WIKIPEDIA

JavaScript

JavaScript (/ˈdʒɑːvəskrɪpt/),[10] often abbreviated **JS**, is a programming language that is one of the core technologies of the World Wide Web, alongside HTML and CSS.[11] As of 2022, 98% of websites use JavaScript on the client side for webpage behavior,[12] often incorporating third-party libraries.[13] All major web browsers have a dedicated JavaScript engine to execute the code on users' devices.

JavaScript is a high-level, often just-in-time compiled language that conforms to the ECMAScript standard.[14] It has dynamic typing, prototype-based object-orientation, and first-class functions. It is multi-paradigm, supporting event-driven, functional, and imperative programming styles. It has application programming interfaces (APIs) for working with text, dates, regular expressions, standard data structures, and the Document Object Model (DOM).

The ECMAScript standard does not include any input/output (I/O), such as networking, storage, or graphics facilities. In practice, the web browser or other runtime system provides JavaScript APIs for I/O.

JavaScript engines were originally used only in web browsers, but are now core components of some servers and a variety of applications. The most popular runtime system for this usage is Node.js.

Although Java and JavaScript are similar in name, syntax, and respective standard libraries, the two languages are distinct and differ greatly in design.

JavaScript

Screenshot of JavaScript source code

Paradigm	Multi-paradigm: event-driven, functional, imperative, procedural, object-oriented programming
Designed by	Brendan Eich of Netscape initially; others have also contributed to the ECMAScript standard
First appeared	December 4, 1995[1]
Stable release	ECMAScript 2021[2] / June 2021
Preview release	ECMAScript 2022[3] / 22 July 2021
Typing discipline	Dynamic, weak, duck
Filename extensions	.js • .cjs • .mjs[4]
Website	www.ecma-international.org/publications-and-standards/standards/ecma-262/ (http://www.ecma-international.org/publications-and-stan

Contents

dards/standards/ecma-262/)

Major implementations
V8, JavaScriptCore, SpiderMonkey, Chakra
Influenced by
Java,[5][6] Scheme,[6] Self,[7] AWK,[8] HyperTalk[9]
Influenced
ActionScript, AssemblyScript, CoffeeScript, Dart, Haxe, JS++, Objective-J, Opa, TypeScript
JavaScript at Wikibooks

History

Creation at Netscape

The first web browser with a graphical user interface, Mosaic, was released in 1993. Accessible to non-technical people, it played a prominent role in the rapid growth of the nascent World Wide Web.[15] The lead developers of Mosaic then founded the Netscape corporation, which released a more polished browser, Netscape Navigator, in 1994. This quickly became the most-used.[16][17]

During these formative years of the Web, web pages could only be static, lacking the capability for dynamic behavior after the page was loaded in the browser. There was a desire in the flourishing web development scene to remove this limitation, so in 1995, Netscape decided to add a scripting language to Navigator. They pursued two routes to achieve this: collaborating with Sun Microsystems to embed the Java programming language, while also hiring Brendan Eich to embed the Scheme language.[6]

Netscape management soon decided that the best option was for Eich to devise a new language, with syntax similar to Java and less like Scheme or other extant scripting languages.[5][6] Although the new language and its interpreter implementation were called LiveScript when first shipped as part of a Navigator beta in September 1995, the name was changed to JavaScript for the official release in December.[6][1][18]

The choice of the JavaScript name has caused confusion, implying that it is directly related to Java. At the time, the dot-com boom had begun and Java was the hot new language, so Eich considered the JavaScript name a marketing ploy by Netscape.[19]

Adoption by Microsoft

Microsoft debuted Internet Explorer in 1995, leading to a browser war with Netscape. On the JavaScript front, Microsoft reverse-engineered the Navigator interpreter to create its own, called JScript.[20]

JScript was first released in 1996, alongside initial support for CSS and extensions to HTML. Each of these implementations was noticeably different from their counterparts in Navigator.[21][22] These differences made it difficult for developers to make their websites work well in both browsers, leading to widespread use of "best viewed in Netscape" and "best viewed in Internet Explorer" logos for several years.[21][23]

The rise of JScript

In November 1996, Netscape submitted JavaScript to Ecma International, as the starting point for a standard specification that all browser vendors could conform to. This led to the official release of the first ECMAScript language specification in June 1997.

The standards process continued for a few years, with the release of ECMAScript 2 in June 1998 and ECMAScript 3 in December 1999. Work on ECMAScript 4 began in 2000.[20]

Meanwhile, Microsoft gained an increasingly dominant position in the browser market. By the early 2000s, Internet Explorer's market share reached 95%.[24] This meant that JScript became the de facto standard for client-side scripting on the Web.

Microsoft initially participated in the standards process and implemented some proposals in its JScript language, but eventually it stopped collaborating on Ecma work. Thus ECMAScript 4 was mothballed.

Growth and standardization

During the period of Internet Explorer dominance in the early 2000s, client-side scripting was stagnant. This started to change in 2004, when the successor of Netscape, Mozilla, released the Firefox browser. Firefox was well received by many, taking significant market share from Internet Explorer.[25]

In 2005, Mozilla joined ECMA International, and work started on the ECMAScript for XML (E4X) standard. This led to Mozilla working jointly with Macromedia (later acquired by Adobe Systems), who were implementing E4X in their ActionScript 3 language, which was based on an ECMAScript 4 draft. The goal became standardizing ActionScript 3 as the new ECMAScript 4. To this end, Adobe Systems released the Tamarin implementation as an open source project. However, Tamarin and ActionScript 3 were too different from established client-side scripting, and without cooperation from Microsoft, ECMAScript 4 never reached fruition.

Meanwhile, very important developments were occurring in open-source communities not affiliated with ECMA work. In 2005, Jesse James Garrett released a white paper in which he coined the term Ajax and described a set of technologies, of which JavaScript was the backbone, to create web applications where data can be loaded in the background, avoiding the need for full page reloads. This sparked a renaissance period of JavaScript, spearheaded by open-source libraries and the communities that formed around them. Many new libraries were created, including jQuery, Prototype, Dojo Toolkit, and MooTools.

Google debuted its Chrome browser in 2008, with the V8 JavaScript engine that was faster than its competition.[26][27] The key innovation was just-in-time compilation (JIT),[28] so other browser vendors needed to overhaul their engines for JIT.[29]

In July 2008, these disparate parties came together for a conference in Oslo. This led to the eventual agreement in early 2009 to combine all relevant work and drive the language forward. The result was the ECMAScript 5 standard, released in December 2009.

Reaching maturity

Ambitious work on the language continued for several years, culminating in an extensive collection of additions and refinements being formalized with the publication of ECMAScript 6 in 2015.[30]

The creation of Node.js in 2009 by Ryan Dahl sparked a significant increase in the usage of JavaScript outside of web browsers. Node combines the V8 engine, an event loop, and I/O APIs, thereby providing a stand-alone JavaScript runtime system.[31][32] As of 2018, Node had been used by millions of developers,[33] and npm had the most modules of any package manager in the world.[34]

The ECMAScript draft specification is currently maintained openly on GitHub, and editions are produced via regular annual snapshots.[35] Potential revisions to the language are vetted through a comprehensive proposal process.[36][37] Now, instead of edition numbers, developers check the status of upcoming features individually.[35]

The current JavaScript ecosystem has many libraries and frameworks, established programming practices, and substantial usage of JavaScript outside of web browsers. Plus, with the rise of single-page applications and other JavaScript-heavy websites, several transpilers have been created to aid the development process.[38]

Trademark

"JavaScript" is a trademark of Oracle Corporation in the United States.[39][40]

Website client-side usage

JavaScript is the dominant client-side scripting language of the Web, with 98% of all websites (mid–2022) using it for this purpose.[12] Scripts are embedded in or included from HTML documents and interact with the DOM. All major web browsers have a built-in JavaScript engine that executes the code on the user's device.

Examples of scripted behavior

- Loading new web page content without reloading the page, via Ajax or a WebSocket. For example, users of social media can send and receive messages without leaving the current page.
- Web page animations, such as fading objects in and out, resizing, and moving them.
- Playing browser games.
- Controlling the playback of streaming media.
- Generating pop-up ads.
- Validating input values of a web form before the data is sent to a web server.
- Logging data about the user's behavior then sending it to a server. The website owner can use this data for analytics, ad tracking, and personalization.
- Redirecting a user to another page.

Libraries and frameworks

Over 80% of websites use a third-party JavaScript library or web framework for their client-side scripting.[13]

jQuery is by far the most popular library, used by over 75% of websites.[13] Facebook created the React library for its website and later released it as open source; other sites, including Twitter, now use it. Likewise, the Angular framework created by Google for its websites, including YouTube and Gmail, is now an open source project used by others.[13]

In contrast, the term "Vanilla JS" has been coined for websites not using any libraries or frameworks, instead relying entirely on standard JavaScript functionality.[41]

Other usage

The use of JavaScript has expanded beyond its web browser roots. JavaScript engines are now embedded in a variety of other software systems, both for server-side website deployments and non-browser applications.

