



Are you a student?

[Get a yearly subscription for \\$45! →](#)

Dismiss



Code



Categories

Series

WEB DEVELOPMENT

The Essentials of Writing High Quality JavaScript

by [Stoyan Stefanov](#) 14 Sep 2011 70 Comments



16



39



243



Are you a student? [Get a yearly Tuts+ subscription for \\$45 →](#)

Twice a month, we revisit some of our readers' favorite posts from throughout the history of **Nettuts+**. This tutorial was first published in October, 2010.

The brilliant [Stoyan Stefanov](#), in promotion of his book, "[JavaScript Patterns](#)," was kind enough to contribute an excerpt of the book for our readers, which details the essentials of writing high quality JavaScript, such as avoiding globals, using single var declarations, pre-caching length in loops, following coding conventions, and more.

This excerpt also includes some habits not necessarily related to the code itself, but more about the overall code creation process, including writing API documentation, conducting peer reviews, and running JSLint. These habits and best practices can help you write better, more understandable, and maintainable code—code to be proud of (and be able to figure out) when revisiting it months and years down the road.

Writing Maintainable Code

Software bugs are costly to fix. And their cost increases over time, especially if the bugs creep into

the publicly released product. It's best if you can fix a bug right away, as soon you find it; this is when the problem your code solves is still fresh in your head. Otherwise you move on to other tasks and forget all about that particular code. Revisiting the code after some time has passed requires:

- Time to relearn and understand the problem
- Time to understand the code that is supposed to solve the problem

Another problem, specific to bigger projects or companies, is that the person who eventually fixes the bug is not the same person who created the bug (and also not the same person who found the bug). It's therefore critical to reduce the time it takes to understand code, either written by yourself some time ago or written by another developer in the team. It's critical to both the bottom line (business revenue) and the developer's happiness, because we would all rather develop something new and exciting instead of spending hours and days maintaining old legacy code.

Another fact of life related to software development in general is that usually more time is spent *reading* code than *writing* it. In times when you're focused and deep into a problem, you can sit down and in one afternoon create a considerable amount of code.

The code will probably work then and there, but as the application matures, many other things happen that require your code to be reviewed, revised, and tweaked. For example:

- Bugs are uncovered.
- New features are added to the application.
- The application needs to work in new environments (for example, new browsers appear on the market).
- The code gets repurposed.
- The code gets completely rewritten from scratch or ported to another architecture or even another language.

As a result of the changes, the few man-hours spent writing the code initially end up in man-weeks spent reading it. That's why creating maintainable code is critical to the success of an application.

Maintainable code means code that:

- Is readable
- Is consistent
- Is predictable
- Looks as if it was written by the same person
- Is documented

Minimizing Globals

JavaScript uses functions to manage scope. A variable declared inside of a function is local to that function and not available outside the function. On the other hand, global variables are those declared outside of any function or simply used without being declared.

Every JavaScript environment has a global object accessible when you use this outside of any function. Every global variable you create becomes a property of the global object. In browsers, for convenience, there is an additional property of the global object called `window` that (usually) points to the global object itself. The following code snippet shows how to create and access a global variable in a browser environment:

```
1 myglobal = "hello"; // antipattern
2 console.log(myglobal); // "hello"
3 console.log(window.myglobal); // "hello"
4 console.log(window["myglobal"]); // "hello"
5 console.log(this.myglobal); // "hello"
```

The Problem with Globals

The problem with global variables is that they are shared among all the code in your JavaScript application or web page. They live in the same global namespace and there is always a chance of naming collisions—when two separate parts of an application define global variables with the same name but with different purposes.

It's also common for web pages to include code not written by the developers of the page, for example:

- A third-party JavaScript library
- Scripts from an advertising partner
- Code from a third-party user tracking and analytics script
- Different kinds of widgets, badges, and buttons

Let's say that one of the third-party scripts defines a global variable, called, for example, `result`. Then later in one of your functions you define another global variable called `result`. The outcome of that is the last `result` variable overwrites the previous ones, and the third-party script may just stop working.

Therefore it's important to be a good neighbor to the other scripts that may be in the same page and use as few global variables as possible. Later in the book you learn about strategies to minimize the number of globals, such as the namespacing pattern or the self-executing immediate functions, but the most important pattern for having fewer globals is to always use `var` to declare 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:

```
1 function sum(x, y) {  
2   // antipattern: implied global  
3   result = x + y;  
4   return result;  
5 }
```

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:

```
1 function sum(x, y) {  
2   var result = x + y;  
3   return result;  
}
```

```
4 | }
```

Another antipattern that creates implied globals is to chain assignments as part of a var declaration. In the following snippet, `a` is local but `b` becomes global, which is probably not what you meant to do:

```
1 | // antipattern, do not use
2 | function foo() {
3 |     var a = b = 0;
4 |     // ...
5 | }
```

If you're wondering why that happens, it's because of the right-to-left evaluation. First, the expression `b = 0` is evaluated and in this case `b` is not declared. The return value of this expression is 0, and it's assigned to the new local variable declared with `var a`. In other words, it's as if you've typed:

```
1 | var a = (b = 0);
```

If you've already declared the variables, chaining assignments is fine and doesn't create unexpected globals. Example:

```
1 | function foo() {
2 |     var a, b;
3 |     // ... a = b = 0; // both local
4 | }
```

Yet another reason to avoid globals is portability. If you want your code to run in different environments (hosts), it's dangerous to use globals because you can accidentally overwrite a host object that doesn't exist in your original environment (so you thought the name was safe to use) but which does in some of the others.

Side Effects When Forgetting var

There's one slight difference between implied globals and explicitly defined ones—the difference is in the ability to undefine these variables using the delete operator:

- Globals created with var (those created in the program outside of any function) cannot be deleted.
- Implied globals created without var (regardless if created inside functions) can be deleted.

