Contents

[JavaScript 3](#_Toc408560783)

[Books 3](#_Toc408560784)

[Skills Matter Events 3](#_Toc408560785)

[Primitive Types 4](#_Toc408560786)

[Number 4](#_Toc408560787)

[String 5](#_Toc408560788)

[Boolean 6](#_Toc408560789)

[Undefined 6](#_Toc408560790)

[Null 6](#_Toc408560791)

[Variables 7](#_Toc408560792)

[Global variables 7](#_Toc408560793)

[Arrays 8](#_Toc408560794)

[Array literal notation 8](#_Toc408560795)

[Check for array-ness 8](#_Toc408560796)

[Objects 9](#_Toc408560797)

[Object literal notation 9](#_Toc408560798)

[Functions 10](#_Toc408560799)

[Features 10](#_Toc408560800)

[Types 11](#_Toc408560801)

[Constructors 11](#_Toc408560802)

[Built-in functions 12](#_Toc408560803)

[Closure 13](#_Toc408560804)

[Example 13](#_Toc408560805)

[Prototypes 14](#_Toc408560806)

[Best Practices 17](#_Toc408560807)

[Use JSLint with use strict directive 17](#_Toc408560808)

[Use single var pattern 17](#_Toc408560809)

[Cache length in for loops 18](#_Toc408560810)

[Use for-in loops for objects 18](#_Toc408560811)

[Avoid implied typecasting 18](#_Toc408560812)

[eval() is evil 19](#_Toc408560813)

[Use recursion carefully 19](#_Toc408560814)

[Design Patterns 21](#_Toc408560815)

[Singleton 21](#_Toc408560816)

[Memoization pattern 21](#_Toc408560817)

[Namespace pattern 21](#_Toc408560818)

[Module pattern 22](#_Toc408560819)

[Chaining pattern 22](#_Toc408560820)

[Mix-ins 23](#_Toc408560821)

[Currying 23](#_Toc408560822)

[Examples 25](#_Toc408560823)

# JavaScript

## Books

* **JavaScript: The Good Parts** by Douglas Crockford
* **Head First HTML5 Programming: Building Web Apps with JavaScript** by E. Robson & E. Freeman
* **Head First JavaScript Programming** by E. Freeman & E. Robson
* **Eloquent JavaScript - A Modern Introduction to Programming** by Marijn Haverbeke
* **Object-Oriented JavaScript** by Stoyan Stefanov
* **JavaScript Patterns** by Stoyan Stefanov
* **High Performance JavaScript** by Nicholas Zakas
* **High Performance Web Sites** by Steve Sounders

Currently reading ...

* **Even Faster Web Sites** by Steve Sounders

## Skills Matter Events

* Javascript Tools and Workflow
* A few JavaScript ideas stolen from AngularJS source

## Primitive Types

### Number

**NaN**

It turns out that despite its name, "Not A Number", NaN is a special value that is also a number.

>>> typeof NaN  
**"number"**  
>>> var a = NaN;  
>>> a  
**NaN**

You get **NaN** when you try to perform an operation that assumes numbers but the operation fails. For example, if you try to multiply 10 by the character "f", the result is **NaN**, because "f" is obviously not a valid operand for a multiplication.

>>> var a = 10 \* "f";  
>>> a  
**NaN**

NaN is contagious, so if you have even only one NaN in your arithmetic operation, the whole result goes down the drain.

>>> 1 + 2 + NaN  
**NaN**

The value **NaN** is not equivalent to anything — including NaN! The core JavaScript function isNaN() can be used to evaluate whether a value is NaN or not.

**Infinity**

There is a special value in JavaScript called Infinity. It represents a number too big for JavaScript to handle. Infinity is indeed a number, as typing typeof Infinity in the console will confim. You can also quickly check that a number with 308 zeros is ok, but 309 zeros is too much. To be precise, the biggest number JavaScript can handle is 1.7976931348623157e+308 while the smallest is 5e-324.

**isFinite()**

isFinite() checks whether the input is a number that is neither Infinity nor NaN.

>>> isFinite(Infinity)  
**false**  
>>> isFinite(12)  
**true**  
>>> isFinite(1e308)  
**true**  
>>> isFinite(1e309)  
**false**

### String

Here's an example that shows the difference between a string object and a primitive string data type.