Initial attempts at promoting server-side JavaScript usage were Netscape Enterprise Server and Microsoft's Internet Information Services,[42][43] but they were small niches.[44] Server-side usage eventually started to grow in the late 2000s, with the creation of Node.js and other approaches.[44]

Electron, Cordova, React Native, and other application frameworks have been used to create many applications with behavior implemented in JavaScript. Other non-browser applications include Adobe Acrobat support for scripting PDF documents[45] and GNOME Shell extensions written in JavaScript.[46]

JavaScript has recently begun to appear in some embedded systems, usually by leveraging Node.js.[47][48][49]

Features

The following features are common to all conforming ECMAScript implementations unless explicitly specified otherwise.

Imperative and structured

JavaScript supports much of the structured programming syntax from C (e.g., `if` statements, `while` loops, `switch` statements, `do while` loops, etc.). One partial exception is scoping: originally JavaScript only had function scoping with `var`; block scoping was added in ECMAScript 2015 with the keywords `let` and `const`. Like C, JavaScript makes a distinction between expressions and statements. One syntactic difference from C is automatic semicolon insertion, which allow semicolons (which terminate statements) to be omitted.[50]

Weakly typed

JavaScript is weakly typed, which means certain types are implicitly cast depending on the operation used.[51]

- The binary `+` operator casts both operands to a string unless both operands are numbers. This is because the addition operator doubles as a concatenation operator
- The binary `-` operator always casts both operands to a number
- Both unary operators (`+`, `-`) always cast the operand to a number

Values are cast to strings like the following:[51]

- Strings are left as-is
- Numbers are converted to their string representation
- Arrays have their elements cast to strings after which they are joined by commas (`,`)
- Other objects are converted to the string `[object Object]` where `Object` is the name of the constructor of the object

Values are cast to numbers by casting to strings and then casting the strings to numbers. These processes can be modified by defining `toString` and `valueOf` functions on the prototype for string and number casting respectively.

JavaScript has received criticism for the way it implements these conversions as the complexity of the rules can be mistaken for inconsistency.[52][51] For example, when adding a number to a string, the number will be cast to a string before performing concatenation, but when subtracting a number from a string, the string is cast to a number before performing subtraction.

JavaScript type conversions

left operand	operator	right operand	result
[] (empty array)	+	[] (empty array)	"" (empty string)
[] (empty array)	+	{} (empty object)	"[object Object]" (string)
false (boolean)	+	[] (empty array)	"false" (string)
"123"(string)	+	1 (number)	"1231" (string)
"123" (string)	−	1 (number)	122 (number)
"123" (string)	−	"abc" (string)	NaN (number)

Often also mentioned is `{} + []` resulting in `0` (number). This is misleading: the `{}` is interpreted as an empty code block instead of an empty object, and the empty array is cast to a number by the remaining unary + operator. If you wrap the expression in parentheses `({} + [])` the curly brackets are interpreted as an empty object and the result of the expression is `"[object Object]"` as expected.[51]

Dynamic

Typing

JavaScript is dynamically typed like most other scripting languages. A type is associated with a value rather than an expression. For example, a variable initially bound to a number may be reassigned to a string.[53] JavaScript supports various ways to test the type of objects, including duck typing.[54]

Run-time evaluation

JavaScript includes an `eval` function that can execute statements provided as strings at run-time.

Object-orientation (prototype-based)

Prototypal inheritance in JavaScript is described by Douglas Crockford as:

> You make prototype objects, and then ... make new instances. Objects are mutable in JavaScript, so we can augment the new instances, giving them new fields and methods. These can then act as prototypes for even newer objects. We don't need classes to make lots of similar objects... Objects inherit from objects. What could be more object oriented than that?[55]

In JavaScript, an object is an associative array, augmented with a prototype (see below); each key provides the name for an object property, and there are two syntactical ways to specify such a name: dot notation (`obj.x = 10`) and bracket notation (`obj['x'] = 10`). A property may be added, rebound, or deleted at run-time. Most properties of an object (and any property that belongs to an object's prototype inheritance chain) can be enumerated using a `for...in` loop.

Prototypes

JavaScript uses prototypes where many other object-oriented languages use classes for inheritance.[56] It is possible to simulate many class-based features with prototypes in JavaScript.[57]

Functions as object constructors

Functions double as object constructors, along with their typical role. Prefixing a function call with *new* will create an instance of a prototype, inheriting properties and methods from the constructor (including properties from the `Object` prototype).[58] ECMAScript 5 offers the `Object.create` method, allowing explicit creation of an instance without automatically inheriting from the `Object` prototype (older environments can assign the prototype to `null`).[59] The constructor's `prototype` property determines the object used for the new object's internal prototype. New methods can be added by modifying the prototype of the function used as a constructor. JavaScript's built-in constructors, such as `Array` or `Object`, also have prototypes that can be modified. While it is possible to modify the `Object` prototype, it is generally considered bad practice because most objects in JavaScript will inherit methods and properties from the `Object` prototype, and they may not expect the prototype to be modified.[60]

Functions as methods

Unlike many object-oriented languages, there is no distinction between a function definition and a method definition. Rather, the distinction occurs during function calling: when a function is called as a method of an object, the function's local *this* keyword is bound to that object for that invocation.

Functional

JavaScript functions are first-class; a function is considered to be an object.[61] As such, a function may have properties and methods, such as `.call()` and `.bind()`.[62] A *nested* function is a function defined within another function. It is created each time the outer function is invoked. In addition, each nested function forms a lexical closure: the lexical scope of the outer function (including any constant, local variable, or argument value) becomes part of the internal state of each inner function object, even after execution of the outer function concludes.[63] JavaScript also supports anonymous functions.

Delegative

JavaScript supports implicit and explicit delegation.

Functions as roles (Traits and Mixins)

JavaScript natively supports various function-based implementations of Role[64] patterns like Traits[65][66] and Mixins.[67] Such a function defines additional behavior by at least one method bound to the `this` keyword within its `function` body. A Role then has to be delegated explicitly via `call` or `apply` to objects that need to feature additional behavior that is not shared via the prototype chain.

Object composition and inheritance

Whereas explicit function-based delegation does cover composition in JavaScript, implicit delegation already happens every time the prototype chain is walked in order to, e.g., find a method that might be related to but is not directly owned by an object. Once the method is found it gets called within this object's context. Thus inheritance in JavaScript is covered by a delegation automatism that is bound to the prototype property of constructor functions.

Miscellaneous

JavaScript is a zero-index language.

Run-time environment

JavaScript typically relies on a run-time environment (e.g., a web browser) to provide objects and methods by which scripts can interact with the environment (e.g., a web page DOM). These environments are single-threaded. JavaScript also relies on the run-time environment to provide the ability to include/import scripts (e.g., HTML `<script>` elements). This is not a language feature per se, but it is common in most JavaScript implementations. JavaScript processes messages from a queue one at a time. JavaScript calls a function associated with each new message, creating a call stack frame with the function's arguments and local variables. The call stack shrinks and grows based on the function's needs. When the call stack is empty upon function completion, JavaScript proceeds to the next message in the queue. This is called the event loop, described as "run to completion" because each message is fully processed before the next message is considered. However, the language's concurrency model describes the event loop as non-blocking: program input/output is performed using events and callback functions. This means, for instance, that JavaScript can process a mouse click while waiting for a database query to return information.[68]

Variadic functions

An indefinite number of parameters can be passed to a function. The function can access them through formal parameters and also through the local `arguments` object. Variadic functions can also be created by using the `bind` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind) method.

Array and object literals

Like many scripting languages, arrays and objects (associative arrays in other languages) can each be created with a succinct shortcut syntax. In fact, these literals form the basis of the JSON data format.

Regular expressions

JavaScript also supports regular expressions in a manner similar to Perl, which provide a concise and powerful syntax for text manipulation that is more sophisticated than the built-in string functions.[69]

Promises and Async/await

JavaScript supports promises and Async/await for handling asynchronous operations. A built-in Promise object provides functionality for handling promises and associating handlers with an asynchronous action's eventual result. Recently, combinator methods were introduced in the JavaScript specification, which allows developers to combine multiple JavaScript promises and do operations based on different scenarios. The methods introduced are: Promise.race, Promise.all, Promise.allSettled and Promise.any. Async/await allows an asynchronous, non-blocking function to be structured in a way similar to an ordinary synchronous function. Asynchronous, non-blocking code can be written, with minimal overhead, structured similar to traditional synchronous, blocking code.

Vendor-specific extensions

Historically, some JavaScript engines supported these non-standard features:

- conditional `catch` clauses (like Java)
- array comprehensions and generator expressions (like Python)
- concise function expressions (`function(args) expr`; this experimental syntax predated arrow functions)
- ECMAScript for XML (E4X), an extension that adds native XML support to ECMAScript (unsupported in Firefox since version 21[70])

Syntax

Simple examples

Variables in JavaScript can be defined using either the `var`,[71] `let`[72] or `const`[73] keywords. Variables defined without keywords will be defined at the global scope.

```
// Declares a function-scoped variable named `x`, and implicitly assigns the
// special value `undefined` to it. Variables without value are automatically
// set to undefined.
var x;

// Variables can be manually set to `undefined` like so
var x2 = undefined;

// Declares a block-scoped variable named `y`, and implicitly sets it to
// `undefined`. The `let` keyword was introduced in ECMAScript 2015.
let y;

// Declares a block-scoped, un-reassignable variable named `z`, and sets it to
// a string literal. The `const` keyword was also introduced in ECMAScript 2015,
// and must be explicitly assigned to.

// The keyword `const` means constant, hence the variable cannot be reassigned
// as the value is `constant`.
const z = "this value cannot be reassigned!";

// Declares a global-scoped variable and assigns 3.  This is generally considered
// bad practice, and will not work if strict mode is on.
t = 3;
```

```
// Declares a variable named `myNumber`, and assigns a number literal (the value
// `2`) to it.
let myNumber = 2;

// Reassigns `myNumber`, setting it to a string literal (the value `"foo"`).
// JavaScript is a dynamically-typed language, so this is legal.
myNumber = "foo";
```

Note the comments in the example above, all of which were preceded with two forward slashes.

