JavaScript fundamentals

Authors: MDN (Mozilla Developer Network) and Goran Kukic

Sources: MDN (https://developer.mozilla.org)

Table of contents

Dear fellow learners and enthusiasts,

If you've been on the hunt for a reliable and comprehensive resource to master the art of JavaScript, you've come to the right place. After years of passive observation and facing the same struggles as many of you, I embarked on a mission to create the ultimate JavaScript learning guide.

Countless YouTube tutorials, Udemy courses, and traditional educational platforms left me with unanswered questions, hindering my confidence during job interviews and actual projects. Determined to bridge this knowledge gap, I set out to craft a script that covers JavaScript in its entirety.

Allow me to introduce "JavaScript Fundamentals," an extensive PDF tutorial that spans an impressive 373 pages, divided into 29 in-depth sections.

This tutorial follows the highly regarded lesson structure from [MDN (Mozilla Developer Network)](), a reliable source for web development knowledge. To enrich the content further, I've seamlessly integrated my own invaluable insights, enhancing the overall learning experience. **Proper attribution is given to "Mozilla Contributors," and links to specific [MDN]() pages are provided at the end of each lesson where its content was used, offering easy access to the original source and complying with the terms of the Creative Commons Attribution-ShareAlike license (CC-BY-SA), v2.5 or later, which permits content reuse from [MDN]() under certain conditions.**

One notable aspect of this tutorial is its PDF format, enabling you to save and print the material at your convenience. Personally, I find comfort in learning from tangible paper resources, having transitioned to the tech industry from another field where books were my go-to learning medium.

You can also access the "JavaScript Fundamentals" tutorial on my public GitHub repository at:

https://github.com/GoranKukic/javascript-fundamentals

Rest assured, I'm committed to keeping this tutorial up to date with new versions of JavaScript as they emerge. And that's not all! My future plans include creating similar valuable resources for TypeScript and React, further aiding your journey into the world of web development.

Whether you're an aspiring developer, preparing for exams, or getting ready for job interviews, my hope is that "JavaScript Fundamentals" becomes an indispensable resource in your learning arsenal.

Happy coding! :)

Introduction

About JavaScript

JavaScript is a high-level, often just-in-time compiled language that conforms to the ECMAScript standard. It has dynamic typing, prototype-based object-orientation, and first-class functions. It is a 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 contains a standard library of objects, such as Array, Date, and Math, and a core set of language elements such as operators, control structures, and statements. Core JavaScript can be extended for a variety of purposes by supplementing it with additional objects, for example:

- **Client-side JavaScript** extends the core language by supplying objects to control a browser and its *Document Object Model* (DOM). For example, client-side extensions allow an application to place elements on an HTML form and respond to user events such as mouse clicks, form input, and page navigation.

- **Server-side JavaScript** extends the core language by supplying objects relevant to running JavaScript on a server. For example, server-side extensions allow an application to communicate with a database, provide continuity of information from one invocation to another of the application, or perform file manipulations on a server.

This means that in the browser, JavaScript can change the way the webpage (DOM) looks. And, likewise, Node.js JavaScript on the server can respond to custom requests sent by code executed in the browser.

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

JavaScript	Java
Object-oriented. No distinction between types of objects. Inheritance is through the prototype mechanism, and properties and methods can be added to any object dynamically.	Class-based. Objects are divided into classes and instances with all inheritance through the class hierarchy. Classes and instances cannot have properties or methods added dynamically.
Variable data types are not declared (dynamic typing, loosely typed).	Variable data types must be declared (static typing, strongly typed).
Cannot automatically write to a hard disk.	Can automatically write to a hard disk.

Adding JavaScript into an HTML document

The <script> Tag

In HTML, JavaScript code is inserted between <script> and </script> tags.
Code can be written **inline**, or loaded from an **external** file if you want to run the same script on several pages without writing the script on each and every page.
Scripts can be placed in the <body>, or in the <head> section of an HTML page, or in both.

JavaScript in <head> section

Scripts to be **executed when** they are **called,** or when an **event is triggered**, go in the head section. When you place a script in the head section, you will ensure that the script is loaded before anyone uses it. So, scripts that contain functions go in the head section of the document. Then we can be sure that the script is loaded before the function is called. Head scripts start **loading** really **early**, **before** the **DOM** gets to main processing; you want libraries here so they have time to get going.

For example: Inline script in `<head>` section

```
<html>
<head>
<script type="text/javascript">
   some statements
```

```
    </script>
  </head>
```

JavaScript in <body> section

Scripts to be executed when the page loads go in the body section. When you place a script in the body section it generates the content of the page.

Body scripts are **loaded while** the **DOM is building**. But there is NO guarantee the DOM will finish before your scripts run, even placed at </body> because pages load asynchronously. If you don't want your script to be placed inside a function, or if your script should write page content, it should be placed in the body section.

For example: External script in <body> section

```
<html>
<head>
</head>
<body>
    <script src="xxx.js"></script>
</body>
</html>
```

async, defer

In modern websites, scripts are often "heavier" than HTML: their download size is larger, and the processing time is also longer.

When the browser loads HTML and comes across a `<script>...</script>` tag, it can't continue building the DOM. It must execute the script right now. The same happens for external scripts `<script src="..."></script>`: **the browser must wait for the script to download, execute the downloaded script, and only then can it process the rest of the page.**

That leads to two important issues:

1. Scripts can't see DOM elements below them, so they can't add handlers etc.
2. If there's a bulky script at the top of the page, it "blocks the page". Users can't see the page content till it downloads and runs

There are some workarounds to that. For instance, we can put a script at the bottom of the page. But this solution is far from perfect. For example, the browser notices the script (and can start downloading it) only after it downloaded the full HTML document. For long HTML documents, that may be a noticeable delay.

Luckily, there are two `<script>` attributes that solve the problem for us: `defer` and `async`.

defer

The defer attribute **tells the browser not to wait for the script**. Instead, the browser will continue to process the HTML, build DOM. The **script loads "in the background", and then runs when the DOM is fully built**.

In other words:
- Scripts with defer never block the page.
- Scripts with defer always execute when the DOM is ready (but before DOMContentLoaded event).

The defer attribute is only for external scripts.

async

The async attribute means that a script is completely independent:
- The browser doesn't block async scripts (like defer).
- Other scripts don't wait for async scripts, and async scripts don't wait for them.
- DOMContentLoaded and async scripts don't wait for each other:
  - o DOMContentLoaded may happen both before an async script (if an async script finishes loading after the page is complete)
  - o …or after an async script (if an async script is short or was in HTTP-cache)

In other words, async **scripts load in the background and run when ready**. **The DOM and other scripts don't wait for them, and async scripts don't wait for them.** A fully independent script that runs when loaded.

The async attribute is only for external scripts like defer.

Summary

Both async and defer have one common thing: downloading of such scripts doesn't block page rendering. But there are also essential differences between them:

	Order	DOMContentLoaded
async	*Load-first order*. Their document order doesn't matter – which loads first runs first	Irrelevant. May load and execute while the document has not yet been fully downloaded. That happens if scripts are small or cached, and the document is long enough.
defer	*Document order* (as they go in the document).	Execute after the document is loaded and parsed (they wait if needed), right before DOMContentLoaded.

In practice, defer is used for scripts that need the whole DOM and/or their relative execution order is important. And async is used for independent scripts, like counters or ads. And their relative execution order does not matter.

Grammar, Variables & Scope

Basics

JavaScript borrows most of its syntax from Java, C, and C++, but it has also been influenced by Awk, Perl, and Python.

JavaScript is **case-sensitive** and uses the **Unicode** character set. For example, the word Früh (which means "early" in German) could be used as a variable name.

```
const Früh = "foobar";
```

But, the variable früh is not the same as Früh because JavaScript is case sensitive.

In JavaScript, instructions are called **statements** and are separated by semicolons (;).
A semicolon is not necessary after a statement if it is written on its own line. But if more than one statement on a line is desired, then they *must* be separated by semicolons.
It is considered best practice, however, to always write a semicolon after a statement, even when it is not strictly needed.

Comments

The syntax of **comments** is the same as in C++ and in many other languages:

```
// a one-line comment

/* this is a longer,
 * multi-line comment
 */
```

You can't nest block comments.
Comments behave like whitespace and are discarded during script execution.

You might also see a third type of comment syntax at the start of some JavaScript files, which looks something like this:

```
#!/usr/bin/env node.
```

This is called **hashbang comment** syntax and is a special comment used to specify the path to a particular JavaScript engine that should execute the script.

Declarations

JavaScript has three kinds of variable declarations:

1. `var` - declares a variable, optionally initializing it to a value.
2. `let` - declares a block-scoped, local variable, optionally initializing it to a value.
3. `const` - declares a block-scoped, read-only named constant.

Variables

You use variables as symbolic names for values in your application. The names of variables, called **identifiers**, conform to certain rules.

A JavaScript identifier usually starts with a letter, underscore (_), or a dollar sign ($). Subsequent characters can also be digits (0 – 9). Because JavaScript is case-sensitive, letters include the characters A through Z (uppercase) as well as a through z (lowercase).

Some examples of legal names are `Number_hits`, `temp99`, `$credit`, and `_name`.

Declaring variables

To declare (create) a variable, we need to **use the var**, **let**, or **const** keyword, followed by the name we want to provide to the variable.

The **var**, **let**, and **const** keywords **tell JavaScript to set aside a portion of memory so that we may store a specific piece of data to it later**.

The **name provided to the variable can be later used to access the location in memory assigned to the variable and retrieve the data which is stored in it**. To assign a value to the variable (initialize the variable with a value), use the assignment operator **=** to set the variable name equal to a piece of data (number, boolean, string, array, object, function, etc.)

```
/ Declare a variable named "x" using the var keyword
var x;

// Declare a variable named "y" using the let keyword
let y;

// Declare a variable named "z" using the const keyword
// Assign a value of 2 to the variable "z" using the assignment operator (=)
// Also called initializing "z" with a value of 2 (see section below on
initialization)
const z = 2;
```

You can declare variables to unpack values using the **destructuring assignment** syntax. For example, `const { bar } = foo`. This will create a variable named `bar` and assign to it the value corresponding to the key of the same name from our object `foo`.

Variables should always be declared before they are used. JavaScript used to allow assigning to undeclared variables, which creates an undeclared global variable. This is an error in strict mode and should be avoided altogether.

Initialization

In a statement like `let x = 42`, the let x part is called a **declaration**, and the `= 42` part is called an **initializer**. The declaration allows the variable to be accessed later in code without throwing a ReferenceError, while the initializer **assigns a value to the variable**. In var and let declarations, the initializer is optional. If a variable is declared without an initializer, it is assigned the value undefined.

```
let x;
console.log(x); // logs "undefined"
```

In essence, `let x = 42` is equivalent to `let x; x = 42`.

const declarations always need an initializer, because they forbid any kind of assignment after declaration, and implicitly initializing it with `undefined` is likely a programmer mistake.

```
const x; // SyntaxError: Missing initializer in const declaration
```

Variable scope

In JavaScript, there are 4 types of variable scope, namely:

1. Global Scope - The default scope for all code running in script mode.
2. Module Scope - The scope for code running in module mode.
3. Function Scope - The scope created with a function.

In addition, variables declared with let or const can belong to an additional scope:

4. Block Scope - The scope created with a pair of curly braces (a block)

Global Scope

The **variables defined outside of any function or curly brackets** are known as global variables and have global scope. Global scope means that the variables can be accessed from any part of that program, any function or conditional state can access that variable.

For example:

```
const x = "declared outside function";

exampleFunction();

function exampleFunction() {
  console.log("Inside function");
  console.log(x);
}
```

```
    console.log("Outside function");
    console.log(x);
```

Note: It is not a good practice to use global variables when they are not needed as every code block will have access to those variables.

Local Scope

If you were to define some variables inside curly brackets {} or inside a specific function then those variables are called local variables. But, with the release of ES6, the local scope was further broken down into two different scopes:

•   Function scope
•   Block scope.

Function Scope

The function scope is the accessibility of the **variables defined inside a function**, these variables **cannot be accessed from any other function** or even outside the function in the main file.

For example:

```
function abc() {
year = 2021;
// the "year" variable can be accessed inside this function
console.log("The year is "+ Year);
}
// the "year" variable cannot be accessed outside here
abc();
```

Block Scope

Block scope is also a sub-type of local scope. The block scope can be defined as the scope of the variables inside the curly brackets {}. Now, these curly brackets can be of <u>loops, or conditional statements, or something else</u>. **You are only allowed to refer to these variables from within the curly brackets {}**. However, variables created with var are not block-scoped, but only local to the function (or global scope) that the block resides within.

Imagine a nested situation:

A block of code enclosed with curly brackets {} containing some block variables.

```
{
  let a = 10;
  const b = 5;
}
```

This block of code is itself inside a function.

```
function addition() {
  {
    let a = 10;
    const b = 5;
  }
}
```

Now when we try to access the variables from inside the function but outside of that specific block.

```
function addition() {
  {
    let a = 10;
    const b = 5;
  }
  console.log(a + b);
}
```

And to access this function we need to invoke it by:

```
addition();
```

Then **we'll be met with an error**, even though they are local variables of the same function.

The keywords **let** and **const** are used to define block variables.
If you are trying to refer to the local variable from outside the function you will get an error **"status (variable name) is not defined".**

Importance of let and const keywords while declaring block variables

If you declare a variable inside **curly brackets {}** without using the **let** and the **const** keywords then the variable will be declared as a "**Global variable**". This is due to the fact <u>keywords have their scopes predefined</u>.

Global variables

Global variables are in fact **properties of the global object**.

In web pages, the global object is `window`, so you can set and access global variables using the `window.variable` syntax. In all environments, you can use the `globalThis` variable (which itself is a global variable) to access global variables.

Consequently, you can access global variables declared in one window or frame from another window or frame by specifying the `window` or `frame` name. For example, if a variable called `phoneNumber` is declared in a document, you can refer to this variable from an `iframe` as `parent.phoneNumber`.

Variable hoisting

JavaScript only hoists declarations, not initializations!

This means that initialization doesn't happen until the associated line of code is executed. Until that point in the execution is reached the variable has its *default* initialization (undefined for a variable declared using var, otherwise uninitialized).

var hoisting

Here we declare and then initialize the value of a `var` after using it. **The default initialization of the `var` is `undefined`**.

```
console.log(num); // Returns 'undefined' from hoisted var declaration (not 6)
var num; // Declaration
num = 6; // Initialization
console.log(num); // Returns 6 after the line with initialization is executed.
```

The same thing happens if we declare and initialize the variable in the same line.

```
console.log(num); // Returns 'undefined' from hoisted var declaration (not 6)
var num = 6; // Initialization and declaration.
console.log(num); // Returns 6 after the line with initialization is executed.
```

If we forget the declaration altogether (and only initialize the value) the variable isn't hoisted. Trying to read the variable before it is initialized results in a ReferenceError exception.

```
console.log(num); /* Throws ReferenceError exception
                    - the interpreter doesn't know about `num` */
num = 6; // Initialization
```

Note however that initialization also causes declaration (if not already declared). The code snippet below will work, because even though it isn't hoisted, the variable is initialized and effectively declared before it is used.

```
a = "Cran"; // Initialize a
b = "berry"; // Initialize b

console.log(`${a}${b}`); // 'Cranberry'
```

let and const hoisting

Variables declared with let and const are also **hoisted but**, unlike var, are **not initialized with a default value**. An exception will be thrown if a variable declared with let or const is read before it is initialized.

```
console.log(num); /* Throws ReferenceError exception as the variable value
                     is uninitialized */
let num = 6; // Initialization
```

Note that it is the order in which code is executed that matters, not the order in which it is written in the source file. The code will succeed provided the line that initializes the variable is executed before any line that reads it.

Constants

You can create a read-only, named constant with the const keyword.
A constant cannot change value through assignment or be re-declared while the script is running.

However, **const only** prevents *re-assignments* **but doesn't prevent** *mutations*. The properties of objects assigned to constants are not protected, so the following statement is executed without problems.

```
const MY_OBJECT = { key: 'value' };
```

```
    MY_OBJECT.key = 'otherValue';
```

Also, the contents of an array are not protected, so the following statement is executed without problems.

```
const MY_ARRAY = ['HTML','CSS'];
MY_ARRAY.push('JAVASCRIPT');
console.log(MY_ARRAY); // ['HTML', 'CSS', 'JAVASCRIPT'];
```

Difference between var, let, and const keyword

1. Scope

var	let	const
Variables declared with var are in the **function scope**.	Variables declared as let are in the **block scope**.	Variables declared as const are in the **block scope**.
``` function f1(){ var a=10; } console.log(a) ```	``` for (let i = 0; i < 3; i++){ console.log(i); } console.log(i); ```	``` { const x = 2; console.log(x); } console.log(x); ```

- In the var tab, when you will run the code, you will see an error as a is not defined because var variables are only accessible in the function scope.
- In the let tab, when you run the code, you will see an error as i is not defined because let variables are only accessible within the block they are declared.
- In the const tab, when you run the code, you will see that x was defined under the block was accessible, but when you try to access x outside that block, you will get an error.

2. Hoisting

Hoisting means that you can define a variable before its declaration.

var	let	const
Allowed	Not allowed	Not allowed

`x = 8;`
`console.log(x);`
`var x;`
	`x = 8;`
	`console.log(x);`
	`let x;`
		`x = 8;`
		`console.log(x);`
		`const x;`

- In the var tab, when you run the code, you will see there is no error and that we can declare var variables after defining them.
- In the let tab, when you run the code, you will get an error as let variables do not support hoisting.
- In the const tab, when you run the code, you will get an error as const variables do not support hoisting.

3. Reassign the value

To reassign a value is to reassign the value of a variable.

var	let	const
Allowed	Allowed	Not allowed

`var v1 = 1;`
`v1 = 30;`
`console.log(v1);`
	`let v1 = 1;`
	`v1 = 30;`
	`console.log(v1);`
		`const v1 = 1;`
		`v1 = 30;`
		`console.log(v1);`

- In the var and let tab, when you run the code, you will see that there is no error and we can define new values to the var and let variables.

- In the const tab, when you run the code, you will get an error as a const variables value cannot be reassigned.

4. Redeclaration of the variable

The redeclaration of a variable means that you can declare the variable again.

var	let	const
Allowed	Not allowed	Not allowed

`var v1 = 1;`
`var v1 = 30;`
`console.log(v1);`
	`let v1 = 1;`
	`let v1 = 30;`
	`console.log(v1);`
		`const v1 = 1;`
		`const v1 = 30;`
		`console.log(v1);`

- In the var tab, when you run the code, you will see that there is no error as we allowed to declare the same variable again.
- In the let and the const tabs, when you run the code, you will get an error as the let and const variables do not allow you to redeclare them again.

Data types in JavaScript

There are 8 data types in JavaScript:

1.	String	Primitive
2.	Number	Primitive
3.	Boolean	Primitive
4.	Undefined	Primitive
5.	Null	Primitive
6.	Object	Non-primitive / Reference
7.	BigInt	Primitive
8.	Symbol	Primitive

String

The String is used for storing **text**. In JavaScript, strings are surrounded by quotes:

- Single quotes: `'Hello'`
- Double quotes: `"Hello"`
- Backticks: `` `Hello` ``

Number

The Number represents **integer and floating numbers** (decimals and exponentials):

```
const number1 = 3;
const number2 = 3.433;
const number3 = 3e5 // 3 * 10^5
```

A number type can also be *+Infinity, -Infinity*, and *NaN* (not a number).

Boolean

This data type represents logical entities. Boolean represents one **of two values: true or false**. It is easier to think of it as a yes/no switch. For example:

```
const dataChecked = true;
const valueCounted = false;
```

Undefined

The undefined data type represents **value that is not assigned**. If a variable is declared but the value is not assigned, then the value of that variable will be undefined. For example:

```
let name;
console.log(name); // undefined
```

It is also possible to explicitly assign a variable value undefined. For example:

```
let name = undefined;
console.log(name); // undefined
```

It is recommended not to explicitly assign undefined to a variable. Usually, null is used to assign 'unknown' or 'empty' value to a variable.

Null

In JavaScript, null is a special value that represents **empty or unknown value**. For example:

```
const number = null;
```

The code above suggests that the number variable is empty.
null is not the same as NULL or Null.

Object

An object is a complex data type that allows us to store **collections of data**. For example:

```javascript
const student = {
    firstName: 'ram',
    lastName: null,
    class: 10
};
```

Array, Object, function, RegEx, Date are types of Object. Object represents refrence / non-primitive / composite data type in JavaScript.

BigInt

In JS, Number type can only represent numbers less than (2^{53} - 1) and more than -(2^{53} - 1). However, if you need to use a larger number than that, you can use the BigInt data type.
A BigInt number is created by appending n to the end of an integer. For example:

```javascript
// BigInt value
const value1 = 900719925124740998n;

// Adding two big integers
const result1 = value1 + 1n;
console.log(result1); // "900719925124740999n"

const value2 = 900719925124740998n;

// Error! BitInt and number cannot be added
const result2 = value2 + 1;
console.log(result2);

Output
900719925124740999n
Uncaught TypeError: Cannot mix BigInt and other types
```

BigInt was introduced in the newer version of JavaScript and is **not supported by many browsers including Safari**.

Symbol

This data type was introduced in a newer version of JavaScript (from ES2015).
A value having the data type Symbol can be referred to as a symbol value. Symbol is an immutable primitive value that is unique. For example:

```
// two symbols with the same description
const value1 = Symbol('hello');
const value2 = Symbol('hello');
```

Though `value1` and `value2` both contain `'hello'`, they are different as they are of the Symbol type.

Primitive vs non-primitive/reference data types/values

Data types in JavaScript are classified into 2 types:

- **Primitive**: - String, Boolean, Number, BigInt, Null, Undefined, Symbol
- **Non-Primitive**: - Object (array, functions) also called object references.

The fundamental difference between primitives and non-primitive is that primitives are immutable and non-primitive are mutable.

Primitives

They are **stored in memory (normally stack)**, variable stores the value itself.
Copying a variable (= assign to the different variable) copies the value.
Primitives are known as being immutable data types because there is **no way to change a primitive value once it gets created**.

```
var string = 'This is a string.';
string = 'H'console.log(string)
console.log(string) // Output -> 'This is a string.'
```

A variable that stored primitive value **can only be reassigned to a new value** (instead of changing the original value, JavaScript creates a new value – and creates a new location in memory), as shown in the example below:

```
var string = 'This is a string.';
string = 'Hello world'
console.log(string) // Output -> 'Hello world'
```

Primitives are **compared by value**. Two values are strictly equal if they have the same value.

```
var number1 = 5;
var number2 = 5;
number1 === number 2; // true
var string1 = 'This is a string.';
var string2 = 'This is a string.';
string1 === string2; // true
```

Here the variable for primitives stores the value, so they are always copied or passed by value.

Non-Primitives / Reference

Stored in memory (heap) variable stores a pointer (address) to a location in memory.
Copying a variable (= assign to a different variable) copies the pointer/reference.
Non-primitive values are mutable data types. The **value of an object can be changed** after it gets created, the location in memory is still the same.

```
var arr = [ 'one', 'two', 'three', 'four', 'five' ];
arr[1] = 'TWO';
console.log(arr)
// [ 'one', 'TWO', 'three', 'four', 'five' ];
```

Objects are **not compared by value**. This means that even if two objects have the same properties and values, they are not strictly equal. The same goes for arrays. Even if they have the same elements that are in the same order, they are not strictly equal.

```
var obj1 = { 'cat': 'playful' };
var obj2 = { 'cat': 'playful' };
obj1 === obj2;  // falsevar arr1 = [ 1, 2, 3, 4, 5 ];
var arr2 = [ 1, 2, 3, 4, 5 ];
arr1 === arr2;  // false
```

Non-primitive values can also be referred to as reference types because they are being **compared by reference instead of value**. Two objects are only strictly equal if they refer to the same underlying object.

```
var obj3 = { 'car': 'purple' }
var obj4 = obj3;
obj3 === obj4;  // true
```

Summary
- Primitive values are immutable
- Primitive values compared by value
- Non-primitive values are mutable
- Non-primitive compare by reference not value

Data type conversion

JavaScript is a dynamically typed language. This means you don't have to specify the data type of a variable when you declare it. It also means that data types are automatically converted as-needed during script execution.

Numbers and the '+' operator

In expressions involving numeric and string values with the + operator, JavaScript converts numeric values to strings.

```
x = "The answer is " + 42; // "The answer is 42"
y = 42 + " is the answer"; // "42 is the answer"
z = "37" + 7; // "377"
```

With all other operators, JavaScript does not convert numeric values to strings. For example:

```
"37" - 7; // 30
"37" * 7; // 259
```

Converting strings to numbers

In the case that a value representing a number is in memory as a string, there are methods for conversion.

```
parseInt()
parseFloat()
```

`parseInt` only returns whole numbers, so its use is diminished for decimals.

An alternative method of retrieving a number from a string is with the + (unary plus) operator:

```
"1.1" + "1.1" // '1.11.1'
(+"1.1") + (+"1.1"); // 2.2
// Note: the parentheses are added for clarity, not required.
```

Literals

Literals represent values in JavaScript. **Literals are constant values that can be assigned to the variables** that are called literals or constants. This section describes the following types of literals:

- Array literals
- Boolean literals
- Numeric literals
- Object literals
- RegExp literals
- String literals

Array literals

An array literal is a list of zero or more expressions, each of which represents an array element, enclosed in square brackets ([]). When you create an array using an array literal, it is initialized with the specified values as its elements, and its length is set to the number of arguments specified.

The following example creates the coffees array with three elements and a length of three:

```
const coffees = ["French Roast", "Colombian", "Kona"];
```

Extra commas in array literals

If you put two commas in a row in an array literal, the array leaves an empty slot for the unspecified element. The following example creates the fish array:

```
const fish = ["Lion", , "Angel"];
```

When you log this array, you will see:

```
console.log(fish);
// [ 'Lion', <1 empty item>, 'Angel' ]
```

Note that the second item is "empty", which is not exactly the same as the actual undefined value. When using array-traversing methods like `Array.prototype.map`, empty slots are skipped. However, index-accessing fish[1] still returns undefined.

In the following example, the length of the array is four, and `myList[1]` and `myList[3]` are missing. Only the last comma is ignored.

```
const myList = ["home", , "school", ,];
```

However, when writing your own code, you should explicitly declare the missing elements as undefined, or at least insert a comment to highlight its absence. Doing this increases your code's clarity and maintainability.

```
const myList = ["home", /* empty */, "school", /* empty */, ];
```

Boolean literals

The Boolean type has two literal values: `true` and `false`.

Numeric literals

JavaScript numeric literals include integer literals in different bases as well as floating-point literals in base-10.

Integer literals

Integer and BigInt literals can be written in decimal (base 10), hexadecimal (base 16), octal (base 8) and binary (base 2).

- A *decimal* integer literal is a sequence of digits without a leading 0 (zero).
- A leading 0 (zero) on an integer literal, or a leading 0o (or 0O) indicates it is in *octal*. Octal integer literals can include only the digits 0 – 7.
- A leading 0x (or 0X) indicates a *hexadecimal* integer literal. Hexadecimal integers can include digits (0 – 9) and the letters a – f and A – F. (The case of a character does not change its value. Therefore: 0xa = 0xA = 10 and 0xf = 0xF = 15.)
- A leading 0b (or 0B) indicates a *binary* integer literal. Binary integer literals can only include the digits 0 and 1.
- A trailing n suffix on an integer literal indicates a BigInt literal. The integer literal can use any of the above bases. Note that leading-zero octal syntax like 0123n is not allowed, but 0o123n is fine.

A floating-point literal can have the following parts:

- An unsigned decimal integer,
- A decimal point (.),
- A fraction (another decimal number),
- An exponent.

For example:

```
3.1415926
.123456789
3.1E+12
.1e-23
```

Object literals

An object literal is a list of zero or more pairs of property names and associated values of an object, enclosed in curly braces ({}).

The following is an example of an object literal. The first element of the car object defines a property, myCar, and assigns to it a new string, "Saturn"; the second element, the getCar property, is immediately assigned the result of invoking the function (carTypes("Honda")); the third element, the special property, uses an existing variable (sales).

```
const sales = "Toyota";

function carTypes(name) {
  return name === "Honda" ? name : `Sorry, we don't sell ${name}.`;
}

const car = { myCar: "Saturn", getCar: carTypes("Honda"), special: sales };

console.log(car.myCar); // Saturn
console.log(car.getCar); // Honda
console.log(car.special); // Toyota
```

Object property names can be any string, including the empty string. If the property name would not be a valid JavaScript identifier or number, it must be enclosed in quotes.

Property names that are not valid identifiers cannot be accessed as a dot (.) property.

```
const unusualPropertyNames = {
  '': 'An empty string',
  '!': 'Bang!'
}
console.log(unusualPropertyNames.'');   // SyntaxError: Unexpected string
console.log(unusualPropertyNames.!);    // SyntaxError: Unexpected token !

Instead, they must be accessed with the bracket notation ([]).

console.log(unusualPropertyNames[""]); // An empty string
console.log(unusualPropertyNames["!"]); // Bang!
```

Enhanced Object literals

Object literals support a range of shorthand syntaxes that include setting the prototype at construction, shorthand for foo: foo assignments, defining methods, making super calls, and computing property names with expressions.

```
const obj = {
  // __proto__
  __proto__: theProtoObj,
  // Shorthand for 'handler: handler'
  handler,
  // Methods
  toString() {
    // Super calls
    return "d " + super.toString();
  },
  // Computed (dynamic) property names
  ["prop_" + (() => 42)()]: 42,
};
```

RegExp literals

A regex literal is a pattern enclosed between slashes. The following is an example of a regex literal.

```
const re = /ab+c/;
```

String literals

A string literal is zero or more characters enclosed in double (") or single (') quotation marks. A string must be delimited by quotation marks of the same type (that is, either both single quotation marks or both double quotation marks).

```
'foo'
"bar"
'1234'
'one line \n another line'
"Joyo's cat"
```

Template literals

Template literals are also available. Template literals are enclosed by the back-tick (`) (grave accent) character instead of double or single quotes.

Template literals provide syntactic sugar for constructing strings.

```
/ Basic literal string creation
`In JavaScript '\n' is a line-feed.`

// Multiline strings
`In JavaScript, template strings can run
 over multiple lines, but double and single
 quoted strings cannot.`

// String interpolation
const name = 'Lev', time = 'today';
`Hello ${name}, how are you ${time}?`
```

Tagged templates

A more advanced form of template literals are tagged templates.
Tags allow you to parse template literals with a function. The first argument of a tag function contains an array of string values. The remaining arguments are related to the expressions.
The tag function can then perform whatever operations on these arguments you wish, and return the manipulated string.

```
const person = "Mike";
const age = 28;

function myTag(strings, personExp, ageExp) {
  const str0 = strings[0]; // "That "
  const str1 = strings[1]; // " is a "
  const str2 = strings[2]; // "."

  const ageStr = ageExp > 99 ? "centenarian" : "youngster";

  // We can even return a string built using a template literal
  return `${str0}${personExp}${str1}${ageStr}${str2}`;
}

const output = myTag`That ${person} is a ${age}.`;

console.log(output);
// That Mike is a youngster.
```

Control flow and error handling

JavaScript supports a compact set of statements, specifically control flow statements, that you can use to incorporate a great deal of interactivity in your application. This chapter provides an overview of these statements.

The JavaScript reference contains exhaustive details about the statements in this chapter. The semicolon (;) character is used to separate statements in JavaScript code.

Block statement

The most basic statement is a *block statement*, which is used to group statements. The block is delimited by a pair of curly brackets:

```
{
  statement1;
  statement2;
  // …
  statementN;
}
```

Block statements are commonly used with control flow statements (`if, for, while`).

```
while (x < 10) {
  x++;
}
```

Conditional statements

A conditional statement is a set of commands that executes if a specified condition is true. JavaScript supports two conditional statements: `if...else` and `switch`.

If...else statement

Use the `if` statement to execute a statement if a logical condition is `true`. Use the optional `else` clause to execute a statement if the condition is `false`.

An if statement looks like this**:**

```
if (condition) {
  statement1;
} else {
  statement2;
}
```

If the `condition` evaluates to `true`, `statement_1` is executed. Otherwise, `statement_2` is executed. `statement_1` and `statement_2` can be any statement, including further nested if statements.

You can also compound the statements using `else  if` to have multiple conditions tested in sequence, as follows:

```
if (condition1) {
  statement1;
} else if (condition2) {
  statement2;
} else if (conditionN) {
  statementN;
} else {
  statementLast;
}
```

To execute multiple statements, group them within a block statement (`{ /* … */ }`).

It's **not good** practice to have `if...else` with an assignment like `x = y` as a condition:

```
if (x = y) {
  // statements here
}
```

Falsy values

The following values evaluate to false (also known as Falsy values):

- `false`
- `undefined`
- `null`
- `0`
- `NaN`
- `the empty string ("")`

All other values—including all objects—evaluate to true when passed to a conditional statement.

Do not confuse the primitive boolean values true and false with the true and false values of the Boolean object!

```
const b = new Boolean(false);
if (b) {
  // this condition evaluates to true
}
if (b == true) {
  // this condition evaluates to false
}
```

switch statement

A switch statement allows a program to evaluate an expression and attempt to match the expression's value to a case label. If a match is found, program executes the associated statement.

```
switch (expression) {
  case label1:
    statements1;
    break;
  case label2:
    statements2;
    break;
  // …
  default:
    statementsDefault;
}
```

- The program first looks for a case clause with a label matching the value of expression and then transfers control to that clause, executing the associated statements.
- If no matching label is found, the program looks for the optional default clause:

  - If a default clause is found, the program transfers control to that clause, executing the associated statements.
  - If no default clause is found, the program resumes execution at the statement following the end of switch.
  - (By convention, the default clause is written as the last clause, but it does not need to be so.)

break statements

The optional break statement associated with each case clause **ensures that the program breaks out of switch block** once the matched statement is executed.
If break is omitted, the program continues execution inside the switch statement (and will evaluate the next case, and so on).

In the following example, if fruitType evaluates to 'Bananas', the program matches the value with case 'Bananas' and executes the associated statement. When break is encountered, the program exits the switch and continues execution from the statement following switch. If break were omitted, the statement for case 'Cherries' would also be executed.

```
switch (fruitType) {
  case "Apples":
    console.log("Apples are $0.32 a pound.");
    break;
  case "Bananas":
    console.log("Bananas are $0.48 a pound.");
    break;
  case "Cherries":
    console.log("Cherries are $3.00 a pound.");
    break;
  default:
    console.log(`Sorry, we are out of ${fruitType}.`);
}
console.log("Is there anything else you'd like?");
```

Exception handling statements

You can throw exceptions using the `throw` statement and handle them using the `try...catch` statements.

- `throw` statement
- `try...catch` statement

Exception types

Just about any object can be thrown in JavaScript. Nevertheless, not all thrown objects are created equal. While it is common to throw numbers or strings as errors, it is frequently more effective to use one of the exception types specifically created for this purpose:

- ECMAScript exceptions
- `DOMException` and `DOMError`

throw statement

Use the throw statement to throw an exception.
You may throw any expression, not just expressions of a specific type. The following code throws several exceptions of varying types:

```
throw "Error2"; // String type
throw 42; // Number type
throw true; // Boolean type
throw {
  toString() {
    return "I'm an object!";
  },
};
```

try...catch statement

The try...catch statement marks a block of statements to try, and specifies one or more responses should an exception be thrown. If an exception is thrown, the try...catch statement catches it.

The try...catch statement consists of a try block, which contains one or more statements, and a catch block, containing statements that specify what to do if an exception is thrown in the try block.

The following example uses a try...catch statement. The example calls a function that retrieves a month name from an array based on the value passed to the function. If the value does not correspond to a month number (1 - 12), an exception is thrown with the value 'InvalidMonthNo' and the statements in the catch block set the monthName variable to 'unknown'.

```javascript
function getMonthName(mo) {
  mo--; // Adjust month number for array index (so that 0 = Jan, 11 = Dec)
  const months = [
    "Jan", "Feb", "Mar", "Apr", "May", "Jun",
    "Jul", "Aug", "Sep", "Oct", "Nov", "Dec",
  ];
  if (months[mo]) {
    return months[mo];
  } else {
    throw new Error("InvalidMonthNo"); // throw keyword is used here
  }
}

try {
  // statements to try
  monthName = getMonthName(myMonth); // function could throw exception
} catch (e) {
  monthName = "unknown";
  logMyErrors(e); // pass exception object to error handler (i.e. your own function)
}
```

The catch block

You can use a catch block to handle all exceptions that may be generated in the try block.

```javascript
catch (catchID) {
  statements
}
```

The catch block specifies an identifier (catchID in the preceding syntax) that holds the value specified by the throw statement. You can use this identifier to get information about the exception that was thrown.

For example, the following code throws an exception. When the exception occurs, control transfers to the catch block.

```
try {
  throw "myException"; // generates an exception
} catch (err) {
  // statements to handle any exceptions
  logMyErrors(err); // pass exception object to error handler
}
```

When logging errors to the console inside a catch block, using `console.error()` rather than `console.log()` is advised for debugging. It formats the message as an error, and adds it to the list of error messages generated by the page.

The finally block

The `finally` block contains statements to be executed after the `try` and `catch` blocks execute.

It is also important to note that the `finally` block will execute whether or not an exception is thrown. If an exception is thrown, however, the statements in the `finally` block execute even if no catch block handles the exception that was thrown.

The following example opens a file and then executes statements that use the file. (Server-side JavaScript allows you to access files.) If an exception is thrown while the file is open, the `finally` block closes the file before the script fails. Using `finally` here ensures that the file is never left open, even if an error occurs.

```
openMyFile();
try {
  writeMyFile(theData); // This may throw an error
} catch (e) {
  handleError(e); // If an error occurred, handle it
} finally {
  closeMyFile(); // Always close the resource
}
```

If the `finally` block returns a value, this value becomes the return value of the entire try...catch...finally production, regardless of any return statements in the `try` and `catch` blocks.

Nesting try...catch statements

You can nest one or more `try...catch` statements.
If an inner try block does *not* have a corresponding catch block:

1. it *must* contain a finally block, and
2. the enclosing try...catch statement's catch block is checked for a match.

Utilizing Error objects

Depending on the type of error, you may be able to use the `name` and `message` properties to get a more refined message.

The name property provides the general class of Error (such as `DOMException` or `Error`), while `message` generally provides a more succinct message than one would get by converting the error object to a string.

If you are throwing your own exceptions, in order to take advantage of these properties (such as if your `catch` block doesn't discriminate between your own exceptions and system ones), you can use the `Error` constructor.

```
function doSomethingErrorProne() {
  if (ourCodeMakesAMistake()) {
    throw new Error("The message");
  } else {
    doSomethingToGetAJavaScriptError();
  }
}

try {
  doSomethingErrorProne();
} catch (e) {
  // Now, we actually use `console.error()`
  console.error(e.name); // 'Error'
  console.error(e.message); // 'The message', or a JavaScript error message
}
```

Loops and iteration

Loops offer a quick and easy way to do something repeatedly. You can think of a loop as a computerized version of the game where you tell someone to take X steps in one direction, then Y steps in another.

There are many different kinds of loops, but they all essentially do the same thing: they repeat an action some number of times.

The statements for loops provided in JavaScript are:

1. for statement
2. do...while statement
3. while statement
4. labeled statement
5. break statement
6. continue statement
7. for...in statement
8. for...of statement

for statement

A `for` loop repeats until a specified condition evaluates to false. The JavaScript `for` loop is similar to the Java and C `for` loop.

A for statement looks as follows:

```
for (initialization; condition; afterthought)
  statement
```

When a for loop executes, the following occurs:

1. The initializing expression `initialization`, if any, is executed. This expression usually initializes one or more loop counters, but the syntax allows an expression of any degree of complexity. This expression can also declare variables.

2. The `condition` expression is evaluated. If the value of the `condition` is true, the loop statements execute. Otherwise, the `for` loop terminates. (If the condition expression is omitted entirely, the `condition` is assumed to be true.)
3. The statement executes. To execute multiple statements, use a block statement ({ }) to group those statements.
4. If present, the update expression `afterthought` is executed.
5. Control returns to Step 2.

Here, the for statement declares the variable i and initializes it to 0. It checks that i is less than the number of options in the `<select>` element, performs the succeeding `if` statement, and increments i by 1 after each pass through the loop.

```
function countSelected(selectObject) {
  let numberSelected = 0;
  for (let i = 0; i < selectObject.options.length; i++) {
    if (selectObject.options[i].selected) {
      numberSelected++;
    }
  }
  return numberSelected;
}

const btn = document.getElementById("btn");

btn.addEventListener("click", () => {
  const musicTypes = document.selectForm.musicTypes;
  console.log(`You have selected ${countSelected(musicTypes)} option(s).`);
});
```

do...while statement

The `do...while` statement repeats **until a specified condition evaluates to false**.

```
do
  statement
while (condition);
```

`statement` is always executed once before the condition is checked. (To execute multiple statements, use a block statement ({ }) to group those statements.)

If condition is true, the statement executes again. At the end of every execution, the condition is checked. When the condition is false, execution stops, and control passes to the statement following do...while.

<div align="center">Example</div>

In the following example, the do loop iterates at least once and reiterates until i is no longer less than 5.

```
let i = 0;
do {
  i += 1;
  console.log(i);
} while (i < 5);
```

while statement

A while statement executes its statements **as long as a specified condition evaluates to true**. A while statement looks as follows:

```
while (condition)
  statement
```

If the condition becomes false, statement within the loop stops executing and control passes to the statement following the loop.

The condition test occurs before statement in the loop is executed. If the condition returns true, statement is executed and the condition is tested again. If the condition returns false, execution stops, and control is passed to the statement following while.

To execute multiple statements, use a block statement ({ }) to group those statements.

<div align="center">Example 1</div>

The following while loop iterates as long as n is less than 3:

```
let n = 0;
let x = 0;
while (n < 3) {
```

```
    n++;
    x += n;
}
```

With each iteration, the loop increments n and adds that value to x. Therefore, x and n take on the following values:

- After the first pass: n = 1 and x = 1
- After the second pass: n = 2 and x = 3
- After the third pass: n = 3 and x = 6

After completing the third pass, the condition n < 3 is no longer true, so the loop terminates.

Example 2

Avoid infinite loops. Make sure the condition in a loop eventually becomes `false`—otherwise, the loop will never terminate! The statements in the following `while` loop execute forever because the condition never becomes `false`:

```
// Infinite loops are bad!
while (true) {
  console.log("Hello, world!");
}
```

labeled statement

A `label` provides a statement with an identifier that lets you refer to it elsewhere in your program. For example, you can use a label to identify a loop, and then use the `break` or `continue` statements to indicate whether a program should interrupt the loop or continue its execution.

The syntax of the labeled statement looks like the following:

```
label:
  statement
```

The value of `label` may be any JavaScript identifier that is not a reserved word. The `statement` that you identify with a label may be any statement.

In this example, the label `markLoop` identifies a `while` loop.

```
markLoop: while (theMark) {
  doSomething();
}
```

break statement

Use the `break` statement to terminate a loop, `switch`, or in conjunction with a labeled statement.

- When you use `break` without a label, it terminates the innermost enclosing `while,` `do-while,` `for,` or `switch` immediately and transfers control to the following statement.
- When you use `break` with a label, it terminates the specified labeled statement.

The syntax of the `break` statement looks like this:

```
break;
break label;
```

1. The first form of the syntax terminates the innermost enclosing loop or `switch`.
2. The second form of the syntax terminates the specified enclosing labeled statement.

Example 1

The following example iterates through the elements in an array until it finds the index of an element whose value is `theValue`:

```
for (let i = 0; i < a.length; i++) {
  if (a[i] === theValue) {
    break;
  }
}
```

Example 2: Breaking to a label

```
let x = 0;
let z = 0;
```

```
labelCancelLoops: while (true) {
  console.log("Outer loops: ", x);
  x += 1;
  z = 1;
  while (true) {
    console.log("Inner loops: ", z);
    z += 1;
    if (z === 10 && x === 10) {
      break labelCancelLoops;
    } else if (z === 10) {
      break;
    }
  }
}
```

continue statement

The `continue` statement can be used to restart a `while`, `do-while`, `for`, or `label` statement.

- When you use `continue` without a label, it terminates the current iteration of the innermost enclosing `while`, `do-while`, or `for` statement and continues execution of the loop with the next iteration. In contrast to the `break` statement, `continue` does not terminate the execution of the loop entirely. In a `while` loop, it jumps back to the condition. In a `for` loop, it jumps to the `increment-expression`.
- When you use `continue` with a label, it applies to the looping statement identified with that label.

The syntax of the `continue` statement looks like the following:

```
continue;
continue label;
```

Example 1

The following example shows a `while` loop with a `continue` statement that executes when the value of i is 3. Thus, n takes on the values 1, 3, 7, and 12.

```
let i = 0;
let n = 0;
while (i < 5) {
  i++;
  if (i === 3) {
    continue;
  }
```

```
    n += i;
    console.log(n);
  }
  //1,3,7,12
```

If you comment out the `continue;`, the loop would run till the end and you would see `1,3,6,10,15`.

Example 2

A statement labeled `checkiandj` contains a statement labeled `checkj`. If `continue` is encountered, the program terminates the current iteration of `checkj` and begins the next iteration. Each time `continue` is encountered, `checkj` reiterates until its condition returns `false`. When `false` is returned, the remainder of the `checkiandj` statement is completed, and `checkiandj` reiterates until its condition returns `false`. When `false` is returned, the program continues at the statement following `checkiandj`.

If `continue` had a label of `checkiandj`, the program would continue at the top of the `checkiandj` statement.

```
  let i = 0;
  let j = 10;
  checkiandj: while (i < 4) {
    console.log(i);
    i += 1;
    checkj: while (j > 4) {
      console.log(j);
     j -= 1;
      if (j % 2 === 0) {
        continue checkj;
      }
      console.log(j, " is odd.");
    }
    console.log("i = ", i);
    console.log("j = ", j);
  }
```

for...in statement

The `for...in` statement iterates a specified variable over all the enumerable properties of an object. For each distinct property, JavaScript executes the specified statements. A `for...in` statement looks as follows:

45

```
for (variable in object)
  statement
```

Example

The following function takes as its argument an object and the object's name. It then iterates over all the object's properties and returns a string that lists the property names and their values.

```
function dumpProps(obj, objName) {
  let result = "";
  for (const i in obj) {
    result += `${objName}.${i} = ${obj[i]}<br>`;
  }
  result += "<hr>";
  return result;
}
```

For an object car with properties make and model, result would be:

```
car.make = Ford
car.model = Mustang
```

Arrays

Although it may be tempting to use this as a way to iterate over Array elements, the for...in statement will return the name of your user-defined properties in addition to the numeric indexes.

Therefore, it is better to use a traditional for loop with a numeric index when iterating over arrays, because the for...in statement iterates over user-defined properties in addition to the array elements, if you modify the Array object (such as adding custom properties or methods).

for...of statement

The for...of statement creates a loop Iterating over iterable objects (including Array, Map, Set, arguments object and so on), invoking a custom iteration hook with statements to be executed for the value of each distinct property.

```
for (variable of object)
  statement
```

The following example shows the difference between a `for...of` loop and a `for...in` loop. While `for...in` iterates over property names, `for...of` iterates over property values:

```
const arr = [3, 5, 7];
arr.foo = "hello";

for (const i in arr) {
  console.log(i);
}
// "0" "1" "2" "foo"

for (const i of arr) {
  console.log(i);
}
// Logs: 3 5 7
```

The `for...of` and for...in statements can also be used with **destructuring**. For example, you can simultaneously loop over the keys and values of an object using `Object.entries()`.

```
const obj = { foo: 1, bar: 2 };

for (const [key, val] of Object.entries(obj)) {
  console.log(key, val);
}
// "foo" 1
// "bar" 2
```

Functions

Functions are one of the fundamental building blocks in JavaScript. A function in JavaScript is a **set of statements that performs a task or calculates a value**, but for a procedure to qualify as a function, it should **take some input** and **return an output** where there is some obvious relationship between the input and the output. To use a function, you must define it somewhere in the scope from which you wish to call it.

Defining functions

Function declarations

A **function definition** (also called a **function declaration**, or **function statement**) consists of the function keyword, followed by:

- The name of the function.
- A list of parameters to the function, enclosed in parentheses and separated by commas.
- The JavaScript statements that define the function, enclosed in curly brackets, { /* ... */ }

For example:

```
function square(number) {
  return number * number;
}
```

Parameters are essentially passed to functions **by value** — so if the code within the body of a function assigns a completely new value to a parameter that was passed to the function, **the change is not reflected globally or in the code which called that function**.

When you pass an object as a parameter, if the function changes the object's properties, that change is visible outside the function.

Function expressions

While the function declaration above is syntactically a statement, functions can also be created by a function expression.

Such a function can be **anonymous**; it does not have to have a name.

For example:

```
const square = function (number) {
  return number * number;
}
const x = square(4); // x gets the value 16
```

However, a name can be provided with a function expression. Providing a name allows the function to refer to itself, and also makes it easier to identify the function in a debugger's stack traces:

```
const factorial = function fac(n) {
  return n < 2 ? 1 : n * fac(n - 1);
}

console.log(factorial(3))
```

Function expressions are <u>convenient when passing a function as an argument to another function.</u> The following example shows a map function that should receive a function as the first argument and an array as a second argument:

```
function map(f, a) {
  const result = new Array(a.length);
  for (let i = 0; i < a.length; i++) {
    result[i] = f(a[i]);
  }
  return result;
}
```

In addition to defining functions as described here, you can also use the Function constructor to create functions from a string at runtime, much like **eval()**.

A **method** is a function that is a property of an object.

Calling functions

Defining a function does not execute it. Defining it names the function and specifies what to do when the function is called.

Calling the function actually performs the specified actions with the indicated parameters. For example, if you define the function square, you could call it as follows:

```
square(5);
```

The preceding statement calls the function with an argument of 5. The function executes its statements and returns the value 25.

Functions must be in scope when they are called, but the function declaration can be hoisted (appear below the call in the code). The scope of a function declaration is the function in which it is declared (or the entire program, if it is declared at the top level).

The arguments of a function are not limited to strings and numbers. You can pass whole objects to a function.

A function can call itself. For example, here is a function that computes **factorials recursively**:

```
function factorial(n) {
  if (n === 0 || n === 1) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

You could then compute the factorials of 1 through 5 as follows:

```
const a = factorial(1); // a gets the value 1
const b = factorial(2); // b gets the value 2
const c = factorial(3); // c gets the value 6
const d = factorial(4); // d gets the value 24
const e = factorial(5); // e gets the value 120
```

There are other ways to call functions. There are often cases where a function needs to be called dynamically, or the number of arguments to a function varies.

Function hoisting

Consider the example below:

```
console.log(square(5)); // 25

function square(n) {
  return n * n;
}
```

This code runs without any error, despite the square() function being called before it's declared. This is because the JavaScript interpreter hoists the entire function declaration to the top of the current scope, so the code above is equivalent to:

```
// All function declarations are effectively at the top of the scope
function square(n) {
  return n * n;
}

console.log(square(5)); // 25
```

Function hoisting only works with function declarations — not with function expressions. The code below will not work.

```
  console.log(square); // ReferenceError: Cannot access 'square' before
initialization
  const square = function (n) {
    return n * n;
  }
```

Function scope

Variables defined inside a function cannot be accessed from anywhere outside the function, because the variable is defined only in the scope of the function. However, a function can access all variables and functions defined inside the scope in which it is defined.

In other words, a function defined in the global scope can access all variables defined in the global scope. A function defined inside another function can also access all variables defined in its parent function.

Scope and the function stack

Recursion

A function can refer to and call itself. There are three ways for a function to refer to itself:

1. The function's name
2. `arguments.callee`
3. An in-scope variable that refers to the function

For example, consider the following function definition:

```
const foo = function bar() {
  // statements go here
}
```

Within the function body, the following are all equivalent:

1. `bar()`
2. `arguments.callee()`
3. `foo()`

A function that calls itself is called a recursive function. In some ways, recursion is <u>analogous to a loop</u>. <u>Both execute the same code multiple times, and both require a condition (to avoid an infinite loop, or rather, infinite recursion in this case)</u>.

Some algorithms cannot be simple iterative loops. For example, getting all the nodes of a tree structure (such as the DOM) is easier via recursion:

```
function walkTree(node) {
  if (node === null) {
    return;
  }
  // do something with node
  for (let i = 0; i < node.childNodes.length; i++) {
    walkTree(node.childNodes[i]);
  }
}
```

Nested functions and closures

You may nest a function within another function. The nested (inner) function is private to its containing (outer) function.
It also forms a **closure**. A closure is an expression (most commonly, a function) that can have free variables together with an environment that binds those variables (that "closes" the expression).

Since a nested function is a closure, this means that <u>a nested function can "inherit" the arguments and variables of its containing function</u>. In other words, the **inner function contains the scope of the outer function**.

To summarize:
- The inner function can be accessed only from statements in the outer function.
- The inner function forms a closure: the inner function can use the arguments and variables of the outer function, while the outer function cannot use the arguments and variables of the inner function

```
function outside(x) {
  function inside(y) {
    return x + y;
  }
  return inside;
}
const fnInside = outside(3); // Think of it like: give me a function that adds 3
                            to   whatever you give it
const result = fnInside(5); // returns 8
const result1 = outside(3)(5); // returns 8
```

Name conflicts

When two arguments or variables in the scopes of a closure have the same name, there is a *name conflict*. More nested scopes take precedence. So, the innermost scope takes the highest precedence, while the outermost scope takes the lowest. This is the scope chain.

```
function outside() {
  const x = 5;
  function inside(x) {
    return x * 2;
  }
  return inside;
}

outside()(10); // returns 20 instead of 10
```

Closures

Closures are one of the most powerful features of JavaScript. JavaScript allows for the **nesting of functions** and grants **the inner function full access to all the variables and functions defined inside the outer function** (and all other variables and functions that the outer function has access to).

However, the outer function does *not* have access to the variables and functions defined inside the inner function. This provides a sort of encapsulation for the variables of the inner function.

Also, since the inner function has access to the scope of the outer function, the variables and functions defined in the outer function will live longer than the duration of the outer function

execution, if the inner function manages to survive beyond the life of the outer function. A closure is created when the inner function is somehow made available to any scope outside the outer function.

```
  const pet = function (name) {   // The outer function defines a variable called
"name"
    const getName = function () {
      // The inner function has access to the "name" variable of the outer function
      return name;
    }
    return getName; // Return the inner function, thereby exposing it to outer
scopes
  }
  const myPet = pet('Vivie');

  myPet(); // Returns "Vivie"
```

Using the arguments object

The **arguments of a function are maintained in an array-like object**. Within a function, you can address the arguments passed to it as follows:

arguments[i]

where i is the ordinal number of the argument, starting at 0. So, the first argument passed to a function would be arguments[0]. The total number of arguments is indicated by arguments.length.

Using the arguments object, you can call a function with more arguments than it is formally declared to accept. This is often useful if you don't know in advance how many arguments will be passed to the function. You can use arguments.length to determine the number of arguments actually passed to the function, and then access each argument using the arguments object.

For example, consider a function that concatenates several strings.

```
  function myConcat(separator) {
    let result = ''; // initialize list
    // iterate through arguments
```

```
    for (let i = 1; i < arguments.length; i++) {
      result += arguments[i] + separator;
    }
    return result;
  }
```

You can pass any number of arguments to this function, and it concatenates each argument into a string "list":

```
  // returns "red, orange, blue, "
  myConcat(', ', 'red', 'orange', 'blue');

  // returns "elephant; giraffe; lion; cheetah; "
  myConcat('; ', 'elephant', 'giraffe', 'lion', 'cheetah');
```

The `arguments` variable is "array-like", but not an array. It is array-like in that it has a numbered index and a `length` property. However, it does *not* possess all of the array-manipulation methods.