>>> var primitive = 'Hello'  
>>> typeof primitive;  
**"string"**

>>> var obj = new String('world');  
>>> typeof obj;  
**"object"**

Primitive strings are not objects, so they don't have any methods or properties. But JavaScript still offers you the syntax to treat primitive strings as objects. String objects are being created (and then destroyed) behind the scenes every time you access a primitive string as if it was an object:

**parseInt()**

Using parseInt() you can get a numeric value from a string. The function accepts a second radix parameter, which is often omitted but shouldn’t be. The problems occur when the string to parse starts with 0: for example, a part of a date entered into a form field. Strings that start with 0 are treated as octal numbers (base 8). To avoid inconsistency and unexpected results, always specify the radix parameter:

var month = "06",  
 year = "09";  
month = parseInt(month, 10);  
year = parseInt(year, 10);

In this example, if you omit the radix parameter like parseInt(year), the returned value will be 0, because “09” assumes octal number (as if you did parseInt(year, 8)) and 09 is not a valid digit in base 8.

Alternative ways to convert a string to a number include:

+"08" // result is 8  
Number("08") // 8

These are often faster than parseInt(), because parseInt(), as the name suggests, parses and doesn’t simply convert. But if you’re expecting input such as “08 hello”, parseInt() will return a number, whereas the others will fail with NaN.

### Boolean

**truthy / falsy**

Most values convert to true with the exception of the following (which convert to false):

* "" (empty string)
* null
* undefined
* 0
* NaN
* false

These six values are sometimes referred to as being *falsy*, while all others are *truthy* (including, for example, the strings "0", " ", and "false").

Apart from the six falsy values, everything else is truthy in JavaScript, including the empty objects. This also means that all boolean objects created with new Boolean() evaluate to true, as they are objects. Let's create two boolean objects, one truthy and one falsy:

>>> var b1 = new Boolean(true)

>>> b1.valueOf()

true

>>> var b2 = new Boolean(false)

>>> b2.valueOf()

false

Now let's convert them to primitive boolean values. They both convert to true because all objects are truthy.

>>> Boolean(b1)

**true**

>>> Boolean(b2)

**true**

### Undefined

When you try to access a variable that doesn't exist, you get the special value undefined. The same will happen when you have declared a variable, but not given it a value yet. JavaScript will initialize it behind the scenes, with the value undefined.

### Null

This is another special data type that can have only one value, the null value. It means no value, an empty value, nothing. The difference with undefined is that if a variable has a value null, it is still defied, it only happens that its value is nothing.

## Variables

### Global variables

It is surprisingly easy to create globals involuntarily because of two JavaScript features. First, you can use variables without even declaring them. And second, JavaScript has the notion of *implied globals*, meaning that any variable you don’t declare becomes a property of the global object (and is accessible just like a properly declared global variable). Consider the following example:

function sum(x, y) {

// antipattern: implied global

**result = x + y;**  
return result;

}

In this code, result is used without being declared. The code works fine, but after calling the function you end up with one more variable result in the global namespace that can be a source of problems.

The rule of thumb is to always declare variables with var, as demonstrated in the improved version of the sum() function:

function sum(x, y) {

**var result = x + y;**  
return result;

}

## Arrays

### Array literal notation

The array literal notation is simpler and preferred.

// array of three elements  
// warning: antipattern  
var a = new Array("one", "two", "three");

// the exact same array  
var a = ["one", "two", "three"];

One more reason to stay away from new Array() is to avoid a possible trap that this constructor has in store for you.

When you pass a single number to the Array() constructor, it doesn’t become the value of the first array element. It sets the length of the array instead. This means that new Array(3) creates an array with length of 3, but no actual elements. If you try to access any of the elements, you get the value undefined because the elements don’t exist.