There is no built-in Input/output functionality in JavaScript, instead it is provided by the run-time environment. The ECMAScript specification in edition 5.1 mentions that "there are no provisions in this specification for input of external data or output of computed results".[74] However, most runtime environments have a `console` object that can be used to print output.[75] Here is a minimalist Hello World program in JavaScript in a runtime environment with a console object:

```
console.log("Hello, World!");
```

In HTML documents, a program like this is required for an output:

```
// Text nodes can be made using the "write" method.
// This is frowned upon, as it can overwrite the document if the document is fully loaded.
document.write('foo');

// Elements can be made too. First, they have to be created in the DOM.
const myElem = document.createElement('span');

// Attributes like classes and the id can be set as well
myElem.classList.add('foo');
myElem.id = 'bar';

// For here, the attribute will look like this: <span data-attr="baz"></span>
myElem.setAttribute('data-atrr', 'baz');

// Finally append it as a child element to the <body> in the HTML
document.body.appendChild(myElem);

// Elements can be imperitavely grabbed with querySelector for one element, or querySelectorAll for multiple
elements that can be loopped with forEach
document.querySelector('.class');
document.querySelector('#id');
document.querySelector('[data-other]');
document.querySelectorAll('.multiple');
```

A simple recursive function to calculate the factorial of a natural number:

```
function factorial(n) {
    //checking the argument for legitimacy. Factorial is defined for positive integers.
    if (isNaN(n)) {
        console.error("Non-numerical argument not allowed.");
        return NaN; //the especial value: Not a Number
    }
    if (n === 0)
        return 1; // 0! = 1
    if ( n < 0)
        return undefined; //factorial of negative numbers is not defined.
    if (n % 1) {
        console.warn(`${n} will be rounded to the closest integer. For non-integers consider using gamma
function instead.`);
        n = Math.round(n);
    }
    //The above checks need not be repeated in the recursion, hence defining the actual recursive part
separately below.

    //The following line is a function expression to recursively compute the factorial. It uses the arrow syntax
introduced in ES6.
    const recursively_compute = a => a > 1 ? a * recursively_compute(a - 1) : 1; //Note the use of the ternary
operator '?'.
```

```
    return recursively_compute(n);
}

factorial(3); // returns 6
```

An anonymous function (or lambda):

```
let counter = function() {
    let count = 0;
    return function() {
        return ++count;
    }
};

let x = counter();
x(); // returns 1
x(); // returns 2
x(); // returns 3
```

This example shows that, in JavaScript, function closures capture their non-local variables by reference.

Arrow functions were first introduced in 6th Edition - ECMAScript 2015. They shorten the syntax for writing functions in JavaScript. Arrow functions are anonymous, so a variable is needed to refer to them in order to invoke them after their creation, unless surrounded by parenthesis and executed immediately.

Example of arrow function:

```
// Arrow functions let us omit the `function` keyword.
// Here `long_example` points to an anonymous function value.
const long_example = (input1, input2) => {
    console.log("Hello, World!");
    const output = input1 + input2;

    return output;
};

// If there are no braces, the arrow function simply returns the expression
// So here it's (input1 + input2)
const short_example = (input1, input2) => input1 + input2;

long_example(2, 3); // Prints "Hello, World!" and returns 5
short_example(2, 5);  // Returns 7

// If an arrow function has only one parameter, the parentheses can be removed.
const no_parentheses = input => input + 2;

no_parentheses(3); // Returns 5

// An arrow function, like other function definitions, can be executed in the same statement as they are
created.
// This is useful when writing libraries to avoid filling the global scope, and for closures.
var three = ((a, b) => a + b) (1, 2);

const generate_multiplier_function = a => (b => isNaN(b) || !b ? a : a*=b);
const five_multiples = generate_multiplier_function(5); //the supplied argument 'seeds' the expression and is
retained by a.
five_multiples(1); //returns 5
five_multiples(3); //returns 15
five_multiples(4); //returns 60
```

In JavaScript, objects are created in the same way as functions; this is known as a function object.

Object example:

```
function Ball(r) {
    this.radius = r; // the "r" argument is local to the ball object
    this.area = Math.PI * (r ** 2); // parentheses don't do anything but clarify

    // objects can contain functions ("method")
    this.show = function() {
        drawCircle(this.radius); // references another function (that draws a circle)
    };
}

let myBall = new Ball(5); // creates a new instance of the ball object with radius 5
myBall.radius++; // object properties can usually be modified from the outside
myBall.show(); // using the inherited "show" function
```

Variadic function demonstration (`arguments` is a special variable):[76]

```
function sum() {
    let x = 0;

    for (let i = 0; i < arguments.length; ++i)
        x += arguments[i];

    return x;
}

sum(1, 2); // returns 3
sum(1, 2, 3); // returns 6


// As of ES6, using the rest operator.
function sum(...args) {
    return args.reduce((a,b) => a+b);
}

sum(1, 2); // returns 3
sum(1, 2, 3); // returns 6
```

Immediately-invoked function expressions are often used to create closures. Closures allow gathering properties and methods in a namespace and making some of them private:

```
let counter = (function() {
    let i = 0; // private property

    return {   // public methods
        get: function() {
            alert(i);
        },
        set: function(value) {
            i = value;
        },
        increment: function() {
            alert(++i);
        }
    };
})(); // module

counter.get();       // shows 0
counter.set(6);
counter.increment(); // shows 7
counter.increment(); // shows 8
```

JavaScript can export and import from modules:[77]

Export example:

```
/* mymodule.js */
// This function remains private, as it is not exported
let sum = (a, b) => {
```

```
    return a + b;
}

// Export variables
export let name = 'Alice';
export let age = 23;

// Export named functions
export function add(num1, num2) {
    return num1 + num2;
}

// Export class
export class Multiplication {
    constructor(num1, num2) {
        this.num1 = num1;
        this.num2 = num2;
    }

    add() {
        return sum(this.num1, this.num2);
    }
}
```

Import example:

```
// Import one property
import { add } from './mymodule.js';
console.log(add(1, 2));
//> 3

// Import multiple properties
import { name, age } from './mymodule.js';
console.log(name, age);
//> "Alice", 23

// Import all properties from a module
import * from './module.js'
console.log(name, age);
//> "Alice", 23
console.log(add(1,2));
//> 3
```

More advanced example

This sample code displays various JavaScript features.

```
/* Finds the lowest common multiple (LCM) of two numbers */
function LCMCalculator(x, y) { // constructor function
    if (isNaN(x*y)) throw new TypeError("Non-numeric arguments not allowed.");
    const checkInt = function(x) { // inner function
        if (x % 1 !== 0)
            throw new TypeError(x + "is not an integer");

        return x;
    };

    this.a = checkInt(x)
    //   semicolons   ^^^^  are optional, a newline is enough
    this.b = checkInt(y);
}
// The prototype of object instances created by a constructor is
// that constructor's "prototype" property.
LCMCalculator.prototype = { // object literal
    constructor: LCMCalculator, // when reassigning a prototype, set the constructor property appropriately
    gcd: function() { // method that calculates the greatest common divisor
        // Euclidean algorithm:
        let a = Math.abs(this.a), b = Math.abs(this.b), t;

        if (a < b) {
```

```
            // swap variables
            // t = b; b = a; a = t;
            [a, b] = [b, a]; // swap using destructuring assignment (ES6)
        }

        while (b !== 0) {
            t = b;
            b = a % b;
            a = t;
        }

        // Only need to calculate GCD once, so "redefine" this method.
        // (Actually not redefinition—it's defined on the instance itself,
        // so that this.gcd refers to this "redefinition" instead of LCMCalculator.prototype.gcd.
        // Note that this leads to a wrong result if the LCMCalculator object members "a" and/or "b" are altered
afterwards.)
        // Also, 'gcd' === "gcd", this['gcd'] === this.gcd
        this['gcd'] = function() {
            return a;
        };

        return a;
    },

    // Object property names can be specified by strings delimited by double (") or single (') quotes.
    "lcm": function() {
        // Variable names do not collide with object properties, e.g., |lcm| is not |this.lcm|.
        // not using |this.a*this.b| to avoid FP precision issues
        let lcm = this.a / this.gcd() * this.b;

        // Only need to calculate lcm once, so "redefine" this method.
        this.lcm = function() {
            return lcm;
        };

        return lcm;
    },

    // Methods can also be declared using es6 syntax
    toString() {
        // Using both es6 template literals and the (+) operator to concatenate values
        return `LCMCalculator: a = ${this.a}, b = ` + this.b;
    }
};

// Define generic output function; this implementation only works for Web browsers
function output(x) {
    document.body.appendChild(document.createTextNode(x));
    document.body.appendChild(document.createElement('br'));
}

// Note: Array's map() and forEach() are defined in JavaScript 1.6.
// They are used here to demonstrate JavaScript's inherent functional nature.
[
    [25, 55],
    [21, 56],
    [22, 58],
    [28, 56]
].map(function(pair) { // array literal + mapping function
    return new LCMCalculator(pair[0], pair[1]);
}).sort((a, b) => a.lcm() - b.lcm()) // sort with this comparative function; => is a shorthand form of a
function, called "arrow function"
    .forEach(printResult);

function printResult(obj) {
    output(obj + ", gcd = " + obj.gcd() + ", lcm = " + obj.lcm());
}
```