This shows that implied globals are technically not real variables, but they are properties of the global object. Properties can be deleted with the delete operator whereas variables cannot:

```
01 // define three globals
02 var global_var = 1;
03 global_novar = 2; // antipattern
04 (function () {
05     global_fromfunc = 3; // antipattern
06 })();
07
08 // attempt to delete
09 delete global_var; // false
10 delete global_novar; // true
11 delete global_fromfunc; // true
12
13 // test the deletion
14 typeof global_var; // "number"
15 typeof global_novar; // "undefined"
16 typeof global_fromfunc; // "undefined"
```

In ES5 strict mode, assignments to undeclared variables (such as the two antipatterns in the preceding snippet) will throw an error.

Access to the Global Object

In the browsers, the global object is accessible from any part of the code via the `window` property (unless you've done something special and unexpected such as declaring a local variable named `window`). But in other environments this convenience property may be called something else (or even not available to the programmer). If you need to access the global object without hard-coding the identifier `window`, you can do the following from any level of nested function scope:

```
1 var global = (function () {
```

```
2   return this;
3 } ());
```

This way you can always get the global object, because inside functions that were invoked as functions (that is, not as constructors with `new`) this should always point to the global object. This is actually no longer the case in ECMAScript 5 in strict mode, so you have to adopt a different pattern when your code is in strict mode. For example, if you're developing a library, you can wrap your library code in an immediate function and then from the global scope, pass a reference to this as a parameter to your immediate function.

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:

```
1  function func() {
2      var a = 1,
3          b = 2,
4          sum = a + b,
5          myobject = {},
6          i,
7          j;
8      // function body...
9  }
```

You use one var statement and declare multiple variables delimited by commas. It's a good practice to also *initialize* the variable with an initial value at the time you declare it. This can prevent logical errors (all uninitialized and declared variables are initialized with the value `undefined`) and also improve the code readability. When you look at the code later, you can get an idea about the intended use of a variable based on its initial value—for example, was it

supposed to be an object or an integer?

You can also do some actual work at the time of the declaration, like the case with `sum = a + b` in the preceding code. Another example is when working with DOM (Document Object Model) references. You can assign DOM references to local variables together with the single declaration, as the following code demonstrates:

```
1 function updateElement() {
2     var el = document.getElementById("result"),
3         style = el.style;
4     // do something with el and style...
5 }
```

Hoisting: A Problem with Scattered vars

JavaScript enables you to have multiple `var` statements anywhere in a function, and they all act as if the variables were declared at the top of the function. This behavior is known as hoisting. This can lead to logical errors when you use a variable and then you declare it further in the function. For JavaScript, as long as a variable is in the same scope (same function), it's considered declared, even when it's used before the var declaration. Take a look at this example:

```
1 // antipattern
2 myname = "global"; // global variable
3 function func() {
4     alert(myname); // "undefined"
5     var myname = "local";
6     alert(myname); // "local"
7 }
8 func();
```

In this example, you might expect that the first `alert()` will prompt "global" and the second will prompt "local." It's a reasonable expectation because, at the time of the first alert, `myname` was not declared and therefore the function should probably "see" the global `myname`. But that's not how it works. The first alert will say "undefined" because `myname` is considered declared as a local variable to the function. (Although the declaration comes after.) All the variable declarations get hoisted to the top of the function. Therefore to avoid this type of confusion, it's best to declare upfront all variables you intend to use.

The preceding code snippet will behave as if it were implemented like so:

```
1 myname = "global"; // global variable
2 function func() {
3     var myname; // same as -> var myname = undefined;
4     alert(myname); // "undefined"
5     myname = "local";
6     alert(myname); // "local"
7 }
8 func();
```

For completeness, let's mention that actually at the implementation level things are a little more complex. There are two stages of the code handling, where variables, function declarations, and formal parameters are created at the first stage, which is the stage of parsing and entering the context. In the second stage, the stage of runtime code execution, function expressions and unqualified identifiers (undeclared variables) are created. But for practical purposes, we can adopt the concept of hoisting, which is actually not defined by ECMAScript standard but is commonly used to describe the behavior.

for Loops

In `for` loops you iterate over `arrays` or array-like objects such as `arguments` and `HTMLCollection` objects. The usual `for` loop pattern looks like the following:

```
1 // sub-optimal loop
2 for (var i = 0; i < myarray.length; i++) {
3     // do something with myarray[i]
4 }
```

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.

`HTMLCollection`s are objects returned by DOM methods such as:

- `document.getElementsByName()`
- `document.getElementsByClassName()`
- `document.getElementsByTagName()`

There are also a number of other `HTMLCollections`, which were introduced before the DOM standard and are still in use today. There include (among others):

- `document.images`: All IMG elements on the page
- `document.links`: All A elements
- `document.forms`: All forms
- `document.forms[0].elements`: All fields in the first form on the page

The trouble with collections is that they are live queries against the underlying document (the HTML page). This means that every time you access any collection's `length`, you're querying the live DOM, and DOM operations are expensive in general.

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:

```
1 | for (var i = 0, max = myarray.length; i < max; i++) {  
2 |     // do something with myarray[i]  
3 | }
```

This way you retrieve the value of `length` only once and use it during the whole loop.

Caching the length when iterating over `HTMLCollections` is faster across all browsers—anywhere between two times faster (Safari 3) and 190 times (IE7).

Note that when you explicitly intend to modify the collection in the loop (for example, by adding more DOM elements), you'd probably like the length to be updated and not constant.

Following the single var pattern, you can also take the var out of the loop and make the loop like:

```
1 | function looper() {
```

```

2 |     var i = 0,
3 |         max,
4 |         myarray = [];
5 |     // ...
6 |     for (i = 0, max = myarray.length; i < max; i++) {
7 |         // do something with myarray[i]
8 |     }
9 | }

```

This pattern has the benefit of consistency because you stick to the single var pattern. A drawback is that it makes it a little harder to copy and paste whole loops while refactoring code. For example, if you're copying the loop from one function to another, you have to make sure you also carry over `i` and `max` into the new function (and probably delete them from the original function if they are no longer needed there).

One last tweak to the loop would be to substitute `i++` with either one of these expressions:

```

1 | i=i+ 1
2 | i += 1

```

JSLint prompts you to do it; the reason being that `++` and `--` promote “excessive trickiness.” If you disagree with this, you can set the JSLint option `plusplus` to `false`. (It's true by default.)

