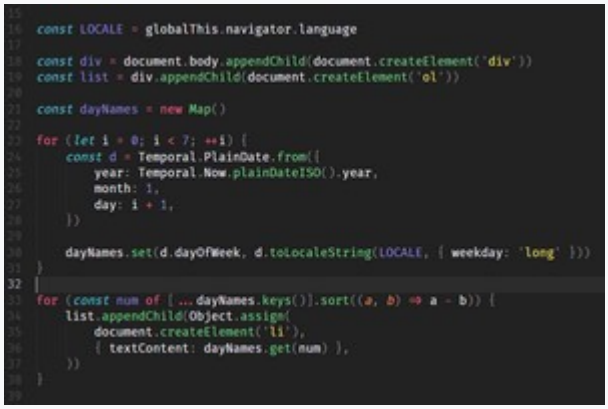
SUBJECT-JavaScript [bca –IIIrd yr]

JavaScript



Screenshot of JavaScript source code

Paradigm	Multi-paradigm: event-driven, functional, imperative, procedural, object-oriented
Designed by	Brendan Eich of Netscape initially; others have also contributed to the ECMAScript standard
First appeared	4 December 1995; 28 years ago[1]

), often abbreviated as **JS**, is a programming language and core technology of the Web, alongside HTML and CSS. 99% of websites use JavaScript on the client side for webpage behavior.[10]

Web browsers have a dedicated JavaScript engine that executes the client code. These engines are also utilized in some servers and a variety of apps. The most popular runtime system for non-browser usage is Node.js.

JavaScript is a high-level, often just-in-time compiled language that conforms to the ECMAScript standard.[11] 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.

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

History

Creation at Netscape

The first popular 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 early World Wide Web.[12] 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.[13]

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 programming language to Navigator. They pursued two routes to achieve this: collaborating with Sun Microsystems to embed the Java language, while also hiring Brendan Eich to embed the Scheme language.[6]

The goal was a "language for the masses",[14] "to help nonprogrammers create dynamic, interactive Web sites".[15] 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][16][17]

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 a popular new language, so Eich considered the JavaScript name a marketing ploy by Netscape.[14]

Adoption by Microsoft

Microsoft debuted Internet Explorer in 1995, leading to a browser war with Netscape. On the JavaScript front, Microsoft created its own interpreter called JScript.[18]

Microsoft first released JScript in 1996, alongside initial support for CSS and extensions to HTML. Each of these implementations was noticeably different from their counterparts in Netscape Navigator.[19][20] 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.[19][21]

The rise of JScript

Brendan Eich later said of this period: "It's still kind of a sidekick language. It's considered slow or annoying. People do pop-ups or those scrolling messages in the old status bar at the bottom of your old browser."[14]

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.[18]

However, the effort to fully standardize the language was undermined by Microsoft gaining an increasingly dominant position in the browser market. By the early 2000s, Internet Explorer's market share reached 95%.[22] 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.[23]

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.[24][25] The key innovation was just-in-time compilation (JIT),[26] so other browser vendors needed to overhaul their engines for JIT.[27]

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.[28]

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.[29][30] As of 2018, Node had been used by millions of developers,[31] and npm had the most modules of any package manager in the world.[32]

The ECMAScript draft specification is currently maintained openly on GitHub,[33] and editions are produced via regular annual snapshots.[33] Potential revisions to the language are vetted through a comprehensive proposal process.[34][35] Now, instead of edition numbers, developers check the status of upcoming features individually.[33]

The current JavaScript ecosystem has many libraries and frameworks, established programming practices, and substantial usage of JavaScript outside of web browsers.[17] Plus, with the rise of single-page applications and other JavaScript-heavy websites, several transpilers have been created to aid the development process.[36]

Trademark

"JavaScript" is a trademark of Oracle Corporation in the United States.[37][38] The trademark was originally issued to Sun Microsystems on 6 May 1997, and was transferred to Oracle when they acquired Sun in 2009.[39]

Website client-side usage

JavaScript is the dominant client-side scripting language of the Web, with 99% of all websites using it for this purpose.[10] 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 or alert boxes.
- 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.
- Storing and retrieving data on the user's device, via the storage or IndexedDB standards.

Libraries and frameworks

Over 80% of websites use a third-party JavaScript library or web framework as part of their client-side scripting.[40]

jQuery is by far the most-used.[40] Other notable ones include Angular, Bootstrap, Lodash, Modernizr, React, Underscore, and Vue.[40] Multiple options can be used in conjunction, such as jQuery and Bootstrap.[41]

However, the term "Vanilla JS" was coined for websites not using any libraries or frameworks at all, instead relying entirely on standard JavaScript functionality.[42]

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,[43][44] but they were small niches.[45] Server-side usage eventually started to grow in the late 2000s, with the creation of Node.js and other approaches.[45]

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[46] and GNOME Shell extensions written in JavaScript.[47]

JavaScript has been used in some embedded systems, usually by leveraging Node.js.[48][49][50]

Execution

JavaScript engine

A JavaScript engine is a software component that executes JavaScript code. The first JavaScript engines were mere interpreters, but all relevant modern engines use just-in-time compilation for improved performance.[51]

JavaScript engines are typically developed by web browser vendors, and every major browser has one. In a browser, the JavaScript engine runs in concert with the rendering engine via the Document Object Model and Web IDL bindings.[52] However, the use of JavaScript engines is not limited to browsers; for example, the V8 engine is a core component of the Node.js runtime system.[53]

Since ECMAScript is the standardized specification of JavaScript, ECMAScript engine is another name for these implementations. With the advent of WebAssembly, some engines can also execute this code in the same sandbox as regular JavaScript code.[54][53]

Runtime system

A JavaScript engine must be embedded within a runtime system (such as a web browser or a standalone system) to enable scripts to interact with the broader environment. The runtime system includes the necessary APIs for input/output operations, such as networking, storage, and graphics, and provides the ability to import scripts.

