You Don't Know JS Yet: Types & Grammar - 2nd Edition

Appendix A: Mixed Environment JavaScript

NOTE: |
:— |
Work in progress |

.

.

.

.

.

.

.

NOTE: |
:— |
Everything below here is previous text from 1st edition, and is only here for reference while 2nd edition work is underway. **Please ignore this stuff.** |

Beyond the core language mechanics we've fully explored in this book, there are several ways that your JS code can behave differently when it runs in the real world. If JS was executing purely inside an engine, it'd be entirely predictable based on nothing but the black-and-white of the spec. But JS pretty much always runs in the context of a hosting environment, which exposes your code to some degree of unpredictability.

For example, when your code runs alongside code from other sources, or when your code runs in different types of JS engines (not just browsers), there are some things that may behave differently.

We'll briefly explore some of these concerns.

Annex B (ECMAScript)

It's a little known fact that the official name of the language is ECMAScript (referring to the ECMA standards body that manages it). What then is "JavaScript"?

JavaScript is the common tradename of the language, of course, but more appropriately, JavaScript is basically the browser implementation of the spec.

The official ECMAScript specification includes "Annex B," which discusses specific deviations from the official spec for the purposes of JS compatibility in browsers.

The proper way to consider these deviations is that they are only reliably present/valid if your code is running in a browser. If your code always runs in browsers, you won't see any observable difference. If not (like if it can run in node.js, Rhino, etc.), or you're not sure, tread carefully.

The main compatibility differences:

- Octal number literals are allowed, such as `0123` (decimal `83`) in non-`strict mode`.
- `window.escape(..)` and `window.unescape(..)` allow you to escape or unescape strings with `%`-delimited hexadecimal escape sequences. For example: `window.escape( "?foo=97%&bar=3%" )` produces `"%3Ffoo%3D97%25%26bar%3D3%25"`.
- `String.prototype.substr` is quite similar to `String.prototype.substring`, except that instead of the second parameter being the ending index (noninclusive), the second parameter is the `length` (number of characters to include).

Web ECMAScript

The Web ECMAScript specification (http://javascript.spec.whatwg.org/) covers the differences between the official ECMAScript specification and the current JavaScript implementations in browsers.

In other words, these items are "required" of browsers (to be compatible with each other) but are not (as of the time of writing) listed in the "Annex B" section of the official spec:

- `<!--` and `-->` are valid single-line comment delimiters.
- `String.prototype` additions for returning HTML-formatted strings: `anchor(..)`, `big(..)`, `blink(..)`, `bold(..)`, `fixed(..)`, `fontcolor(..)`, `fontsize(..)`, `italics(..)`, `link(..)`, `small(..)`, `strike(..)`, and `sub(..)`. **Note:** These are very rarely used in practice, and are generally discouraged in favor of other built-in DOM APIs or user-defined utilities.
- `RegExp` extensions: `RegExp.$1` .. `RegExp.$9` (match-groups) and `RegExp.lastMatch`/`RegExp["$&"]` (most recent match).
- `Function.prototype` additions: `Function.prototype.arguments` (aliases internal `arguments` object) and `Function.caller` (aliases internal `arguments.caller`). **Note:** `arguments` and thus `arguments.caller` are

deprecated, so you should avoid using them if possible. That goes doubly so for these aliases – don't use them!

Note: Some other minor and rarely used deviations are not included in our list here. See the external "Annex B" and "Web ECMAScript" documents for more detailed information as needed.

Generally speaking, all these differences are rarely used, so the deviations from the specification are not significant concerns. **Just be careful** if you rely on any of them.

Host Objects

The well-covered rules for how variables behave in JS have exceptions to them when it comes to variables that are auto-defined, or otherwise created and provided to JS by the environment that hosts your code (browser, etc.) – so called, "host objects" (which include both built-in `object`s and `function`s).

For example:

```
var a = document.createElement( "div" );

typeof a;                                   // "object" -- as expected
Object.prototype.toString.call( a );    // "[object HTMLDivElement]"

a.tagName;                                  // "DIV"
```

`a` is not just an `object`, but a special host object because it's a DOM element. It has a different internal `[[Class]]` value (`"HTMLDivElement"`) and comes with predefined (and often unchangeable) properties.

Another such quirk has already been covered, in the "Falsy Objects" section in Chapter 4: some objects can exist but when coerced to `boolean` they (confoundingly) will coerce to `false` instead of the expected `true`.

Other behavior variations with host objects to be aware of can include:

- not having access to normal `object` built-ins like `toString()`
- not being overwritable
- having certain predefined read-only properties
- having methods that cannot be `this`-overriden to other objects
- and more...

Host objects are critical to making our JS code work with its surrounding environment. But it's important to note when you're interacting with a host

object and be careful assuming its behaviors, as they will quite often not conform to regular JS `objects`.