### Check for array-ness

Using the typeof operator with array operands returns “object.”

console.log(typeof [1, 2]); // "object"

Although this behaviour makes sense (arrays are objects), it’s not too helpful. Often you need to know if a value actually is an array. ECMAScript 5 defines a new method Array.isArray(), which returns true if the argument is an array. For example:

Array.isArray([]); // true

If this new method is not available in your environment, you can make the check by calling the Object.prototype.toString() method. If you invoke the call() method of toString in the context of an array, it should return the string “[object Array]”. If the context is an object, it should return the string “[object Object]”. So you can do something like this:

if (typeof Array.isArray === "undefined") {

Array.isArray = function (arg) {

return Object.prototype.toString.call(arg) === "[object Array]";

};

}

## Objects

### Object literal notation

var dog = {

name: 'Benji',

talk: function(){

alert('Woof, woof!');

}

};

There are two ways to access a property of an object:

* Using square bracket notation, for example dog[“name”]
* Using the dot notation, for example dog.name

One case where you need square brackets is if the name of the property you need to access is not known beforehand. During runtime, it is dynamically stored in a variable:

var key = 'name';

dog[key];

Accessing a non-existing property:

>>> typeof dog.breed

**"undefied"**

Adding a property:

dog.breed = 'Border Terrier';

Deleting a property:

delete dog.breed;

Using this value:

When you say this, you are actually saying "this object" or "the current object":

var dog = {

name: 'Benji',

sayName: function() {

return this.name;

}

}

## Functions

### Features

There are two main features of functions in JavaScript that make them special:

* the first is that functions are *objects* and
* the second is that they provide scope.

In general, when you think of a function in JavaScript, think of an object, with the only special feature that this object is invokable, meaning it can be executed.

As functions are objects they can:

* have properties e.g. functionname.length which returns the number of parameters

function test(param1, param2) {

// function body

}

>>> test.length

**2**

* be assigned to variables or deleted

>>> var sum = function(a, b) {return a + b;}

>>> var add = sum;

>>> delete sum

**true**

>>> typeof sum;

**"undefied"**

>>> typeof add;

**"function"**

>>> add(1, 2);

**3**

* be passed into functions as arguments
* be returned from functions

A function may not require any parameters, but if it does and you forget to pass them, JavaScript will assign the value undefined to the ones you skipped.

By using the arguments array you can improve the sum() function to accept any number of parameters and add them all up.

function sumOnSteroids() {

var i, res = 0;

var number\_of\_params = arguments.length;

for (i = 0; i < number\_of\_params; i++) {

res += arguments[i];

}

return res;

}

One way of handling optional parameters:

// If parameter type is not passed in, it is set to 'any'

function (action, arg, type) {

var pubtype = type || 'any';

// ...

}

### Types

Function declaration:

function f(){return 1;}

Function literal notation

var f = function(){return 1;}

Anonymous function:

function(a){return a;}

Named function expression:

var add = function add(a, b) {

return a + b;

};

Immediate function:

**(**function() {

alert('watch out!');

}());

### Constructors

As a best practice the first letter should be uppercase. When a constructor function is called with the new operator it returns the value this.

var developer = new Person();

Here’s a pattern that helps you make sure your constructor always behaves as a constructor. Instead of adding all members to this, you add them to that and then return that.

function Person() {

var that = {};

that.name = "Scott";

return that;

}

### Built-in functions

**Date()**

The Date constructor can be used in different ways. One thing to remember is that the month starts from 0:

// Produces a date object for the current time.  
new Date();

// February (!) 1st, 1980  
new Date(1980, 1, 1);

**RegExp()**

The regex objects provide two methods you can use to find matches: test() and exec() . They both accept a string parameter. test() returns a boolean (true when there's a match, false otherwise), while exec() returns an array of matched strings. Obviously exec() is doing more work, so use test() unless you really need to do something with the matches. People often use regular expressions for validation purposes, in this case test() would probably be enough.

var re = new RegExp('j.\*t');

var rei = new RegExp('j.\*t', 'i');

No match, because of the capital J:

>>> re.test("Javascript")

>>> /j.\*t/.test("Javascript")  
**false**

Case insensitive test gives a positive result:

>>> rei.test("Javascript")

>>> /j.\*t/i.test("Javascript")  
**true**

The same test using exec() returns an array and you can access the fist element as shown below:

>>> rei.exec("Javascript")[0]

>>> /j.\*t/i.exec("Javascript")[0]  
**"Javascript"**

## Closure

### Example

The following function makes it possible to dynamically create function values that add a certain number to their argument:

function makeAdder(amount) {

return function(number) {

return number + amount;

};

}

>>> var addTwo = makeAdder(2);

>>> addTwo(3);

→ 5

This feature is called *closure*, and a function that “closes over” some local variables is called *a closure*.