JavaScript is a single-threaded language. The runtime processes messages from a queue one at a time, and it 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 I/O is performed using events and callback functions. This means, for example, that JavaScript can process a mouse click while waiting for a database query to return information.[55]

The notable standalone runtimes are Node.js, Deno, and Bun.

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.[56]

Weakly typed

JavaScript is weakly typed, which means certain types are implicitly cast depending on the operation used.[57]

- 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:[57]

- 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.[58][57] 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 the expression is wrapped in parentheses - `({} + [])` – the curly brackets are interpreted as an empty object and the result of the expression is `"[object Object]"` as expected.[57]

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.[59] JavaScript supports various ways to test the type of objects, including duck typing.[60]

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?[61]

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.[62] It is possible to simulate many class-based features with prototypes in JavaScript.[63]

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).[64] 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`).[65] 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.[66]

Functions as methods

Unlike in many object-oriented languages, in JavaScript 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.[67] As such, a function may have properties and methods, such as `.call()` and `.bind()`.[68]

Lexical closure

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.[69]

Anonymous function

Functions as roles (Traits and Mixins).

JavaScript natively supports various function-based implementations of Role[70] patterns like Traits[71][72] and Mixins.[73] 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

Zero-based numbering

JavaScript is a zero-index language.

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` method.

Array and object literals

Like in 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

In a manner similar to Perl, JavaScript also supports regular expressions, which provide a concise and powerful syntax for text manipulation that is more sophisticated than the built-in string functions.[74]

Promises and Async/await

JavaScript supports promises and Async/await for handling asynchronous operations.[citation needed]

Promises

A built-in Promise object provides functionality for handling promises and associating handlers with an asynchronous action's eventual result. Recently, the JavaScript specification introduced combinator methods, which allow 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

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 similarly 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[75])

Syntax

Simple examples

Variables in JavaScript can be defined using either the `var`,[76] `let`[77] or `const`[78] 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 is generally considered bad practice and let and const are usually
preferred.
var x;

// Variables can be manually set to `undefined` like so
let 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 examples 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".[79] However, most runtime environments have a `console` object that can be used to print output.[80] 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';

// After setting this, the tag will look like this: `<span class="foo"
id="bar" data-attr="baz"></span>`
myElem.setAttribute('data-attr', 'baz'); // Which could also be written as
`myElem.dataset.attr = 'baz'`

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

// Elements can be imperatively grabbed with querySelector for one element,
or querySelectorAll for multiple elements that can be looped with forEach
document.querySelector('.class'); // Selects the first element with the
"class" class
document.querySelector('#id'); // Selects the first element with an `id` of
"id"
document.querySelector('[data-other]'); // Selects the first element with the
"data-other" attribute
document.querySelectorAll('.multiple'); // Returns an Array-like NodeList of
all elements with the "multiple" class
```

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 special 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 recursivelyCompute = a => a > 1 ? a * recursivelyCompute(a - 1) :
1; // Note the use of the ternary operator `?`.
    return recursivelyCompute(n);
}

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

An anonymous function (or lambda):

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

const 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.
let 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 can be created as instances of a class.

Object class example:

```
class Ball {

   constructor(radius) {
      this.radius = radius;
      this.area = Math.PI * ( radius ** 2 );
   }

   // Classes (and thus objects) can contain functions known as methods
   show() {
      console.log(this.radius);
   }
};

const 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 logs "6"
```

In JavaScript, objects can be instantiated directly from a function.

Object functional example:

```
function Ball(radius) {

   const area = Math.PI * ( radius ** 2 );
   const obj = { radius, area };

   // Objects are mutable, and functions can be added as properties.
   obj.show = () => console.log(obj.radius);
   return obj;
};

const myBall = Ball(5); // Creates a new ball object with radius 5. No "new"
keyword needed.
myBall.radius++; // The instance property can be modified.
```

```
myBall.show();   // Using the "show" function logs "6" - the new instance
value.
```

Variadic function demonstration (`arguments` is a special variable):[81]

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

Generator objects (in the form of generator functions) provide a function which can be called, exited, and re-entered while maintaining internal context (statefulness).[82]

```
function* rawCounter() {
    yield 1;
    yield 2;
}
```

```
function* dynamicCounter() {
    let count = 0;
    while (true) {
        // It is not recommended to utilize while true loops in most cases.
        yield ++count;
    }
}

// Instances
const counter1 = rawCounter();
const counter2 = dynamicCounter();

// Implementation
counter1.next(); // {value: 1, done: false}
counter1.next(); // {value: 2, done: false}
counter1.next(); // {value: undefined, done: true}

counter2.next(); // {value: 1, done: false}
counter2.next(); // {value: 2, done: false}
counter2.next(); // {value: 3, done: false}
// ...infinitely
```

JavaScript can export and import from modules:[83]

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
```