One notable example of a host object that you probably interact with regularly is the `console` object and its various functions (`log(..)`, `error(..)`, etc.). The `console` object is provided by the *hosting environment* specifically so your code can interact with it for various development-related output tasks.

In browsers, `console` hooks up to the developer tools' console display, whereas in node.js and other server-side JS environments, `console` is generally connected to the standard-output (`stdout`) and standard-error (`stderr`) streams of the JavaScript environment system process.

Global DOM Variables

You're probably aware that declaring a variable in the global scope (with or without `var`) creates not only a global variable, but also its mirror: a property of the same name on the `global` object (`window` in the browser).

But what may be less common knowledge is that (because of legacy browser behavior) creating DOM elements with `id` attributes creates global variables of those same names. For example:

```
<div id="foo"></div>
```

And:

```
if (typeof foo == "undefined") {
    foo = 42;       // will never run
}

console.log( foo ); // HTML element
```

You're perhaps used to managing global variable tests (using `typeof` or `.. in window` checks) under the assumption that only JS code creates such variables, but as you can see, the contents of your hosting HTML page can also create them, which can easily throw off your existence check logic if you're not careful.

This is yet one more reason why you should, if at all possible, avoid using global variables, and if you have to, use variables with unique names that won't likely collide. But you also need to make sure not to collide with the HTML content as well as any other code.

Native Prototypes

One of the most widely known and classic pieces of JavaScript *best practice* wisdom is: **never extend native prototypes**.

Whatever method or property name you come up with to add to `Array.prototype` that doesn't (yet) exist, if it's a useful addition and well-designed, and properly named, there's a strong chance it *could* eventually end up being added to the spec – in which case your extension is now in conflict.

Here's a real example that actually happened to me that illustrates this point well.

I was building an embeddable widget for other websites, and my widget relied on jQuery (though pretty much any framework would have suffered this gotcha). It worked on almost every site, but we ran across one where it was totally broken.

After almost a week of analysis/debugging, I found that the site in question had, buried deep in one of its legacy files, code that looked like this:

```
// Netscape 4 doesn't have Array.push
Array.prototype.push = function(item) {
    this[this.length] = item;
};
```

Aside from the crazy comment (who cares about Netscape 4 anymore!?), this looks reasonable, right?

The problem is, `Array.prototype.push` was added to the spec sometime subsequent to this Netscape 4 era coding, but what was added is not compatible with this code. The standard `push(..)` allows multiple items to be pushed at once. This hacked one ignores the subsequent items.

Basically all JS frameworks have code that relies on `push(..)` with multiple elements. In my case, it was code around the CSS selector engine that was completely busted. But there could conceivably be dozens of other places susceptible.

The developer who originally wrote that `push(..)` hack had the right instinct to call it `push`, but didn't foresee pushing multiple elements. They were certainly acting in good faith, but they created a landmine that didn't go off until almost 10 years later when I unwittingly came along.

There's multiple lessons to take away on all sides.

First, don't extend the natives unless you're absolutely sure your code is the only code that will ever run in that environment. If you can't say that 100%, then extending the natives is dangerous. You must weigh the risks.

Next, don't unconditionally define extensions (because you can overwrite natives accidentally). In this particular example, had the code said this:

```
if (!Array.prototype.push) {
    // Netscape 4 doesn't have Array.push
    Array.prototype.push = function(item) {
        this[this.length] = item;
    };
}
```

The `if` statement guard would have only defined this hacked `push()` for JS environments where it didn't exist. In my case, that probably would have been OK. But even this approach is not without risk:

1. If the site's code (for some crazy reason!) was relying on a `push(..)` that ignored multiple items, that code would have been broken years ago when the standard `push(..)` was rolled out.
2. If any other library had come in and hacked in a `push(..)` ahead of this `if` guard, and it did so in an incompatible way, that would have broken the site at that time.

What that highlights is an interesting question that, frankly, doesn't get enough attention from JS developers: **Should you EVER rely on native built-in behavior** if your code is running in any environment where it's not the only code present?

The strict answer is **no**, but that's awfully impractical. Your code usually can't redefine its own private untouchable versions of all built-in behavior relied on. Even if you *could*, that's pretty wasteful.

So, should you feature-test for the built-in behavior as well as compliance-testing that it does what you expect? And what if that test fails – should your code just refuse to run?

```
// don't trust Array.prototype.push
(function(){
    if (Array.prototype.push) {
        var a = [];
        a.push(1,2);
        if (a[0] === 1 && a[1] === 2) {
            // tests passed, safe to use!
            return;
        }
    }

    throw Error(
        "Array#push() is missing/broken!"
    );
})();
```

In theory, that sounds plausible, but it's also pretty impractical to design tests for every single built-in method.