Function parameters

There are two special kinds of parameter syntax: *default parameters* and *rest parameters*.

Default parameters

In JavaScript, parameters of functions default to `undefined`. However, in some situations it might be useful to set a different default value. This is exactly what default parameters do.

In the following example, if no value is provided for `b`, its value would be `undefined` when evaluating `a*b`, and a call to `multiply` would normally have returned `NaN`. However, this is prevented by the second line in this example:

```
  function multiply(a, b) {
    b = typeof b !== 'undefined' ?  b : 1;
    return a * b;
  }
```

```
  multiply(5); // 5
```

With default parameters, a manual check in the function body is no longer necessary. You can put 1 as the default value for b in the function head:

```
function multiply(a, b = 1) {
  return a * b;
}

multiply(5); // 5
```

Rest parameters

The rest parameter syntax allows us to represent an indefinite number of arguments as an array. In the following example, the function multiply uses rest parameters to collect arguments from the second one to the end. The function then multiplies these by the first argument.

```
function multiply(multiplier, ...theArgs) {
  return theArgs.map((x) => multiplier * x);
}

const arr = multiply(2, 1, 2, 3);
console.log(arr); // [2, 4, 6]
```

Arrow functions

An arrow function expression has a shorter syntax compared to function expressions and **does not have its own** `this, arguments, super,` or `new.target`. Arrow functions are always anonymous.

Two factors influenced the introduction of arrow functions: shorter functions and non-binding of `this`.

Shorter functions

In some functional patterns, shorter functions are welcome. Compare:

```
const a = [
  'Hydrogen',
  'Helium',
  'Lithium',
  'Beryllium'
];

const a2 = a.map(function(s) { return s.length; });

console.log(a2); // [8, 6, 7, 9]

const a3 = a.map((s) => s.length);

console.log(a3); // [8, 6, 7, 9]
```

No separate this

Until arrow functions, every new function defined its own `this` value (a new object in the case of a constructor, undefined in strict mode function calls, the base object if the function is called as an "object method", etc.). This proved to be less than ideal with an object-oriented style of programming.

An arrow function does not have its own `this`; the `this` value of the enclosing execution context is used. Thus, in the following code, the `this` within the function that is passed to setInterval has the same value as `this` in the enclosing function:

```
function Person() {
  this.age = 0;

  setInterval(() => {
    this.age++; // `this` properly refers to the person object
  }, 1000);
}
```

```
const p = new Person();
```

Predefined functions

JavaScript has many top-level, built-in functions. Some of them are:

- `eval()` - The `eval()` method evaluates JavaScript code represented as a string.
- `parseFloat()` - The `parseFloat()` function parses a string argument and returns a floating point number.
- `parseInt()` - The parseInt() function parses a string argument and returns an integer of the specified radix (the base in mathematical numeral systems).
- `isNaN()` - The `isNaN()` function determines whether a value is NaN or not.
- `isFinite()` - The global `isFinite()` function determines whether the passed value is a finite number. If needed, the parameter is first converted to a number.

return statement

The `return` statement ends the function execution and specifies a value to be returned to the function caller.

```
function getRectArea(width, height) {
  if (width > 0 && height > 0) {
    return width * height;
  }
  return 0;
}

console.log(getRectArea(3, 4));
// expected output: 12

console.log(getRectArea(-3, 4));
// expected output: 0
```

The expression whose value is to be returned. If omitted, undefined is returned instead.

This lesson was adapted from MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions

Expressions and operators

At a high level, an expression is a valid unit of code that resolves to a value. There are two types of expressions: those that have side effects (such as assigning values) and those that purely evaluate.

The expression `x = 7` is an example of the first type. This expression uses the `=` operator to assign the value seven to the variable `x`. The expression itself evaluates to `7`.

The expression `3 + 4` is an example of the second type. This expression uses the `+` operator to add 3 and 4 together and produces a value, `7`.

The *precedence* of operators determines the order they are applied when evaluating an expression.

For example:

```
const x = 1 + 2 * 3;
const y = 2 * 3 + 1;
```

Despite `*` and `+` coming in different orders, both expressions would result in 7 because `*` has precedence over `+`, so the `*`-joined expression will always be evaluated first. You can override operator precedence by using parentheses ().

JavaScript has both binary and unary operators, and one special ternary operator, the conditional operator.
A **unary** operator requires a single operand, either before or after the operator. Also `delete, typeof, void` are type of unary operators.

```
i++;
++i;
```

A **binary** operator requires two operands, one before the operator and one after the operator:

```
a * b;
2 + 4;
```

While a **ternary** operator is a **conditional** operator that requires three operands.

```
var numtype = (num%2==0) ? "even": "odd";
```

Assignment operators

An assignment operator assigns a value to its left operand based on the value of its right operand. The simple assignment operator is equal (=), which assigns the value of its right operand to its left operand.

There are also compound assignment operators that are shorthand for the operations listed in the following table:

Name	Shorthand operator	Meaning
Assignment	x = f()	x = f()
Addition assignment	x += f()	x = x + f()
Subtraction assignment	x -= f()	x = x - f()
Multiplication assignment	x *= f()	x = x * f()
Division assignment	x /= f()	x = x / f()
Remainder assignment	x %= f()	x = x % f()
Exponentiation assignment	x **= f()	x = x ** f()
Left shift assignment	x <<= f()	x = x << f()
Right shift assignment	x >>= f()	x = x >> f()
Unsigned right shift assignment	x >>>= f()	x = x >>> f()
Bitwise AND assignment	x &= f()	x = x & f()
Bitwise XOR assignment	x ^= f()	x = x ^ f()
Bitwise OR assignment	x |= f()	x = x	f()
Logical AND assignment	x &&= f()	x && (x = f())
Logical OR assignment	x ||= f()	x || (x = f())
Nullish coalescing assignment	x ??= f()	x ?? (x = f())

Assigning to properties

If an expression evaluates to an object, then the left-hand side of an assignment expression may make assignments to properties of that expression. For example:

```
const obj = {};

obj.x = 3;
console.log(obj.x); // Prints 3.
console.log(obj); // Prints { x: 3 }.

const key = "y";
obj[key] = 5;
console.log(obj[key]); // Prints 5.
console.log(obj); // Prints { x: 3, y: 5 }
```

If an expression does not evaluate to an object, then assignments to properties of that expression do not assign:

```
const val = 0;
val.x = 3;

console.log(val.x); // Prints undefined.
console.log(val); // Prints 0.
```

Destructuring

Destructuring assignment syntax is a JavaScript expression that makes it possible to extract data from arrays or objects using a syntax that mirrors the construction of array and object literals.

```
const foo = ["one", "two", "three"];

// without destructuring
const one = foo[0];
const two = foo[1];
const three = foo[2];

// with destructuring
const [one, two, three] = foo;
```

Evaluation and nesting

In general, assignments are used within a variable declaration (i.e., with `const`, `let`, or `var`) or as standalone statements).

```
// Declares a variable x and initializes it to the result of f().
// The result of the x = f() assignment expression is discarded.
let x = f();

x = g(); // Reassigns the variable x to the result of g().
```

However, like other expressions, assignment expressions like `x = f()` evaluate into a result value. Although this result value is usually not used, it can then be used by another expression.

By chaining or nesting an assignment expression, its result can itself be assigned to another variable. It can be logged, it can be put inside an array literal or function call, and so on.

```
let x;
const y = (x = f()); // Or equivalently: const y = x = f();
console.log(y); // Logs the return value of the assignment x = f().

console.log(x = f()); // Logs the return value directly
```

In the case of logical assignments, `x &&= f()`, `x ||= f()`, and `x ??= f()`, the return value is that of the logical operation without the assignment, so `x && f()`, `x || f()`, and `x ?? f()`, respectively.

When chaining these expressions without parentheses or other grouping operators like array literals, the assignment expressions are grouped right to left (they are right-associative), but they are evaluated left to right.

Avoid assignment chains

Putting a variable chain in a `const`, `let`, or `var` statement often does not work. Only the outermost/leftmost variable would get declared; other variables within the assignment chain are not declared by the `const/let/var` statement. For example:

```
const z = y = x = f();
```

This statement seemingly declares the variables x, y, and z. However, it only actually declares the variable z. y and x are either invalid references to nonexistent variables (in strict mode) or, worse, would implicitly create global variables for x and y in sloppy mode.

Comparison operators

A comparison operator compares its operands and returns a logical value based on whether the comparison is true. The operands can be numerical, string, logical, or object values. Strings are compared based on standard lexicographical ordering, using Unicode values. In most cases, if the two operands are not of the same type, JavaScript attempts to convert them to an appropriate type for the comparison. This behavior generally results in comparing the operands numerically. The sole exceptions to type conversion within comparisons involve the === and !== operators, which perform strict equality and inequality comparisons.

The following table describes the comparison operators in terms of this sample code:

```
const var1 = 3;
const var2 = 4;
```

Operator	Description	Examples returning true

Equal (==)	Returns true if the operands are equal.	`3 == var1`
		`"3" == var1`
		`3 == '3'`

Not equal (!=)	Returns true if the operands are not equal.	`var1 != 4` `var2 != "3"`
Strict equal (===)	Returns true if the operands are equal and of the same type.	`3 === var1`
Strict not equal (!==)	Returns true if the operands are of the same type but not equal, or are of a different type.	`var1 !== "3"` `3 !== '3'`
Greater than (>)	Returns true if the left operand is greater than the right operand.	`var2 > var1` `"12" > 2`
Greater than or equal (>=)	Returns true if the left operand is greater than or equal to the right operand.	`var2 >= var1` `var1 >= 3`

Less than (<)	Returns `true` if the left operand is less than the right operand.	`var1 < var2` `"2" < 12`
Less than or equal (<=)	Returns `true` if the left operand is less than or equal to the right operand.	`var1 <= var2` `var2 <= 5`

Note: => is not a comparison operator but rather is the notation for Arrow functions.

Arithmetic operators

An arithmetic operator takes numerical values (either literals or variables) as their operands and returns a single numerical value. The standard arithmetic operators are addition (+), subtraction (-), multiplication (*), and division (/).

```
1 / 2; // 0.5
1 / 2 === 1.0 / 2.0; // this is true
```

In addition to the standard arithmetic operations (+, -, *, /), JavaScript provides the arithmetic operators listed in the following table:

Operator	Description	Example
Remainder (%)	Binary operator. Returns the integer remainder of dividing the two operands.	12 % 5 returns 2.
Increment (++)	Unary operator. Adds one to its operand. If used as a prefix operator (`++x`), returns the value of its operand after adding one; if used as a postfix operator (`x++`), returns the value of its operand before adding one.	If x is 3, then ++x sets x to 4 and returns 4, whereas x++ returns 3 and, only then, sets x to 4.
Decrement (--)	Unary operator. Subtracts one from its operand. The return value is analogous to that for the increment operator.	If x is 3, then `--x` sets x to 2 and returns 2, whereas x`--` returns 3 and, only then, sets x to 2.
Unary negation (-)	Unary operator. Returns the negation of its operand.	If x is 3, then `-x` returns -3.

Unary plus (+)	Unary operator. Attempts to convert the operand to a number, if it is not already.	+"3" returns 3. +true returns 1.
Exponentiation operator (**)	Calculates the base to the exponent power, that is, base^exponent	2 ** 3 returns 8. 10 ** -1 returns 0.1.

Bitwise operators

A bitwise operator treats their operands as a set of 32 bits (zeros and ones), rather than as decimal, hexadecimal, or octal numbers. Bitwise operators perform their operations on such binary representations, but they return standard JavaScript numerical values.

JavaScript has six bitwise operators that allow you to manipulate the binary representation of numbers at the bit level. Here are the bitwise operators in JavaScript:

1. Bitwise AND (&) - Returns a 1 in each bit position for which the corresponding bits of both operands are 1.
2. Bitwise OR (|) - Returns a 1 in each bit position for which the corresponding bits of either or both operands are 1.
3. Bitwise XOR (^) - Returns a 1 in each bit position for which the corresponding bits of either but not both operands are 1.
4. Bitwise NOT (~) - Inverts the bits of its operand. In other words, it changes every 0 to a 1 and every 1 to a 0.
5. Left shift (<<) - Shifts the bits of the left-hand operand to the left by the number of positions specified by the right-hand operand.
6. Right shift (>>) - Shifts the bits of the left-hand operand to the right by the number of positions specified by the right-hand operand.

Here's an example of using bitwise operators in JavaScript:

```
let a = 5; // binary representation: 0101
let b = 3; // binary representation: 0011

console.log(a & b); // 0001 (1 in binary)
console.log(a | b); // 0111 (7 in binary)
console.log(a ^ b); // 0110 (6 in binary)
console.log(~a); // -6 (in 2's complement representation)
console.log(a << 1); // 1010 (10 in binary)
console.log(a >> 1); // 0010 (2 in binary)
```

Logical operators

Logical operators are typically used with Boolean (logical) values; when they are, they return a Boolean value. However, the && and || operators actually return the value of one of the specified operands, so if these operators are used with non-Boolean values, they may return a non-Boolean value. The logical operators are described in the following table.

Operator	Usage	Description
Logical AND (&&)	`expr1 && expr2`	Returns `expr1` if it can be converted to `false`, otherwise, returns `expr2`. Thus, when used with Boolean values, && returns `true` if both operands are true; otherwise, returns `false`.
Logical OR (||)	`expr1 || expr2`	Returns `expr1` if it can be converted to `true`, otherwise, returns `expr2`. Thus, when used with Boolean values, || returns `true` if either operand is true, if both are false, returns `false`.
Logical NOT (!)	`!expr`	Returns `false` if its single operand that can be converted to true, otherwise, returns `true`.

The following code shows examples of the && (logical AND) operator.

```
const a1 = true && true; // t && t returns true
const a2 = true && false; // t && f returns false
const a3 = false && true; // f && t returns false
const a4 = false && 3 === 4; // f && f returns false
const a5 = "Cat" && "Dog"; // t && t returns Dog
const a6 = false && "Cat"; // f && t returns false
const a7 = "Cat" && false; // t && f returns false
```

The following code shows examples of the || (logical OR) operator.

```
const o1 = true || true; // t || t returns true
const o2 = false || true; // f || t returns true
const o3 = true || false; // t || f returns true
const o4 = false || 3 === 4; // f || f returns false
const o5 = "Cat" || "Dog"; // t || t returns Cat
const o6 = false || "Cat"; // f || t returns Cat
const o7 = "Cat" || false; // t || f returns Cat
```

The following code shows examples of the ! (logical NOT) operator.

```
const n1 = !true; // !t returns false
const n2 = !false; // !f returns true
const n3 = !"Cat"; // !t returns false
```

Short-circuit evaluation

As logical expressions are evaluated left to right, they are tested for possible "short-circuit" evaluation using the following rules:

- `false && anything` is short-circuit evaluated to false.
- `true || anything` is short-circuit evaluated to true.

BigInt operators

Most operators that can be used between numbers can be used between BigInt values as well.

```
// BigInt addition
const a = 1n + 2n; // 3n
// Division with BigInts round towards zero
const b = 1n / 2n; // 0n
// Bitwise operations with BigInts do not truncate either side
const c = 40000000000000000n >> 2n; // 10000000000000000n
```

One exception is unsigned right shift (>>>), which is not defined for BigInt values. This is because a BigInt does not have a fixed width, so technically it does not have a "highest bit".

BigInts and numbers are not mutually replaceable — you cannot mix them in calculations.

```
const a = 1n + 2; // TypeError: Cannot mix BigInt and other types
```

This is because BigInt is neither a subset nor a superset of numbers. BigInts have higher precision than numbers when representing large integers, but cannot represent decimals, so implicit conversion on either side might lose precision. Use explicit conversion to signal whether you wish the operation to be a number operation or a BigInt one.

```
const a = Number(1n) + 2; // 3
const b = 1n + BigInt(2); // 3n
```

You can compare BigInts with numbers.

```
const a = 1n > 2; // false
const b = 3 > 2n; // true
```

String operators

In addition to the **comparison** operators, which can be used on string values, the **concatenation** operator (+) concatenates two string values together, returning another string that is the union of the two operand strings.

For example,

```
console.log("my " + "string"); // console logs the string "my string".
```

The shorthand assignment operator += can also be used to concatenate strings.

For example,

```
let mystring = "alpha";
mystring += "bet"; // evaluates to "alphabet" and assigns this value to mystring
```

Conditional (ternary) operator

The conditional operator is the only JavaScript operator that takes three operands. The operator can have one of two values based on a condition. The syntax is:

```
condition ? val1 : val2
```

If condition is `true`, the operator has the value of `val1`. Otherwise it has the value of `val2`. You can use the conditional operator anywhere you would use a standard operator. For example:

```
const status = age >= 18 ? "adult" : "minor";
```

This statement assigns the value `"adult"` to the variable status if `age` is eighteen or more. Otherwise, it assigns the value `"minor"` to status.

Comma operator

The comma operator (,) evaluates both of its operands and returns the value of the last operand. This operator is primarily used inside a for loop, to allow multiple variables to be updated each time through the loop. It is regarded bad style to use it elsewhere, when it is not necessary.

For example, if a is a 2-dimensional array with 10 elements on a side, the following code uses the comma operator to update two variables at once. The code prints the values of the diagonal elements in the array:

```
const x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
const a = [x, x, x, x, x];

for (let i = 0, j = 9; i <= j; i++, j--) {
  //                             ^
  console.log(`a[${i}][${j}]= ${a[i][j]}`);
}
```

Unary operators

A unary operation is an operation with only one operand.

delete

The `delete` operator deletes an object's property. The syntax is:

```
delete object.property;
delete object[propertyKey];
delete objectName[index];
```

If the `delete` operator succeeds, it removes the property from the object. Trying to access it afterwards will yield `undefined`. The `delete` operator returns true if the operation is possible; it returns `false` if the operation is not possible.

```
delete Math.PI; // returns false (cannot delete non-configurable properties)
```

Deleting array elements

Since arrays are just objects, it's technically possible to delete elements from them. This is however regarded as a bad practice, try to avoid it. When you delete an array property, the array **length is not affected** and other elements are not re-indexed. To achieve that behavior, it is much better to just overwrite the element with the value undefined. To actually manipulate the array, use the various array methods such as splice.

typeof

In JavaScript, the `typeof` operator is used to determine the type of a given value or expression. It returns a string that specifies the data type of the operand.

The syntax for `typeof` is as follows:

```
typeof operand
```

Here, the `operand` can be any valid JavaScript expression.

The possible return values for `typeof` are:

- "`undefined`" - if the operand is an undefined value
- "`boolean`" - if the operand is a boolean value
- "`number`" - if the operand is a number
- "`string`" - if the operand is a string
- "`bigint`" - if the operand is a BigInt
- "`symbol`" - if the operand is a symbol

- "function" - if the operand is a function
- "object" - if the operand is an object or null

It's worth noting that typeof null returns "object", which is a quirk of JavaScript that has been present since the early days of the language and cannot be changed for compatibility reasons.

void

The void operator specifies an expression to be evaluated without returning a value. expression is a JavaScript expression to evaluate. The parentheses surrounding the expression are optional, but it is good style to use them to avoid precedence issues.

Relational operators

A relational operator compares its operands and returns a Boolean value based on whether the comparison is true.

in

The in operator returns true if the specified property is in the specified object. The syntax is:

```
propNameOrNumber in objectName
```

where propNameOrNumber is a string, numeric, or symbol expression representing a property name or array index, and objectName is the name of an object.

The following examples show some uses of the in operator.

```
// Arrays
const trees = ["redwood", "bay", "cedar", "oak", "maple"];
0 in trees; // returns true
3 in trees; // returns true
6 in trees; // returns false
"bay" in trees; // returns false
// (you must specify the index number, not the value at that index)
"length" in trees; // returns true (length is an Array property)

// built-in objects
"PI" in Math; // returns true
const myString = new String("coral");
"length" in myString; // returns true
```

```
  // Custom objects
  const mycar = { make: "Honda", model: "Accord", year: 1998 };
  "make" in mycar; // returns true
"model" in mycar; // returns true
```

instanceof

The `instanceof` operator returns `true` if the specified object is of the specified object type. The syntax is:

```
objectName instanceof objectType
```

where `objectName` is the name of the object to compare to `objectType`, and `objectType` is an object type, such as `Date` or `Array`. Use `instanceof` when you need to confirm the type of an object at runtime. For example, the following code uses `instanceof` to determine whether `theDay` is a `Date` object. Because `theDay` is a `Date` object, the statements in the if statement execute.

```
const theDay = new Date(1995, 12, 17);
if (theDay instanceof Date) {
  // statements to execute
}
```

Basic expressions

All operators eventually operate on one or more basic expressions. These basic expressions include identifiers and literals, but there are a few other kinds as well. They are briefly introduced below, and their semantics are described in detail in their respective reference sections.

this

`this` is a keyword in JavaScript that refers to the current object that a method or function is a property of. The value of `this` depends on how the method or function is called.

Here are a few common ways that `this` is used in JavaScript:

1. Global context In the global context, `this` refers to the global object, which is `window` in a web browser or `global` in Node.js.
2. Function context When `this` is used inside a function, it refers to the object that the function is a method of. For example:

```
   const obj = { name: "John", sayName: function() { console.log(this.name); } };
obj.sayName(); // logs "John"
```

In this example, `this` inside the `sayName` function refers to the `obj` object.

3. Event handlers In event handlers, `this` refers to the DOM element that the event is attached to. For example:

```
   const button = document.querySelector('button'); button.addEventListener('click',
function() { console.log(this); // logs the button element });
```

4. Constructor functions In constructor functions, `this` refers to the new object that is created when the constructor is called. For example:

```
   function Person(name, age) { this.name = name; this.age = age; } const john = new
Person("John", 30); console.log(john.name); // logs "John"
```

In this example, `this` inside the `Person` function refers to the new object that is created when `new Person()` is called.

These are just a few examples of how `this` is used in `JavaScript`. The behavior of `this` can be complex and can depend on a number of factors, so it's important to understand its context and usage in different scenarios.

Grouping operator

The grouping operator ( ) controls the precedence of evaluation in expressions. For example, you can override multiplication and division first, then addition and subtraction to evaluate addition first.

```
   const a = 1;
   const b = 2;
   const c = 3;

   // default precedence
   a + b * c      // 7
   // evaluated by default like this
   a + (b * c)    // 7

   // now overriding precedence
```

```
// addition before multiplication
(a + b) * c    // 9

// which is equivalent to
a * c + b * c // 9
```

new

You can use the new operator to create an instance of a user-defined object type or of one of the built-in object types. Use new as follows:

```
const objectName = new objectType(param1, param2, /* …, */ paramN);
```

super

The super keyword is used to call functions on an object's parent. It is useful with classes to call the parent constructor, for example.

```
super(args); // calls the parent constructor.
super.functionOnParent(args);
```

Numbers and dates

Numbers

In JavaScript, numbers are implemented in double-precision 64-bit binary format IEEE 754 (i.e., a number between $\pm2^{-1022}$ and $\pm2^{+1023}$, or about $\pm10^{-308}$ to $\pm10^{+308}$, with a numeric precision of 53 bits). Integer values up to $\pm2^{53} - 1$ can be represented exactly.

In addition to being able to represent floating-point numbers, the number type has three symbolic values: +Infinity, -Infinity, and NaN (not-a-number).

You can use four types of number literals: decimal, binary, octal, and hexadecimal.

Decimal numbers

```
1234567890
42
```

Decimal literals can start with a zero (0) followed by another decimal digit, but if all digits after the leading 0 are smaller than 8, the number is interpreted as an octal number. This is considered a legacy syntax, and number literals prefixed with 0, whether interpreted as octal or decimal, cause a syntax error in strict mode — so, use the 0o prefix instead.

```
0888 // 888 parsed as decimal
0777 // parsed as octal, 511 in decimal
```

Binary numbers

Binary number syntax uses a leading zero followed by a lowercase or uppercase Latin letter "B" (0b or 0B). If the digits after the 0b are not 0 or 1, the following SyntaxError is thrown: "Missing binary digits after 0b".

```
0b10000000000000000000000000000000 // 2147483648
0b01111111100000000000000000000000 // 2139095040
0B00000000011111111111111111111111 // 8388607
```

Octal numbers

The standard syntax for octal numbers is to prefix them with 0o. For example:

```
0O755 // 493
0o644 // 420
```

There's also a legacy syntax for octal numbers — by prefixing the octal number with a zero: 0644 === 420 and "\045" === "%". If the digits after the 0 are outside the range 0 through 7, the number will be interpreted as a decimal number.

```
const n = 0755; // 493
const m = 0644; // 420
```

Strict mode forbids this octal syntax.

Hexadecimal numbers

Hexadecimal number syntax uses a leading zero followed by a lowercase or uppercase Latin letter "X" (0x or 0X). If the digits after 0x are outside the range (0123456789ABCDEF), the following SyntaxError is thrown: "Identifier starts immediately after numeric literal".

```
0xFFFFFFFFFFFFFFFFF // 295147905179352830000
0x123456789ABCDEF   // 81985529216486900
0XA                 // 10
```

Exponentiation

```
0e-5   // 0
0e+5   // 0
```

```
5e1    // 50
175e-2 // 1.75
1e3    // 1000
1e-3   // 0.001
1E3    // 1000
```

Number object

In JavaScript, the `Number object` is a **built-in object** that provides a way to work with numerical values. It provides a set of methods and properties that can be used to perform mathematical operations and manipulate numbers in various ways.

Here are some of the most commonly used methods and properties of the `Number` object:

1. `Number()`: This method is used to convert a value to a number. For example:

```
Number("10"); // returns 10
Number("10.5"); // returns 10.5
```

2. `isNaN()`: This method is used to check whether a value is NaN (Not-a-Number). For example:

```
isNaN("hello"); // returns true
isNaN(10); // returns false
```

3. `parseInt()`: This method is used to convert a string to an integer. For example:

```
parseInt("10"); // returns 10
parseInt("10.5"); // returns 10
```

4. `parseFloat()`: This method is used to convert a string to a floating-point number. For example:

```
parseFloat("10.5"); // returns 10.5
parseFloat("10"); // returns 10
```

5. `toFixed()`: This method is used to round a number to a specified number of decimal places and return the result as a string. For example:

```
let num = 10.5678;
num.toFixed(2); // returns "10.57"
```

6. `toString()`: This method is used to convert a number to a string. For example:

```
let num = 10;
num.toString(); // returns "10"
```

7. `MAX_VALUE` and `MIN_VALUE`: These properties represent the largest and smallest possible values for a number in JavaScript. For example:

```
Number.MAX_VALUE; // returns 1.7976931348623157e+308
Number.MIN_VALUE; // returns 5e-324
```

These are just a few examples of the methods and properties available on the `Number` object in JavaScript. The `Number` object can be a powerful tool for working with numerical data in your JavaScript code.

Math object

The `Math` object in JavaScript is a built-in object that provides a set of *properties* and *methods* for performing mathematical operations. It is not a constructor, and its methods and properties are static, which means they can be accessed directly without creating an instance of the object.

Some of the properties of the `Math` object include:

- `Math.PI`: represents the ratio of the circumference of a circle to its diameter, which is approximately 3.141592653589793.
- `Math.E`: represents Euler's number, which is approximately 2.718281828459045.
- `Math.LN2`: represents the natural logarithm of 2, which is approximately 0.6931471805599453.

The `Math` object also provides a set of methods for performing mathematical operations, such as:

- `Math.abs(x)`: returns the absolute value of a number `x`.
- `Math.floor(x)`: returns the largest integer less than or equal to a number `x`.
- `Math.max(x1, x2, ..., xn)`: returns the largest of the given numbers.
- `Math.pow(x, y)`: returns the result of raising `x` to the power of `y`.
- `Math.random()`: returns a random number between 0 and 1.

Here's an example of using the `Math` object to generate a random number between two values:

```
let min = 1; let max = 10;
let randomNumber = Math.floor(Math.random() * (max - min + 1) + min);
console.log(randomNumber);
```

This code generates a random number between 1 and 10 (inclusive) and logs it to the console. The `Math.random()` method returns a random number between 0 and 1, and the `Math.floor()` method is used to round the result down to the nearest integer. The formula `(max - min + 1) + min` is used to generate a random number between `min` and `max`.

BigInts

`BigInt` is a relatively new addition to the JavaScript language that provides a way to represent integers that are larger than the maximum safe integer value of `Number` type. The `BigInt` data type can represent arbitrarily large integers with precision, limited only by the amount of memory available to the JavaScript engine.

To create a `BigInt`, you add letter "n" to the end of a numeric literal or use `BigInt()` constructor:

```
const bigNumber = 12345678901234567890123456789n;
const anotherBigNumber = BigInt("12345678901234567890123456789");
```

Note that you must use the `BigInt()` constructor when converting a string to a `BigInt`.

`BigInt` values can be used with standard arithmetic operators like +, -, *, /, and %, and can also be used in comparison operations like >, <, ==, and !=. However, `BigInt` values cannot be mixed with `Number` values in the same operation, and trying to do so will result in a `TypeError`:

```
const bigNumber = 12345678901234567890n;
const anotherBigNumber = BigInt("98765432109876543210");
const sum = bigNumber + anotherBigNumber; // OK
const product = bigNumber * 2; // TypeError!
```

To work with `BigInt` values, you may need to use functions and libraries that support them. For example, the `Math` object does not provide functions for working with `BigInt` values. Some popular libraries that support `BigInt` include `big-integer,` `bigint-crypto-utils,` and `jsbn`.

Date object

`Date` object in JavaScript provides a way to work with dates and times. It represents a single moment in time and can be used to perform various operations on dates such as formatting, parsing, and arithmetic. Date object has a large number of methods, but it does not have any properties.

To create a new `Date` object, you can use one of the following syntaxes:

```
const now = new Date(); // current date and time
const dateFromString = new Date("2022-03-11T15:00:00Z"); // date from ISO string
const dateFromTimestamp = new Date(1647067200000); // date from Unix timestamp
const dateFromParts = new Date(2022, 2, 11, 15, 0, 0); // date from individual
components
```

The first example creates a `Date` object representing the current date and time. The second example creates a `Date` object from an ISO date/time string. The third example creates a `Date` object from a Unix timestamp, which represents the number of milliseconds since January 1, 1970, 00:00:00 UTC. The fourth example creates a `Date` object from individual year, month, day, hour, minute, and second components.

The `Date` object provides a variety of methods for working with dates and times, including:

- `getFullYear()`: Returns the year of the date as a four-digit number (e.g. 2022).
- `getMonth()`: Returns the month of the date as a zero-based index (0 for January, 1 for February, etc.).
- `getDate()`: Returns the day of the month as a number (1-31).
- `getDay()`: Returns the day of the week as a zero-based index (0 for Sunday, 1 for Monday, etc.).
- `getHours()`: Returns the hour of the day as a number (0-23).
- `getMinutes()`: Returns the minute of the hour as a number (0-59).
- `getSeconds()`: Returns the second of the minute as a number (0-59).
- `getTime()`: Returns the Unix timestamp of the date in milliseconds.
- `toLocaleString()`: Returns a string representing the date in a human-readable format, according to the user's locale.
- `toISOString()`: Returns an ISO-formatted string representing the date (e.g. "2022-03-11T15:00:00.000Z").

You can also perform arithmetic operations on `Date` objects, such as adding or subtracting days, hours, or minutes:

```
const date = new Date("2022-03-11T15:00:00Z");
date.setDate(date.getDate() + 1); // add one day
date.setHours(date.getHours() - 2); // subtract two hours
```

This code creates a `Date` object representing March 11th, 2022 at 3:00pm UTC. It then adds one day to the date and subtracts two hours, resulting in a new `Date` object representing March 12th, 2022 at 1:00pm UTC.

Text formatting

This chapter introduces how to work with strings and text in JavaScript.

Strings

JavaScript's String type is used to represent textual data. It is a set of "elements" of 16-bit unsigned integer values (UTF-16 code units). Each element in the String occupies a position in the String. The first element is at index 0, the next at index 1, and so on. The length of a String is the number of elements in it. You can create strings using **string literals** or **string objects**.

String literals

You can create simple strings using either single or double quotes:

```
'foo'
"bar"
```

More advanced strings can be created using escape sequences:

Hexadecimal escape sequences

The number after \x is interpreted as a hexadecimal number.

```
"\xA9" // "©"
```

Unicode escape sequences

The Unicode escape sequences require at least four hexadecimal digits following \u.

```
"\u00A9" // "©"
```

Unicode code point escapes

With Unicode code point escapes, any character can be escaped using hexadecimal numbers so it is possible to use Unicode code points up to 0x10FFFF. With simple Unicode escapes it is often necessary to write the surrogate halves separately to achieve the same result.

```
"\u{2F804}"

// the same with simple Unicode escapes
"\uD87E\uDC04"
```

String objects

The String object is a wrapper around the string primitive data type.

```
const foo = new String("foo"); // Creates a String object
console.log(foo); // [String: 'foo']
typeof foo; // 'object'
```

You can call any of the methods of the `String` object on a string literal value — JavaScript automatically converts the string literal to a temporary `String` object, calls the method, then discards the temporary `String` object. You can also use the `length` property with a string literal.

A String object has one property, length, that indicates the number of UTF-16 code units in the string. For example, the following code assigns helloLength the value 13, because "Hello, World!" has 13 characters, each represented by one UTF-16 code unit. You can access each code unit using an array bracket style. You can't change individual characters because strings are immutable array-like objects:

```
const hello = "Hello, World!";
const helloLength = hello.length;
hello[0] = "L"; // This has no effect, because strings are immutable
hello[0]; // This returns "H"
```

Characters whose Unicode scalar values are greater than U+FFFF (such as some rare Chinese/Japanese/Korean/Vietnamese characters and some emoji) are stored in UTF-16 with two surrogate code units each.

Methods of String

Method	Description
`charAt()`, `charCodeAt()`, `codePointAt()`	Return the character or character code at the specified position in string.
`indexOf()`, `lastIndexOf()`	Return the position of specified substring in the string or last position of specified substring, respectively.
`startsWith()`, `endsWith()`, `includes()`	Returns whether or not the string starts, ends or contains a specified string.
`concat()`	Combines the text of two strings and returns a new string.
`split()`	Splits a String object into an array of strings by separating the string into substrings.
`slice()`	Extracts a section of a string and returns a new string.
`substring()`, `substr()`	Return the specified subset of the string, either by specifying the start and end indexes or the start index and a length.
`match()`, `matchAll()`, `replace()`, `replaceAll()`, `search()`	Work with regular expressions.
`toLowerCase()`, `toUpperCase()`	Return the string in all lowercase or all uppercase, respectively.
normalize()	Returns the Unicode Normalization Form of the calling string value.
repeat()	Returns a string consisting of the elements of the object repeated the given times.
trim()	Trims whitespace from the beginning and end of the string.

Multi-line template literals

Template literals are string literals allowing embedded expressions. You can use multi-line strings and string interpolation features with them.

Template literals are enclosed by backtick (grave accent) characters (`) instead of double or single quotes. Template literals can contain placeholders. These are indicated by the dollar sign and curly braces (${expression}).

Multi-lines

Any new line characters inserted in the source are part of the template literal. Using normal strings, you would have to use the following syntax in order to get multi-line strings:

```
console.log(
  "string text line 1\n\
string text line 2",
);
// "string text line 1
// string text line 2"
```

To get the same effect with multi-line strings, you can now write:

```
console.log(`string text line 1
string text line 2`);
// "string text line 1
// string text line 2"
```

Embedded expressions

In order to embed expressions within normal strings, you would use the following syntax:

```
const five = 5;
const ten = 10;
console.log(
  "Fifteen is " + (five + ten) + " and not " + (2 * five + ten) + ".",
);
```

```
// "Fifteen is 15 and not 20."
```

Now, with template literals, you are able to make use of the syntactic sugar making substitutions like this more readable:

```
const five = 5;
const ten = 10;
console.log(`Fifteen is ${five + ten} and not ${2 * five + ten}.`);
// "Fifteen is 15 and not 20."
```

Internationalization

The `Intl` object is the namespace for the ECMAScript Internationalization API, which provides language sensitive string comparison, number formatting, and date and time formatting. The constructors for `Intl.Collator, Intl.NumberFormat,` and `Intl.DateTimeFormat` objects are properties of the `Intl` object.

Date and time formatting

The `Intl.DateTimeFormat` object is useful for formatting date and time. The following formats a date for English as used in the United States. (The result is different in another time zone.)

```
// July 17, 2014 00:00:00 UTC:
const july172014 = new Date("2014-07-17");

const options = {
  year: "2-digit",
  month: "2-digit",
  day: "2-digit",
  hour: "2-digit",
  minute: "2-digit",
  timeZoneName: "short",
};
const americanDateTime = new Intl.DateTimeFormat("en-US", options).format;

// Local timezone vary depending on your settings
// In CEST, logs: 07/17/14, 02:00 AM GMT+2
// In PDT, logs: 07/16/14, 05:00 PM GMT-7
console.log(americanDateTime(july172014));
```

Number formatting

The `Intl.NumberFormat` object is useful for formatting numbers, for example currencies.

```
const gasPrice = new Intl.NumberFormat("en-US", {
  style: "currency",
  currency: "USD",
  minimumFractionDigits: 3,
});

console.log(gasPrice.format(5.259)); // $5.259

const hanDecimalRMBInChina = new Intl.NumberFormat("zh-CN-u-nu-hanidec", {
  style: "currency",
  currency: "CNY",
});

console.log(hanDecimalRMBInChina.format(1314.25)); // ￥ 一,三一四.二五
```

Collation

The `Intl.Collator` object is useful for comparing and sorting strings.
For example, there are actually two different sort of orders in German, phonebook and a dictionary. Phonebook sort emphasizes sound, and it's as if "ä", "ö", and so on were expanded to "ae", "oe", and so on prior to sorting.

```
const names = ["Hochberg", "Hönigswald", "Holzman"];

const germanPhonebook = new Intl.Collator("de-DE-u-co-phonebk");

// as if sorting ["Hochberg", "Hoenigswald", "Holzman"]:
console.log(names.sort(germanPhonebook.compare).join(", "));
// "Hochberg, Hönigswald, Holzman"
```

Some German words conjugate with extra umlauts, so in dictionaries, it's sensible to order ignoring umlauts (except when ordering words differing only by umlauts: schon before schön).

```
const germanDictionary = new Intl.Collator("de-DE-u-co-dict");

// as if sorting ["Hochberg", "Honigswald", "Holzman"]:
console.log(names.sort(germanDictionary.compare).join(", "));
// "Hochberg, Holzman, Hönigswald"
```

Regular expressions

Regular expressions are patterns used to match character combinations in strings. In JavaScript, regular expressions are also objects. These patterns are used with the `exec()` and `test()` methods of `RegExp`, and with the `match()`, `matchAll()`, `replace()`, `replaceAll()`, `search()`, and `split()` methods of `String`.

Creating a regular expressions

You construct a regular expression in one of two ways:

- Using a regular expression literal, which consists of a pattern enclosed between slashes, as follows:

```
const re = /ab+c/;
```

- Or calling the constructor function of the RegExp object, as follows:

```
const re = new RegExp("ab+c");
```

Using the constructor function provides runtime compilation of the regular expression. Use the constructor function when you know the regular expression pattern will be changing, or you don't know the pattern and are getting it from another source, such as user input.

Writing a regular expression pattern

A regular expression pattern is composed of simple characters, such as `/abc/`, or a combination of simple and special characters, such as `/ab*c/` or `/Chapter (\d+)\.\d*/`. The last example includes parentheses, which are used as a memory device. The match made with this part of the pattern is remembered for later use, as described in Using groups.

Using simple patterns

Simple patterns are constructed of characters for which you want to find a direct match. For example, the pattern `/abc/` matches character combinations in strings only when the exact sequence "abc" occurs (all characters together and in that order). Such a match would succeed in the strings "`Hi, do you know your abc's?`" and "`The latest airplane designs evolved from slabcraft.`". In both cases the match is with the substring "abc". There is no match in the string "`Grab crab`" because while it contains the substring "`ab c`", it does not contain the exact substring "abc".

Using special characters

When the search for a match requires something more than a direct match, such as finding one or more b's, or finding white space, you can include special characters in the pattern. For example, to match a single "a" followed by zero or more "b"s followed by "c", you'd use the pattern `/ab*c/`: the `*` after "b" means "0 or more occurrences of the preceding item." In the string "`cbbabbbbcdebc`", this pattern will match the substring "`abbbbc`".

The following pages provide lists of the different special characters that fit into each category, along with descriptions and examples.

Assertions

Assertions include boundaries, which indicate the beginnings and endings of lines and words, and other patterns indicating in some way that a match is possible (including look-ahead, look-behind, and conditional expressions).

Character classes

Distinguish different types of characters. For example, distinguishing between letters and digits.

Groups and backreferences

Groups group multiple patterns as a whole, and capturing groups provide extra submatch information when using a regular expression pattern to match against a string. Backreferences refer to a previously captured group in the same regular expression.

Quantifiers

Indicate the numbers of characters or expressions to match.

Unicode property escapes

Distinguish based on unicode character properties, for example, upper- and lower-case letters, math symbols, and punctuation.

If you want to look at all the special characters that can be used in regular expressions in a single table, see the following:

Characters / constructs	Corresponding article
`[xyz], [^xyz], ., \d, \D, \w, \W, \s, \S, \t, \r, \n, \v, \f, [\b], \0, \cX, \xhh, \uhhhh, \u{hhhh}, x|y`	Character classes
`^, $, \b, \B, x(?=y), x(?!y), (?<=y)x, (?<!y)x`	Assertions
`(x), (?<Name>x), (?:x), \n, \k<Name>`	Groups and backreferences
`x*, x+, x?, x{n}, x{n,}, x{n,m}`	Quantifiers
`\p{UnicodeProperty}, \P{UnicodeProperty}`	Unicode property escapes

Escaping

If you need to use any of the special characters literally (actually searching for a "*", for instance), you must escape it by putting a backslash in front of it. For instance, to search for "a" followed by "*" followed by "b", you'd use `/a\*b/` — the backslash "escapes" the "*", making it literal instead of special.

Similarly, if you're writing a regular expression literal and need to match a slash ("/"), you need to escape that (otherwise, it terminates the pattern). For instance, to search for the string "/example/" followed by one or more alphabetic characters, you'd use `/\/example\/[a-z]+/i`—the backslashes before each slash make them literal.

To match a literal backslash, you need to escape the backslash. For instance, to match the string "C:\" where "C" can be any letter, you'd use `/[A-Z]:\\/` — the first backslash escapes the one after it, so the expression searches for a single literal backslash.

If using the RegExp constructor with a string literal, remember that the backslash is an escape in string literals, so to use it in the regular expression, you need to escape it at the string literal level. `/a\*b/` and new `RegExp("a\\*b")` create the same expression, which searches for "a" followed by a literal "*" followed by "b".

If escape strings are not already part of your pattern you can add them using `String.prototype.replace()`:

```
function escapeRegExp(string) {
  return string.replace(/[.*+?^${}()|[\]\\]/g, "\\$&"); // $& means the whole
                                                                   matched string
}
```

Using regular expressions in JavaScript

Regular expressions are used with the `RegExp` methods `test()` and `exec()` and with the String methods `match()`, `replace()`, `search()`, and `split()`.

Method	Description
exec()	Executes a search for a match in a string. It returns an array of information or `null` on a mismatch.
test()	Tests for a match in a string. It returns `true` or `false`.
match()	Returns an array containing all of the matches, including capturing groups, or `null` if no match is found.
matchAll()	Returns an iterator containing all of the matches, including capturing groups.
search()	Tests for a match in a string. It returns the index of the match, or -1 if the search fails.
replace()	Executes a search for a match in a string, and replaces the matched substring with a replacement substring.

`replaceAll()`	Executes a search for all matches in a string, and replaces the matched substrings with a replacement substring.
`split()`	Uses a regular expression or a fixed string to break a string into an array of substrings.

In the following example, the script uses the `exec()` method to find a match in a string.

```
const myRe = /d(b+)d/g;
const myArray = myRe.exec("cdbbdbsbz");
```

Advanced searching with flags

Regular expressions have optional flags that allow for functionality like global searching and case-insensitive searching. These flags can be used separately or together in any order, and are included as part of the regular expression.

Flag	Description	Corresponding property
d	Generate indices for substring matches.	`hasIndices`
g	Global search.	`global`
i	Case-insensitive search.	`ignoreCase`
m	Allows ^ and $ to match newline characters.	`multiline`
s	Allows . to match newline characters.	`dotAll`
u	"Unicode"; treat a pattern as a sequence of Unicode code points.	`unicode`
y	Perform a "sticky" search that matches starting at the current position in the target string.	`sticky`

To include a flag with the regular expression, use this syntax:

```
const re = /pattern/flags;
```

or

```
const re = new RegExp("pattern", "flags");
```

Note that the flags are an integral part of a regular expression. They cannot be added or removed later.

For example, `re = /\w+\s/g` creates a regular expression that looks for one or more characters followed by a space, and it looks for this combination throughout the string.

```
const re = /\w+\s/g;
const str = "fee fi fo fum";
const myArray = str.match(re);
console.log(myArray);

// ["fee ", "fi ", "fo "]
```

You could replace the line:

```
const re = /\w+\s/g;
```

with:

```
const re = new RegExp("\\w+\\s", "g");
```

and get the same result.

The `m` flag is used to specify that a multiline input string should be treated as multiple lines. If the `m` flag is used, `^` and `$` match at the start or end of any line within the input string instead of the start or end of the entire string.

Using the global search flag with exec()

`RegExp.prototype.exec()` method with the g flag returns each match and its position iteratively.

```
const str = "fee fi fo fum";
const re = /\w+\s/g;
console.log(re.exec(str)); // ["fee ", index: 0, input: "fee fi fo fum"]
console.log(re.exec(str)); // ["fi ", index: 4, input: "fee fi fo fum"]
console.log(re.exec(str)); // ["fo ", index: 7, input: "fee fi fo fum"]
console.log(re.exec(str)); // null
```

In contrast, `String.prototype.match()` method returns all matches at once, but without their position.

```
console.log(str.match(re)); // ["fee ", "fi ", "fo "]
```

Using unicode regular expressions

The "u" flag is used to create "unicode" regular expressions; that is, regular expressions which support matching against unicode text. This is mainly accomplished through the use of Unicode property escapes, which are supported only within "unicode" regular expressions.

For example, following regular expression can be used to match against arbitrary unicode "word":

```
/\p{L}*/u;
```

There are a number of other differences between unicode and non-unicode regular expressions that one should be aware of:

- Unicode regular expressions do not support so-called "identity escapes"; For example, /\a/ is a valid regular expression matching the letter 'a', but /\a/u is not.
- Curly brackets need to be escaped when not used as quantifiers. For example, /{/ is a valid regular expression matching the curly bracket '{', but /{/u is not — instead, the bracket should be escaped and /\{/u should be used instead.
- The - character is interpreted differently within character classes. For example, /[\w-:]/ is a valid regular expression matching a word character, a -, or :, but /[\w-:]/u is an invalid regular expression, as \w to : is not a well-defined range of characters.

This lesson was adapted from MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_expressions

Indexed collections

This chapter introduces collections of data that are ordered by an index value. This includes **arrays** and **array-like constructs** such as `Array` objects and `TypedArray` objects.

For example, consider an array called `emp`, which contains employees' names indexed by their numerical employee number. So `emp[0]` would be employee number zero, `emp[1]` employee number one, and so on.

JavaScript does not have an explicit array data type. However, you can use the predefined `Array` object and its methods to work with arrays in your applications. The `Array` object has methods for manipulating arrays, such as joining, reversing, and sorting them. It has a property for determining the array length and other properties for use with regular expressions.

We will focus on arrays in this article, but many of the same concepts apply to typed arrays as well, since arrays and typed arrays share many similar methods.

Creating an array

The following statements create equivalent arrays:

```
const arr1 = new Array(element0, element1, /* … ,*/ elementN);
const arr2 = Array(element0, element1, /* … ,*/ elementN);
const arr3 = [element0, element1, /* … ,*/ elementN];
```

`element0, element1, …, elementN` is a list of values for the array's elements. When these values are specified, the array is initialized with them as the array's elements. The array's `length` property is set to the number of arguments.

The bracket syntax is called an "array literal" or "array initializer", and is generally preferred.

To create an array with non-zero length, but without any items, either of the following can be used:

```
// This...
const arr1 = new Array(arrayLength);
```

```
// ...results in the same array as this
const arr2 = Array(arrayLength);

// This has exactly the same effect
const arr3 = [];
arr3.length = arrayLength;
```

Note: In the above code, `arrayLength` must be a `Number`. Otherwise, an array with a single element (the provided value) will be created. Calling `arr.length` will return `arrayLength`, but the array doesn't contain any elements. A `for...in` loop will not find any property on the array.

In addition to a newly defined variable as shown above, arrays can also be assigned as a property of a new or an existing object:

```
const obj = {};
// …
obj.prop = [element0, element1, /* … ,*/ elementN];

// OR
const obj = { prop: [element0, element1, /* … ,*/ elementN] };
```

If you wish to initialize an array with a single element, and the element happens to be a `Number`, you must use the bracket syntax. When a single `Number` value is passed to the `Array()` constructor or function, it is interpreted as an `arrayLength`, not as a single element.

```
// This creates an array with only one element: the number 42.
const arr = [42];

// This creates an array with no elements and arr.length set to 42.
const arr = Array(42);
```

Calling `Array(N)` results in a `RangeError`, if N is a non-whole number whose fractional portion is non-zero. The following example illustrates this behavior.

```
const arr = Array(9.3); // RangeError: Invalid array length
```

If your code needs to create arrays with single elements of an arbitrary data type, it is safer to use array literals. Alternatively, create an empty array first before adding the single element to it.

You can also use the `Array.of` static method to create arrays with single element.

```
const wisenArray = Array.of(9.3); // wisenArray contains only one element 9.3
```

Referring to array elements

Because elements are also properties, you can access them using property accessors. Suppose you define the following array:

```
const myArray = ["Wind", "Rain", "Fire"];
```

You can refer to the first element of the array as `myArray[0]`, the second element of the array as `myArray[1]`, etc... The index of the elements begins with zero.

Note: You can also use property accessors to access other properties of the array, like with an object.

```
const arr = ["one", "two", "three"];
arr[2]; // three
arr["length"]; // 3
```

Populating an array

You can populate an array by assigning values to its elements. For example:

```
const emp = [];
emp[0] = "Casey Jones";
emp[1] = "Phil Lesh";
emp[2] = "August West";
```

If you supply a non-integer value to the array operator in the code above, a property will be created in the object representing the array, instead of an array element.

```
const arr = [];
arr[3.4] = "Oranges";
console.log(arr.length); // 0
console.log(Object.hasOwn(arr, 3.4)); // true
```

You can also populate an array when you create it:

```
const myArray = new Array("Hello", myVar, 3.14159);
// OR
const myArray = ["Mango", "Apple", "Orange"];
```

Understanding length

At the implementation level, JavaScript's arrays actually store their elements as standard object properties, using the array index as the property name.

The `length` property is special. Its value is always a positive integer greater than the index of the last element if one exists. (In the example below, `'Dusty'` is indexed at 30, so `cats.length` returns 30 + 1).

Remember, JavaScript Array indexes are 0-based: they start at 0, not 1. This means that the **length property will be one more than the highest index stored in the array**:

```
const cats = [];
cats[30] = ["Dusty"];
console.log(cats.length); // 31
```

You can also assign to the `length` property.

Writing a value that is shorter than the number of stored items truncates the array. Writing 0 empties it entirely:

```
const cats = ["Dusty", "Misty", "Twiggy"];
console.log(cats.length); // 3

cats.length = 2;
console.log(cats); // [ 'Dusty', 'Misty' ] - Twiggy has been removed
```

```
cats.length = 0;
console.log(cats); // []; the cats array is empty

cats.length = 3;
console.log(cats); // [ <3 empty items> ]
```

Iterating over arrays

A common operation is to iterate over the values of an array, processing each one in some way. The simplest way to do this is as follows:

```
const colors = ["red", "green", "blue"];
for (let i = 0; i < colors.length; i++) {
  console.log(colors[i]);
}
```

If you know that none of the elements in your array evaluate to `false` in a boolean context—if your array consists only of DOM nodes, for example—you can use a more efficient idiom:

```
const divs = document.getElementsByTagName("div");
for (let i = 0, div; (div = divs[i]); i++) {
  /* Process div in some way */
}
```

This avoids the overhead of checking the length of the array, and ensures that the `div` variable is reassigned to the current item each time around the loop for added convenience.

The `forEach()` method provides another way of iterating over an array:

```
const colors = ["red", "green", "blue"];
colors.forEach((color) => console.log(color));
// red
// green
// blue
```

The function passed to `forEach` is executed once for every item in the array, with the array item passed as the argument to the function. Unassigned values are not iterated in a `forEach` loop.

Note that the elements of an array that are omitted when the array is defined are not listed when iterating by `forEach`, but are listed when `undefined` has been manually assigned to the element:

```
const sparseArray = ["first", "second", , "fourth"];

sparseArray.forEach((element) => {
  console.log(element);
});
// Logs:
// first
// second
// fourth

if (sparseArray[2] === undefined) {
  console.log("sparseArray[2] is undefined"); // true
}

const nonsparseArray = ["first", "second", undefined, "fourth"];

nonsparseArray.forEach((element) => {
  console.log(element);
});
// Logs:
// first
// second
// undefined
// fourth
```

Since JavaScript array elements are saved as standard object properties, it is not advisable to iterate through JavaScript arrays using `for...in` loops, because normal elements and all enumerable properties will be listed.