Two variations of the for pattern introduce some micro-optimizations because they:

- Use one less variable (no `max`)
- Count down to `0`, which is usually faster because it's more efficient to compare to 0 than to the length of the array or to anything other than `0`

The first modified pattern is:

```

1 | var i, myarray = [];
2 | for (i = myarray.length; i--;) {
3 |     // do something with myarray[i]
4 | }

```

And the second uses a `while` loop:

```

1 | var myarray = [],

```

```
2     i = myarray.length;
3   while (i--) {
4     // do something with myarray[i]
5   }
```

These are micro-optimizations and will only be noticed in performance-critical operations. Additionally, JSLint will complain about the use of `i--`.

for-in Loops

`for-in` loops should be used to iterate over nonarray 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.

It's important to use the method `hasOwnProperty()` when iterating over object properties to filter out properties that come down the prototype chain.

Consider the following example:

```
01  // the object
02  var man = {
03    hands: 2,
04    legs: 2,
05    heads: 1
06  };
07
08  // somewhere else in the code
09  // a method was added to all objects
10  if (typeof Object.prototype.clone === "undefined") {
11    Object.prototype.clone = function () {};
12  }
```

In this example we have a simple object called `man` defined with an object literal. Somewhere

before or after `man` was defined, the `Object` prototype was augmented with a useful method called `clone()`. The prototype chain is live, which means all objects automatically get access to the new method. To avoid having the `clone()` method show up when enumerating `man`, you need to call `hasOwnProperty()` to filter out the prototype properties. Failing to do the filtering can result in the function `clone()` showing up, which is undesired behavior in mostly all scenarios:

```
01 // 1.
02 // for-in loop
03 for (var i in man) {
04     if (man.hasOwnProperty(i)) { // filter
05         console.log(i, ":", man[i]);
06     }
07 }
08 /* result in the console
09 hands : 2
10 legs : 2
11 heads : 1
12 */
13 // 2.
14 // antipattern:
15 // for-in loop without checking hasOwnProperty()
16 for (var i in man) {
17     console.log(i, ":", man[i]);
18 }
19 /*
20 result in the console
21 hands : 2
22 legs : 2
23 heads : 1
24 clone: function()
25 */
```

Another pattern for using `hasOwnProperty()` is to call that method off of the `Object.prototype`, like so:

```
1 for (var i in man) {
2     if (Object.prototype.hasOwnProperty.call(man, i)) { // filter
3         console.log(i, ":", man[i]);
4     }
5 }
```

The benefit is that you can avoid naming collisions in case the `man` object has redefined `hasOwnProperty`. Also to avoid the long property lookups all the way to `Object`, you can use a local variable to “cache” it:

```

1  var i, hasOwn = Object.prototype.hasOwnProperty;
2  for (i in man) {
3      if (hasOwn.call(man, i)) { // filter
4          console.log(i, ":", man[i]);
5      }
6  }

```

Strictly speaking, not using `hasOwnProperty()` is not an error.

Depending on the task and the confidence you have in the code, you may skip it and slightly speed up the loops. But when you're not sure about the contents of the object (and its prototype chain), you're safer just adding the

`hasOwnProperty()` check.

A formatting variation (which doesn't pass JSLint) skips a curly brace and puts the if on the same line. The benefit is that the loop statement reads more like a complete thought ("for each element that has an own property `x`, do something with `x`"). Also there's less indentation before you get to the main purpose of the loop:

```

1  // Warning: doesn't pass JSLint
2  var i, hasOwn = Object.prototype.hasOwnProperty;
3  for (i in man) if (hasOwn.call(man, i)) { // filter
4      console.log(i, ":", man[i]);
5  }

```

(Not) Augmenting Built-in Prototypes

Augmenting the prototype property of constructor functions is a powerful way to add functionality, but it can be too powerful sometimes.

It's tempting to augment prototypes of built-in constructors such as `Object()`, `Array()`, or `Function()`, but it can seriously hurt maintainability, because it will make your code less predictable. Other developers using your code will probably expect the built-in JavaScript methods to work consistently and will not expect your additions.

Additionally, properties you add to the prototype may show up in loops that don't use

`hasOwnProperty()`, so they can create confusion.

Therefore it's best if you don't augment built-in prototypes. You can make an exception of the rule only when all these conditions are met:

- It's expected that future ECMAScript versions or JavaScript implementations will implement this functionality as a built-in method consistently. For example, you can add methods described in ECMAScript 5 while waiting for the browsers to catch up. In this case you're just defining the useful methods ahead of time.
- You check if your custom property or method doesn't exist already—maybe already implemented somewhere else in the code or already part of the JavaScript engine of one of the browsers you support.
- You clearly document and communicate the change with the team.

If these three conditions are met, you can proceed with the custom addition to the prototype, following this pattern:

```
1  if (typeof Object.prototype.myMethod !== "function") {
2      Object.prototype.myMethod = function () {
3          // implementation...
4      };
5  }
```

switch Pattern

You can improve the readability and robustness of your `switch` statements by following this pattern:

```
01  var inspect_me = 0,
02      result = '';
03  switch (inspect_me) {
04      case 0:
05          result = "zero";
06          break;
07      case 1:
08          result = "one";
```

```
09     break;
10 default:
11     result = "unknown";
12 }
```

The style conventions followed in this simple example are:

- Aligning each `case` with `switch` (an exception to the curly braces indentation rule).
- Indenting the code within each case.
- Ending each `case` with a clear `break;`.
- Avoiding fall-throughs (when you omit the break intentionally). If you're absolutely convinced that a fall-through is the best approach, make sure you document such cases, because they might look like errors to the readers of your code.
- Ending the `switch` with a `default:` to make sure there's always a sane result even if none of the cases matched.

Avoiding Implied Typcasting

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 typcasting, always use the `===` and `!==` operators that check both the values and the type of the expressions you compare:

```
1  var zero = 0;
2  if (zero === false) {
3      // not executing because zero is 0, not false
4  }
5
6  // antipattern
7  if (zero == false) {
8      // this block is executed...
9  }
```

There's another school of thought that subscribes to the opinion that it's redundant to use `===` when `==` is sufficient. For example, when you use `typeof` you know it returns a string, so there's no reason to use strict equality. However, JSLint requires strict equality; it does make the code look

consistent and reduces the mental effort when reading code. (“Is this `==` intentional or an omission?”)