## Prototypes

JavaScript is classified as having a prototype-based object model. The functions in JavaScript are objects and they contain methods and properties.

Every function e.g. a constructor gets a prototype property pointing to a blank object with a constructor property pointing to the created function.

prototype is a property that gets created as soon as you define a function. Its initial value is an empty object.

>>> function foo(a, b){return a \* b;}

>>> typeof foo.prototype

**"object"**

One way of implementing inheritance:

function Shape(){

this.name = 'shape';

this.toString = function() {return this.name;};

}

function TwoDShape(){

}

function Triangle(side, height) {

this.side = side;

this.height = height;

this.getArea = function(){return this.side \* this.height / 2;};

}

The code that performs the inheritance magic is as follows:

TwoDShape.prototype = new Shape();

Triangle.prototype = new TwoDShape();

When you completely overwrite the prototype (as opposed to just augmenting it), this has some negative side effects on the constructor property. Therefore, it's a good idea to reset the constructor after inheriting:

TwoDShape.prototype.constructor = TwoDShape;

Triangle.prototype.constructor = Triangle;

Augment prototype:

TwoDShape.prototype.name = '2D shape';

Triangle.prototype.name = 'Triangle';

Now let's test what we have so far. Creating a Triangle object and calling its own getArea() method works as expected:

>>> var my = new Triangle(5, 10);

>>> my.getArea();

**25**

Although the my object doesn't have its own toString() method, it inherited one and can call it. Note how the inherited method toString() binds the this object to my.

>>> my.toString()

**"Triangle"**

It's interesting to note what the JavaScript engine does when you call my.toString() :

* It loops through all of the properties of my and doesn't find a method called toString() .
* It looks at the object that my.\_\_proto\_\_ points to; this object is the instance new TwoDShape() created during the inheritance process.
* Now the JavaScript engine loops through the instance of TwoDShape and doesn't find a toString() method. It then checks the \_\_proto\_\_ of that object. This time \_\_proto\_\_ points to the instance created by new Shape().
* The instance of new Shape() is examined and toString() is fially found!
* This method is invoked in the context of my, meaning that this points to my.

If you ask my, "who's your constructor?" it will report it correctly because of the constructor property reset that we did after inheriting:

>>> my.constructor

**Triangle(side, height)**

Using the instanceof operator you can validate that my is an instance of all three constructors.

>>> my instanceof Shape

**true**

>>> my instanceof TwoDShape

**true**

>>> my instanceof Triangle

**true**

>>> my instanceof Array

**false**

The same happens when you call the isPropertyOf() method of the constructors  
passing my:

>>> Shape.prototype.isPrototypeOf(my)

**true**

>>> TwoDShape.prototype.isPrototypeOf(my)

**true**

>>> Triangle.prototype.isPrototypeOf(my)

**true**

>>> String.prototype.isPrototypeOf(my)

**false**

You can also play with hasOwnProperty() to see the difference between the own property versus a property coming down the prototype chain.

>>> my.hasOwnProperty('side')  
**true**  
>>> my.hasOwnProperty('name')  
**false**

## Best Practices

### Use JSLint with use strict directive

JSLint (<http://jslint.com>) is a JavaScript code quality tool created by Douglas Crockford that inspects your code and warns about potential problems. It’s highly recommended that you run your code through JSLint. The tool “will hurt your feelings” as its creator warns, but only in the beginning. You can quickly learn from your mistakes and adopt the essential habits of a professional JavaScript programmer. Having no JSLint error in your code also helps you be more confident in the code, knowing that you didn’t make a simple omission or syntax error in a hurry. In its default settings, JSLint expects your code to be strict mode–compliant.

"use strict";

The "use strict" directive is new in JavaScript 1.8.5 (ECMAScript version 5). It is not a statement, but a literal expression, ignored by earlier versions of JavaScript. The purpose of "use strict" is to indicate that the code should be executed in "strict mode". With strict mode, you cannot, for example, use undeclared variables.