Array methods

The Array object has the following methods:

concat()

The `concat()` method joins two or more arrays and returns a new array.

```
let myArray = ["1", "2", "3"];
myArray = myArray.concat("a", "b", "c");
// myArray is now ["1", "2", "3", "a", "b", "c"]
```

join()

The `join()` method joins all elements of an array into a string.

```
const myArray = ["Wind", "Rain", "Fire"];
const list = myArray.join(" - "); // list is "Wind - Rain - Fire"
```

push()

The `push()` method adds one or more elements to the end of an array and returns the resulting `length` of the array.

```
const myArray = ["1", "2"];
myArray.push("3"); // myArray is now ["1", "2", "3"]
```

pop()

The `pop()` method removes the last element from an array and returns that element.

```
const myArray = ["1", "2", "3"];
const last = myArray.pop();
// myArray is now ["1", "2"], last = "3"
```

shift()

The `shift()` method removes the first element from an array and returns that element.

```
const myArray = ["1", "2", "3"];
const first = myArray.shift();
// myArray is now ["2", "3"], first is "1"
```

unshift()

The `unshift()` method adds one or more elements to the front of an array and returns the new `length` of the array.

```
const myArray = ["1", "2", "3"];
myArray.unshift("4", "5");
// myArray becomes ["4", "5", "1", "2", "3"]
```

slice()

The `slice()` method extracts a section of an array and returns a new array.

```
let myArray = ["a", "b", "c", "d", "e"];
myArray = myArray.slice(1, 4); // [ "b", "c", "d"]
// starts at index 1 and extracts all elements
// until index 3
```

splice()

The `splice()` method removes elements from an array and (optionally) replaces them. It returns the items which were removed from the array.

```
const myArray = ["1", "2", "3", "4", "5"];
myArray.splice(1, 3, "a", "b", "c", "d");
// myArray is now ["1", "a", "b", "c", "d", "5"]
// This code started at index one (or where the "2" was),
// removed 3 elements there, and then inserted all consecutive
// elements in its place.
```

at()

The `at()` method returns the element at the specified index in the array, or `undefined` if the index is out of range. It's notably used for negative indices that access elements from the end of the array.

```
const myArray = ["a", "b", "c", "d", "e"];
myArray.at(-2); // "d", the second-last element of myArray
```

reverse()

The `reverse()` method transposes the elements of an array, in place: the first array element becomes the last and the last becomes the first. It returns a reference to the array.

```
const myArray = ["1", "2", "3"];
myArray.reverse();
// transposes the array so that myArray = ["3", "2", "1"]
```

flat()

The `flat()` method returns a new array with all sub-array elements concatenated into it recursively up to the specified depth.

```
let myArray = [1, 2, [3, 4]];
myArray = myArray.flat();
// myArray is now [1, 2, 3, 4], since the [3, 4] subarray is flattened
```

sort()

The `sort()` method sorts the elements of an array in place, and returns a reference to the array.

```
const myArray = ["Wind", "Rain", "Fire"];
myArray.sort();
// sorts the array so that myArray = ["Fire", "Rain", "Wind"]
```

`sort()` can also take a callback function to determine how array elements are compared. The callback function is called with two arguments, which are two values from the array. The function compares these two values and returns a positive number, negative number, or zero, indicating the order of the two values. For instance, the following will sort the array by the last letter of a string:

```
const sortFn = (a, b) => {
  if (a[a.length - 1] < b[b.length - 1]) {
    return -1; // Negative number => a < b, a comes before b
  } else if (a[a.length - 1] > b[b.length - 1]) {
    return 1; // Positive number => a > b, a comes after b
  }
  return 0; // Zero => a = b, a and b keep their original order
};
myArray.sort(sortFn);
// sorts the array so that myArray = ["Wind","Fire","Rain"]
```

- if a is less than b by the sorting system, return -1 (or any negative number)
- if a is greater than b by the sorting system, return 1 (or any positive number)
- if a and b are considered equivalent, return 0.

indexOf()

The `indexOf()` method searches the array for `searchElement` and returns the index of the first match.

```
const a = ["a", "b", "a", "b", "a"];
console.log(a.indexOf("b")); // 1

// Now try again, starting from after the last match
console.log(a.indexOf("b", 2)); // 3
console.log(a.indexOf("z")); // -1, because 'z' was not found
```

lastIndexOf()

The `lastIndexOf()` method works like `indexOf`, but starts at the end and searches backwards.

```
const a = ["a", "b", "c", "d", "a", "b"];
console.log(a.lastIndexOf("b")); // 5

// Now try again, starting from before the last match
console.log(a.lastIndexOf("b", 4)); // 1
console.log(a.lastIndexOf("z")); // -1
```

forEach()

The `forEach()` method executes `callback` on every array item and returns `undefined`.

```
const a = ["a", "b", "c"];
a.forEach((element) => {
  console.log(element);
});
// Logs:
// a
// b
// c
```

The `forEach` method (and others below) that take a callback are known as *iterative methods*, because they iterate over the entire array in some fashion. Each one takes an optional second argument called `thisArg`. If provided, `thisArg` becomes the value of the `this` keyword inside the body of the callback function. If not provided, as with other cases where a function is invoked outside of an explicit object context, `this` will refer to the global object (`window`, `globalThis`, etc.) when the function is not strict, or `undefined` when the function is strict.

Note: The `sort()` method introduced above is not an iterative method, because its callback function is only used for comparison and may not be called in any particular order based on element order. `sort()` does not accept the `thisArg` parameter either.

map()

The `map()` method returns a new array of the return value from executing `callback` on every array item.

```
const a1 = ["a", "b", "c"];
const a2 = a1.map((item) => item.toUpperCase());
console.log(a2); // ['A', 'B', 'C']
```

flatmap()

The `flatmap()` method runs `map()` followed by a `flat()` of depth 1.

```
const a1 = ["a", "b", "c"];
const a2 = a1.flatMap((item) => [item.toUpperCase(), item.toLowerCase()]);
console.log(a2); // ['A', 'a', 'B', 'b', 'C', 'c']
```

filter()

The `filter()` method returns a new array containing the items for which `callback` returned `true`.

```
const a1 = ["a", 10, "b", 20, "c", 30];
const a2 = a1.filter((item) => typeof item === "number");
console.log(a2); // [10, 20, 30]
```

find()

The `find()` method returns the first item for which `callback` returned `true`.

```
const a1 = ["a", 10, "b", 20, "c", 30];
const i = a1.find((item) => typeof item === "number");
console.log(i); // 10
```

findLast()

The `findLast()` method returns the last item for which `callback` returned `true`.

```
const a1 = ["a", 10, "b", 20, "c", 30];
const i = a1.findLast((item) => typeof item === "number");
console.log(i); // 30
```

findIndex()

The `findIndex()` method returns the index of the first item for which `callback` returned `true`.

```
const a1 = ["a", 10, "b", 20, "c", 30];
const i = a1.findIndex((item) => typeof item === "number");
console.log(i); // 1
```

findLastIndex()

The `findLastIndex()` method returns the index of the last item for which `callback` returned `true`.

```
const a1 = ["a", 10, "b", 20, "c", 30];
const i = a1.findLastIndex((item) => typeof item === "number");
console.log(i); // 5
```

every()

The `every()` method returns `true` if `callback` returns `true` for every item in the array.

```
function isNumber(value) {
  return typeof value === "number";
}
const a1 = [1, 2, 3];
```

```
console.log(a1.every(isNumber)); // true
const a2 = [1, "2", 3];
console.log(a2.every(isNumber)); // false
```

some()

The some() method returns true if callback returns true for at least one item in the array.

```
function isNumber(value) {
  return typeof value === "number";
}
const a1 = [1, 2, 3];
console.log(a1.every(isNumber)); // true
const a2 = [1, "2", 3];
console.log(a2.every(isNumber)); // false
```

reduce()

The reduce() method applies callback(accumulator, currentValue, currentIndex, array) for each value in the array for the purpose of reducing the list of items down to a single value. The reduce function returns the final value returned by callback function.

If initialValue is specified, then callback is called with initialValue as the first parameter value and the value of the first item in the array as the second parameter value.

If initialValue is *not* specified, then callback's first two parameter values will be the first and second elements of the array. On *every* subsequent call, the first parameter's value will be whatever callback returned on the previous call, and the second parameter's value will be the next value in the array.

If callback needs access to the index of the item being processed, or access to the entire array, they are available as optional parameters.

```
const a = [10, 20, 30];
const total = a.reduce(
  (accumulator, currentValue) => accumulator + currentValue,
  0,
);
console.log(total); // 60
```

The `reduceRight()` method works like `reduce()`, but starts with the last element.

`reduce` and `reduceRight` are the least obvious of the iterative array methods. They should be used for algorithms that combine two values recursively in order to reduce a sequence down to a single value.

Sparse arrays

Arrays can contain "empty slots", which are not the same as slots filled with the value `undefined`. Empty slots can be created in one of the following ways:

```
// Array constructor:
const a = Array(5); // [ <5 empty items> ]

// Consecutive commas in array literal:
const b = [1, 2, , , 5]; // [ 1, 2, <2 empty items>, 5 ]

// Directly setting a slot with index greater than array.length:
const c = [1, 2];
c[4] = 5; // [ 1, 2, <2 empty items>, 5 ]

// Elongating an array by directly setting .length:
const d = [1, 2];
d.length = 5; // [ 1, 2, <3 empty items> ]

// Deleting an element:
const e = [1, 2, 3, 4, 5];
delete e[2]; // [ 1, 2, <1 empty item>, 4, 5 ]
```

In some operations, empty slots behave as if they are filled with `undefined`.

```
const arr = [1, 2, , , 5]; // Create a sparse array

// Indexed access
console.log(arr[2]); // undefined

// For...of
for (const i of arr) {
  console.log(i);
}
```

```
// Logs: 1 2 undefined undefined 5

// Spreading
const another = [...arr]; // "another" is [ 1, 2, undefined, undefined, 5 ]
```

But in others (most notably array iteration methods), empty slots are skipped.

```
const mapped = arr.map((i) => i + 1); // [ 2, 3, <2 empty items>, 6 ]
arr.forEach((i) => console.log(i)); // 1 2 5
const filtered = arr.filter(() => true); // [ 1, 2, 5 ]
const hasFalsy = arr.some((k) => !k); // false

// Property enumeration
const keys = Object.keys(arr); // [ '0', '1', '4' ]
for (const key in arr) {
  console.log(key);
}
// Logs: '0' '1' '4'
// Spreading into an object uses property enumeration, not the array's iterator
const objectSpread = { ...arr }; // { '0': 1, '1': 2, '4': 5 }
```

Multi-dimensional arrays

Arrays can be nested, meaning that an array can contain another array as an element. Using this characteristic of JavaScript arrays, multi-dimensional arrays can be created.

The following code creates a two-dimensional array.

```
const a = new Array(4);
for (let i = 0; i < 4; i++) {
  a[i] = new Array(4);
  for (let j = 0; j < 4; j++) {
    a[i][j] = `[${i}, ${j}]`;
  }
}
```

This example creates an array with the following rows:

```
Row 0: [0, 0] [0, 1] [0, 2] [0, 3]
Row 1: [1, 0] [1, 1] [1, 2] [1, 3]
```

```
Row 2: [2, 0] [2, 1] [2, 2] [2, 3]
Row 3: [3, 0] [3, 1] [3, 2] [3, 3]
```

Using arrays to store other properties

Arrays can also be used like objects, to store related information.

```
const arr = [1, 2, 3];
arr.property = "value";
console.log(arr.property); // "value"
```

For example, when an array is the result of a match between a regular expression and a string, the array returns properties and elements that provide information about the match. An array is the return value of `RegExp.prototype.exec()`, `String.prototype.match()`, and `String.prototype.split()`. For information on using arrays with regular expressions, see Regular Expressions.

Working with array-like objects

Some JavaScript objects, such as the `NodeList` returned by `document.getElementsByTagName()` or the `arguments` object made available within the body of a function, look and behave like arrays on the surface but do not share all of their methods. The `arguments` object provides a `length` attribute but does not implement array methods like `forEach()`.

Array methods cannot be called directly on array-like objects.

```
function printArguments() {
  arguments.forEach((item) => {
    console.log(item);
  }); // TypeError: arguments.forEach is not a function
}
```

But you can call them indirectly using `Function.prototype.call()`.

```
function printArguments() {
  Array.prototype.forEach.call(arguments, (item) => {
    console.log(item);
  });
}
```

Array prototype methods can be used on strings as well, since they provide sequential access to their characters in a similar way to arrays:

```
Array.prototype.forEach.call("a string", (chr) => {
  console.log(chr);
});
```

Keyed collections

This chapter introduces collections of data that are indexed by a key; `Map` and `Set` objects contain elements that are iterable in the order of insertion.

Maps

Map object

In JavaScript, a Map object is a collection of key-value pairs where any value (both objects and primitive values) may be used as either a key or a value. This means that, unlike an array, **keys are not restricted to numbers or strings, they can be any type of value**.

Map objects are similar to Objects but have some key differences. Maps allow any value to be used as a key, whereas objects only allow strings or symbols to be used as keys. Also, the size of a Map object can be easily retrieved using the size property, while with Objects, you need to manually keep track of the number of properties.

To create a Map object, you can use the Map() constructor like this:

```
const myMap = new Map();
```

You can also initialize a Map object with an array of key-value pairs like this:

```
const myMap = new Map([
  ['key1', 'value1'],
  ['key2', 'value2'],
  ['key3', 'value3']
]);
```

Here, `key1`, `key2`, and `key3` are the keys, and `value1`, `value2`, and `value3` are the corresponding values.

Map objects provide several methods to manipulate and retrieve data from the Map. Here are some commonly used methods:

- set(key, value) - sets the value for the specified key in the Map object.
- get(key) - returns the value associated with the specified key in the Map object, or undefined if the key does not exist.
- has(key) - returns a Boolean indicating whether the specified key exists in the Map object.
- delete(key) - removes the key-value pair associated with the specified key from the Map object.
- clear() - removes all key-value pairs from the Map object.
- size - returns the number of key-value pairs in the Map object.

Here's an example of how you can use some of these methods:

```
const myMap = new Map();
myMap.set('key1', 'value1');
myMap.set('key2', 'value2');
myMap.set('key3', 'value3');

console.log(myMap.get('key1')); // Output: "value1"
console.log(myMap.has('key2')); // Output: true

myMap.delete('key3');
console.log(myMap); // Output: Map(2) { 'key1' => 'value1', 'key2' => 'value2' }

console.log(myMap.size); // Output: 2

myMap.clear();
console.log(myMap); // Output: Map(0) {}
```

Object and Map compared

Traditionally, objects have been used to map strings to values. Objects allow you to set keys to values, retrieve those values, delete keys, and detect whether something is stored at a key. Map objects, however, have a few more advantages that make them better maps.

- The keys of an Object are Strings or Symbols, where they can be of any value for a Map.
- You can get the size of a Map easily, while you have to manually keep track of size for an Object.
- The iteration of maps is in the insertion order of the elements.
- An Object has a prototype, so there are default keys in the map. (This can be bypassed using map = Object.create(null).)

These three tips can help you to decide whether to use a Map or an Object:

- Use maps over objects when keys are unknown until run time, and when all keys are the same type and all values are the same type.
- Use maps if there is a need to store primitive values as keys because object treats each key as a string whether it's a number value, boolean value or any other primitive value.
- Use objects when there is logic that operates on individual elements.

WeakMap object

A WeakMap object is another type of Map in JavaScript, but with a key difference: **the keys in a WeakMap must be objects, and these objects are weakly held**. This means that if there are no other references to a key object, it can be garbage-collected by the JavaScript engine even if it is still in the WeakMap.

In other words, a WeakMap allows you to associate values with objects without creating strong references to those objects. This can be useful in situations where you want to store some data that is associated with an object, but you don't want to prevent that object from being garbage-collected if there are no other references to it.

To create a WeakMap object, you can use the WeakMap() constructor, like this:

```
const myWeakMap = new WeakMap();
```

You can then add key-value pairs to the WeakMap object using the `set()` method, like this:

```
const keyObject = {};
myWeakMap.set(keyObject, "value");
```

Here, `keyObject` is an object that acts as the key in the WeakMap, and `"value"` is the value associated with that key.

Some key differences between a WeakMap and a regular Map are:

- Only objects can be used as keys in a WeakMap.
- WeakMaps are not iterable (i.e., you cannot use a `for...of` loop to iterate over the key-value pairs).
- WeakMaps do not have a `size` property or a `clear()` method.

Some of the methods available to WeakMap objects include `set()`, `get()`, `has()`, and `delete()`. These methods allow you to add, retrieve, check for existence, and remove key-value pairs in the WeakMap.

Here's an example of how you can use a WeakMap object:

```
const myWeakMap = new WeakMap();
const keyObject = {};

myWeakMap.set(keyObject, "value");

console.log(myWeakMap.get(keyObject)); // Output: "value"

keyObject = null; // Removing the only reference to keyObject
console.log(myWeakMap.get(keyObject)); // Output: undefined (keyObject has been
                                              garbage-collected)
```

Sets

Set object

A Set object is a collection of unique values in JavaScript (an **array-like object without keys, and where every value is unique and is not indexed like in an array**). It can store any type of value, including primitives (like numbers, strings, and booleans) and object references. However, each value can only occur once in the Set object.

To create a Set object, you can use the `Set()` constructor, like this:.

```
const mySet = new Set();
```

You can then add values to the Set object using the add() method, like this:

```
mySet.add(1);
mySet.add("hello");
mySet.add(true);
```

Here, we've added the number 1, the string "hello", and the boolean true to the Set object.

Some methods available to Set objects include `add()`, `delete()`, `has()`, and `clear()`. These methods allow you to add, remove, check for existence, and clear values from the Set object.

Here's an example of how you can use a Set object:

```
const mySet = new Set();

mySet.add(1);
mySet.add(2);
mySet.add(3);

console.log(mySet.has(2)); // Output: true

mySet.delete(2);

console.log(mySet); // Output: Set { 1, 3 }

mySet.clear();

console.log(mySet); // Output: Set(0) {}
```

In this example, we create a Set object and add the numbers 1, 2, and 3 to it. We then use the `has()` method to check if the value 2 exists in the Set object (which it does), and use the `delete()` method to remove the value 2. Finally, we use the `clear()` method to remove all values from the Set object.

Converting between Array and Set

You can create an `Array` from a `Set` using `Array.from` or the spread syntax. Also, the `Set` constructor accepts an `Array` to convert in the other direction.

Set objects store *unique values* — so any duplicate elements from an Array are deleted when converting!

```
Array.from(mySet);
[...mySet2];

mySet2 = new Set([1, 2, 3, 4]);
```

Array and Set compared

Traditionally, a set of elements has been stored in arrays in JavaScript in a lot of situations. The `Set` object, however, has some advantages:

- Deleting Array elements by value (`arr.splice(arr.indexOf(val), 1)`) is very slow.
- `Set` objects let you delete elements by their value. With an array, you would have to `splice` based on an element's index.
- The value NaN cannot be found with `indexOf` in an array.
- `Set` objects store unique values. You don't have to manually keep track of duplicates.

WeakSet object

`WeakSet` objects are **collections of objects**. An object in the `WeakSet` may only occur once. It is unique in the `WeakSet`'s collection, and objects are not enumerable.
The main differences to the `Set` object are:

- In contrast to `Sets`, `WeakSets` are **collections of *objects only***, and not of arbitrary values of any type.
- The `WeakSet` is *weak*: References to objects in the collection are held weakly. If there is no other reference to an object stored in the `WeakSet`, they can be garbage collected. That also means that there is no list of current objects stored in the collection.
- `WeakSets` are not enumerable.

The use cases of WeakSet objects are limited. They will not leak memory, so it can be safe to use DOM elements as a key and mark them for tracking purposes, for example.

Key and value equality of Map and Set

Both the key equality of `Map` objects and the value equality of `Set` objects are based on the SameValueZero algorithm:

- Equality works like the identity comparison operator `===`.
- `-0` and `+0` are considered equal.
- `NaN` is considered equal to itself (contrary to `===`).

Working with objects

JavaScript is designed on a simple object-based paradigm. An object is a collection of properties, and a property is an association between a name (or key) and a value. A property's value can be a function, in which case the property is known as a method.

In JavaScript, an object is a standalone entity, with properties and type. Compare it with a cup, for example. A cup is an object, with properties. A cup has a color, a design, weight, a material it is made of, etc. In the same way, JavaScript objects can have properties, which define their characteristics.

Creating new objects

You can create an object using an object initializer. Alternatively, you can first create a constructor function and then instantiate an object by invoking that function with the `new` operator.

Using object initializers

Object initializers are also called *object literals*. "Object initializer" is consistent with the terminology used by C++.

The syntax for an object using an object initializer is:

```
const obj = {
  property1: value1, // property name may be an identifier
  2: value2, // or a number
  "property n": value3, // or a string
};
```

Each property name before colons is an identifier (either a name, a number, or a string literal), and each `valueN` is an expression whose value is assigned to the property name. The property name can also be an expression.

In this example, the newly created object is assigned to a variable obj — this is optional. If you do not need to refer to this object elsewhere, you do not need to assign it to a variable.

Object initializers are expressions, and each object initializer results in a new object being created whenever the statement in which it appears is executed. Identical object initializers create distinct objects that do not compare to each other as equal.

The following example creates `myHonda` with three properties. Note that the `engine` property is also an object with its own properties.

```
const myHonda = {
  color: "red",
  wheels: 4,
  engine: { cylinders: 4, size: 2.2 },
};
```

Objects created with initializers are called *plain objects*, because they are instances of `Object`, but not any other object type. Some object types have special initializer syntaxes — for example, *array initializers* and *regex literals*.

Using a constructor function

Alternatively, you can create an object with these two steps:

1. Define the object type by writing a constructor function. There is a strong convention, with good reason, to use a capital initial letter.
2. Create an instance of the object with `new`.

To define an object type, create a function for the object type that specifies its name, properties, and methods. For example, suppose you want to create an object type for cars. You want this type of object to be called `Car`, and you want it to have properties for make, model, and year. To do this, you would write the following function:

```
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}
```

Notice the use of `this` to assign values to the object's properties based on the values passed to the function.

Now you can create an object called `myCar` as follows:

```
const myCar = new Car("Eagle", "Talon TSi", 1993);
```

This statement creates `myCar` and assigns it the specified values for its properties. Then the value of `myCar.make` is the string `"Eagle"`, `myCar.model` is the string `"Talon  TSi"`, `myCar.year` is the integer `1993`, and so on. The order of arguments and parameters should be the same.

You can create any number of `Car` objects by calls to `new`. For example,

```
const kenscar = new Car("Nissan", "300ZX", 1992);
const vpgscar = new Car("Mazda", "Miata", 1990);
```

An object can have a property that is itself another object. For example, suppose you define an object called `Person` as follows:

```
function Person(name, age, sex) {
  this.name = name;
  this.age = age;
  this.sex = sex;
}
```

and then instantiate two new Person objects as follows:

```
const rand = new Person("Rand McKinnon", 33, "M");
const ken = new Person("Ken Jones", 39, "M");
```

Then, you can rewrite the definition of `Car` to include an `owner` property that takes a `Person` object, as follows:

```
function Car(make, model, year, owner) {
  this.make = make;
  this.model = model;
  this.year = year;
  this.owner = owner;
}
```

To instantiate the new objects, you then use the following:

```
const car1 = new Car("Eagle", "Talon TSi", 1993, rand);
const car2 = new Car("Nissan", "300ZX", 1992, ken);
```

Notice that instead of passing a literal string or integer value when creating the new objects, the above statements pass the objects rand and ken as the arguments for the owners. Then if you want to find out the name of the owner of car2, you can access the following property:

```
car2.owner.name;
```

You can always add a property to a previously defined object. For example, the statement

```
car1.color = "black";
```

adds a property color to car1, and assigns it a value of "black". However, this does not affect any other objects. To add the new property to all objects of the same type, you have to add the property to the definition of the Car object type.

Using the Object.create() method

Objects can also be created using the Object.create() method. This method can be very useful, because it allows you to choose the prototype object for the object you want to create, without having to define a constructor function:

```
// Animal properties and method encapsulation
const Animal = {
  type: "Invertebrates", // Default value of properties
  displayType() {
    // Method which will display type of Animal
    console.log(this.type);
  },
};

// Create new animal type called animal1
const animal1 = Object.create(Animal);
animal1.displayType(); // Logs: Invertebrates

// Create new animal type called fish
```

```
const fish = Object.create(Animal);
fish.type = "Fishes";
fish.displayType(); // Logs: Fishes
```

Objects and properties

You can create an object using an object initializer. Alternatively, you can first create a constructor function and then instantiate an object by invoking that function with the new operator.

A JavaScript object has properties associated with it. Object properties are basically the same as variables, except that they are associated with objects, not scopes. The properties of an object define the characteristics of the object.

For example, this example creates an object named myCar, with properties named make, model, and year, with their values set to "Ford", "Mustang", and 1969:

```
const myCar = {
  make: "Ford",
  model: "Mustang",
  year: 1969,
};
```

Like JavaScript variables, property names are case-sensitive. Property names can only be strings or Symbols — all keys are converted to strings unless they are Symbols. Array indices are, in fact, properties with string keys that contain integers.

Accessing properties

You can access a property of an object by its property name. Property accessors come in two syntaxes: dot notation and bracket notation. For example, you could access the properties of the myCar object as follows:

```
// Dot notation
myCar.make = "Ford";
myCar.model = "Mustang";
myCar.year = 1969;

// Bracket notation
myCar["make"] = "Ford";
```

```
myCar["model"] = "Mustang";
myCar["year"] = 1969;
```

An object property name can be any JavaScript string or symbol, including an empty string. However, you cannot use dot notation to access a property whose name is not a valid JavaScript identifier. For example, a property name that has a space or a hyphen, that starts with a number, or that is held inside a variable can only be accessed using the bracket notation. This notation is also very useful when property names are to be dynamically determined, i.e. not determinable until runtime. Examples are as follows:

```
const myObj = {};
const str = "myString";
const rand = Math.random();
const anotherObj = {};

// Create additional properties on myObj
myObj.type = "Dot syntax for a key named type";
myObj["date created"] = "This key has a space";
myObj[str] = "This key is in variable str";
myObj[rand] = "A random number is the key here";
myObj[anotherObj] = "This key is object anotherObj";
myObj[""] = "This key is an empty string";

console.log(myObj);
// {
//   type: 'Dot syntax for a key named type',
//   'date created': 'This key has a space',
//   myString: 'This key is in variable str',
//   '0.6398914448618778': 'A random number is the key here',
//   '[object Object]': 'This key is object anotherObj',
//   '': 'This key is an empty string'
// }
console.log(myObj.myString); // 'This key is in variable str'
```

In the above code, the key `anotherObj` is an object, which is neither a string nor a symbol. When it is added to the `myObj`, JavaScript calls the `toString()` method of `anotherObj`, and use the resulting string as the new key.

You can also access properties with a string value stored in a variable. The variable must be passed in bracket notation. In the example above, the variable str held "myString" and it is "myString" which is the property name. Therefore, `myObj.str` will return as undefined.

```
str = "myString";
myObj[str] = "This key is in variable str";
```

```
console.log(myObj.str); // undefined

console.log(myObj[str]); // 'This key is in variable str'
console.log(myObj.myString); // 'This key is in variable str'
```

This allows accessing any property as determined at runtime:

```
let propertyName = "make";
myCar[propertyName] = "Ford";

// access different properties by changing the contents of the variable
propertyName = "model";
myCar[propertyName] = "Mustang";

console.log(myCar); // { make: 'Ford', model: 'Mustang' }
```

However, beware of using square brackets to access properties whose names are given by external input. This may make your code susceptible to object injection attacks.

Nonexistent properties of an object have value `undefined` (and not `null`).

```
myCar.nonexistentProperty; // undefined
```

Enumerating properties

There are three native ways to list/traverse object properties:

- `for...in` loops. This method traverses all of the enumerable string properties of an object as well as its prototype chain.
- `Object.keys(myObj)`. This method returns an array with only the enumerable own string property names ("keys") in the object `myObj`, but not those in the prototype chain.
- `Object.getOwnPropertyNames(myObj)`. This method returns an array containing all the own string property names in the object `myObj`, regardless of if they are enumerable or not.

You can use the bracket notation with `for...in` to iterate over all the enumerable properties of an object. To illustrate how this works, the following function displays the properties of the object when you pass the object and the object's name as arguments to the function:

```
function showProps(obj, objName) {
  let result = "";
  for (const i in obj) {
    // Object.hasOwn() is used to exclude properties from the object's
    // prototype chain and only show "own properties"
    if (Object.hasOwn(obj, i)) {
      result += `${objName}.${i} = ${obj[i]}\n`;
    }
  }
  console.log(result);
}
```

So, the function call `showProps(myCar, 'myCar')` would print the following:

```
myCar.make = Ford
myCar.model = Mustang
myCar.year = 1969
```

The above is equivalent to:

```
function showProps(obj, objName) {
  let result = "";
  Object.keys(obj).forEach((i) => {
    result += `${objName}.${i} = ${obj[i]}\n`;
  });
  console.log(result);
}
```

Deleting properties

You can remove a non-inherited property using the delete operator. The following code shows how to remove a property.

```
// Creates a new object, myobj, with two properties, a and b.
const myobj = new Object();
myobj.a = 5;
myobj.b = 12;
```

```
// Removes the a property, leaving myobj with only the b property.
delete myobj.a;
console.log("a" in myobj); // false
```

Inheritance

All objects in JavaScript inherit from at least one other object. The object being inherited from is known as the prototype, and the inherited properties can be found in the prototype object of the constructor. See Inheritance and the `prototype` chain for more information.

Defining properties for all objects of one type

You can add a property to all objects created through a certain constructor using the `prototype` property. This defines a property that is shared by all objects of the specified type, rather than by just one instance of the object. The following code adds a `color` property to all objects of type `Car`, and then reads the property's value from an instance `car1`.

```
Car.prototype.color = "red";
console.log(car1.color); // "red"
```

Defining methods

A *method* is a function associated with an object, or, put differently, a **method is a property of an object that is a function**. Methods are defined the way normal functions are defined, except that they have to be assigned as the property of an object. See also method definitions for more details. An example is:

```
objectName.methodName = functionName;

const myObj = {
  myMethod: function (params) {
    // do something
  },

  // this works too!
  myOtherMethod(params) {
    // do something else
  },
```

```
  };
```

where `objectName` is an existing object, `methodName` is the name you are assigning to the method, and `functionName` is the name of the function.

You can then call the method in the context of the object as follows:

```
  object.methodName(params);
```

Methods are typically defined on the `prototype` object of the constructor, so that all objects of the same type share the same method. For example, you can define a function that formats and displays the properties of the previously-defined `Car` objects.

```
  Car.prototype.displayCar = function () {
    const result = `A Beautiful ${this.year} ${this.make} ${this.model}`;
    console.log(result);
  };
```

Notice the use of `this` to refer to the object to which the method belongs. Then you can call the `displayCar` method for each of the objects as follows:

```
  car1.displayCar();
  car2.displayCar();
```

Using this for object references

JavaScript has a special keyword, `this`, that you can use within a method to refer to the current object. For example, suppose you have 2 objects, `Manager` and `Intern`. Each object has its own `name`, `age` and `job`. In the function `sayHi()`, notice the use of `this.name`. When added to the 2 objects, the same function will print the message with the name of the respective object it's attached to.

```
  const Manager = {
    name: "Karina",
    age: 27,
    job: "Software Engineer",
  };
```

```
const Intern = {
  name: "Tyrone",
  age: 21,
  job: "Software Engineer Intern",
};

function sayHi() {
  console.log(`Hello, my name is ${this.name}`);
}

// add sayHi function to both objects
Manager.sayHi = sayHi;
Intern.sayHi = sayHi;

Manager.sayHi(); // Hello, my name is Karina
Intern.sayHi(); // Hello, my name is Tyrone
```

`this` is a "hidden parameter" of a function call that's passed in by specifying the object before the function that was called. For example, in `Manager.sayHi()`, `this` is the `Manager` object, because `Manager` comes before the function `sayHi()`. If you access the same function from another object, `this` will change as well. If you use other methods to call the function, like `Function.prototype.call()` or `Reflect.apply()`, you can explicitly pass the value of `this` as an argument.

Defining getters and setters

A **getter is a function associated with a property that gets the value of a specific property**. A **setter is a function associated with a property that sets the value of a specific property**. Together, they can indirectly represent the value of a property.

Getters and setters can be either

- defined within object initializers, or
- added later to any existing object.

Within object initializers, getters and setters are defined like regular methods, but prefixed with the keywords `get` or `set`. The getter method must not expect a parameter, while the setter method expects exactly one parameter (the new value to set). For instance:

```
const myObj = {
  a: 7,
  get b() {
```

```
    return this.a + 1;
  },
  set c(x) {
    this.a = x / 2;
  },
};

console.log(myObj.a); // 7
console.log(myObj.b); // 8, returned from the get b() method
myObj.c = 50; // Calls the set c(x) method
console.log(myObj.a); // 25
```

The `myObj` object's properties are:

- `myObj.a` — a number
- `myObj.b` — a getter that returns `myObj.a` plus 1
- `myObj.c` — a setter that sets the value of `myObj.a` to half of value `myObj.c` is being set to

Getters and setters can also be added to an object at any time after creation using the `Object.defineProperties()` method. This method's first parameter is the object on which you want to define the getter or setter. The second parameter is an object whose property names are the getter or setter names, and whose property values are objects for defining the getter or setter functions. Here's an example that defines the same getter and setter used in the previous example:

```
const myObj = { a: 0 };

Object.defineProperties(myObj, {
  b: {
    get() {
      return this.a + 1;
    },
  },
  c: {
    set(x) {
      this.a = x / 2;
    },
  },
});

myObj.c = 10; // Runs the setter, which assigns 10 / 2 (5) to the 'a' property
console.log(myObj.b); // Runs the getter, which yields a + 1 or 6
```

Which of the two forms to choose depends on your programming style and task at hand. If you can change the definition of the original object, you will probably define getters and setters

through the original initializer. This form is more compact and natural. However, if you need to add getters and setters later — maybe because you did not write the particular object — then the second form is the only possible form. The second form better represents the dynamic nature of JavaScript, but it can make the code hard to read and understand.

Comparing objects

In JavaScript, objects are a reference type. Two distinct objects are never equal, even if they have the same properties. Only comparing the same object reference with itself yields true.

```
// Two variables, two distinct objects with the same properties

const fruit = { name: "apple" };
const fruitbear = { name: "apple" };

fruit == fruitbear; // return false
fruit === fruitbear; // return false
```

```
// Two variables, a single object

const fruit = { name: "apple" };
const fruitbear = fruit; // Assign fruit object reference to fruitbear

// Here fruit and fruitbear are pointing to same object

fruit == fruitbear; // return true
fruit === fruitbear; // return true

fruit.name = "grape";
console.log(fruitbear); // { name: "grape" }; not { name: "apple" }
```

Using classes

In JavaScript, classes are a relatively new feature that were introduced in ECMAScript 2015 (ES6) as a way to provide a simpler and more intuitive syntax for creating objects and defining their behavior.

A class in JavaScript is a template or **blueprint for creating objects that share similar properties and methods**. It defines the properties and methods that each object created from the class will have but does not actually create any objects until it is instantiated with the new keyword.

Overview of classes

If you have some hands-on experience with JavaScript or have followed along with the guide, you probably have already used classes, even if you haven't created one. For example, this may seem familiar to you:

```
const bigDay = new Date(2019, 6, 19);
console.log(bigDay.toLocaleDateString());
if (bigDay.getTime() < Date.now()) {
  console.log("Once upon a time...");
}
```

On the first line, we created an instance of the class `Date` and called it `bigDay`. On the second line, we called a method `toLocaleDateString()` on the `bigDay` instance, which returns a string. Then, we compared two numbers: one returned from the `getTime()` method, the other directly called from the `Date` class *itself*, as `Date.now().`

`Date` is a built-in class of JavaScript. From this example, we can get some basic ideas of what classes do:

- Classes create objects through the `new` operator.
- Each object has some properties (data or method) added by the class.
- The class stores some properties (data or method) itself, which are usually used to interact with instances.

These correspond to the three key features of classes:

- Constructor;
- Instance methods and instance fields;
- Static methods and static fields.

Declaring a class

Classes are usually created with class declarations.

```
class MyClass {
  // class body...
}
```

Within a class body, there is a range of features available.

```
class MyClass {
  // Constructor
  constructor() {
    // Constructor body
  }
  // Instance field
  myField = "foo";
  // Instance method
  myMethod() {
    // myMethod body
  }
  // Static field
  static myStaticField = "bar";
  // Static method
  static myStaticMethod() {
    // myStaticMethod body
  }
  // Static block
  static {
    // Static initialization code
  }
  // Fields, methods, static fields, and static methods all have
  // "private" forms
  #myPrivateField = "bar";
}
```

If you came from a pre-ES6 world, you may be more familiar with using functions as constructors. The pattern above would roughly translate to the following with function constructors:

```javascript
function MyClass() {
  this.myField = "foo";
  // Constructor body
}
MyClass.myStaticField = "bar";
MyClass.myStaticMethod = function () {
  // myStaticMethod body
};
MyClass.prototype.myMethod = function () {
  // myMethod body
};

(function () {
  // Static initialization code
})();
```

Constructing a class

After a class has been declared, you can create instances of it using the new operator.

```javascript
const myInstance = new MyClass();
console.log(myInstance.myField); // 'foo'
myInstance.myMethod();
```

Typical function constructors can both be constructed with new and called without new. However, attempting to "call" a class without new will result in an error.

```javascript
const myInstance = MyClass(); // TypeError: Class constructor MyClass cannot be
                              invoked without 'new'
```

Class declaration hoisting

Unlike function declarations, class declarations are not hoisted which means you cannot use a class before it is declared.

```
new MyClass(); // ReferenceError: Cannot access 'MyClass' before initialization
class MyClass {}
```

This behavior is similar to variables declared with `let` and `const`.

Class expressions

Similar to functions, class declarations also have their expression counterparts.

```
const MyClass = class {
  // Class body...
};
```

Class expressions can have names as well. The expression's name is only visible to the class's body.

```
const MyClass = class MyClassLongerName {
  // Class body. Here MyClass and MyClassLongerName point to the same class.
};
new MyClassLongerName(); // ReferenceError: MyClassLongerName is not defined
```

Constructor

Perhaps the most important job of a class is to act as a "factory" for objects. For example, when we use the `Date` constructor, we expect it to give a new object which represents the date data we passed in — which we can then manipulate with other methods the instance exposes. In classes, the instance creation is done by the constructor.

As an example, we would create a class called `Color`, which represents a specific color. Users create colors by passing in an RGB triplet.

```
class Color {
  constructor(r, g, b) {
    this.values = [r, g, b]; // Assign the RGB values as a property of `this`.
  }
}
```

Open your browser's devtools, paste the above code into the console, and then create an instance:

```
const red = new Color(255, 0, 0);
console.log(red);
```

You should see some output like this:

```
Object { values: (3) […] }
  values: Array(3) [ 255, 0, 0 ]
```

You have successfully created a `Color` instance, and the instance has a `values` property, which is an array of the RGB values you passed in. That is pretty much equivalent to the following:

```
function createColor(r, g, b) {
  return {
    values: [r, g, b],
  };
}
```

The constructor's syntax is exactly the same as a normal function — which means you can use other syntaxes, like rest parameters:

```
class Color {
  constructor(...values) {
    this.values = values;
  }
}
const red = new Color(255, 0, 0);
// Creates an instance with the same shape as above.
```

Each time you call `new`, a different instance is created.

```
const red = new Color(255, 0, 0);
const anotherRed = new Color(255, 0, 0);
console.log(red === anotherRed); // false
```

Within a class constructor, the value of `this` points to the newly created instance. You can assign properties to it, or read existing properties (especially methods — which we will cover next).

The `this` value will be automatically returned as the result of `new`. You are advised to not return any value from the constructor — because if you return a non-primitive value, it will become the value of the `new` expression, and the value of `this` is dropped.

```
class MyClass {
  constructor() {
    this.myField = "foo";
    return {};
  }
}

console.log(new MyClass().myField); // undefined
```

Instance methods

If a class only has a constructor, it is not much different from a `createX` factory function which just creates plain objects. However, the power of classes is that they can be used as "templates" which automatically assign methods to instances.

For example, for `Date` instances, you can use a range of methods to get different information from a single date value, such as the year, month, day of the week, etc. You can also set those values through the `setX` counterparts like `setFullYear`.

For our own `Color` class, we can add a method called `getRed` which returns the red value of the color.

```
class Color {
   constructor(r, g, b) {
    this.values = [r, g, b];
  }
  getRed() {
    return this.values[0];
  }
}

const red = new Color(255, 0, 0);
console.log(red.getRed()); // 255
```

Without methods, you may be tempted to define the function within the constructor:

```
class Color {
  constructor(r, g, b) {
    this.values = [r, g, b];
    this.getRed = function () {
      return this.values[0];
    };
  }
}
```

This also works. However, the problem is that this creates a new function every time a Color instance is created, even when they all do the same thing!

In contrast, if you use a method, it will be shared between all instances. A function can be shared between all instances, but still, its behavior differs when different instances call it because the value of this is different. If you are curious where this method is stored — it's defined on the prototype of all instances or `Color.prototype.`

Similarly, we can create a new method called setRed, which sets the red value of the color.

```
class Color {
  constructor(r, g, b) {
    this.values = [r, g, b];
  }
  getRed() {
    return this.values[0];
  }
  setRed(value) {
    this.values[0] = value;
  }
}

const red = new Color(255, 0, 0);
red.setRed(0);
console.log(red.getRed()); // 0; of course, it should be called "black" at this
stage!
```

Private fields

You might be wondering: why do we want to go to the trouble of using `getRed` and `setRed` methods when we can directly access the `values` array on the instance?

```
class Color {
  constructor(r, g, b) {
    this.values = [r, g, b];
  }
}

const red = new Color(255, 0, 0);
red.values[0] = 0;
console.log(red.values[0]); // 0
```

There is a philosophy in object-oriented programming called "**encapsulation**". This means you should not access the underlying implementation of an object, but instead use well-abstracted methods to interact with it. For example, if we suddenly decided to represent colors as HSL instead:

```
class Color {
  constructor(r, g, b) {
    // values is now an HSL array!
    this.values = rgbToHSL([r, g, b]);
  }
  getRed() {
    return this.values[0];
  }
  setRed(value) {
    this.values[0] = value;
  }
}
const red = new Color(255, 0, 0);
console.log(red.values[0]); // 0; It's not 255 anymore, because the H value for
                                pure    red is 0
```

The user assumption that `values` mean the RGB value suddenly collapses, and it may cause their logic to break. So, if you are an implementor of a class, you would want to hide the internal data structure of your instance from your user, both to keep the API clean and to prevent the user's code from breaking when you do some "harmless refactors". In classes, this is done through *private fields*.

A private field is an identifier prefixed with # (the hash symbol). The hash is an integral part of the field's name, which means a private property can never have a name clash with a public property. In order to refer to a private field anywhere in the class, you must declare it in the class body (you can't create private property on the fly). Apart from this, a private field is pretty much equivalent to a normal property.

```
class Color {
  // Declare: every Color instance has a private field called #values.
  #values;
  constructor(r, g, b) {
    this.#values = [r, g, b];
  }
  getRed() {
    return this.#values[0];
  }
  setRed(value) {
    this.#values[0] = value;
  }
}

const red = new Color(255, 0, 0);
console.log(red.getRed()); // 255
```

Accessing private fields outside the class is an early syntax error. The language can guard against this because #privateField is a special syntax, so it can do some static analysis and find all usage of private fields before even evaluating the code.

```
console.log(red.#values);
// SyntaxError: Private field '#values' must be declared in an enclosing class
```

Private fields in JavaScript are *hard private*: if the class does not implement methods that expose these private fields, there's absolutely no mechanism to retrieve them from outside the class.

If we leave the values property exposed, our users can easily circumvent that check by assigning to values[0] directly, and create invalid colors.

A class method can read the private fields of other instances, as long as they belong to the same class.

```
class Color {
  #values;
  constructor(r, g, b) {
    this.#values = [r, g, b];
  }
  redDifference(anotherColor) {
    // #values doesn't necessarily need to be accessed from this:
    // you can access private fields of other instances belonging
    // to the same class.
    return this.#values[0] - anotherColor.#values[0];
  }
}

const red = new Color(255, 0, 0);
const crimson = new Color(220, 20, 60);
red.redDifference(crimson); // 35
```

However, if `anotherColor` is not a Color instance, #values won't exist. (Even if another class has an identically named #values private field, it's not referring to the same thing and cannot be accessed here.) Accessing a nonexistent private property throws an error instead of returning `undefined` like normal properties do. If you don't know if a private field exists on an object and you wish to access it without using `try/catch` to handle the error, you can use the `in` operator.

```
class Color {
  #values;
  constructor(r, g, b) {
    this.#values = [r, g, b];
  }
  redDifference(anotherColor) {
    if (!(#values in anotherColor)) {
      throw new TypeError("Color instance expected");
    }
    return this.#values[0] - anotherColor.#values[0];
  }
}
```

Keep in mind that the # is a special identifier syntax, and you can't use the field name as if it's a string. "#values" in `anotherColor` would look for a property name literally called "#values", instead of a private field.

There are some limitations in using private properties: the same name can't be declared twice in a single class, and they can't be deleted. Both lead to early syntax errors.

Methods, getters, and setters can be private as well. They're useful when you have something complex that the class needs to do internally but no other part of the code should be allowed to call.

For example, imagine creating HTML custom elements that should do something somewhat complicated when clicked/tapped/otherwise activated. Furthermore, the somewhat complicated things that happen when the element is clicked should be restricted to this class, because no other part of the JavaScript will (or should) ever access it.

```
class Counter extends HTMLElement {
  #xValue = 0;
  constructor() {
    super();
    this.onclick = this.#clicked.bind(this);
  }
  get #x() {
    return this.#xValue;
  }
  set #x(value) {
    this.#xValue = value;
    window.requestAnimationFrame(this.#render.bind(this));
  }
  #clicked() {
    this.#x++;
  }
  #render() {
    this.textContent = this.#x.toString();
  }
  connectedCallback() {
    this.#render();
  }
}

customElements.define("num-counter", Counter);
```

In this case, pretty much every field and method is private to the class. Thus, it presents an interface to the rest of the code that's essentially just like a built-in HTML element. No other part of the program has the power to affect any of the internals of Counter.

Accessor fields

color.getRed() and color.setRed() allow us to read and write to the red value of a color. However, using methods to simply access a property is still somewhat unergonomic in JavaScript. Accessor fields allow us to manipulate something as if its an "actual property".

```
class Color {
  constructor(r, g, b) {
    this.values = [r, g, b];
  }
  get red() {
    return this.values[0];
  }
  set red(value) {
    this.values[0] = value;
  }
}

const red = new Color(255, 0, 0);
red.red = 0;
console.log(red.red); // 0
```

It looks as if the object has a property called red — but actually, no such property exists on the instance! There are only two methods, but they are prefixed with get and set, which allows them to be manipulated as if they were properties.

If a field only has a getter but no setter, it will be effectively read-only.

```
class Color {
  constructor(r, g, b) {
    this.values = [r, g, b];
  }
  get red() {
    return this.values[0];
  }
}

const red = new Color(255, 0, 0);
red.red = 0;
console.log(red.red); // 255
```

In strict mode, the `red.red = 0` line will throw a type error: "`Cannot set property red of #<Color>` `which has only a getter`". In non-strict mode, the assignment is silently ignored.

Public fields

Private fields also have their public counterparts, which allow every instance to have a property. Fields are usually designed to be independent of the constructor's parameters.

```
class MyClass {
  luckyNumber = Math.random();
}
console.log(new MyClass().luckyNumber); // 0.5
console.log(new MyClass().luckyNumber); // 0.3
```

Public fields are almost equivalent to assigning a property to this. For example, the above example can also be converted to:

```
class MyClass {
  constructor() {
    this.luckyNumber = Math.random();
  }
}
```

Static properties

With the `Date` example, we have also encountered the `Date.now()` method, which returns the current date. This method does not belong to any date instance — it belongs to the class itself. However, it's put on the `Date` class instead of being exposed as a global `DateNow()` function, because it's most useful when dealing with date instances.

Static properties are a group of class features that are defined on the class itself, rather than on individual instances of the class. These features include:

- Static methods
- Static fields
- Static getters and setters

Everything also has private counterparts. For example, for our `Color` class, we can create a static method that checks whether a given triplet is a valid RGB value:

```
class Color {
  static isValid(r, g, b) {
    return r >= 0 && r <= 255 && g >= 0 && g <= 255 && b >= 0 && b <= 255;
  }
}

Color.isValid(255, 0, 0); // true
Color.isValid(1000, 0, 0); // false
```

Static properties are very similar to their instance counterparts, except that:

- They are all prefixed with `static`, and
- They are not accessible from instances.

```
console.log(new Color(0, 0, 0).isValid); // undefined
```

There is also a special construct called a *static initialization block*, which is a block of code that runs when the class is first loaded.

```
class MyClass {
  static {
    MyClass.myStaticProperty = "foo";
  }
}

console.log(MyClass.myStaticProperty); // 'foo'
```

Static initialization blocks are almost equivalent to immediately executing some code after a class has been declared. The only difference is that they have access to static private properties.

Extends and inheritance

A key feature that classes bring about (in addition to ergonomic encapsulation with private fields) is *inheritance*, which means one object can "borrow" a large part of another object's behaviors, while overriding or enhancing certain parts with its own logic.

For example, suppose our `Color` class now needs to support transparency. We may be tempted to add a new field that indicates its transparency:

```
class Color {
  #values;
  constructor(r, g, b, a = 1) {
    this.#values = [r, g, b, a];
  }
  get alpha() {
    return this.#values[3];
  }
  set alpha(value) {
    if (value < 0 || value > 1) {
      throw new RangeError("Alpha value must be between 0 and 1");
    }
    this.#values[3] = value;
  }
}
```

However, this means every instance — even the vast majority which aren't transparent (those with an alpha value of 1) — will have to have the extra alpha value, which is not very elegant. Plus, if the features keep growing, our `Color` class will become very bloated and hard to maintain.

Instead, in object-oriented programming, we would create a *derived class*. The **derived class** has access to all public properties of the parent class. In JavaScript, derived classes are declared with an `extends` clause, which indicates the class it extends from.

```
class ColorWithAlpha extends Color {
  #alpha;
  constructor(r, g, b, a) {
    super(r, g, b);
    this.#alpha = a;
  }
  get alpha() {
    return this.#alpha;
  }
  set alpha(value) {
    if (value < 0 || value > 1) {
      throw new RangeError("Alpha value must be between 0 and 1");
```

```
    }
    this.#alpha = value;
  }
}
```

There are a few things that have immediately come to attention. First is that in the constructor, we are calling `super(r, g, b)`. It is a language requirement to call `super()` before accessing `this`. The `super()` call calls the parent class's constructor to initialize `this` — here it's roughly equivalent to `this = new Color(r, g, b)`. You can have code before `super()`, but you cannot access `this` before `super()` — the language prevents you from accessing the uninitialized `this`.

After the parent class is done with modifying `this`, the derived class can do its own logic. Here we added a private field called `#alpha`, and also provided a pair of getter/setters to interact with them.

A derived class inherits all methods from its parent. For example, although `ColorWithAlpha` doesn't declare a `get red()` accessor itself, you can still access red because this behavior is specified by the parent class:

```
const color = new ColorWithAlpha(255, 0, 0, 0.5);
console.log(color.red); // 255
```

Derived classes can also override methods from the parent class. For example, all classes implicitly inherit the *Object* class, which defines some basic methods like `toString()`. However, the base `toString()` method is notoriously useless, because it prints `[object Object]` in most cases:

```
console.log(red.toString()); // [object Object]
```

Instead, our class can override it to print the color's RGB values:

```
class Color {
  #values;
  // …
  toString() {
    return this.#values.join(", ");
  }
}

console.log(new Color(255, 0, 0).toString()); // '255, 0, 0'
```

Within derived classes, you can access the parent class's methods by using `super`. This allows you to build enhancement methods and avoid code duplication.

```
class ColorWithAlpha extends Color {
  #alpha;
  // …
  toString() {
    // Call the parent class's toString() and build on the return value
    return `${super.toString()}, ${this.#alpha}`;
  }
}

console.log(new ColorWithAlpha(255, 0, 0, 0.5).toString()); // '255, 0, 0, 0.5'
```

When you use `extends`, the static methods inherit from each other as well, so you can also override or enhance them.

```
class ColorWithAlpha extends Color {
  // ...
  static isValid(r, g, b, a) {
    // Call the parent class's isValid() and build on the return value
    return super.isValid(r, g, b) && a >= 0 && a <= 1;
  }
}

console.log(ColorWithAlpha.isValid(255, 0, 0, -1)); // false
```

Derived classes don't have access to the parent class's private fields — this is another key aspect to JavaScript private fields being "hard private". Private fields are scoped to the class body itself and do not grant access to *any* outside code.

```
class ColorWithAlpha extends Color {
  log() {
    console.log(this.#values); // SyntaxError: Private field '#values' must be
                                          declared in an enclosing class
  }
}
```

A class can only extend from one class. This prevents problems in multiple inheritance like the diamond problem. However, due to the dynamic nature of JavaScript, it's still possible to achieve the effect of multiple inheritance through class composition and mixins.

Instances of derived classes are also instances of the base class.

149

```
const color = new ColorWithAlpha(255, 0, 0, 0.5);
console.log(color instanceof Color); // true
console.log(color instanceof ColorWithAlpha); // true
```

Why classes?

Classes introduce a *paradigm*, or a way to organize your code. Classes are the foundations of object-oriented programming, which is built on concepts like *inheritance* and *polymorphism* (especially subtype polymorphism). However, many people are philosophically against certain OOP practices and don't use classes as a result.

For example, one thing that makes Date objects infamous is that they're *mutable*.

```
function incrementDay(date) {
  return date.setDate(date.getDate() + 1);
}
const date = new Date(); // 2019-06-19
const newDay = incrementDay(date);
console.log(newDay); // 2019-06-20
// The old date is modified as well!?
console.log(date); // 2019-06-20
```

Mutability and internal state are important aspects of object-oriented programming, but often make code hard to reason with — because any seemingly innocent operation may have unexpected side effects and change the behavior in other parts of the program.

In order to reuse code, we usually resort to extending classes, which can create big hierarchies of inheritance patterns.
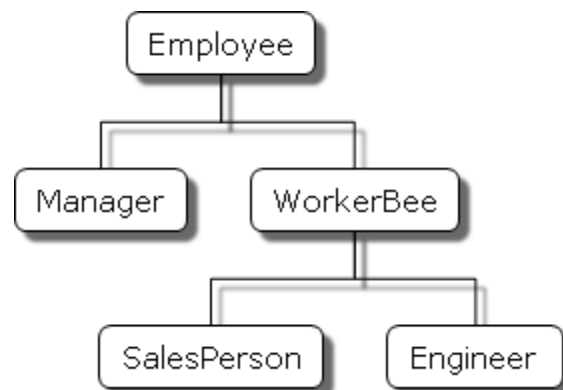
However, it is often hard to describe inheritance cleanly when one class can only extend one other class. Often, we want the behavior of multiple classes. In Java, this is done through interfaces; in JavaScript, it can be done through mixins. But at the end of the day, it's still not very convenient.

On the brighter side, classes are a very powerful way to organize our code on a higher level. For example, without the `Color` class, we may need to create a dozen of utility functions:

```
function isRed(color) {
  return color.red === 255;
}
function isValidColor(color) {
  return (
    color.red >= 0 &&
    color.red <= 255 &&
    color.green >= 0 &&
    color.green <= 255 &&
    color.blue >= 0 &&
    color.blue <= 255
  );
}
```

But with classes, we can congregate them all under the `Color` namespace, which improves readability. In addition, the introduction of private fields allows us to hide certain data from downstream users, creating a clean API.

In general, you should consider using classes when you want to create objects that store their own internal data and expose a lot of behavior. Take built-in JavaScript classes as examples:

- The `Map` and `Set` classes store a collection of elements and allow you to access them by key using `get()`, `set()`, `has()`, etc.
- The `Date` class stores a date as a Unix timestamp (a number) and allows you to format, update, and read individual date components.
- The `Error` class stores information about a particular exception, including the error message, stack trace, cause, etc. It's one of the few classes that come with a rich inheritance structure: there are multiple built-in classes like `TypeError` and `ReferenceError` that extend `Error`. In the case of errors, this inheritance allows refining the semantics of errors: each error class represents a specific type of error, which can be easily checked with `instanceof`.

JavaScript allows flexible object-oriented code organization.

This lesson was adapted from MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_classes

Using promises

A Promise is an **object representing the eventual completion or failure of an asynchronous operation**. Since most people are consumers of already-created promises, this guide will explain the consumption of returned promises before explaining how to create them.

Essentially, a promise is a returned object to which you attach callbacks, instead of passing callbacks into a function.