Avoiding eval()

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:

```
1 // antipattern
2 var property = "name";
3 alert(eval("obj." + property));
4
5 // preferred
6 var property = "name";
7 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. Behind the scenes, JavaScript still has to evaluate and execute the string you pass as programming code:

```
1 // antipatterns
2 setTimeout("myFunc()", 1000);
3 setTimeout("myFunc(1, 2, 3)", 1000);
4
5 // preferred
6 setTimeout(myFunc, 1000);
```

```

7 |   setTimeout(function () {
8 |       myFunc(1, 2, 3);
9 |   }, 1000);

```

Using the new `Function()` constructor is similar to `eval()` and should be approached with care. It could be a powerful construct but is often misused. If you absolutely must use `eval()`, you can consider using new `Function()` instead. There is a small potential benefit because the code evaluated in new `Function()` will be running in a local function scope, so any variables defined with `var` in the code being evaluated will not become globals automatically. Another way to prevent automatic globals is to wrap the `eval()` call into an immediate function.

Consider the following example. Here only `un` remains as a global variable polluting the namespace:

```

01 | console.log(typeof un); // "undefined"
02 | console.log(typeof deux); // "undefined"
03 | console.log(typeof trois); // "undefined"
04 |
05 | var jsstring = "var un = 1; console.log(un);";
06 | eval(jsstring); // logs "1"
07 |
08 | jsstring = "var deux = 2; console.log(deux);";
09 | new Function(jsstring)(); // logs "2"
10 |
11 | jsstring = "var trois = 3; console.log(trois);";
12 | (function () {
13 |     eval(jsstring);
14 | })(); // logs "3"
15 |
16 | console.log(typeof un); // number
17 | console.log(typeof deux); // undefined
18 | console.log(typeof trois); // undefined

```

Another difference between `eval()` and the Function constructor is that `eval()` can interfere with the scope chain whereas `Function` is much more sandboxed. No matter where you execute `Function`, it sees only the global scope. So it can do less local variable pollution. In the following example, `eval()` can access and modify a variable in its outer scope, whereas Function cannot (also note that using Function or new Function is identical):

```

01 | (function () {
02 |     var local = 1;
03 |     eval("local = 3; console.log(local)"); // logs 3

```

```

04     console.log(local); // logs 3
05 } ());
06
07 (function () {
08     var local = 1;
09     Function("console.log(typeof local);") (); // logs undefined
10 } ());

```

Number Conversions with 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) in ECMAScript 3; however, this has changed in ES5. To avoid inconsistency and unexpected results, always specify the radix parameter:

```

1  var month = "06",
2      year = "09";
3  month = parseInt(month, 10);
4  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:

```

1  +"08" // result is 8
2  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`.

Coding Conventions

It's important to establish and follow coding conventions—they make your code consistent, *predictable*, and much easier to read and understand. A new developer joining the team can read through the conventions and be productive much sooner, understanding the code written by any other team member.

Many flamewars have been fought in meetings and on mailing lists over specific aspects of certain coding conventions (for example, the code indentation—tabs or spaces?). So if you're the one suggesting the adoption of conventions in your organization, be prepared to face resistance and hear different but equally strong opinions. Remember that it's much more important to establish and consistently follow a convention, any convention, than what the exact details of that convention will be.

Indentation

Code without indentation is impossible to read. The only thing worse is code with inconsistent indentation, because it looks like it's following a convention, but it may have confusing surprises along the way. It's important to standardize the use of indentation.

Some developers prefer indentation with tabs, because anyone can tweak their editor to display the tabs with the individually preferred number of spaces. Some prefer spaces—usually four. It doesn't matter as long as everyone in the team follows the same convention. This book, for example, uses four-space indentation, which is also the default in JSLint.

And what should you indent? The rule is simple—anything within curly braces. This means the bodies of functions, loops (`do`, `while`, `for`, `for-in`), `ifs`, `switches`, and `object` properties in the `object` literal notation. The following code shows some examples of using indentation:

```
01  function outer(a, b) {
02      var c = 1,
03          d = 2,
04          inner;
05      if (a > b) {
06          inner = function () {
07              return {
08                  r: c - d
09              };
06          };
07      }
```

```

10     };
11     } else {
12         inner = function () {
13             return {
14                 r: c + d
15             };
16         };
17     }
18     return inner;
19 }

```

Curly Braces

Curly braces should always be used, even in cases when they are optional. Technically, if you have only one statement in an `if` or a `for`, curly braces are not required, but you should always use them anyway. It makes the code more consistent and easier to update.

Imagine you have a for loop with one statement only. You could omit the braces and there will be no syntax error:

```

1 // bad practice
2 for (var i = 0; i < 10; i += 1)
3     alert(i);

```

But what if, later on, you add another line in the body of the loop?

```

1 // bad practice
2 for (var i = 0; i < 10; i += 1)
3     alert(i);
4     alert(i + " is " + (i % 2 ? "odd" : "even"));

```

The second alert is outside the loop although the indentation may trick you. The best thing to do in the long run is to always use the braces, even for one-line blocks:

```

1 // better
2 for (var i = 0; i < 10; i += 1) {
3     alert(i);
4 }

```

Similarly for if conditions:

```
01 // bad
02 if (true)
03     alert(1);
04 else
05     alert(2);
06
07 // better
08 if (true) {
09     alert(1);
10 } else {
11     alert(2);
12 }
```

Opening Brace Location

Developers also tend to have preferences about where the opening curly brace should be—on the same line or on the following line?