The following output should be displayed in the browser window.

```
LCMCalculator: a = 28, b = 56, gcd = 28, lcm = 56
LCMCalculator: a = 21, b = 56, gcd = 7, lcm = 168
LCMCalculator: a = 25, b = 55, gcd = 5, lcm = 275
LCMCalculator: a = 22, b = 58, gcd = 2, lcm = 638
```

Security

JavaScript and the DOM provide the potential for malicious authors to deliver scripts to run on a client computer via the Web. Browser authors minimize this risk using two restrictions. First, scripts run in a sandbox in which they can only perform Web-related actions, not general-purpose programming tasks like creating files. Second, scripts are constrained by the same-origin policy: scripts from one Web site do not have access to information such as usernames, passwords, or cookies sent to another site. Most JavaScript-related security bugs are breaches of either the same origin policy or the sandbox.

There are subsets of general JavaScript—ADsafe, Secure ECMAScript (SES)—that provide greater levels of security, especially on code created by third parties (such as advertisements).[78][79] Closure Toolkit is another project for safe embedding and isolation of third-party JavaScript and HTML.[80]

Content Security Policy is the main intended method of ensuring that only trusted code is executed on a Web page.

Cross-site vulnerabilities

A common JavaScript-related security problem is cross-site scripting (XSS), a violation of the same-origin policy. XSS vulnerabilities occur when an attacker can cause a target Web site, such as an online banking website, to include a malicious script in the webpage presented to a victim. The script in this example can then access the banking application with the privileges of the victim, potentially disclosing secret information or transferring money without the victim's authorization. A solution to XSS vulnerabilities is to use *HTML escaping* whenever displaying untrusted data.

Some browsers include partial protection against *reflected* XSS attacks, in which the attacker provides a URL including malicious script. However, even users of those browsers are vulnerable to other XSS attacks, such as those where the malicious code is stored in a database. Only correct design of Web applications on the server-side can fully prevent XSS.

XSS vulnerabilities can also occur because of implementation mistakes by browser authors.[81]

Another cross-site vulnerability is cross-site request forgery (CSRF). In CSRF, code on an attacker's site tricks the victim's browser into taking actions the user did not intend at a target site (like transferring money at a bank). When target sites rely solely on cookies for request authentication, requests originating from code on the attacker's site can carry the same valid login credentials of the initiating user. In general, the solution to CSRF is to require an authentication value in a hidden form field, and not only in the cookies, to authenticate any request that might have lasting effects. Checking the HTTP Referrer header can also help.

"JavaScript hijacking" is a type of CSRF attack in which a `<script>` tag on an attacker's site exploits a page on the victim's site that returns private information such as JSON or JavaScript. Possible solutions include:

- requiring an authentication token in the POST and GET parameters for any response that returns private information.

Misplaced trust in the client

Developers of client-server applications must recognize that untrusted clients may be under the control of attackers. The application author cannot assume that their JavaScript code will run as intended (or at all) because any secret embedded in the code could be extracted by a determined adversary. Some implications are:

- Web site authors cannot perfectly conceal how their JavaScript operates because the raw source code must be sent to the client. The code can be obfuscated, but obfuscation can be reverse-engineered.
- JavaScript form validation only provides convenience for users, not security. If a site verifies that the user agreed to its terms of service, or filters invalid characters out of fields that should only contain numbers, it must do so on the server, not only the client.
- Scripts can be selectively disabled, so JavaScript cannot be relied on to prevent operations such as right-clicking on an image to save it.[82]
- It is considered very bad practice to embed sensitive information such as passwords in JavaScript because it can be extracted by an attacker.[83]

Misplaced trust in developers

Package management systems such as npm and Bower are popular with JavaScript developers. Such systems allow a developer to easily manage their program's dependencies upon other developers' program libraries. Developers trust that the maintainers of the libraries will keep them secure and up to date, but that is not always the case. A vulnerability has emerged because of this blind trust. Relied-upon libraries can have new releases that cause bugs or vulnerabilities to appear in all programs that rely upon the libraries. Inversely, a library can go unpatched with known vulnerabilities out in the wild. In a study done looking over a sample of 133,000 websites, researchers found 37% of the websites included a library with at least one known vulnerability.[84] "The median lag between the oldest library version used on each website and the newest available version of that library is 1,177 days in ALEXA, and development of some libraries still in active use ceased years ago."[84] Another possibility is that the maintainer of a library may remove the library entirely. This occurred in March 2016 when Azer Koçulu removed his repository from npm. This caused tens of thousands of programs and websites depending upon his libraries to break.[85][86]

Browser and plugin coding errors

JavaScript provides an interface to a wide range of browser capabilities, some of which may have flaws such as buffer overflows. These flaws can allow attackers to write scripts that would run any code they wish on the user's system. This code is not by any means limited to another JavaScript application. For example, a buffer overrun exploit can allow an attacker to gain access to the operating system's API with superuser privileges.

These flaws have affected major browsers including Firefox,[87] Internet Explorer,[88] and Safari.[89]

Plugins, such as video players, Adobe Flash, and the wide range of ActiveX controls enabled by default in Microsoft Internet Explorer, may also have flaws exploitable via JavaScript (such flaws have been exploited in the past).[90][91]

In Windows Vista, Microsoft has attempted to contain the risks of bugs such as buffer overflows by running the Internet Explorer process with limited privileges.[92] Google Chrome similarly confines its page renderers to their own "sandbox".

Sandbox implementation errors

Web browsers are capable of running JavaScript outside the sandbox, with the privileges necessary to, for example, create or delete files. Such privileges are not intended to be granted to code from the Web.

Incorrectly granting privileges to JavaScript from the Web has played a role in vulnerabilities in both Internet Explorer[93] and Firefox.[94] In Windows XP Service Pack 2, Microsoft demoted JScript's privileges in Internet Explorer.[95]

Microsoft Windows allows JavaScript source files on a computer's hard drive to be launched as general-purpose, non-sandboxed programs (see: Windows Script Host). This makes JavaScript (like VBScript) a theoretically viable vector for a Trojan horse, although JavaScript Trojan horses are uncommon in practice.[96]

Hardware vulnerabilities

In 2015, a JavaScript-based proof-of-concept implementation of a rowhammer attack was described in a paper by security researchers.[97][98][99][100]

In 2017, a JavaScript-based attack via browser was demonstrated that could bypass ASLR. It's called "ASLR⊕Cache" or AnC.[101][102]

In 2018, the paper that announced the Spectre attacks against Speculative Execution in Intel and other processors included a JavaScript implementation.[103]

Development tools

Important tools have evolved with the language.

- Every major web browser has built-in web development tools, including a JavaScript debugger.
- Static program analysis tools, such as ESLint and JSLint, scan JavaScript code for conformance to a set of standards and guidelines.
- Some browsers have built-in profilers. Stand-alone profiling libraries have also been created, such as benchmark.js and jsbench.[104][105]
- Many text editors have syntax highlighting support for JavaScript code.

Related technologies

Java

A common misconception is that JavaScript is the same as Java. Both indeed have a C-like syntax (the C language being their most immediate common ancestor language). They are also typically sandboxed (when used inside a browser), and JavaScript was designed with Java's syntax and standard library in mind. In particular, all Java keywords were reserved in original JavaScript, JavaScript's standard library follows Java's naming conventions, and JavaScript's `Math` and `Date` objects are based on classes from Java 1.0.[106]

Java and JavaScript both first appeared in 1995, but Java was developed by James Gosling of Sun Microsystems and JavaScript by Brendan Eich of Netscape Communications.

The differences between the two languages are more prominent than their similarities. Java has static typing, while JavaScript's typing is dynamic. Java is loaded from compiled bytecode, while JavaScript is loaded as human-readable source code. Java's objects are class-based, while JavaScript's are prototype-based. Finally, Java did not support functional programming until Java 8, while JavaScript has done so from the beginning, being influenced by Scheme.

JSON

JSON, or JavaScript Object Notation, is a general-purpose data interchange format that is defined as a subset of JavaScript's object literal syntax.

