mdn web docs

# JavaScript language overview

JavaScript is a multi-paradigm, dynamic language with types and operators, standard built-in objects, and methods. Its syntax is based on the Java and C languages — many structures from those languages apply to JavaScript as well. JavaScript supports object-oriented programming with object prototypes and classes. It also supports functional programming since functions are first-class that can be easily created via expressions and passed around like any other object.

This page serves as a quick overview of various JavaScript language features, written for readers with background in other languages, such as C or Java.

## Data types

Let's start off by looking at the building blocks of any language: the types. JavaScript programs manipulate values, and those values all belong to a type. JavaScript offers seven primitive types:

- Number: used for all number values (integer and floating point) except for very big integers.
- BigInt: used for arbitrarily large integers.
- String: used to store text.
- Boolean: `true` and `false` — usually used for conditional logic.
- Symbol: used for creating unique identifiers that won't collide.
- Undefined: indicating that a variable has not been assigned a value.
- Null: indicating a deliberate non-value.

Everything else is known as an Object. Common object types include:

- `Function`
- `Array`
- `Date`
- `RegExp`
- `Error`

Functions aren't special data structures in JavaScript — they are just a special type of object that can be called.

## Numbers

JavaScript has two built-in numeric types: Number and BigInt.

The Number type is a IEEE 754 64-bit double-precision floating point value , which means integers can be safely represented between $-(2^{53} - 1)$ and $2^{53} - 1$ without loss of precision, and floating point numbers can be stored all the way up to $1.79 \times 10^{308}$. Within numbers, JavaScript does not distinguish between floating point numbers and integers.

```
console.log(3 / 2); // 1.5, not 1
```

So an apparent integer is in fact implicitly a float. Because of IEEE 754 encoding, sometimes floating point arithmetic can be imprecise.

```
console.log(0.1 + 0.2); // 0.30000000000000004
```

For operations that expect integers, such as bitwise operations, the number will be converted to a 32-bit integer.

Number literals can also have prefixes to indicate the base (binary, octal, decimal, or hexadecimal), or an exponent suffix.

```
console.log(0b111110111); // 503
console.log(0o767); // 503
console.log(0x1f7); // 503
console.log(5.03e2); // 503
```

The BigInt type is an arbitrary length integer. Its behavior is similar to C's integer types (e.g. division truncates to zero), except it can grow indefinitely. BigInts are specified with a number literal and an `n` suffix.

```
console.log(-3n / 2n); // -1n
```

The standard arithmetic operators are supported, including addition, subtraction, remainder arithmetic, etc. BigInts and numbers cannot be mixed in arithmetic operations.

The `Math` object provides standard mathematical functions and constants.

```
Math.sin(3.5);
const circumference = 2 * Math.PI * r;
```

There are three ways to convert a string to a number:

- `parseInt()`, which parses the string for an integer.
- `parseFloat()`, which parses the string for a floating-point number.
- The `Number()` function, which parses a string as if it's a number literal and supports many different number representations.

You can also use the unary plus `+` as a shorthand for `Number()`.

Number values also include `NaN` (short for "Not a Number") and `Infinity`. Many "invalid math" operations will return `NaN` — for example, if attempting to parse a non-numeric string, or using `Math.log()` on a negative value. Division by zero produces `Infinity` (positive or negative).

`NaN` is contagious: if you provide it as an operand to any mathematical operation, the result will also be `NaN`. `NaN` is the only value in JavaScript that's not equal to itself (per IEEE 754 specification).

## Strings

Strings in JavaScript are sequences of Unicode characters. This should be welcome news to anyone who has had to deal with internationalization. More accurately, they are UTF-16 encoded.

```
console.log("Hello, world");
console.log("你好，世界！"); // Nearly all Unicode characters can be written literally in
string literals
```

Strings can be written with either single or double quotes — JavaScript does not have the distinction between characters and strings. If you want to represent a single character, you just use a string consisting of that single character.

```
console.log("Hello"[1] === "e"); // true
```

To find the length of a string (in code units), access its `length` property.

Strings have utility methods to manipulate the string and access information about the string. Because all primitives are immutable by design, these methods return new strings.