Chaining

A common need is to execute two or more asynchronous operations back to back, where each subsequent operation starts when the previous operation succeeds, with the result from the previous step. In the old days, doing several asynchronous operations in a row would lead to the classic callback pyramid of doom:

```
doSomething(function (result) {
  doSomethingElse(result, function (newResult) {
    doThirdThing(newResult, function (finalResult) {
      console.log(`Got the final result: ${finalResult}`);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

With promises, we accomplish this by creating a promise chain. The API design of promises makes this great, because callbacks are attached to the returned promise object, instead of being passed into a function.

Here's the magic: the `then()` function returns a **new promise**, different from the original:

```
const promise = doSomething();
const promise2 = promise.then(successCallback, failureCallback);
```

This second promise (`promise2`) represents the completion not just of `doSomething()`, but also of the `successCallback` or `failureCallback` you passed in — which can be other asynchronous functions returning a promise.

With this pattern, you can create longer chains of processing, where each promise represents the completion of one asynchronous step in the chain. In addition, the arguments to then are optional, and `catch(failureCallback)` is short for `then(null, failureCallback)` — so if your error handling code is the same for all steps, you can attach it to the end of the chain:

```
doSomething()
  .then(function (result) {
    return doSomethingElse(result);
  })
  .then(function (newResult) {
    return doThirdThing(newResult);
  })
  .then(function (finalResult) {
    console.log(`Got the final result: ${finalResult}`);
  })
  .catch(failureCallback);
```

You might see this expressed with arrow functions instead:

```
doSomething()
  .then((result) => doSomethingElse(result))
  .then((newResult) => doThirdThing(newResult))
  .then((finalResult) => {
    console.log(`Got the final result: ${finalResult}`);
  })
  .catch(failureCallback);
```

Important: **Always return results**, otherwise callbacks won't catch the result of a previous promise (with arrow functions, `() => x` is short for `() => { return x; }`)

Therefore, whenever your operation encounters a promise, return it and defer its handling to the next then handler.

```
const listOfIngredients = [];

doSomething()
  .then((url) =>
    fetch(url)
      .then((res) => res.json())
      .then((data) => {
        listOfIngredients.push(data);
```

```
      }),
    )
    .then(() => {
      console.log(listOfIngredients);
    });

  // OR

  doSomething()
    .then((url) => fetch(url))
    .then((res) => res.json())
    .then((data) => {
      listOfIngredients.push(data);
    })
    .then(() => {
      console.log(listOfIngredients);
    });
```

Nesting

Nesting is a control structure to limit the scope of `catch` statements. Specifically, a nested `catch` only catches failures in its scope and below, not errors higher up in the chain outside the nested scope. When used correctly, this gives greater precision in error recovery:

```
  doSomethingCritical()
    .then((result) =>
      doSomethingOptional(result)
        .then((optionalResult) => doSomethingExtraNice(optionalResult))
        .catch((e) => {}),
    ) // Ignore if optional stuff fails; proceed.
    .then(() => moreCriticalStuff())
    .catch((e) => console.error(`Critical failure: ${e.message}`));
```

The inner error-silencing catch handler only catches failures from `doSomethingOptional()` and `doSomethingExtraNice()`, after which the code resumes with `moreCriticalStuff()`. Importantly, if `doSomethingCritical()` fails, its error is caught by the final (outer) catch only, and does not get swallowed by the inner catch handler.

Chaining after a catch

It's possible to chain after a failure, i.e. a `catch`, which is useful to accomplish new actions even after an action failed in the chain. Read the following example:

```
new Promise((resolve, reject) => {
  console.log("Initial");

  resolve();
})
  .then(() => {
    throw new Error("Something failed");

    console.log("Do this");
  })
  .catch(() => {
    console.error("Do that");
  })
  .then(() => {
    console.log("Do this, no matter what happened before");
  });
```

This will output the following text:

```
Initial
Do that
Do this, no matter what happened before
```

Note: "Do this" is not displayed because the "Something failed" error caused a rejection.

Common mistakes

```
// Bad example! Spot 3 mistakes!

doSomething()
  .then(function (result) {
    // Forgot to return promise from inner chain + unnecessary nesting
    doSomethingElse(result).then((newResult) => doThirdThing(newResult));
  })
  .then(() => doFourthThing());
// Forgot to terminate chain with a catch!
```

The first mistake is to not chain things together properly. This happens when we create a new promise but forget to return it. As a consequence, the chain is broken — or rather, we have two

independent chains racing. This means `doFourthThing()` won't wait for `doSomethingElse()` or `doThirdThing()` to finish, and will run concurrently with them.

The second mistake is to nest unnecessarily, enabling the first mistake. Nesting also limits the scope of inner error handlers, which—if unintended—can lead to uncaught errors.

The third mistake is forgetting to terminate chains with catch.

A good rule is to always either return or terminate promise chains, and as soon as you get a new promise, return it immediately, to flatten things:

```
doSomething()
  .then(function (result) {
    // If using a full function expression: return the promise
    return doSomethingElse(result);
  })
  // If using arrow functions: omit the braces and implicitly return the result
  .then((newResult) => doThirdThing(newResult))
  // Even if the previous chained promise returns a result, the next one
  // doesn't necessarily have to use it. You can pass a handler that doesn't
  // consume any result.
  .then((/* result ignored */) => doFourthThing())
  // Always end the promise chain with a catch handler to avoid any
  // unhandled rejections!
  .catch((error) => console.error(error));
```

Error handling

You might remember seeing `failureCallback` three times in the pyramid of doom earlier, compared to only once at the end of the promise chain:

```
doSomething()
  .then((result) => doSomethingElse(result))
  .then((newResult) => doThirdThing(newResult))
  .then((finalResult) => console.log(`Got the final result: ${finalResult}`))
  .catch(failureCallback);
```

If there's an exception, the browser will look down the chain for `.catch()` handlers or `onRejected`. This is very much modeled after how synchronous code works:

```
  try {
    const result = syncDoSomething();
    const newResult = syncDoSomethingElse(result);
    const finalResult = syncDoThirdThing(newResult);
    console.log(`Got the final result: ${finalResult}`);
  } catch (error) {
    failureCallback(error);
  }
```

This symmetry with asynchronous code culminates in the `async/await` syntax:

```
async function foo() {
  try {
    const result = await doSomething();
    const newResult = await doSomethingElse(result);
    const finalResult = await doThirdThing(newResult);
    console.log(`Got the final result: ${finalResult}`);
  } catch (error) {
    failureCallback(error);
  }
}
```

It builds on promises — for example, `doSomething()` is the same function as before, so there's minimal refactoring needed to change from promises to `async/await`.

Promise rejection events

If a promise rejection event is not handled by any handler, it bubbles to the top of the call stack, and the host needs to surface it. On the web, whenever a promise is rejected, one of two events is sent to the global scope. The two events are:

```
unhandledrejection
```

Sent when a promise is rejected but there is no rejection handler available.

```
rejectionhandled
```

Sent when a handler is attached to a rejected `promise` that has already caused an unhandledrejection event.

In both cases, the event (of type `PromiseRejectionEvent`) has as members a `promise` property indicating the promise that was rejected, and a `reason` property that provides the reason given for the promise to be rejected.

In Node.js, handling promise rejection is slightly different. You capture unhandled rejections by adding a handler for the Node.js `unhandledRejection` event (notice the difference in capitalization of the name), like this:

```
process.on("unhandledRejection", (reason, promise) => {
  // Add code here to examine the "promise" and "reason" values
});
```

Composition

There are four composition tools for running asynchronous operations concurrently: `Promise.all()`, `Promise.allSettled()`, `Promise.any()`, and `Promise.race()`.

We can start operations at the same time and wait for them all to finish like this:

```
Promise.all([func1(), func2(), func3()]).then(([result1, result2, result3]) => {
  // use result1, result2 and result3
});
```

If one of the promises in the array rejects, Promise.all() immediately rejects the returned promise and aborts the other operations. This may cause unexpected state or behavior. Promise.allSettled() is another composition tool that ensures all operations are complete before resolving.

These methods all run promises concurrently — a sequence of promises are started simultaneously and do not wait for each other. Sequential composition is possible using some clever JavaScript:

```
[func1, func2, func3]
  .reduce((p, f) => p.then(f), Promise.resolve())
  .then((result3) => {
    /* use result3 */
```

```
    });
```

In this example, we reduce an array of asynchronous functions down to a promise chain. The code above is equivalent to:

```
Promise.resolve()
  .then(func1)
  .then(func2)
  .then(func3)
  .then((result3) => {
    /* use result3 */
  });
```

This can be made into a reusable compose function, which is common in functional programming:

```
const applyAsync = (acc, val) => acc.then(val);
const composeAsync =
  (...funcs) =>
  (x) =>
    funcs.reduce(applyAsync, Promise.resolve(x));
```

The `composeAsync()` function accepts any number of functions as arguments and returns a new function that accepts an initial value to be passed through the composition pipeline:

```
const transformData = composeAsync(func1, func2, func3);
const result3 = transformData(data);
```

Sequential composition can also be done more succinctly with async/await:

```
let result;
for (const f of [func1, func2, func3]) {
  result = await f(result);
}
/* use last result (i.e. result3) */
```

However, before you compose promises sequentially, consider if it's really necessary — it's always better to run promises concurrently so that they don't unnecessarily block each other unless one promise's execution depends on another's result.

Creating a Promise around an old callback API

A `Promise` can be created from scratch using its constructor. This should be needed only to wrap old APIs.

In an ideal world, all asynchronous functions would already return promises. Unfortunately, some APIs still expect success and/or failure callbacks to be passed in the old way. The most obvious example is the `setTimeout()` function:

```
setTimeout(() => saySomething("10 seconds passed"), 10 * 1000);
```

Mixing old-style callbacks and promises is problematic. If `saySomething()` fails or contains a programming error, nothing catches it. This is intrinsic to the design of setTimeout.

Luckily we can wrap `setTimeout` in a promise. The best practice is to wrap the callback-accepting functions at the lowest possible level, and then never call them directly again:

```
const wait = (ms) => new Promise((resolve) => setTimeout(resolve, ms));

wait(10 * 1000)
  .then(() => saySomething("10 seconds"))
  .catch(failureCallback);
```

The promise constructor takes an executor function that lets us resolve or reject a promise manually. Since `setTimeout()` doesn't really fail, we left out reject in this case. For more information on how the executor function works, see the `Promise()` reference.

Timing

Guarantees

In the callback-based API, **when and how the callback gets called** depends on the API implementor. For example, the callback may be called synchronously or asynchronously:

```
function doSomething(callback) {
  if (Math.random() > 0.5) {
    callback();
  } else {
    setTimeout(() => callback(), 1000);
  }
}
```

To avoid surprises, functions passed to `then()` will never be called synchronously, even with an already-resolved promise:

```
Promise.resolve().then(() => console.log(2));
console.log(1);
// Logs: 1, 2
```

Instead of running immediately, the passed-in function is put on a microtask queue, which means it runs later, just before control is returned to the event loop; i.e. pretty soon:

```
const wait = (ms) => new Promise((resolve) => setTimeout(resolve, ms));

wait(0).then(() => console.log(4));
Promise.resolve()
  .then(() => console.log(2))
  .then(() => console.log(3));
console.log(1); // 1, 2, 3, 4
```

Task queues vs. microtasks

Promise callbacks are handled as a microtask whereas `setTimeout()` callbacks are handled as task queues.

```
const promise = new Promise((resolve, reject) => {
  console.log("Promise callback");
  resolve();
}).then((result) => {
  console.log("Promise callback (.then)");
});

setTimeout(() => {
  console.log("event-loop cycle: Promise (fulfilled)", promise);
```

```
    }, 0);

    console.log("Promise (pending)", promise);
```

The code above will output:

```
Promise callback
Promise (pending) Promise {<pending>}
Promise callback (.then)
event-loop cycle: Promise (fulfilled) Promise {<fulfilled>}
```

Iterators and generators

Iterators and Generators bring the concept of iteration directly into the core language and provide a mechanism for customizing the behavior of `for...of` loops.

Iterators

In JavaScript an **iterator** is an object which defines a sequence and potentially a return value upon its termination.

Specifically, an iterator is any object which implements the <u>Iterator protocol</u> by having a `next()` method that returns an object with two properties:

`value`   - The next value in the iteration sequence.

`done`   - This is `true` if the last value in the sequence has already been consumed. If `value` is present alongside `done`, it is the iterator's return value.

Once created, an iterator object can be iterated explicitly by repeatedly calling `next()`. Iterating over an iterator is said to consume the iterator, because it is generally only possible to do once. After a terminating value has been yielded additional calls to `next()` should continue to return `{done: true}`.

The most common iterator in JavaScript is the Array iterator, which returns each value in the associated array in sequence.

While it is easy to imagine that all iterators could be expressed as arrays, this is not true. Arrays must be allocated in their entirety, but iterators are consumed only as necessary. Because of this, iterators can express sequences of unlimited size, such as the range of integers between `0` and `Infinity`.

Here is an example which can do just that. It allows creation of a simple range iterator which defines a sequence of integers from `start` (inclusive) to `end` (exclusive) spaced `step` apart. Its final return value is the size of the sequence it created, tracked by the variable `iterationCount`.

```
function makeRangeIterator(start = 0, end = Infinity, step = 1) {
  let nextIndex = start;
  let iterationCount = 0;

  const rangeIterator = {
    next() {
      let result;
      if (nextIndex < end) {
        result = { value: nextIndex, done: false };
        nextIndex += step;
        iterationCount++;
        return result;
      }
      return { value: iterationCount, done: true };
    },
  };
  return rangeIterator;
}
```

Using the iterator then looks like this:

```
const it = makeRangeIterator(1, 10, 2);

let result = it.next();
while (!result.done) {
  console.log(result.value); // 1 3 5 7 9
  result = it.next();
}

console.log("Iterated  over  sequence  of  size:",  result.value);  //  [5  numbers
returned,    that took interval in between: 0 to 10]
```

It is not possible to know reflectively whether a particular object is an iterator. If you need to do this, use Iterables.

Generator functions

While custom iterators are a useful tool, their creation requires careful programming due to the need to explicitly maintain their internal state. **Generator functions** provide a powerful alternative: they allow you to define an iterative algorithm by writing a single function whose execution is not continuous. Generator functions are written using the `function*` syntax.

When called, generator functions do not initially execute their code. Instead, they return a special type of iterator, called a **Generator**. When a value is consumed by calling the generator's `next` method, the Generator function executes until it encounters the `yield` keyword.

The function can be called as many times as desired, and returns a new Generator each time. Each Generator may only be iterated once.

We can now adapt the example from above. The behavior of this code is identical, but the implementation is much easier to write and read.

```
function* makeRangeIterator(start = 0, end = Infinity, step = 1) {
  let iterationCount = 0;
  for (let i = start; i < end; i += step) {
    iterationCount++;
    yield i;
  }
  return iterationCount;
}
```

Iterables

An object is **iterable** if it defines its iteration behavior, such as what values are looped over in a `for...of` construct. Some built-in types, such as `Array` or `Map`, have a default iteration behavior, while other types (such as `Object`) do not.

In order to be iterable, an object must implement the **@@iterator** method. This means that the object (or one of the objects up its prototype chain) must have a property with a `Symbol.iterator` key.

It may be possible to iterate over an iterable more than once, or only once. It is up to the programmer to know which is the case. Iterables which can iterate only once (such as Generators) customarily return `this` from their @@iterator method, whereas iterables which can be iterated many times must return a new iterator on each invocation of @@iterator.

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

const it = makeIterator();
```

```
for (const itItem of it) {
  console.log(itItem);
}

console.log(it[Symbol.iterator]() === it); // true

// This example show us generator(iterator) is iterable object,
// which has the @@iterator method return the it (itself),
// and consequently, the it object can iterate only _once_.

// If we change it's @@iterator method to a function/generator
// which returns a new iterator/generator object, (it)
// can iterate many times

it[Symbol.iterator] = function* () {
  yield 2;
  yield 1;
};
```

User-defined iterables

You can make your own iterables like this:

```
const myIterable = {
  *[Symbol.iterator]() {
    yield 1;
    yield 2;
    yield 3;
  },
};
```

User-defined iterables can be used in `for...of` loops or the spread syntax as usual.

```
for (const value of myIterable) {
  console.log(value);
}
// 1
// 2
// 3

[...myIterable]; // [1, 2, 3]
```

Built-in iterables

`String`, `Array`, `TypedArray`, `Map` and `Set` are all built-in iterables, because their prototype objects all have a `Symbol.iterator` method.

Syntaxes expecting iterables

Some statements and expressions expect iterables. For example: `the for-of` loops, `yield*`.

```
for (const value of ["a", "b", "c"]) {
  console.log(value);
}
// "a"
// "b"
// "c"

[..."abc"];
// ["a", "b", "c"]

function* gen() {
  yield* ["a", "b", "c"];
}

gen().next();
// { value: "a", done: false }

[a, b, c] = new Set(["a", "b", "c"]);
a;
// "a"
```

Advanced generators

Generators compute their `yield`ed values on *demand*, which allows them to efficiently represent sequences that are expensive to compute (or even infinite sequences, as demonstrated above).

The `next()` method also accepts a value, which can be used to modify the internal state of the generator. A value passed to `next()` will be received by `yield`.

A value passed to the first invocation of `next()` is always ignored.

Here is the fibonacci generator using `next(x)` to restart the sequence:

```
function* fibonacci() {
  let current = 0;
  let next = 1;
  while (true) {
    const reset = yield current;
    [current, next] = [next, next + current];
    if (reset) {
      current = 0;
      next = 1;
    }
  }
}

const sequence = fibonacci();
console.log(sequence.next().value); // 0
console.log(sequence.next().value); // 1
console.log(sequence.next().value); // 1
console.log(sequence.next().value); // 2
console.log(sequence.next().value); // 3
console.log(sequence.next().value); // 5
console.log(sequence.next().value); // 8
console.log(sequence.next(true).value); // 0
console.log(sequence.next().value); // 1
console.log(sequence.next().value); // 1
console.log(sequence.next().value); // 2
```

You can force a generator to throw an exception by calling its `throw()` method and passing the exception value it should throw. This exception will be thrown from the current suspended context of the generator, as if the yield that is currently suspended were instead a `throw value` statement.

If the exception is not caught from within the generator, it will propagate up through the call to `throw()`, and subsequent calls to `next()` will result in the done property being true.

Generators have a `return(value)` method that returns the given value and finishes the generator itself.

Meta programming

The `Proxy` and `Reflect` objects allow you to intercept and define custom behavior for fundamental language operations (e.g. property lookup, assignment, enumeration, function invocation, etc.). With the help of these two objects you are able to program at the meta level of JavaScript.

Proxies

Proxy objects allow you to intercept certain operations and to implement custom behaviors.

For example, getting a property on an object:

```
const handler = {
  get(target, name) {
    return name in target ? target[name] : 42;
  },
};

const p = new Proxy({}, handler);
p.a = 1;
console.log(p.a, p.b); // 1, 42
```

The `Proxy` object defines a `target` (an empty object here) and a `handler` object, in which a `get` trap is implemented. Here, an object that is proxied will not return `undefined` when getting undefined properties, but will instead return the number `42`.

Terminology

The following terms are used when talking about the functionality of proxies.

handler
  Placeholder object which contains traps.

traps
  The methods that provide property access. (This is analogous to the concept of *traps* in operating systems.)

target

Object which the proxy virtualizes. It is often used as storage backend for the proxy. Invariants (semantics that remain unchanged) regarding object non-extensibility or non-configurable properties are verified against the target.

invariants

Semantics that remain unchanged when implementing custom operations are called *invariants*. If you violate the invariants of a handler, a `TypeError` will be thrown.

Handlers and traps

The following table summarizes the available traps available to `Proxy` objects.

Handler / trap	Interceptions	Invariants
`handler.getPrototypeOf()`	`Object.getPrototypeOf()` `Reflect.getPrototypeOf()` `__proto__` `Object.prototype.isPrototypeOf()` `instanceof`	• getPrototypeOf method must return an object or null. • If `target` is not extensible, `Object.getPrototypeOf(proxy)` method must return the same value as `Object.getPrototypeOf(target)`.
`handler.setPrototypeOf()`	`Object.setPrototypeOf()` `Reflect.setPrototypeOf()`	If `target` is not extensible, the prototype parameter must be the same value as `Object.getPrototypeOf(target)`.
`handler.isExtensible()`	`Object.isExtensible()` `Reflect.isExtensible()`	`Object.isExtensible(proxy)` must return the same value as `Object.isExtensible(target)`.
`handler.preventExtensions()`	`Object.preventExtensions()` `Reflect.preventExtensions()`	`Object.preventExtensions(proxy)` only returns `true` if `Object.isExtensible(proxy)` is `false`.

`handler.getOwnPrope`
`rtyDescriptor()`
	`Object.getOwnPropertyDesc`
	`riptor()`
	
	`Reflect.getOwnPropertyDes`
	`criptor()`
		• `getOwnPropertyDescriptor` must return an object or undefined.
		• A property cannot be reported as non-existent if it exists as a non-configurable own property of `target`.
		• A property cannot be reported as non-existent if it exists as an own property of `target` and `target` is not extensible.
		• A property cannot be reported as existent if it does not exists as an own property of `target` and `target` is not extensible.
		• A property cannot be reported as non-configurable if it does not exist as an own property of `target` or if it exists as a configurable own property of `target`.
		• The result of `Object.getOwnPropertyDescriptor(`*`target`*`)` can be applied to `target` using `Object.definePropert`
		`y` and will not throw an exception.

`handler.definePrope`
`rty()`
	`Object.defineProperty()`
	
	`Reflect.defineProperty()`
		• A property cannot be added if `target` is not extensible.
		• A property cannot be added as (or modified to be) non-configurable if it does not exist as a non-configurable own property of `target`.
		• A property may not be non-configurable if a corresponding configurable property of `target` exists.
		• If a property has a corresponding target object property, then `Object.defineProperty(target,`
		`prop, descriptor)` will not throw an exception.
		• In strict mode, a `false` value returned from the `defineProperty` handler will throw a `TypeError` exception.

`handler.has()`	Property query:
	    `foo in proxy`
	
	Inherited property query:
	    `foo in`
	    `Object.create(`*`proxy`*`)`
	    `Reflect.has()`
		• A property cannot be reported as non-existent, if it exists as a non-configurable own property of `target.`
		• A property cannot be reported as non-existent if it exists as an own property of `target` and `target` is not extensible.

handler.get()	Property access:
	    `proxy[foo]`
	    `proxy.bar`
	
	Inherited property access:
	    `Object.create(proxy)[`
	    `foo]`
	    `Reflect.get()`
		• The value reported for a property must be the same as the value of the corresponding `target` property if `target`'s property is a non-writable, non-configurable data property.
		• The value reported for a property must be undefined if the corresponding `target` property is non-configurable accessor property that has undefined as its `[[Get]]` attribute.

handler.set()	Property assignment:
	    `proxy[foo] = bar`
	    `proxy.foo = bar`
	
	Inherited property assignment:
	    `Object.create(`*proxy*`)[`
	    `foo] = bar`
	    `Reflect.set()`
		• Cannot change the value of a property to be different from the value of the corresponding `target` property if the corresponding `target` property is a non-writable, non-configurable data property.
		• Cannot set the value of a property if the corresponding `target` property is a non-configurable accessor property that has undefined as its `[[Set]]` attribute.
		• In strict mode, a `false` return value from the set handler will throw a `TypeError` exception.

handler.deleteProperty()	Property deletion:
	    `delete proxy[foo]`
	    `delete proxy.foo`
	    `Reflect.deleteProperty()`
		A property cannot be deleted if it exists as a non-configurable own property of `target`.

handler.ownKeys()	`Object.getOwnPropertyNames()`
	
	`Object.getOwnPropertySymbols()`
	
	`Object.keys()`
	`Reflect.ownKeys()`
		• The result of `ownKeys` is a List.
		• The Type of each result List element is either `String` or `Symbol`.
		• The result List must contain the keys of all non-configurable own properties of `target`.
		• If the `target` object is not extensible, then the result List must contain all the keys of the own properties of `target` and no other values.

handler.apply()	`proxy(..args)`
	`Function.prototype.apply()` and `Function.prototype.call()`
	`Reflect.apply()`
		There are no invariants for the `handler.apply` method.

handler.construct()	`new proxy(...args)` `Reflect.construct()`	The result must be an `Object`.

Revocable Proxy

The `Proxy.revocable()` method is used to create a revocable `Proxy` object. This means that the proxy can be revoked via the function `revoke` and switches the proxy off.

Afterwards, any operation on the proxy leads to a `TypeError`.

```
const revocable = Proxy.revocable(
  {},
  {
    get(target, name) {
      return `[[${name}]]`;
    },
  },
);
const proxy = revocable.proxy;
console.log(proxy.foo); // "[[foo]]"

revocable.revoke();

console.log(proxy.foo); // TypeError: Cannot perform 'get' on a proxy that has been
                        //                              revoked
proxy.foo = 1; // TypeError: Cannot perform 'set' on a proxy that has been revoked
delete proxy.foo; // TypeError: Cannot perform 'deleteProperty' on a proxy that has
                  //                       been revoked
console.log(typeof proxy); // "object", typeof doesn't trigger any trap
```

Reflection

`Reflect` is a built-in object that provides methods for interceptable JavaScript operations. The methods are the same as those of the proxy handlers.

`Reflect` is not a function object.

`Reflect` helps with forwarding default operations from the handler to the `target`.

With `Reflect.has()` for example, you get the `in` operator as a function:

```
Reflect.has(Object, "assign"); // true
```

A better apply() function

Before `Reflect`, you typically use the `Function.prototype.apply()` method to call a function with a given `this` value and `arguments` provided as an array (or an array-like object).

```
Function.prototype.apply.call(Math.floor, undefined, [1.75]);
```

With `Reflect.apply` this becomes less verbose and easier to understand:

```
Reflect.apply(Math.floor, undefined, [1.75]);
// 1

Reflect.apply(String.fromCharCode, undefined, [104, 101, 108, 108, 111]);
// "hello"

Reflect.apply(RegExp.prototype.exec, /ab/, ["confabulation"]).index;
// 4

Reflect.apply("".charAt, "ponies", [3]);
// "i"
```

Checking if property definition has been successful

With `Object.defineProperty`, which returns an object if successful, or throws a `TypeError` otherwise, you would use a `try...catch` block to catch any error that occurred while defining a property. Because `Reflect.defineProperty` returns a Boolean success status, you can just use an `if...else` block here:

```
if (Reflect.defineProperty(target, property, attributes)) {
  // success
} else {
  // failure
}
```

This lesson was adapted from MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Meta_programming

JavaScript modules

This guide gives you all you need to get started with JavaScript module syntax.

A background on modules

JavaScript programs started off pretty small — most of its usage in the early days was to do isolated scripting tasks, providing a bit of interactivity to your web pages where needed, so large scripts were generally not needed. Fast forward a few years and we now have complete applications being run in browsers with a lot of JavaScript, as well as JavaScript being used in other contexts (Node.js, for example).

It has therefore made sense in recent years to start thinking about providing mechanisms for splitting JavaScript programs up into separate modules that can be imported when needed. Node.js has had this ability for a long time, and there are a number of JavaScript libraries and frameworks that enable module usage (for example, other CommonJS and AMD-based module systems like RequireJS, and more recently Webpack and Babel).

The good news is that modern browsers have started to support module functionality natively. This can only be a good thing — browsers can optimize loading of modules, making it more efficient than having to use a library and do all of that extra client-side processing and extra round trips.

Basic example structure

In our first example (see basic-modules) we have a file structure as follows:

```
index.html
main.js
modules/
    canvas.js
    square.js
```

The modules directory's two modules are described below:

- `canvas.js` — contains functions related to setting up the canvas:
  - `create()` — creates a canvas with a specified `width` and `height` inside a wrapper `<div>` with a specified ID, which is itself appended inside a specified parent element. Returns an object containing the canvas's 2D context and the wrapper's ID.
  - `createReportList()` — creates an unordered list appended inside a specified wrapper element, which can be used to output report data into. Returns the list's ID.
- `square.js` — contains:
  - `name` — a constant containing the string 'square'.
  - `draw()` — draws a square on a specified canvas, with a specified size, position, and color. Returns an object containing the square's size, position, and color.
  - `reportArea()` — writes a square's area to a specific report list, given its length.
  - `reportPerimeter()` — writes a square's perimeter to a specific report list, given its length.

Aside — .mjs versus .js

Throughout this article, we've used `.js` extensions for our module files, but in other resources you may see the `.mjs` extension used instead. V8's documentation recommends this, for example. The reasons given are:

- It is good for clarity, i.e. it makes it clear which files are modules, and which are regular JavaScript.
- It ensures that your module files are parsed as a module by runtimes such as Node.js, and build tools such as Babel.

To get modules to work correctly in a browser, you need to make sure that your server is serving them with a `Content-Type` header that contains a JavaScript MIME type such as text/javascript. If you don't, you'll get a strict MIME type checking error along the lines of "The server responded with a non-JavaScript MIME type" and the browser won't run your JavaScript. Most servers already set the correct type for `.js` files, but not yet for `.mjs` files. Servers that already serve `.mjs` files correctly include GitHub Pages and http-server for Node.js.

For learning and portability purposes, we decided to keep to .js.

If you really value the clarity of using .mjs for modules versus using .js for "normal" JavaScript files, but don't want to run into the problem described above, you could always use .mjs during development and convert them to .js during your build step.

It is also worth noting that:

- Some tools may never support `.mjs`.
- The `<script type="module">` attribute is used to denote when a module is being pointed to, as you'll see below.

Exporting module features

The first thing you do to get access to module features is export them. This is done using the export statement.

The easiest way to use it is to place it in front of any items you want exported out of the module, for example:

```
export const name = "square";

export function draw(ctx, length, x, y, color) {
  ctx.fillStyle = color;
  ctx.fillRect(x, y, length, length);

  return { length, x, y, color };
}
```

You can export functions, `var`, `let`, `const`, and — as we'll see later — classes. They need to be top-level items; you can't use export inside a function, for example.

A more convenient way of exporting all the items you want to export is to use a single export statement at the end of your module file, followed by a comma-separated list of the features you want to export wrapped in curly braces. For example:

```
export { name, draw, reportArea, reportPerimeter };
```

Importing features into your script

Once you've exported some features out of your module, you need to import them into your script to be able to use them. The simplest way to do this is as follows:

```
import { name, draw, reportArea, reportPerimeter } from "./modules/square.js";
```

You use the `import` statement, followed by a comma-separated list of the features you want to import wrapped in curly braces, followed by the keyword `from`, followed by the *module specifier*.

The module specifier provides a string that the JavaScript environment can resolve to a path to the module file. In a browser, this could be a path relative to the site root, which for our basic-modules example would be `/js-examples/module-examples/basic-modules`. However, here we are instead using the dot (`.`) syntax to mean "the current location", followed by the relative path to the file we are trying to find. This is much better than writing out the entire absolute path each time, as relative paths are shorter and make the URL portable — the example will still work if you move it to a different location in the site hierarchy.

So for example:

```
/js-examples/module-examples/basic-modules/modules/square.js
```

becomes

```
./modules/square.js
```

Once you've imported the features into your script, you can use them just like they were defined inside the same file. The following is found in main.js, below the import lines:

```
const myCanvas = create("myCanvas", document.body, 480, 320);
const reportList = createReportList(myCanvas.id);

const square1 = draw(myCanvas.ctx, 50, 50, 100, "blue");
reportArea(square1.length, reportList);
reportPerimeter(square1.length, reportList);
```

Note: **The imported values are read-only views of the features that were exported**. Similar to **const variables**, you cannot re-assign the variable that was imported, but you can still modify properties of object values. The value can only be re-assigned by the module exporting it. See the import reference for an example.

Importing modules using import maps

Import maps allow developers to instead specify almost any text they want in the module specifier when importing a module; the map provides a corresponding value that will replace the text when the module URL is resolved.

For example, the `imports` key in the import map below defines a "module specifier map" JSON object where the property names can be used as module specifiers, and the corresponding values will be substituted when the browser resolves the module URL. The values must be absolute or relative URLs. Relative URLs are resolved to absolute URL addresses using the base URL of the document containing the import map.

```
<script type="importmap">
  {
    "imports": {
      "shapes": "./shapes/square.js",
      "shapes/square": "./modules/shapes/square.js",
      "https://example.com/shapes/": "/shapes/square/",
      "https://example.com/shapes/square.js": "./shapes/square.js",
      "../shapes/square": "./shapes/square.js"
    }
  }
</script>
```

The import map is defined using a JSON object inside a `<script>` element with the type attribute set to `importmap`. There can only be one import map in the document, and because it is used to resolve which modules are loaded in both static and dynamic imports, it must be declared before any `<script>` elements that import modules.

With this map you can now use the property names above as module specifiers. If there is no trailing forward slash on the module specifier key then the whole module specifier key is matched and substituted. For example, below we match bare module names, and remap a URL to another path.

```
// Bare module names as module specifiers
import { name as squareNameOne } from "shapes";
import { name as squareNameTwo } from "shapes/square";

// Remap a URL to another URL
```

```
   import { name as squareNameThree } from
"https://example.com/shapes/mod/square.js";
```

If the module specifier has a trailing forward slash then the value must have one as well, and the key is matched as a "path prefix". This allows remapping of whole classes of URLs.

```
   // Remap a URL as a prefix ( https://example.com/shapes/)
   import { name as squareNameFour } from "https://example.com/shapes/square.js";
```

It is possible for multiple keys in an import map to be valid matches for a module specifier. For example, a module specifier of `shapes/circle/` could match the module specifier keys `shapes/` and `shapes/circle/`. In this case the browser will select the most specific (longest) matching module specifier key.

Import maps allow modules to be imported using bare module names (as in Node.js), and can also simulate importing modules from packages, both with and without file extensions. While not shown above, they also allow particular versions of a library to be imported, based on the path of the script that is importing the module.

Feature detection

You can check support for import maps using the `HTMLScriptElement.supports()` static method (which is itself broadly supported):

```
   if (HTMLScriptElement.supports?.("importmap")) {
     console.log("Browser supports import maps.");
   }
```

Importing modules as bare names

In some JavaScript environments, such as Node.js, you can use bare names for the module specifier. This works because the environment can resolve module names to a standard location in the file system. For example, you might use the following syntax to import the "square" module.

```
   import { name, draw, reportArea, reportPerimeter } from "square";
```

To use bare names on a browser you need an import map, which provides the information needed by the browser to resolve module specifiers to URLs (JavaScript will throw a `TypeError` if it attempts to import a module specifier that can't be resolved to a module location).

Below you can see a map that defines a `square` module specifier key, which in this case maps to a relative address value.

```
<script type="importmap">
  {
    "imports": {
      "square": "./shapes/square.js"
    }
  }
</script>
```

With this map we can now use a bare name when we import the module:

```
import { name as squareName, draw } from "square";
```

Remapping module paths

Module specifier map entries, where both the specifier key and its associated value have a trailing forward slash (/), can be used as a path-prefix. This allows the remapping of a whole set of import URLs from one location to another.

Packages of modules

The following JSON import map definition maps `lodash` as a bare name, and the module specifier prefix `lodash/` to the path `/node_modules/lodash-es/` (resolved to the document base URL):

```
{
  "imports": {
    "lodash": "/node_modules/lodash-es/lodash.js",
    "lodash/": "/node_modules/lodash-es/"
  }
}
```

With this mapping you can import both the whole "package", using the bare name, and modules within it (using the path mapping):

```
import _ from "lodash";
import fp from "lodash/fp.js";
```

It is possible to import `fp` above without the `.js` file extension, but you would need to create a bare module specifier key for that file, such as `lodash/fp`, rather than using the path. This may be reasonable for just one module, but scales poorly if you wish to import many modules.

General URL remapping

A module specifier key doesn't have to be path — it can also be an absolute URL (or a URL-like relative path like `./`, `../`, `/`). This may be useful if you want to remap a module that has absolute paths to a resource with your own local resources.

```
{
  "imports": {
    "https://www.unpkg.com/moment/": "/node_modules/moment/"
  }
}
```

Scoped modules for version management

Ecosystems like Node use package managers such as npm to manage modules and their dependencies. The package manager ensures that each module is separated from other modules and their dependencies. As a result, while a complex application might include the same module multiple times with several different versions in different parts of the module graph, users do not need to think about this complexity.

Import maps similarly allow you to have multiple versions of dependencies in your application and refer to them using the same module specifier. You implement this with the `scopes` key, which allows you to provide module specifier maps that will be used depending on the path of the script performing the import. The example below demonstrates this.

```
{
  "imports": {
    "coolmodule": "/node_modules/coolmodule/index.js"
```

```
    },
    "scopes": {
      "/node_modules/dependency/": {
        "coolmodule": "/node_modules/some/other/location/coolmodule/index.js"
      }
    }
  }
```

With this mapping, if a script with an URL that contains `/node_modules/dependency/` imports `coolmodule`, the version in `/node_modules/some/other/location/coolmodule/index.js` will be used. The map in `imports` is used as a fallback if there is no matching scope in the scoped map, or the matching scopes don't contain a matching specifier. For example, if `coolmodule` is imported from a script with a non-matching scope path, then the module specifier map in imports will be used instead, mapping to the version in `/node_modules/coolmodule/index.js`.

Note that the path used to select a scope does not affect how the address is resolved. The value in the mapped path does not have to match the scopes path, and relative paths are still resolved to the base URL of the script that contains the import map.

Just as for module specifier maps, you can have many scope keys, and these may contain overlapping paths. If multiple scopes match the referrer URL, then the most specific scope path is checked first (the longest scope key) for a matching specifier. The browsers will fall back to the next most specific matching scoped path if there is no matching specifier, and so on. If there is no matching specifier in any of the matching scopes, the browser checks for a match in the module specifier map in the `imports` key.

Scoped modules for version management

Script files used by websites often have hashed filenames to simplify caching. The downside of this approach is that if a module changes, any modules that import it using its hashed filename will also need to be updated/regenerated. This potentially results in a cascade of updates, which is wasteful of network resources.

Import maps provide a convenient solution to this problem. Rather than depending on specific hashed filenames, applications and scripts instead depend on an un-hashed version of the module name (address). An import map like the one below then provides a mapping to the actual script file.

```
{
  "imports": {
    "main_script": "/node/srcs/application-fg7744e1b.js",
```

```
      "dependency_script": "/node/srcs/dependency-3qn7e4b1q.js"
    }
  }
```

If `dependency_script` changes, then its hash contained in the file name changes as well. In this case, we only need to update the import map to reflect the changed name of the module. We don't have to update the source of any JavaScript code that depends on it, because the specifier in the import statement does not change.

Applying the module to your HTML

Now we just need to apply the `main.js` module to our HTML page. This is very similar to how we apply a regular script to a page, with a few notable differences.

First of all, you need to include `type="module"` in the `<script>` element, to declare this script as a module. To import the `main.js` script, we use this:

```
<script type="module" src="main.js"></script>
```

You can also embed the module's script directly into the HTML file by placing the JavaScript code within the body of the `<script>` element:

```
<script type="module">
  /* JavaScript module code here */
</script>
```

The script into which you import the module features basically acts as the top-level module. If you omit it, Firefox for example gives you an error of "SyntaxError: import declarations may only appear at top level of a module".

You can only use `import` and `export` statements inside modules, not regular scripts.

Other differences between modules and standard scripts

- You need to pay attention to local testing — if you try to load the HTML file locally (i.e. with a `file://` URL), you'll run into CORS errors due to JavaScript module security requirements. You need to do your testing through a server.

- Also, note that you might get different behavior from sections of script defined inside modules as opposed to in standard scripts. This is because modules use strict mode automatically.

- There is no need to use the `defer` attribute (see <script> attributes) when loading a module script; modules are deferred automatically.

- Modules are only executed once, even if they have been referenced in multiple `<script>` tags.

- Last but not least, let's make this clear — **module features are imported into the scope of a single script — they aren't available in the global scope**. Therefore, you will only be able to access imported features in the script they are imported into, and you won't be able to access them from the JavaScript console, for example. You'll still get syntax errors shown in the DevTools, but you'll not be able to use some of the debugging techniques you might have expected to use.

Module-defined variables are scoped to the module unless explicitly attached to the global object. However, globally-defined variables are available within the module. For example, given the following code:

```
<!DOCTYPE html>
<html lang="en-US">
  <head>
    <meta charset="UTF-8" />
    <title></title>
    <link rel="stylesheet" href="" />
  </head>
  <body>
    <div id="main"></div>
    <script>
      // A var statement creates a global variable.
      var text = "Hello";
    </script>
    <script type="module" src="./render.js"></script>
  </body>
</html>
```

```
/* render.js */
document.getElementById("main").innerText = text;
```

The page would still render `Hello`, because the global variables `text` and `document` are available in the module. (Also note from this example that a module doesn't necessarily need an import/export statement — the only thing needed is for the entry point to have `type="module"`.)

Default exports versus named exports

The functionality we've exported so far has been comprised of named exports — each item (be it a function, const, etc.) has been referred to by its name upon export, and that name has been used to refer to it on import as well.

There is also a type of export called the default export — this is designed to make it easy to have a default function provided by a module, and also helps JavaScript modules to interoperate with existing CommonJS and AMD module systems.

Let's look at an example as we explain how it works. In our basic-modules `square.js` you can find a function called `randomSquare()` that creates a square with a random color, size, and position. We want to export this as our default, so at the bottom of the file we write this:

```
export default randomSquare;
```

Note the lack of curly braces.

We could instead prepend export default onto the function and define it as an anonymous function, like this:

```
export default function (ctx) {
  // …
}
```

Over in our `main.js` file, we import the default function using this line:

```
import randomSquare from "./modules/square.js";
```

Again, note the lack of curly braces. This is because there is only one default export allowed per module, and we know that `randomSquare` is it. The above line is basically shorthand for:

```
import { default as randomSquare } from "./modules/square.js";
```

Renaming imports and exports

In JavaScript, you can rename imports and exports using the `as` keyword. This can be useful when you want to use a different name for an imported or exported entity than the one defined in the original module.

Here's an example of how to rename imports and exports using the `as` keyword:

```
// my-module.js
export const foo = 'foo';
export const bar = 'bar';
export const baz = 'baz';
```

In this example, we have a module that exports three variables: `foo`, `bar`, and `baz`. We can import these variables into another file and rename them like this:

```
// main.js
import { foo as myFoo, bar as myBar, baz as myBaz } from './my-module.js';

console.log(myFoo); // logs 'foo'
console.log(myBar); // logs 'bar'
console.log(myBaz); // logs 'baz'
```

In this example, we're importing the variables `foo`, `bar`, and `baz` from `my-module.js` using the `import` statement. We're also renaming each variable using the `as` keyword: `foo` is renamed to `myFoo`, `bar` is renamed to `myBar`, and `baz` is renamed to `myBaz`.

We can also use the `as` keyword to rename entities when exporting from a module:

```
// my-module.js
const foo = 'foo';
const bar = 'bar';
const baz = 'baz';

export { foo as myFoo, bar as myBar, baz as myBaz };
```

In this example, we're exporting the variables `foo`, `bar`, and `baz`, but we're renaming each variable using the `as` keyword: `foo` is renamed to `myFoo`, `bar` is renamed to `myBar`, and `baz` is renamed to `myBaz`. This allows other modules to import these variables with the new names:

```
// main.js
import { myFoo, myBar, myBaz } from './my-module.js';

console.log(myFoo); // logs 'foo'
console.log(myBar); // logs 'bar'
console.log(myBaz); // logs 'baz'
```

In general, renaming imports and exports can be useful for avoiding naming collisions, making code more readable, or adhering to naming conventions.

Creating a module object

In JavaScript, a module is a self-contained unit of code that can be imported and exported to other modules. A module can encapsulate variables, functions, classes, or other entities, making them private and only accessible to other modules through a defined interface.

To create a module object in JavaScript, you can use the `Module` function which is built into modern web browsers. Here's an example of how to create a simple module object:

```
// my-module.js
const privateVariable = 'This variable is private';

function privateFunction() {
  console.log('This function is private');
}

export const publicVariable = 'This variable is public';

export function publicFunction() {
```

```
    console.log('This function is public');
  }
```

In this example, we've defined a module that contains a private variable and function, as well as a public variable and function. The public entities are exported using the export keyword.

To use this module in another file, we can import the public entities using the import keyword:

```
// main.js
import { publicVariable, publicFunction } from './my-module.js';

console.log(publicVariable); // logs 'This variable is public'

publicFunction(); // logs 'This function is public'
```

Here, we've imported the publicVariable and publicFunction entities from my-module.js using the import statement. We can then use these entities in our main file.

It's worth noting that the Module function is not supported in all browsers. If you need to support older browsers, you may need to use a module bundler like Webpack, Rollup, or Parcel to create modules instead. These bundlers can convert your module code into a format that is compatible with older browsers.

Modules and classes

To export a class as a module, you can use the export keyword followed by the class keyword:

```
// my-class.js
export class MyClass {
  constructor() {
    // constructor code
  }

  myMethod() {
    // method code
  }
}
```

In this example, we're exporting a class called MyClass from a module called my-class.js. The class has a constructor and a method called myMethod.

To import this class in another module, you can use the `import` keyword followed by the class name and the path to the module file:

```
// main.js
import { MyClass } from './my-class.js';

const myObject = new MyClass();
myObject.myMethod();
```

In this example, we're importing the `MyClass` class from the `my-class.js` module file using the `import` statement. We're then creating a new instance of the class and calling the `myMethod` method.

You can also use the `as` keyword to rename the imported class:

```
// main.js
import { MyClass as MyRenamedClass } from './my-class.js';

const myObject = new MyRenamedClass();
myObject.myMethod();
```

In this example, we're importing the `MyClass` class from the `my-class.js` module file and renaming it to `MyRenamedClass` using the `as` keyword. We're then creating a new instance of the renamed class and calling the `myMethod` method.

Overall, exporting and importing classes as modules in JavaScript follows the same syntax as exporting and importing other types of entities such as functions and variables.

Aggregating modules

Aggregating modules in JavaScript means combining multiple modules into a single, larger module. This can be done to simplify the codebase, reduce the number of requests made to the server, or improve performance.

One common way to aggregate modules is by using an ES6 module bundler like Webpack, Rollup, or Parcel. These bundlers can analyze your code and create a single, optimized JavaScript file that includes all of your modules.

Here's an example:

Suppose we have three modules called `foo.js`, `bar.js`, and `baz.js`:

```
// foo.js
export function foo() {
  console.log('foo');
}

// bar.js
export function bar() {
  console.log('bar');
}

// baz.js
export function baz() {
  console.log('baz');
}
```

We can aggregate these modules into a single module by creating a new module called `all.js` that imports and re-exports the modules:

```
// all.js
export * from './foo.js';
export * from './bar.js';
export * from './baz.js';
```

In this example, we're using the `export *` syntax to re-export all of the functions from each of the three modules. We can then import these functions from `all.js` in our main application file:

```
// main.js
import { foo, bar, baz } from './all.js';

foo(); // logs 'foo'
bar(); // logs 'bar'
baz(); // logs 'baz'
```

By aggregating the three modules into a single file, we've reduced the number of requests made to the server and simplified the codebase.

However, it's worth noting that aggregating modules can lead to larger file sizes, which can negatively impact performance. As with any optimization technique, it's important to weigh the benefits against the costs and to test the performance of your application to ensure that it meets your performance goals.

Dynamic module loading

Dynamic module loading is a feature in JavaScript that allows you to load modules on demand, rather than loading them all at once when the page is initially loaded. This can be particularly useful for large web applications where you may have a large number of modules that aren't always needed, or for cases where you want to reduce initial load times.

The most common way to dynamically load a module in JavaScript is by using the `import()` function, which returns a promise that resolves to the module namespace object. Here's an example:

```
import('./myModule.js')
  .then(module => {
    // Do something with the module
  })
  .catch(error => {
    // Handle any errors that occurred while loading the module
  });
```

In this example, we're using `import()` to load the `myModule.js` file. When the module is loaded, the promise resolves with a reference to the module namespace object, which we can then use to access the module's exports.

It's worth noting that dynamic module loading is a relatively new feature in JavaScript and may not be supported in all browsers. You can check if a browser supports dynamic module loading using the `supportsDynamicImport` property on the `import` object:

```
if ('import' in window && 'supportsDynamicImport' in window.import) {
  // Dynamic module loading is supported
} else {
  // Dynamic module loading is not supported
}
```

Overall, dynamic module loading is a powerful feature in JavaScript that can help you improve the performance of your web applications by only loading the modules that you need, when you need them.

<div style="border:1px solid black; text-align:center">

Top level await

</div>

Top-level `await` is a feature in ECMAScript 2021 (ES2021) that allows you to use `await` at the top level of a module, outside of an `async` function. Prior to ES2021, `await` could only be used inside an `async` function.

With top-level `await`, you can use `await` to wait for a promise to resolve before continuing execution of the module. This can simplify your code and make it easier to work with asynchronous operations.

Here's an example of how to use top-level `await` in a JavaScript module:

```javascript
// my-module.js
const myPromise = new Promise(resolve => {
  setTimeout(() => {
    resolve('Hello, world!');
  }, 1000);
});

export default async function() {
  const result = await myPromise;
  console.log(result);
}
```

In this example, we're defining a module that exports an `async` function. Inside the function, we're defining a promise that resolves with the string 'Hello, world!' after a 1 second delay. We're then using `await` to wait for the promise to resolve and logging the result to the console.

We can then import and use the module in another file like this:

```javascript
// main.js
import myFunction from './my-module.js';

await myFunction();
```

In this example, we're using top-level `await` to wait for the `myFunction` function to complete before continuing execution of the `main.js` module.

It's worth noting that top-level `await` is only available in modules, not in scripts. If you want to use top-level `await`, you need to use an ES module with an `import` statement.

Cyclic imports

Cyclic imports, also known as circular dependencies, occur when two or more modules depend on each other in a way that creates an endless loop. This can cause issues in your JavaScript code and can make it difficult to understand and debug.

To understand cyclic imports, let's start by looking at how modules work in JavaScript. In the CommonJS module system (which is used in Node.js), a module can export values using the `module.exports` object:

```
// moduleA.js
const moduleB = require('./moduleB');

module.exports = {
  a: 1,
  b: moduleB
};

// moduleB.js
const moduleA = require('./moduleA');

module.exports = {
  c: 2,
  d: moduleA
};
```

In this example, `moduleA` requires `moduleB`, and `moduleB` requires `moduleA`. This creates a cyclic dependency between the two modules. When you try to run this code, you'll get an error like this:

```
Error: Cannot find module './moduleA'
```

This error occurs because Node.js tries to load `moduleA` when it is required by `moduleB`, but it has not finished loading `moduleA` yet. This creates an endless loop and causes a stack overflow.

To fix this issue, you need to break the cycle. One way to do this is to restructure your code so that the modules don't depend on each other directly. You can move the common functionality to a separate module and have both modules depend on it:

```
// common.js
module.exports = {
  sharedFunction: function() {}
};

// moduleA.js
const common = require('./common');
const moduleB = require('./moduleB');

module.exports = {
  a: 1,
  b: moduleB,
  sharedFunction: common.sharedFunction
};

// moduleB.js
const common = require('./common');
const moduleA = require('./moduleA');

module.exports = {
  c: 2,
  d: moduleA,
  sharedFunction: common.sharedFunction
};
```

In this example, we've created a separate module common that exports a shared function. Both `moduleA` and `moduleB` require common and use its `sharedFunction`. This way, there is no longer a direct circular dependency between `moduleA` and `moduleB`.

Another way to break the cycle is to use dynamic imports (two lessons back – "Dynamic module loading"). Dynamic imports are using `.then` keyword which allows you to import a module only when you need it, rather than at the top of your code.

You should usually avoid cyclic imports in your project, because they make your code more error-prone. Some common cycle-elimination techniques are:

- Merge the two modules into one.
- Move the shared code into a third module.
- Move some code from one module to the other.

However, cyclic imports can also occur if the libraries depend on each other, which is harder to fix.

Authoring "isomorphic" modules

An isomorphic module is a JavaScript module that can run in both client-side (browser) and server-side (Node.js) environments. This is also known as writing "universal" or "cross-platform" JavaScript code.

To author an isomorphic module, you need to take into account the differences between the browser and Node.js environments. For example, in the browser, you can't use modules that depend on Node.js-specific APIs, like the `fs` (file system) module. Similarly, in Node.js, you can't use modules that depend on browser-specific APIs, like the `window` object.

To overcome these differences, you can use techniques like conditional imports and exports, feature detection, and environment-specific configuration. Here's an example of a simple isomorphic module:

```
// isomorphicModule.js

let isBrowser = false;

if (typeof window !== "undefined") {
  isBrowser = true;
}

let fs;

if (!isBrowser) {
  fs = require("fs");
}

export function readFromFile(path) {
  if (isBrowser) {
    // use browser-specific code to read file
  } else {
    // use Node.js-specific code to read file
    return fs.readFileSync(path, "utf-8");
  }
}

export function writeToConsole(data) {
  if (isBrowser) {
```

```
      console.log(data);
    } else {
      console.error(data);
    }
  }
}
```

In this example, we're checking whether the module is running in a browser or in Node.js by looking for the existence of the `window` object. We're also conditionally importing the `fs` module if we're running in Node.js, since it's not available in the browser.

We're exporting two functions: `readFromFile` and `writeToConsole`. The `readFromFile` function uses environment-specific code to read a file, while `writeToConsole` writes data to the console using console.log in the browser and `console.error` in Node.js.

By using these techniques, you can create modules that work seamlessly in both browser and Node.js environments.

Troubleshooting

Here are a few tips that may help you if you are having trouble getting your modules to work. Feel free to add to the list if you discover more!

- We mentioned this before, but to reiterate: `.mjs` files need to be loaded with a MIME-type of `text/javascript` (or another JavaScript-compatible MIME-type, but `text/javascript` is recommended), otherwise you'll get a strict MIME type checking error like "The server responded with a non-JavaScript MIME type".

- If you try to load the HTML file locally (i.e. with a `file://` URL), you'll run into CORS errors due to JavaScript module security requirements. You need to do your testing through a server. GitHub pages is ideal as it also serves `.mjs` files with the correct MIME type.

- Because `.mjs` is a non-standard file extension, some operating systems might not recognize it, or try to replace it with something else. For example, we found that macOS was silently adding on `.js` to the end of `.mjs` files and then automatically hiding the file extension. So all of our files were actually coming out as `x.mjs.js`. Once we turned off automatically hiding file extensions, and trained it to accept `.mjs`, it was OK.

This lesson was adapted from MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules

this

A function's `this` keyword behaves a little differently in JavaScript compared to other languages. It also has some differences between strict mode and non-strict mode.

In most cases, the value of `this` is determined by how a function is called (runtime binding). It can't be set by assignment during execution, and it may be different each time the function is called. The `bind()` method can set the value of a function's `this` regardless of how it's called, and arrow functions don't provide their own `this` binding (it retains the `this` value of the enclosing lexical context).

```
const test = {
  prop: 42,
  func: function() {
    return this.prop;
  },
};

console.log(test.func());
// Expected output: 42
```

Syntax

```
this
```

Value

In non–strict mode, `this` is always a reference to an object. In strict mode, it can be any value. For more information on how the value is determined, see the description below.

Description

The value of this depends on in which context it appears: function, class, or global.

Function context

When you write a function in your code, it's like a little machine that does some work for you. The function might need to know some information to do its job, so you can give it some parameters to use.

But there's one more special thing that the function has: it has a secret code word called "`this`". The value of "`this`" depends on how you use the function. If you use the function with an object like this: `obj.f()`, then "`this`" means `obj`.

Think of "`this`" like a hidden extra parameter that the function gets without you having to write it down. It helps the function know what object it should be working with.

