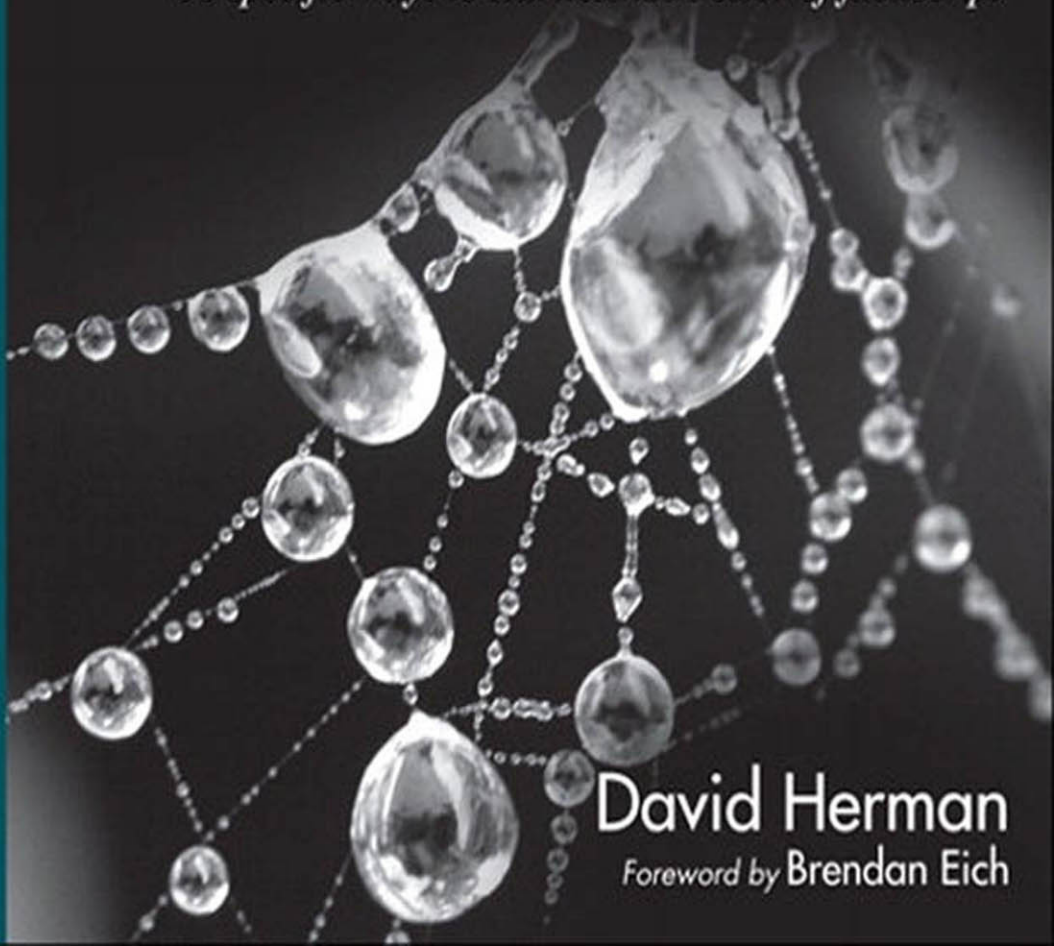
Effective SOFTWARE DEVELOPMENT SERIES

Scott Meyers, Consulting Editor

Effective JAVASCRIPT

68 Specific Ways to Harness the Power of JavaScript



David Herman

Foreword by Brendan Eich

Praise for *Effective JavaScript*

"Living up to the expectation of an Effective Software Development Series programming book, *Effective JavaScript* by Dave Herman is a must-read for anyone who wants to do serious JavaScript programming. The book provides detailed explanations of the inner workings of JavaScript, which helps readers take better advantage of the language."

—Erik Arvidsson, senior software engineer

"It's uncommon to have a programming language wonk who can speak in such comfortable and friendly language as David does. His walk through the syntax and semantics of JavaScript is both charming and hugely insightful; reminders of gotchas complement realistic use cases, paced at a comfortable curve. You'll find when you finish the book that you've gained a strong and comprehensive sense of mastery."

—Paul Irish, developer advocate, Google Chrome

"Before reading *Effective JavaScript*, I thought it would be just another book on how to write better JavaScript. But this book delivers that and so much more—it gives you a deep understanding of the language. And this is crucial. Without that understanding you'll know absolutely nothing whatever about the language itself. You'll only know how other programmers write their code.

"Read this book if you want to become a really good JavaScript developer. I, for one, wish I had it when I first started writing JavaScript."