WebAssembly

Since 2017, web browsers have supported WebAssembly, a binary format that enables a JavaScript engine to execute performance-critical portions of web page scripts close to native speed.[107] WebAssembly code runs in the same sandbox as regular JavaScript code.

asm.js is a subset of JavaScript that served as the forerunner of WebAssembly.[108]

Transpilers

JavaScript is the dominant client-side language of the Web, and many websites are script-heavy. Thus transpilers have been created to convert code written in other languages, which can aid the development process.[38]

References

1. Press release announcing JavaScript (https://web.archive.org/web/20070916144913/https://wp.netscape.com/newsref/pr/newsrelease67.html), "Netscape and Sun announce JavaScript", PR Newswire, December 4, 1995
2. "ECMAScript® 2021 language specification" (https://www.ecma-international.org/publications-and-standards/standards/?order=last-change). June 2021. Retrieved 27 July 2021.
3. https://tc39.es/ecma262/; retrieved: 27 July 2021; publication date: 22 July 2021.
4. "nodejs/node-eps" (https://github.com/nodejs/node-eps/blob/master/002-es-modules.md). *GitHub*. Archived (https://web.archive.org/web/20200829024713/https://github.com/nodejs/node-eps/blob/master/002-es-modules.md) from the original on 2020-08-29. Retrieved 2018-07-05.

5. Seibel, Peter (September 16, 2009). *Coders at Work: Reflections on the Craft of Programming* (https://books.google.com/books?id=nneBa6-mWfgC&q=The+immediate+concern+at+Netscape+was+it+must+look+like+Java.&pg=PA141). ISBN 9781430219484. Archived (https://web.archive.org/web/20201224233514/https://books.google.com/books?id=nneBa6-mWfgC&q=The+immediate+concern+at+Netscape+was+it+must+look+like+Java.&pg=PA141) from the original on December 24, 2020. Retrieved December 25, 2018. "Eich: The immediate concern at Netscape was it must look like Java."
6. "Chapter 4. How JavaScript Was Created" (https://speakingjs.com/es5/ch04.html). *speakingjs.com*. Archived (https://web.archive.org/web/20200227184037/https://speakingjs.com/es5/ch04.html) from the original on 2020-02-27. Retrieved 2017-11-21.
7. "Popularity – Brendan Eich" (https://brendaneich.com/2008/04/popularity/).
8. "Brendan Eich: An Introduction to JavaScript, JSConf 2010" (https://www.youtube.com/watch?v=1EyRscXrehw). *YouTube*. p. 22m. Archived (https://web.archive.org/web/20200829024704/https://www.youtube.com/watch?v=1EyRscXrehw) from the original on August 29, 2020. Retrieved November 25, 2019. "Eich: "function", eight letters, I was influenced by AWK."
9. Eich, Brendan (1998). "Foreword". In Goodman, Danny (ed.). *JavaScript Bible* (https://archive.org/details/javascriptbible000good) (3rd ed.). John Wiley & Sons. ISBN 0-7645-3188-3. LCCN 97078208 (https://lccn.loc.gov/97078208). OCLC 38888873 (https://www.worldcat.org/oclc/38888873). OL 712205M (https://openlibrary.org/books/OL712205M).
10. "JavaScript" (https://www.dictionary.com/browse/javascript). *Dictionary.com*. 2016-04-27. Archived (https://web.archive.org/web/20210809164119/https://www.dictionary.com/browse/javascript) from the original on August 9, 2021. Retrieved August 9, 2021.
11. Flanagan, David (18 April 2011). *JavaScript: the definitive guide* (https://www.worldcat.org/title/javascript-the-definitive-guide/oclc/686709345?referer=br&ht=edition). Beijing; Farnham: O'Reilly. p. 1. ISBN 978-1-4493-9385-4. OCLC 686709345 (https://www.worldcat.org/oclc/686709345). "JavaScript is part of the triad of technologies that all Web developers must learn: HTML to specify the content of web pages, CSS to specify the presentation of web pages, and JavaScript to specify the behavior of web pages."
12. "Usage statistics of JavaScript as client-side programming language on websites" (https://w3techs.com/technologies/details/cp-javascript/). *w3techs.com*. 2021-04-09. Archived (http://wayback.archive-it.org/all/20220213043439/https://w3techs.com/technologies/details/cp%2Djavascript) from the original on 2022-02-13. Retrieved 2021-04-09.
13. "Usage statistics of JavaScript libraries for websites" (https://w3techs.com/technologies/overview/javascript_library). *w3techs.com*. Archived (https://archive.today/20120526001910/http://w3techs.com/technologies/overview/javascript_library/all) from the original on 2012-05-26. Retrieved 2021-04-09.
14. "ECMAScript® 2020 Language Specification" (https://tc39.es/ecma262/#sec-overview). Archived (https://web.archive.org/web/20200508053013/https://tc39.es/ecma262/#sec-overview) from the original on 2020-05-08. Retrieved 2020-05-08.
15. "Bloomberg Game Changers: Marc Andreessen" (https://www.bloomberg.com/video/67758394). *Bloomberg*. Bloomberg. March 17, 2011. Archived (https://web.archive.org/web/20120516093712/https://www.bloomberg.com/video/67758394/) from the original on May 16, 2012. Retrieved December 7, 2011.

16. Enzer, Larry (August 31, 2018). "The Evolution of the Web Browsers" (https://web.archive.org/web/20180831174847/https://www.mwdwebsites.com/nj-web-design-web-browsers.html). *Monmouth Web Developers*. Archived from the original (https://www.mwdwebsites.com/nj-web-design-web-browsers.html) on August 31, 2018. Retrieved August 31, 2018.
17. Dickerson, Gordon (August 31, 2018). "Learn the History of Web Browsers" (https://washingtonindependent.com/learn-the-history-of-web-browsers/). *washingtonindependent.com*. Retrieved August 31, 2018.
18. "TechVision: Innovators of the Net: Brendan Eich and JavaScript" (https://web.archive.org/web/20080208124612/https://wp.netscape.com/comprod/columns/techvision/innovators_be.html). Archived from the original (https://wp.netscape.com/comprod/columns/techvision/innovators_be.html) on February 8, 2008.
19. Fin JS (June 17, 2016), *Brendan Eich – CEO of Brave* (https://www.youtube.com/watch?v=XOmhtfTrRxc&t=2m5s), archived (https://web.archive.org/web/20190210054957/https://www.youtube.com/watch?v=XOmhtfTrRxc&t=2m5s) from the original on February 10, 2019, retrieved February 7, 2018
20. "Chapter 5. Standardization: ECMAScript" (https://web.archive.org/web/20211101184346/http://speakingjs.com/es5/ch05.html). *speakingjs.com*. Archived from the original (https://speakingjs.com/es5/ch05.html) on 1 November 2021. Retrieved 1 November 2021.
21. Champeon, Steve (April 6, 2001). "JavaScript, How Did We Get Here?" (https://web.archive.org/web/20160719020828/https://archive.oreilly.com/pub/a/javascript/2001/04/06/js_history.html). *oreilly.com*. Archived from the original (https://archive.oreilly.com/pub/a/javascript/2001/04/06/js_history.html) on July 19, 2016. Retrieved July 16, 2016.
22. "Microsoft Internet Explorer 3.0 Beta Now Available" (https://news.microsoft.com/1996/05/29/microsoft-internet-explorer-3-0-beta-now-available/). *microsoft.com*. Microsoft. May 29, 1996. Archived (https://web.archive.org/web/20201124154053/https://news.microsoft.com/1996/05/29/microsoft-internet-explorer-3-0-beta-now-available/) from the original on November 24, 2020. Retrieved July 16, 2016.
23. McCracken, Harry (September 16, 2010). "The Unwelcome Return of "Best Viewed with Internet Explorer" " (https://www.technologizer.com/2010/09/16/the-unwelcome-return-of-best-viewed-with-internet-explorer/). *technologizer.com*. Archived (https://web.archive.org/web/20180623192402/https://www.technologizer.com/2010/09/16/the-unwelcome-return-of-best-viewed-with-internet-explorer/) from the original on June 23, 2018. Retrieved July 16, 2016.
24. Baker, Loren (November 24, 2004). "Mozilla Firefox Internet Browser Market Share Gains to 7.4%" (https://www.searchenginejournal.com/mozilla-firefox-internet-browser-market-share-gains-to-74/1082/). *Search Engine Journal*. Archived (https://web.archive.org/web/20210507013607/https://www.searchenginejournal.com/mozilla-firefox-internet-browser-market-share-gains-to-74/1082/) from the original on May 7, 2021. Retrieved May 8, 2021.
25. Weber, Tim (May 9, 2005). "The assault on software giant Microsoft" (https://web.archive.org/web/20170925233936/https://news.bbc.co.uk/2/hi/business/4508897.stm). *BBC News*. Archived from the original (https://news.bbc.co.uk/2/hi/business/4508897.stm) on September 25, 2017.