For example:

```
function getThis() {
  return this;
}

const obj1 = { name: "obj1" };
const obj2 = { name: "obj2" };

obj1.getThis = getThis;
obj2.getThis = getThis;

console.log(obj1.getThis()); // { name: 'obj1', getThis: [Function: getThis] }
console.log(obj2.getThis()); // { name: 'obj2', getThis: [Function: getThis] }
```

Note how the function is the same, but based on how it's invoked, the value of this is different. This is analogous to how function parameters work.

The value of `this` is not the object that has the function as an own property, but the object that is used to call the function. You can prove this by calling a method of an object up in the prototype chain.

```
const obj3 = {
  __proto__: obj1,
  name: "obj3",
};
console.log(obj3.getThis()); // { name: 'obj3' }
```

The value of `this` always changes based on how a function is called, even when the function was defined on an object at creation:

```
const obj4 = {
  name: "obj4",
  getThis() {
    return this;
  },
};

const obj5 = { name: "obj5" };

obj5.getThis = obj4.getThis;
console.log(obj5.getThis()); // { name: 'obj5', getThis: [Function: getThis] }
```

If the value that the method is accessed on is a primitive, `this` will be a primitive value as well — but only if the function is in strict mode.

```
function getThisStrict() {
  "use strict"; // Enter strict mode
  return this;
}

// Only for demonstration — you should not mutate built-in prototypes
Number.prototype.getThisStrict = getThisStrict;
console.log(typeof (1).getThisStrict()); // "number"
```

If the function is called without being accessed on anything, `this` will be `undefined` — but only if the function is in strict mode.

```
console.log(typeof getThisStrict()); // "undefined"
```

In non-strict mode, a special process called this substitution ensures that the value of this is always an object. This means:

- If a function is called with `this` set to undefined or `null`, this gets substituted with `globalThis`.
- If the function is called with `this` set to a primitive value, this gets substituted with the primitive value's wrapper object.

200

```
function getThis() {
  return this;
}

// Only for demonstration — you should not mutate built-in prototypes
Number.prototype.getThis = getThis;
console.log(typeof (1).getThis()); // "object"
console.log(getThis() === globalThis); // true
```

In typical function calls, this is implicitly passed like a parameter through the function's prefix (the part before the dot). You can also explicitly set the value of this using the `Function.prototype.call()`, `Function.prototype.apply()`, or `Reflect.apply()` methods. Using `Function.prototype.bind()`, you can create a new function with a specific value of this that doesn't change regardless of how the function is called. When using these methods, the this substitution rules above still apply if the function is non-strict.

Callbacks

When a function is passed as a callback, the value of `this` depends on how the callback is called, which is determined by the implementor of the API. Callbacks are typically called with a `this` value of `undefined` (calling it directly without attaching it to any object), which means if the function is non–strict, the value of this is the global object (`globalThis`). This is the case for iterative array methods, the `Promise()` constructor, etc.

```
function logThis() {
  "use strict";
  console.log(this);
}

[1, 2, 3].forEach(logThis); // undefined, undefined, undefined
```

Some APIs allow you to set a `this` value for invocations of the callback. For example, all iterative array methods and related ones like `Set.prototype.forEach()` accept an optional `thisArg` parameter.

```
[1, 2, 3].forEach(logThis, { name: "obj" });
// { name: 'obj' }, { name: 'obj' }, { name: 'obj' }
```

Occasionally, a callback is called with a `this` value other than undefined. For example, the reviver parameter of `JSON.parse()` and the `replacer` parameter of `JSON.stringify()` are both called with `this` set to the object that the property being parsed/serialized belongs to.

Arrow functions

In arrow functions, `this` retains the value of the enclosing lexical context's `this`. In other words, when evaluating an arrow function's body, the language does not create a new `this` binding.

For example, in global code, `this` is always `globalThis` regardless of strictness, because of the global context binding:

```
const globalObject = this;
const foo = () => this;
console.log(foo() === globalObject); // true
```

Arrow functions create a closure over the `this` value of its surrounding scope, which means arrow functions behave as if they are "auto-bound" — no matter how it's invoked, `this` is set to what it was when the function was created (in the example above, the global object). The same applies to arrow functions created inside other functions: their `this` remains that of the enclosing lexical context. See example below.

Furthermore, when invoking arrow functions using `call()`, `bind()`, or `apply()`, the `thisArg` parameter is ignored. You can still pass other arguments using these methods, though.

```
const obj = { name: "obj" };

// Attempt to set this using call
console.log(foo.call(obj) === globalObject); // true

// Attempt to set this using bind
const boundFoo = foo.bind(obj);
console.log(boundFoo() === globalObject); // true
```

Constructors

When a function is used as a constructor (with the `new` keyword), its `this` is bound to the new object being constructed, no matter which object the constructor function is accessed on. The value of `this` becomes the value of the `new` expression unless the constructor returns another non–primitive value.

```
function C() {
  this.a = 37;
}

let o = new C();
console.log(o.a); // 37

function C2() {
  this.a = 37;
  return { a: 38 };
}

o = new C2();
console.log(o.a); // 38
```

In the second example (`C2`), because an object was returned during construction, the new object that this was bound to gets discarded. (This essentially makes the statement `this.a = 37;` dead code. It's not exactly dead because it gets executed, but it can be eliminated with no outside effects.)

super

When a function is invoked in the `super.method()` form, the `this` inside the `method` function is the same value as the `this` value around the `super.method()` call, and is generally not equal to the object that `super` refers to. This is because super.method is not an object member access like the ones above — it's a special syntax with different binding rules. For examples, see the `super` reference.

Class context

A class can be split into two contexts: static and instance. Constructors, methods, and instance field initializers (public or private) belong to the instance context. Static methods, static field initializers, and static initialization blocks belong to the static context. The `this` value is different in each context.

Class constructors are always called with `new`, so their behavior is the same as function constructors: the `this` value is the new instance being created. Class methods behave like methods in object literals — the `this` value is the object that the method was accessed on. If the method is not transferred to another object, `this` is generally an instance of the class.

Static methods are not properties of `this`. They are properties of the class itself. Therefore, they are generally accessed on the class, and `this` is the value of the class (or a subclass). Static initialization blocks are also evaluated with `this` set to the current class.

Field initializers are also evaluated in the context of the class. Instance fields are evaluated with `this` set to the instance being constructed. Static fields are evaluated with `this` set to the current class. This is why arrow functions in field initializers are bound to the class.

```
class C {
  instanceField = this;
  static staticField = this;
}

const c = new C();
console.log(c.instanceField === c); // true
console.log(C.staticField === C); // true
```

Derived class constructors

Unlike base class constructors, derived constructors have no initial `this` binding. Calling `super()` creates a `this` binding within the constructor and essentially has the effect of evaluating the following line of code, where `Base` is the base class:

```
this = new Base();
```

Warning: Referring to `this` before calling `super()` will throw an error.

Derived classes must not return before calling `super()`, unless the constructor returns an object (so the `this` value is overridden) or the class has no constructor at all.

```
class Base {}
class Good extends Base {}
class AlsoGood extends Base {
  constructor() {
    return { a: 5 };
  }
}
class Bad extends Base {
  constructor() {}
}
```

```
  new Good();
  new AlsoGood();
  new Bad(); // ReferenceError: Must call super constructor in derived class before
             accessing 'this' or returning from derived constructor
```

Global context

In the global execution context (outside of any functions or classes; may be inside blocks or arrow functions defined in the global scope), the `this` value depends on what execution context the script runs in. Like callbacks, the `this` value is determined by the runtime environment (the caller).

At the top level of a script, `this` refers to `globalThis` whether in strict mode or not. This is generally the same as the global object — for example, if the source is put inside an HTML `<script>` element and executed as a script, `this === window`.

Note: `globalThis` is generally the same concept as the global object (i.e. adding properties to `globalThis` makes them global variables) — this is the case for browsers and Node — but hosts are allowed to provide a different value for `globalThis` that's unrelated to the global object.

```
  // In web browsers, the window object is also the global object:
  console.log(this === window); // true

  this.b = "MDN";
  console.log(window.b); // "MDN"
  console.log(b); // "MDN"
```

If the source is loaded as a module (for HTML, this means adding `type="module"` to the `<script>` tag), `this` is always `undefined` at the top level.

If the source is executed with `eval()`, this is the same as the enclosing context for direct eval, or `globalThis` (as if it's run in a separate global script) for indirect eval.

```
  function test() {
    // Direct eval
    console.log(eval("this") === this);
    // Indirect eval, non-strict
    console.log(eval?.("this") === globalThis);
    // Indirect eval, strict
    console.log(eval?.("'use strict'; this") === globalThis);
  }
```

```
test.call({ name: "obj" }); // Logs 3 "true"
```

Note that some source code, while looking like the global scope, is actually wrapped in a function when executed. For example, Node.js CommonJS modules are wrapped in a function and executed with the `this` value set to `module.exports`. Event handler attributes are executed with `this` set to the element they are attached to.

Object literals don't create a `this` scope — only functions (methods) defined within the object do. Using `this` in an object literal inherits the value from the surrounding scope.

```
const obj = {
  a: this,
};

console.log(obj.a === window); // true
```

Examples

this in function contexts

The value of this depends on how the function is called, not how it's defined.

```
// An object can be passed as the first argument to call
// or apply and this will be bound to it.
const obj = { a: "Custom" };

// Variables declared with var become properties of the global object.
var a = "Global";

function whatsThis() {
  return this.a; // The value of this is dependent on how the function is called
}

whatsThis(); // 'Global'; this in the function isn't set, so it defaults to the
             // global/window object in non-strict mode
obj.whatsThis = whatsThis;
obj.whatsThis(); // 'Custom'; this in the function is set to obj
```

Using `call()` and `apply()`, you can pass the value of this as if it's an actual parameter.

this and object conversion

In non–strict mode, if a function is called with a `this` value that's not an object, the this value is substituted with an object. `null` and `undefined` become `globalThis`. Primitives like 7 or `'foo'` are converted to an object using the related constructor, so the primitive number 7 is converted to a `Number` wrapper class and the string `'foo'` to a `String` wrapper class.

```
function bar() {
  console.log(Object.prototype.toString.call(this));
}

bar.call(7); // [object Number]
bar.call("foo"); // [object String]
bar.call(undefined); // [object Window]
```

The bind() method

Calling `f.bind(someObject)` creates a new function with the same body and scope as `f`, but the value of `this` is permanently bound to the first argument of `bind`, regardless of how the function is being called.

```
function f() {
  return this.a;
}

const g = f.bind({ a: "azerty" });
console.log(g()); // azerty

const h = g.bind({ a: "yoo" }); // bind only works once!
console.log(h()); // azerty

const o = { a: 37, f, g, h };
console.log(o.a, o.f(), o.g(), o.h()); // 37,37, azerty, azerty
```

this in arrow functions

Arrow functions create closures over the `this` value of the enclosing execution context. In the following example, we create `obj` with a method `getThisGetter` that returns a function that

returns the value of `this`. The returned function is created as an arrow function, so its `this` is permanently bound to the `this` of its enclosing function. The value of this inside `getThisGetter` can be set in the call, which in turn sets the return value of the returned function.

```
const obj = {
  getThisGetter() {
    const getter = () => this;
    return getter;
  },
};
```

We can call `getThisGetter` as a method of `obj`, which sets this inside the body to `obj`. The returned function is assigned to a variable `fn`. Now, when calling `fn`, the value of this returned is still the one set by the call to `getThisGetter`, which is obj. If the returned function is not an arrow function, such calls would cause the `this` value to be `globalThis` or undefined in strict mode.

```
const fn = obj.getThisGetter();
console.log(fn() === obj); // true
```

But be careful if you unbind the method of `obj` without calling it, because `getThisGetter` is still a method that has a varying this value. Calling `fn2()()` in the following example returns `globalThis`, because it follows the this from `fn2`, which is `globalThis` since it's called without being attached to any object.

```
const fn2 = obj.getThisGetter;
console.log(fn2()() === globalThis); // true
```

This behavior is very useful when defining callbacks. Usually, each function expression creates its own `this` binding, which shadows the `this` value of the upper scope. Now, you can define functions as arrow functions if you don't care about the `this` value, and only create this bindings where you do (e.g. in class methods). See example with `setTimeout()`.

this with a getter or setter

`this` in getters and setters is based on which object the property is accessed on, not which object the property is defined on. A function used as getter or setter has its `this` bound to the object from which the property is being set or gotten.

```
function sum() {
  return this.a + this.b + this.c;
}

 const o = {
  a: 1,
  b: 2,
  c: 3,
  get average() {
    return (this.a + this.b + this.c) / 3;
  },
};

Object.defineProperty(o, "sum", {
  get: sum,
  enumerable: true,
  configurable: true,
});

console.log(o.average, o.sum); // 2, 6
```

As a DOM event handler

When a function is used as an event handler, its `this` is set to the element on which the listener is placed (some browsers do not follow this convention for listeners added dynamically with methods other than `addEventListener()`).

```
// When called as a listener, turns the related element blue
function bluify(e) {
  // Always true
  console.log(this === e.currentTarget);
  // true when currentTarget and target are the same object
  console.log(this === e.target);
  this.style.backgroundColor = "#A5D9F3";
}

// Get a list of every element in the document
const elements = document.getElementsByTagName("*");

// Add bluify as a click listener so when the element is clicked on, it turns blue
for (const element of elements) {
  element.addEventListener("click", bluify, false);
}
```

this in inline event handlers

When the code is called from an inline event handler attribute, its `this` is set to the DOM element on which the listener is placed:

```
<button onclick="alert(this.tagName.toLowerCase());">Show this</button>
```

The above alert shows button. Note, however, that only the outer code has its `this` set this way:

```
<button onclick="alert((function () { return this; })());">
  Show inner this
</button>
```

In this case, the inner function's `this` isn't set, so it returns the global/window object (i.e. the default object in non–strict mode where `this` isn't set by the call).

Bound methods in classes

Just like with regular functions, the value of `this` within methods depends on how they are called. Sometimes it is useful to override this behavior so that `this` within classes always refers to the class instance. To achieve this, bind the class methods in the constructor:

```
class Car {
  constructor() {
    // Bind sayBye but not sayHi to show the difference
    this.sayBye = this.sayBye.bind(this);
  }

sayHi() {
    console.log(`Hello from ${this.name}`);
  }
  sayBye() {
    console.log(`Bye from ${this.name}`);
  }
  get name() {
    return "Ferrari";
  }
}
```

```
class Bird {
  get name() {
    return "Tweety";
  }
}
const car = new Car();
const bird = new Bird();

// The value of 'this' in methods depends on their caller
car.sayHi(); // Hello from Ferrari
bird.sayHi = car.sayHi;
bird.sayHi(); // Hello from Tweety

// For bound methods, 'this' doesn't depend on the caller
bird.sayBye = car.sayBye;
bird.sayBye(); // Bye from Ferrari
```

Note: Classes are always in strict mode. Calling methods with an undefined `this` will throw an error if the method tries to access properties on `this`.

Note, however, that auto-bound methods suffer from the same problem as using arrow functions for class properties: each instance of the class will have its own copy of the method, which increases memory usage. Only use it where absolutely necessary. You can also mimic the implementation of `Intl.NumberFormat.prototype.format()`: define the property as a getter that returns a bound function when accessed and saves it, so that the function is only created once and only created when necessary.

this in with statements

Although with statements are deprecated and not available in strict mode, they still serve as an exception to the normal this binding rules. If a function is called within a with statement and that function is a property of the scope object, the `this` value is set to the scope object, as if the `obj1.` prefix exists.

```
const obj1 = {
  foo() {
    return this;
  },
};
with (obj1) {
  console.log(foo() === obj1); // true
}
```

This lesson was adapted from MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this

Client-side web APIs

When writing client-side JavaScript for websites or applications, you will quickly encounter Application Programming Interfaces (APIs). APIs are programming features for manipulating different aspects of the browser and operating system the site is running on, or manipulating data from other websites or services. In this module, we will explore what APIs are, and how to use some of the most common APIs you'll come across often in your development work.

Introduction to web APIs

First up, we'll start by looking at APIs from a high level — what are they, how do they work, how to use them in your code, and how are they structured? We'll also take a look at what the different main classes of APIs are, and what kind of uses they have.

What are APIs?

Application Programming Interfaces (APIs) are constructs made available in programming languages to allow developers to create complex functionality more easily. They abstract more complex code away from you, providing some easier syntax to use in its place.

As a real-world example, think about the electricity supply in your house. If you want to use an appliance in your house, you plug it into a plug socket and it works. You don't try to wire it directly into the power supply — to do so would be really inefficient and, if you are not an electrician, difficult and dangerous to attempt.

In the same way, if you want to say, program some 3D graphics, it is a lot easier to do it using an API written in a higher-level language such as JavaScript or Python, rather than try to directly write low-level code (say C or C++) that directly controls the computer's GPU or other graphics functions.

APIs in client-side JavaScript

Client-side JavaScript, in particular, has many APIs available to it — these are not part of the JavaScript language itself, rather they are built on top of the core JavaScript language, providing you with extra superpowers to use in your JavaScript code. They generally fall into two categories:

- **Browser APIs** are built into your web browser and are able to expose data from the browser and surrounding computer environment and do useful complex things with it. For example, the Web Audio API provides JavaScript constructs for manipulating audio in the browser — taking an audio track, altering its volume, applying effects to it, etc. In the background, the browser is actually using some complex lower-level code (e.g. C++ or Rust) to do the actual audio processing. But again, this complexity is abstracted away from you by the API.

- **Third-party APIs** are not built into the browser by default, and you generally have to retrieve their code and information from somewhere on the Web. For example, the Twitter API allows you to do things like displaying your latest tweets on your website. It provides a special set of constructs you can use to query the Twitter service and return specific information.

Relationship between JavaScript, APIs, and other JavaScript tools

So above, we talked about what client-side JavaScript APIs are, and how they relate to the JavaScript language. Let's recap this to make it clearer, and also mention where other JavaScript tools fit in:

- JavaScript — A high-level scripting language built into browsers that allow you to implement functionality on web pages/apps. Note that JavaScript is also available in other programming environments, such as Node.
- Browser APIs — constructs built into the browser that sits on top of the JavaScript language and allows you to implement functionality more easily.
- Third-party APIs — constructs built into third-party platforms (e.g. Twitter, Facebook) that allow you to use some of those platform's functionality in your own web pages (for example, display your latest Tweets on your web page).
- JavaScript libraries — Usually one or more JavaScript files containing custom functions that you can attach to your web page to speed up or enable writing common functionality. Examples include jQuery, Mootools, and React.
- JavaScript frameworks — The next step up from libraries, JavaScript frameworks (e.g. Angular and Ember) tend to be packages of HTML, CSS, JavaScript, and other technologies that you install and then use to write an entire web application from scratch. **The key difference between a library and a framework is "Inversion of Control". When calling a method from a library, the developer is in control. With a framework, the control is inverted: the framework calls the developer's code**.

What can APIs do?

There are a huge number of APIs available in modern browsers that allow you to do a wide variety of things in your code.

Common browser APIs

In particular, the most common categories of browser APIs you'll use (and which we'll cover in this module in greater detail) are:

- **APIs for manipulating documents** loaded into the browser. The most obvious example is the DOM (Document Object Model) API, which allows you to manipulate HTML and CSS.
- **APIs that fetch data from the server** to update parts of a webpage without reloading the entire page. The main API used for this is the Fetch API, although older code might still use the XMLHttpRequest API. You may also come across the term **Ajax**, which describes this technique.
- **APIs for drawing and manipulating graphics** are widely supported in browsers — the most popular ones are Canvas and WebGL, which allow you to programmatically update the pixel data contained in an HTML `<canvas>` element to create 2D and 3D scenes.
- **Audio and Video APIs** allow manipulation and control of multimedia elements like audio and video, including adding effects and custom UI controls.
- **Device APIs** enable you to interact with device hardware: for example, accessing the device GPS to find the user's position using the Geolocation API.
- **Client-side storage APIs** allow storing data on the client-side, enabling offline functionality and state persistence between page loads.

Common third-party APIs

Third-party APIs come in a large variety, some of the more popular ones that you are likely to make use of sooner or later are:

- The Twitter API, which allows you to do things like displaying your latest tweets on your website.
- Map APIs, like Mapquest and the Google Maps API, which allow you to do all sorts of things with maps on your web pages.
- The Facebook suite of APIs, which enables you to use various parts of the Facebook ecosystem to benefit your app, such as by providing app login using Facebook login, accepting in-app payments, rolling out targeted ad campaigns, etc.
- The YouTube API, which allows you to embed YouTube videos on your site, search YouTube, build playlists, and more.

- The Twilio API, which provides a framework for building voice and video call functionality into your app, sending SMS/MMS from your apps, and more.

How do APIs work?

Different JavaScript APIs work in slightly different ways, but generally, they have common features and similar themes to how they work.

They are based on objects

Your code interacts with APIs using one or more JavaScript objects, which serve as containers for the data the API uses (contained in object properties), and the functionality the API makes available (contained in object methods).

Let's return to the example of the Web Audio API — this is a fairly complex API, which consists of a number of objects. The most obvious ones are:

- `AudioContext`, which represents an audio graph that can be used to manipulate audio playing inside the browser, and has a number of methods and properties available to manipulate that audio.
- `MediaElementAudioSourceNode`, which represents an `<audio>` element containing sound you want to play and manipulate inside the audio context.
- `AudioDestinationNode`, which represents the destination of the audio, i.e. the device on your computer that will actually output it — usually your speakers or headphones.

So how do these objects interact? In this web audio example, we have an HTML page with an audio player. Here's the HTML code:

```
<audio src="outfoxing.mp3"></audio>

<button class="paused">Play</button>
<br />
<input type="range" min="0" max="1" step="0.01" value="1" class="volume" />
```

The JavaScript code does the following:

1. It creates an `AudioContext` to manipulate the audio track:

```
const AudioContext = window.AudioContext || window.webkitAudioContext;
const audioCtx = new AudioContext();
```

2. It selects the audio, play button, and volume slider elements, and creates a MediaElementAudioSourceNode representing the audio source:

```
const audioElement = document.querySelector("audio");
const playBtn = document.querySelector("button");
const volumeSlider = document.querySelector(".volume");

const audioSource = audioCtx.createMediaElementSource(audioElement);
```

3. It sets up event handlers to toggle play/pause and reset the display when the audio finishes:

```
playBtn.addEventListener("click", () => {
  // Check if the audio context is in the suspended state (autoplay policy)
  if (audioCtx.state === "suspended") {
    audioCtx.resume();
  }

  // Toggle play/pause
  if (playBtn.getAttribute("class") === "paused") {
    audioElement.play();
    playBtn.setAttribute("class", "playing");
    playBtn.textContent = "Pause";
  } else if (playBtn.getAttribute("class") === "playing") {
    audioElement.pause();
    playBtn.setAttribute("class", "paused");
    playBtn.textContent = "Play";
  }
});

audioElement.addEventListener("ended", () => {
  // Reset display when audio finishes
  playBtn.setAttribute("class", "paused");
  playBtn.textContent = "Play";
});
```

4. It creates a GainNode to adjust the volume and sets up an event listener for the volume slider:

```
const gainNode = audioCtx.createGain();

volumeSlider.addEventListener("input", () => {
  // Adjust the audio graph's gain (volume) based on the slider value
  gainNode.gain.value = volumeSlider.value;
});
```

5. Finally, it connects the audio nodes together:

```
audioSource.connect(gainNode).connect(audioCtx.destination);
```

This code sets up an audio player, handles play/pause functionality, adjusts the volume, and connects the audio elements together to control playback and volume using the Web Audio API.

They have recognizable entry points

When using an API, you should make sure you know where the entry point is for the API. In The Web Audio API, this is pretty simple — it is the `AudioContext` object, which needs to be used to do any audio manipulation whatsoever.

The Document Object Model (DOM) API also has a simple entry point — its features tend to be found hanging off the `Document` object, or an instance of an HTML element that you want to affect in some way, for example:

```
const em = document.createElement("em"); // create a new em element
const para = document.querySelector("p"); // reference an existing p element
em.textContent = "Hello there!"; // give em some text content
para.appendChild(em); // embed em inside para
```

The Canvas API also relies on getting a context object to use to manipulate things, although in this case, it's a graphical context rather than an audio context. Its context object is created by getting a reference to the `<canvas>` element you want to draw on, and then calling its `HTMLCanvasElement.getContext()` method:

```
const canvas = document.querySelector("canvas");
const ctx = canvas.getContext("2d");
```

Anything that we want to do to the canvas is then achieved by calling properties and methods of the context object (which is an instance of CanvasRenderingContext2D), for example:

```
Ball.prototype.draw = function () {
  ctx.beginPath();
  ctx.fillStyle = this.color;
  ctx.arc(this.x, this.y, this.size, 0, 2 * Math.PI);
  ctx.fill();
};
```

**They often use events to handle changes in state**

Some web APIs contain no events, but most contain at least a few. We already saw a number of event handlers in use in our Web Audio API example above:

```
// play/pause audio
playBtn.addEventListener("click", () => {
  // check if context is in suspended state (autoplay policy)
  if (audioCtx.state === "suspended") {
    audioCtx.resume();
  }
  // if track is stopped, play it
  if (playBtn.getAttribute("class") === "paused") {
    audioElement.play();
    playBtn.setAttribute("class", "playing");
    playBtn.textContent = "Pause";
    // if track is playing, stop it
  } else if (playBtn.getAttribute("class") === "playing") {
    audioElement.pause();
    playBtn.setAttribute("class", "paused");
    playBtn.textContent = "Play";
  }
});
// if track ends
audioElement.addEventListener("ended", () => {
  playBtn.setAttribute("class", "paused");
  playBtn.textContent = "Play";
});
```

They have additional security mechanisms where appropriate

WebAPI features are subject to the same security considerations as JavaScript and other web technologies (for example same-origin policy), but they sometimes have additional security mechanisms in place. For example, some of the more modern WebAPIs will only work on pages served over HTTPS due to them transmitting potentially sensitive data (examples include Service Workers and Push).

In addition, some WebAPIs request permission to be enabled from the user once calls to them are made in your code. As an example, the Notifications API asks for permission using a pop-up dialog box:
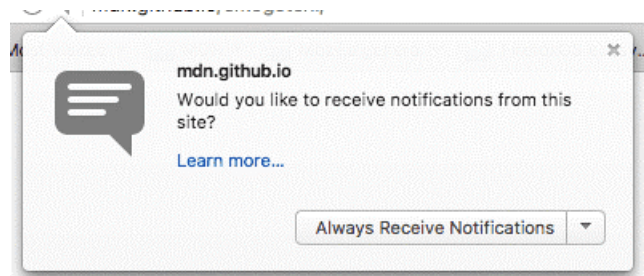
The Web Audio and `HTMLMediaElement` APIs are subject to a security mechanism called autoplay policy — this basically means that you can't automatically play audio when a page loads — you've got to allow your users to initiate audio play through a control like a button. This is done because autoplaying audio is usually really annoying and we really shouldn't be subjecting our users to it.

Manipulating documents

When writing web pages and apps, one of the most common things you'll want to do is manipulate the document structure in some way. This is usually done by using the Document Object Model (DOM), a set of APIs for controlling HTML and styling information that makes heavy use of the `Document` object.

The important parts of a web browser

Web browsers are very complicated pieces of software with a lot of moving parts, many of which can't be controlled or manipulated by a web developer using JavaScript.

Despite the limitations, Web APIs still give us access to a lot of functionality that enables us to do a great many things with web pages. There are a few really obvious bits you'll reference regularly in your code — consider the following diagram, which represents the main parts of a browser directly involved in viewing web pages:

- The window is the browser tab that a web page is loaded into; this is represented in JavaScript by the `Window` object. Using methods available on this object you can do things like return the window's size (see `Window.innerWidth` and `Window.innerHeight`), manipulate the document loaded into that window, store data specific to that document on the client-side, and more.
- The navigator represents the state and identity of the browser (i.e. the user-agent) as it exists on the web. In JavaScript, this is represented by the `Navigator` object. You can use this object to retrieve things like the user's preferred language, a media stream from the user's webcam, etc.
- The document (represented by the DOM in browsers) is the actual page loaded into the window, and is represented in JavaScript by the `Document` object. You can use this object to return and manipulate information on the HTML and CSS that comprises the document, for example get a reference to an element in the DOM, change its text content, apply new styles to it, create new elements and add them to the current element as children, or even delete it altogether.
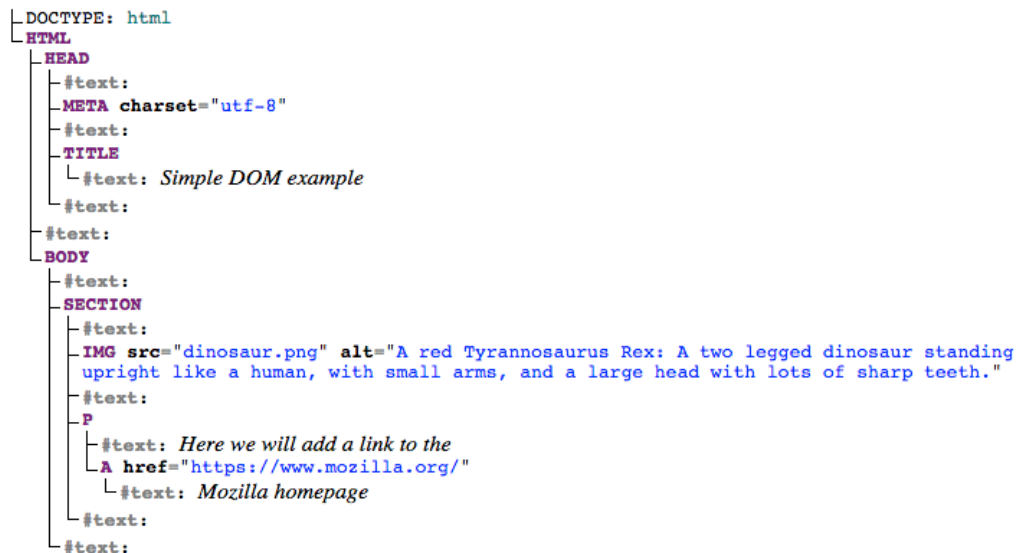
The document object model

The document currently loaded in each one of your browser tabs is represented by a document object model. This is a "tree structure" representation created by the browser that enables the HTML structure to be easily accessed by programming languages.

We have created a simple example page, the HTML source code looks like this:

```html
<!DOCTYPE html>
<html lang="en-US">
  <head>
    <meta charset="utf-8" />
    <title>Simple DOM example</title>
  </head>
  <body>
    <section>
      <img src="dinosaur.png" alt="Two-legged dinosaur standing upright."/>
      <p>
        Here we will add a link to the
        <a href="https://www.mozilla.org/">Mozilla homepage</a>
      </p>
    </section>
  </body>
</html>
```

The DOM on the other hand looks like this:

```
 DOCTYPE: html
 HTML
   HEAD
     #text:
     META charset="utf-8"
     #text:
     TITLE
       #text: Simple DOM example
     #text:
   #text:
   BODY
     #text:
     SECTION
       #text:
       IMG src="dinosaur.png" alt="A red Tyrannosaurus Rex: A two legged dinosaur standing
       upright like a human, with small arms, and a large head with lots of sharp teeth."
       #text:
       P
         #text: Here we will add a link to the
         A href="https://www.mozilla.org/"
           #text: Mozilla homepage
       #text:
     #text:
```

Each entry in the tree is called a **node**. You can see in the diagram above that some nodes represent elements (identified as HTML, HEAD, META and so on) and others represent text (identified as #text). There are other types of nodes as well, but these are the main ones you'll encounter.

Nodes are also referred to by their position in the tree relative to other nodes:

221

- **Root node**: The top node in the tree, which in the case of HTML is always the HTML node (other markup vocabularies like SVG and custom XML will have different root elements).
- **Child node**: A node *directly* inside another node. For example, IMG is a child of SECTION in the above example.
- **Descendant node**: A node *anywhere* inside another node. For example, IMG is a child of SECTION in the above example, and it is also a descendant. IMG is not a child of BODY, as it is two levels below it in the tree, but it is a descendant of BODY.
- **Parent node**: A node which has another node inside it. For example, BODY is the parent node of SECTION in the above example.
- **Sibling nodes**: Nodes that sit on the same level in the DOM tree. For example, IMG and P are siblings in the above example.

Basic DOM manipulation

To manipulate an element inside the DOM, you first need to select it and store a reference to it inside a variable. Inside your script element, add the following line:

```
const link = document.querySelector("a");
```

Now we have the element reference stored in a variable, we can start to manipulate it using properties and methods available to it. These properties and methods are defined on interfaces of the DOM specification, which defines the structure and behavior of HTML documents. The DOM API provides a way to interact with and manipulate HTML elements using JavaScript.

First of all, let's change the text inside the link by updating the value of the Node.textContent property. Add the following line below the previous one:

```
link.textContent = "Mozilla Developer Network";
```

We should also change the URL the link is pointing to, so that it doesn't go to the wrong place when it is clicked on. Add the following line, again at the bottom:

```
link.href = "https://developer.mozilla.org";
```

Note that, as with many things in JavaScript, there are many ways to select an element and store a reference to it in a variable. `Document.querySelector()` is the recommended modern approach. It is convenient because it allows you to select elements using CSS selectors. The above `querySelector()` call will match the first `<a>` element that appears in the document. If you wanted to match and do things to multiple elements, you could use `Document.querySelectorAll()`, which matches every element in the document that matches the selector, and stores references to them in an array-like object called a `NodeList`.

There are older methods available for grabbing element references, such as:

- `Document.getElementById()`, which selects an element with a given `id` attribute value, e.g. `<p id="myId">My paragraph</p>`. The ID is passed to the function as a parameter, i.e. `const elementRef = document.getElementById('myId')`.

- `Document.getElementsByTagName()`, which returns an array-like object containing all the elements on the page of a given type, for example `<p>`s, `<a>`s, etc. The element type is passed to the function as a parameter, i.e. `const elementRefArray = document.getElementsByTagName('p')`.

These two work better in older browsers than the modern methods like `querySelector()`, but are not as convenient. Have a look and see what others you can find!

Creating and placing new nodes

Let's go further and look at how we can create new elements.

1. Going back to the current example, let's start by grabbing a reference to our `<section>` element — add the following code at the bottom of your existing script:

```
const sect = document.querySelector("section");
```

2. Now let's create a new paragraph using `Document.createElement()` and give it some text content in the same way as before:

```
const para = document.createElement("p");
para.textContent = "We hope you enjoyed the ride.";
```

3. You can now append the new paragraph at the end of the section using `Node.appendChild()`:

223

```
sect.appendChild(para);
```

4. Finally for this part, let's add a text node to the paragraph the link sits inside, to round off the sentence nicely. First we will create the text node using `Document.createTextNode()`:

```
const text = document.createTextNode(
 " — the premier source for web development knowledge.",
);
```

5. Now we'll grab a reference to the paragraph the link is inside, and append the text node to it:

```
const linkPara = document.querySelector("p");
linkPara.appendChild(text);
```

That's most of what you need for adding nodes to the DOM — you'll make a lot of use of these methods when building dynamic interfaces (we'll look at some examples later).

Moving and removing elements

There may be times when you want to move nodes or delete them from the DOM altogether. This is perfectly possible.

If we wanted to move the paragraph with the link inside it to the bottom of the section, we could do this:

```
sect.appendChild(linkPara);
```

If you wanted to make a copy and add that as well, you'd need to use `Node.cloneNode()` instead.

Removing a node is pretty simple as well, at least when you have a reference to the node to be removed and its parent. In our current case, we just use `Node.removeChild()`, like this:

```
sect.removeChild(linkPara);
```

When you want to remove a node based only on a reference to itself, which is fairly common, you can use `Element.remove()`:

```
linkPara.remove();
```

This method is not supported in older browsers. They have no method to tell a node to remove itself, so you'd have to do the following:

```
linkPara.parentNode.removeChild(linkPara);
```

Manipulating styles

It is possible to manipulate CSS styles via JavaScript in a variety of ways.

To start with, you can get a list of all the stylesheets attached to a document using `Document.stylesheets`, which returns an array-like object with `CSSStyleSheet` objects. You can then add/remove styles as wished. However, we're not going to expand on those features because they are a somewhat archaic and difficult way to manipulate style. There are much easier ways.

The first way is to add inline styles directly onto elements you want to **dynamically style**. This is done with the `HTMLElement.style` property, which contains inline styling information for each element in the document. You can set properties of this object to directly update element styles.

```
para.style.color = "white";
para.style.backgroundColor = "black";
para.style.padding = "10px";
para.style.width = "250px";
para.style.textAlign = "center";
```

After page reload if you look at that paragraph in your browser's Page Inspector/DOM inspector, you'll see that these lines are indeed adding inline styles to the document:

```
<p
  style="color: white; background-color: black; padding: 10px;
  width: 250px;     text-align: center;">
  We hope you enjoyed the ride.
</p>
```

There is another common way to dynamically manipulate styles on your document, which we'll look at now.

Delete the previous five lines you added to the JavaScript, and add the following inside your HTML `<head>`:

```
<style>
  .highlight {
    color: white;
    background-color: black;
    padding: 10px;
    width: 250px;
    text-align: center;
  }
</style>
```

Now we'll turn to a very useful method for general HTML manipulation – `Element.setAttribute()` — this takes two arguments, the attribute you want to set on the element, and the value you want to set it to. In this case, we will set a class name to highlight on our paragraph:

```
para.setAttribute("class", "highlight");
```

Refresh your page, and you'll see no change — the CSS is still applied to the paragraph, but this time by giving it a class that is selected by our CSS rule, not as inline CSS styles.

Which method you choose is up to you; both have their advantages and disadvantages. The first method takes less setup and is good for simple uses, whereas the second method is more purist (no mixing CSS and JavaScript, no inline styles, which are seen as a bad practice). As you start building larger and more involved apps, you will probably start using the second method more, but it is really up to you.

Fetching data from the server

Another very common task in modern websites and applications is retrieving individual data items from the server to update sections of a webpage without having to load an entire new page. This seemingly small detail has had a huge impact on the performance and behavior of sites, so in this

article, we'll explain the concept and look at technologies that make it possible: in particular, the **Fetch API**.

What is the problem here?

A web page consists of an HTML page and (usually) various other files, such as stylesheets, scripts, and images. The basic model of page loading on the Web is that your browser makes one or more HTTP requests to the server for the files needed to display the page, and the server responds with the requested files. If you visit another page, the browser requests the new files, and the server responds with them.
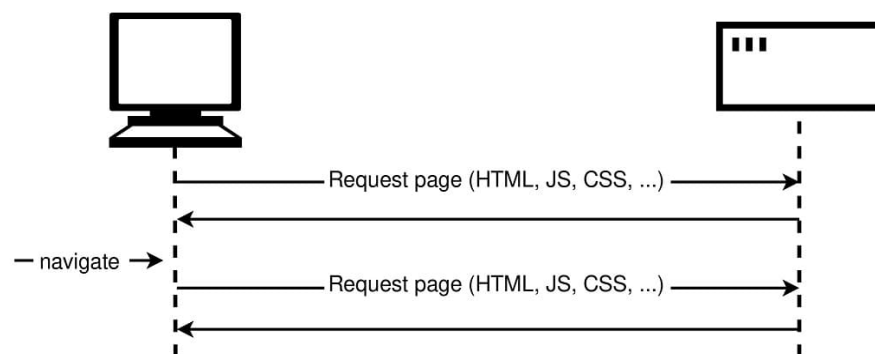
This model works perfectly well for many sites. But consider a website that's very data-driven.

The trouble with the traditional model here is that we'd **have to fetch and load the entire page**, even when we only need to update one part of it. This is inefficient and can result in a poor user experience.

So instead of the traditional model, many websites use JavaScript APIs to request data from the server and **update the page content without a page load**. So, for example when the user searches for a new product, the browser only requests the data which is needed to update the page — the set of new books to display, for instance.
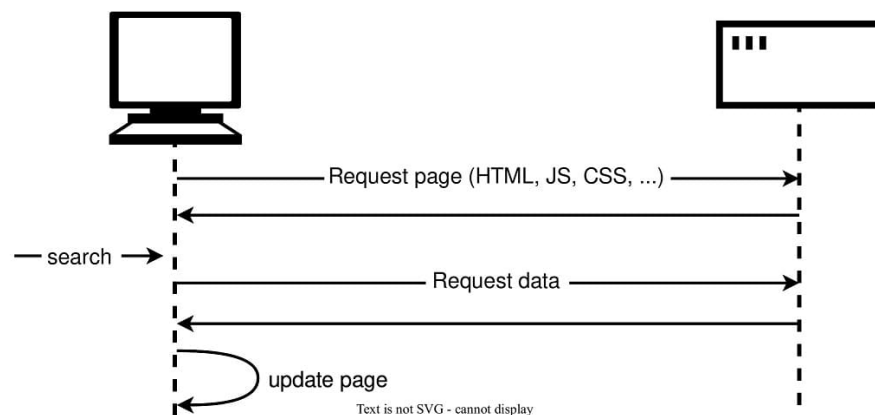
The main API here is the Fetch API. This enables JavaScript running on a page to make an HTTP request to a server to retrieve specific resources. When the server provides them, the JavaScript can use the data to update the page, typically by using DOM manipulation APIs. The data requested is often JSON, which is a good format for transferring structured data, but can also be HTML or just text.

This is a common pattern for data-driven sites such as Amazon, YouTube, eBay, and so on. With this model:

- Page updates are a lot quicker and you don't have to wait for the page to refresh, meaning that the site feels faster and more responsive.
- Less data is downloaded on each update, meaning less wasted bandwidth. This may not be such a big issue on a desktop on a broadband connection, but it's a major issue on mobile devices and in countries that don't have ubiquitous fast internet service.

Note: In the early days, this general technique was known as Asynchronous JavaScript and XML (Ajax), because it tended to request XML data. This is normally not the case these days (you'd be more likely to request JSON), but the result is still the same, and the term "Ajax" is still often used to describe the technique.

To speed things up even further, some sites also store assets and data on the user's computer when they are first requested, meaning that on subsequent visits they use the local versions instead of downloading fresh copies every time the page is first loaded. The content is only reloaded from the server when it has been updated. This technique is known as Caching or Client-Side Caching.

The Fetch API

The Fetch API is a modern web standard that provides a way to make network requests (e.g., HTTP requests) from a web browser or a JavaScript application. It offers a more powerful and flexible alternative to the older XMLHttpRequest (XHR) object for handling asynchronous data retrieval. The API revolves around concepts such as Request and Response objects, CORS (Cross-Origin Resource Sharing), and the HTTP Origin header semantics.

The Fetch API is incorporated into all modern web browsers, making it available for use in web development across different platforms. The implementation of the Fetch API within browsers is typically done in lower-level languages like C++

With the Fetch API, you can initiate HTTP requests to fetch resources such as JSON data, HTML pages, images, or other types of files from a server. It supports various HTTP methods, including GET, POST, PUT, DELETE, and more.

The Fetch API uses Request and Response objects (and other things involved with network requests), as well as related concepts such as CORS and the HTTP Origin header semantics.

For making a request and fetching a resource, use the `fetch()` method. It is a global method in both `Window` and `Worker` contexts. This makes it available in pretty much any context you might want to fetch resources in.

The `fetch()` method takes one mandatory argument, the path to the resource you want to fetch. It returns a Promise that resolves to the Response to that request — as soon as the server responds with headers — even if the server response is an HTTP error status. You can also optionally pass in an `init` options object as the second argument.

Once a Response is retrieved, there are a number of methods available to define what the body content is and how it should be handled.

Here's a basic example of how to use the Fetch API to make a GET request and handle the response:

```
fetch('https://api.example.com/data') // Initiating GET request to the specified URL
    .then(response => { // Promise resolves to the Response object
        if (!response.ok) { // Check if the response was successful (status code 200-299)
            throw new Error('Network response fail'); // Throw error if response failed
        }
        return response.json(); // Parsing the response body as JSON and returning another Promise
    })
    .then(data => { // Promise resolves to the parsed JSON data
        // Process the retrieved data
        console.log(data); // Logging the retrieved data to the console
    })
    .catch(error => { // Handling any errors that occurred during the request or parsing
        // Handle any errors that occurred during the request
        console.error('Error:', error); // Logging the error message to the console
    });
```

In this example, `fetch()` is called with the URL of the resource you want to fetch. The fetch() function returns a Promise that resolves to a Response object representing the server's response. The Promise is resolved asynchronously when the response is received from the server.

You can then chain `.then()` methods to handle the response. In the first `.then()`, we check if the response was successful (`response.ok`). If not, an error is thrown. Otherwise, we call `response.json()` to parse the response body as JSON and return another Promise.

The second `.then()` receives the parsed JSON data as the argument, and you can process it as needed.

If any errors occur during the request, they can be caught and handled in the `.catch()` block.

The Fetch API also allows you to set request headers, pass data in the request body, handle different types of responses (e.g., text, blob, array buffer), and more. It provides a flexible and modern way to handle network requests in JavaScript applications.

The Fetch API includes various interfaces that you can utilize:

- **fetch()**: The method used to fetch a resource.
- **Headers**: Represents response/request headers, allowing you to query them and perform actions based on the results.
- **Request**: Represents a resource request.
- **Response**: Represents the response to a request.

By leveraging these interfaces and methods, you can efficiently handle network requests, retrieve resources from servers, and process the received data in JavaScript applications.

The XMLHttpRequest API

The `XMLHttpRequest` API is a built-in browser API that allows you to make HTTP requests from JavaScript in a browser. It provides a way to communicate with servers and retrieve data without reloading the entire web page.

Here's an example of how to use the `XMLHttpRequest` API to make a GET request and handle the response:

```
// Create a new XMLHttpRequest object
const xhr = new XMLHttpRequest();
```

```
  // Configure the request: GET method, URL
  xhr.open('GET', 'https://api.example.com/data', true);

  // Set up a callback function to handle the response
  xhr.onload = function() {
    // Check if the request was successful (status code 200)
    if (xhr.status === 200) {
      // Process the response data
      const responseData = JSON.parse(xhr.responseText);
      console.log(responseData);
    } else {
      // Handle the error or non-200 status code
      console.error('Request failed. Status:', xhr.status);
    }
  };

  // Send the request
  xhr.send();
```

In this example:

1. We create a new `XMLHttpRequest` object using the `new XMLHttpRequest()` constructor.
2. We configure the request using the `open()` method. Here, we specify the HTTP method (GET in this case), the URL of the resource we want to retrieve (`https://api.example.com/data`), and `true` to indicate that the request should be asynchronous.
3. We set up an `onload` event handler to handle the response when it's received. Inside the callback function, we check the status code (`xhr.status`) to see if the request was successful (status code 200). If it was, we process the response data by parsing it as JSON and logging it to the console. If the status code is not 200, we handle the error or non-200 status code.
4. Finally, we send the request to the server using the `send()` method.

The `XMLHttpRequest` API provides various methods and properties to handle different types of requests, set headers, handle progress events, and more. However, it has some limitations and complexities, which led to the development of newer APIs like the Fetch API that provide a more modern and flexible approach to making network requests in JavaScript applications.

The `XMLHttpRequest` API is commonly used to implement AJAX (Asynchronous JavaScript and XML) functionality in web applications. It allows you to make asynchronous HTTP requests from JavaScript, enabling dynamic updates to web content without requiring a full page reload. AJAX, using the `XMLHttpRequest` API, enhances interactivity and responsiveness of web applications, enabling features like auto-complete search, real-time updates, and data fetching without page

reloads by allowing data exchange between the client and server without interrupting the user's browsing experience.

It's important to note that the term "AJAX" is often used more broadly to refer to the concept of asynchronous data exchange between the client and server, even if alternative techniques or APIs are used instead of the `XMLHttpRequest` API. For example, newer APIs like the Fetch API and the use of libraries like Axios are also commonly used for implementing AJAX functionality in modern web applications.

Third-party APIs

The APIs we've covered so far are built into the browser, but not all APIs are. Many large websites and services such as Google Maps, Twitter, Facebook, PayPal, etc. provide APIs allowing developers to make use of their data (e.g. displaying your twitter stream on your blog) or services (e.g. using Facebook login to log in your users). This article looks at the difference between browser APIs and 3rd party APIs and shows some typical uses of the latter.

What are third party APIs?

Third party APIs are APIs provided by third parties — generally companies such as Facebook, Twitter, or Google — to allow you to access their functionality via JavaScript and use it on your site. One of the most obvious examples is using mapping APIs to display custom maps on your pages.

Let's look at a <u>Simple Mapquest API example</u>, and use it to illustrate how third-party APIs differ from browser APIs.

They are found on third-party servers

Browser APIs are built into the browser — you can access them from JavaScript immediately. For example, the Web Audio API we saw in the Introductory article is accessed using the native `AudioContext` object. For example:

```
const audioCtx = new AudioContext();
// …
const audioElement = document.querySelector("audio");
// …
const audioSource = audioCtx.createMediaElementSource(audioElement);
```

Third party APIs, on the other hand, are located on third party servers. To access them from JavaScript you first need to connect to the API functionality and make it available on your page. This typically involves first linking to a JavaScript library available on the server via a `<script>` element, as seen in our Mapquest example:

```
<script
  src="https://api.mqcdn.com/sdk/mapquest-js/v1.3.2/mapquest.js"
  defer></script>
<link
  rel="stylesheet"
  href="https://api.mqcdn.com/sdk/mapquest-js/v1.3.2/mapquest.css" />
```

You can then start using the objects available in that library. For example:

```
const map = L.mapquest.map("map", {
  center: [53.480759, -2.242631],
  layers: L.mapquest.tileLayer("map"),
  zoom: 12,
});
```

Here we are creating a variable to store the map information in, then creating a new map using the `mapquest.map()` method, which takes as its parameters the ID of a `<div>` element you want to display the map in ('map'), and the coordinates of the center of the map, a map layer of type map to show (created using the `mapquest.tileLayer()` method), and the default zoom level.

They usually require API keys

Security for browser APIs tends to be handled by permission prompts. The purpose of these is so that the user knows what is going on in the websites they visit and is less likely to fall victim to someone using an API in a malicious way.

Third-party APIs have a slightly different permissions system — they tend to use developer keys to allow developers access to the API functionality, which is more to protect the API vendor than the user.

You'll find a line similar to the following in the Mapquest API example:

```
L.mapquest.key = "YOUR-API-KEY-HERE";
```

This line specifies an API or developer key to use in your application — the developer of the application must apply to get a key, and then include it in their code to be allowed access to the API's functionality. In our example, we've just provided a placeholder.]]

Requiring a key enables the API provider to hold users of the API accountable for their actions. When the developer has registered for a key, they are then known to the API provider, and action can be taken if they start to do anything malicious with the API (such as tracking people's location or trying to spam the API with loads of requests to stop it working, for example). The easiest action would be to just revoke their API privileges.

RESTful API

Now let's look at another API example — the <u>New York Times API</u>. This API allows you to retrieve New York Times news story information and display it on your site. This type of API is known as a RESTful API — instead of getting data using the features of a JavaScript library like we did with Mapquest, we get data by making HTTP requests to specific URLs, with data like search terms and other properties encoded in the URL (often as URL parameters). This is a common pattern you'll encounter with APIs.

A RESTful API, or Representational State Transfer API, is an architectural style for designing networked applications. It is commonly used in web development to provide a standardized way for different systems to communicate with each other over the internet.

RESTful APIs are based on a set of principles and constraints that define how the API should behave. These principles include:

1. Client-Server Architecture: The API separates the client (user interface) and the server (data storage), allowing them to evolve independently.
2. Stateless: Each request from the client to the server contains all the necessary information for the server to understand and process the request. The server does not maintain any client-specific state between requests.
3. Uniform Interface: The API uses a consistent set of well-defined methods and standard protocols, such as HTTP, to interact with resources. The standard methods include GET (retrieve a resource), POST (create a new resource), PUT (update a resource), and DELETE (remove a resource).
4. Resource-Based: Resources, such as data objects or services, are identified by unique URLs (Uniform Resource Locators). Clients can interact with these resources using the standard HTTP methods.

5. Representations: Resources can have multiple representations, such as JSON, XML, or HTML. Clients can request a specific representation format based on their needs.
6. Hypermedia as the Engine of Application State (HATEOAS): The API provides links or hypermedia in the response, allowing clients to navigate through the API dynamically by following these links.

By following these principles, RESTful APIs promote scalability, simplicity, and interoperability. They have become the de facto standard for building web services and are widely used in various industries for integrating different systems and enabling communication between applications.

REST API vs RESTful API

A REST API is an application programming interface that adheres strictly to the principles and constraints of the REST architectural style. A **REST API strictly implements all the core principles/rules of REST**, providing a consistent and standardized approach to interact with web services.

A RESTful API is an <u>informal term</u> used to describe an API that is designed and implemented following the principles of REST architecture. **RESTful API tries to implement the principles of REST, but it might not be 100% following all guidelines**. It allows for some flexibility or adaptations in its design and implementation while still being inspired by the REST architectural style.

In summary, the main difference lies in the strictness to the REST principles. A REST API strictly adheres to all the principles and constraints, while a RESTful API aims to follow the principles but allows for some flexibility in its implementation. Both terms refer to APIs that share the fundamental characteristics of REST, providing a scalable and standardized way for clients to interact with servers over the internet.

Overview of Different API Types and Their Characteristics

There are various types of APIs, each with its own characteristics and purposes. Here are some common API types:

1. **RESTful API**: A RESTful API follows the principles of the REST architectural style, using HTTP methods like GET, POST, PUT, and DELETE to interact with resources over the internet. It emphasizes a resource-centric approach and is widely used for building web services.

2. **RPC-style API**: RPC (Remote Procedure Call) APIs focus on exposing remote functions or procedures that can be invoked by clients. Clients send requests specifying the function to be executed along with any necessary parameters. The server processes the request and returns the result.

3. **SOAP-based API**: SOAP (Simple Object Access Protocol) APIs use the SOAP protocol for communication. They typically use XML for data exchange and rely on a rigid contract-based approach. SOAP APIs often have complex payloads and provide extensive support for features like security, reliability, and transactions.

4. **GraphQL API**: GraphQL is a query language and runtime for APIs. With GraphQL APIs, clients can specify precisely the data they need using a flexible and powerful query syntax. The server responds with a JSON payload containing only the requested data, reducing over-fetching or under-fetching of data.

5. **WebSocket API**: WebSocket APIs provide full-duplex communication channels between clients and servers. They enable real-time, bidirectional communication, allowing for constant data updates without the need for frequent polling. WebSocket APIs are suitable for applications requiring live data, such as chat applications or collaborative tools.

These are just a few examples of API types, and there are others as well, such as gRPC (a high-performance RPC framework) and event-driven APIs. The choice of API type depends on the specific requirements of your application, the level of real-time interaction needed, and the preferences of your development team.

Drawing graphics

The browser contains some very powerful graphics programming tools, from the Scalable Vector Graphics (SVG) language, to APIs for drawing on HTML `<canvas>` elements, (see The Canvas API and WebGL). This article provides an introduction to canvas, and further resources to allow you to learn more.

Graphics on the Web

The Web was originally just text, which was very boring, so images were introduced — first via the <img> element and later via CSS properties such as background-image, and SVG.

This however was still not enough. While you could use CSS and JavaScript to animate (and otherwise manipulate) SVG vector images — as they are represented by markup — there was still no way to do the same for bitmap images, and the tools available were rather limited. The Web

still had no way to effectively create animations, games, 3D scenes, and other requirements commonly handled by lower level languages such as C++ or Java.

The situation started to improve when browsers began to support the <canvas> element and associated **Canvas API** in 2004. As you'll see below, canvas provides some useful tools for creating 2D animations, games, data visualizations, and other types of applications, especially when combined with some of the other APIs the web platform provides, but can be difficult or impossible to make accessible.