So, what should we do? Should we *trust but verify* (feature- and compliance-test) **everything**? Should we just assume existence is compliance and let breakage (caused by others) bubble up as it will?

There's no great answer. The only fact that can be observed is that extending native prototypes is the only way these things bite you.

If you don't do it, and no one else does in the code in your application, you're safe. Otherwise, you should build in at least a little bit of skepticism, pessimism, and expectation of possible breakage.

Having a full set of unit/regression tests of your code that runs in all known environments is one way to surface some of these issues earlier, but it doesn't do anything to actually protect you from these conflicts.

Shims/Polyfills

It's usually said that the only safe place to extend a native is in an older (non-spec-compliant) environment, since that's unlikely to ever change – new browsers with new spec features replace older browsers rather than amending them.

If you could see into the future, and know for sure what a future standard was going to be, like for `Array.prototype.foobar`, it'd be totally safe to make your own compatible version of it to use now, right?

```
if (!Array.prototype.foobar) {
    // silly, silly
    Array.prototype.foobar = function() {
        this.push( "foo", "bar" );
    };
}
```

If there's already a spec for `Array.prototype.foobar`, and the specified behavior is equal to this logic, you're pretty safe in defining such a snippet, and in that case it's generally called a "polyfill" (or "shim").

Such code is **very** useful to include in your code base to "patch" older browser environments that aren't updated to the newest specs. Using polyfills is a great way to create predictable code across all your supported environments.

Tip: ES5-Shim (https://github.com/es-shims/es5-shim) is a comprehensive collection of shims/polyfills for bringing a project up to ES5 baseline, and similarly, ES6-Shim (https://github.com/es-shims/es6-shim) provides shims for new APIs added as of ES6. While APIs can be shimmed/polyfilled, new syntax generally cannot. To bridge the syntactic divide, you'll want to also

use an ES6-to-ES5 transpiler like Traceur (https://github.com/google/traceur-compiler/wiki/Getting-Started).

If there's likely a coming standard, and most discussions agree what it's going to be called and how it will operate, creating the ahead-of-time polyfill for future-facing standards compliance is called "prollyfill" (probably-fill).

The real catch is if some new standard behavior can't be (fully) polyfilled/prollyfilled.

There's debate in the community if a partial-polyfill for the common cases is acceptable (documenting the parts that cannot be polyfilled), or if a polyfill should be avoided if it purely can't be 100% compliant to the spec.

Many developers at least accept some common partial polyfills (like for instance `Object.create(..)`), because the parts that aren't covered are not parts they intend to use anyway.

Some developers believe that the `if` guard around a polyfill/shim should include some form of conformance test, replacing the existing method either if it's absent or fails the tests. This extra layer of compliance testing is sometimes used to distinguish "shim" (compliance tested) from "polyfill" (existence checked).

The only absolute take-away is that there is no absolute *right* answer here. Extending natives, even when done "safely" in older environments, is not 100% safe. The same goes for relying upon (possibly extended) natives in the presence of others' code.

Either should always be done with caution, defensive code, and lots of obvious documentation about the risks.

`<script>`s

Most browser-viewed websites/applications have more than one file that contains their code, and it's common to have a few or several `<script src=..></script>` elements in the page that load these files separately, and even a few inline-code `<script> .. </script>` elements as well.

But do these separate files/code snippets constitute separate programs or are they collectively one JS program?

The (perhaps surprising) reality is they act more like independent JS programs in most, but not all, respects.

The one thing they *share* is the single `global` object (`window` in the browser), which means multiple files can append their code to that shared namespace and they can all interact.

So, if one `script` element defines a global function `foo()`, when a second `script` later runs, it can access and call `foo()` just as if it had defined the function itself.

But global variable scope *hoisting* (see the *Scope & Closures* title of this series) does not occur across these boundaries, so the following code would not work (because `foo()`'s declaration isn't yet declared), regardless of if they are (as shown) inline `<script> .. </script>` elements or externally loaded `<script src=..></script>` files:

```
<script>foo();</script>

<script>
  function foo() { .. }
</script>
```

But either of these *would* work instead:

```
<script>
  foo();
  function foo() { .. }
</script>
```

Or:

```
<script>
  function foo() { .. }
</script>

<script>foo();</script>
```

Also, if an error occurs in a `script` element (inline or external), as a separate standalone JS program it will fail and stop, but any subsequent `script`s will run (still with the shared `global`) unimpeded.

