The Past, Present, and Future of JavaScript

Where We've Been, Where We Are, and What Lies Ahead

Dr. Axel Rauschmayer

O'REILLY®

The Past, Present, and Future of JavaScript

Axel Rauschmayer

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

The Past, Present, and Future of JavaScript
by Axel Rauschmayer

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Mac Slocum
Production Editor: Melanie Yarbrough
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

July 2012: First Edition.

Revision History for the First Edition:
2012-07-20 First release
See *http://oreilly.com/catalog/errata.csp?isbn=9781449339968* for release details.

ISBN: 978-1-449-33996-8
1344364583

Table of Contents

The Past, Present, and Future of JavaScript

Over recent years, JavaScript has seen an impressive rise in popularity. Initially, that popularity was fueled by the success of client-side scripting, but JavaScript is increasingly being used in a general-purpose capacity, e.g. to write server and desktop applications. This article examines the JavaScript phenomenon from three angles:

We conclude with a wish list and a few parting thoughts.

The Past

In 1995, Netscape's Navigator was the dominant web browser and the company decided to add interactivity to HTML pages, via a lightweight programming language. They hired Brendan Eich to implement it. He finished a first version of the language in 10 days, in May 1995. Its initial name was Mocha which was changed to the more Netscape-esque LiveScript in September. In December, Netscape and Sun had a licensing agreement that led to the programming language's final name, JavaScript. At that point, it was included in Netscape Navigator 2.0B3.

The name "JavaScript" hints at the originally intended role for the language: Sun's Java was to provide the large-scale building blocks for web applications, while JavaScript was to be the glue, connecting the blocks. Obviously, that shared client-side responsibility never transpired: JavaScript now dominates the browser, Java is mostly dead there.

JavaScript was influenced by several programming languages: The Lisp dialect Scheme gave it its rules for variable scoping, including closures. The Self programming language—a Smalltalk descendant—gave it prototypal inheritance (object-based as opposed to class-based).

Because JavaScript was supposed to support Java, Netscape management demanded that its syntax be similar to Java's. That ruled out using other existing scripting languages such as Perl, Python or TCL. However, even though JavaScript's syntax is similar to Java's, it is quite a different language. [1 on page 36]

Standardization

After JavaScript came out, Microsoft implemented the same language, under the different name JScript, in Internet Explorer 3.0 (August 1996). Netscape decided to standardize JavaScript and asked the standards organization Ecma International to host the standard. Work on a specification called ECMA-262 started in November 1996. The first edition of ECMA-262 came out in June 1997. Because Sun (now Oracle) had a trademark on the word Java, the language to be standardized couldn't be called JavaScript. Hence, the following naming was chosen: ECMAScript is the name of the standard language, its implementations are officially called JavaScript, JScript, etc. Naturally, when talking unofficially about either the standard or its implementations, one mostly uses the term JavaScript. The current version of ECMAScript is 5.1, which means the same as ECMA-262, edition 5.1. That version has also become an ISO standard: ISO/IEC 16262:2011.

ECMA-262 is managed and evolved by Ecma's Technical Committee 39 (short: TC39). Its members are companies such as Microsoft, Mozilla, or Google, which appoint employees to participate in committee work—three names among several: Brendan Eich, Allen Wirfs-Brock (editor of ECMA-262) and David Herman. TC39's work includes discussing the design of upcoming versions via open channels such as a mailing list.

Reaching consensus and creating a standard is not always easy, but thanks to it, JavaScript is a truly open language, with implementations by multiple vendors that are remarkably compatible. That compatibility is made possible by a very detailed and concrete specification. For example, it often uses pseudocode to specify things. The specification is complemented by a test suite called test262 that checks an ECMAScript implementation for compliance. It is interesting to note that ECMAScript is not managed by the World Wide Web Consortium (W3C). TC39 and the W3C collaborate wherever there is overlap between JavaScript and HTML5.

Historic JavaScript Milestones

It took JavaScript a long time to make an impact. Many technologies existed for a while until they were discovered by the mainstream. This section describes what happened since JavaScript's creation until today. Throughout, only the most popular projects are mentioned and many are ignored, even if they were first. Two examples: the Dojo Toolkit is listed, but there is also the lesser-known qooxdoo, which was created around the same time. And Node.js is listed, even though Jaxer existed before it.

1997—Dynamic HTML. Dynamic HTML allows one to dynamically change the content and appearance of a web page. That is achieved by manipulating the Document Object Model (DOM) of the page: changing content, changing style, showing and hiding elements, etc. Dynamic HTML appeared first in Internet Explorer 4 and in Netscape Navigator 4.

1999—XMLHttpRequest. This API lets a client-side script send an HTTP or HTTPS request to a server and get back data, usually in a text format (XML, HTML, JSON). It was introduced in Internet Explorer 5.

2001—JSON, a JavaScript-based data exchange format. In 2001, Douglas Crockford named and documented JSON (JavaScript Object Notation)—a Lisp-like idea to use JavaScript syntax to store data in text format. JSON uses JavaScript literals for objects, arrays, strings, numbers, and booleans to represent structured data. Example:

```
{

    "first": "Jane",

    "last": "Porter",

    "married": true,

    "born": 1890,

    "friends": [ "Tarzan", "Cheeta" ]

}
```

Over the years, JSON has become a popular lightweight alternative to XML, especially when structured data is to be represented and not markup. Naturally, JSON is easy to consume via JavaScript.

2004—Dojo Toolkit, a framework for programming JavaScript in the large. The Dojo Toolkit facilitates programming in the large by providing the necessary infrastructure: an inheritance library, a module system, an API for desktop-style graphical widgets, etc.

2005—Ajax, browser-based desktop-class applications. Ajax is a collection of technologies that brings a level of interactivity to web pages that rivals that of desktop applications. One impressive example of what can be achieved via Ajax was introduced in February 2005: Google Maps. You were able to pan and zoom over a map of the world, but only the content that was currently visible was downloaded to the browser. After Google Maps came out, Jesse James Garrett noticed that it shared certain traits with other interactive websites. He called these traits "Ajax," an acronym for Asynchronous JavaScript and XML. The two cornerstones of Ajax are: First, loading content asynchronously in the background (via XMLHttpRequest). Second, dynamically updating the current page with the results (via dynamic HTML). That was a considerable usability improvement from always performing complete page reloads.