The `+` operator is overloaded for strings: when one of the operands is a string, it performs string concatenation instead of number addition. A special template literal syntax allows you to write strings with embedded expressions more succinctly. Unlike Python's f-strings or C#'s interpolated strings, template literals use backticks (not single or double quotes).

```
const age = 25;
console.log("I am " + age + " years old."); // String concatenation
console.log(`I am ${age} years old.`); // Template literal
```

## Other types

JavaScript distinguishes between `null`, which indicates a deliberate non-value (and is only accessible through the `null` keyword), and `undefined`, which indicates absence of value. There are many ways to obtain `undefined`:

- A `return` statement with no value (`return;`) implicitly returns `undefined`.
- Accessing a nonexistent object property (`obj.iDontExist`) returns `undefined`.
- A variable declaration without initialization (`let x;`) will implicitly initialize the variable to `undefined`.

JavaScript has a Boolean type, with possible values `true` and `false` — both of which are keywords. Any value can be converted to a boolean according to the following rules:

1. `false`, `0`, empty strings (`""`), `NaN`, `null`, and `undefined` all become `false`.
2. All other values become `true`.

You can perform this conversion explicitly using the `Boolean()` function:

```
Boolean(""); // false
Boolean(234); // true
```

However, this is rarely necessary, as JavaScript will silently perform this conversion when it expects a boolean, such as in an `if` statement (see Control structures). For this reason, we sometimes speak of "truthy" and "falsy", meaning values that become `true` and `false`, respectively, when used in boolean contexts.

Boolean operations such as `&&` (logical and), `||` (logical or), and `!` (logical not) are supported; see Operators.

The Symbol type is often used to create unique identifiers. Every symbol created with the

`Symbol()` function is guaranteed to be unique. In addition, there are registered symbols, which are shared constants, and well-known symbols, which are utilized by the language as "protocols" for certain operations. You can read more about them in the symbol reference.

## Variables

Variables in JavaScript are declared using one of three keywords: `let`, `const`, or `var`.

`let` allows you to declare block-level variables. The declared variable is available from the block it is enclosed in.

```
let a;
let name = "Simon";

// myLetVariable is *not* visible out here

for (let myLetVariable = 0; myLetVariable < 5; myLetVariable++) {
  // myLetVariable is only visible in here
}

// myLetVariable is *not* visible out here
```

`const` allows you to declare variables whose values are never intended to change. The variable is available from the block it is declared in.

```
const Pi = 3.14; // Declare variable Pi
console.log(Pi); // 3.14
```

A variable declared with `const` cannot be reassigned.

```
  const Pi = 3.14;
  Pi = 1; // will throw an error because you cannot change a constant variable.
```

`const` declarations only prevent reassignments — they don't prevent mutations of the variable's value, if it's an object.

```
const obj = {};
obj.a = 1; // no error
console.log(obj); // { a: 1 }
```

`var` declarations can have surprising behaviors (for example, they are not block-scoped), and they are discouraged in modern JavaScript code.

If you declare a variable without assigning any value to it, its value is `undefined`. You can't declare a `const` variable without an initializer, because you can't change it later anyway.

`let` and `const` declared variables still occupy the entire scope they are defined in, and are in a region known as the temporal dead zone before the actual line of declaration. This has some interesting interactions with variable shadowing, which don't occur in other languages.

```
function foo(x, condition) {
  if (condition) {
    console.log(x);
    const x = 2;
    console.log(x);
  }
}

foo(1, true);
```

In most other languages, this would log "1" and "2", because before the `const x = 2` line, `x` should still refer to the parameter `x` in the upper scope. In JavaScript, because each declaration occupies the entire scope, this would throw an error on the first `console.log`: "Cannot access 'x' before initialization". For more information, see the reference page of `let`.

JavaScript is dynamically typed. Types (as described in the previous section) are only associated with values, but not with variables. For `let`-declared variables, you can always change its type through reassignment.

```
let a = 1;
a = "foo";
```

## Operators

JavaScript's numeric operators include `+`, `-`, `*`, `/`, `%` (remainder), and `**` (exponentiation). Values are assigned using `=`. Each binary operator also has a compound assignment counterpart such as `+=` and `-=`, which extend out to `x = x operator y`.

```
x += 5;
x = x + 5;
```

You can use `++` and `--` to increment and decrement respectively. These can be used as a prefix or postfix operators.