```
1 if (true) {
2     alert("It's TRUE!");
3 }
```

OR:

```
1 if (true)
2 {
3     alert("It's TRUE!");
4 }
```

In this specific example, it's a matter of preference, but there are cases in which the program might behave differently depending on where the brace is. This is because of the `semicolon insertion mechanism`—JavaScript is not picky when you choose not to end your lines properly with a semicolon and adds it for you. This behavior can cause troubles when a function returns an object literal and the opening brace is on the next line:

```
1 // warning: unexpected return value
2 function func() {
```

```

3     return
4     // unreachable code follows
5     {
6         name : "Batman"
7     }
8 }

```

If you expect this function to return an object with a `name` property, you'll be surprised. Because of the implied semicolons, the function returns `undefined`. The preceding code is equivalent to this one:

```

1 // warning: unexpected return value
2 function func() {
3     return undefined;
4     // unreachable code follows
5     {
6         name : "Batman"
7     }
8 }

```

In conclusion, always use curly braces and always put the opening one on the same line as the previous statement:

```

1 function func() {
2     return {
3         name : "Batman"
4     };
5 }

```

A note on semicolons: Just like with the curly braces, you should always use semicolons, even when they are implied by the JavaScript parsers. This not only promotes discipline and a more rigorous approach to the code but also helps resolve ambiguities, as the previous example showed.

White Space

The use of white space can also contribute to improved readability and consistency of the code. In written English sentences you use intervals after commas and periods. In JavaScript you follow the

same logic and add intervals after list-like expressions (equivalent to commas) and end-of-statements (equivalent to completing a “thought”).

Good places to use a white space include:

- After the semicolons that separate the parts of a for loop: for example, `for (var i = 0; i < 10; i += 1) {...}`
- Initializing multiple variables (i and max) in a `for` loop: `for (var i = 0, max = 10; i < max; i += 1) {...}`
- After the commas that delimit array items: `var a = [1, 2, 3];`
- After commas in object properties and after colons that divide property names and their values: `var o = {a: 1, b: 2};`
- Delimiting function arguments: `myFunc(a, b, c)`
- Before the curly braces in function declarations: `function myFunc() {}`
- After `function` in anonymous function expressions: `var myFunc = function () {};`

Another good use for white space is to separate all operators and their operands with spaces, which basically means use a space before and after `+, -, *, =, <, >, <=, >=, ===, !==, &&, ||, +=,` and so on:

```
01 // generous and consistent spacing
02 // makes the code easier to read
03 // allowing it to "breathe"
04 var d = 0,
05     a = b + 1;
06 if (a && b && c) {
07     d = a % c;
08     a += d;
09 }
10
11 // antipattern
12 // missing or inconsistent spaces
13 // make the code confusing
14 var d = 0,
15     a = b + 1;
16 if (a && b && c) {
17     d = a % c;
18     a += d;
19 }
```

And a final note about white space—curly braces spacing. It’s good to use a space:

- Before opening curly braces (`{`) in functions, `if-else` cases, loops, and object literals
- Between the closing curly brace (`}`) and `else` or `while`

A case against liberal use of white space might be that it could increase the file size, but minification takes care of this issue.

An often-overlooked aspect of code readability is the use of vertical white space. You can use blank lines to separate units of code, just as paragraphs are used in literature to separate ideas.

Naming Conventions

Another way to make your code more predictable and maintainable is to adopt naming conventions. That means choosing names for your variables and functions in a consistent manner.

Below are some naming convention suggestions that you can adopt as-is or tweak to your liking. Again, having a convention and following it consistently is much more important than what that convention actually is.

Capitalizing Constructors

JavaScript doesn't have classes but has constructor functions invoked with `new`:

```
1 | var adam = new Person();
```

Because constructors are still just functions, it helps if you can tell, just by looking at a function name, whether it was supposed to behave as a constructor or as a normal function.

Naming constructors with a capital first letter provides that hint. Using lowercase for functions and methods indicates that they are not supposed to be called with `new`:

```
1 function MyConstructor() {...}
2 function myFunction() {...}
```

Separating Words

When you have multiple words in a variable or a function name, it's a good idea to follow a convention as to how the words will be separated. A common convention is to use the so-called *camel case*. Following the camel case convention, you type the words in lowercase, only capitalizing the first letter in each word.

For your constructors, you can use *upper camel case*, as in `MyConstructor()`, and for function and method names, you can use *lower camel case*, as in `myFunction()`, `calculateArea()` and `getFirstName()`.

And what about variables that are not functions? Developers commonly use lower camel case for variable names, but another good idea is to use all lowercase words delimited by an underscore: for example, `first_name`, `favorite_bands`, and `old_company_name`. This notation helps you visually distinguish between functions and all other identifiers—primitives and objects.

ECMAScript uses camel case for both methods and properties, although the multiword property names are rare (`lastIndex` and `ignoreCase` properties of regular expression objects).

Other Naming Patterns

Sometimes developers use a naming convention to make up or substitute language features.

For example, there is no way to define constants in JavaScript (although there are some built-in such as `Number.MAX_VALUE`), so developers have adopted the convention of using all-caps for naming variables that shouldn't change values during the life of the program, like:

```
1 // precious constants, please don't touch
2 var PI = 3.14,
3
```

```
MAX_WIDTH = 800;
```

There's another convention that competes for the use of all caps: using capital letters for names of global variables. Naming globals with all caps can reinforce the practice of minimizing their number and can make them easily distinguishable.

Another case of using a convention to mimic functionality is the private members convention. Although you can implement true privacy in JavaScript, sometimes developers find it easier to just use an underscore prefix to denote a private method or property. Consider the following example:

```
01 | var person = {
02 |   getName: function () {
03 |     return this._getFirst() + ' ' + this._getLast();
04 |   },
05 |
06 |   _getFirst: function () {
07 |     // ...
08 |   },
09 |   _getLast: function () {
10 |     // ...
11 |   }
12 | };
```

In this example `getName()` is meant to be a public method, part of the stable API, whereas `_getFirst()` and `_getLast()` are meant to be private. They are still normal public methods, but using the underscore prefix warns the users of the person object that these methods are not guaranteed to work in the next release and shouldn't be used directly. Note that JSLint will complain about the underscore prefixes, unless you set the option `nomen: false`.

Following are some varieties to the `_private` convention:

- Using a trailing underscore to mean private, as in `name_` and `getElements_()`
-
- Using one underscore prefix for `_protected` properties and two for `__private` properties
- In Firefox some internal properties not technically part of the language are available, and they are named with a two underscores prefix and a two underscore suffix, such as `__proto__` and `__parent__`

Writing Comments

You have to comment your code, even if it's unlikely that someone other than you will ever touch it. Often when you're deep into a problem you think it's obvious what the code does, but when you come back to the code after a week, you have a hard time remembering how it worked exactly.

You shouldn't go overboard commenting the obvious: every single variable or every single line. But you usually need to document all functions, their arguments and return values, and also any interesting or unusual algorithm or technique. Think of the comments as hints to the future readers of the code; the readers need to understand what your code does without reading much more than just the comments and the function and property names. When you have, for example, five or six lines of code performing a specific task, the reader can skip the code details if you provide a one-line description describing the *purpose* of the code and why it's there. There's no hard and fast rule or ratio of comments-to-code; some pieces of code (think regular expressions) may actually require more comments than code.

The most important habit, yet hardest to follow, is to keep the comments up to date, because outdated comments can mislead and be much worse than no comments at all.

About the Author

Stoyan Stefanov is a Yahoo! web developer and an [author](#), contributor, and tech reviewer of various O'Reilly books. He speaks regularly about web development topics at conferences and on his blog at www.phpied.com. Stoyan is the creator of the smush.it image optimization tool, YUI contributor and architect of Yahoo's performance optimization tool YSlow 2.0.

Buy the Book



This article is an excerpt from "[JavaScript Patterns](#)," by [O'Reilly Media](#).

Advertisement

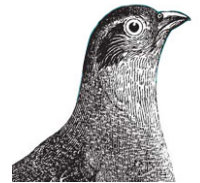
Difficulty:

Intermediate

Categories:

Web Development

JavaScript



Translations Available:

Tuts+ tutorials are translated by our community members. If you'd like to translate this post into another language, [let us know!](#)

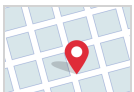
About Stoyan Stefanov



Stoyan Stefanov is a Yahoo! web developer and an [author](#), contributor, and tech reviewer of various O'Reilly

[+ Expand Bio](#)

Related Courses



Custom Interactive Maps With the Google Maps API

David East



Node for the Front-End Developer

Jason Rhodes



Build a Multi-Player Card Game With Meteor

Andrew Burgess

Advertisement

Related Tutorials



The Beginner's Guide to Type Coercion: A Practical Example

[Code](#)



Bubble.js: A 1.6K Solution to a Common Problem

[Code](#)



Deb.js: the Tiniest Debugger in the World

[Code](#)



JavaScript Animation That Works (Part 3 of 4)

[Code](#)

Jobs



Ruby Software Developer
at Red Nova Labs in Kansas City, KS, USA



Junior/Graduate Web Developer
in Bournemouth, Bournemouth, UK



Junior/Mid Level PHP Developer
in Los Angeles, CA, USA



Market Item



SocialKit - Social Networking Platform

Category: social-networking

\$40.00

70 Comments

Nettuts+

 Login ▾

Sort by Best ▾

Share  Favorite ★



Join the discussion...



Rob Anderson • 3 years ago

Great article and very helpful - thanks for putting this together!

One small thing I noticed in your example about whitespace:

```
// antipattern
// missing or inconsistent spaces
// make the code confusing
var d = 0,
a = b + 1;
if (a && b && c) {
d = a % c;
a += d;
}
```

This looks the same as the example above it, e.g.

```
// generous and consistent spacing
// makes the code easier to read
// allowing it to "breathe"
var d = 0,
a = b + 1;
if (a && b && c) {
d = a % c;
a += d;
}
```


10 ^ | v • Reply • Share ›



Anthony Bailey • 4 years ago

"Maintainable code means code that: ..."

...has tests.

4 ^ | v • Reply • Share ›



Darren L • 3 years ago

I am going to have to pick this book up. This is an excellent article and just from reading this the book must be awesome.

1 ^ | v • Reply • Share ›



Jerold Anderson • 4 years ago

Good grief! I'm so sick of hearing people say "eval() is evil." It is not evil and there are VERY good uses for eval. The article is well written and has many good points. Please refrain from repeating Douglas Crockford regularly. Doug is a smart guy but this comment about eval being evil has caused many people that write JavaScript to do stupid things in their scripting. I can give many examples where eval is absolutely necessary and not evil, so please stop saying that eval is evil; it is not.

1 ^ | v • Reply • Share ›



Jared ➔ **Jerold Anderson** • 3 years ago

If you think you need eval, you are doing it wrong.

1 ^ | v • Reply • Share ›



Alberto Cole ➔ **Jerold Anderson** • 4 years ago

Give us a couple examples pls :)

1 ^ | v • Reply • Share ›



JSGuy ➔ **Jerold Anderson** • 8 months ago

<http://stackoverflow.com/quest...>

^ | v • Reply • Share ›



Tom • 4 years ago

Very, very nice post!

But I can see no difference in the good and bad example for whitespace!?

Am I missing sth.? :D

1 ^ | v • Reply • Share ›



Andrés Mejía • 4 years ago

I don't agree with several things he says. For example:

"it's good to use a space:

- * Before opening curly braces ({} in functions, if-else cases, loops, and object literals
- * Between the closing curly brace (}) and else or while"

Why is this "good"? Just because this is this guy's personal preference doesn't make it "good". Use spaces if you want, or don't. If this guy feels his eyes itch when he reads "function(a,b,c)" instead of "function(a, b, c)" then he's really a deplorable coder.

Don't like the tone of the book. It's like "do this and don't do that. This is good, that is bad". It almost sounds like the Pope: "Don't use condoms. They are bad!".

1 ^ | v • Reply • Share ›



Ryan Sharp ➔ Andrés Mejía • 3 years ago

It's nothing to do with preferences. Crockford happens to agree with him and as he say himself, strict style is very important because so little about the language itself is strict. There are many reason for these common style choices, most of which you are probably ignorant to. I suggest you read more and posture less.

2 ^ | v • Reply • Share ›



Luis ➔ Andrés Mejía • 3 years ago

Well, after all, we're talking about languages. It doesn't matter if it's a programming language or a natural one. Imagine a book where the author simply ignores the use of spaces after/before parenthesis and brackets or after the commas in a list. I guess you would not like it and probably won't read it. So it is with your code.

1 ^ | v • Reply • Share ›



Boner Jam ➔ Andrés Mejía • 4 years ago

"...awesome." *

^ | v • Reply • Share ›



Konrad Dzwinel • 8 months ago

Turns out that "for Loops" advice is no longer true: <http://jsperf.com/fastest-arra...>

^ | v • Reply • Share ›



Julian • 4 years ago

First!

^ | v • Reply • Share ›



Montana Flynn ➔ Julian • 4 years ago

After writing such an in depth tutorial that must have taken hours, I am sure Stoyan would like if the comments had at least a bit of substance

3 ^ | v • Reply • Share ›



Andrés Mejía ➔ Montana Flynn • 4 years ago

You're just envious you were 13 minutes late!

^ | v • Reply • Share ›



哈哈胡子 • 8 months ago

Very helpful for newbies. I wish I had read this earlier...

^ | v • Reply • Share ›



Roy Truelove • 8 months ago

I find that switching to coffeescript alleviates a good number of these issues out of the box.

^ | v • Reply • Share ›



punund • a year ago

"return" example has nothing to do with semicolon insertion rules. JS requires that returned value starts on the same line with "return" operator.

^ | v • Reply • Share ›



jatazoulja • 2 years ago

I wish I had this as autoformat for IDE like eclipse (aptana and nb has no problem with this stuff...)

^ | v • Reply • Share ›



Bosn Ma • 2 years ago

Very awesome JavaScript statistics!

^ | v • Reply • Share ›



Ajain • 2 years ago

Thanks :)

^ | v • Reply • Share ›



jim • 2 years ago

When cases get complex, enclose them in their own blocks.

When refactoring, this technique helps take advantage of modern code editors that support select-to-matching-brace.