Ajax marked the mainstream breakthrough of JavaScript and dynamic web applications. It is interesting to note how long that took—at that point, the Ajax ingredients had been available for years. Since the inception of Ajax, other data formats have become popular (JSON instead of XML), other protocols are used (e.g., Web Sockets in addition to HTTP) and bi-directional communication is possible. But the basic techniques are still the same. However, the term Ajax is used much less these days and has mostly been replaced by the more comprehensive term HTML5 (JavaScript plus browser APIs).

2005—Apache CouchDB, a JavaScript-centric database. Roughly, CouchDB is a JSON database: You feed it JSON objects, without the need to specify a schema in advance. Additionally, you can define views and indexes via JavaScript functions that perform map/reduce operations. Hence, CouchDB is a very good fit for JavaScript because you can directly work with native data. Compared to a relational database, there is no mapping-related impedance mismatch. Compared to an object database, you avoid many complications because only data is stored, not behavior. CouchDB is just one of several similar "NoSQL" databases. Most of them have excellent JavaScript support.

2006—jQuery, helping with DOM manipulation. The browser DOM (Document Object Model) is one of the most painful parts of client-side web development. jQuery made DOM manipulation fun by abstracting over browser differences and by providing a powerful fluent-style API for querying and modifying the DOM.

2007—WebKit becomes the foundation of the mobile web. Based on prior work by KDE, WebKit is an HTML engine that was introduced by Apple in 2003. It was open-sourced in 2005. With the introduction of the iPhone in 2007, it became *the* foundation of the mobile web. WebKit is now the main engine for Android and the only engine for iOS, and dominates the mobile

market. That means if you want to write a cross-platform mobile application, then web technology is currently your best bet (unless you need features that are only available natively).

2008—V8, JavaScript can be fast. When Google introduced its Chrome web browser, one of its highlights was a fast JavaScript engine called V8. It changed the perception of JavaScript as being slow and led to a speed race with other browser vendors that we are still profiting from. V8 is open source and can be used as a stand-alone component whenever one needs a fast embedded language that is widely known.

2009—Node.js, JavaScript on the server. Node.js lets you implement servers that perform well under load. To do so, it uses event-driven non-blocking I/O and JavaScript (via V8). Node.js creator Ryan Dahl mentions the following reasons for choosing JavaScript:

- "Because it's bare and does not come with I/O APIs." [Node.js can thus introduce its own non-blocking APIs.]
- "Web developers use it already." [JavaScript is a widely known language, especially in a web context.]
- "DOM API is event-based. Everyone is already used to running without threads and on an event loop." [Web developers are not scared of callbacks.]

Dahl was able to build on prior work on event-driven servers and server-side JavaScript (mainly the CommonJS project).

The appeal of Node.js for JavaScript programmers goes beyond being able to program in a familiar language; you get to use the same language on both client and server. That allows you to do clever things such as a fallback for browsers that don't run sufficiently sophisticated versions of JavaScript: Assemble the pages on the server—with the same code that you are using on JavaScript-enabled clients (examples: FunctionSource's web server, Yahoo Cocktails).

The Present

Dynamic web applications led to JavaScript's initial popularity. The Web had become an exciting ecosystem for applications, and you needed JavaScript to be part of it. As outlined above, many more pieces have since been added to the JavaScript puzzle that helped make it an appealing general-purpose language. JavaScript programs have become fast thanks to modern JavaScript engines. They can use JavaScript-centric databases and exchange data with their environment (e.g. with web services) via JSON. Server-side JavaScript allows one to use the same language for the server and for the client. Node.js

also makes it possible to use JavaScript for build scripts and shell scripts. Lastly, JavaScript is possibly the most open programming language there is: no single party controls it, there is a well-written specification for it (whose value cannot be overestimated), and there are several highly compatible implementations available.

The Future

The future brings many exciting developments for JavaScript: ECMAScript.next will fix quirks and add new features. It will also be a better target language for compilers. Support for concurrency is important and several options for JavaScript are currently being evaluated. Finally, HTML5 is becoming a compelling platform, not just for web applications, but also for mobile applications and desktop applications.

Evolving the Language

After ECMAScript 3, there was a period where TC39 was divided on how to evolve the language. A meeting in August 2008 resolved the division. The agreement was to develop both an incremental update (which eventually became ECMAScript 5) and major new language features. Those features were code-named Harmony, due to the nature of the meeting. ECMAScript Harmony is too large in scope to fit into a single version. Hence, there is another code name, ECMAScript.next, for the next version of ECMAScript. ECMAScript.next will probably become ECMAScript 6. Harmony's requirements and goals are described as follows on the ECMAScript wiki [2 on page 36]:

Requirements:

- New features require concrete demonstrations.
- Keep the language pleasant for casual developers.
- Preserve the "start small and iteratively prototype" nature of the language.

Goal 1 (of several) is for ECMAScript Harmony to "be a better language for writing ..."

a. complex applications;
b. libraries (possibly including the DOM) shared by those applications;
c. code generators targeting the new edition.

How features are added to ECMAScript.next

The process for adding a new feature to the standard is as follows. First, a proposal for the feature is written, by a "champion", an expert in the relevant

domain. That avoids some of the dangers of "design by committee". Next, promising proposals are field-tested via prototype implementations in one or more browsers. If the feature proves itself, its specification is added to the ECMAScript.next draft [3 on page 36]. Naturally, while taking feedback from the field test into account.

The following sections describe several ECMAScript.next proposals. Some of them are likely to be included (e.g. `let`), others are still under discussion (e.g., class declarations, more number types). Until the projected completion of ECMAScript.next at the end of 2013, nothing is set in stone. Features might change or TC39 might consider ECMAScript.next as having become too large and discard features.