Strict mode is supported in:

* Internet Explorer from version 10
* Firefox from version 4
* Chrome from version 13
* Safari from version 5.1
* Opera from version 12

### Use single var pattern

Using a single var statement at the top of your functions is a useful pattern to adopt. It has the following benefits:

* Provides a single place to look for all the local variables needed by the function
* Prevents logical errors when a variable is used before it’s defined
* Helps you remember to declare variables and therefore minimize globals
* Is less code (to type and to transfer over the wire)

The single var pattern looks like this:

function func() {

var a = 1,

b = 2,

sum = a + b,

myobject = {},

i,

j;

// function body...  
}

### Cache length in for loops

The usual for loop pattern looks like the following:

// sub-optimal loop  
for (var i = 0; i < myarray.length; i++) {

// do something with myarray[i]  
}

A problem with this pattern is that the length of the array is accessed on every loop iteration. This can slow down your code, especially when myarray is not an array but an HTMLCollection object.

HTMLCollections are objects returned by DOM methods such as:

* document.getElementsByName()
* document.getElementsByClassName()
* document.getElementsByTagName()

That’s why a better pattern for for loops is to cache the length of the array (or collection) you’re iterating over, as shown in the following example:

for (var i = 0, max = myarray.length; i < max; i++) {

// do something with myarray[i]  
}

### Use for-in loops for objects

For-in loops should be used to iterate over non array objects. Looping with for-in is also called enumeration.

Technically, you can also use for-in to loop over arrays (because in JavaScript arrays are objects), but it’s not recommended. It may lead to logical errors if the array object has already been augmented with custom functionality. Additionally, the order (the sequence) of listing the properties is not guaranteed in a for-in. So it’s preferable to use normal for loops with arrays and for-in loops for objects.

### Avoid implied typecasting

JavaScript implicitly typecasts variables when you compare them. That’s why comparisons such as false == 0 or "" == 0 return true. To avoid confusion caused by the implied typecasting, always use the === and !== operators that check both the values and the type of the expressions you compare:

var zero = 0;  
if (zero === false) {  
// not executing because zero is 0, not false  
}

// antipattern  
if (zero == false) {  
// this block is executed...  
}

### eval() is evil

If you spot the use of eval() in your code, remember the mantra “eval() is evil.” This function takes an arbitrary string and executes it as JavaScript code. When the code in question is known beforehand (not determined at runtime), there’s no reason to use eval(). If the code is dynamically generated at runtime, there’s often a better way to achieve the goal without eval(). For example, just using square bracket notation to access dynamic properties is better and simpler:

// antipattern  
var property = "name";  
alert(eval("obj." + property));

// preferred  
var property = "name";  
alert(obj[property]);

Using eval() also has security implications, because you might be executing code (for example coming from the network) that has been tampered with. This is a common antipattern when dealing with a JSON response from an Ajax request. In those cases it’s better to use the browsers’ built-in methods to parse the JSON response to make sure it’s safe and valid. For browsers that don’t support JSON.parse() natively, you can use a library from JSON.org.

It’s also important to remember that passing strings to setInterval(), setTimeout(), and the Function() constructor is, for the most part, similar to using eval() and therefore should be avoided.

### Use recursion carefully

// Example without recursion

function power(number, exponent) {

var result = 1;

if (exponent === undefined) {

exponent = 2;

}

for (var count = 0; count < exponent; count++) {

result \*= base;

}

return result;

}

// Example with recursion

function power(base, exponent) {

if (exponent == 0) {

return 1;

} else {

return base \* power(base, exponent - 1);

}

By calling itself, the function produces the same effect that was produced with a for loop before. There is one important problem: In most JavaScript implementations, this second version is about 10 times slower than the first one. In JavaScript, running through a simple loop is a *lot* cheaper than calling a function multiple times. On top of that, using a sufficiently large exponent to this function might cause the stack to overflow.

* Dilemma of speed versus elegance

## Design Patterns

### Singleton

The most basic implementation of the singleton in JavaScript is the object literal:

var single = {};

As you know, functions are objects and they have properties. You can assign the single instance to a property of the constructor function.

function Logger() {

if (typeof Logger.single\_instance === "undefined") {

Logger.single\_instance = this;

}

return Logger.single\_instance;

}

If you write var a = new Logger() , a will point to the newly created Logger.single\_instance property. A subsequent call var b = new Logger() will result in b pointing to the same Logger.single\_instance property, which is exactly what you wanted.