The below example shows a simple 2D canvas-based bouncing balls animation that we originally met in our Introducing JavaScript objects module:
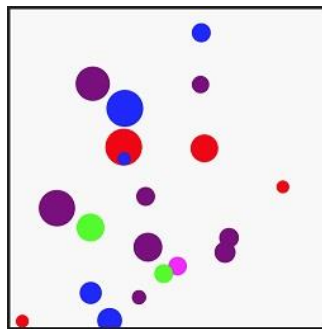
Around 2006–2007, Mozilla started work on an experimental 3D canvas implementation. This became **WebGL**, which gained traction among browser vendors, and was standardized around 2009–2010. WebGL allows you to create real 3D graphics inside your web browser; the below example shows a simple rotating WebGL cube:

Active learning: Getting started with a <canvas>

This example demonstrates how to get started with an HTML <canvas> element and perform some basic setup for drawing on it using JavaScript.

```
<!DOCTYPE html>
<html>
<head>
  <title>Canvas Example</title>
  <style>
    canvas {
      border: 1px solid black;
    }
  </style>
</head>
<body>
  <canvas id="myCanvas" width="320" height="240">
    <p>Canvas not supported.</p>
  </canvas>

  <script>
    const canvas = document.getElementById("myCanvas");
    const ctx = canvas.getContext("2d");

    ctx.fillStyle = "rgb(0, 0, 0)";
    ctx.fillRect(0, 0, canvas.width, canvas.height);
  </script>
</body>
</html>
```

This code sets up a canvas element with a fallback content (`<p>Canvas not supported.</p>`) for browsers that don't support the canvas feature. Inside the script tag, it retrieves the canvas element using its ID, gets the 2D rendering context, sets the fill style to black, and fills the entire canvas with a black rectangle.

When you open this HTML file in a browser, it will display a black rectangle inside the canvas element with a size of 320 pixels by 240 pixels. The canvas element provides a drawing area where you can create various 2D and 3D graphics using JavaScript and the HTML5 canvas API.

2D canvas basics

As we said above, all drawing operations are done by manipulating a `CanvasRenderingContext2D` object (in our case, ctx). Many operations need to be given coordinates to pinpoint exactly where to draw something — the top left of the canvas is point (0, 0), the horizontal (x) axis runs from left to right, and the vertical (y) axis runs from top to bottom.
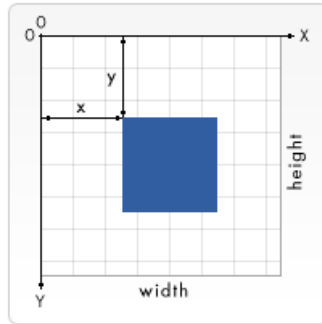
Drawing shapes tends to be done using the rectangle shape primitive, or by tracing a line along a certain path and then filling in the shape. Below we'll show how to do both.

Simple rectangles

```
<!DOCTYPE html>
<html>
<head>
  <title>Canvas Example</title>
  <style>
    canvas {
      border: 1px solid black;
    }
  </style>
</head>
<body>
  <canvas id="myCanvas" width="320" height="240">
    <p>Canvas not supported.</p>
  </canvas>

  <script>
    const canvas = document.getElementById("myCanvas");
    const ctx = canvas.getContext("2d");

    ctx.fillStyle = "rgb(255, 0, 0)";
    ctx.fillRect(50, 50, 100, 150);

    ctx.fillStyle = "rgb(0, 255, 0)";
    ctx.fillRect(75, 75, 100, 100);

    ctx.fillStyle = "rgba(255, 0, 255, 0.75)";
    ctx.fillRect(25, 100, 175, 50);
  </script>
</body>
</html>
```

In this example, we first retrieve the canvas element and its 2D rendering context. Then we use the `fillStyle` property to set the fill color, and the `fillRect` method to draw rectangles on the canvas.

The first rectangle is red (`rgb(255, 0, 0)`) and has its top-left corner positioned at (`50, 50`). It is 100 pixels wide and 150 pixels tall.

The second rectangle is green (`rgb(0, 255, 0)`) and has its top-left corner positioned at (`75, 75`). It is 100 pixels wide and 100 pixels tall.

The third rectangle is semi-transparent purple (`rgba(255, 0, 255, 0.75)`) and has its top-left corner positioned at (`25, 100`). It is 175 pixels wide and 50 pixels tall.

Save and refresh the HTML file in a browser, and you will see the canvas displaying the three rectangles in the specified positions and colors. Feel free to experiment and add more rectangles of your own!

Strokes and line widths

So far we've looked at drawing filled rectangles, but you can also draw rectangles that are just outlines (called strokes in graphic design). To set the color you want for your stroke, you use the `strokeStyle` property; drawing a stroke rectangle is done using `strokeRect`.

Add the following to the previous example, again below the previous JavaScript lines:

```
ctx.strokeStyle = "rgb(255, 255, 255)";
ctx.strokeRect(25, 25, 175, 200);
```

The default width of strokes is 1 pixel; you can adjust the `lineWidth` property value to change this (it takes a number representing the number of pixels wide the stroke is). Add the following line in between the previous two lines:

```
ctx.lineWidth = 5;
```

Now you should see that your white outline has become much thicker! That's it for now. At this point your example should look like this:
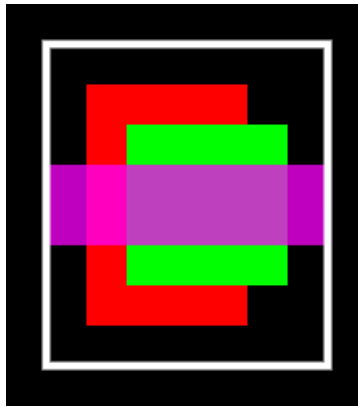
Drawing paths

If you want to draw anything more complex than a rectangle, you need to draw a path. Basically, this involves writing code to specify exactly what path the pen should move along on your canvas to trace the shape you want to draw. Canvas includes functions for drawing straight lines, circles, Bézier curves, and more.

Let's start the section off by making a fresh copy of our canvas template (1_canvas_template), in which to draw the new example.

We'll be using some common methods and properties across all of the below sections:

- `beginPath()` — start drawing a path at the point where the pen currently is on the canvas. On a new canvas, the pen starts out at (0, 0).
- `moveTo()` — move the pen to a different point on the canvas, without recording or tracing the line; the pen "jumps" to the new position.
- `fill()` — draw a filled shape by filling in the path you've traced so far.
- `stroke()` — draw an outline shape by drawing a stroke along the path you've drawn so far.
- You can also use features like `lineWidth` and `fillStyle`/`strokeStyle` with paths as well as rectangles.

A typical, simple path-drawing operation would look something like so:

```
ctx.fillStyle = "rgb(255, 0, 0)";
ctx.beginPath();
ctx.moveTo(50, 50);
// draw your path
ctx.fill();
```

Drawing lines

Let's draw an equilateral triangle on the canvas.

1. First of all, add the following helper function to the bottom of your code. This converts degree values to radians, which is useful because whenever you need to provide an angle value in JavaScript, it will nearly always be in radians, but humans usually think in degrees.

```
function degToRad(degrees) {
  return (degrees * Math.PI) / 180;
}
```

2. Next, start off your path by adding the following below your previous addition; here we set a color for our triangle, start drawing a path, and then move the pen to (50, 50) without drawing anything. That's where we'll start drawing our triangle.

```
ctx.fillStyle = "rgb(255, 0, 0)";
ctx.beginPath();
ctx.moveTo(50, 50);
```

3. Now add the following lines at the bottom of your script:

```
ctx.lineTo(150, 50);
const triHeight = 50 * Math.tan(degToRad(60));
ctx.lineTo(100, 50 + triHeight);
ctx.lineTo(50, 50);
ctx.fill();
```

Let's run through this in order: First we draw a line across to (150, 50) — our path now goes 100 pixels to the right along the x axis. Second, we work out the height of our equilateral triangle, using a bit of simple trigonometry. Basically, we are drawing the triangle pointing downwards. The angles in an equilateral triangle are always 60 degrees; to work out the height we can split it down the middle into two right-angled triangles, which will each have angles of 90 degrees, 60 degrees, and 30 degrees. In terms of the sides:

- The longest side is called the **hypotenuse**
- The side next to the 60 degree angle is called the **adjacent** — which we know is 50 pixels, as it is half of the line we just drew.

- The side opposite the 60 degree angle is called the **opposite**, which is the height of the triangle we want to calculate.
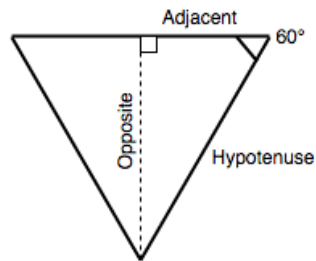
One of the basic trigonometric formulae states that the length of the adjacent multiplied by the tangent of the angle is equal to the opposite, hence we come up with 50 * `Math.tan(degToRad(60))`. We use our `degToRad()` function to convert 60 degrees to radians, as `Math.tan()` expects an input value in radians.

4. With the height calculated, we draw another line to (`100,  50 + triHeight`). The X coordinate is simple; it must be halfway between the previous two X values we set. The Y value on the other hand must be 50 plus the triangle height, as we know the top of the triangle is 50 pixels from the top of the canvas.

5. The next line draws a line back to the starting point of the triangle.

6. Last of all, we run `ctx.fill()` to end the path and fill in the shape.

Drawing circles

Now let's look at how to draw a circle in canvas. This is accomplished using the arc() method, which draws all or part of a circle at a specified point.

1. Let's add an arc to our canvas — add the following to the bottom of your code:

```
ctx.fillStyle = "rgb(0, 0, 255)";
ctx.beginPath();
ctx.arc(150, 106, 50, degToRad(0), degToRad(360), false);
ctx.fill();
```

`arc()` takes six parameters. The first two specify the position of the arc's center (X and Y, respectively). The third is the circle's radius, the fourth and fifth are the start and end angles at which to draw the circle (so specifying 0 and 360 degrees gives us a full circle), and the sixth

parameter defines whether the circle should be drawn counterclockwise (anticlockwise) or clockwise (false is clockwise).

2. Let's try adding another arc:

```
ctx.fillStyle = "yellow";
ctx.beginPath();
ctx.arc(200, 106, 50, degToRad(-45), degToRad(45), true);
ctx.lineTo(200, 106);
ctx.fill();
```

The pattern here is very similar, but with two differences:

- We have set the last parameter of `arc()` to true, meaning that the arc is drawn counterclockwise, which means that even though the arc is specified as starting at -45 degrees and ending at 45 degrees, we draw the arc around the 270 degrees not inside this portion. If you were to change true to false and then re-run the code, only the 90 degree slice of the circle would be drawn.

- Before calling `fill()`, we draw a line to the center of the circle. This means that we get the rather nice Pac-Man-style cutout rendered. If you removed this line (try it!) then re-ran the code, you'd get just an edge of the circle chopped off between the start and end point of the arc. This illustrates another important point of the canvas — if you try to fill an incomplete path (i.e. one that is not closed), the browser fills in a straight line between the start and end point and then fills it in.

That's it for now; your final example should look like this:
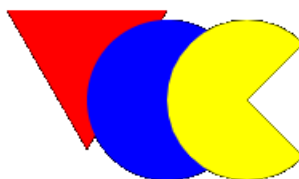
Text

Canvas also has features for drawing text. Text is drawn using two methods:

244

- `fillText()` — draws filled text.
- `strokeText()` — draws outline (stroke) text.

Both of these take three properties in their basic usage: the text string to draw and the X and Y coordinates of the point to start drawing the text at. This works out as the **bottom left** corner of the **text box** (literally, the box surrounding the text you draw), which might confuse you as other drawing operations tend to start from the top left corner — bear this in mind.

There are also a number of properties to help control text rendering such as font, which lets you specify font family, size, etc. It takes as its value the same syntax as the CSS font property.

Canvas content is not accessible to screen readers. Text painted to the canvas is not available to the DOM, but must be made available to be accessible. In this example, we include the text as the value for aria-label.

Try adding the following block to the bottom of your JavaScript:

```
ctx.strokeStyle = "white";
ctx.lineWidth = 1;
ctx.font = "36px arial";
ctx.strokeText("Canvas text", 50, 50);

ctx.fillStyle = "red";
ctx.font = "48px georgia";
ctx.fillText("Canvas text", 50, 150);

canvas.setAttribute("aria-label", "Canvas text");
```

Here we draw two lines of text, one outline and the other stroke. The final example should look like so:
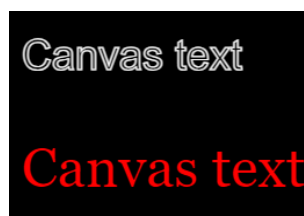
Drawing images onto canvas

The HTML5 canvas element provides a powerful feature for rendering graphics, including images. In this lesson, we will learn how to draw images on a canvas using JavaScript.

Step 1: Setting up the Canvas.

- Create an HTML file and add a canvas element to it. Give it an id for easy reference.

```
<canvas id="myCanvas"></canvas>
```

Step 2: Drawing an Image on the Canvas.

- Get a reference to the element using its id, and 2D rendering context of the canvas.

```
const canvas = document.getElementById("myCanvas");
const ctx = canvas.getContext("2d");
```

- Create a new Image object and set its source to the path or URL of the image you want to draw.

```
const image = new Image();
image.src = "path/to/image.jpg";
```

- Wait for the image to load using the `load` event. Add an event listener to the image and use the `drawImage()` method inside the event handler.

```
image.addEventListener("load", () => { ctx.drawImage(image, 0, 0); });
```

- Save the changes and open the HTML file in a browser. You should see the image displayed on the canvas.

Step 3: Manipulating the Image.

To display only a part of the image or resize it, you can use the more complex version of `drawImage()`. It takes additional parameters to define the area of the image to cut out and the position and size of the drawn image on the canvas.

```
  ctx.drawImage(image, sourceX, sourceY, sourceWidth, sourceHeight, destX, destY,
destWidth, destHeight);
```

- `sourceX, sourceY`: The coordinates of the top-left corner of the area to cut out from the image.

- sourceWidth, sourceHeight: The width and height of the area to cut out.
- destX, destY: The coordinates of the top-left corner where the image should be drawn on the canvas.
- destWidth, destHeight: The width and height at which to draw the image.

Loops and animations

You won't experience the full power of canvas unless you update or animate it in some way. After all, canvas does provide scriptable images! If you aren't going to change anything, then you might as well just use static images and save yourself all the work.

Creating a loop

In this lesson, we will learn how to use loops in canvas to draw rotating triangles. It's a fun way to play with canvas and create interesting designs.
Step 1: Setting up the Canvas:

- Create an HTML file and add a canvas element to it. Give it an id for easy reference.

```
<canvas id="myCanvas"></canvas>
```

Step 2: Drawing Rotating Triangles:

- Create a JavaScript file (e.g., script.js) and open it.
- Get a reference to the element via id, and get the 2D rendering context of the canvas.

```
const canvas = document.getElementById("myCanvas");
const ctx = canvas.getContext("2d");
```

- Set the origin point of the canvas to the center using the translate() method.

```
const width = canvas.width;
const height = canvas.height;
ctx.translate(width / 2, height / 2);
```

- Define utility functions for converting degrees to radians and generating random numbers, and set initial values for the length and moveOffset variables.

```
function degToRad(degrees) {
    return (degrees * Math.PI) / 180;
}

function rand(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
}

let length = 250; let moveOffset = 20;
```

- Use a for loop to draw the rotating triangles.

```
for (let i = 0; i < length; i++) {
  ctx.fillStyle = `rgba(${255 - length}, 0, ${255 - length}, 0.9)`;
  ctx.beginPath();
  ctx.moveTo(moveOffset, moveOffset);
  ctx.lineTo(moveOffset + length, moveOffset);
  const triHeight = (length / 2) * Math.tan(degToRad(60));
  ctx.lineTo(moveOffset + length / 2, moveOffset + triHeight);
  ctx.lineTo(moveOffset, moveOffset);
  ctx.fill();

  length--;
  moveOffset += 0.7;
  ctx.rotate(degToRad(5));
}
```

That's it! The final example should look like so:
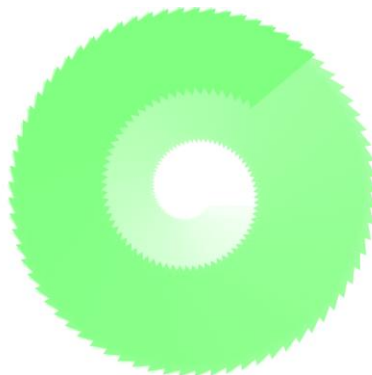
Animations

Canvas animations involve continuously updating and redrawing the canvas to create smooth and dynamic visual effects. They create the illusion of movement by continuously updating and redrawing elements on the canvas.

To achieve smooth animations, the `requestAnimationFrame()` method is used to schedule callback functions that update and redraw the canvas before the next repaint. This method ensures synchronization with the browser's rendering loop. If an animation needs to be stopped before completion, the `cancelAnimationFrame()` method can be called, providing the unique identifier returned by `requestAnimationFrame()`. This effectively halts the animation loop and prevents further updates to the canvas.

A simple character animation

```
<canvas id="myCanvas"></canvas>
<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");
  const spriteWidth = 100; // Width of each frame in the sprite sheet
  const spriteHeight = 100; // Height of each frame in the sprite sheet
  const totalFrames = 6; // Total number of frames in the sprite sheet
  const frameWidth = spriteWidth / totalFrames; // Width of each frame on the canvas
  const spriteSheet = new Image();
  spriteSheet.src = "spritesheet.png";
  let currentFrame = 0; // Index of the current frame in the sprite sheet

  function animate() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    ctx.drawImage(
      spriteSheet,
      currentFrame * spriteWidth,
      0,
      spriteWidth,
      spriteHeight,
      0,
      0,
      frameWidth,
      spriteHeight
    );

    currentFrame = (currentFrame + 1) % totalFrames;
    requestAnimationFrame(animate);
  }

  spriteSheet.addEventListener("load", () => {
    animate();
```

```
    });
</script>
```

Here how spritesheet looks like:

Explanation:

1. The HTML file includes a canvas element where the animation will be displayed.
2. In JavaScript, we get a reference to the canvas and the 2D rendering context.
3. We define the dimensions of each frame in the sprite sheet (`spriteWidth` and `spriteHeight`) and the total number of frames (`totalFrames`).
4. We calculate the width of each frame on the canvas (`frameWidth`) by dividing the sprite width by the total frames.
5. We create a new `Image` object and set its source to the sprite sheet image file (`spritesheet.png`).
6. The `animate` function is called to start the animation loop.
7. Inside the `animate` function, we clear the canvas using `clearRect()`.
8. We use `drawImage()` to draw the current frame of the sprite sheet on the canvas. By specifying the source rectangle from the sprite sheet (`currentFrame * spriteWidth`), we can display the correct portion of the sprite sheet. The destination rectangle on the canvas is specified as (`0, 0, frameWidth, spriteHeight`).
9. We increment the `currentFrame` index and wrap it around to `0` when it exceeds the total number of frames. This creates the looping effect for the animation.
10. The animation loop is created using `requestAnimationFrame(animate)` to continuously update and redraw the canvas at the browser's optimal frame rate.
11. Finally, we add an event listener to the `spriteSheet` image object to start the animation once the image has finished loading.

This example demonstrates how to animate a character walking using a sprite sheet. By updating the `currentFrame` index and drawing the corresponding frame on the canvas, we create the illusion of movement.

WebGL

It's now time to leave 2D behind, and take a quick look at 3D canvas. 3D canvas content is specified using the **WebGL API**, which is a completely **separate API from the 2D canvas API**, even though they both render onto <canvas> elements.

WebGL is based on OpenGL (Open Graphics Library), and allows you to communicate directly with the computer's GPU. As such, writing raw WebGL is closer to low level languages such as C++ than regular JavaScript; it is quite complex but incredibly powerful.

Using a library

Because of its complexity, most people write 3D graphics code using a third party JavaScript library such as Three.js, PlayCanvas, or Babylon.js. Most of these work in a similar way, providing functionality to create primitive and custom shapes, position viewing cameras and lighting, covering surfaces with textures, and more. They handle the WebGL for you, letting you work on a higher level.

Yes, using one of these means learning another new API (a third-party one, in this case), but they are a lot simpler than coding raw WebGL.

Creating 3D rotating cube in Three.js

```
      const scene = new THREE.Scene();
      const camera = new THREE.PerspectiveCamera (
75, window.innerWidth / window.innerHeight, 0.1, 1000);

      const renderer = new THREE.WebGLRenderer();
      renderer.setSize(window.innerWidth, window.innerHeight);
      document.body.appendChild(renderer.domElement);

      // Create Cube
      const geometry = new THREE.BoxGeometry(1, 1, 1);
      const material = new THREE.MeshBasicMaterial({ color: 0x00ff00 });
      const cube = new THREE.Mesh(geometry, material);
      scene.add(cube);

      // Animation
      function animate() {
        requestAnimationFrame(animate);
        cube.rotation.x += 0.01;
        cube.rotation.y += 0.01;
        renderer.render(scene, camera);
      }
```

```
    // Start Animation
    animate();
```

Explanation:

1. In JavaScript, we set up the basic elements for our scene: `scene`, `camera`, and `renderer`. The `scene` represents the 3D environment, the `camera` determines the perspective and view of the scene, and the `renderer` handles rendering the scene on the canvas.
2. We create a cube using `BoxGeometry` with a width, height, and depth of 1. Then we create a `MeshBasicMaterial` with a green color for the cube.
3. We add the cube to the scene using `scene.add(cube).`
4. The `animate` function is created to handle the animation loop using `requestAnimationFrame`. It is called recursively to continuously update and render the scene.
5. Inside the `animate` function, we update the rotation of the cube by incrementing its `rotation.x` and rotation.y properties. This creates a smooth rotating effect.
6. Finally, we call `renderer.render(scene, camera)` to render the updated scene with the cube's rotation onto the canvas.

When you open this HTML file in a browser, you should see a rotating 3D cube displayed on the screen. The cube will continue to rotate indefinitely due to the animation loop created by `requestAnimationFrame` in the `animate` function.

Video and Audio APIs

HTML comes with elements for embedding rich media in documents — `<video>` and `<audio>` — which in turn come with their own APIs for controlling playback, seeking, etc. This article shows you how to do common tasks such as creating custom playback controls.

HTML video and audio

The `<video>` and `<audio>` elements allow us to embed video and audio into web pages. As we showed in Video and audio content, a typical implementation looks like this:

```
<video controls>
  <source src="rabbit320.mp4" type="video/mp4" />
  <source src="rabbit320.webm" type="video/webm" />
```

```
  <p>
    Your browser doesn't support HTML video. Here is a
    <a href="rabbit320.mp4">link to the video</a> instead.
  </p>
</video>
```


You can review what all the HTML features do in the article linked above; for our purposes here, the most interesting attribute is controls, which enables the default set of playback controls. If you don't specify this, you get no playback controls.

This is not as immediately useful for video playback, but it does have advantages. One big issue with the native browser controls is that they are different in each browser — not very good for cross-browser support! Another big issue is that the native controls in most browsers aren't very keyboard-accessible.

You can solve both these problems by hiding the native controls (by removing the controls attribute), and programming your own with HTML, CSS, and JavaScript. In the next section, we'll look at the basic tools we have available to do this.

The HTMLMediaElement API

The HTMLMediaElement API provides more flexibility and control over the media playback. With the API, you can programmatically control various aspects of the media element, such as **starting and pausing playback, changing the source dynamically, seeking to a specific time, adjusting the volume, and handling custom events**. This is useful when you need to create a custom user interface or implement specific functionality beyond what the default controls offer.

Using the HTMLMediaElement API allows you to have complete control over the media playback experience, but it also requires more coding and implementation effort compared to using the native controls.

```html
<!DOCTYPE html>
<html>
<head>
  <title>HTMLMediaElement API Example</title>
</head>
<body>
  <video id="myVideo" src="video.mp4"></video>
  <button id="playButton">Play</button>
  <button id="pauseButton">Pause</button>

  <script>
    // Get references to the video element and control buttons
    const video = document.getElementById('myVideo');
    const playButton = document.getElementById('playButton');
    const pauseButton = document.getElementById('pauseButton');

    // Add click event listeners to the buttons
    playButton.addEventListener('click', () => {
      video.play(); // play() method is part of the HTMLMediaElement API
    });

    pauseButton.addEventListener('click', () => {
      video.pause(); // pause() method is part of the HTMLMediaElement API
    });
  </script>
</body>
</html>
```

In this example, we have a video element with an id of "myVideo" and two buttons with id attributes of "playButton" and "pauseButton". When the "Play" button is clicked, the play() method is called on the video element, and it starts playing. Similarly, when the "Pause" button is clicked, the pause() method is called to pause the video.

Note that this is a basic example, and the HTMLMediaElement API provides more advanced functionality and additional methods.

Client-side storage

Modern web browsers support a number of ways for websites to store data on the user's computer — with the user's permission — then retrieve it when necessary. This lets you persist data for long-term storage, save sites or documents for offline use, retain user-specific settings for your site, and more. This article explains the very basics of how these work.

Client-side storage?

We talked about the difference between static sites and dynamic sites. Most major modern websites are dynamic — they store data on the server using some kind of database (server-side storage), then run server-side code to retrieve needed data, insert it into static page templates, and serve the resulting HTML to the client to be displayed by the user's browser.

Client-side storage works on similar principles, but has different uses. It consists of JavaScript APIs that allow you to store data on the client (i.e. on the user's machine) and then retrieve it when needed. This has many distinct uses, such as:

- Personalizing site preferences (e.g. showing a user's choice of custom widgets, color scheme, or font size).
- Persisting previous site activity (e.g. storing the contents of a shopping cart from a previous session, remembering if a user was previously logged in).
- Saving data and assets locally so a site will be quicker (and potentially less expensive) to download, or be usable without a network connection.
- Saving web application-generated documents locally for use offline

Often client-side and server-side storage are used together. For example, you could download a batch of music files (perhaps used by a web game or music player application), store them inside a client-side database, and play them as needed. The user would only have to download the music files once — on subsequent visits, they would be retrieved from the database instead.

Old school: Cookies

The concept of client-side storage has been around for a long time. Since the early days of the web, sites have used cookies to store information to personalize user experience on websites. They're the earliest form of client-side storage commonly used on the web.

These days, there are easier mechanisms available for storing client-side data, however, they are still used commonly to store data related to user personalization and state, e.g. session IDs and access tokens.

New school: Web Storage and IndexedDB

The "easier" features we mentioned above are as follows:

- The Web Storage API provides a mechanism for storing and retrieving smaller, data items consisting of a name and a corresponding value. This is useful when you just need to store some simple data, like the user's name, whether they are logged in, what color to use for the background of the screen, etc.
- The IndexedDB API provides the browser with a complete database system for storing complex data. This can be used for things from complete sets of customer records to even complex data types like audio or video files.

The Cache API

The `Cache` API is designed for storing HTTP responses to specific requests, and is very useful for doing things like storing website assets offline so the site can subsequently be used without a network connection. Cache is usually used in combination with the Service Worker API, although it doesn't have to be.

The use of Cache and Service Workers is an advanced topic, and we won't be covering it in great detail in this article, although we will show an example in the Offline asset storage section below.

Storing simple data — web storage

The Web Storage API is very easy to use — you store simple name/value pairs of data (limited to strings, numbers, etc.) and retrieve these values when needed.

Basic syntax

Here's an explanation of the basic syntax with example code:

1. Storing Data: To store data in web storage, you use the `setItem()` method provided by the `localStorage` object. The syntax is as follows:

```
localStorage.setItem("name", "Chris");
```

In this example, the name "`Chris`" is stored with the key "`name`" in the web storage.

2. Retrieving Data: To retrieve data from web storage, you use the `getItem()` method provided by the `localStorage` object. The syntax is as follows:

```
let myName = localStorage.getItem("name");
```

In this example, the value stored with the key "name" is retrieved and assigned to the myName variable.

3. Removing Data: To remove data from web storage, you use the removeItem() method provided by the localStorage object. The syntax is as follows:

```
localStorage.removeItem("name");
```

In this example, the item with the key "name" is removed from the web storage.

It's important to note that the localStorage object is used in this example, which persists data even after the browser is closed and reopened. If you want data to be available only during the browser session, you can use the sessionStorage object instead.

After executing these operations, you can check the value of the myName variable or use the getItem() method again to see if the data item exists or has been removed.

The data persists!

One key feature of web storage is that the **data persists between page loads** (and even when the browser is shut down, in the case of localStorage). Let's look at this in action.

1. Storing Data:

```
localStorage.setItem("name", "Chris");
```

This line of code stores the value "Chris" with the key "name" in the web storage (localStorage).

2. Retrieving Data:

```
let myName = localStorage.getItem("name"); myName;
```

These lines retrieve the value associated with the key "name" from the web storage and assign it to the variable myName. By accessing myName, you should see the value "Chris" being displayed.

3. Persisting Data Between Browser Sessions: Now, close the browser completely and reopen it.

4. Retrieving Data Again:

```
let myName = localStorage.getItem("name"); myName;
```

After reopening the browser, these lines of code retrieve the value associated with the key `"name"` from the web storage. The variable `myName` should still hold the value `"Chris"`.
The key feature of web storage, especially `localStorage`, is that the stored data persists between page loads and even when the browser is shut down and reopened. In this example, the value `"Chris"` remains available in the web storage even after closing and reopening the browser.

Separate storage for each domain

There is a separate data store for each domain (each separate web address loaded in the browser). You will see that if you load two websites (say google.com and amazon.com) and try storing an item on one website, it won't be available on the other website.

This makes sense — you can imagine the security issues that would arise if websites could see each other's data!

Storing complex data — IndexedDB

The IndexedDB API (sometimes abbreviated IDB) is a complete database system available in the browser in which you can store complex related data, the types of which aren't limited to simple values like strings or numbers. You can store videos, images, and pretty much anything else in an IndexedDB instance.

The IndexedDB API allows you to create a database, then create object stores within that database. Object stores are like tables in a relational database, and each object store can contain a number of objects. To learn more about the IndexedDB API, see Using IndexedDB.

However, this does come at a cost: IndexedDB is much more complex to use than the Web Storage API. In this section, we'll really only scratch the surface of what it is capable of, but we will give you enough to get started.

Database Initial Setup

```javascript
let db;
const openRequest = window.indexedDB.open("notes_db", 1);

openRequest.addEventListener(
    "error", () => console.error("Database failed to open"));

openRequest.addEventListener("success", () => {
  db = openRequest.result;
  displayData();
});

openRequest.addEventListener("upgradeneeded", (e) => {
  db = e.target.result;

  const objectStore = db.createObjectStore(
      "notes_os", { keyPath: "id", autoIncrement: true });
  objectStore.createIndex("title", "title", { unique: false });
  objectStore.createIndex("body", "body", { unique: false });

  console.log("Database setup complete");
});
```

In this example, we initialize the IndexedDB database by creating an instance of it. We specify the database name as "notes_db" and the version as 1. The openRequest variable holds the request to open the database.

We add event listeners to handle errors, success, and upgrades. If the database opening encounters an error, an error message is logged. If the database opens successfully, the db variable is assigned to the result, and the displayData() function is called. During an upgrade, the upgradeneeded event is triggered, allowing us to define the structure of the object store. Here, we create an object store called "notes_os" with an auto-incrementing key and two indexes for the title and body fields.

Adding Data to the Database

```javascript
function addData(note) {
  const transaction = db.transaction(["notes_os"], "readwrite");
  const objectStore = transaction.objectStore("notes_os");
  const addRequest = objectStore.add(note);

  addRequest.addEventListener("success", () => {
    console.log("Note added successfully");
```

```
    displayData();
  });

  transaction.addEventListener("error", () =>
    console.log("Transaction not opened due to error")
  );
}
```

The `addData` function takes a `note` object as a parameter and adds it to the "`notes_os`" object store. It starts a read/write transaction, accesses the object store, and uses the `add()` method to add the note.

After the note is successfully added, a success event is triggered, and the `displayData()` function is called to update the display. If an error occurs during the transaction, an error message is logged.

Displaying the Data

```
function displayData() {
  const objectStore = db.transaction("notes_os").objectStore("notes_os");

  while (list.firstChild) {
    list.removeChild(list.firstChild);
  }

  objectStore.openCursor().addEventListener("success", (e) => {
    const cursor = e.target.result;

    if (cursor) {
      const listItem = createListItem(cursor.value);
      list.appendChild(listItem);

      cursor.continue();
    }
  });
}

function createListItem(note) {
  const listItem = document.createElement("li");
  const h3 = document.createElement("h3");
  const para = document.createElement("p");

  h3.textContent = note.title;
  para.textContent = note.body;

  listItem.appendChild(h3);
  listItem.appendChild(para);
```

260

```
    return listItem;
  }
```

The `displayData` function retrieves data from the "`notes_os`" object store and displays it on the page. It starts a transaction, accesses the object store, and uses the `openCursor()` method to iterate over the records.

Inside the success event handler for the cursor, the function creates a list item using the `createListItem` helper function. The `createListItem` function takes a note object and creates HTML elements representing the note's title and body. The list item is then appended to the `<ul>` element.

The cursor continues to the next record until there are no more records.

Deleting a Note

```
  function deleteItem(e) {
    const noteId = parseInt(e.target.parentNode.getAttribute("data-note-id"));
    const transaction = db.transaction(["notes_os"], "readwrite");
    const objectStore = transaction.objectStore("notes_os");
    const deleteRequest = objectStore.delete(noteId);

    deleteRequest.addEventListener("success", () => {
      console.log("Note deleted successfully");
      displayData();
    });

    transaction.addEventListener("error", () =>
      console.log("Transaction not opened due to error")
    );
  }
```

The `deleteItem` function is triggered when the delete button associated with a note is clicked. It retrieves the note's ID from the data attribute of the parent list item. A read/write transaction is initiated, and the note is deleted from the "`notes_os`" object store using the `delete()` method. After the deletion is successful, the `displayData()` function is called to update the display. If an error occurs during the transaction, an error message is logged.

Storing Complex Data via IndexedDB

As we mentioned above, IndexedDB can be used to store more than just text strings. You can store just about anything you want, including complex objects such as video or image blobs. And it isn't much more difficult to achieve than any other type of data.

In this example, we'll demonstrate how to store video blobs in an IndexedDB database and display them in a web page. The code snippets provided will highlight the most relevant parts of the example.

Storing the Video Data:

First, we define an array of video names that we want to fetch and store in the IndexedDB database:

```
const videos = [
  { name: "crystal" },
  { name: "elf" },
  { name: "frog" },
  { name: "pig" },
  { name: "rabbit" },
];
```

Inside the `init()` function, we iterate through the video names. We check if each video exists in the database using `objectStore.get()`. If the video is present, we display it from IndexedDB using the `displayVideo()` function. Otherwise, we fetch it from the network using `fetchVideoFromNetwork()`:

```
function init() {
  for (const video of videos) {
    const objectStore = db.transaction("videos_os").objectStore("videos_os");
    const request = objectStore.get(video.name);
    request.addEventListener("success", () => {
      if (request.result) {
        displayVideo(
          request.result.mp4,
          request.result.webm,
          request.result.name
        );
      } else {
        fetchVideoFromNetwork(video);
      }
    });
```

```
  }
}
```

Inside the `fetchVideoFromNetwork()` function, we use `fetch()` to fetch the MP4 and WebM versions of the video. We extract the response bodies as blobs using the `response.blob()` method. Since both fetch requests are asynchronous, we use `Promise.all()` to wait for both promises to fulfill. Then, we display the video using `displayVideo()` and store it in IndexedDB using `storeVideo()`:

```
const mp4Blob = fetch(`videos/${video.name}.mp4`).then((response) =>
  response.blob()
);
const webmBlob = fetch(`videos/${video.name}.webm`).then((response) =>
  response.blob()
);

Promise.all([mp4Blob, webmBlob]).then((values) => {
  displayVideo(values[0], values[1], video.name);
  storeVideo(values[0], values[1], video.name);
});
```

The `storeVideo()` function opens a `readwrite` transaction, gets a reference to the object store, creates an object representing the video record, and adds it to the database using `IDBObjectStore.add()`:

```
function storeVideo(mp4, webm, name) {
  const objectStore = db
    .transaction(["videos_os"], "readwrite")
    .objectStore("videos_os");

  const request = objectStore.add({ mp4, webm, name });

  request.addEventListener("success", () =>
    console.log("Record addition attempt finished")
  );
  request.addEventListener("error", () => console.error(request.error));
}
```

Displaying the Video Data:

The `displayVideo()` function creates the necessary DOM elements to embed the video in the web page. It also converts the video blobs into object URLs using `URL.createObjectURL()` and sets them as the source URLs for the `<source>` elements within the `<video>` element:

```
function displayVideo(mp4Blob, webmBlob, title) {
  const mp4URL = URL.createObjectURL(mp4Blob);
  const webmURL = URL.createObjectURL(webmBlob);
  const article = document.createElement("article");
  const h2 = document.createElement("h2");
  h2.textContent = title;
  const video = document.createElement("video");
  video.controls = true;
  const source1 = document.createElement("source");
  source1.src = mp4URL;
  source1.type = "video/mp4";
  const source2 = document.createElement("source");
  source2.src = webmURL;
  source2.type = "video/webm";

  section.appendChild(article);
  article.appendChild(h2);
  article.appendChild(video);
  video.appendChild(source1);
  video.appendChild(source2);
}
```

The `displayVideo()` function creates the necessary HTML structure and appends it to the page. The video blobs are displayed within the `<video>` element, allowing users to control and view the videos.

These code snippets demonstrate how to store complex data (video blobs) in IndexedDB and retrieve and display them in a web page.

Please note that this example assumes you have already initialized and set up the IndexedDB database and object store appropriately.

Offline asset storage

The above example already shows how to create an app that will store large assets in an IndexedDB database, avoiding the need to download them more than once. This is already a great improvement to the user experience, but there is still one thing missing — the main HTML, CSS, and JavaScript files still need to be downloaded each time the site is accessed, meaning that it won't work when there is no network connection.
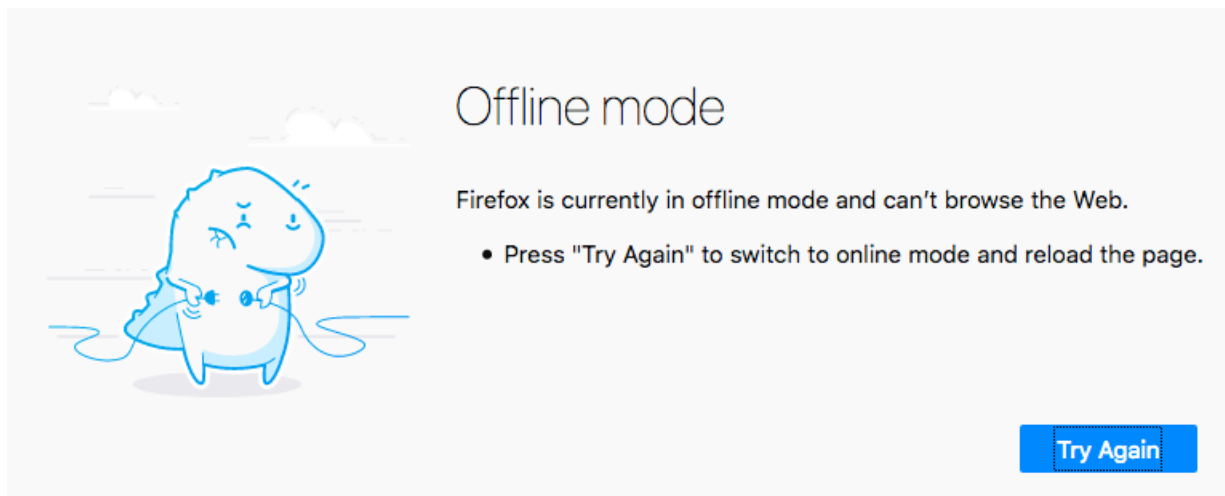
This is where Service workers and the closely-related Cache API come in.

A service worker is a JavaScript file that is registered against a particular origin (website, or part of a website at a certain domain) when it is accessed by a browser. When registered, it can control pages available at that origin. It does this by sitting between a loaded page and the network and intercepting network requests aimed at that origin.

When it intercepts a request, it can do anything you wish to it (see use case ideas), but the classic example is saving the network responses offline and then providing those in response to a request instead of the responses from the network. In effect, it allows you to make a website work completely offline.

The Cache API is another client-side storage mechanism, with a bit of a difference — it is designed to save HTTP responses, and so works very well with service workers.

<div align="center">A service worker example</div>

Here's an explanation and code example that demonstrates offline functionality using the Cache API and IndexedDB.

Registering the Service Worker:

In the main JavaScript file (e.g., index.js), we check if the `serviceWorker` property exists in the `navigator` object to detect if service workers are supported. If supported, we use the `navigator.serviceWorker.register()` method to register the service worker located in the `sw.js` file. This registration allows the service worker to control pages within the same directory or subdirectories:

```
if ("serviceWorker" in navigator) {
  navigator.serviceWorker
    .register("/sw.js")
    .then(() => console.log("Service Worker Registered"));
}
```

Installing the Service Worker:

When a page controlled by the service worker is accessed, the service worker is installed. The `install` event is triggered, and you can write code inside the service worker to respond to the installation. In the `sw.js` file, we listen for the install event and use the `waitUntil()` method to ensure the installation is completed successfully. Inside the `waitUntil()` promise, we use the `caches.open()` method to open a new cache object, and then we use `cache.addAll()` to fetch and store specific assets in the cache:

```
self.addEventListener("install", (e) => {
  e.waitUntil(
    caches.open("video-store").then((cache) =>
      cache.addAll([
        "/index.html",
        "/style.css",
        "/index.js",
        "/video1.mp4",
        "/video2.mp4",
      ])
    )
  );
});
```

Responding to Requests:

After the service worker is installed, it can respond to further network requests. In the `sw.js` file, we listen for the `fetch` event and log the requested asset's URL. We then use `caches.match()` to check if a matching request is found in the cache. If a match is found, we return the cached response. Otherwise, we fetch the response from the network:

```
self.addEventListener("fetch", (e) => {
  console.log(e.request.url);
  e.respondWith(
    caches.match(e.request).then((response) => response || fetch(e.request))
  );
```

```
    });
```

Testing Offline Functionality:

To test the offline functionality, you need to load the page multiple times to ensure the service worker is installed. Once installed, you can simulate being offline by disconnecting from the network, enabling the "Work Offline" option in Firefox, or using Chrome DevTools to go offline. When you refresh the page in offline mode, it should still load successfully since the assets are stored in the cache and served by the service worker.

This example demonstrates how a service worker can cache and serve assets, allowing a web app to function offline. The Cache API is used to store static assets, while the service worker intercepts network requests to serve responses from the cache when available.

Please note that you need to adjust the file paths in the example to match your specific file structure.

Equality comparisons and sameness

JavaScript provides three different value-comparison operations:

- `===` — strict equality (triple equals)
- `==` — loose equality (double equals)
- `Object.is()`

Which operation you choose depends on what sort of comparison you are looking to perform. Briefly:

- Double equals (`==`) will perform a type conversion when comparing two things, and will handle `NaN`, `-0`, and `+0` specially to conform to IEEE 754 (so `NaN != NaN`, and `-0 == +0`);
- Triple equals (`===`) will do the same comparison as double equals (including the special handling for `NaN`, `-0`, and `+0`) but without type conversion; if the types differ, `false` is returned.
- `Object.is()` does no type conversion and no special handling for `NaN`, `-0`, and `+0` (giving it the same behavior as `===` except on those special numeric values).

They correspond to three of four equality algorithms in JavaScript:

- IsLooselyEqual: `==`
- IsStrictlyEqual: `===`
- SameValue: `Object.is()`
- SameValueZero: used by many built-in operations

Note that the distinction between these all have to do with their handling of primitives; none of them compares whether the parameters are conceptually similar in structure. For any non-primitive objects `x` and `y` which have the same structure but are distinct objects themselves, all of the above forms will evaluate to `false`.

Strict equality using ===

Strict equality compares two values for equality. Neither value is implicitly converted to some other value before being compared. If the values have different types, the values are considered unequal. If the values have the same type, are not numbers, and have the same value, they're considered equal. Finally, if both values are numbers, they're considered equal if they're both not `NaN` and are the same value, or if one is `+0` and one is `-0`.

```
const num = 0;
const obj = new String("0");
const str = "0";

console.log(num === num); // true
console.log(obj === obj); // true
console.log(str === str); // true

console.log(num === obj); // false
console.log(num === str); // false
console.log(obj === str); // false
console.log(null === undefined); // false
console.log(obj === null); // false
console.log(obj === undefined); // false
```

Strict equality is almost always the correct comparison operation to use. For all values except numbers, it uses the obvious semantics: a value is only equal to itself. For numbers it uses slightly different semantics to gloss over two different edge cases. The first is that floating point zero is either positively or negatively signed. This is useful in representing certain mathematical solutions, but as most situations don't care about the difference between +0 and -0, strict equality treats them as the same value. The second is that floating point includes the concept of a not-a-number value, NaN, to represent the solution to certain ill-defined mathematical problems: negative infinity added to positive infinity, for example. Strict equality treats NaN as unequal to every other value — including itself. (The only case in which (x !== x) is true is when x is NaN.)

Besides ===, strict equality is also used by array index-finding methods including Array.prototype.indexOf(),Array.prototype.lastIndexOf(),TypedArray.prototype.indexOf() or TypedArray.prototype.lastIndexOf(), and case-matching. This means you cannot use indexOf(NaN) to find the index of a NaN value in an array, or use NaN as a case value in a switch statement and make it match anything.

```
console.log([NaN].indexOf(NaN)); // -1
switch (NaN) {
  case NaN:
    console.log("Surprise"); // Nothing is logged
}
```

Loose equality using ==

Loose equality is *symmetric*: `A  ==  B` always has identical semantics to `B  ==  A` for any values of A and B (except for the order of applied conversions). The behavior for performing loose equality using `==` is as follows:

1. If the operands have the same type, they are compared as follows:
   - Object: return `true` only if both operands reference the same object.
   - String: return `true` only if both operands have the same characters in the same order.
   - Number: return `true` only if both operands have the same value. `+0` and `-0` are treated as the same value. If either operand is `NaN`, return `false`; so `NaN` is never equal to `NaN`.
   - Boolean: return true only if operands are both true or both false.
   - BigInt: return true only if both operands have the same value.
   - Symbol: return true only if both operands reference the same symbol.

2. If one of the operands is `null` or `undefined`, the other must also be `null` or `undefined` to return `true`. Otherwise return `false`.

3. If one of the operands is an object and the other is a primitive, convert the object to a primitive.

4. At this step, both operands are converted to primitives (one of String, Number, Boolean, Symbol, and BigInt). The rest of the conversion is done case-by-case.
   - If they are of the same type, compare them using step 1.
   - If one of the operands is a Symbol but the other is not, return `false`.
   - If one of the operands is a Boolean but the other is not, convert the boolean to a number: `true` is converted to 1, and `false` is converted to 0. Then compare the two operands loosely again.
   - Number to String: convert the string to a number. Conversion failure results in `NaN`, which will guarantee the equality to be `false`.
   - Number to BigInt: compare by their numeric value. If the number is ±Infinity or `NaN`, return `false`.
   - String to BigInt: convert the string to a BigInt using the same algorithm as the BigInt() constructor. If conversion fails, return `false`.

Traditionally, and according to ECMAScript, all primitives and objects are loosely unequal to `undefined` and `null`. But most browsers permit a very narrow class of objects (specifically, the document.all object for any page), in some contexts, to act as if they emulate the value `undefined`.

Loose equality is one such context: `null == A` and `undefined == A` evaluate to `true` if, and only if, `A` is an object that emulates `undefined`. In all other cases an object is never loosely equal to `undefined` or `null`.

In most cases, using loose equality is discouraged. The result of a comparison using strict equality is easier to predict, and may evaluate more quickly due to the lack of type coercion.

The following example demonstrates loose equality comparisons involving the number primitive `0`, the bigint primitive `0n`, the string primitive `'0'`, and an object whose `toString()` value is `'0'`.

```
const num = 0;
const big = 0n;
const str = "0";
const obj = new String("0");

console.log(num == str); // true
console.log(big == num); // true
console.log(str == big); // true

console.log(num == obj); // true
console.log(big == obj); // true
console.log(str == obj); // true
```

Loose equality is only used by the `==` operator.

Same-value equality using Object.is()

Same-value equality determines whether two values are functionally identical in all contexts. (This use case demonstrates an instance of the Liskov substitution principle.) One instance occurs when an attempt is made to mutate an immutable property:

```
// Add an immutable NEGATIVE_ZERO property to the Number constructor.
Object.defineProperty(Number, "NEGATIVE_ZERO", {
  value: -0,
  writable: false,
  configurable: false,
  enumerable: false,
});
function attemptMutation(v) {
  Object.defineProperty(Number, "NEGATIVE_ZERO", { value: v });
}
```

`Object.defineProperty` will throw an exception when attempting to change an immutable property, but it does nothing if no actual change is requested. If v is -0, no change has been requested, and no error will be thrown. Internally, when an immutable property is redefined, the newly-specified value is compared against the current value using same-value equality.

Same-value equality is provided by the `Object.is` method. It's used almost everywhere in the language where a value of equivalent identity is expected.

Same-value equality using Object.is()

Similar to same-value equality, but +0 and -0 are considered equal.

Same-value-zero equality is not exposed as a JavaScript API, but can be implemented with custom code:

```
function sameValueZero(x, y) {
  if (typeof x === "number" && typeof y === "number") {
    // x and y are equal (may be -0 and 0) or they are both NaN
    return x === y || (x !== x && y !== y);
  }
  return x === y;
}
```

Same-value-zero only differs from strict equality by treating `NaN` as equivalent, and only differs from same-value equality by treating -0 as equivalent to 0. This makes it usually have the most sensible behavior during searching, especially when working with `NaN`. It's used by `Array.prototype.includes()`, `TypedArray.prototype.includes()`, as well as `Map` and `Set` methods for comparing key equality.

Comparing equality methods

People often compare double equals and triple equals by saying one is an "enhanced" version of the other. For example, double equals could be said as an extended version of triple equals, because the former does everything that the latter does, but with type conversion on its operands — for example, 6 == "6". Alternatively, it can be claimed that double equals is the baseline, and triple equals is an enhanced version, because it requires the two operands to be the same type, so it adds an extra constraint.

However, this way of thinking implies that the equality comparisons form a one-dimensional "spectrum" where "totally strict" lies on one end and "totally loose" lies on the other. This model falls short with `Object.is`, because it isn't "looser" than double equals or "stricter" than triple equals, nor does it fit somewhere in between (i.e., being both stricter than double equals, but looser than triple equals). We can see from the sameness comparisons table below that this is due to the way that `Object.is` handles `NaN`. Notice that if `Object.is(NaN, NaN)` evaluated to `false`, we could say that it fits on the loose/strict spectrum as an even stricter form of triple equals, one that distinguishes between `-0` and `+0`. The `NaN` handling means this is untrue, however. Unfortunately, `Object.is` has to be thought of in terms of its specific characteristics, rather than its looseness or strictness with regard to the equality operators.

x	y	==	===	Object.is	SameValueZero
undefined	undefined	☑ true	☑ true	☑ true	☑ true
null	null	☑ true	☑ true	☑ true	☑ true
true	true	☑ true	☑ true	☑ true	☑ true
false	false	☑ true	☑ true	☑ true	☑ true
'foo'	'foo'	☑ true	☑ true	☑ true	☑ true
0	0	☑ true	☑ true	☑ true	☑ true
+0	-0	☑ true	☑ true	✘ false	☑ true
+0	0	☑ true	☑ true	☑ true	☑ true
-0	0	☑ true	☑ true	✘ false	☑ true
0n	-0n	☑ true	☑ true	☑ true	☑ true
0	false	☑ true	✘ false	✘ false	✘ false
""	false	☑ true	✘ false	✘ false	✘ false
""	0	☑ true	✘ false	✘ false	✘ false
'0'	0	☑ true	✘ false	✘ false	✘ false
'17'	17	☑ true	✘ false	✘ false	✘ false
[1, 2]	'1,2'	☑ true	✘ false	✘ false	✘ false
new String('foo')	'foo'	☑ true	✘ false	✘ false	✘ false
null	undefined	☑ true	✘ false	✘ false	✘ false
null	false	✘ false	✘ false	✘ false	✘ false
undefined	false	✘ false	✘ false	✘ false	✘ false
{ foo: 'bar' }	{ foo: 'bar' }	✘ false	✘ false	✘ false	✘ false
new String('foo')	new String('foo')	✘ false	✘ false	✘ false	✘ false
0	null	✘ false	✘ false	✘ false	✘ false
0	NaN	✘ false	✘ false	✘ false	✘ false
'foo'	NaN	✘ false	✘ false	✘ false	✘ false
NaN	NaN	✘ false	✘ false	☑ true	☑ true

When to use Object.is() versus triple equals

In general, the only time `Object.is`'s special behavior towards zeros is likely to be of interest is in the pursuit of certain meta-programming schemes, especially regarding property descriptors, when it is desirable for your work to mirror some of the characteristics of `Object.defineProperty`. If your use case does not require this, it is suggested to avoid `Object.is` and use `===` instead. Even if your requirements involve having comparisons between two `NaN` values evaluate to true, generally it is easier to special-case the `NaN` checks (using the `isNaN` method available from previous versions of ECMAScript) than it is to work out how surrounding computations might affect the sign of any zeros you encounter in your comparison.

Here's a non-exhaustive list of built-in methods and operators that might cause a distinction between `-0` and `+0` to manifest itself in your code:

- (unary negation)

    Consider the following example:

```
const stoppingForce = obj.mass * -obj.velocity;
```

    If `obj.velocity` is `0` (or computes to `0`), a `-0` is introduced at that place and propagates out into `stoppingForce`.

`Math.atan2, Math.ceil, Math.pow, Math.round`

    In some cases, it's possible for a `-0` to be introduced into an expression as a return value of these methods even when no `-0` exists as one of the parameters. For example, using `Math.pow` to raise `-Infinity` to the power of any negative, odd exponent evaluates to `-0`. Refer to the documentation for the individual methods.

`Math.floor, Math.max, Math.min, Math.sin, Math.sqrt, Math.tan`

    It's possible to get a `-0` return value out of these methods in some cases where a `-0` exists as one of the parameters. E.g., `Math.min(-0, +0)` evaluates to `-0`. Refer to the documentation for the individual methods.

`~, <<, >>`

Each of these operators uses the ToInt32 algorithm internally. Since there is only one representation for 0 in the internal 32-bit integer type, -0 will not survive a round trip after an inverse operation. E.g., both Object.is(~~(-0), -0) and Object.is(-0 << 2 >> 2, -0) evaluate to false.

Relying on `Object.is` when the signedness of zeros is not taken into account can be hazardous. Of course, when the intent is to distinguish between `-0` and `+0`, it does exactly what's desired.

Caveat: Object.is() and NaN

The `Object.is` specification treats all instances of `NaN` as the same object. However, since typed arrays are available, we can have distinct floating point representations of `NaN` which don't behave identically in all contexts. For example:

```
const f2b = (x) => new Uint8Array(new Float64Array([x]).buffer);
const b2f = (x) => new Float64Array(x.buffer)[0];
// Get a byte representation of NaN
const n = f2b(NaN);
// Change the first bit, which is the sign bit and doesn't matter for NaN
n[0] = 1;
const nan2 = b2f(n);
console.log(nan2); // NaN
console.log(Object.is(nan2, NaN)); // true
console.log(f2b(NaN)); // Uint8Array(8) [0, 0, 0, 0, 0, 0, 248, 127]
console.log(f2b(nan2)); // Uint8Array(8) [1, 0, 0, 0, 0, 0, 248, 127]
```

Enumerability and ownership of properties

Every property in JavaScript objects can be classified by three factors:

- Enumerable or non-enumerable;
- String or symbol;
- Own property or inherited property from the prototype chain.