Block scoping via let and const

In current JavaScript, variables are function-scoped—if you declare a variable via `var`, it exists within all of the innermost function that surrounds it. In the following example, `tmp` is not confined to the then-block:

```
function order(x, y) {

    console.log(tmp); // undefined

    if (x > y) {

        var tmp = x;

        x = y;

        y = tmp;

    }

    return [x, y];

}
```

ECMAScript.next will additionally have block-scoped bindings. Those are supported by `let`, `const` and block functions (function declarations becoming block-scoped). The `let` statement is a block-scoped version of `var` ("`let` is the new `var`"). Variables declared by it exist only within the innermost inclosing block. For example:

```
function order(x, y) {

    console.log(tmp); // ReferenceError: tmp is not defined

    if (x > y) {

        let tmp = x;
```

```
        x = y;
        y = tmp;
    }
    return [x, y];
}
```

`const` is similar to `let`, but variables declared by it can only be initialized once and cannot be changed afterward—they are read-only. Furthermore, you get an error if you read the variable before it has been initialized. For example:

```
const BUFFER_SIZE = 256;
```

Handling parameters, assignment

ECMAScript.next will make the special variable `arguments` obsolete. Instead, one will be able to use parameter default values and rest parameters:

```
// Parameter default values
function foo(mandatory, optional1=123, optional2="abc") { ... }


// Rest parameters
function bar(arg1, ...remainingArguments) { ... }
```

Assignment can be destructuring, it can extract values from the right-hand side, via a simple form of pattern matching:

```
// Swap the values of a and b
let [b, a] = [a, b];


// Assign the value of property x to horiz
let { x: horiz } = { x: 5, y: 3 };
console.log(horiz);  // 5
```

Destructuring works even for parameters, which is handy if you want to have labeled optional parameters:

```
// Declare
function moveBy(point, { x = 0, y = 0 }) {
    ...
```

```
}

// Call
moveBy(somePoint, { x: 2, y: 7 });
moveBy(somePoint, {});
```

One can also allow the object with the optional parameters to be omitted:

```
// Declare
function moveBy(point, { x = 0, y = 0 } = { x: 0, y: 0 }) {
    ...
}

// Call
moveBy(somePoint);
```

The invocation analog of a rest parameter is the spread operator:

```
function myfunc(...args) {
    otherfunc(...args);
}
```

In ECMAScript 5, you would write the following code, instead:

```
function myfunc() {
    otherfunc.apply(null, arguments);
}
```

Arrow functions

JavaScript uses the same construct, the function, for both methods and non-method functions. The problem with the former is that it is potentially confusing to read the word `function` in the context of a method definition. The problem with the latter is that you want to access the `this` of the surrounding context ("lexical `this`"), instead of shadowing it with your own `this`. To work around this limitation, one often uses a pattern such as `that = this` (*):

```
let jane = {
    name: "Jane",
```

```
    logHello: function (friends) {
        var that = this;  // (*)
        friends.forEach(function (friend) {
            console.log(that.name + " says hello to " + friend);
        });
    }
}
```

ECMAScript.next fixes both problems. First, there is a shorter notation for defining methods (*). Second, it has arrow functions with lexical `this` (**):

```
let jane = {
    name: "Jane",

    logHello(friends) {  // (*)
        friends.forEach(friend => {  // (**)
            console.log(this.name + " says hello to " + friend);
        });
    }
}
```

An arrow function achieves lexical `this` via binding, so the following two statements are roughly equivalent (`bind` creates two function objects, an arrow function only a single one):

```
let func = (a, b) => { return this.offset + a + b }

let func = function (a, b) { return this.offset + a + b }.bind(this);
```

The body of an arrow function can also be an expression, making code much more readable. Compare:

```
let squares = [ 1, 2, 3 ].map(function (x) { return x * x });

let squares = [ 1, 2, 3 ].map(x => x * x);
```

Object literals

There are several proposals for constructs that help with JavaScript object-orientation. Object literals will get a large upgrade:

```
let name = "foo";

let obj = MyProto <| {        // "prototype for" operator

    myMethod(arg1, arg2) {           // method definition

        super.myMethod(arg1, arg2);    // super reference

    },

    [name]: 123,                 // computed property key

    bar                         // property value shorthand

}
```

Features shown above:

Prototype-for operator (<|) lets you specify the prototype of a literal, which is certainly more convenient than `Object.create()`. And it mostly obviates the need for the special property `__proto__` which is non-standard and problematic.

Method definitions can be made via a shorter syntax.

Super-references let you refer to a property appearing later in the prototype chain. The search for such a property starts in the prototype of the object where the method is located. Note that this way of referring to super-properties works for both subtypes (where one can, e.g., refer to a method of the supertype that has been overridden) and directly constructed prototype chains.

Computed property keys are a way to specify the name of a property via an expression (which is especially useful for private name objects—see below).

A property value shorthand eliminates some redundancy when filling an object literal with data. `bar` is an abbreviation for

```
bar: bar
```

More examples:

```
{ foo }   // same as { foo: foo }

{ x, y }  // same as { x: x, y: y }
```

Object extension literals

With super-references, a method is much tighter bound to the object it is stored in and needs to have a reference to it. The method definition syntax above takes care of that and is thus the preferred way of creating methods. An object extension literal ensures that this syntax can also be used to add methods to an existing object. Example:

```
obj.{

    anotherMethod(x) {

        return super.anotherMethod(10 + x);

    }

}
```

If you want to add the above method "manually," you can't use assignment, you have to use the new function `Object.defineMethod`:

```
Object.defineMethod(obj, "anotherMethod", function (x) {

    return super.anotherMethod(10 + x);

});
```

Another example:

```
function Point(x, y) {

    this.{ x, y };

}
```

This is equivalent to:

```
function Point(x, y) {

    this.x = x;

    this.y = y;

}
```

Private name objects

If you want to keep data private in JavaScript, you currently have two main options: You can either put it into the environment of the constructor, or you can keep it in properties with special names (e.g. starting with an underscore). Using the environment of the constructor means that only methods that are added there can access private data, but not, say, "public" methods in the

prototype. To do so, they need "privileged" methods that have been added to the instance in the constructor.