The `+` operator also does string concatenation:

```
"hello" + " world"; // "hello world"
```

If you add a string to a number (or other value) everything is converted into a string first. This might trip you up:

```
"3" + 4 + 5; // "345"
3 + 4 + "5"; // "75"
```

Adding an empty string to something is a useful way of converting it to a string itself.

Comparisons in JavaScript can be made using `<`, `>`, `<=` and `>=`, which work for both strings and numbers. For equality, the double-equals operator performs type coercion if you give it different types, with sometimes interesting results. On the other hand, the triple-equals operator does not attempt type coercion, and is usually preferred.

```
123 == "123"; // true
1 == true; // true

123 === "123"; // false
1 === true; // false
```

The double-equals and triple-equals also have their inequality counterparts: `!=` and `!==`.

JavaScript also has bitwise operators and logical operators. Notably, logical operators don't work with boolean values only — they work by the "truthiness" of the value.

```
const a = 0 && "Hello"; // 0 because 0 is "falsy"
const b = "Hello" || "world"; // "Hello" because both "Hello" and "world" are "truthy"
```

The `&&` and `||` operators use short-circuit logic, which means whether they will execute their second operand is dependent on the first. This is useful for checking for null objects before accessing their attributes:

```
const name = o && o.getName();
```

Or for caching values (when falsy values are invalid):

```
const name = cachedName || (cachedName = getName());
```

For a comprehensive list of operators, see the guide page or reference section. You may be especially interested in the operator precedence.

## Grammar

JavaScript grammar is very similar to the C family. There are a few points worth mentioning:

- Identifiers can have Unicode characters, but they cannot be one of the reserved words.
- Comments are commonly `//` or `/* */`, while many other scripting languages like Perl, Python, and Bash use `#`.
- Semicolons are optional in JavaScript — the language automatically inserts them when needed. However, there are certain caveats to watch out, since unlike Python, semicolons are still part of the syntax.

For an in-depth look at the JavaScript grammar, see the reference page for lexical grammar.

## Control structures

JavaScript has a similar set of control structures to other languages in the C family. Conditional statements are supported by `if` and `else`; you can chain them together:

```
let name = "kittens";
if (name === "puppies") {
  name += " woof";
} else if (name === "kittens") {
  name += " meow";
} else {
  name += "!";
}
name === "kittens meow";
```

JavaScript doesn't have `elif`, and `else if` is really just an `else` branch comprised of a single `if` statement.

JavaScript has `while` loops and `do...while` loops. The first is good for basic looping; the second is for loops where you wish to ensure that the body of the loop is executed at least once:

```
while (true) {
  // an infinite loop!
}

let input;
do {
  input = get_input();
} while (inputIsNotValid(input));
```

JavaScript's `for` loop is the same as that in C and Java: it lets you provide the control information for your loop on a single line.

```
for (let i = 0; i < 5; i++) {
  // Will execute 5 times
}
```

JavaScript also contains two other prominent for loops: `for...of`, which iterates over iterables, most notably arrays, and `for...in`, which visits all enumerable properties of an object.

```
for (const value of array) {
  // do something with value
}

for (const property in object) {
  // do something with object property
}
```

The `switch` statement can be used for multiple branches based on equality checking.

```
switch (action) {
  case "draw":
    drawIt();
    break;
  case "eat":
    eatIt();
    break;
  default:
    doNothing();
}
```

Similar to C, case clauses are conceptually the same as labels, so if you don't add a `break` statement, execution will "fall through" to the next level. However, they are not actually jump tables — any expression can be part of the `case` clause, not just string or number literals, and they would be evaluated one-by-one until one equals the value being matched. Comparison takes place between the two using the `===` operator.

Unlike some languages like Rust, control-flow structures are statements in JavaScript, meaning you can't assign them to a variable, like `const a = if (x) { 1 } else { 2 }`.