### Memoization pattern

You can add custom properties to your functions at any time. One use case for custom  
properties is to cache the results (the return value) of a function, so the next time the  
function is called, it doesn’t have to redo potentially heavy computations. Caching the  
results of a function is also known as memoization.

### Namespace pattern

Namespaces help reduce the number of globals required by our programs and at the same time also help avoid naming collisions or excessive name prefixing.

// global object

var MYAPP = {};

// constructors

MYAPP.Parent = function () {};

MYAPP.Child = function () {};

// a variable

MYAPP.some\_var = 1;

// an object container

MYAPP.modules = {};

// nested objects

MYAPP.modules.module1 = {};

MYAPP.modules.module1.data = {a: 1, b: 2};

MYAPP.modules.module2 = {};

### Module pattern

The module pattern is widely used because it provides structure and helps organize your code as it grows. Unlike other languages, JavaScript doesn’t have special syntax for packages, but the module pattern provides the tools to create self-contained decoupled pieces of code, which can be treated as black boxes of functionality and added, replaced, or removed according to the (ever-changing) requirements of the software you’re writing. The module pattern is a combination of several patterns, namely:

* Namespaces
* Immediate functions
* Private and privileged members

The notion of *privileged methods* doesn’t involve any specific syntax; it’s just a name given to the public methods that have access to the private members (and hence have more privileges).

* Declaring dependencies

It’s a good idea to declare the modules your code relies on at the top of your function or module. The declaration involves creating only a local variable and pointing to the desired module:

var myFunction = function () {

// dependencies

var event = YAHOO.util.Event,

dom = YAHOO.util.Dom;

// use event and dom variables

// for the rest of the function...

};

### Chaining pattern

The chaining pattern enables you to call methods on an object one after the other, without assigning the return values of the previous operations to variables and without having to split your calls on multiple lines:

myobj.method1("hello").method2().method3("world").method4();

When you create methods that have no meaningful return value, you can have them return this, the instance of the object they are working with. This will enable consumers of that object to call the next method chained to the previous:

var obj = {

value: 1,

increment: function () {

this.value += 1;

**return this;**

},

add: function (v) {

this.value += v;

**return this;**

},

shout: function () {

alert(this.value);

}

};

// chain method calls

obj.increment().add(3).shout(); // 5

// as opposed to calling them one by one

obj.increment();

obj.add(3);

obj.shout(); // 5

### Mix-ins

Instead of copying from one object, you can copy from any number of objects and mix them all into a new object. The implementation is simple; just loop through arguments and copy every property of every object passed to the function:

function mix() {

var arg, prop, child = {};

for (arg = 0; arg < arguments.length; arg += 1) {

for (prop in arguments[arg]) {

if (arguments[arg].hasOwnProperty(prop)) {

child[prop] = arguments[arg][prop];

}

}

}

return child;

}

Now that you have a generic mix-in function, you can pass any number of objects to it, and the result will be a new object that has the properties of all the source objects. Here is an example use:

var cake = mix(

{eggs: 2, large: true},

{butter: 1, salted: true},

{flour: "3 cups"},

{sugar: "sure!"}

);

### Currying

Currying has nothing to do with the spicy Indian dish; it comes from the name of the mathematician Haskell Curry. (The Haskell programming language is also named after him.) Currying is a transformation process—we transform a function. An alternative name for currying could be *schönfinkelisation*, after the name of another mathematician, Moses Schönfinkel, the original inventor of this transformation.

In JavaScript we can modify the add() function into a curried one that will handle partial application. Let’s take an example:

// a curried add()

// accepts partial list of arguments

function add(x, y) {

var oldx = x, oldy = y;

if (typeof oldy === "undefined") { // partial

return function (newy) {

return oldx + newy;

};

}

// full application

return x + y;

}

// test

typeof add(5); // "function"

add(3)(4); // 7

// create and store a new function

var add2000 = add(2000);

add2000(10); // 10

## Examples

A few JavaScript ideas stolen from AngularJS source:  
<https://github.com/SamuraiPrinciple/24-11-2014>

* Download and extract 24-11-2014-master.zip
* Open SpecRunner.html in browser
* Open and change 2-proto.js by replacing “\_\_” with correct value
* Refresh SpecRunner.html to check if tests are passing