The following example is not realistic, but it illustrates how things work.

```
// Private data: factor

function Multiplier(factor) {

    // Privileged method:

    this.getFactor = function() {

        return factor;

    };

}

// Public method:

MyType.prototype.multiply = function (value) {

    return this.getFactor() * value;

};
```

The following code is similar, but keeps the private data in the property `_factor`. The name starting with an underscore marks that property as private. The problem with this approach is that such properties occupy the same namespace as public properties and pollute it, when they should be hidden. And the private data is not safe from outside access.

```
function Multiplier(factor) {

    // Private data:

    this._factor = factor;

}

// Public method:

MyType.prototype.multiply = function (value) {

    return this._factor * value;

};
```

The proposal "private name objects" for ECMAScript.next combines the advantages of both approaches and works as follows. Currently, property names have to be strings. The proposal additionally allows one to use name objects that have to be explicitly created and are unique and unforgeable:

```
import Name from "@name";

let factorKey = new Name();  // create private name object


function Multiplier(factor) {

    // Private data:

    this[factorKey] = factor;

}

// Public method:

MyType.prototype.multiply = function (value) {

    return this[factorKey] * value;

};
```

The property whose name is the name object `factorKey` cannot be found via any enumeration mechanism (`Object.getOwnPropertyNames()`, `for...in`, etc.). And one can only access its value if one has its name. Hence, there is no namespace pollution and the private data is safe.

Class declarations

Implementing single types via constructor functions is reasonably easy in JavaScript, but subtyping is too complicated and making super-references is clumsy.

Right now, you have to write:

```
// Supertype

function Point(x, y) {

    this.x = x;

    this.y = y;

}

Point.prototype.toString = function () {

    return "("+this.x+", "+this.y+")";
```

```
};

// Subtype
function ColorPoint(x, y, color) {
    Point.call(this, x, y);
    this.color = color;
}
ColorPoint.prototype = Object.create(Point.prototype);
ColorPoint.prototype.constructor = ColorPoint;
ColorPoint.prototype.toString = function () {
    return this.color+" "+Point.prototype.toString.call(this);
};
```

If the proposal "maximally minimal classes" is accepted, you'll be able to write the following, instead.

```
// Supertype
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    toString() {
        return "("+this.x+", "+this.y+")";
    }
}

// Subtype
class ColorPoint extends Point {
    constructor(x, y, color) {
```

```
        super.constructor(x, y);

        this.color = color;

    }

    toString() {

        return this.color+" "+super.toString();

    }

}
```

Modules

JavaScript has no built-in support for modules with imports and exports. You can implement quite powerful module systems in the language, but there are several competing standards. The two most important ones are: Asynchronous Module Definitions on the client side and Node.js modules (which stay very close to the CommonJS standard) on the server side. ECMAScript.next will provide a common module standard for the language. The following shows the module definition syntax in action.

```
import 'http://json.org/modules/json2.js' as JSON;

import { add, sub } from Math;


export const HELLO_STRING = JSON.stringify({ hello: 'world' });

export function total(...args) {

    args.reduce((prevResult, elem) => add(prevResult, elem));

}
```

ECMAScript.next modules are more static than module systems implemented in JavaScript. That enables benefits such as better performance, pre-fetching, and compile-time checking (find unbound variables etc.).

Loops and iteration

One can use the `for...in` loop to iterate over objects. However, that loop has several quirks. ECMAScript.next will introduce a new iteration infrastructure, including a `for...of` loop that fixes the `for...in` quirks. For example, it iterates over the elements of an array, not over the property names:

```
let arr = [ "hello", "world" ];

for (let elem of arr) {
```

```
    console.log(elem);
}
```

Output:

```
hello
world
```

Iterating over objects has also become more convenient:

```
let obj = { first: "Jane", last: "Doe" };

// Iterate over properties
for (let [name, value] of obj) {
    console.log(name + " = " + value);
}

// Iterate over property names
import keys from "@iter";  // returns an iterable (see below)
for (let name of keys(obj)) {
    console.log(name);
}
```

`for...of` works in conjunction with an iteration protocol: An object can implement a custom way of being iterated over if it has a method `iterate()` that returns a so-called iterator object (short: iterator). An object with such a method is called an iterable. The name of the method is a name object (see above). An iterator has a single method `next()` that, when invoked often enough, returns all the "elements" of the iterable. If it is invoked more times than the number of elements, a `StopIteration` exception is thrown. For example:

```
import iterate from "@iter";  // name object

function iterArray(arr) {
    let i = -1;
    // The returned object is both iterable and iterator
    return {
```

```
            [iterate]() {  // property name is a name object
                return this;  // an iterator
            },
            next() {  // iterator method
                i++;
                if (i < arr.length) {
                    return arr[i]
                } else {
                    throw new StopIteration();
                }
            }
        }
    }
```

`iterArray()` is used as follows:

```
    for (let elem of iterArray(["a", "b"])) {
        console.log(elem);
    }
```

Generators

Generators are lightweight co-routines. When invoked, they create an object that wraps a function. One can continue the evaluation of that function via the `next()` method and pause the execution via the `yield` operator (inside the function); `yield` is closely related to `return` and provides the value that `next()` returns. A generator is produced via the keyword `function*` (alas, one couldn't use the keyword `generator` because that might break existing code). Let's take the following non-generator function `iterTreeFunc` and turn it into a generator.

```
    // Iterate over a tree of nested arrays
    function iterTreeFunc(tree, callback) {
        if (Array.isArray(tree)) {
            // inner node
```

```
            for(let i=0; i < tree.length; i++) {
                iterTreeFunc(tree[i], callback);
            }
        } else {
            // leaf
            callback(tree);
        }
    }
```

Interaction:

```
> iterTreeFunc([[0, 1], 2], function (x) { console.log(x) });
0
1
2
```

`iterTreeFunc` looks as follows if written as a generator:

```
function* iterTree(tree) {
    if (Array.isArray(tree)) {
        // inner node
        for(let i=0; i < tree.length; i++) {
            yield* iterTree(tree[i]); // yield recursively
        }
    } else {
        // leaf
        yield tree;
    }
}
```

Interaction:

```
> let gen = iterTree([[0, 1], 2]);
> gen.next()
```

```
0

> gen.next()

1

> gen.next()

2

> gen.next()

Exception: StopIteration
```

As you can see, the generator object is also an iterator and can thus be iterated over:

```
for (let x of iterTree([[0, 1], 2])) {

    console.log(x);

}
```

task.js—an application of generators. If you want to see an intriguing application of generators, take a look at David Herman's task.js library. It lets you write asynchronous code in a synchronous style, via generators. The following is an example of task.js-enabled code (slightly edited from task.js on GitHub).

```
spawn(function* () {

    try {

        var [foo, bar] = yield join(

            read("foo.json"), read("bar.json"))

            .timeout(1000);

        render(foo);

        render(bar);

    } catch (e) {

        console.log("read failed: " + e);

    }

});
```