```
switch (inspect_me) {  
  case 0: {  
    result = "zero";  
    break;  
  }  
  case 1: {  
    result = "one";  
    break;  
  }  
  default: {
```

```
result = "unknown";  
}  
}
```

^ | v • Reply • Share ›



Tony • 2 years ago

Wooah Stoyan , thanks so much!

^ | v • Reply • Share ›



DH • 3 years ago

Read the book. Great stuff.

For anyone who wants to do something smarter than \$()

^ | v • Reply • Share ›



Raj Sandhu • 3 years ago

I love this article and it is so well written that gives complete information about Coding Best Practice. Thanks :)

^ | v • Reply • Share ›



basic • 3 years ago

what about closures, object literals and stuff?
this is too basic.

^ | v • Reply • Share ›



anon ➔ **basic** • 9 months ago

It's in the actual book

1 ^ | v • Reply • Share ›



Ijaas Yunoos • 3 years ago

Quick question in regard to global variables.

```
(function($){  
  
var windowWidth = $(window).width();  
  
$(window).resize(function(){  
windowWidth = $(window).width();  
})  
  
})(jQuery);
```

does the second windowWidth create a global variable or just amend the variable created earlier?

If it does create a global variable what is the best way to avoid this.

^ | v • Reply • Share ›



Jared → Ijaas Yunoos • 3 years ago

It will change the state of windowWidth you had declared with "var" earlier in this closure. It does not create a global.

^ | v • Reply • Share ›



Ijaas → Jared • 3 years ago

awesome, thanks.

^ | v • Reply • Share ›



Jared • 3 years ago

In example of an anti-pattern is jQuery's global \$. This breaks MooTools. I see a lot of jQuery libraries depend on the global \$. It is disgusting.

^ | v • Reply • Share ›



Abhishek Dilliwal • 3 years ago

I use to hear about JSLint and the benefits but I had never tried. I will try to follow up with these guidelines. the most impressive one is not to use eval(), In my most recent project I was about to use this... I thought of scenario where the ajax sends some JS code to execute... and then use eval for the same... but now i need to change my mind... thanks!

^ | v • Reply • Share ›



Kevin deLeon • 4 years ago

Great article, and definitely prompted me to grab the book. JavaScript can be such a pain in the ass, and become unwieldy very quickly. It's great to read an article (and hopefully the book too) that goes into some depth when explaining and introducing JS best practices.

^ | v • Reply • Share ›



Brij • 4 years ago

Great Article!!

I couldn't stop to write comment.

Really, The essential information for JS beginners.

^ | v • Reply • Share ›



Luiz Lopes • 4 years ago

Awesome article, really useful. I am going to implement some of these patterns right away, and possibly add them to the company's coding convention.

Cheers!

^ | v • Reply • Share ›



Kotes • 4 years ago



Very nice article about JS... Thanks Stoyan Stefanov

^ | v • Reply • Share ›



webincrediblez.com • 4 years ago

Good source of information about Javascript....

^ | v • Reply • Share ›



Richard • 4 years ago

In other words JavaScript is so fragile that we're exposed ad nauseum to books and articles on how not to break it. But you'll notice I clicked on the link. I'll come right out and say it, JavaScript sucks - but we still have to deal with it.

^ | v • Reply • Share ›



pratheep → Richard • 8 months ago

Duh

^ | v • Reply • Share ›



Elvin • 4 years ago

If you're still on the fence about buying this brilliant book, I found an in-depth review here:

<http://blog.krawaller.se/book-...>

^ | v • Reply • Share ›



Henrique • 4 years ago

Those articles always remember me how bad and inconsistent Javascript is. Unfortunately, it's what we have available to work with.

Hopefully, there's people like the author who have been burned already and know the path to the gold pot.

Good article.

^ | v • Reply • Share ›



Zecc • 4 years ago

There was nothing new here for me.

Now there's a boost of confidence for me! :)

Anyway, a typo jumped at me (I'm picky): "constrictors" instead of "constructors"

^ | v • Reply • Share ›



Jaspal Singh • 4 years ago

Very good article on Javascript.

Thanks for sharing.

^ | v • Reply • Share ›



noodle • 4 years ago

Appears to be lifted directly from the appendices of Doug Crockford's "Javascript: The Good Parts" - only with a bit more explanation.

^ | v • Reply • Share ›



fritz from london • 4 years ago

Good reference, agree with most of it (I like white spacing like(this) but that's just a matter of preference)

Good luck with the book's sales :-)

^ | v • Reply • Share ›



Ryan Sharp → fritz from london • 3 years ago

No, no it isn't. It's a matter of readability and clarity in deeply nested blocks of code. It's bad practice to use white-space inside parentheses like that in any C-family language. Please forget your silly "preference" and go learn why.

1 ^ | v • Reply • Share ›