Enumerable properties are those properties whose internal enumerable flag is set to true, which is the default for properties created via simple assignment or via a property initializer. Properties defined via `Object.defineProperty` and such are not enumerable by default. Most iteration means (such as `for...in` loops and `Object.keys`) only visit enumerable keys.

Ownership of properties is determined by whether the property belongs to the object directly and not to its prototype chain.

All properties, enumerable or not, string or symbol, own or inherited, can be accessed with dot notation or bracket notation. In this section, we will focus on JavaScript means that visit a group of object properties one-by-one.

Querying object properties

There are four built-in ways to query a property of an object. They all support both string and symbol keys. The following table summarizes when each method returns `true`.

	Enumerable, own	Enumerable, inherited	Non-enumerable, own	Non-enumerable, inherited
`propertyIsEnumerable()`	true ☑	false ✖	false ✖	false ✖
`hasOwnProperty()`	true ☑	false ✖	true ☑	false ✖
`Object.hasOwn()`	true ☑	false ✖	true ☑	false ✖
`in`	true ☑	true ☑	true ☑	true ☑

Traversing object properties

There are many methods in JavaScript that traverse a group of properties of an object. Sometimes, these properties are returned as an array; sometimes, they are iterated one-by-one in a loop; sometimes, they are used for constructing or mutating another object. The following table summarizes when a property may be visited.

Methods that only visit string properties or only symbol properties will have an extra note. ☑ means a property of this type will be visited; ✘ means it will not.

	Enumerable, own	Enumerable, inherited	Non-enumerable, own	Non-enumerable, inherited

`Object.keys`
`Object.values`
`Object.entries`
	☑
	(strings)
		✘	✘	✘

`Object.getOwnPropertyNames`	☑ (strings)	✘	☑ (strings)	✘
`Object.getOwnPropertySymbols`	☑ (symbols)	✘	☑ (symbols)	✘
`Object.getOwnPropertyDescriptors`	☑	✘	☑	✘
`Reflect.ownKeys`	☑	✘	☑	✘
`for...in`	☑ (strings)	☑ (strings)	✘	✘
`Object.assign` `(After the first parameter)`	☑	✘	✘	✘
`Object spread`	☑	✘	✘	✘

Obtaining properties by enumerability/ownership

Note that this is not the most efficient algorithm for all cases, but useful for a quick demonstration.

- Detection can occur by `SimplePropertyRetriever.theGetMethodYouWant(obj).includes(prop)`
- Iteration can occur by `SimplePropertyRetriever.theGetMethodYouWant(obj).forEach((value, prop) => {});` (or use `filter()`, `map()`, etc.)

```javascript
const SimplePropertyRetriever = {
  getOwnEnumerables(obj) {
    return this._getPropertyNames(obj, true, false, this._enumerable);
    // Or could use for...in filtered with Object.hasOwn or just this: return
    Object.keys(obj);
  },
  getOwnNonenumerables(obj) {
    return this._getPropertyNames(obj, true, false, this._notEnumerable);
  },
  getOwnEnumerablesAndNonenumerables(obj) {
    return this._getPropertyNames(
      obj,
      true,
      false,
      this._enumerableAndNotEnumerable,
    );
    // Or just use: return Object.getOwnPropertyNames(obj);
  },
  getPrototypeEnumerables(obj) {
    return this._getPropertyNames(obj, false, true, this._enumerable);
  },
  getPrototypeNonenumerables(obj) {
    return this._getPropertyNames(obj, false, true, this._notEnumerable);
  },
  getPrototypeEnumerablesAndNonenumerables(obj) {
    return this._getPropertyNames(
      obj,
      false,
      true,
      this._enumerableAndNotEnumerable,
    );
  },
  getOwnAndPrototypeEnumerables(obj) {
    return this._getPropertyNames(obj, true, true, this._enumerable);
    // Or could use unfiltered for...in
  },
  getOwnAndPrototypeNonenumerables(obj) {
    return this._getPropertyNames(obj, true, true, this._notEnumerable);
  },
  getOwnAndPrototypeEnumerablesAndNonenumerables(obj) {
    return this._getPropertyNames(
      obj,
      true,
      true,
      this._enumerableAndNotEnumerable,
    );
  },
  // Private static property checker callbacks
  _enumerable(obj, prop) {
```

278

```
      return Object.prototype.propertyIsEnumerable.call(obj, prop);
    },
    _notEnumerable(obj, prop) {
      return !Object.prototype.propertyIsEnumerable.call(obj, prop);
    },
    _enumerableAndNotEnumerable(obj, prop) {
      return true;
    },
    // Inspired by http://stackoverflow.com/a/8024294/271577
    _getPropertyNames(obj, iterateSelf, iteratePrototype, shouldInclude) {
      const props = [];
      do {
        if (iterateSelf) {
          Object.getOwnPropertyNames(obj).forEach((prop) => {
            if (props.indexOf(prop) === -1 && shouldInclude(obj, prop)) {
              props.push(prop);
            }
          });
        }
        if (!iteratePrototype) {
          break;
        }
        iterateSelf = true;
        obj = Object.getPrototypeOf(obj);
      } while (obj);
      return props;
    },
  };
```

This lesson was adapted from MDN Web Docs:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Enumerability_and_ownership_of_properties

Closures

A **closure** is the combination of a function bundled together (enclosed) with references to its surrounding state (the **lexical environment**). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

Lexical scoping

Consider the following example code:

```
function init() {
  var name = "Mozilla"; // name is a local variable created by init
  function displayName() {
    // displayName() is the inner function, that forms the closure
    console.log(name); // use variable declared in the parent function
  }
  displayName();
}
init();
```

init() creates a local variable called `name` and a function called `displayName()`. The `displayName()` function is an inner function that is defined inside `init()` and is available only within the body of the `init()` function. Note that the `displayName()` function has no local variables of its own. However, since inner functions have access to the variables of outer functions, `displayName()` can access the variable name declared in the parent function, `init()`.

If you run the code using you can notice that the `console.log()` statement within the `displayName()` function successfully displays the value of the `name` variable, which is declared in its parent function. This is an example of *lexical scoping*, which describes how a parser resolves variable names when functions are nested. The word *lexical* refers to the fact that lexical scoping uses the location where a variable is declared within the source code to determine where that variable is available. Nested functions have access to variables declared in their outer scope.

In this particular example, the scope is called a *function scope*, because the variable is accessible and only accessible within the function body where it's declared.

Scoping with let and const

Traditionally (before ES6), JavaScript only had two kinds of scopes: function scope and global scope. Variables declared with `var` are either function-scoped or global-scoped, depending on whether they are declared within a function or outside a function. This can be tricky, because blocks with curly braces do not create scopes:

```
if (Math.random() > 0.5) {
  var x = 1;
} else {
  var x = 2;
}
console.log(x);
```

For people from other languages (e.g. C, Java) where blocks create scopes, the above code should throw an error on the `console.log` line, because we are outside the scope of `x` in either block. However, because blocks don't create scopes for `var`, the var statements here actually create a global variable. There is also a practical example introduced below that illustrates how this can cause actual bugs when combined with closures – see "Creating closures in loops: A common mistake" section .

In ES6, JavaScript introduced the `let` and `const` declarations, which, among other things like temporal dead zones, allow you to create block-scoped variables.

```
if (Math.random() > 0.5) {
  const x = 1;
} else {
  const x = 2;
}
console.log(x); // ReferenceError: x is not defined
```

In essence, blocks are finally treated as scopes in ES6, but only if you declare variables with let or const. In addition, ES6 introduced modules, which introduced another kind of scope. Closures are able to capture variables in all these scopes, which we will introduce later.

Closure

Consider the following example code:

```
function makeFunc() {
  const name = "Mozilla";
  function displayName() {
    console.log(name);
  }
  return displayName;
}
const myFunc = makeFunc();
myFunc();
```

Running this code has exactly the same effect as the previous example of the `init()` function above. What's different (and interesting) is that the `displayName()` inner function is returned from the outer function before being executed.

At first glance, it might seem unintuitive that this code still works. In some programming languages, the local variables within a function exist for just the duration of that function's execution. Once `makeFunc()` finishes executing, you might expect that the name variable would no longer be accessible. However, because the code still works as expected, this is obviously not the case in JavaScript.

The reason is that functions in JavaScript form closures. A *closure* is the combination of a function and the lexical environment within which that function was declared. This environment consists of any local variables that were in-scope at the time the closure was created. In this case, `myFunc` is a reference to the instance of the function `displayName` that is created when `makeFunc` is run. The instance of `displayName` maintains a reference to its lexical environment, within which the variable name exists. For this reason, when `myFunc` is invoked, the variable `name` remains available for use, and "Mozilla" is passed to `console.log`.

Here's a slightly more interesting example—a `makeAdder` function:

```
function makeAdder(x) {
  return function (y) {
    return x + y;
  };
}

const add5 = makeAdder(5);
```

```
const add10 = makeAdder(10);

console.log(add5(2)); // 7
console.log(add10(2)); // 12
```

In this example, we have defined a function `makeAdder(x)`, that takes a single argument x, and returns a new function. The function it returns takes a single argument y, and returns the sum of x and y.

In essence, `makeAdder` is a function factory. It creates functions that can add a specific value to their argument. In the above example, the function factory creates two new functions—one that adds five to its argument, and one that adds 10.

`add5` and `add10` both form closures. They share the same function body definition, but store different lexical environments. In `add5`'s lexical environment, x is 5, while in the lexical environment for `add10`, x is 10.

Practical closures

Closures are useful because they let you associate data (the lexical environment) with a function that operates on that data. This has obvious parallels to object-oriented programming, where objects allow you to associate data (the object's properties) with one or more methods.

Consequently, you can use a closure anywhere that you might normally use an object with only a single method.

Situations where you might want to do this are particularly common on the web. Much of the code written in front-end JavaScript is event-based. You define some behavior, and then attach it to an event that is triggered by the user (such as a click or a keypress). The code is attached as a callback (a single function that is executed in response to the event).

For instance, suppose we want to add buttons to a page to adjust the text size. One way of doing this is to specify the font-size of the `body` element (in pixels), and then set the size of the other elements on the page (such as headers) using the relative `em` unit:

```
body {
  font-family: Helvetica, Arial, sans-serif;
  font-size: 12px;
}
```

```
h1 {
  font-size: 1.5em;
}

h2 {
  font-size: 1.2em;
}
```

Such interactive text size buttons can change the font-size property of the body element, and the adjustments are picked up by other elements on the page thanks to the relative units.

Here's the JavaScript:

```
function makeSizer(size) {
  return function () {
    document.body.style.fontSize = `${size}px`;
  };
}

const size12 = makeSizer(12);
const size14 = makeSizer(14);
const size16 = makeSizer(16);
```

size12, size14, and size16 are now functions that resize the body text to 12, 14, and 16 pixels, respectively. You can attach them to buttons (in this case hyperlinks) as demonstrated in the following code example.

```
document.getElementById("size-12").onclick = size12;
document.getElementById("size-14").onclick = size14;
document.getElementById("size-16").onclick = size16;
```

```
<button id="size-12">12</button>
<button id="size-14">14</button>
<button id="size-16">16</button>
```

Emulating private methods with closures

Languages such as Java allow you to declare methods as private, meaning that they can be called only by other methods in the same class.

JavaScript, prior to classes, didn't have a native way of declaring private methods, but it was possible to emulate private methods using closures. Private methods aren't just useful for restricting access to code. They also provide a powerful way of managing your global namespace.

The following code illustrates how to use closures to define public functions that can access private functions and variables. Note that these closures follow the Module Design Pattern.

```javascript
const counter = (function () {
  let privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }

  return {
    increment() {
      changeBy(1);
    },

    decrement() {
      changeBy(-1);
    },

    value() {
      return privateCounter;
    },
  };
})();

console.log(counter.value()); // 0.

counter.increment();
counter.increment();
console.log(counter.value()); // 2.

counter.decrement();
console.log(counter.value()); // 1.
```

In previous examples, each closure had its own lexical environment. Here though, there is a single lexical environment that is shared by the three functions: `counter.increment`, `counter.decrement`, and `counter.value`.

The shared lexical environment is created in the body of an anonymous function, which is executed as soon as it has been defined (also known as an IIFE). The lexical environment contains two private items: a variable called `privateCounter`, and a function called `changeBy`. You can't access either of these private members from outside the anonymous function. Instead, you can access them using the three public functions that are returned from the anonymous wrapper.

Those three public functions form closures that share the same lexical environment. Thanks to JavaScript's lexical scoping, they each have access to the `privateCounter` variable and the `changeBy` function.

```
const makeCounter = function () {
  let privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment() {
      changeBy(1);
    },

    decrement() {
      changeBy(-1);
    },

    value() {
      return privateCounter;
    },
  };
};

const counter1 = makeCounter();
const counter2 = makeCounter();
console.log(counter1.value()); // 0.
counter1.increment();
counter1.increment();
console.log(counter1.value()); // 2.
counter1.decrement();
console.log(counter1.value()); // 1.
console.log(counter2.value()); // 0.
```

Notice how the two counters maintain their independence from one another. Each closure references a different version of the `privateCounter` variable through its own closure. Each time one of the counters is called, its lexical environment changes by changing the value of this variable. Changes to the variable value in one closure don't affect the value in the other closure.

Note: Using closures in this way provides benefits that are normally associated with object-oriented programming. In particular, data hiding and encapsulation.

Closure scope chain

Every closure has three scopes:

- Local scope (Own scope)
- Enclosing scope (can be block, function, or module scope)
- Global scope

A common mistake is not realizing that in the case where the outer function is itself a nested function, access to the outer function's scope includes the enclosing scope of the outer function— effectively creating a chain of function scopes. To demonstrate, consider the following example code.

```
// global scope
const e = 10;
function sum(a) {
  return function (b) {
    return function (c) {
      // outer functions scope
      return function (d) {
        // local scope
        return a + b + c + d + e;
      };
    };
  };
}

console.log(sum(1)(2)(3)(4)); // 20
```

You can also write without anonymous functions:

```
// global scope
const e = 10;
function sum(a) {
  return function sum2(b) {
    return function sum3(c) {
      // outer functions scope
      return function sum4(d) {
        // local scope
        return a + b + c + d + e;
      };
    };
  };
}

const sum2 = sum(1);
const sum3 = sum2(2);
const sum4 = sum3(3);
const result = sum4(4);
console.log(result); // 20
```

In the example above, there's a series of nested functions, all of which have access to the outer functions' scope. In this context, we can say that closures have access to all outer function scopes.

Closures can capture variables in block scopes and module scopes as well. For example, the following creates a closure over the block-scoped variable y:

```
function outer() {
  const x = 5;
  if (Math.random() > 0.5) {
    const y = 6;
    return () => console.log(x, y);
  }
}

outer()(); // Logs 5 6
```

Closures over modules can be more interesting.

```
// myModule.js
let x = 5;
export const getX = () => x;
export const setX = (val) => {
  x = val;
};
```

Here, the module exports a pair of getter-setter functions, which close over the module-scoped variable x. Even when x is not directly accessible from other modules, it can be read and written with the functions.

```
import { getX, setX } from "./myModule.js";

console.log(getX()); // 5
setX(6);
console.log(getX()); // 6
```

Closures can close over imported values as well, which are regarded as live bindings, because when the original value changes, the imported one changes accordingly.

```
// myModule.js
export let x = 1;
export const setX = (val) => {
  x = val;
};
```

```
// closureCreator.js
import { x } from "./myModule.js";
export const getX = () => x; // Close over an imported live binding
```

```
import { getX } from "./closureCreator.js";
import { setX } from "./myModule.js";

console.log(getX()); // 1
setX(2);
console.log(getX()); // 2
```

Creating closures in loops: A common mistake

Prior to the introduction of the `let` keyword, a common problem with closures occurred when you created them inside a loop. To demonstrate, consider the following example code.

```
<p id="help">Helpful notes will appear here</p>
```

```
<p>Email: <input type="text" id="email" name="email" /></p>
<p>Name: <input type="text" id="name" name="name" /></p>
<p>Age: <input type="text" id="age" name="age" /></p>
```

```
function showHelp(help) {
  document.getElementById("help").textContent = help;
}

function setupHelp() {
  var helpText = [
    { id: "email", help: "Your email address" },
    { id: "name", help: "Your full name" },
    { id: "age", help: "Your age (you must be over 16)" },
  ];

  for (var i = 0; i < helpText.length; i++) {
    // Culprit is the use of `var` on this line
    var item = helpText[i];
    document.getElementById(item.id).onfocus = function () {
      showHelp(item.help);
    };
  }
}

setupHelp();
```

The `helpText` array defines three helpful hints, each associated with the ID of an input field in the document. The loop cycles through these definitions, hooking up an `onfocus` event to each one that shows the associated help method.

If you try this code out, you'll see that it doesn't work as expected. No matter what field you focus on, the message about your age will be displayed.

The reason for this is that the functions assigned to `onfocus` form closures; they consist of the function definition and the captured environment from the `setupHelp` function's scope. Three closures have been created by the loop, but each one shares the same single lexical environment, which has a variable with changing values (`item`). This is because the variable item is declared with var and thus has function scope due to hoisting. The value of `item.help` is determined when the `onfocus` callbacks are executed. Because the loop has already run its course by that time, the item variable object (shared by all three closures) has been left pointing to the last entry in the `helpText` list.

One solution in this case is to use more closures: in particular, to use a function factory as described earlier:

```
function showHelp(help) {
  document.getElementById("help").textContent = help;
}

function makeHelpCallback(help) {
  return function () {
    showHelp(help);
  };
}

function setupHelp() {
  var helpText = [
    { id: "email", help: "Your email address" },
    { id: "name", help: "Your full name" },
    { id: "age", help: "Your age (you must be over 16)" },
  ];

  for (var i = 0; i < helpText.length; i++) {
    var item = helpText[i];
    document.getElementById(item.id).onfocus = makeHelpCallback(item.help);
  }
}

setupHelp();
```

This works as expected. Rather than the callbacks all sharing a single lexical environment, the makeHelpCallback function creates a new lexical environment for each callback, in which help refers to the corresponding string from the helpText array.

One other way to write the above using anonymous closures is:

```
function showHelp(help) {
  document.getElementById("help").textContent = help;
}

function setupHelp() {
  var helpText = [
    { id: "email", help: "Your email address" },
    { id: "name", help: "Your full name" },
    { id: "age", help: "Your age (you must be over 16)" },
  ];

  for (var i = 0; i < helpText.length; i++) {
```

```
    (function () {
      var item = helpText[i];
      document.getElementById(item.id).onfocus = function () {
        showHelp(item.help);
      };
    })(); // Immediate event listener attachment with the current value of item
          (preserved until iteration).
  }
}

setupHelp();
```

If you don't want to use more closures, you can use the `let` or `const` keyword:

```
function showHelp(help) {
  document.getElementById("help").textContent = help;
}

function setupHelp() {
  const helpText = [
    { id: "email", help: "Your email address" },
    { id: "name", help: "Your full name" },
    { id: "age", help: "Your age (you must be over 16)" },
  ];

  for (let i = 0; i < helpText.length; i++) {
    const item = helpText[i];
    document.getElementById(item.id).onfocus = () => {
      showHelp(item.help);
    };
  }
}

setupHelp();
```

This example uses `const` instead of `var`, so every closure binds the block-scoped variable, meaning that no additional closures are required.

Another alternative could be to use `forEach()` to iterate over the `helpText` array and attach a listener to each `<input>`, as shown:

```
function showHelp(help) {
  document.getElementById("help").textContent = help;
}
```

```
function setupHelp() {
  var helpText = [
    { id: "email", help: "Your email address" },
    { id: "name", help: "Your full name" },
    { id: "age", help: "Your age (you must be over 16)" },
  ];

  helpText.forEach(function (text) {
    document.getElementById(text.id).onfocus = function () {
      showHelp(text.help);
    };
  });
}

setupHelp();
```

Performance considerations

As mentioned previously, each function instance manages its own scope and closure. Therefore, it is unwise to unnecessarily create functions within other functions if closures are not needed for a particular task, as it will negatively affect script performance both in terms of processing speed and memory consumption.

For instance, when creating a new object/class, methods should normally be associated to the object's prototype rather than defined into the object constructor. The reason is that whenever the constructor is called, the methods would get reassigned (that is, for every object creation).

Consider the following case:

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
  this.getName = function () {
    return this.name;
  };

  this.getMessage = function () {
    return this.message;
  };
}
```

Because the previous code does not take advantage of the benefits of using closures in this particular instance, we could instead rewrite it to avoid using closures as follows:

```javascript
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
}
MyObject.prototype = {
  getName() {
    return this.name;
  },
  getMessage() {
    return this.message;
  },
};
```

However, redefining the prototype is not recommended. The following example instead appends to the existing prototype:

```javascript
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
}
MyObject.prototype.getName = function () {
  return this.name;
};
MyObject.prototype.getMessage = function () {
  return this.message;
};
```

In the two previous examples, the inherited prototype can be shared by all objects and the method definitions need not occur at every object creation. See Inheritance and the prototype chain for more.

Inheritance and the prototype chain

Inheritance and the prototype chain are important concepts in JavaScript that allow objects to inherit properties and methods from other objects.

In JavaScript, every object has a hidden property called [[Prototype]] which points to another object, its prototype. When you try to access a property or method on an object that doesn't exist on that object, JavaScript looks for it on its prototype, and if it's not found there, on the prototype's prototype, and so on up the chain, until the property is either found or the end of the chain is reached.

So, when it comes to inheritance, JavaScript only has one construct: objects. Each object has a private property which holds a link to another object called its prototype. That prototype object has a prototype of its own, and so on until an object is reached with `null` as its prototype. By definition, `null` has no prototype, and acts as the final link in this prototype chain. It is possible to mutate any member of the prototype chain or even swap out the prototype at runtime, so concepts like static dispatching do not exist in JavaScript.

Inheritance with the prototype chain

Inheriting properties

In JavaScript, an object is like a container for properties, which are like variables or values that belong to the object. These properties can be accessed using dot notation or square brackets.

Every object in JavaScript has a hidden link to another object, which is called its prototype. This prototype object can also have its own properties.

When you try to access a property of an object, JavaScript first looks for that property on the object itself. If the property is not found on the object, JavaScript then looks for the property on the object's prototype. If the property is still not found, it looks on the prototype's prototype, and so on, until either the property is found or the end of the prototype chain is reached.

So, when you use dot notation or square brackets to access a property on an object, JavaScript checks the object itself and then checks up the prototype chain until it finds the property or reaches the end of the chain.

There are several ways to specify the `[[Prototype]]` of an object, which are listed in a later section. For now, we will use the __proto__ syntax for illustration. It's worth noting that the { __proto__:

... } syntax is different from the `obj.__proto__` accessor: the former is standard and not deprecated. In an object literal like { `a: 1, b: 2, __proto__: c` }, the value of `c` (which has to be either `null` or another object) will become the `[[Prototype]]` of the object represented by the literal, while the other keys like `a` and `b` will become the own properties of the object. This syntax reads very naturally, since `[[Prototype]]` is just an "internal property" of the object.

Here is what happens when trying to access a property:

```
const o = {
  a: 1,
  b: 2,
  // __proto__ sets the [[Prototype]]. It's specified as another object literal.
  __proto__: {
    b: 3,
    c: 4,
  },
};

// o.[[Prototype]] has properties b and c.
// o.[[Prototype]].[[Prototype]] is Object.prototype (we will explain
// what that means later).
// Finally, o.[[Prototype]].[[Prototype]].[[Prototype]] is null.
// This is the end of the prototype chain, as null,
// by definition, has no [[Prototype]].
// Thus, the full prototype chain looks like:
// { a: 1, b: 2 } ---> { b: 3, c: 4 } ---> Object.prototype ---> null

console.log(o.a); // 1
// Is there an 'a' own property on o? Yes, and its value is 1.

console.log(o.b); // 2
// Is there a 'b' own property on o? Yes, and its value is 2.
// The prototype also has a 'b' property, but it's not visited.
// This is called Property Shadowing

console.log(o.c); // 4
// Is there a 'c' own property on o? No, check its prototype.
// Is there a 'c' own property on o.[[Prototype]]? Yes, its value is 4.

console.log(o.d); // undefined
// Is there a 'd' own property on o? No, check its prototype.
// Is there a 'd' own property on o.[[Prototype]]? No, check its prototype.
// o.[[Prototype]].[[Prototype]] is Object.prototype and
// there is no 'd' property by default, check its prototype.
// o.[[Prototype]].[[Prototype]].[[Prototype]] is null, stop searching,
// no property found, return undefined.
```

Setting a property to an object creates an own property. The only exception to the getting and setting behavior rules is when it's intercepted by a getter or setter.

Similarly, you can create longer prototype chains, and a property will be sought on all of them.

```
const o = {
  a: 1,
  b: 2,
  // __proto__ sets the [[Prototype]]. It's specified here
  // as another object literal.
  __proto__: {
    b: 3,
    c: 4,
    __proto__: {
      d: 5,
    },
  },
};

// { a: 1, b: 2 } ---> { b: 3, c: 4 } ---> { d: 5 } ---> Object.prototype ---> null

console.log(o.d); // 5
```

Inheriting "methods"

JavaScript does not have "methods" in the form that class-based languages define them. In JavaScript, any function can be added to an object in the form of a property. An inherited function acts just as any other property, including property shadowing as shown above (in this case, a form of method overriding).

When an inherited function is executed, the value of this points to the inheriting object, not to the prototype object where the function is an own property.

```
const parent = {
  value: 2,
  method() {
    return this.value + 1;
  },
};

console.log(parent.method()); // 3
// When calling parent.method in this case, 'this' refers to parent
```

```
  // child is an object that inherits from parent
  const child = {
    __proto__: parent,
  };
  console.log(child.method()); // 3
  // When child.method is called, 'this' refers to child.
  // So when child inherits the method of parent,
  // The property 'value' is sought on child. However, since child
  // doesn't have an own property called 'value', the property is
  // found on the [[Prototype]], which is parent.value.

  child.value = 4; // assign the value 4 to the property 'value' on child.
  // This shadows the 'value' property on parent.
  // The child object now looks like:
  // { value: 4, __proto__: { value: 2, method: [Function] } }
  console.log(child.method()); // 5
  // Since child now has the 'value' property, 'this.value' means
  // child.value instead
```

Constructors

The power of prototypes is that we can reuse a set of properties if they should be present on every instance — especially for methods. Suppose we are to create a series of boxes, where each box is an object that contains a value which can be accessed through a `getValue` function. A naive implementation would be:

```
const boxes = [
  { value: 1, getValue() { return this.value; } },
  { value: 2, getValue() { return this.value; } },
  { value: 3, getValue() { return this.value; } },
];
```

This is subpar, because each instance has its own function property that does the same thing, which is redundant and unnecessary. Instead, we can move `getValue` to the `[[Prototype]]` of all boxes:

```
const boxPrototype = {
  getValue() {
    return this.value;
  },
};
```

298

```
  const boxes = [
    { value: 1, __proto__: boxPrototype },
    { value: 2, __proto__: boxPrototype },
    { value: 3, __proto__: boxPrototype },
  ];
```

This way, all boxes' `getValue` method will refer to the same function, lowering memory usage. However, manually binding the __proto__ for every object creation is still very inconvenient. This is when we would use a constructor function, which automatically sets the `[[Prototype]]` for every object manufactured. Constructors are functions called with `new`.

```
  // A constructor function
  function Box(value) {
    this.value = value;
  }

  // Properties all boxes created from the Box() constructor
  // will have
  Box.prototype.getValue = function () {
    return this.value;
  };

  const boxes = [new Box(1), new Box(2), new Box(3)];
```

We say that `new Box(1)` is an instance created from the `Box` constructor function. `Box.prototype` is not much different from the `boxPrototype` object we created previously — it's just a plain object. Every instance created from a constructor function will automatically have the constructor's prototype property as its `[[Prototype]]` — that is, `Object.getPrototypeOf(new Box())` `===` `Box.prototype`. `Constructor.prototype` by default has one own property: `constructor`, which references the constructor function itself — that is, `Box.prototype.constructor` `===` `Box`. This allows one to access the original constructor from any instance.

Note: If a non-primitive is returned from the constructor function, that value will become the result of the `new` expression. In this case the `[[Prototype]]` may not be correctly bound — but this should not happen much in practice.

The above constructor function can be rewritten in <u>classes</u> as:

```
  class Box {
    constructor(value) {
      this.value = value;
    }
```

```
  // Methods are created on Box.prototype
  getValue() {
    return this.value;
  }
}
```

Classes are syntax sugar over constructor functions, which means you can still manipulate `Box.prototype` to change the behavior of all instances. However, because classes are designed to be an abstraction over the underlying prototype mechanism, we will use the more-lightweight constructor function syntax for this tutorial to fully demonstrate how prototypes work.

Because `Box.prototype` references the same object as the `[[Prototype]]` of all instances, we can change the behavior of all instances by mutating `Box.prototype`.

```
function Box(value) {
  this.value = value;
}
Box.prototype.getValue = function () {
  return this.value;
};
const box = new Box(1);

// Mutate Box.prototype after an instance has already been created
Box.prototype.getValue = function () {
  return this.value + 1;
};
box.getValue(); // 2
```

A corollary is, *re-assigning* `Constructor.prototype` (`Constructor.prototype = ...`) is a bad idea for two reasons:

- The `[[Prototype]]` of instances created before the reassignment is now referencing a different object from the `[[Prototype]]` of instances created after the reassignment — mutating one's `[[Prototype]]` no longer mutates the other.
- Unless you manually re-set the `constructor` property, the constructor function can no longer be traced from `instance.constructor`, which may break user expectation. Some built-in operations will read the `constructor` property as well, and if it is not set, they may not work as expected.

`Constructor.prototype` is only useful when constructing instances. It has nothing to do with `Constructor.[[Prototype]]`, which is the constructor function's *own* prototype, which is `Function.prototype` - that is `Object.getPrototypeOf(Constructor) === Function.prototype`.

Implicit constructors of literals

Some literal syntaxes in JavaScript create instances that implicitly set the [[Prototype]]. For example:

```
// Object literals (without the `__proto__` key) automatically
// have `Object.prototype` as their `[[Prototype]]`
const object = { a: 1 };
Object.getPrototypeOf(object) === Object.prototype; // true

// Array literals automatically have `Array.prototype` as their `[[Prototype]]`
const array = [1, 2, 3];
Object.getPrototypeOf(array) === Array.prototype; // true

// RegExp literals automatically have `RegExp.prototype` as their `[[Prototype]]`
const regexp = /abc/;
Object.getPrototypeOf(regexp) === RegExp.prototype; // true
```

We can "de-sugar" them into their constructor form.

```
const array = new Array(1, 2, 3);
const regexp = new RegExp("abc");
```

For example, "array methods" like `map()` are simply methods defined on `Array.prototype`, which is why they are automatically available on all array instances.

Warning: There is one misfeature that used to be prevalent — extending `Object.prototype` or one of the other built-in prototypes. An example of this misfeature is, defining `Array.prototype.myMethod = function () {...}` and then using `myMethod` on all array instances.

This misfeature is called *monkey patching*. Doing monkey patching risks forward compatibility, because if the language adds this method in the future but with a different signature, your code will break. It has led to incidents like the SmooshGate, and can be a great nuisance for the language to advance since JavaScript tries to "not break the web".

The **only** good reason for extending a built-in prototype is to backport the features of newer JavaScript engines, like `Array.prototype.forEach`.

It may be interesting to note that due to historical reasons, some built-in constructors' prototype property are instances themselves. For example, `Number.prototype` is a number `0`, `Array.prototype` is an empty array, and `RegExp.prototype` is `/(?:)/`.

```
Number.prototype + 1; // 1
Array.prototype.map((x) => x + 1); // []
String.prototype + "a"; // "a"
RegExp.prototype.source; // "(?:)"
Function.prototype(); // Function.prototype is a no-op function by itself
```

However, this is not the case for user-defined constructors, nor for modern constructors like `Map`.

```
Map.prototype.get(1);
// Uncaught TypeError: get method called on incompatible Map.prototype
```

Building longer inheritance chains

The `Constructor.prototype` property will become the `[[Prototype]]` of the constructor's instances, as-is — including `Constructor.prototype`'s own `[[Prototype]]`. By default, `Constructor.prototype` is a *plain* object - that is, `Object.getPrototypeOf(Constructor.prototype)` `=== Object.prototype`. The only exception is `Object.prototype` itself, whose `[[Prototype]]` is `null` — that is, `Object.getPrototypeOf(Object.prototype) === null`. Therefore, a typical constructor will build the following prototype chain:

```
function Constructor() {}

const obj = new Constructor();
// obj ---> Constructor.prototype ---> Object.prototype ---> null
```

To build longer prototype chains, we can set the `[[Prototype]]` of `Constructor.prototype` via the `Object.setPrototypeOf()` function.

```
function Base() {}
function Derived() {}
```

```
// Set the `[[Prototype]]` of `Derived.prototype`
// to `Base.prototype`
Object.setPrototypeOf(Derived.prototype, Base.prototype);

const obj = new Derived();
// obj ---> Derived.prototype ---> Base.prototype ---> Object.prototype ---> null
```

In class terms, this is equivalent to using the extends syntax.

```
class Base {}
class Derived extends Base {}


const obj = new Derived();
// obj ---> Derived.prototype ---> Base.prototype ---> Object.prototype ---> null
```

You may also see some legacy code using Object.create() to build the inheritance chain. However, because this reassigns the prototype property and removes the constructor property, it can be more error-prone, while performance gains may not be apparent if the constructors haven't created any instances yet.

```
function Base() {}
function Derived() {}
// Re-assigns `Derived.prototype` to a new object
// with `Base.prototype` as its `[[Prototype]]`
// DON'T DO THIS — use Object.setPrototypeOf to mutate it instead
Derived.prototype = Object.create(Base.prototype);
```

Inspecting prototypes: a deeper dive

Let's look at what happens behind the scenes in a bit more detail. In JavaScript, as mentioned above, functions are able to have properties. All functions have a special property named prototype. For the best learning experience, it is highly recommended that you open a console, navigate to the "console" tab, copy-and-paste in the below JavaScript code, and run it by pressing the Enter/Return key.

```
function doSomething() {}
console.log(doSomething.prototype);
// It does not matter how you declare the function; a
```

```
// function in JavaScript will always have a default
// prototype property — with one exception: an arrow
// function doesn't have a default prototype property:
const doSomethingFromArrowFunction = () => {};
console.log(doSomethingFromArrowFunction.prototype);
```

As seen above, doSomething() has a default prototype property, as demonstrated by the console. After running this code, the console should have displayed an object that looks similar to this.

```
{
  constructor: ƒ doSomething(),
  [[Prototype]]: {
    constructor: ƒ Object(),
    hasOwnProperty: ƒ hasOwnProperty(),
    isPrototypeOf: ƒ isPrototypeOf(),
    propertyIsEnumerable: ƒ propertyIsEnumerable(),
    toLocaleString: ƒ toLocaleString(),
    toString: ƒ toString(),
    valueOf: ƒ valueOf()
  }
}
```

Note: The Chrome console uses [[Prototype]] to denote the object's prototype, following the spec's terms; Firefox uses <prototype>. For consistency we will use [[Prototype]].

We can add properties to the prototype of doSomething(), as shown below.

```
function doSomething() {}
doSomething.prototype.foo = "bar";
console.log(doSomething.prototype);
```

This results in:

```
{
  foo: "bar",
  constructor: ƒ doSomething(),
  [[Prototype]]: {
    constructor: ƒ Object(),
    hasOwnProperty: ƒ hasOwnProperty(),
    isPrototypeOf: ƒ isPrototypeOf(),
    propertyIsEnumerable: ƒ propertyIsEnumerable(),
    toLocaleString: ƒ toLocaleString(),
```

```
      toString: ƒ toString(),
      valueOf: ƒ valueOf()
    }
  }
}
```

We can now use the new operator to create an instance of doSomething() based on this prototype. To use the new operator, call the function normally except prefix it with new. Calling a function with the new operator returns an object that is an instance of the function. Properties can then be added onto this object.

Try the following code:

```
function doSomething() {}
doSomething.prototype.foo = "bar"; // add a property onto the prototype
const doSomeInstancing = new doSomething();
doSomeInstancing.prop = "some value"; // add a property onto the object
console.log(doSomeInstancing);
```

This results in an output similar to the following:

```
{
  prop: "some value",
  [[Prototype]]: {
    foo: "bar",
    constructor: ƒ doSomething(),
    [[Prototype]]: {
      constructor: ƒ Object(),
      hasOwnProperty: ƒ hasOwnProperty(),
      isPrototypeOf: ƒ isPrototypeOf(),
      propertyIsEnumerable: ƒ propertyIsEnumerable(),
      toLocaleString: ƒ toLocaleString(),
      toString: ƒ toString(),
      valueOf: ƒ valueOf()
    }
  }
}
```

As seen above, the [[Prototype]] of doSomeInstancing is doSomething.prototype. But, what does this do? When you access a property of doSomeInstancing, the runtime first looks to see if doSomeInstancing has that property.

If `doSomeInstancing` does not have the property, then the runtime looks for the property in `doSomeInstancing.[[Prototype]]` (a.k.a. `doSomething.prototype`). If `doSomeInstancing.[[Prototype]]` has the property being looked for, then that property on `doSomeInstancing.[[Prototype]]` is used.

Otherwise, if `doSomeInstancing.[[Prototype]]` does not have the property, then `doSomeInstancing.[[Prototype]].[[Prototype]]` is checked for the property. By default, the `[[Prototype]]` of any function's prototype property is `Object.prototype`. So, `doSomeInstancing.[[Prototype]].[[Prototype]]` (a.k.a. `doSomething.prototype.[[Prototype]]` (a.k.a. `Object.prototype`)) is then looked through for the property being searched for.

If the property is not found in `doSomeInstancing.[[Prototype]].[[Prototype]]`, then `doSomeInstancing.[[Prototype]].[[Prototype]].[[Prototype]]` is looked through. However, there is a problem: `doSomeInstancing.[[Prototype]].[[Prototype]].[[Prototype]]` does not exist, because `Object.prototype.[[Prototype]]` is `null`. Then, and only then, after the entire prototype chain of [[Prototype]]'s is looked through, the runtime asserts that the property does not exist and conclude that the value at the property is undefined.

Let's try entering some more code into the console:

```
function doSomething() {}
doSomething.prototype.foo = "bar";
const doSomeInstancing = new doSomething();
doSomeInstancing.prop = "some value";
console.log("doSomeInstancing.prop:     ", doSomeInstancing.prop);
console.log("doSomeInstancing.foo:      ", doSomeInstancing.foo);
console.log("doSomething.prop:          ", doSomething.prop);
console.log("doSomething.foo:           ", doSomething.foo);
console.log("doSomething.prototype.prop:", doSomething.prototype.prop);
console.log("doSomething.prototype.foo: ", doSomething.prototype.foo);
```

This results in the following:

```
doSomeInstancing.prop:      some value
doSomeInstancing.foo:       bar
doSomething.prop:           undefined
doSomething.foo:            undefined
doSomething.prototype.prop: undefined
doSomething.prototype.foo:  bar
```

Different ways of creating and mutating prototype chains

We have encountered many ways to create objects and change their prototype chains. We will systematically summarize the different ways, comparing each approach's pros and cons.

Objects created with syntax constructs

```
const o = { a: 1 };
// The newly created object o has Object.prototype as its [[Prototype]]
// Object.prototype has null as its prototype.
// o ---> Object.prototype ---> null

const b = ["yo", "whadup", "?"];
// Arrays inherit from Array.prototype
// (which has methods indexOf, forEach, etc.)
// The prototype chain looks like:
// b ---> Array.prototype ---> Object.prototype ---> null

function f() {
  return 2;
}
// Functions inherit from Function.prototype
// (which has methods call, bind, etc.)
// f ---> Function.prototype ---> Object.prototype ---> null

const p = { b: 2, __proto__: o };
// It is possible to point the newly created object's [[Prototype]] to
// another object via the __proto__ literal property. (Not to be confused
// with Object.prototype.__proto__ accessors)
// p ---> o ---> Object.prototype ---> null
```

Pros and cons of using the __proto__ key in object initializers

Pro(s)	Supported in all modern engines. Pointing the __proto__ key to something that is not an object only fails silently without throwing an exception. Contrary to the `Object.prototype.__proto__` setter, __proto__ in object literal initializers is standardized and optimized, and can even be more performant than `Object.create`. Declaring extra own properties on the object at creation is more ergonomic than `Object.create`.
Con(s)	Not supported in IE10 and below. Likely to be confused with `Object.prototype.__proto__` accessors for people unaware of the difference.

With constructor functions

```
function Graph() {
  this.vertices = [];
  this.edges = [];
}

Graph.prototype.addVertex = function (v) {
  this.vertices.push(v);
};

const g = new Graph();
// g is an object with own properties 'vertices' and 'edges'.
// g.[[Prototype]] is the value of Graph.prototype when new Graph() is executed.
```

Pros and cons of using constructor functions

Pro(s)	Supported in all engines — going all the way back to IE 5.5. Also, it is very fast, very standard, and very JIT-optimizable.
Con(s)	- In order to use this method, the function in question must be initialized. During this initialization, the constructor may store unique information that must be generated per-object. This unique information would only be generated once, potentially leading to problems. - The initialization of the constructor may put unwanted methods onto the object. Both of those are generally not problems in practice.

With Object.create()

Calling Object.create() creates a new object. The [[Prototype]] of this object is the first argument of the function:

```
const a = { a: 1 };
// a ---> Object.prototype ---> null

const b = Object.create(a);
// b ---> a ---> Object.prototype ---> null
console.log(b.a); // 1 (inherited)

const c = Object.create(b);
// c ---> b ---> a ---> Object.prototype ---> null
```

```
const d = Object.create(null);
// d ---> null (d is an object that has null directly as its prototype)
console.log(d.hasOwnProperty);
// undefined, because d doesn't inherit from Object.prototype
```

Pros and cons of `Object.create`

Pro(s)	Supported in all modern engines. Allows directly setting `[[Prototype]]` of an object at creation time, which permits the runtime to further optimize the object. Also allows the creation of objects without a prototype, using `Object.create(null)`.
Con(s)	Not supported in IE8 and below. However, as Microsoft has discontinued extended support for systems running IE8 and below, that should not be a concern for most applications. Additionally, the slow object initialization can be a performance black hole if using the second argument, because each object-descriptor property has its own separate descriptor object. When dealing with hundreds of thousands of object descriptors in the form of objects, that lag time might become a serious issue.

With classes

```
class Polygon {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}

class Square extends Polygon {
  constructor(sideLength) {
    super(sideLength, sideLength);
  }

  get area() {
    return this.height * this.width;
  }

  set sideLength(newLength) {
    this.height = newLength;
    this.width = newLength;
  }
}

const square = new Square(2);
// square --> Square.prototype --> Polygon.prototype --> Object.prototype --> null
```

Pros and cons of classes

Pro(s)	Supported in all modern engines. Very high readability and maintainability. Private properties are a feature with no trivial replacement in prototypical inheritance.
Con(s)	Classes, especially with private properties, are less optimized than traditional ones (although engine implementors are working to improve this). Not supported in older environments and transpilers are usually needed to use classes in production.

With Object.setPrototypeOf()

While all methods above will set the prototype chain at object creation time, `Object.setPrototypeOf()` allows mutating the `[[Prototype]]` internal property of an existing object.

```
const obj = { a: 1 };
const anotherObj = { b: 2 };
Object.setPrototypeOf(obj, anotherObj);
// obj ---> anotherObj ---> Object.prototype ---> null
```

Pros and cons of `Object.setPrototypeOf`

Pro(s)	Supported in all modern engines. Allows the dynamic manipulation of an object's prototype and can even force a prototype on a prototype-less object created with `Object.create(null)`.
Con(s)	Ill-performing. Should be avoided if it's possible to set the prototype at object creation time. Many engines optimize the prototype and try to guess the location of the method in memory when calling an instance in advance; but setting the prototype dynamically disrupts all those optimizations. It might cause some engines to recompile your code for de-optimization, to make it work according to the specs. Not supported in IE8 and below.

With Object.setPrototypeOf()

All objects inherit the `Object.prototype.__proto__` setter, which can be used to set the `[[Prototype]]` of an existing object (if the `__proto__` key is not overridden on the object).

Warning: `Object.prototype.__proto__` accessors are non-standard and deprecated. You should almost always use `Object.setPrototypeOf` instead.

```
const obj = {};
// DON'T USE THIS: for example only.
obj.__proto__ = { barProp: "bar val" };
obj.__proto__.__proto__ = { fooProp: "foo val" };
console.log(obj.fooProp);
console.log(obj.barProp);
```

Pros and cons of setting the __proto__ property

Pro(s)	Supported in all modern engines. Setting __proto__ to something that is not an object only fails silently. It does not throw an exception.
Con(s)	Non-performant and deprecated. Many engines optimize the prototype and try to guess the location of the method in the memory when calling an instance in advance; but setting the prototype dynamically disrupts all those optimizations and can even force some engines to recompile for de-optimization of your code, to make it work according to the specs. Not supported in IE10 and below. The __proto__ setter is normative optional, so it may not work across all platforms. You should almost always use Object.setPrototypeOf instead.

Performance

The lookup time for properties that are high up on the prototype chain can have a negative impact on the performance, and this may be significant in the code where performance is critical. Additionally, trying to access nonexistent properties will always traverse the full prototype chain.

Also, when iterating over the properties of an object, every enumerable property that is on the prototype chain will be enumerated. To check whether an object has a property defined on itself and not somewhere on its prototype chain, it is necessary to use the hasOwnProperty or Object.hasOwn methods. All objects, except those with null as [[Prototype]], inherit hasOwnProperty from Object.prototype — unless it has been overridden further down the prototype chain. To give you a concrete example, let's take the above graph example code to illustrate it:

```
function Graph() {
  this.vertices = [];
  this.edges = [];
}

Graph.prototype.addVertex = function (v) {
  this.vertices.push(v);
};

const g = new Graph();
// g ---> Graph.prototype ---> Object.prototype ---> null

g.hasOwnProperty("vertices"); // true
Object.hasOwn(g, "vertices"); // true

g.hasOwnProperty("nope"); // false
Object.hasOwn(g, "nope"); // false

g.hasOwnProperty("addVertex"); // false
Object.hasOwn(g, "addVertex"); // false

Object.getPrototypeOf(g).hasOwnProperty("addVertex"); // true
```

Conclusion

JavaScript may be a bit confusing for developers coming from Java or C++, as it's all dynamic, all runtime, and it has no static types at all. Everything is either an object (instance) or a function (constructor), and even functions themselves are instances of the `Function` constructor. Even the "classes" as syntax constructs are just constructor functions at runtime.

All constructor functions in JavaScript have a special property called `prototype`, which works with the new operator. The reference to the prototype object is copied to the internal `[[Prototype]]` property of the new instance. For example, when you do `const a1 = new A()`, JavaScript (after creating the object in memory and before running function `A()` with `this` defined to it) sets `a1.[[Prototype]] = A.prototype`. When you then access properties of the instance, JavaScript first checks whether they exist on that object directly, and if not, it looks in `[[Prototype]]`. `[[Prototype]]` is looked at recursively, i.e. `a1.doSomething`, `Object.getPrototypeOf(a1).doSomething`, `Object.getPrototypeOf(Object.getPrototypeOf(a1)).doSomething` etc., until it's found or `Object.getPrototypeOf` returns `null`. This means that all properties defined on `prototype` are effectively shared by all instances, and you can even later change parts of `prototype` and have the changes appear in all existing instances.

If, in the example above, you do `const a1 = new A(); const a2 = new A();`, then `a1.doSomething` would actually refer to `Object.getPrototypeOf(a1).doSomething` — which is the same as the `A.prototype.doSomething` you defined, i.e. `Object.getPrototypeOf(a1).doSomething === Object.getPrototypeOf(a2).doSomething === A.prototype.doSomething`.

It is essential to understand the prototypal inheritance model before writing complex code that makes use of it. Also, be aware of the length of the prototype chains in your code and break them up if necessary to avoid possible performance problems. Further, the native prototypes should **never** be extended unless it is for the sake of compatibility with newer JavaScript features.
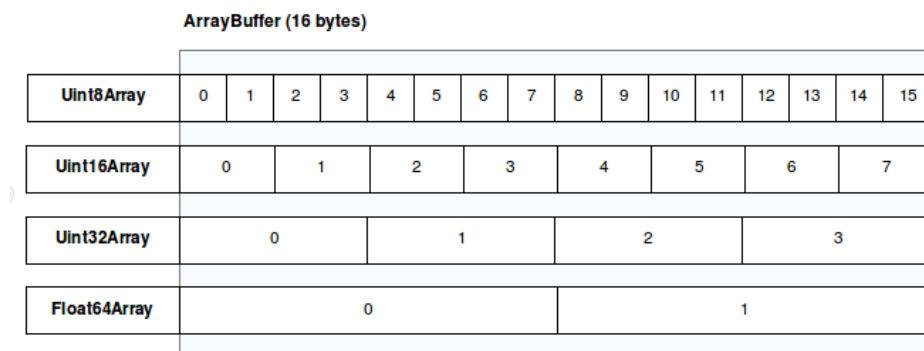
JavaScript typed arrays

JavaScript typed arrays are array-like objects that provide a mechanism for reading and writing raw binary data in memory buffers.

`Array` objects grow and shrink dynamically and can have any JavaScript value. JavaScript engines perform optimizations so that these arrays are fast. However, as web applications become more and more powerful, adding features such as audio and video manipulation, access to raw data using WebSockets, and so forth, it has become clear that there are times when it would be helpful for JavaScript code to be able to quickly and easily manipulate raw binary data. This is where typed arrays come in. Each entry in a JavaScript typed array is a raw binary value in one of a number of supported formats, from 8-bit integers to 64-bit floating-point numbers.

Typed array objects share many of the same methods as arrays with similar semantics. However, typed arrays are not to be confused with normal arrays, as calling `Array.isArray()` on a typed array returns `false`. Moreover, not all methods available for normal arrays are supported by typed arrays (e.g. push and pop).

Buffers and views: typed array architecture

To achieve maximum flexibility and efficiency, JavaScript typed arrays split the implementation into *buffers* and *views*. A buffer (implemented by the `ArrayBuffer` object) is an object representing a chunk of data; it has no format to speak of, and offers no mechanism for accessing its contents. In order to access the memory contained in a buffer, you need to use a view. A view provides a context — that is, a data type, starting offset, and number of elements — that turns the data into an actual typed array.

ArrayBuffer (16 bytes)

Uint8Array	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Uint16Array	0		1		2		3		4		5		6		7	
Uint32Array	0				1				2				3			
Float64Array	0								1							

ArrayBuffer

The `ArrayBuffer` is a data type that is used to represent a generic, fixed-length binary data buffer. You can't directly manipulate the contents of an ArrayBuffer; instead, you create a typed array view or a `DataView` which represents the buffer in a specific format, and use that to read and write the contents of the buffer.

Typed array views

Typed array views have self-descriptive names and provide views for all the usual numeric types like `Int8`, `Uint32`, `Float64` and so forth. There is one special typed array view, `Uint8ClampedArray`, which clamps the values between `0` and `255`. This is useful for Canvas data processing, for example.

Type	Value Range	Size in bytes	Description	Web IDL type	Equivalent C type
Int8Array	-128 to 127	1	8-bit two's complement signed integer	byte	int8_t
Uint8Array	0 to 255	1	8-bit unsigned integer	octet	uint8_t
Uint8ClampedArray	0 to 255	1	8-bit unsigned integer (clamped)	octet	uint8_t
Int16Array	-32768 to 32767	2	16-bit two's complement signed integer	short	int16_t
Uint16Array	0 to 65535	2	16-bit unsigned integer	unsigned short	uint16_t
Int32Array	-2147483648 to 2147483647	4	32-bit two's complement signed integer	long	int32_t
Uint32Array	0 to 4294967295	4	32-bit unsigned integer	unsigned long	uint32_t
Float32Array	-3.4E38 to 3.4E38 and 1.2E-38 is the min positive number	4	32-bit IEEE floating point number (7 significant digits e.g., 1.123456)	unrestricted float	float
Float64Array	-1.8E308 to 1.8E308 and 5E-324 is the min positive number	8	64-bit IEEE floating point number (16 significant digits e.g., 1.123...15)	unrestricted double	double
BigInt64Array	-263 to 263 - 1	8	64-bit two's complement signed integer	bigint	int64_t (signed long long)

BigUint64Array	0 to 264 - 1	8	64-bit unsigned integer	`bigint`	uint64_t (unsigned long long)

DataView

The `DataView` is a low-level interface that provides a getter/setter API to read and write arbitrary data to the buffer. This is useful when dealing with different types of data, for example. Typed array views are in the native byte-order (see Endianness) of your platform. With a `DataView` you are able to control the byte-order. It is big-endian by default and can be set to little-endian in the getter/setter methods.

Web APIs using typed arrays

These are some examples of APIs that make use of typed arrays; there are others, and more are being added all the time.

```
FileReader.prototype.readAsArrayBuffer()
```

The `FileReader.prototype.readAsArrayBuffer()` method starts reading the contents of the specified `Blob` or `File`.

```
XMLHttpRequest.prototype.send()
```

`XMLHttpRequest` instances' `send()` method now supports typed arrays and `ArrayBuffer` objects as argument.

```
ImageData.data
```

Is a `Uint8ClampedArray` representing a one-dimensional array containing the data in the RGBA order, with integer values between `0` and `255` inclusive.

Examples

Using views with buffers

First of all, we will need to create a buffer, here with a fixed length of 16-bytes:

```
const buffer = new ArrayBuffer(16);
```

At this point, we have a chunk of memory whose bytes are all pre-initialized to 0. There's not a lot we can do with it, though. We can confirm that it is indeed 16 bytes long, and that's about it:

```
if (buffer.byteLength === 16) {
  console.log("Yes, it's 16 bytes.");
} else {
  console.log("Oh no, it's the wrong size!");
}
```

Before we can really work with this buffer, we need to create a view. Let's create a view that treats the data in the buffer as an array of 32-bit signed integers:

```
const int32View = new Int32Array(buffer);
```

Now we can access the fields in the array just like a normal array:

```
for (let i = 0; i < int32View.length; i++) {
  int32View[i] = i * 2;
}
```

This fills out the 4 entries in the array (4 entries at 4 bytes each makes 16 total bytes) with the values 0, 2, 4, and 6.

Multiple views on the same data

Things start to get really interesting when you consider that you can create multiple views onto the same data. For example, given the code above, we can continue like this:

```
const int16View = new Int16Array(buffer);

for (let i = 0; i < int16View.length; i++) {
  console.log(`Entry ${i}: ${int16View[i]}`);
}
```

Here we create a 16-bit integer view that shares the same buffer as the existing 32-bit view and we output all the values in the buffer as 16-bit integers. Now we get the output `0, 0, 2, 0, 4, 0, 6, 0`.

You can go a step farther, though. Consider this:

```
int16View[0] = 32;
console.log(`Entry 0 in the 32-bit array is now ${int32View[0]}`);
```

The output from this is "`Entry 0 in the 32-bit array is now 32`".

In other words, the two arrays are indeed viewed on the same data buffer, treating it as different formats. You can do this with any view types.

Working with complex data structures

By combining a single buffer with multiple views of different types, starting at different offsets into the buffer, you can interact with data objects containing multiple data types. This lets you, for example, interact with complex data structures from WebGL or data files.

Consider this C structure:

```
struct someStruct {
  unsigned long id;
  char username[16];
  float amountDue;
};
```

You can access a buffer containing data in this format like this:

```
const buffer = new ArrayBuffer(24);

// ... read the data into the buffer ...

const idView = new Uint32Array(buffer, 0, 1);
const usernameView = new Uint8Array(buffer, 4, 16);
const amountDueView = new Float32Array(buffer, 20, 1);
Copy to ClipboardCopy to ClipboardCopy to Clipboard
Then you can access, for example, the amount due with amountDueView[0].
```

Then you can access, for example, the amount due with `amountDueView[0]`.

Conversion to normal arrays

After processing a typed array, it is sometimes useful to convert it back to a normal array in order to benefit from the `Array` prototype. This can be done using `Array.from()`.