26. "Big browser comparison test: Internet Explorer vs. Firefox, Opera, Safari and Chrome" (https://www.pcgameshardware.com/aid,687738/Big-browser-comparison-test-Internet-Explorer-vs-Firefox-Opera-Safari-and-Chrome-Update-Firefox-35-Final/Practice/). *PC Games Hardware*. Computec Media AG. 3 July 2009. Archived (https://web.archive.org/web/20120502043027/http://www.pcgameshardware.com/aid,687738/Big-browser-comparison-test-Internet-Explorer-vs-Firefox-Opera-Safari-and-Chrome-Update-Firefox-35-Final/Practice/) from the original on May 2, 2012. Retrieved June 28, 2010.
27. Purdy, Kevin (June 11, 2009). "Lifehacker Speed Tests: Safari 4, Chrome 2" (https://lifehacker.com/lifehacker-speed-tests-safari-4-chrome-2-and-more-5286869). *Lifehacker*. Archived (https://web.archive.org/web/20210414095403/https://lifehacker.com/lifehacker-speed-tests-safari-4-chrome-2-and-more-5286869) from the original on April 14, 2021. Retrieved May 8, 2021.
28. "TraceMonkey: JavaScript Lightspeed, Brendan Eich's Blog" (https://brendaneich.com/2008/08/tracemonkey-javascript-lightspeed/). Archived (https://web.archive.org/web/20151204091540/https://brendaneich.com/2008/08/tracemonkey-javascript-lightspeed/) from the original on December 4, 2015. Retrieved July 22, 2020.
29. "Mozilla asks, 'Are we fast yet?' " (https://www.wired.com/2010/09/mozilla-asks-are-we-fast-yet/). *Wired*. Archived (https://web.archive.org/web/20180622213244/https://www.wired.com/2010/09/mozilla-asks-are-we-fast-yet/) from the original on June 22, 2018. Retrieved January 18, 2019.
30. "ECMAScript 6: New Features: Overview and Comparison" (https://es6-features.org/). *es6-features.org*. Archived (https://web.archive.org/web/20180318064130/https://es6-features.org/) from the original on March 18, 2018. Retrieved March 19, 2018.
31. Professional Node.js: Building JavaScript Based Scalable Software (https://books.google.com/books?id=ZH6bpbcrlvYC&printsec=frontcover&dq=nodejs&hl=en&sa=X#v=onepage&q=nodejs&f=false) Archived (https://web.archive.org/web/20170324021220/https://books.google.com/books?id=ZH6bpbcrlvYC&printsec=frontcover&dq=nodejs&hl=en&sa=X#v=onepage&q=nodejs&f=false) 2017-03-24 at the Wayback Machine, John Wiley & Sons, 01-Oct-2012
32. Sams Teach Yourself Node.js in 24 Hours (https://books.google.com/books?id=KGt-FxUEj48C&pg=PT24&dq=nodejs&hl=en&sa=X#v=onepage&q=nodejs&f=false) Archived (https://web.archive.org/web/20170323192039/https://books.google.com/books?id=KGt-FxUEj48C&pg=PT24&dq=nodejs&hl=en&sa=X#v=onepage&q=nodejs&f=false) 2017-03-23 at the Wayback Machine, Sams Publishing, 05-Sep-2012
33. Lawton, George (19 July 2018). "The secret history behind the success of npm and Node" (https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/The-secret-history-behind-the-success-of-npm-and-Node). *TheServerSide*. Archived (https://web.archive.org/web/20210802165613/https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/The-secret-history-behind-the-success-of-npm-and-Node) from the original on 2 August 2021. Retrieved 2 August 2021.
34. Brown, Paul (13 January 2017). "State of the Union: npm" (https://www.linux.com/news/state-union-npm/). *Linux.com*. Archived (https://web.archive.org/web/20210802165614/https://www.linux.com/news/state-union-npm/) from the original on 2 August 2021. Retrieved 2 August 2021.
35. Branscombe, Mary (2016-05-04). "JavaScript Standard Moves to Yearly Release Schedule; Here is What's New for ES16" (https://thenewstack.io/whats-new-es2016/). *The New Stack*. Archived (https://web.archive.org/web/20210116181757/https://thenewstack.io/whats-new-es2016/) from the original on 2021-01-16. Retrieved 2021-01-15.

36. "The TC39 Process" (https://tc39.es/process-document/). *tc39.es*. Ecma International. Archived (https://web.archive.org/web/20210207105535/https://tc39.es/process-document/) from the original on 2021-02-07. Retrieved 2021-01-15.
37. "ECMAScript proposals" (https://github.com/tc39/proposals/blob/master/README.md). TC39. Archived (https://web.archive.org/web/20201204221147/https://github.com/tc39/proposals/blob/master/README.md) from the original on 2020-12-04. Retrieved 2021-01-15.
38. Ashkenas, Jeremy. "List of languages that compile to JS" (https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS). *GitHub*. Archived (https://web.archive.org/web/20200131233044/https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS) from the original on January 31, 2020. Retrieved February 6, 2020.
39. "U.S. Trademark Serial No. 75026640" (https://tsdr.uspto.gov/#caseNumber=75026640&caseType=SERIAL_NO&searchType=statusSearch). *uspto.gov*. United States Patent and Trademark Office. 1997-05-06. Archived (https://web.archive.org/web/20210713022850/https://tsdr.uspto.gov/#caseNumber=75026640&caseType=SERIAL_NO&searchType=statusSearch) from the original on 2021-07-13. Retrieved 2021-05-08.
40. "Legal Notices" (https://www.oracle.com/legal/trademarks.html). *oracle.com*. Oracle Corporation. Archived (https://web.archive.org/web/20210605142505/https://www.oracle.com/legal/trademarks.html) from the original on 2021-06-05. Retrieved 2021-05-08.
41. "Vanilla JS" (https://vanilla-js.com/). *vanilla-js.com*. 2020-06-16. Archived (https://web.archive.org/web/20200616052335/https://vanilla-js.com/) from the original on June 16, 2020. Retrieved June 17, 2020.
42. "Server-Side JavaScript Guide" (https://docs.oracle.com/cd/E19957-01/816-6411-10/contents.htm). *oracle.com*. Oracle Corporation. December 11, 1998. Archived (https://web.archive.org/web/20210311173120/https://docs.oracle.com/cd/E19957-01/816-6411-10/contents.htm) from the original on March 11, 2021. Retrieved May 8, 2021.
43. Clinick, Andrew (July 14, 2000). "Introducing JScript .NET" (https://msdn.microsoft.com/en-us/library/ms974588.aspx). *Microsoft Developer Network*. Microsoft. Archived (https://web.archive.org/web/20171110201649/https://msdn.microsoft.com/en-us/library/ms974588.aspx) from the original on November 10, 2017. Retrieved April 10, 2018. "[S]ince the 1996 introduction of JScript version 1.0 ... we've been seeing a steady increase in the usage of JScript on the server—particularly in Active Server Pages (ASP)"
44. Mahemoff, Michael (December 17, 2009). "Server-Side JavaScript, Back with a Vengeance" (https://readwrite.com/2009/12/17/server-side_javascript_back_with_a_vengeance/). *readwrite.com*. Archived (https://web.archive.org/web/20160617030219/https://readwrite.com/2009/12/17/server-side_javascript_back_with_a_vengeance/) from the original on June 17, 2016. Retrieved July 16, 2016.
45. "JavaScript for Acrobat" (https://www.adobe.com/devnet/acrobat/javascript.html). *adobe.com*. 2009-08-07. Archived (https://web.archive.org/web/20090807065130/https://www.adobe.com/devnet/acrobat/javascript.html) from the original on August 7, 2009. Retrieved August 18, 2009.
46. treitter (2013-02-02). "Answering the question: "How do I develop an app for GNOME?" " (https://treitter.livejournal.com/14871.html). *livejournal.com*. Archived (https://web.archive.org/web/20130211032900/https://treitter.livejournal.com/14871.html) from the original on 2013-02-11. Retrieved 2013-02-07.

47. "Tessel 2... Leverage all the libraries of Node.JS to create useful devices in minutes with Tessel" (https://tessel.io/). *tessel.io*. Archived (https://web.archive.org/web/20210526212559/https://tessel.io/) from the original on 2021-05-26. Retrieved 2021-05-08.
48. "Node.js Raspberry Pi GPIO Introduction" (https://www.w3schools.com/nodejs/nodejs_raspberrypi_gpio_intro.asp). *w3schools.com*. Archived (https://web.archive.org/web/20210813192938/https://www.w3schools.com/nodejs/nodejs_raspberrypi_gpio_intro.asp) from the original on 2021-08-13. Retrieved 2020-05-03.
49. "Espruino – JavaScript for Microcontrollers" (https://www.espruino.com/). *espruino.com*. Archived (https://web.archive.org/web/20200501010722/https://www.espruino.com/) from the original on 2020-05-01. Retrieved 2020-05-03.
50. Flanagan, David (August 17, 2006). *JavaScript: The Definitive Guide: The Definitive Guide* (https://books.google.com/books?id=2weL0iAfrEMC). "O'Reilly Media, Inc.". p. 16. ISBN 978-0-596-55447-7. Archived (https://web.archive.org/web/20200801065235/https://books.google.com/books?id=2weL0iAfrEMC) from the original on August 1, 2020. Retrieved March 29, 2019.
51. Korolev, Mikhail (2019-03-01). "JavaScript quirks in one image from the Internet" (https://dev.to/mkrl/javascript-quirks-in-one-image-from-the-internet-52m7). *The DEV Community*. Archived (https://web.archive.org/web/20191028204723/https://dev.to/mkrl/javascript-quirks-in-one-image-from-the-internet-52m7) from the original on October 28, 2019. Retrieved October 28, 2019.
52. "Wat" (https://www.destroyallsoftware.com/talks/wat). *www.destroyallsoftware.com*. 2012. Archived (https://web.archive.org/web/20191028204723/https://www.destroyallsoftware.com/talks/wat) from the original on October 28, 2019. Retrieved October 28, 2019.
53. "JavaScript data types and data structures – JavaScript | MDN" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures). *Developer.mozilla.org*. February 16, 2017. Archived (https://web.archive.org/web/20170314230542/https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures) from the original on March 14, 2017. Retrieved February 24, 2017.
54. Flanagan 2006, pp. 176–178.
55. Crockford, Douglas. "Prototypal Inheritance in JavaScript" (https://javascript.crockford.com/prototypal.html). Archived (https://web.archive.org/web/20130813163035/https://javascript.crockford.com/prototypal.html) from the original on 13 August 2013. Retrieved 20 August 2013.
56. "Inheritance and the prototype chain" (https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Inheritance_and_the_prototype_chain). *Mozilla Developer Network*. Mozilla. Archived (https://web.archive.org/web/20130425144207/https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Inheritance_and_the_prototype_chain) from the original on April 25, 2013. Retrieved April 6, 2013.
57. Herman, David (2013). *Effective JavaScript*. Addison-Wesley. p. 83. ISBN 978-0-321-81218-6.
58. Haverbeke, Marijn (2011). *Eloquent JavaScript*. No Starch Press. pp. 95–97. ISBN 978-1-59327-282-1.
59. Katz, Yehuda (12 August 2011). "Understanding "Prototypes" in JavaScript" (https://yehudakatz.com/2011/08/12/understanding-prototypes-in-javascript/). Archived (https://web.archive.org/web/20130405154842/https://yehudakatz.com/2011/08/12/understanding-prototypes-in-javascript/) from the original on 5 April 2013. Retrieved April 6, 2013.
60. Herman, David (2013). *Effective JavaScript*. Addison-Wesley. pp. 125–127. ISBN 978-0-321-81218-6.