JavaScript errors are handled using the `try...catch` statement.

```
try {
  buildMySite("./website");
} catch (e) {
  console.error("Building site failed:", e);
}
```

Errors can be thrown using the `throw` statement. Many built-in operations may throw as well.

```
function buildMySite(siteDirectory) {
  if (!pathExists(siteDirectory)) {
    throw new Error("Site directory does not exist");
  }
}
```

In general, you can't tell the type of the error you just caught, because anything can be thrown from a `throw` statement. However, you can usually assume it's an `Error` instance, as is the example above. There are some subclasses of `Error` built-in, like `TypeError` and `RangeError`, that you can use to provide extra semantics about the error. There's no conditional catch in JavaScript — if you only want to handle one type of error, you need to catch everything, identify the type of error using `instanceof`, and then rethrow the other cases.

```
try {
  buildMySite("./website");
} catch (e) {
  if (e instanceof RangeError) {
    console.error("Seems like a parameter is out of range:", e);
    console.log("Retrying...");
    buildMySite("./website");
  } else {
    // Don't know how to handle other error types; throw them so
    // something else up in the call stack may catch and handle it
    throw e;
  }
}
```

If an error is uncaught by any `try...catch` in the call stack, the program will exit.

For a comprehensive list of control flow statements, see the reference section.

## Objects

JavaScript objects can be thought of as collections of key-value pairs. As such, they are similar to:

- Dictionaries in Python.
- Hashes in Perl and Ruby.
- Hash tables in C and C++.
- HashMaps in Java.
- Associative arrays in PHP.

JavaScript objects are hashes. Unlike objects in statically typed languages, objects in

JavaScript do not have fixed shapes — properties can be added, deleted, re-ordered, mutated, or dynamically queried at any time. Object keys are always strings or symbols — even array indices, which are canonically integers, are actually strings under the hood.

Objects are usually created using the literal syntax:

```
const obj = {
  name: "Carrot",
  for: "Max",
  details: {
    color: "orange",
    size: 12,
  },
};
```

Object properties can be accessed using dot ( `.` ) or brackets ( `[]` ). When using the dot notation, the key must be a valid identifier. Brackets, on the other hand, allow indexing the object with a dynamic key value.

```
// Dot notation
obj.name = "Simon";
const name = obj.name;

// Bracket notation
obj["name"] = "Simon";
const name = obj["name"];

// Can use a variable to define a key
const userName = prompt("what is your key?");
obj[userName] = prompt("what is its value?");
```

Property access can be chained together:

```
obj.details.color; // orange
obj["details"]["size"]; // 12
```

Objects are always references, so unless something is explicitly copying the object, mutations to an object would be visible to the outside.

```
const obj = {};
function doSomething(o) {
  o.x = 1;
}
doSomething(obj);
console.log(obj.x); // 1
```

This also means two separately created objects will never be equal ( `!==` ), because they are different references. If you hold two references of the same object, mutating one would be observable through the other.

```
const me = {};
const stillMe = me;
me.x = 1;
console.log(stillMe.x); // 1
```

For more on objects and prototypes, see the `Object` reference page. For more information on the object initializer syntax, see its reference page.

This page has omitted all details about object prototypes and inheritance because you can usually achieve inheritance with classes without touching the underlying mechanism (which you may have heard to be abstruse). To learn about them, see Inheritance and the prototype chain.

## Arrays

Arrays in JavaScript are actually a special type of object. They work very much like regular objects (numerical properties can naturally be accessed only using `[]` syntax) but they have one magic property called `length` . This is always one more than the highest index in the array.

Arrays are usually created with array literals:

```
const a = ["dog", "cat", "hen"];
a.length; // 3
```

JavaScript arrays are still objects — you can assign any properties to them, including arbitrary number indices. The only "magic" is that `length` will be automatically updated when you set a particular index.

```
const a = ["dog", "cat", "hen"];
a[100] = "fox";
console.log(a.length); // 101
console.log(a); // ['dog', 'cat', 'hen', empty × 97, 'fox']
```

The array we got above is called a sparse array because there are uninhabited slots in the middle, and will cause the engine to deoptimize it from an array to a hash table. Make sure your array is densely populated!