Writing the same code with callbacks is much more complicated:

```
var foo, bar;

var tid = setTimeout(

    function () { failure(new Error("timed out")) },

    1000);


var xhr1 = makeXHR("foo.json",

                   function (txt) { foo = txt; success() },

                   failure);

var xhr2 = makeXHR("bar.json",

                   function (txt) { bar = txt; success() },

                   failure);


function success() {

    if (typeof foo === "string" && typeof bar === "string") {

        cancelTimeout(tid);

        xhr1 = xhr2 = null;

        render(foo);

        render(bar);

    }

}


function failure(e) {

    if (xhr1) {

        xhr1.abort();

        xhr1 = null;

    }

    if (xhr2) {

        xhr2.abort();
```

```
            xhr2 = null;
        }
        console.log("read failed: " + e);
    }
```

More number types

All numbers in JavaScript are 64-bit floating point (IEEE 754 double precision). Of those 64 bits, only 53 are available for integers. Naturally, one faces the problem of limited precision when it comes to very large numbers or very small fractions. But the main problem for JavaScript in non-scientific computing is that it can't handle 64-bit integers, which are very common, e.g. as database keys. Then the question is how to best add support for more number types to the language. ECMAScript.next will probably do so via value objects. Value objects are objects (non-primitive values) that are immutable. Where normal objects (reference objects) are compared by reference, value objects are compared by value (one "looks inside" such objects when comparing).

```
{} === {}  // false

uint64(123) === uint64(123)  // true
```

`uint64` is a constructor for 64-bit unsigned integers. Not using the `new` operator and the name starting with a lowercase letter are indications that value objects are different from reference objects. Value objects are considered objects by `typeof`:

```
typeof uint64(123)  // "object"
```

That means that current code won't be confused when it is confronted with a value object. In contrast, adding new primitive types to the language would likely cause more problems. For ECMAScript.next, value objects will probably only be used to introduce new number types. Later, programmers might be able to implement their own object value types, possibly including overloaded operators. Thanks to `uint64` values being immutable and compared by value, JavaScript engines have the option of directly working with (unboxed) 64-bit values. That should bring considerable performance benefits to the language. The current prototype implementation [4 on page 36] in Firefox gives you just `uint64` (unsigned 64-bit integers) and `int64` (signed 64-bit integers), plus properly working operators (`+`, `*`, etc.). There are also literals such as `-123L` for `int64(-123)`.

For example:

```
> 123L + 7L

130L
```

More number types might be supported later.

Binary data

JavaScript is increasingly used in areas where it has to process binary data as efficiently as possible. Examples: handle network protocols, decode video, interoperate with native APIs and processors, and encode and decode various serialization formats. In order to support these use cases, there is a proposal for binary data types:

```
const Point = new StructType({ x: uint32, y: uint32 });

const Triangle = new ArrayType(Point, 3);

let pt = new Triangle([

    { x: 0, y: 3 }, { x: 0, y: 3 }, { x: 0, y: 3 }

]);
```

`Point` and `Triangle` are binary data types, and can be used to parse and generate binary data.

Quasi-literals for string interpolation

String interpolation such as templating is a very common operation in JavaScript. Quasi-literals help with it. A quasi literal is written as

```
quasiHandler`Hello ${firstName} ${lastName}`
```

Roughly, this is just a different way of writing a function call:

```
quasiHandler("Hello ", firstName, " ", lastName)
```

The actual handler invocation is a bit more complex. Among other things, it allows the handler to make the distinction between the static parts (such as `"Hello"`) and the dynamic parts (such as `firstName`). Some handlers benefit from caching the former, e.g. when the same invocation is made multiple times in a loop.

A few usage examples:

Raw strings: string literals with multiple lines of text and no interpretation of escaped characters.

```
let str = raw`This is a text
```

```
with multiple lines.

Escapes are not interpreted,

\n is not a newline.`;
```

Simple string interpolation:

```
alert(`Error: expected ${expected}, but got ${actual}`);
```

By omitting the handler name, the above code invokes a default handler. In Node.js, the same command would be written as

```
alert(util.format("Error: expected %s, but got %s", expected, actual));
```

Parameterized regular expression literals that ignore whitespace:

```
re`\d+ ( ${localeSpecificDecimalPoint} \d+ )?`
```

Above, `localeSpecificDecimalPoint` contains either a dot or a comma, and will be quoted appropriately before being inserted into the regular expression. Additionally, whitespace is ignored, so you can insert spaces and newlines to make the regular expression more readable.

Arguments for query languages:

```
$`a.${className}[href=~'//${domain}/']`
```

Similarly to the regular expression example, `className` and `domain` are to be inserted into the query language command and will be escaped properly.

Localizable messages:

```
alert(msg`Welcome to ${siteName}, you are visitor

          number ${visitorNumber}:d!`);
```

Above, the static parts are used to look up a translation to the currently used language. The dynamic parts are inserted into the result, possibly after having been reordered. The marker `:d` indicates that `visitorNumber` should be displayed with a locale-specific decimal separator (English 1,300 versus German 1.300).

Templates:

```
let myTmpl = tmpl`

<h1>${{title}}</h1>

${{content}}

`;
```

The handler will get the template content in a parsed (segmented) version and compile it. Note that `{title}` is an object literal and an abbreviation for `{ title: title }`.