61. "Function – JavaScript" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function). *MDN Web Docs*. Retrieved 2021-10-30.
62. "Properties of the Function Object" (https://es5.github.com/#x15.3.4-toc). Es5.github.com. Archived (https://web.archive.org/web/20130128185825/https://es5.github.com/#x15.3.4-toc) from the original on January 28, 2013. Retrieved May 26, 2013.
63. Flanagan 2006, p. 141.
64. The many talents of JavaScript for generalizing Role-Oriented Programming approaches like Traits and Mixins (https://peterseliger.blogspot.de/2014/04/the-many-talents-of-javascript.html#the-many-talents-of-javascript-for-generalizing-role-oriented-programming-approaches-like-traits-and-mixins) Archived (https://web.archive.org/web/20171005050713/https://peterseliger.blogspot.de/2014/04/the-many-talents-of-javascript.html#the-many-talents-of-javascript-for-generalizing-role-oriented-programming-approaches-like-traits-and-mixins) 2017-10-05 at the Wayback Machine, Peterseliger.blogpsot.de, April 11, 2014.
65. Traits for JavaScript (https://soft.vub.ac.be/~tvcutsem/traitsjs/) Archived (https://web.archive.org/web/20140724052500/https://soft.vub.ac.be/~tvcutsem/traitsjs/) 2014-07-24 at the Wayback Machine, 2010.
66. "Home | CocktailJS" (https://cocktailjs.github.io/). *Cocktailjs.github.io*. Archived (https://web.archive.org/web/20170204083608/https://cocktailjs.github.io/) from the original on February 4, 2017. Retrieved February 24, 2017.
67. Angus Croll, A fresh look at JavaScript Mixins (https://javascriptweblog.wordpress.com/2011/05/31/a-fresh-look-at-javascript-mixins/) Archived (https://web.archive.org/web/20200415004603/https://javascriptweblog.wordpress.com/2011/05/31/a-fresh-look-at-javascript-mixins/) 2020-04-15 at the Wayback Machine, published May 31, 2011.
68. "Concurrency model and Event Loop" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop). *Mozilla Developer Network*. Archived (https://web.archive.org/web/20150905045241/https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop) from the original on September 5, 2015. Retrieved August 28, 2015.
69. Haverbeke, Marijn (2011). *Eloquent JavaScript*. No Starch Press. pp. 139–149. ISBN 978-1-59327-282-1.
70. "E4X – Archive of obsolete content | MDN" (https://web.archive.org/web/20140724100129/https://developer.mozilla.org/en-US/docs/Archive/Web/E4X). *Mozilla Developer Network*. Mozilla Foundation. February 14, 2014. Archived from the original (https://developer.mozilla.org/en-US/docs/Archive/Web/E4X) on July 24, 2014. Retrieved July 13, 2014.
71. "var – JavaScript – MDN" (https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Statements/var). The Mozilla Developer Network. Archived (https://web.archive.org/web/20121223162713/https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Statements/var) from the original on December 23, 2012. Retrieved December 22, 2012.
72. "let" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let). *MDN web docs*. Mozilla. Archived (https://web.archive.org/web/20190528140803/https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let) from the original on May 28, 2019. Retrieved June 27, 2018.
73. "const" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const). *MDN web docs*. Mozilla. Archived (https://web.archive.org/web/20180628044054/https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const) from the original on June 28, 2018. Retrieved June 27, 2018.

74. "ECMAScript Language Specification – ECMA-262 Edition 5.1" (https://www.ecma-international.org/ecma-262/5.1/#sec-4). Ecma International. Archived (https://web.archive.org/web/20121126044218/https://ecma-international.org/ecma-262/5.1/#sec-4) from the original on November 26, 2012. Retrieved December 22, 2012.
75. "console" (https://developer.mozilla.org/en-US/docs/DOM/console). *Mozilla Developer Network*. Mozilla. Archived (https://web.archive.org/web/20130228112150/https://developer.mozilla.org/en-US/docs/DOM/console) from the original on February 28, 2013. Retrieved April 6, 2013.
76. "arguments" (https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Functions_and_function_scope/arguments). *Mozilla Developer Network*. Mozilla. Archived (https://web.archive.org/web/20130413230225/https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Functions_and_function_scope/arguments) from the original on April 13, 2013. Retrieved April 6, 2013.
77. "JavaScript modules" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules). *MDN Web Docs*. Mozilla. Archived (https://web.archive.org/web/20220717083604/https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules) from the original on 17 July 2022. Retrieved 28 July 2022.
78. "Making JavaScript Safe for Advertising" (https://www.adsafe.org/). ADsafe. Archived (https://web.archive.org/web/20210706153324/https://www.adsafe.org/) from the original on 2021-07-06. Retrieved 2021-05-08.
79. "Secure ECMA Script (SES)" (https://code.google.com/p/es-lab/wiki/SecureEcmaScript). Archived (https://web.archive.org/web/20130515073412/https://code.google.com/p/es-lab/wiki/SecureEcmaScript) from the original on May 15, 2013. Retrieved May 26, 2013.
80. "Google Caja Project" (https://developers.google.com/caja/). *Google*. Archived (https://web.archive.org/web/20210122083321/https://developers.google.com/caja/) from the original on 2021-01-22. Retrieved 2021-07-09.
81. "Mozilla Cross-Site Scripting Vulnerability Reported and Fixed – MozillaZine Talkback" (https://www.mozillazine.org/talkback.html?article=4392). *Mozillazine.org*. Archived (https://web.archive.org/web/20110721230916/http://www.mozillazine.org/talkback.html?article=4392) from the original on July 21, 2011. Retrieved February 24, 2017.
82. Kottelin, Thor (17 June 2008). "Right-click "protection"? Forget about it" (https://web.archive.org/web/20110809195359/https://blog.anta.net/2008/06/17/right-click-%E2%80%9Cprotection%E2%80%9D-forget-about-it/). *blog.anta.net*. Archived from the original (https://blog.anta.net/2008/06/17/right-click-%E2%80%9Cprotection%E2%80%9D-forget-about-it/) on 28 July 2022. Retrieved 28 July 2022.
83. Rehorik, Jan (29 November 2016). "Why You Should Never Put Sensitive Data in Your JavaScript" (https://www.serviceobjects.com/blog/why-you-should-never-put-sensitive-data-in-your-javascript/). *ServiceObjects Blog*. ServiceObjects. Archived (https://web.archive.org/web/20190603142957/https://www.serviceobjects.com/blog/why-you-should-never-put-sensitive-data-in-your-javascript/) from the original on June 3, 2019. Retrieved June 3, 2019.
84. Lauinger, Tobias; Chaabane, Abdelberi; Arshad, Sajjad; Robertson, William; Wilson, Christo; Kirda, Engin (December 21, 2016). *Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web* (https://web.archive.org/web/20170329045344/https://www.ccs.neu.edu/home/arshad/publications/ndss2017jslibs.pdf) (PDF). *Northeastern University*. arXiv:1811.00918 (https://arxiv.org/abs/1811.00918). doi:10.14722/ndss.2017.23414 (https://doi.org/10.14722%2Fndss.2017.23414). ISBN 978-1-891562-46-4. S2CID 17885720 (https://api.semanticscholar.org/CorpusID:17885720). Archived from the original (https://www.ccs.neu.edu/home/arshad/publications/ndss2017jslibs.pdf) (PDF) on 29 March 2017. Retrieved 28 July 2022.