—Anton Kovalyov, developer of JSHint

"If you're looking for a book that gives you formal but highly readable insights into the JavaScript language, look no further. Intermediate JavaScript developers will find a treasure trove of knowledge inside, and even highly skilled JavaScripters are almost guaranteed to learn a thing or ten. For experienced practitioners of other languages looking to dive headfirst into JavaScript, this book is a must-read for quickly getting up to speed. No matter what your background, though, author Dave Herman does a fantastic job of exploring JavaScript—its beautiful parts, its warts, and everything in between."

—Rebecca Murphey, senior JavaScript developer, Bocoup

"*Effective JavaScript* is essential reading for anyone who understands that JavaScript is no mere toy and wants to fully grasp the power it has to offer. Dave Herman brings users a deep, studied, and practical understanding of the language, guiding them through example after example to help them come to the same conclusions he has. This is not a book for those looking for shortcuts; rather, it is hard-won experience distilled into a guided tour. It's one of the few books on JavaScript that I'll recommend without hesitation."

—Alex Russell, TC39 member, software engineer, Google

"Rarely does anyone have the opportunity to study alongside a master in their craft. This book is just that—the JavaScript equivalent of a time-traveling philosopher visiting fifth century BC to study with Plato."

—Rick Waldron, JavaScript evangelist, Bocoup

```
var constructor = function() { return null; };
var f = function f() {
    return constructor();
};
f(); // {} (in ES3 environments)
```

This program looks like it should produce `null`, but it actually produces a new object, because the named function expression inherits `Object.prototype.constructor` (i.e., the `Object` constructor function) in its scope. And just like `with`, the scope is affected by dynamic changes to `Object.prototype`. One part of a program could add or delete properties to `Object.prototype` and variables within named function expressions everywhere would be affected.

Thankfully, ES5 corrected this mistake. But some JavaScript environments continue to use the obsolete object scoping. Worse, some are even less standards-compliant and use objects as scopes *even for anonymous function expressions!* Then, even removing the function expression's name in the preceding example produces an object instead of the expected `null`:

```
var constructor = function() { return null; };
var f = function() {
    return constructor();
};
f(); // {} (in nonconformant environments)
```

The best way to avoid these problems on systems that pollute their function expressions' scopes with objects is to avoid ever adding new properties to `Object.prototype` and avoid using local variables with any of the names of the standard `Object.prototype` properties.

The next bug seen in popular JavaScript engines is hoisting named function expressions as if they were declarations. For example:

```
var f = function g() { return 17; };
g(); // 17 (in nonconformant environments)
```

To be clear, this is *not* standards-compliant behavior. Worse, some JavaScript environments even treat the two functions `f` and `g` as distinct objects, leading to unnecessary memory allocation! A reasonable workaround for this behavior is to create a local variable of the same name as the function expression and assign it to `null`:

```
var f = function g() { return 17; };
var g = null;
```

Redeclaring the variable with `var` ensures that `g` is bound even in those environments that do not erroneously hoist the function

expression, and setting it to `null` ensures that the duplicate function can be garbage-collected.

It would certainly be reasonable to conclude that named function expressions are just too problematic to be worth using. A less austere response would be to use named function expressions during development for debugging, and to run code through a preprocessor to anonymize all function expressions before shipping. But one thing is certain: You should always be clear about what platforms you are shipping on (see Item 1). The worst thing you could do is to litter your code with workarounds that aren't even necessary for the platforms you support.

Things to Remember

- Use named function expressions to improve stack traces in `Error` objects and debuggers.
- Beware of pollution of function expression scope with `Object.prototype` in ES3 and buggy JavaScript environments.
- Beware of hoisting and duplicate allocation of named function expressions in buggy JavaScript environments.
- Consider avoiding named function expressions or removing them before shipping.
- If you are shipping in properly implemented ES5 environments, you've got nothing to worry about.

Item 15: Beware of Unportable Scoping of Block-Local Function Declarations

The saga of context sensitivity continues with nested function declarations. It may surprise you to know that there is no standard way to declare functions inside a local block. Now, it's perfectly legal and customary to nest a function declaration at the top of another function:

```
function f() { return "global"; }

function test(x) {
    function f() { return "local"; }

    var result = [];
    if (x) {
        result.push(f());
    }
```

```
    result.push(f());
    return result;
}

test(true);  // ["local", "local"]
test(false); // ["local"]
```

But it's an entirely different story if we move `f` into a local block:

```
function f() { return "global"; }

function test(x) {
    var result = [];
    if (x) {
        function f() { return "local"; } // block-local

        result.push(f());
    }
    result.push(f());
    return result;
}

test(true);  // ?
test(false); // ?
```

You might expect the first call to test to produce the array `["local", "global"]` and the second to produce `["global"]`, since the inner `f` appears to be local to the `if` block. But recall that JavaScript is not block-scoped, so the inner `f` should be in scope for the whole body of `test`. A reasonable second guess would be `["local", "local"]` and `["local"]`. And in fact, some JavaScript environments behave this way. But not all of them! Others *conditionally* bind the inner `f` at runtime, based on whether its enclosing block is executed. (Not only does this make code harder to understand, but it also leads to slow performance, not unlike `with` statements.)