You can create `script` elements dynamically from your code, and inject them into the DOM of the page, and the code in them will behave basically as if loaded normally in a separate file:

```
var greeting = "Hello World";

var el = document.createElement( "script" );

el.text = "function foo(){ alert( greeting );\
 } setTimeout( foo, 1000 );";

document.body.appendChild( el );
```

Note: Of course, if you tried the above snippet but set `el.src` to some file URL instead of setting `el.text` to the code contents, you'd be dynamically creating an externally loaded `<script src=..></script>` element.

One difference between code in an inline code block and that same code in an external file is that in the inline code block, the sequence of characters `</script>` cannot appear together, as (regardless of where it appears) it would be interpreted as the end of the code block. So, beware of code like:

```
<script>
  var code = "<script>alert( 'Hello World' )</script>";
</script>
```

It looks harmless, but the `</script>` appearing inside the `string` literal will terminate the script block abnormally, causing an error. The most common workaround is:

```
"</sc" + "ript>";
```

Also, beware that code inside an external file will be interpreted in the character set (UTF-8, ISO-8859-8, etc.) the file is served with (or the default), but that same code in an inline `script` element in your HTML page will be interpreted by the character set of the page (or its default).

Warning: The `charset` attribute will not work on inline script elements.

Another deprecated practice with inline `script` elements is including HTML-style or X(HT)ML-style comments around inline code, like:

```
<script>
<!--
alert( "Hello" );
//-->
</script>

<script>
<!--//--><![CDATA[//><!--
alert( "World" );
//--><!]]>
</script>
```

Both of these are totally unnecessary now, so if you're still doing that, stop it!

Note: Both `<!--` and `-->` (HTML-style comments) are actually specified as valid single-line comment delimiters (`var x = 2; <!-- valid comment` and `--> another valid line comment`) in JavaScript (see the "Web ECMAScript" section earlier), purely because of this old technique. But never use them.

Reserved Words

The ES5 spec defines a set of "reserved words" in Section 7.6.1 that cannot be used as standalone variable names. Technically, there are four categories: "keywords", "future reserved words", the `null` literal, and the `true` / `false` boolean literals.

Keywords are the obvious ones like `function` and `switch`. Future reserved words include things like `enum`, though many of the rest of them (`class`, `extends`, etc.) are all now actually used by ES6; there are other strict-mode only reserved words like `interface`.

StackOverflow user "art4theSould" creatively worked all these reserved words into a fun little poem (http://stackoverflow.com/questions/26255/reserved-keywords-in-javascript/12114140#12114140):

> Let this long package float, Goto private class if short. While protected with debugger case, Continue volatile interface. Instanceof super synchronized throw, Extends final export throws.
>
> Try import double enum? - False, boolean, abstract function, Implements typeof transient break! Void static, default do, Switch int native new. Else, delete null public var In return for const, true, char ... Finally catch byte.

Note: This poem includes words that were reserved in ES3 (`byte`, `long`, etc.) that are no longer reserved as of ES5.

Prior to ES5, the reserved words also could not be property names or keys in object literals, but that restriction no longer exists.

So, this is not allowed:

```
var import = "42";
```

But this is allowed:

```
var obj = { import: "42" };
console.log( obj.import );
```

You should be aware though that some older browser versions (mainly older IE) weren't completely consistent on applying these rules, so there are places where using reserved words in object property name locations can still cause issues. Carefully test all supported browser environments.

Implementation Limits

The JavaScript spec does not place arbitrary limits on things such as the number of arguments to a function or the length of a string literal, but these limits exist nonetheless, because of implementation details in different engines.

For example:

```
function addAll() {
    var sum = 0;
    for (var i=0; i < arguments.length; i++) {
        sum += arguments[i];
    }
    return sum;
}

var nums = [];
for (var i=1; i < 100000; i++) {
    nums.push(i);
}

addAll( 2, 4, 6 );              // 12
addAll.apply( null, nums );     // should be: 499950000
```

In some JS engines, you'll get the correct `499950000` answer, but in others (like Safari 6.x), you'll get the error: "RangeError: Maximum call stack size exceeded."

Examples of other limits known to exist:

- maximum number of characters allowed in a string literal (not just a string value)
- size (bytes) of data that can be sent in arguments to a function call (aka stack size)
- number of parameters in a function declaration
- maximum depth of non-optimized call stack (i.e., with recursion): how long a chain of function calls from one to the other can be
- number of seconds a JS program can run continuously blocking the browser
- maximum length allowed for a variable name
- ...

It's not very common at all to run into these limits, but you should be aware that limits can and do exist, and importantly that they vary between engines.

Review

We know and can rely upon the fact that the JS language itself has one standard and is predictably implemented by all the modern browsers/engines. This is a very good thing!

But JavaScript rarely runs in isolation. It runs in an environment mixed in with code from third-party libraries, and sometimes it even runs in engines/environments that differ from those found in browsers.

Paying close attention to these issues improves the reliability and robustness of your code.