Out-of-bounds indexing doesn't throw. If you query a non-existent array index, you'll get a value of `undefined` in return:

```
const a = ["dog", "cat", "hen"];
console.log(typeof a[90]); // undefined
```

Arrays can have any elements and can grow or shrink arbitrarily.

```
const arr = [1, "foo", true];
arr.push({});
// arr = [1, "foo", true, {}]
```

Arrays can be iterated with the `for` loop, as you can in other C-like languages:

```
for (let i = 0; i < a.length; i++) {
  // Do something with a[i]
}
```

Or, since arrays are iterable, you can use the `for...of` loop, which is synonymous to C++/Java's `for (int x : arr)` syntax:

```
for (const currentValue of a) {
  // Do something with currentValue
}
```

Arrays come with a plethora of array methods. Many of them would iterate the array — for example, `map()` would apply a callback to every array element, and return a new array:

```
const babies = ["dog", "cat", "hen"].map((name) => `baby ${name}`);
// babies = ['baby dog', 'baby cat', 'baby hen']
```

## Functions

Along with objects, functions are the core component in understanding JavaScript. The most basic function declaration looks like this:

```
function add(x, y) {
  const total = x + y;
  return total;
}
```

A JavaScript function can take 0 or more parameters. The function body can contain as many statements as you like and can declare its own variables which are local to that function. The `return` statement can be used to return a value at any time, terminating the function. If no return statement is used (or an empty return with no value), JavaScript returns `undefined`.

Functions can be called with more or fewer parameters than it specifies. If you call a function without passing the parameters it expects, they will be set to `undefined`. If you pass more parameters than it expects, the function will ignore the extra parameters.

```
add(); // NaN
// Equivalent to add(undefined, undefined)

add(2, 3, 4); // 5
// added the first two; 4 was ignored
```

There are a number of other parameter syntaxes available. For example, the rest parameter syntax allows collecting all the extra parameters passed by the caller into an array, similar to Python's `*args`. (Since JS doesn't have named parameters on the language level, there's no `**kwargs`.)

```
function avg(...args) {
  let sum = 0;
  for (const item of args) {
    sum += item;
  }
  return sum / args.length;
}

avg(2, 3, 4, 5); // 3.5
```

In the above code, the variable `args` holds all the values that were passed into the function.

The rest parameter will store all arguments after where it's declared, but not before. In other words, `function avg(firstValue, ...args)` will store the first value passed into the function in the `firstValue` variable and the remaining arguments in `args`.

If a function accepts a list of arguments and you already hold an array, you can use the spread syntax in the function call to spread the array as a list of elements. For instance: `avg(...numbers)`.

We mentioned that JavaScript doesn't have named parameters. It's possible, though, to implement them using object destructuring, which allows objects to be conveniently packed and unpacked.

```
// Note the { } braces: this is destructuring an object
function area({ width, height }) {
  return width * height;
}

// The { } braces here create a new object
console.log(area({ width: 2, height: 3 }));
```

There's also the default parameter syntax, which allows omitted parameters (or those passed as `undefined`) to have a default value.

```
function avg(firstValue, secondValue, thirdValue = 0) {
  return (firstValue + secondValue + thirdValue) / 3;
}

avg(1, 2); // 1, instead of NaN
```

## Anonymous functions

JavaScript lets you create anonymous functions — that is, functions without names. In practice, anonymous functions are typically used as arguments to other functions, immediately assigned to a variable that can be used to invoke the function, or returned from another function.

```
// Note that there's no function name before the parentheses
const avg = function (...args) {
  let sum = 0;
  for (const item of args) {
    sum += item;
  }
  return sum / args.length;
};
```

That makes the anonymous function invocable by calling `avg()` with some arguments — that is, it's semantically equivalent to declaring the function using the `function avg() {}` declaration syntax.

There's another way to define anonymous functions — using an arrow function expression.

```
// Note that there's no function name before the parentheses
const avg = (...args) => {
  let sum = 0;
  for (const item of args) {
    sum += item;
  }
  return sum / args.length;
};

// You can omit the `return` when simply returning an expression
const sum = (a, b, c) => a + b + c;
```

Arrow functions are not semantically equivalent to function expressions — for more information, see its reference page.

There's another way that anonymous functions can be useful: it can be simultaneously declared and invoked in a single expression, called an Immediately invoked function expression (IIFE):

```
(function () {
  // …
})();
```

For use-cases of IIFEs, you can read emulating private methods with closures.