85. Collins, Keith (March 27, 2016). "How one programmer broke the internet by deleting a tiny piece of code" (https://qz.com/646467/how-one-programmer-broke-the-internet-by-deleting-a-tiny-piece-of-code/). *Quartz*. Archived (https://web.archive.org/web/20170222200836/https://qz.com/646467/how-one-programmer-broke-the-internet-by-deleting-a-tiny-piece-of-code/) from the original on February 22, 2017. Retrieved February 22, 2017.
86. SC Magazine UK, Developer's 11 lines of deleted code 'breaks the internet' (https://www.scmagazineuk.com/developers-11-lines-of-deleted-code-breaks-the-internet/article/532050/) Archived (https://web.archive.org/web/20170223041434/https://www.scmagazineuk.com/developers-11-lines-of-deleted-code-breaks-the-internet/article/532050/) February 23, 2017, at the Wayback Machine
87. Mozilla Corporation, Buffer overflow in crypto.signText() (https://www.mozilla.org/security/announce/2006/mfsa2006-38.html) Archived (https://web.archive.org/web/20140604014705/https://www.mozilla.org/security/announce/2006/mfsa2006-38.html) 2014-06-04 at the Wayback Machine
88. Festa, Paul (August 19, 1998). "Buffer-overflow bug in IE" (https://web.archive.org/web/20021225190522/https://news.com.com/2100-1001-214620.html). *CNET*. Archived from the original (https://news.com.com/2100-1001-214620.html) on December 25, 2002.
89. SecurityTracker.com, Apple Safari JavaScript Buffer Overflow Lets Remote Users Execute Arbitrary Code and HTTP Redirect Bug Lets Remote Users Access Files (https://securitytracker.com/alerts/2006/Mar/1015713.html) Archived (https://web.archive.org/web/20100218102849/https://securitytracker.com/alerts/2006/Mar/1015713.html) 2010-02-18 at the Wayback Machine
90. SecurityFocus, Microsoft WebViewFolderIcon ActiveX Control Buffer Overflow Vulnerability (https://www.securityfocus.com/bid/19030/info) Archived (https://web.archive.org/web/20111011091819/http://www.securityfocus.com/bid/19030/info) 2011-10-11 at the Wayback Machine
91. Fusion Authority, Macromedia Flash ActiveX Buffer Overflow (https://www.fusionauthority.com/security/3234-macromedia-flash-activex-buffer-overflow.htm) Archived (https://web.archive.org/web/20110813160055/https://www.fusionauthority.com/security/3234-macromedia-flash-activex-buffer-overflow.htm) August 13, 2011, at the Wayback Machine
92. "Protected Mode in Vista IE7 – IEBlog" (https://blogs.msdn.com/ie/archive/2006/02/09/528963.aspx). *Blogs.msdn.com*. February 9, 2006. Archived (https://web.archive.org/web/20100123103719/https://blogs.msdn.com/ie/archive/2006/02/09/528963.aspx) from the original on January 23, 2010. Retrieved February 24, 2017.
93. US CERT, Vulnerability Note VU#713878: Microsoft Internet Explorer does not properly validate source of redirected frame (https://www.kb.cert.org/vuls/id/713878) Archived (https://web.archive.org/web/20091030051811/https://www.kb.cert.org/vuls/id/713878/) 2009-10-30 at the Wayback Machine
94. Mozilla Foundation, Mozilla Foundation Security Advisory 2005–41: Privilege escalation via DOM property overrides (https://www.mozilla.org/security/announce/2005/mfsa2005-41.html) Archived (https://web.archive.org/web/20140604014832/https://www.mozilla.org/security/announce/2005/mfsa2005-41.html) 2014-06-04 at the Wayback Machine
95. Andersen, Starr (2004-08-09). "Part 5: Enhanced Browsing Security" (https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-xp/bb457150(v=technet.10)). TechNet. *Microsoft Docs*. Changes to Functionality in Windows XP Service Pack 2. Retrieved 2021-10-20.

96. For one example of a rare JavaScript Trojan Horse, see Symantec Corporation, JS.Seeker.K (https://www.symantec.com/security_response/writeup.jsp?docid=2003-100111-0931-99) Archived (https://web.archive.org/web/20110913210848/http://www.symantec.com/security_response/writeup.jsp?docid=2003-100111-0931-99) 2011-09-13 at the Wayback Machine
97. Gruss, Daniel; Maurice, Clémentine; Mangard, Stefan (July 24, 2015). "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript". arXiv:1507.06955 (https://arxiv.org/abs/1507.06955) [cs.CR (https://arxiv.org/archive/cs.CR)].
98. Jean-Pharuns, Alix (July 30, 2015). "Rowhammer.js Is the Most Ingenious Hack I've Ever Seen" (https://motherboard.vice.com/en_us/article/9akpwz/rowhammerjs-is-the-most-ingenious-hack-ive-ever-seen). *Motherboard*. Vice. Archived (https://web.archive.org/web/20180127084042/https://motherboard.vice.com/en_us/article/9akpwz/rowhammerjs-is-the-most-ingenious-hack-ive-ever-seen) from the original on January 27, 2018. Retrieved January 26, 2018.
99. Goodin, Dan (August 4, 2015). "DRAM 'Bitflipping' exploit for attacking PCs: Just add JavaScript" (https://arstechnica.com/information-technology/2015/08/dram-bitflipping-exploit-for-attacking-pcs-just-add-javascript/). *Ars Technica*. Archived (https://web.archive.org/web/20180127143154/https://arstechnica.com/information-technology/2015/08/dram-bitflipping-exploit-for-attacking-pcs-just-add-javascript/) from the original on January 27, 2018. Retrieved January 26, 2018.
100. Auerbach, David (July 28, 2015). "Rowhammer security exploit: Why a new security attack is truly terrifying" (https://www.slate.com/articles/technology/bitwise/2015/07/rowhammer_security_exploit_why_a_new_security_attack_is_truly_terrifying.html). *slate.com*. Archived (https://web.archive.org/web/20150730004023/https://www.slate.com/articles/technology/bitwise/2015/07/rowhammer_security_exploit_why_a_new_security_attack_is_truly_terrifying.html) from the original on July 30, 2015. Retrieved July 29, 2015.
101. AnC (https://www.vusec.net/projects/anc/) Archived (https://web.archive.org/web/20170316055626/https://www.vusec.net/projects/anc/) 2017-03-16 at the Wayback Machine VUSec, 2017
102. New ASLR-busting JavaScript is about to make drive-by exploits much nastier (https://arstechnica.com/security/2017/02/new-aslr-busting-javascript-is-about-to-make-drive-by-exploits-much-nastier/) Archived (https://web.archive.org/web/20170316024419/https://arstechnica.com/security/2017/02/new-aslr-busting-javascript-is-about-to-make-drive-by-exploits-much-nastier/) 2017-03-16 at the Wayback Machine Ars Technica, 2017
103. Spectre Attack (https://spectreattack.com/spectre.pdf) Archived (https://web.archive.org/web/20180103225843/https://spectreattack.com/spectre.pdf) 2018-01-03 at the Wayback Machine Spectre Attack
104. "Benchmark.js" (https://benchmarkjs.com/). *benchmarkjs.com*. Archived (https://web.archive.org/web/20161219182724/https://benchmarkjs.com/) from the original on 2016-12-19. Retrieved 2016-11-06.
105. JSBEN.CH. "JSBEN.CH Performance Benchmarking Playground for JavaScript" (https://jsben.ch). *jsben.ch*. Archived (https://web.archive.org/web/20210227052409/https://jsben.ch/) from the original on 2021-02-27. Retrieved 2021-08-13.
106. Eich, Brendan (April 3, 2008). "Popularity" (https://brendaneich.com/2008/04/popularity/). Archived (https://web.archive.org/web/20110703020955/https://brendaneich.com/2008/04/popularity/) from the original on July 3, 2011. Retrieved January 19, 2012.

107. "Edge Browser Switches WebAssembly to 'On' -- Visual Studio Magazine" (https://visualstudiomagazine.com/articles/2017/11/06/edge-webassembly.aspx). *Visual Studio Magazine*. Archived (https://web.archive.org/web/20180210002432/https://visualstudiomagazine.com/articles/2017/11/06/edge-webassembly.aspx) from the original on 2018-02-10. Retrieved 2018-02-09.
108. "frequently asked questions" (https://asmjs.org/faq.html). asm.js. Archived (https://web.archive.org/web/20140604012024/https://asmjs.org/faq.html) from the original on June 4, 2014. Retrieved April 13, 2014.

Further reading

- Flanagan, David. *JavaScript: The Definitive Guide*. 7th edition. Sebastopol, California: O'Reilly, 2020.
- Haverbeke, Marijn. *Eloquent JavaScript*. 3rd edition. No Starch Press, 2018. 472 pages. ISBN 978-1593279509.*(download)* (https://eloquentjavascript.net/)
- Zakas, Nicholas. *Principles of Object-Oriented JavaScript*, 1st edition. No Starch Press, 2014. 120 pages. ISBN 978-1593275402.

External links

- JavaScript (https://curlie.org/Computers/Programming/Languages/JavaScript/) at Curlie
- "JavaScript: The First 20 Years" (https://www.pldi21.org/prerecorded_hopl.12.html). Retrieved 2022-02-06.

Retrieved from "https://en.wikipedia.org/w/index.php?title=JavaScript&oldid=1102150915"

This page was last edited on 3 August 2022, at 16:33 (UTC).