is no reason to create complex objects. Primitive values execute much faster:

- Instead of using `new Array()`, use *array literals*: **[ ]**
- Instead of using `new RegExp()`, use *pattern literals*: **/( )/**
- Instead of using `new Function()`, use *function expressions*: **function( ) { }**
- Instead of using `new Object()`, use *object literals*: **{ }**

Example

```
var x1 = {};            // new object
var x2 = "";            // new primitive string
var x3 = 0;             // new primitive number
var x4 = false;         // new primitive boolean
var x5 = [];            // new array object
var x6 = /()/;          // new regexp object
var x7 = function(){};  // new function object.
```

JavaScript objects are mutable

Objects are mutable: They are addressed by *reference*, not *value*. If **person** is an object, the following statement will *not* create a copy of **person**:

```
var x = person; // will not create a copy of person
```

The object **x** is *not a copy* of **person**. It *is* **person**. Both **x** and **person** are the same object. Any changes to **x** will also change **person**, because **person** and **x** are the *same object*.

Example

```
var person = {firstName: 'John', lastName: 'Doe', age: 50, eyeColor: 'blue'};

var x = person;
x.age = 10;         // This will change both x.age and person.age
```

Note that JavaScript *variables* are not mutable. This only applies to JavaScript *objects*.

JavaScript Object Properties

Properties are the most important part of any JavaScript object.

JavaScript properties

Properties are the values associated with a JavaScript object. A JavaScript object is a *collection of unordered properties*. Properties can usually be changed, added, and deleted, but some are read-only.

Accessing JavaScript properties

The syntax for accessing the property of an object is:

`objectName.property // person.age`

or

`objectName["property"] // person["age"]`

or

`objectName[expression] // x = "age"; person[x]`

- Here the expression must evaluate to a property name.

Examples

```
person.firstName + " is " + person.age + " years old.";
// or
person["firstName"] + " is " + person["age"] + " years old.";
```

JavaScript for..in loop

The JavaScript **for..in** statement loops through the properties of an object.

Syntax

```
for (variable in object) {
    code to be executed
}
```

The block of code inside the **for..in** loop will be executed once for each property. The following example loops through the properties of an object.

Example

```
var person = {fname:'John', lname:'Doe', age:25};

for (x in person) {
    txt += person[x];
}
```

Adding new properties

You can add new properties to an existing object by simply giving it a value. Assume here that the **person** object already exists - you can then give it new properties:

```
person.nationality = 'English';
```

You cannot use *reserved words* for property (or method) names. All JavaScript naming rules apply.

Deleting properties

The **delete** keyword deletes a property from an object.

Example

```
delete person.age; // or delete person['age'];
```

The **delete** keyword deletes both the value of the property and the property itself. After deletion, the property cannot be used unless it is added back in again. The **delete** operator is designed to be used on object properties. It has no effect on variables or functions. The **delete** operator should not be used on predefined JavaScript object properties. That can *crush your application*.

Property Attributes

All properties have a *name*. In addition, they also have a *value*.

The value is one of a property's attributes. Other attributes are:

- *enumerable*
- *configurable*
- *writable*

These attributes define how the property can be accessed: *is it readable?*, *is it writable?*, etc.

In JavaScript all attributes can be *read*, but only the value attribute can be changed (and then only if the property is writable).

- *ECMAScript 5* has methods for both getting and setting all property attributes.

Prototype properties

JavaScript objects *inherit* the properties of their *prototype*. The **delete** keyword does not delete inherited properties, but if you delete a prototype properly it will affect all objects inherited from the prototype.

JavaScript Object Methods

JavaScript methods

JavaScript methods are actions that can be performed on objects. A JavaScript *method* is a property containing a *function definition*. Methods are *functions stored as object properties*.

Accessing object methods

You create an object method with the following syntax:

`methodName : function() { code lines }`

You access an object method with the following syntax:

`objectName.methodName()`

The **fullName** property (created earlier) will execute (as a function) when it is invoked with **()**. The following example accesses the **fullName()** method of a **person** object:

```
name = person.fullName(); // returns the function's return value as expected.
```

If you access the **fullName** *property*, without the parentheses, it will return the *function definition*:

```
name = person.fullName; // returns function definition, not the function's return value.
```

Using built-in methods

This example uses the **toUpperCase()** method of the **String** object to convert text to uppercase:

```
var message = 'Hello World';
var x = message.toUpperCase();
```

The value of **x**, after execution of the above code, will be: `HELLO WORLD`

Adding new methods

Adding methods to an object is done inside the constructor function.

Example

```
function person(firstName, lastName, age, eyeColor) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
    this.eyeColor = eyeColor;
    this.changeName = function(name) {
        this.lastName = name;
    }
}
```

Here the **changeName()** function assigns the value of **name** to the **person** object's **lastName** property.

Example

```
myMother.changeName('Doe');
```

JavaScript knows which person you are talking about by *substituting* **this** with **myMother**.

JavaScript Object Prototypes

Every JavaScript object has a *prototype*. The prototype is also an object. All JavaScript objects inherit their properties and methods from their prototype.

JavaScript prototypes

- Again, all JavaScript objects inherit their properties and methods from their prototype.
- Objects created using an *object literal*, or with **new Object()**, inherit from a prototype called **Object.prototype**.
- Objects created with **new Date()** inherit from **Date.prototype**.
- **Object.prototype** is on the top of the *prototype chain*.
- All JavaScript objects (**Date**, **Array**, **RegExp**, **Function**, ...) inherit from **Object.prototype**.

Creating a prototype

The standard way to create an *object prototype* is to use an *object constructor function*.

Example

```
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyeColor;
}
```

With a constructor function, you can use the **new** keyword to create new objects from the same prototype.

Example

```
var myFather = new Person('John', 'Doe', 50, 'blue');
// etc
```

The constructor function is the prototype of **Person** objects. It is considered good practice to name a constructor function with an upper-case first letter.

Adding properties and methods to objects

You will want to add new properties (or methods) to:

- an existing object:

  ```
  myFather.nationality = 'English';
  myFather.name = function() {
    return this.firstName + " " + this.lastName;
  }
  ```

- all existing objects of a given type
- an object prototype
  - You can't do this the same way as with an existing object, because *the prototype is not an existing object*.
  - `Person.nationality = 'English'` will not work.
  - To add a new property to a prototype, you need to add it to the constructor function:

```
function Person(first, last, age, eyecolor) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eyeColor;
  this.nationality = 'English';
}
```

- Prototype properties can have *prototype values* (i.e. *default* values).

Adding methods to a prototype

Your constructor function can also define methods.

Example

```
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
    this.name = function() {return this.firstName + " " + this.lastName;};
}
```

Using the prototype property

The JavaScript **prototype** property allows you to add new properties to an existing prototype.

Example

```
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
}

Person.prototype.nationality = 'English';
```

The JavaScript **prototype** property also allows you to add new methods to an existing prototype.

Example

```
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
}
```

```
Person.prototype.name = function() {
    return this.firstName + " " + this.lastName;
};
```

Only modify *your own* prototypes. Never modify the prototypes of standard JavaScript objects.