## Recursive functions

JavaScript allows you to call functions recursively. This is particularly useful for dealing with tree structures, such as those found in the browser DOM.

```
function countChars(elm) {
  if (elm.nodeType === 3) {
    // TEXT_NODE
    return elm.nodeValue.length;
  }
  let count = 0;
  for (let i = 0, child; (child = elm.childNodes[i]); i++) {
    count += countChars(child);
  }
  return count;
}
```

Function expressions can be named as well, which allows them to be recursive.

```
const charsInBody = (function counter(elm) {
  if (elm.nodeType === 3) {
    // TEXT_NODE
    return elm.nodeValue.length;
  }
  let count = 0;
  for (let i = 0, child; (child = elm.childNodes[i]); i++) {
    count += counter(child);
  }
  return count;
})(document.body);
```

The name provided to a function expression as above is only available to the function's own scope. This allows more optimizations to be done by the engine and results in more readable code. The name also shows up in the debugger and some stack traces, which can save you time when debugging.

If you are used to functional programming, beware of the performance implications of recursion in JavaScript. Although the language specification specifies tail-call optimization , only JavaScriptCore (used by Safari) has implemented it, due to the difficulty of recovering stack traces and debuggability. For deep recursion, consider using iteration instead to avoid stack overflow.

## Functions are first-class objects

JavaScript functions are first-class objects. This means that they can be assigned to variables, passed as arguments to other functions, and returned from other functions. In addition, JavaScript supports closures out-of-the-box without explicit capturing, allowing you to conveniently apply functional programming styles.

```
// Function returning function
const add = (x) => (y) => x + y;
// Function accepting function
const babies = ["dog", "cat", "hen"].map((name) => `baby ${name}`);
```

Note that JavaScript functions are themselves objects — like everything else in JavaScript — and you can add or change properties on them just like we've seen earlier in the Objects section.

## Inner functions

JavaScript function declarations are allowed inside other functions. An important detail of nested functions in JavaScript is that they can access variables in their parent function's scope:

```
function parentFunc() {
  const a = 1;

  function nestedFunc() {
    const b = 4; // parentFunc can't use this
    return a + b;
  }
  return nestedFunc(); // 5
}
```

This provides a great deal of utility in writing more maintainable code. If a called function relies on one or two other functions that are not useful to any other part of your code, you can nest those utility functions inside it. This keeps the number of functions that are in the global scope down.

This is also a great counter to the lure of global variables. When writing complex code, it is often tempting to use global variables to share values between multiple functions, which leads to code that is hard to maintain. Nested functions can share variables in their parent, so you can use that mechanism to couple functions together without polluting your global namespace.

## Classes

JavaScript offers the class syntax that's very similar to languages like Java.

```
class Person {
  constructor(name) {
    this.name = name;
  }
  sayHello() {
    return `Hello, I'm ${this.name}!`;
  }
}

const p = new Person("Maria");
console.log(p.sayHello());
```

JavaScript classes are just functions that must be instantiated with the `new` operator. Every time a class is instantiated, it returns an object containing the methods and properties that the class specified. Classes don't enforce any code organization — for example, you can have functions returning classes, or you can have multiple classes per file. Here's an example of how ad hoc the creation of a class can be: it's just an expression returned from an arrow function. This pattern is called a mixin.

```
const withAuthentication = (cls) =>
  class extends cls {
    authenticate() {
      // …
    }
  };

class Admin extends withAuthentication(Person) {
  // …
}
```

Static properties are created by prepending `static`. Private properties are created by prepending a hash `#` (not `private`). The hash is an integral part of the property name. (Think about `#` as `_` in Python.) Unlike most other languages, there's absolutely no way to read a private property outside the class body — not even in derived classes.

For a detailed guide on various class features, you can read the guide page.

## Asynchronous programming

JavaScript is single-threaded by nature. There's no paralleling ; only concurrency . Asynchronous programming is powered by an event loop, which allows a set of tasks to be queued and polled for completion.