Proxies

Proxies allow you to put a handler "in front of" an object to intercept operations applied to it. Given the following definitions:

```
// target points to an object

let proxy = Proxy(target, handler);
```

Each of the following operations triggers a method invocation on `handler`:

```
proxy["foo"]             → handler.get(target, "foo", proxy)

proxy["foo"] = 123       → handler.set(target, "foo", 123, proxy)

"foo" in proxy           → handler.has(target, "foo")

for (key in proxy) {...} → handler.enumerate(target)
```

Proxies can also be put into the prototype chain of an object:

```
let child = Object.create(proxy)
```

Operations on `child` that are passed on to the prototype `proxy` obviously still trigger handler method invocations:

```
child["foo"]             → handler.get(target, "foo", child)

child["foo"] = 123       → handler.set(target, "foo", 123, child)

"foo" in child           → handler.has(target, "foo")

for (key in child) {...} → handler.enumerate(target)
```

One application of this is to turn a proxy into a sentinel for methods that don't exist:

```
let handler = {

    get(target, name, receiver) {

        return (...args) => {

            console.log("Missing method "+name+", arguments: "+args);

        }

    }

};
```

```
let proxy = Proxy({}, handler);
```

Using the handler:

```
$ let obj = Object.create(proxy);

$ obj.foo(1, 2)

Missing method foo, arguments: 1, 2

undefined
```

Use cases for proxies include meta-programming tasks such as:

- Sending all method invocations to a remote object
- Implementing data access objects for a database
- Data binding
- Logging

Collections

JavaScript does not have proper collection types. Hence, people often use objects as maps from strings to values and as sets of strings. But using them correctly in this role is difficult. ECMAScript.next will provide three collection types: `Map`, `Set` and `WeakMap`. Maps can have arbitrary keys (not just strings) and are used as follows:

```
let map = new Map();

let obj = {};


map.set(true, 1); // key not converted to string!

map.set("true", 2);

map.set(obj, 3);


console.log(map.get(true)); // 1

console.log(map.get("true")); // 2

console.log(map.get(obj)); // 3


console.log(map.has(obj)); // true

console.log(map.has({})); // false
```

```
map.delete(obj);

console.log(map.has(obj)); // false
```

The following code demonstrates sets:

```
let set = new Set();


set.add("hello");


console.log(set.has("hello")); // true

console.log(set.has("world")); // false
```

A weak map is a map that holds objects without preventing them from being garbage-collected: Neither keys nor values are held strongly — if they are only referenced by weak maps, then they will be disposed of. Furthermore, one cannot iterate over keys, values or entries of weak maps. You must have a key in order to get to a value. That enables security applications where secrets are kept in a shared data structure. One use case is to hold instance-private data in a shared global weak map. The keys of the weak map are the instances; the values are objects with the private data. If an instance is garbage-collected, the associated private data will be disposed of, too.

API improvements

Several smaller improvements of the standard library have been proposed. Some of them provide new functionality:

```
> "abc".repeat(3)

'abcabcabc'

> "abc".startsWith("ab")

true

> "abc".endsWith("bc")

true
```

Others fix quirks:

```
> Array.of(1, 2, 3)

[ 1, 2, 3 ]

> Array.of(3)

[ 3 ]
```

```
> Number.isNaN("abc")

false
```

The above constructs fix the following quirks:

```
> new Array(1, 2, 3)  // OK

[ 1, 2, 3 ]

> new Array(3)  // quirk: empty array of length 3

[ , ,  ]

> isNaN("abc")  // quirk: converts to number first

true
```

JavaScript as a Compilation Target

ECMAScript Harmony goal (1c) might be surprising: "Be a better language for writing code generators." But JavaScript is indeed increasingly becoming the target language for compilers. A few examples:

- The Google Web Toolkit (GWT) allows one to write web applications completely in Java. On the server, Java is executed directly. On the client, it is compiled to JavaScript.
- CoffeeScript is a dialect of JavaScript that has a different syntax, avoiding braces and semicolons, and simplifies several tasks.
- Google's Traceur compiles a variant of ECMAScript.next to JavaScript, on the fly.
- Emscripten compiles LLVM bitcode to JavaScript. That bitcode can be generated from C and C++, which means that many interesting C-based projects suddenly run on JavaScript engines. A few examples: SQLite (an SQL database), eSpeak (a speech synthesizer), FreeType (a TrueType font rendering engine). Surprisingly, the generated JavaScript code runs quite fast.
- Minification transforms a JavaScript program into a more compact version of itself without changing what it does. Measures taken are: stripping out comments and newlines, using shorter variables names, etc.

Some even call JavaScript the "assembly language of the web." But while being a good compilation target has been a goal for ECMAScript.next, making it a good programming language in its own right is even more important, so languages compiling to JavaScript will always face stiff competition from JavaScript itself.

Staying in the source language

When a source language is compiled to JavaScript, we want to work with the former as much as possible. Alas, that is currently infeasible when it comes to running the code:

- Exceptions report lines in the target code, not in the source language code.
- Output in the browser console links back to target code.
- Debugging has to be done in the target code.

Source Maps help with staying in the source language in all three cases: If a file `file.orig` (where `orig` is `java`, `cs`, etc.) has been compiled to a JavaScript file `file.js`, then a Source Map is a companion file `file.js.map`. It maps locations in `file.js` to locations in `file.orig`. That mapping is used to report errors with `file.orig`'s line numbers and to link to its source from the console. Working with a debugger in the original language will also eventually be possible. Firefox and WebKit already have preliminary support for Source Maps.

Being a versatile compilation target

Several features in ECMAScript.next make it a more versatile compilation target.

Avoiding stack growth for tail calls. A tail call is a function call that is performed at the end of a function:

```
function logNumbers(start, end) {

    if (start >= end) return;

    console.log(start);

    logNumbers(start+1, end);  // tail call

}
```

Under current JavaScript, the stack grows with each tail call. But that is unnecessary because the call does not need to go back to its call site (e.g. to return a value). Most functional programming languages avoid stack growth in such cases. There is a proposal that would allow JavaScript engines to do the same. Naturally, that would make JavaScript a more appealing compilation target for functional languages.

More useful features:

- Improved support for binary data avoids compiler writers being limited by JavaScript's built-in types.

- Data parallelism (similar to the River Trail approach described above) is being discussed for Harmony (post ECMAScript.next).

Concurrency

For a long time, computing speed increases came from faster processor cores. But the rate of those increases has slowed down. Further speed improvements must come from using more cores, in parallel. Therefore, if a programming language is to be general purpose, it must support concurrency. Such support is difficult to get right, but several promising approaches have been proposed for JavaScript.