Sachin • 4 years ago

I have just recently learned JavaScript and these points are very helpful....especially i was looking more on Globals...excellent....thanks a lot.

^ | v • Reply • Share ›



Kartlos Tchavelachvili • 4 years ago

Thanks! very well explained!

^ | v • Reply • Share ›



Daniel • 4 years ago

It seems like basic stuff to me but a pretty good advice to some unknown out there i guess. In some points, JS is not worth the oop... e.g. the virtual visibility

^ | v • Reply • Share ›



Leroy • 4 years ago

Hey Stoyan, this is an amazing tutorial that has something for all levels of javascript programmers. Thank you for sharing your knowledge with us.

Got to get 2 javascript books now. One by a certain Crockford, and then this :)

Thanks again.

^ | v • Reply • Share ›



Zlatan Halilovic • 4 years ago

I'm definitely ordering the book from amazon in a couple of days :)

^ | v • Reply • Share ›



Q_theDreadlockedNinja • 4 years ago

Or you could simply read The Good Parts by Douglas Crawford.

^ | v • Reply • Share ›



helium • 4 years ago

That was fantastically detailed post, that was extremely informative. thanks alot!!

^ | v • Reply • Share ›



Elaine • 4 years ago

I completely agree with everything you said here. Great advice :)

^ | v • Reply • Share ›



usingjquery • 4 years ago

Great read, will definitely be getting the book! Thanks for sharing

^ | v • Reply • Share ›



vapour • 4 years ago

very good! thanks

^ | v • Reply • Share ›



Michael • 4 years ago

Thanks for the excerpt! I lot of really good stuff in there. I have to admit that since I've started using jQuery, my use of "normal" Javascript has decreased. I lot of good reminders and new stuff in here. Just might have to get the book!

^ | v • Reply • Share ›



Alex Bars • 4 years ago

Good book, but seems that it partially can be said that it's book is a "The Good Parts" exhaustively explained, but has good advices, Thanks!!

PS: Doug' Crockford's The Good Parts book!!

^ | v • Reply • Share ›



Daniel • 4 years ago

Absolutely AWESOME. Even for being an excerpt from a book, this is hands down one the best articles Ive ever read on js patterns. The selection is clearly indicative of the quality of the rest of the text. Im buying the book right now.

Thank you!

^ | v • Reply • Share ›



Roger Padilla • 4 years ago

This book seems great, thanks!

^ | v • Reply • Share ›

^ | v • Reply • Share ›



esranull • 4 years ago

nice post thanks a lot

^ | v • Reply • Share ›



Crysfel • 4 years ago

This is a really good Post specially for those who are starting in the JavaScript language, I hope people realize how worth is this information :)

Best regards

^ | v • Reply • Share ›



paperreduction • 4 years ago

fantastic read. we'll be adopting some of this into our own standards, that's for sure. (can't wait to get my copy of the book).

^ | v • Reply • Share ›



Jason Crane • 4 years ago

Great article, looking forward to the book arriving in the mail.

You may want to take a look at the example code in the White Space section; I think both the suggested example and the antipattern are the same, unless I'm missing something.

^ | v • Reply • Share ›



Frank • 4 years ago

Awesome stuff. Good to see a definitive answer on things like bracket placement and whitespace. I'll be buying the new book!

^ | v • Reply • Share ›



Ben • 4 years ago

Thanks, excellent article :)

However there is a minor oversight in the "White Space" section. The two code examples are identical (the "antipattern" snippet is identical to the other snippet).

^ | v • Reply • Share ›



Stoyan ➔ Ben • 4 years ago

Thanks for noticing, Ben. I suspect Jeffrey has a way to reformat code before it goes to the web site.

In my original example, the antipattern is simply formatted horrendously :)

And thanks everyone for the kind words!

And thanks everyone for the kind words.

^ | v • Reply • Share ›



sherif • 4 years ago

Amazing .. Really brilliant !!

^ | v • Reply • Share ›



Dan • 4 years ago

I'm glad to see that this book has been future proofed to cover ES5 (just got it yesterday)!

Advertisement

Teaching skills to millions worldwide.

 [Tutorials](#)

 [Courses](#)

 [eBooks](#)


 [Jobs](#)

 [Blog](#)

Follow Us

 [Subscribe to Blog](#)

 [Follow us on Twitter](#)

 [Be a fan on Facebook](#)

 [Circle us on Google+](#)

 [Tutorials](#)

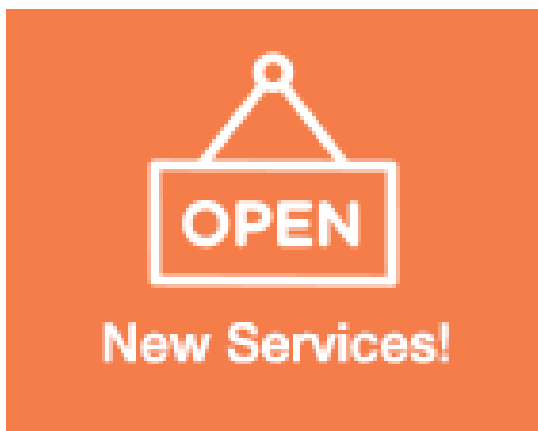
 [Courses](#)

 [eBooks](#)



Add more features to your website such as user profiles, payment gateways, image galleries and more.

Browse WordPress Plugins



Microlancer is now Envato Studio! Custom digital services like logo design, WordPress installaton, video production and more.

Check out Envato Studio

[Pricing](#)

[FAQ](#)

[Support](#)

[Write For Us](#)

[Advertise](#)

[Privacy Policy](#)

[Terms of Use](#)

[Tuts+ Community Rules](#)

© 2014 Envato Pty Ltd.