There are three idiomatic ways to write asynchronous code in JavaScript:

- Callback-based (such as `setTimeout()`)
- `Promise`-based
- `async` / `await`, which is a syntactic sugar for Promises

For example, here's how a file-read operation may look like in JavaScript:

```
// Callback-based
fs.readFile(filename, (err, content) => {
  // This callback is invoked when the file is read, which could be after a while
  if (err) {
    throw err;
  }
  console.log(content);
});
// Code here will be executed while the file is waiting to be read

// Promise-based
```

```
fs.readFile(filename)
  .then((content) => {
    // What to do when the file is read
    console.log(content);
  })
  .catch((err) => {
    throw err;
  });
// Code here will be executed while the file is waiting to be read

// Async/await
async function readFile(filename) {
  const content = await fs.readFile(filename);
  console.log(content);
}
```

The core language doesn't specify any asynchronous programming features, but it's crucial when interacting with the external environment — from asking user permissions, to fetching data, to reading files . Keeping the potentially long-running operations async ensures that other processes can still run while this one waits — for example, the browser will not freeze while waiting for the user to click a button to grant permission.

If you have an async value, it's not possible to get its value synchronously. For example, if you have a promise, you can only access the eventual result via the `then()` method. Similarly, `await` can only be used in an async context, which is usually an async function or a module. Promises are never blocking — only the logic depending on the promise's result will be deferred; everything else continues to execute in the meantime. If you are a functional programmer, you may recognize promises as monads which can be mapped with `then()` (however, they are not proper monads because they auto-flatten; i.e. you can't have a `Promise<Promise<T>>` ).

In fact, the single-threaded model has made Node.js a popular choice for server-side programming due to its non-blocking IO, making handling a large number of database or file-system requests very performant. However, CPU-bound (computationally intensive) tasks that are pure JavaScript will still block the main thread. To achieve real paralleling, you may need to use workers.

To learn more about asynchronous programming, you can read about using promises or follow the asynchronous JavaScript tutorial.

## Modules

JavaScript also specifies a module system supported by most runtimes. A module is usually a file, identified by it's file path or URL. You can use the `import` and `export` statements to exchange data between modules:

```
import { foo } from "./foo.js";

// Unexported variables are local to the module
const b = 2;

export const a = 1;
```

Unlike Haskell, Python, Java, etc., JavaScript module resolution is entirely host-defined — it's usually based on URLs or file paths, so relative file paths "just work" and are relative to the current module's path instead of some project root path.

However, the JavaScript language doesn't offer standard library modules — all core functionalities are powered by global variables like `Math` and `Intl` instead. This is due to the long history of JavaScript lacking a module system, and the fact that opting into the module system involves some changes to the runtime setup.

Different runtimes may use different module systems. For example, Node.js uses the package manager npm and is mostly file-system based, while Deno and browsers are fully URL-based and modules can be resolved from HTTP URLs.

For more information, see the modules guide page.

## Language and runtime

Throughout this page, we've constantly mentioned that certain features are language-level while others are runtime-level.

JavaScript is a general-purpose scripting language. The core language specification focuses on pure computational logic. It doesn't deal with any input/output — in fact, without extra runtime-level APIs (most notably `console.log()`), a JavaScript program's behavior is entirely unobservable.

A runtime, or a host, is something that feeds data to the JavaScript engine (the interpreter), provides extra global properties, and provides hooks for the engine to interact with the outside world. Module resolution, reading data, printing messages, sending network requests, etc. are all runtime-level operations. Since its inception, JavaScript has been adopted in various environments, such as browsers (which provide APIs like DOM), Node.js (which provides APIs like file system access ), etc. JavaScript has been successfully integrated in web (which was its primary purpose), mobile apps, desktop apps, server-side apps, serverless, embedded systems, and more. While you learn about JavaScript core features, it's also important to understand host-provided features in order to put the knowledge to use. For example, you can read about all web platform APIs, which are implemented by browsers, and sometimes non-browsers.

## Further exploration

This page offers a very basic insight into how various JavaScript features compare with other languages. If you want to learn more about the language itself and the nuances with each feature, you can read the JavaScript guide and the JavaScript reference.

There are some essential parts of the language that we have omitted due to space and complexity, but you can explore on your own:

- Inheritance and the prototype chain
- Closures
- Regular expressions
- Iteration