Web Workers [5 on page 36] are the current weapon of choice for concurrent computation in JavaScript. All modern browsers support it and so does Node.js. A worker is JavaScript code that runs in a new operating system thread and has limited access to its environment (e.g., it can't access the browser's DOM). Workers and the main thread can only communicate by sending simple messages, either strings or JSON data. All of these limitations make workers a robust mechanism—one is relatively safe from concurrency problems such as deadlocks. The following code shows how the main thread starts a worker, waits for messages via a "message" listener, and sends a message to the worker (the string `"Hello worker!"`).

```
var worker = new Worker('worker_code.js');

worker.onmessage = function (event) {

    console.log("Message from worker: "+event.data);

};

worker.postMessage("Hello worker!");
```

The code in `worker_code.js` looks as follows. It listens for "message" events from the main thread and sends the message `"Hello main!"` to that thread.

```
self.onmessage = function (event) {

    console.log("Message from main thread: "+event.data);

};

self.postMessage("Hello main!");
```

Normal JavaScript code runs in the same thread as the user interface. Thus, executing long-running operations in the background via a worker keeps the UI responsive. If a worker needs more threads, it has the option to start new workers, so-called subworkers.

WebCL [6 on page 36] brings to JavaScript a subset of the OpenCL standard [7 on page 36] that allows one to send tasks to multiple processing cores, including ones in graphical processing units. In OpenCL terminology, one runs kernels (pieces of code) on devices. Each device has one or more compute units, where the actual computation is performed. Work (kernel + parameters) is sent to devices via command queues, where it waits until a computation unit is free. On one hand, WebCL is very flexible. You can precisely control where your code should be executed. And two kinds of parallelism are supported:

- Data parallelism: several instances of the same kernel run in parallel, each of them has its own data that it operates on.
- Task parallelism: different kernels run in parallel, similar to the Unix model of spawning processes.

On the other hand, WebCL requires much manual work: You have to manage devices yourself and need to write the OpenCL code in a C dialect.

River Trail [8 on page 36] is an experiment by Intel Labs that adds data parallelism to JavaScript, but without having to explicitly control it, as with WebCL. It introduces the new type `ParallelArray` with transformation methods that are parameterized via a function implementing the transformation (a so-called elemental function). Arrays have similar methods (e.g. `Array.prototype.map`), but `ParallelArray`'s methods execute their elemental functions several times in parallel. The following code uses `ParallelArray`:

```
var a = new ParallelArray(1, 2, 3);

var squares = a.map(function (x) { return x * x });
console.log(String(squares));  // [1, 4, 9]

var cubes = a.combine(function (index) {
    return a.get(index) * squares.get(index);
});
console.log(cubes);  // [1, 8, 27]
```

The current River Trail prototype is an extension for Firefox. To distribute the work, it uses OpenCL, which must be installed on the operating system.

Conveniently, one has the option of using a sequential implementation of `ParallelArray`, in pure JavaScript, if River Trail is not installed.

Writing Non-Web Applications in JavaScript

HTML5 is slowly becoming a complete layer for writing full-featured cross-platform applications. Similar to, say, the Java platform.

Mobile apps: PhoneGap—PhoneGap is a project that allows you to write mobile apps in HTML5. Those apps will run on seven platforms: iOS (iPhone, iPod touch), Android, Blackberry, webOS, Windows Phone, Symbian, and Bada. Apart from HTML5 APIs, there are also PhoneGap-specific APIs for accessing native features such as accelerometer, camera and contacts. Given how well web technologies are supported by mobile platforms these days, HTML5 has become the best choice for writing cross-platform mobile apps. PhoneGap allows you to deploy those apps natively.

Desktop apps: Mozilla's Open Web Apps project—The most interesting aspect of this project is that it supports desktop operating systems: It lets you install a web application as a native application on Windows, Mac OS and Android. Such an application is run via the Firefox code, but in its own process. That is, it looks and feels like a separate application. It also has its own profile (preferences, cookies, history, etc.) and its windows don't have browser chrome. For example, there is no address bar.

These two efforts are complemented by the WebAPI project, which is about making more native capabilities available via HTML5 APIs. Examples:

- WebTelephony: Answer and place phone calls.
- Camera API: Access a built-in camera for recording photos and live video. Part of WebRTC, where it is needed for video conferencing.
- WiFi Information API: Detect available WiFi networks, find out their signal strength, the name of the currently connected network, etc.
- Contacts API: Read and modify a device's native address book.

All previous solutions hosted HTML5 on top of a native layer. But there are also operating systems where HTML5 *is* the native layer (or at least very tightly integrated with it). Examples:

- 2009: The GUI layer of the Palm webOS (now Open webOS) is based on HTML.
- 2009: Google's Chrome/Chromium OS could be called a "browser operating system."
- 2011: Microsoft's Windows 8 makes JavaScript a first-class language. Several important applications are written in JavaScript, such as the native application store and the email program.

- 2011: Mozilla's Boot to Gecko project is similar to Chrome OS, but targets all mobile devices and especially mobile phones. Chrome OS mostly targets a notebook-like form factor.

HTML5 greatly profits from these endeavors because they advance the platform. If all you have is HTML5, you quickly find out what native APIs you are missing. Their replacements can then become the blueprints for cross-platform APIs.

A JavaScript Wishlist

While many interesting JavaScript-related projects are underway, there are still a few things that are missing. This section describes those things.

A good integrated development environment (IDE). The thing I miss most in JavaScript, coming from Java, is a good IDE. Several already exist, but none of them work nearly as well for JavaScript as, say, Eclipse does for Java (to be fair, they are constantly being improved and writing an IDE for a language takes time). One piece of the IDE puzzle is to automatically compute type information (e.g. which types the parameters of a function normally have), via type inference. New type inference approaches are currently being experimented with. Hopefully, class declarations will eliminate the multitude of inheritance libraries that JavaScript currently has, which would help here. After ECMAScript.next, two proposals might be accepted that would be equally helpful. First, guards allow one to annotate variable declarations, parameter declarations, function results and property definitions. The annotations enforce user-defined invariants, including types. Three examples from the proposal:

```
let x :: Number = 37;

function f(p :: String, q :: MyType) :: Boolean { ... }

let o = {a :: Number : 42, b: "b"};
```

Second, multiple inheritance is another type system mechanism that is currently implemented by libraries in JavaScript. It comes in many variations, under names such as mixins and traits. There is a proposal to add a version of traits to ECMAScript. Roughly, this proposal lets you assemble a "class" from "class fragments" (traits).

The thing to keep in mind is that we shouldn't just apply the ideas of current static IDEs to JavaScript; we should take JavaScript's dynamic nature into consideration. You can have an environment where you rarely restart a program, but continually evolve it. We thus need to revisit lessons learned from dynamic development environments such as Lisp on the Genera Operating System, various Smalltalk systems, and Self.