What does the ECMAScript standard have to say about this state of affairs? Surprisingly, almost nothing. Until ES5, the standard did not even acknowledge the existence of block-local function declarations; function declarations are officially specified to appear only at the outermost level of other functions or of a program. ES5 even recommends turning function declarations in nonstandard contexts into a warning or error, and popular JavaScript implementations report them as an error in strict mode—a strict-mode program with a block-local function declaration will report a syntax error. This helps detect unportable code, and it clears a path for future versions of the

standard to specify more sensible and portable semantics for block-local declarations.

In the meantime, the best way to write portable functions is to avoid ever putting function declarations in local blocks or substatements. If you want to write a nested function declaration, put it at the outermost level of its parent function, as shown in the original version of the code. If, on the other hand, you need to choose between functions conditionally, the best way to do this is with `var` declarations and function expressions:

```
function f() { return "global"; }

function test(x) {
    var g = f, result = [];
    if (x) {
        g = function() { return "local"; }

        result.push(g());
    }
    result.push(g());
    return result;
}
```

This eliminates the mystery of the scoping of the inner variable (renamed here to `g`): It is unconditionally bound as a local variable, and only the assignment is conditional. The result is unambiguous and fully portable.

Things to Remember

- Always keep function declarations at the outermost level of a program or a containing function to avoid unportable behavior.
- Use `var` declarations with conditional assignment instead of conditional function declarations.

Item 16: Avoid Creating Local Variables with `eval`

JavaScript's `eval` function is an incredibly powerful and flexible tool. Powerful tools are easy to abuse, so they're worth understanding. One of the simplest ways to run afoul of `eval` is to allow it to interfere with scope.

Calling `eval` interprets its argument as a JavaScript program, but that program runs in the local scope of the caller. The global variables of the embedded program get created as locals of the calling program:

```
function test(x) {
    eval("var y = x;"); // dynamic binding
    return y;
}
test("hello"); // "hello"
```

This example looks clear, but it behaves subtly differently than the var declaration would behave if it were directly included in the body of test. The var declaration is only executed when the eval function is called. Placing an eval in a conditional context brings its variables into scope only if the conditional is executed:

```
var y = "global";
function test(x) {
    if (x) {
        eval("var y = 'local';"); // dynamic binding
    }
    return y;
}
test(true);  // "local"
test(false); // "global"
```

Basing scoping decisions on the dynamic behavior of a program is almost always a bad idea. The result is that simply understanding which binding a variable refers to requires following the details of how the program executes. This is especially tricky when the source code passed to eval is not even defined locally:

```
var y = "global";
function test(src) {
    eval(src); // may dynamically bind
    return y;
}
test("var y = 'local';"); // "local"
test("var z = 'local';"); // "global"
```

This code is brittle and unsafe: It gives external callers the power to change the internal scoping of the test function. Expecting eval to modify its containing scope is also not safe for compatibility with ES5 strict mode, which runs eval in a nested scope to prevent this kind of pollution. A simple way to ensure that eval does not affect outer scopes is to run it in an explicitly nested scope:

```
var y = "global";
function test(src) {
    (function() { eval(src); })();
    return y;
}
```

```
test("var y = 'local';"); // "global"
test("var z = 'local';"); // "global"
```

Things to Remember

- Avoid creating variables with `eval` that pollute the caller's scope.
- If `eval` code might create global variables, wrap the call in a nested function to prevent scope pollution.

Item 17: Prefer Indirect `eval` to Direct `eval`

The `eval` function has a secret weapon: It's more than just a function.

Most functions have access to the scope where they are defined, and nothing else. But `eval` has access to the full scope *at the point where it's called.* This is such immense power that when compiler writers first tried to optimize JavaScript, they discovered that `eval` made it difficult to make any function calls efficient, since every function call needed to make its scope available at runtime in case the function turned out to be `eval`.

As a compromise, the language standard evolved to distinguish two different ways of calling `eval`. A function call involving the identifier `eval` is considered a "direct" call to `eval`:

```
var x = "global";
function test() {
    var x = "local";
    return eval("x"); // direct eval
}
test(); // "local"
```

In this case, compilers are required to ensure that the executed program has complete access to the local scope of the caller. The other kind of call to `eval` is considered "indirect," and evaluates its argument in global scope. For example, binding the `eval` function to a different variable name and calling it through the alternate name causes the code to lose access to any local scope:

```
var x = "global";
function test() {
    var x = "local";
    var f = eval;
    return f("x"); // indirect eval
}
test(); // "global"
```