A merging of the browser platform and Node.js. Currently, the two JavaScript platforms browser and Node.js diverge in two ways: They use different module systems and their APIs are different. The former difference will be eliminated via ECMAScript.next's common module system. The latter is especially unfortunate because more and more HTML5-based applications are written for native deployment, which has more in common with a server-side environment than with a client-side environment. Node.js has the highly useful Node Package Manager (npm) and its vibrant community. Many of the npm packages would be just as useful in a browser. Yes, there are differences, but I don't see why they couldn't be abstracted over.

Facilitating the upgrade to ECMAScript.next. Handling language versions is much more complicated on the web than for traditional languages, which are explicitly installed on desktop and server systems by their administrators. Challenges include:

1. **New language versions must not break existing code:** When language implementers introduce a new JavaScript version, it is always a forced upgrade for language users. Hence, it must never break existing code. With non-web languages, that mandate is less strict because one only upgrades if one is ready. For major upgrades, it is quite common that breaking changes are introduced.
2. **New code won't run on all browsers:** When language users switch to a new version, they have no control over the version that is supported by the engines that their code will run on.

What is the best way to tackle challenge No. 1? One could prefix each piece of code with a pragma (a meta-statement) indicating the language version. The obvious advantage is that that would give one a lot of freedom to clean up the language. But there is also a considerable downside. It would be cumbersome to use, as code can appear in many places on an HTML page: code loaded from the network, code in script tags, and code in HTML attributes. In some cases, there are only small snippets of code. JavaScript programmers clearly aren't fond of this approach, as the limited adoption of ECMAScript 5's strict mode has shown. Furthermore, it would split JavaScript into two different dialects (say, ECMAScript 5 and ECMAScript 6) that would further diverge over time.

Instead, TC39 took an approach called "One JavaScript." It introduces modules as a new syntactic context where one can use a cleaned-up version of JavaScript. Those clean-ups are breaking changes, but they are minor: some old features are dropped; some previously legal (but questionable) practices cause errors. Apart from that, all new features only add to the language and can be used everywhere. As a result, code that only uses new features and best

practices is easy to move between contexts. And old code (in old contexts) won't break.

Challenge No. 2 remains. Manually maintaining two code bases, one for ECMAScript.next engines, one for older engines, is not an option. Sometimes one can shim (implement) new APIs on older versions. Kris Kowal's es5-shim is a good example of a shim: It enables much of ECMAScript 5's functionality on older browsers. However, with many interesting ECMAScript.next features, shimming is impossible. The only option is to compile ECMAScript.next code to an older ECMAScript version, probably ECMAScript 3.

Then one has to decide whether to compile on the fly (in the browser) or statically (during development time). Google's Traceur is a compiler that supports the former approach. Some of the tools and techniques pioneered by CoffeeScript might help with the latter approach.

A better standard library. Compared to other dynamic languages, such as Python, JavaScript's standard library leaves much to be desired. TC39 members emphasize that they are not library designers. They see their job as providing the foundations for good library design. After good libraries have emerged, those can be standardized. In the words of Allen Wirfs-Brock:

> "We will freeze innovation by prematurely making functionality a standardized immutable part of the platform." [Because one has less freedom to evolve libraries in a web programming language.]

What makes adding a library to ECMAScript.next difficult is that it has to both be proven and make use of new features (where appropriate). Many libraries exist that seem good candidates for being standardized. Two examples:

- Underscore.js is a popular library with various utility functions for objects, arrays, etc.
- XRegExp provides many improvements to JavaScript regular expressions. While quasi-literals would make that library easier to use, its improvements should really be added to `RegExp` and regular expression literals.

Alas, for data structures, there is no comprehensive library in sight. Lastly, some ECMAScript APIs are standardized separately from ECMA-262. That allows such APIs to be added to current JavaScript engines, without having to wait for ECMAScript.next. One example is the ECMAScript Internationalization API:

> The ECMAScript Internationalization API Specification supports collation (string comparison), number formatting, and date and time formatting, and lets applications choose the language and tailor the functionality to their needs.

Conclusion

JavaScript is a versatile and flexible language. It is constantly being used in new areas, due to its appeal as a language that is both widely known and completely open (open standard, open implementations). I am still seeing much hostility toward JavaScript, mainly due to two reasons: First, some concepts (such as prototypal inheritance) are unusual. Second, it has quirks. In both cases, the answer is educating people. They need to keep an open mind about unusual features and learn patterns to work around the quirks. Prime example: simulating block-scoping via immediately-invoked function expressions. The current version, ECMAScript 5, will stay with us for a while. Its current market share is roughly 50%. Wirfs-Brock estimates [10 on page 37] that it will become the baseline for JavaScript applications by mid 2013 and remain so for 5 years.

ECMAScript.next, the next version of JavaScript, takes on the difficult task of fixing the quirks and introducing new features, while remaining backward compatible. Therefore, there will always be some rough edges, but we'll still get a powerful and much less quirky language. Hence, these are good times to be or become a JavaScript programmer. The ecosystem is already thriving, yet barely getting started.

References

The following are references and sources for this article:

- [1] "The A-Z of Programming Languages: JavaScript" by Naomi Hamilton for Computerworld.
- [2] "Harmony", ECMAScript Wiki.
- [3] Draft Specification for ECMAScript.next (Ecma-262 Edition 6)
- [4] Firefox bug 749786 — prototype int64/uint64 value objects, with operators and literal syntax
- [5] "Using web workers", Mozilla Development Network.
- [6] "WebCL — Heterogeneous parallel computing in HTML5 web browsers", The Khronos Group Inc.
- [7] OpenCL Programming Guide for Mac OS X: OpenCL Overview
- [8] Project "River Trail" by Intel Labs
- [9] "The JavaScript World Domination Plan at 16 Years", Brendan Eich. Presentation on InfoQ.

- [10] "JavaScript Today and Tomorrow: Evolving the Ambient Language of the Ambient Computing Era", Allen Wirfs-Brock. Presentation on InfoQ.

About the Author

Dr. Axel Rauschmayer is a consultant and trainer for JavaScript, web technologies, and information management. He has been programming since 1985, developing web applications since 1995, and held his first talk on Ajax in 2006. In 1999, he was technical manager at an Internet startup that later expanded internationally.