```
const typedArray = new Uint8Array([1, 2, 3, 4]);
const normalArray = Array.from(typedArray);
```

as well as the spread syntax

```
const typedArray = new Uint8Array([1, 2, 3, 4]);
const normalArray = [...typedArray];
```

or using the following code where `Array.from()` is unsupported.

```
const typedArray = new Uint8Array([1, 2, 3, 4]);
const normalArray = Array.prototype.slice.call(typedArray);
```

Memory management

Low-level languages like C, have manual memory management primitives such as `malloc()` and `free()`. In contrast, JavaScript automatically allocates memory when objects are created and frees it when they are not used anymore (garbage collection). This automaticity is a potential source of confusion: it can give developers the false impression that they don't need to worry about memory management.

Memory life cycle

Regardless of the programming language, the memory life cycle is pretty much always the same:

1. Allocate the memory you need
2. Use the allocated memory (read, write)
3. Release the allocated memory when it is not needed anymore

The second part is explicit in all languages. The first and last parts are explicit in low-level languages but are mostly implicit in high-level languages like JavaScript.

Allocation in JavaScript

Value initialization

In order to not bother the programmer with allocations, JavaScript will automatically allocate memory when values are initially declared.

```
const n = 123; // allocates memory for a number
const s = "azerty"; // allocates memory for a string

const o = {
  a: 1,
  b: null,
}; // allocates memory for an object and contained values

// (like object) allocates memory for the array and
// contained values
const a = [1, null, "abra"];

function f(a) {
  return a + 2;
} // allocates a function (which is a callable object)
```

```
  // function expressions also allocate an object
  someElement.addEventListener(
    "click",
    () => {
      someElement.style.backgroundColor = "blue";
    },
    false,
  );
```

Allocation via function calls

Some function calls result in object allocation.

```
  const d = new Date(); // allocates a Date object
  const e = document.createElement("div"); // allocates a DOM element
```

Some methods allocate new values or objects:

```
  const s = "azerty";
  const s2 = s.substr(0, 3); // s2 is a new string
  // Since strings are immutable values,
  // JavaScript may decide to not allocate memory,
  // but just store the [0, 3] range.

  const a = ["ouais ouais", "nan nan"];
  const a2 = ["generation", "nan nan"];
  const a3 = a.concat(a2);
  // new array with 4 elements being
  // the concatenation of a and a2 elements.
```

Using values

Using values basically means reading and writing in allocated memory. This can be done by reading or writing the value of a variable or an object property or even passing an argument to a function.

Release when the memory is not needed anymore

The majority of memory management issues occur at this phase. The most difficult aspect of this stage is determining when the allocated memory is no longer needed.

Low-level languages require the developer to manually determine at which point in the program the allocated memory is no longer needed and to release it.

Some high-level languages, such as JavaScript, utilize a form of automatic memory management known as garbage collection (GC). The purpose of a garbage collector is to monitor memory allocation and determine when a block of allocated memory is no longer needed and reclaim it. This is automatic process.

Garbage collection

As stated above, the general problem of automatically finding whether some memory "is not needed anymore" is undecidable. As a consequence, garbage collectors implement a restriction of a solution to the general problem.

References

The main concept that garbage collection algorithms rely on is the concept of *reference*. Within the context of memory management, an object is said to reference another object if the former has access to the latter (either implicitly or explicitly). For instance, a JavaScript object has a reference to its prototype (implicit reference) and to its properties values (explicit reference).

In this context, the notion of an "object" is extended to something broader than regular JavaScript objects and also contain function scopes (or the global lexical scope).

Reference-counting garbage collection

Note: no modern browser uses reference-counting for garbage collection anymore.

Reference-counting garbage collection is a type of automatic memory management technique used in some programming languages, including JavaScript.

In this approach, every object in memory has a reference count, which represents the number of references to that object in the program. When an object is created, its reference count is set to 1. Every time the object is assigned to a variable or used as a parameter in a function call, its reference count is incremented. Conversely, every time a reference to the object is removed (for example, when a variable goes out of scope), its reference count is decremented.

Once an object's reference count reaches 0, it means that no other part of the program is referencing that object. At this point, the garbage collector can safely remove the object from memory to free up space.

```
let obj1 = { name: "John" }; // reference count for obj1 is 1
let obj2 = obj1; // reference count for obj1 and obj2 is 2
obj1 = null; // reference count for obj2 is 1
obj2 = null; // reference count for obj2 is 0, so the object can be garbage collected
```

In this example, we create two objects (`obj1 and obj2`) and set `obj2` equal to `obj1`. This means that both `obj1` and `obj2` reference the same object in memory, so the reference count for that object is 2. We then set `obj1` to `null`, which decrements the reference count for the object to 1. Finally, we set `obj2` to `null`, which decrements the reference count to 0 and allows the garbage collector to remove the object from memory.

Reference-counting has some advantages, such as immediate reclamation of memory when an object's reference count drops to zero, but it also has some limitations. For example, it can't detect reference cycles, where objects reference each other in a circular manner, causing a memory leak. That's why modern JavaScript engines use more sophisticated garbage collection techniques, such as mark-and-sweep and generational garbage collection.

Circular references are a common cause of memory leaks.

Mark-and-sweep algorithm

The Mark-and-sweep algorithm is a garbage collection technique used by many modern programming languages, including JavaScript. It works by identifying and collecting objects that are no longer being used by the program, freeing up memory for new objects to be created. Here's how the Mark-and-sweep algorithm works:

1. Mark Phase: The first phase of the algorithm is the mark phase. The garbage collector starts at the root objects, which are usually global variables and objects referenced from the execution stack. The algorithm then recursively traverses all reachable objects, marking them as live or reachable. Objects that are not reachable are considered garbage.

2. Sweep Phase: Once all reachable objects are marked, the algorithm moves on to the sweep phase. During this phase, the garbage collector looks through the memory and frees up any unmarked objects. This frees up the memory used by the unmarked objects for use by the program.

3. Fragmentation Phase: Finally, there is the fragmentation phase, which is optional in some implementations. This phase compacts the memory and eliminates any fragmentation that may have occurred as a result of the sweep phase.

Here's an example of how the Mark-and-sweep algorithm works:

```
    // Create some objects
    let obj1 = { foo: 'bar' };
    let obj2 = { baz: 'qux' };
    let obj3 = { quux: 'corge' };

    // Assign references to the objects
    let arr = [obj1, obj2];
    let obj4 = { arr: arr, obj: obj3 };

    // Remove references to the objects
    obj1 = null;
    obj2 = null;
    obj3 = null;
    arr = null;
    obj4 = null;

    // Run the garbage collector
    // Mark phase will mark all objects in the array and the object that references it
    // The sweep phase will then free up the memory used by the unmarked objects
```

In this example, we create four objects and assign references to them. We then remove all references to the objects, making them garbage. Finally, we run the garbage collector, which will mark all objects that are reachable and free up any memory used by unmarked objects.

The Mark-and-sweep algorithm is a powerful technique for managing memory in programming languages. It allows programs to allocate and deallocate memory dynamically without worrying about memory leaks or other issues. While it can be slower than other garbage collection techniques, it is often more reliable and easier to use.

Configuring an engine's memory model

JavaScript engines typically offer flags that expose the memory model. For example, Node.js offers additional options and tools that expose the underlying V8 mechanisms for configuring and debugging memory issues. This configuration may not be available in browsers, and even less so for web pages (via HTTP headers, etc.).

The max amount of available heap memory can be increased with a flag:

```
node --max-old-space-size=6000 index.js
```

We can also expose the garbage collector for debugging memory issues using a flag and the Chrome Debugger:

```
node --expose-gc --inspect index.js
```

Data structures aiding memory management

Although JavaScript does not directly expose the garbage collector API, the language offers several data structures that indirectly observe garbage collection and can be used to manage memory usage.

WeakMaps and WeakSets

WeakMaps and WeakSets are special types of objects in JavaScript that allow you to associate data with objects without preventing the garbage collector from reclaiming memory when the object is no longer needed.

A WeakMap is a collection of key-value pairs where the keys must be objects and the values can be any type of value. The WeakMap object holds weak references to the keys, which means that if the key object has no other references, it can be garbage collected even if it's still in the WeakMap.

Here's an example of how to use WeakMap:

```
let myWeakMap = new WeakMap();
let obj1 = {};
let obj2 = {};

myWeakMap.set(obj1, "hello");
myWeakMap.set(obj2, "world");

console.log(myWeakMap.get(obj1)); // "hello"
console.log(myWeakMap.get(obj2)); // "world"

obj1 = null; // The key object is no longer referenced anywhere
console.log(myWeakMap.get(obj1)); // undefined
```

In this example, we create a WeakMap object `myWeakMap` and add two key-value pairs to it using the `set()` method. We then retrieve the values associated with the keys using the `get()` method. Finally, we set `obj1` to `null`, which means it's no longer referenced anywhere, and when we try to get the value associated with it, it returns undefined.

A WeakSet is a collection of objects where each object can only appear once. Like a WeakMap, a WeakSet holds weak references to the objects it contains. If an object has no other references, it can be garbage collected even if it's still in the WeakSet.

Here's an example of how to use WeakSet:

```
let myWeakSet = new WeakSet();
let obj1 = {};
let obj2 = {};

myWeakSet.add(obj1);
myWeakSet.add(obj2);

console.log(myWeakSet.has(obj1)); // true
console.log(myWeakSet.has(obj2)); // true

obj1 = null; // The object is no longer referenced anywhere
console.log(myWeakSet.has(obj1)); // false
```

In this example, we create a WeakSet object `myWeakSet` and add two objects to it using the `add()` method. We then check if the WeakSet contains each object using the `has()` method. Finally, we set `obj1` to `null`, which means it's no longer referenced anywhere, and when we check if the WeakSet contains it, it returns false.

WeakMaps and WeakSets are useful when you need to associate data with an object but don't want to prevent the object from being garbage collected. They can be used in situations where you need to track some data associated with an object, but the object itself can be destroyed at any time.

WeakRefs and FinalizationRegistry

WeakRefs and FinalizationRegistry are features introduced in ECMAScript 2021, which is the latest version of the JavaScript language. They provide additional tools for managing memory in JavaScript, especially in situations where you need to keep track of objects that may be garbage collected.

A WeakRef is a weak reference to an object that does not prevent that object from being garbage collected. This is different from a normal reference, which would keep the object alive as long as the reference exists. A WeakRef allows you to track an object without affecting its lifetime.

Here's an example of how to use WeakRef:

```
let obj = { foo: 'bar' };
let weakRef = new WeakRef(obj);

console.log(weakRef.deref()); // { foo: 'bar' }

obj = null; // The object is no longer referenced anywhere
console.log(weakRef.deref()); // null
```

In this example, we create an object `obj` and a WeakRef `weakRef` to it. We then use the `deref()` method to get the object associated with the WeakRef. Finally, we set obj to `null`, which means it's no longer referenced anywhere, and when we try to get the object associated with the WeakRef again, it returns `null`.

A FinalizationRegistry is a registry of functions that will be called when an object is garbage collected. The registry allows you to associate cleanup functions with objects, which can be useful for freeing resources or cleaning up after an object.

Here's an example of how to use FinalizationRegistry:

```
let registry = new FinalizationRegistry((value) => {
  console.log(`Cleaning up ${value}`);
});
let obj = { foo: 'bar' };
registry.register(obj, 'myValue');
console.log(registry.has(obj)); // true
obj = null; // The object is no longer referenced anywhere
console.log(registry.has(obj)); // false

// The cleanup function will be called when the object is garbage collected
```

In this example, we create a FinalizationRegistry object `registry` with a cleanup function that logs a message. We then create an object `obj` and register it with the registry using the `register()` method, along with a value `'myValue'`. We check if the registry has the object using the `has()` method, and then set `obj` to `null`. Finally, we see that the registry no longer has the object and expect the cleanup function to be called when the object is garbage collected.

WeakRefs and FinalizationRegistry are useful for managing memory in situations where you need to keep track of objects that may be garbage collected. They can be used to track resources associated with objects and ensure that those resources are properly freed when the object is no longer needed.

This lesson was adapted from MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Typed_arrays

The event loop

JavaScript has a runtime model based on an event loop, which is responsible for executing the code, collecting and processing events, and executing queued sub-tasks. This model is quite different from models in other languages like C and Java.

Runtime concepts

The following sections explain a theoretical model. Modern JavaScript engines implement and heavily optimize the described semantics.

Visual representation
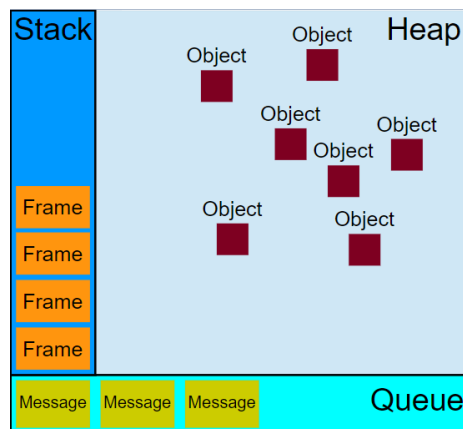
Stack

Function calls form a stack of *frames*.

```
function foo(b) {
  const a = 10;
  return a + b + 11;
}
function bar(x) {
  const y = 3;
  return foo(x * y);
}
const baz = bar(7); // assigns 42 to baz
```

Order of operations:

1. When calling `bar`, a first frame is created containing references to `bar`'s arguments and local variables.
2. When `bar` calls `foo`, a second frame is created and pushed on top of the first one, containing references to `foo`'s arguments and local variables.
3. When `foo` returns, the top frame element is popped out of the stack (leaving only bar's call frame).
4. When `bar` returns, the stack is empty.

Note that the arguments and local variables may continue to exist, as they are stored outside the stack — so they can be accessed by any nested functions long after their outer function has returned.

Heap

Objects are allocated in a heap which is just a name to denote a large (mostly unstructured) region of memory.

Queue

A JavaScript runtime uses a message queue, which is a **list of messages to be processed**. Each message has an associated function that gets called to handle the message.

At some point during the event loop, the runtime starts handling the messages on the queue, **starting with the oldest one**. To do so, the message is removed from the queue and its corresponding function is called with the message as an input parameter. As always, calling a function creates a new stack frame for that function's use.

The processing of functions continues until the stack is once again empty. Then, the event loop will process the next message in the queue (if there is one).

Event loop

The event loop got its name because of how it's usually implemented, which usually resembles:

```
while (queue.waitForMessage()) {
  queue.processNextMessage();
}
```

`queue.waitForMessage()` waits synchronously for a message to arrive (if one is not already available and waiting to be handled).

"Run-to-completion"

Each message is processed completely before any other message is processed.

This offers some nice properties when reasoning about your program, including the fact that whenever a function runs, it cannot be preempted and will run entirely before any other code runs (and can modify data the function manipulates). This differs from C, for instance, where if a function runs in a thread, it may be stopped at any point by the runtime system to run some other code in another thread.

A downside of this model is that if a message takes too long to complete, the web application is unable to process user interactions like click or scroll. The browser mitigates this with the "a script is taking too long to run" dialog. A good practice to follow is to make message processing short and if possible cut down one message into several messages.

Adding messages

In web browsers, messages are added anytime an event occurs and there is an event listener attached to it. If there is no listener, the event is lost. So a click on an element with a click event handler will add a message — likewise with any other event.

The first two arguments to the function `setTimeout` are a message to add to the queue and a time value (optional; defaults to 0). The time value represents the (minimum) delay after which the message will be pushed into the queue. If there is no other message in the queue, and the stack is empty, the message is processed right after the delay. However, if there are messages, the

`setTimeout` message will have to wait for other messages to be processed. For this reason, the second argument indicates a minimum time — not a guaranteed time.

Here is an example that demonstrates this concept (`setTimeout` does not run immediately after its timer expires):

```
const seconds = new Date().getTime() / 1000;

setTimeout(() => {
  // prints out "2", meaning that the callback is not called immediately after 500
      milliseconds.
  console.log(`Ran after ${new Date().getTime() / 1000 - seconds} seconds`);
}, 500);

while (true) {
  if (new Date().getTime() / 1000 - seconds >= 2) {
    console.log("Good, looped for 2 seconds");
    break;
  }
}
```

Zero delays

Zero delay doesn't mean the call back will fire-off after zero milliseconds. Calling `setTimeout` with a delay of `0` (zero) milliseconds doesn't execute the callback function after the given interval.

The execution depends on the number of waiting tasks in the queue. In the example below, the message `"this is just a message"` will be written to the console before the message in the callback gets processed, because the delay is the minimum time required for the runtime to process the request (not a guaranteed time).

The `setTimeout` needs to wait for all the code for queued messages to complete even though you specified a particular time limit for your `setTimeout`.

```
(() => {
  console.log("this is the start");

  setTimeout(() => {
    console.log("Callback 1: this is a msg from call back");
  }); // has a default time value of 0

  console.log("this is just a message");
```

```
  setTimeout(() => {
    console.log("Callback 2: this is a msg from call back");
  }, 0);

  console.log("this is the end");
})();

// "this is the start"
// "this is just a message"
// "this is the end"
// "Callback 1: this is a msg from call back"
// "Callback 2: this is a msg from call back"
```

Several runtimes communicating together

A web worker or a cross-origin `iframe` has its own stack, heap, and message queue. Two distinct runtimes can only communicate through sending messages via the `postMessage` method. This method adds a message to the other runtime if the latter listens to `message` events.

Never blocking

A very interesting property of the event loop model is that JavaScript, unlike a lot of other languages, never blocks. Handling I/O is typically performed via events and callbacks, so when the application is waiting for an IndexedDB query to return or an XHR request to return, it can still process other things like user input.

Legacy exceptions exist like `alert` or synchronous XHR, but it is considered good practice to avoid them. Beware: exceptions to the exception do exist (but are usually implementation bugs, rather than anything else).

This lesson was adapted from MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Event_loop

Document Object Model (DOM)

The Document Object Model (DOM) **connects web pages to scripts or programming languages by representing the structure of a document** — such as the HTML representing a web page — in memory. Usually, it refers to JavaScript, even though modeling HTML, SVG, or XML documents as objects are not part of the core JavaScript language.

The DOM represents a document with a logical tree. **Each branch of the tree ends in a node, and each node contains objects**. **DOM methods allow programmatic access to the tree**. With them, you can change the document's structure, style, or content.

Nodes can also have event handlers attached to them. Once an event is triggered, the event handlers get executed.

DOM interfaces

DOM interfaces, also known as DOM APIs or DOM objects, are a **standardized set of programming tools provided by web browsers to allow developers to interact with web documents, such as HTML, XML, XHTML, or SVG files**, in a structured way.

The DOM is organized into different levels, each representing a specific version of the DOM specification:

1. DOM Level 1: The first level provided basic functionality for accessing and modifying the elements in an HTML or XML document. It laid the foundation for the subsequent DOM levels.
2. DOM Level 2: Building upon the foundation of Level 1, this level introduced additional interfaces and features. It added support for event handling, allowing developers to respond to user interactions like clicks and keyboard input. DOM Level 2 also included enhanced navigation and manipulation capabilities.
3. DOM Level 3: This level further extended the DOM by adding support for advanced features such as the ability to use XPath expressions to navigate and select elements in a document. Additionally, it included improvements to event handling and introduced more precise control over the document.

These DOM interfaces represent the elements of a web page as a tree-like structure, where each element is a node in the tree. Developers can use methods and properties provided by these interfaces to traverse the tree, read and modify content, create new elements, and respond to user actions and events. This enables developers to build dynamic and interactive web applications by

manipulating the content and appearance of web pages in real-time using JavaScript or other scripting languages. The different DOM levels ensure that browsers adhere to standardized APIs, allowing developers to write consistent and cross-browser compatible code.

HTML DOM

HTML DOM (Document Object Model) is a programming interface provided by web browsers that represent the structure and content of an HTML document as a tree-like structure of objects. It allows developers to interact with web pages dynamically using JavaScript or other scripting languages.

Key points about HTML DOM:
1. **Tree-like Structure**: The DOM represents an HTML document as a tree of objects, where each HTML element is a node in the tree. The "document" object represents the entire HTML page and serves as the root of the tree. All other elements (e.g., headings, paragraphs, images) are its children, forming a hierarchical structure.
2. **Access and Manipulate Elements**: With the HTML DOM, developers can access specific elements in the HTML document using methods like **getElementById**, **getElementsByTagName**, **getElementsByClassName**, **querySelector**, etc. Once accessed, they can modify the elements' attributes, content, or even add and remove elements dynamically.
3. **Dynamic Interaction**: The HTML DOM enables developers to create interactive web pages by adding event listeners to elements. This way, they can respond to user actions (e.g., clicks, keypresses) and trigger appropriate actions or functions.
4. **Cross-Browser Compatibility**: The HTML DOM provides a standardized way to interact with HTML documents, ensuring that code works consistently across different web browsers.

By utilizing the HTML DOM, developers can build powerful and dynamic web applications, enhance user experience, and create responsive and interactive content on web pages.

Introduction to the DOM

What is the DOM?

The Document Object Model (DOM) is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects; that way, programming languages can interact with the page.

A web page is a document that can be either displayed in the browser window or as the HTML source. In both cases, it is the same document but the Document Object Model (DOM) representation allows it to be manipulated. As an object-oriented representation of the web page, it can be modified with a scripting language such as JavaScript.

For example, the DOM specifies that the `querySelectorAll` method in this code snippet must return a list of all the `<p>` elements in the document:

```
const paragraphs = document.querySelectorAll("p");
// paragraphs[0] is the first <p> element
// paragraphs[1] is the second <p> element, etc.
alert(paragraphs[0].nodeName);
```

All of the properties, methods, and events available for manipulating and creating web pages are organized into objects. For example, the document object that represents the document itself, any table objects that implement the `HTMLTableElement` DOM interface for accessing HTML tables, and so forth, are all objects.

The DOM is built using multiple APIs that work together. The core DOM defines the entities describing any document and the objects within it. This is expanded upon as needed by other APIs that add new features and capabilities to the DOM. For example, the HTML DOM API adds support for representing HTML documents to the core DOM, and the SVG API adds support for representing SVG documents.

DOM and JavaScript

The previous short example, like nearly all examples, is JavaScript. That is to say, it is written in JavaScript, but uses the DOM to access the document and its elements. The DOM is not a programming language, but without it, the JavaScript language wouldn't have any model or notion of web pages, HTML documents, SVG documents, and their component parts. The document as a whole, the head, tables within the document, table headers, text within the table cells, and all other elements in a document are parts of the document object model for that document. They can all be accessed and manipulated using the DOM and a scripting language like JavaScript.

The DOM is not part of the JavaScript language, but is instead a Web API used to build websites. JavaScript can also be used in other contexts. For example, Node.js runs JavaScript programs on

a computer but provides a different set of APIs, and the DOM API is not a core part of the Node.js runtime.

The DOM was designed to be independent of any particular programming language, making the structural representation of the document available from a single, consistent API. Even if most web developers will only use the DOM through JavaScript, implementations of the DOM can be built for any language, as this Python example demonstrates:

```python
# Python DOM example
import xml.dom.minidom as m
doc = m.parse(r"C:\Projects\Py\chap1.xml")
doc.nodeName # DOM property of document object
p_list = doc.getElementsByTagName("para")
```

Accessing the DOM

You don't have to do anything special to begin using the DOM. You use the API directly in JavaScript from within what is called a script, a program run by a browser.

When you create a script, whether inline in a <script> element or included in the web page, you can immediately begin using the API for the document or window objects to manipulate the document itself, or any of the various elements in the web page (the descendant elements of the document). Your DOM programming may be something as simple as the following example, which displays a message on the console by using the `console.log()` function:

```html
<body onload="console.log('Welcome to my home page!');">
  …
</body>
```

As it is generally not recommended to mix the structure of the page (written in HTML) and manipulation of the DOM (written in JavaScript), the JavaScript parts will be grouped together here, and separated from the HTML.

For example, the following function creates a new `h1` element, adds text to that element, and then adds it to the tree for the document:

```
<html lang="en">
  <head>
    <script>
      // run this function when the document is loaded
      window.onload = () => {
        // create a couple of elements in an otherwise empty HTML page
        const heading = document.createElement("h1");
        const headingText = document.createTextNode("Big Head!");
        heading.appendChild(headingText);
        document.body.appendChild(heading);
      };
    </script>
  </head>
  <body></body>
</html>
```

Fundamental data types

This page tries to describe the various objects and types in simple terms. But there are a number of different data types being passed around the API that you should be aware of.

Note: Because the vast majority of code that uses the DOM revolves around manipulating HTML documents, it's common to refer to the nodes in the DOM as elements, although strictly speaking **not every node is an element**.

The following table briefly describes these data types.

Data type (Interface)	Description
Document	When a member returns an object of type document (e.g., the `ownerDocument` property of an element returns the document to which it belongs), this object is the root `document` object itself. The DOM document Reference chapter describes the `document` object.
Node	Every object located in document is node of some kind. In an HTML document, an object can be an element node but also a text node or attribute node.
Element	The `element` type is based on `node`. It refers to an element or a node of type `element` returned by a member of the DOM API. Rather than

	saying, for example, that the `document.createElement()` method returns an object reference to a `node`, we just say that this method returns the `element` that has just been created in the DOM. `element` objects implement the DOM Element interface and also the more basic Node interface, both of which are included together in this reference.
	In an HTML document, elements are further enhanced by the HTML DOM API's `HTMLElement` interface as well as other interfaces describing capabilities of specific kinds of elements (for instance, `HTMLTableElement` for `<table>` elements).
NodeList	A `nodeList` is an array of elements, like the kind that is returned by the method `document.querySelectorAll()`. Items in a `nodeList` are accessed by index in either of two ways: • list.item(1) • list[1] These two are equivalent. In the first, `item()` is the single method on the nodeList object. The latter uses the typical array syntax to fetch the second item in the list.
Attr	When an `attribute` is returned by a member (e.g., by the `createAttribute()` method), it is an object reference that exposes a special interface for attributes. Attributes are nodes in the DOM just like elements are, though you may rarely use them as such.
NamedNodeMap	A `namedNodeMap` is like an array, but the items are accessed by name or index, though this latter case is merely a convenience for enumeration, as they are in no particular order in the list. A `namedNodeMap` has an `item()` method for this purpose, and you can also add and remove items from a `namedNodeMap`.

There are also some common terminology considerations to keep in mind. It's common to refer to any `Attr` node as an `attribute`, for example, and to refer to an array of DOM nodes as a `nodeList`. You'll find these terms and others to be introduced and used throughout the documentation.

DOM interfaces

This guide is about the objects and the actual things you can use to manipulate the DOM hierarchy. There are many points where understanding how these work can be confusing. For example, the object representing the HTML `form` element gets its name property from the `HTMLFormElement` interface but its `className` property from the `HTMLElement` interface. In both cases, the property you want is in that form object.

But the relationship between objects and the interfaces that they implement in the DOM can be confusing, and so this section attempts to say a little something about the actual interfaces in the DOM specification and how they are made available.

Interfaces and objects

Many objects implement several different interfaces. The table object, for example, implements a specialized `HTMLTableElement` interface, which includes such methods as `createCaption` and `insertRow`. But since it's also an HTML element, `table` implements the `Element` interface. And finally, since an HTML element is also, as far as the DOM is concerned, a node in the tree of nodes that make up the object model for an HTML or XML page, the table object also implements the more basic `Node` interface, from which `Element` derives.

When you get a reference to a `table` object, as in the following example, you routinely use all three of these interfaces interchangeably on the object, perhaps without knowing it.

```
const table = document.getElementById("table");
const tableAttrs = table.attributes; // Node/Element interface
for (let i = 0; i < tableAttrs.length; i++) {
  // HTMLTableElement interface: border attribute
  if (tableAttrs[i].nodeName.toLowerCase() === "border") {
    table.border = "1";
  }
}
// HTMLTableElement interface: summary attribute
table.summary = "note: increased border";
```

Core interfaces in the DOM

This section lists some of the most commonly-used interfaces in the DOM. The idea is not to describe what these APIs do here but to give you an idea of the sorts of methods and properties you will see very often as you use the DOM.

The `document` and `window` objects are the objects whose interfaces you generally use most often in DOM programming. In simple terms, **the `window` object represents something like the browser, and the `document` object is the root of the document itself**.

`Element` **inherits from the generic `Node` interface, and together these two interfaces provide many of the methods and properties you use on individual elements.** These elements may also have specific interfaces for dealing with the kind of data those elements hold, as in the `table` object example in the previous section.

The following is a brief list of common APIs in web and XML page scripting using the DOM.

- `document.querySelector()`
- `document.querySelectorAll()`
- `document.createElement()`
- `Element.innerHTML`
- `Element.setAttribute()`
- `Element.getAttribute()`
- `EventTarget.addEventListener()`
- `HTMLElement.style`
- `Node.appendChild()`
- `window.onload`
- `window.scrollTo()`

Using the Document Object Model

The Document Object Model (DOM) is an API for manipulating DOM trees of HTML and XML documents (among other tree-like documents). This API is at the root of the description of a page and serves as a base for scripting on the web.
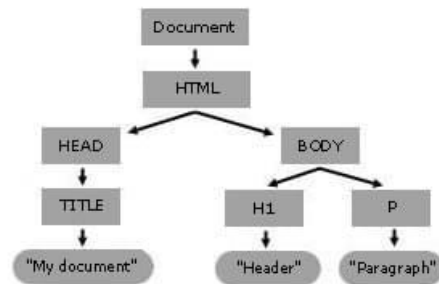
What is a DOM tree?

A DOM tree is a tree structure whose nodes represent an HTML or XML document's contents. Each HTML or XML document has a DOM tree representation. For example, consider the following document:

```
<html lang="en">
```

```
   <head>
     <title>My Document</title>
   </head>
   <body>
     <h1>Header</h1>
     <p>Paragraph</p>
   </body>
</html>
```

It has a DOM tree that looks like this:

The tree-like structure shown in the example may look similar to the original HTML or XML document, but it might not be exactly identical due to whitespace differences.

When a browser parsing an HTML or XML document to create a DOM tree, the DOM representation is designed to capture the structural hierarchy and relationships between elements in the document. However, the actual DOM tree might not preserve all the details of the original document, especially when it comes to whitespace.

Whitespace in this context refers to spaces, tabs, line breaks, and other characters used for formatting and indentation in the source code of the document.

So, while the visual representation of the DOM tree in the example might look similar to the original HTML or XML document, it might not be identical in terms of preserving all the whitespace details from the source code. The DOM tree focuses on the structural content of the document, not its visual appearance.

What does the Document API do?

The Document API, also known as the DOM API (Document Object Model API), is a powerful tool that allows you to manipulate the structure and content of a web page represented by a DOM tree. The DOM tree is a tree-like representation of the HTML or XML document that a web page consists of.

The Document API gives web developers the ability to create new HTML or XML documents from scratch or modify existing ones dynamically. This is done by using JavaScript to access the `document` property of the global object, which represents the current web page. The `document` object implements the Document interface and provides various methods and properties to interact with the DOM tree.

In the provided example, let's consider a simple HTML document with a header (`<h2>Header</h2>`) and a paragraph (`<p>Paragraph</p>`). The author wants to use JavaScript to change the header text and add a second paragraph dynamically when a button is clicked.

```
<html lang="en">
  <head>
    <title>My Document</title>
  </head>
  <body>
    <input type="button" value="Change this document." onclick="change()" />
    <h2>Header</h2>
    <p>Paragraph</p>
  </body>
</html>
```

```
function change() {
  // document.getElementsByTagName("h2") returns a NodeList of the <h2>
  // elements in the document, and the first is number 0:
  const header = document.getElementsByTagName("h2").item(0);

  // The firstChild of the header is a Text node:
  header.firstChild.data = "A dynamic document";

  // Now header is "A dynamic document".

  // Access the first paragraph
  const para = document.getElementsByTagName("p").item(0);
  para.firstChild.data = "This is the first paragraph.";

  // Create a new Text node for the second paragraph
  const newText = document.createTextNode("This is the second paragraph.");

  // Create a new Element to be the second paragraph
  const newElement = document.createElement("p");

  // Put the text in the paragraph
  newElement.appendChild(newText);
```

```
    // Put the paragraph on the end of the document by appending it to
    // the body (which is the parent of para)
    para.parentNode.appendChild(newElement);
}
```

DOM Traversing

DOM traversing refers to the process of navigating or moving through the Document Object Model (DOM) tree to access, find, or manipulate specific elements or nodes within a web page. The DOM tree is a hierarchical representation of the HTML or XML document's structure, and traversing allows developers to interact with and manipulate its nodes dynamically.

DOM traversing is commonly performed using various DOM methods and properties that allow you to move between parent and child nodes, sibling nodes, or find elements based on specific criteria. Some of the most frequently used DOM traversing methods and properties include:

1. `getElementById`: Retrieves an element by its unique ID attribute.
2. `querySelector`: Finds the first element that matches a specified CSS selector.
3. `querySelectorAll`: Retrieves all elements that match a specified CSS selector.
4. `parentNode`: Accesses the parent node of a given element.
5. `childNodes`: Returns a collection of all child nodes of an element (including text nodes, element nodes, and comment nodes).
6. `firstChild`: Gets the first child node of an element.
7. `lastChild:` Gets the last child node of an element.
8. `nextSibling`: Accesses the next sibling node (i.e., the next node at the same level within the parent).
9. `previousSibling`: Accesses the previous sibling node (i.e., the previous node at the same level within the parent).

DOM traversing is often used in JavaScript to perform various tasks, such as:

1. Modifying the content of specific elements.
2. Adding or removing elements dynamically.
3. Traversing the DOM to find and interact with elements based on user interactions (e.g., clicking a button).
4. Finding and selecting elements for specific actions, like form validation.

By using DOM traversing methods and properties effectively, developers can create dynamic and interactive web pages, as well as perform various manipulations on the document's structure and content to provide a rich user experience.

Locating DOM elements using selectors

The Selectors API provides methods that make it quick and easy to retrieve `Element` nodes from the DOM by matching against a set of selectors. This is much faster than past techniques, wherein it was necessary to, for example, use a loop in JavaScript code to locate the specific items you needed to find.

The NodeSelector interface

This specification adds two new methods to any objects implementing the `Document`, `DocumentFragment`, or `Element` interfaces:

`querySelector()`
Returns the first matching `Element` node within the node's subtree. If no matching node is found, `null` is returned.

`querySelectorAll()`
Returns a `NodeList` containing all matching `Element` nodes within the node's subtree, or an empty `NodeList` if no matches are found.

Note: The `NodeList` returned by `querySelectorAll()` is not live, which means that changes in the DOM are not reflected in the collection. This is different from other DOM querying methods that return live node lists. To return live `NodeList` we need to use methods like `document.getElementsByTagName()`, `document.getElementsByClassName()`, `element.getElementsByTagName()`, `element.getElementsByClassName()`.

Selectors

The selector methods accept selectors to determine what element or elements should be returned. This includes selector lists so you can group multiple selectors in a single query.

To protect the user's privacy, some pseudo-classes are not supported or behave differently. For example `:visited` will return no matches and `:link` is treated as `:any-link`.

Only elements can be selected, so pseudo-classes are not supported.

Examples

To select all paragraph (`p`) elements in a document whose classes include `warning` or `note`, you can do the following:

```
const special = document.querySelectorAll("p.warning, p.note");
```

You can also query by ID. For example:

```
const el = document.querySelector("#main, #basic, #exclamation");
```

After executing the above code, `el` contains the first element in the document whose ID is one of `main`, `basic`, or `exclamation`.

How to create a DOM tree (XML example)

This page describes how to use the DOM API in JavaScript to create XML documents.

Consider the following XML document:

```
<?xml version="1.0"?>
<people>
  <person first-name="eric" middle-initial="H" last-name="jung">
    <address street="321 south st" city="denver" state="co" country="usa"/>
    <address street="123 main st" city="arlington" state="ma" country="usa"/>
  </person>

  <person first-name="jed" last-name="brown">
    <address street="321 north st" city="atlanta" state="ga" country="usa"/>
  </person>
</people>
```

You can use the DOM API to create an in-memory representation of this document:

```
const doc = document.implementation.createDocument("", "", null);
const peopleElem = doc.createElement("people");

const personElem1 = doc.createElement("person");
personElem1.setAttribute("first-name", "eric");
personElem1.setAttribute("middle-initial", "h");
personElem1.setAttribute("last-name", "jung");

const addressElem1 = doc.createElement("address");
addressElem1.setAttribute("street", "321 south st");
addressElem1.setAttribute("city", "denver");
addressElem1.setAttribute("state", "co");
addressElem1.setAttribute("country", "usa");
personElem1.appendChild(addressElem1);

const addressElem2 = doc.createElement("address");
addressElem2.setAttribute("street", "123 main st");
addressElem2.setAttribute("city", "arlington");
addressElem2.setAttribute("state", "ma");
addressElem2.setAttribute("country", "usa");
personElem1.appendChild(addressElem2);

const personElem2 = doc.createElement("person");
personElem2.setAttribute("first-name", "jed");
personElem2.setAttribute("last-name", "brown");

const addressElem3 = doc.createElement("address");
addressElem3.setAttribute("street", "321 north st");
addressElem3.setAttribute("city", "atlanta");
addressElem3.setAttribute("state", "ga");
addressElem3.setAttribute("country", "usa");
personElem2.appendChild(addressElem3);

peopleElem.appendChild(personElem1);
peopleElem.appendChild(personElem2);
doc.appendChild(peopleElem);
```

DOM trees can be:

- queried using XPath expressions
- converted to strings using XMLSerializer
- posted to a web server using XMLHttpRequest
- transformed using XSLT or XLink.

How whitespace is handled by HTML, CSS, and in the DOM

The presence of whitespace in the DOM can cause layout problems and make manipulation of the content tree difficult in unexpected ways, depending on where it is located. This article explores when difficulties can occur, and looks at what can be done to mitigate resulting problems.

What is whitespace?

Whitespace is any **string of text composed only of spaces, tabs or line breaks** (to be precise, CRLF sequences, carriage returns or line feeds). These characters allow you to format your code in a way that will make it easily readable by yourself and other people. In fact, much of our source code is full of these whitespace characters, and we only tend to get rid of it in a production build step to reduce code download sizes.

HTML largely ignores whitespace?

In the case of HTML, whitespace is largely ignored — whitespace in between words is treated as a single character, and whitespace at the start and end of elements and outside elements is ignored. Take the following minimal example:

```
<!DOCTYPE html>

  <h1>      Hello      World!      </h1>
```

This source code contains a couple of line feeds after the DOCTYPE and a bunch of space characters before, after, and inside the <h1> element, but the browser doesn't seem to care at all and just shows the words "Hello World!" as if these characters didn't exist at all:

Hello World!

This is so that whitespace characters don't impact the layout of your page. Creating space around and inside elements is the job of CSS.

What happens to whitespace?
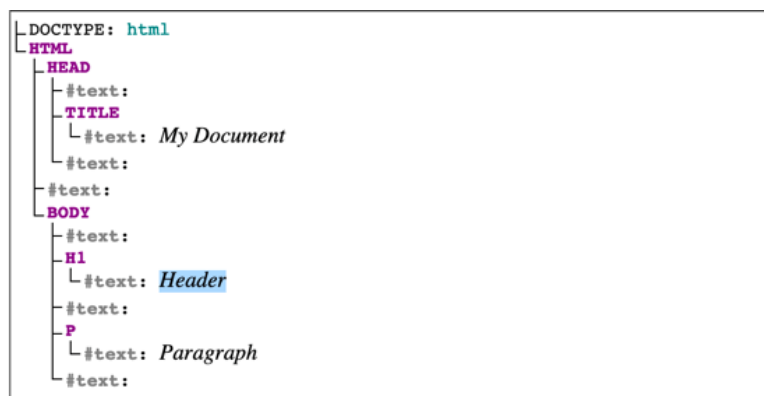
They don't just disappear, however.

Any whitespace characters that are outside of HTML elements in the original document are represented in the DOM. This is needed internally so that the editor can preserve the formatting of documents. This means that:

- There will be some text nodes that contain only whitespace, and
- Some text nodes will have whitespace at the beginning or end.

Take the following document, for example:

```
<!doctype html>
<html lang="en-US">
  <head>
    <meta charset="UTF-8" />
    <title>My Document</title>
  </head>
  <body>
    <h1>Header</h1>
    <p>Paragraph</p>
  </body>
</html>
```

The DOM tree for this looks like so:

Conserving whitespace characters in the DOM is useful in many ways, but there are certain places where this makes certain layouts more difficult to implement and causes problems for developers who want to iterate through nodes in the DOM.

How does CSS process whitespace?

Most whitespace characters are ignored, not all of them are. In the earlier example one of the spaces between "Hello" and "World!" still exists when the page is rendered in a browser. There

are rules in the browser engine that decide which whitespace characters are useful and which aren't — these are specified at least in part in CSS Text Module Level 3, and especially the parts about the CSS white-space property and whitespace processing details, but we also offer an easier explanation below.

<div align="center">Example</div>

Let's take another example. To make it easier, we've added a comment that shows all spaces with ◦, all tabs with →, and all line breaks with ⏎:

This example:

```
<h1>   Hello
      <span> World!</span>    </h1>

<!--
<h1>◦◦◦Hello◦⏎
→→→→<span>◦World!</span>→◦◦</h1>
-->
```

is rendered in the browser like so:

Hello World!

<div align="center">Explanation</div>

The `<h1>` element contains only inline elements. In fact it contains:

- A text node (consisting of some spaces, the word "Hello" and some tabs).
- An inline element (the `<span>`, which contains a space, and the word "World!").
- Another text node (consisting only of tabs and spaces).

Because of this, it establishes what is called an inline formatting context. This is one of the possible layout rendering contexts that browser engines work with.

Inside this context, whitespace character processing can be summarized as follows:

1. First, all spaces and tabs immediately before and after a line break are ignored so, if we take our example markup from before:

```
   <h1>∘∘∘Hello∘⏎
→→→→<span>∘World!</span>→∘∘</h1>
```

...and apply this first rule, we get:

```
   <h1>∘∘∘Hello⏎
<span>∘World!</span>→∘∘</h1>
```

2. Next, all tab characters are handled as space characters, so the example becomes:

```
   <h1>∘∘∘Hello⏎
<span>∘World!</span>∘∘∘</h1>
```

3. Next, line breaks are converted to spaces:

```
   <h1>∘∘∘Hello∘<span>∘World!</span>∘∘∘</h1>
```

4. After that, any space immediately following another space (even across two separate inline elements) is ignored, so we end up with:

```
   <h1>∘Hello∘<span>World!</span>∘</h1>
```

5. And finally, sequences of spaces at the beginning and end of an element are removed, so we finally get this:

```
   <h1>Hello∘<span>World!</span></h1>
```

This is why people visiting the web page will see the phrase "Hello World!" nicely written at the top of the page, rather than a weirdly indented "Hello" followed but an even more weirdly indented "World!" on the line below that.

Whitespace in block formatting contexts

Above we just looked at elements that contain inline elements, and inline formatting contexts. If an element contains at least one block element, then it instead establishes what is called a block formatting context.

Within this context, whitespace is treated very differently.

Example

Let's take a look at an example to explain how. We've marked the whitespace characters as before.

We have 3 text nodes that contain only whitespace, one before the first `<div>`, one between the 2 `<div>`s, and one after the second `<div>`.

```
<body>
  <div>  Hello  </div>

    <div>  World!    </div>
</body>

<!--
<body>⏎
→<div>∘∘Hello∘∘</div>⏎
⏎
  ∘∘∘<div>∘∘World!∘∘</div>∘∘⏎
</body>
-->
```

This renders like so:

Hello
World!

Explanation

We can summarize how the whitespace here is handled as follows (there may be some slight differences in exact behavior between browsers, but this basically works):

1. Because we're inside a block formatting context, everything must be a block, so our 3 text nodes also become blocks, just like the 2 <div>s. Blocks occupy the full width available and are stacked on top of each other, which means that, starting from the example above:

351

```
    <body>⏎
→<div>∘∘Hello∘∘</div>⏎
⏎
  ∘∘∘<div>∘∘World!∘∘</div>∘∘⏎
  </body>
```

...we end up with a layout composed of this list of blocks:

```
<block>⏎→</block>
<block>∘∘Hello∘∘</block>
<block>⏎∘∘∘</block>
<block>∘∘World!∘∘</block>
<block>∘∘⏎</block>
```

2. This is then simplified further by applying the processing rules for whitespace in inline formatting contexts to these blocks:

```
<block></block>
<block>Hello</block>
<block></block>
<block>World!</block>
<block></block>
```

3. The 3 empty blocks we now have are not going to occupy any space in the final layout, because they don't contain anything, so we'll end up with only 2 blocks taking up space in the page. People viewing the web page see the words "Hello" and "World!" on 2 separate lines as you'd expect 2 <div>s to be laid out. The browser engine has essentially ignored all of the whitespace that was added in the source code.

Spaces in between inline and inline-block elements

Let's move on to look at a few issues that can arise due to whitespace, and what can be done about them. First of all, we'll look at what happens with spaces in between inline and inline-block elements. In fact, we saw this already in our very first example, when we described how whitespace is processed inside inline formatting contexts.

We said that there were rules to ignore most characters but that word-separating characters remain. When you're only dealing with block-level elements such as <p> that only contain inline

elements such as `<em>`, `<strong>`, `<span>`, etc., you don't normally care about this because the extra whitespace that does make it to the layout is helpful to separate the words in the sentence.

It gets more interesting however when you start using inline-block elements. These elements behave like inline elements on the outside, and blocks on the inside, and are often used to display more complex pieces of UI than just text, side-by-side on the same line, for example navigation menu items.

Because they are blocks, many people expect that they will behave as such, but really they don't. If there is formatting whitespace between adjacent inline elements, this will result in space in the layout, just like the spaces between words in text.

<div align="center">Example</div>

Consider this example (again, we've included an HTML comment that shows the whitespace characters in the HTML):

```
.people-list {
  list-style-type: none;
  margin: 0;
  padding: 0;
}

.people-list li {
  display: inline-block;
  width: 2em;
  height: 2em;
  background: #f06;
  border: 1px solid;
}
```

```
<ul class="people-list">
  <li></li>

  <li></li>

  <li></li>

  <li></li>

  <li></li>
</ul>

<!--
```

```
  <ul class="people-list">⏎
  ∘∘<li></li>⏎
⏎
  ∘∘<li></li>⏎
⏎
  ∘∘<li></li>⏎
⏎
  ∘∘<li></li>⏎
⏎
  ∘∘<li></li>⏎
  </ul>
  -->
```
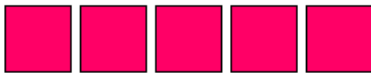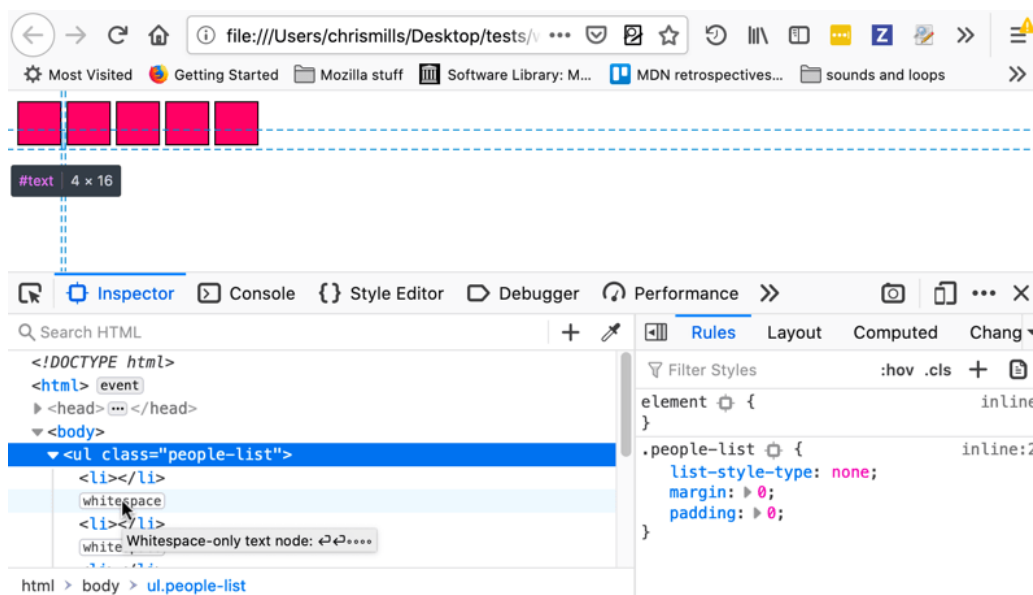

You probably don't want the gaps in between the blocks — depending on the use case (is this a list of avatars, or horizontal nav buttons?), you probably want the element sides flush with each other, and to be able to control any spacing yourself.

The Firefox DevTools HTML Inspector will highlight text nodes, and also show you exactly what area the elements are taking up — useful if you are wondering what is causing the problem.

Solutions

There are a few ways of getting around this problem:

Use Flexbox to create the horizontal list of items instead of trying an inline-block solution. This handles everything for you, and is definitely the preferred solution:

```
ul {
  list-style-type: none;
  margin: 0;
  padding: 0;
  display: flex;
}
```

If you need to rely on `inline-block`, you could set the `font-size` of the list to 0. This only works if your blocks are not sized with ems (based on the font-size, so the block size would also end up being 0). rems would be a good choice here:

```
ul {
  font-size: 0;
  /* … */
}

li {
  display: inline-block;
  width: 2rem;
  height: 2rem;
  /* … */
}
```

Or you could set negative margin on the list items:

```
li {
  display: inline-block;
  width: 2rem;
  height: 2rem;
  margin-right: -0.25rem;
}
```

You can also solve this problem by putting your list items all on the same line in the source, which causes the whitespace nodes to not be created in the first place:

```
<li></li><li></li><li></li><li></li><li></li>
```

DOM traversal and whitespace

When trying to do DOM manipulation in JavaScript, you can also encounter problems because of whitespace nodes. For example, if you have a reference to a parent node and want to affect its first element child using `Node.firstChild`, if there is a rogue whitespace node just after the opening parent tag you will not get the result you are expecting. The text node would be selected instead of the element you want to affect.

As another example, if you have a certain subset of elements that you want to do something to based on whether they are empty (have no child nodes) or not, you could check whether each element is empty using something like `Node.hasChildNodes()`, but again, if any target elements contain text nodes, you could end up with false results.

Whitespace helper functions

The JavaScript code below defines several functions that make it easier to deal with whitespace in the DOM:

```
/**
 * Throughout, whitespace is defined as one of the characters
 *   "\t" TAB \u0009
 *   "\n" LF  \u000A
 *   "\r" CR  \u000D
 *   " "  SPC \u0020
 *
 * This does not use JavaScript's "\s" because that includes non-breaking
 * spaces (and also some other characters).
 */

/**
 * Determine whether a node's text content is entirely whitespace.
 *
 * @param nod  A node implementing the |CharacterData| interface (i.e.,
 *             a |Text|, |Comment|, or |CDATASection| node
 * @return     True if all of the text content of |nod| is whitespace,
 *             otherwise false.
 */
function is_all_ws(nod) {
  return !/[^\t\n\r ]/.test(nod.textContent);
}
```

```
/**
 * Determine if a node should be ignored by the iterator functions.
 *
 * @param nod   An object implementing the DOM1 |Node| interface.
 * @return      true if the node is:
 *                  1) A |Text| node that is all whitespace
 *                  2) A |Comment| node
 *              and otherwise false.
 */

function is_ignorable(nod) {
  return (
    nod.nodeType === 8 || // A comment node
    (nod.nodeType === 3 && is_all_ws(nod))
  ); // a text node, all ws
}

/**
 * Version of |previousSibling| that skips nodes that are entirely
 * whitespace or comments. (Normally |previousSibling| is a property
 * of all DOM nodes that gives the sibling node, the node that is
 * a child of the same parent, that occurs immediately before the
 * reference node.)
 *
 * @param sib  The reference node.
 * @return     Either:
 *                 1) The closest previous sibling to |sib| that is not
 *                    ignorable according to |is_ignorable|, or
 *                 2) null if no such node exists.
 */
function node_before(sib) {
  while ((sib = sib.previousSibling)) {
    if (!is_ignorable(sib)) {
      return sib;
    }
  }
  return null;
}

/**
 * Version of |nextSibling| that skips nodes that are entirely
 * whitespace or comments.
 *
 * @param sib  The reference node.
 * @return     Either:
 *                 1) The closest next sibling to |sib| that is not
 *                    ignorable according to |is_ignorable|, or
 *                 2) null if no such node exists.
 */
function node_after(sib) {
```

```
  while ((sib = sib.nextSibling)) {
    if (!is_ignorable(sib)) {
      return sib;
    }
  }
  return null;
}

/**
 * Version of |lastChild| that skips nodes that are entirely
 * whitespace or comments. (Normally |lastChild| is a property
 * of all DOM nodes that gives the last of the nodes contained
 * directly in the reference node.)
 *
 * @param sib  The reference node.
 * @return     Either:
 *                1) The last child of |sib| that is not
 *                   ignorable according to |is_ignorable|, or
 *                2) null if no such node exists.
 */
function last_child(par) {
  let res = par.lastChild;
  while (res) {
    if (!is_ignorable(res)) {
      return res;
    }
    res = res.previousSibling;
  }
  return null;
}

/**
 * Version of |firstChild| that skips nodes that are entirely
 * whitespace and comments.
 *
 * @param sib  The reference node.
 * @return     Either:
 *                1) The first child of |sib| that is not
 *                   ignorable according to |is_ignorable|, or
 *                2) null if no such node exists.
 */
function first_child(par) {
  let res = par.firstChild;
  while (res) {
    if (!is_ignorable(res)) {
      return res;
    }
    res = res.nextSibling;
  }
  return null;
}
```

```
/**
 * Version of |data| that doesn't include whitespace at the beginning
 * and end and normalizes all whitespace to a single space. (Normally
 * |data| is a property of text nodes that gives the text of the node.)
 *
 * @param txt   The text node whose data should be returned
 * @return      A string giving the contents of the text node with
 *              whitespace collapsed.
 */
function data_of(txt) {
  let data = txt.textContent;
  data = data.replace(/[\t\n\r ]+/g, " ");
  if (data[0] === " ") {
    data = data.substring(1, data.length);
  }
  if (data[data.length - 1] === " ") {
    data = data.substring(0, data.length - 1);
  }
  return data;
}
```

<div align="center">Example</div>

The following code demonstrates the use of the functions above. It iterates over the children of an element (whose children are all elements) to find the one whose text is "`This is the third paragraph`", and then changes the class attribute and the contents of that paragraph.

```
let cur = first_child(document.getElementById("test"));
while (cur) {
  if (data_of(cur.firstChild) === "This is the third paragraph.") {
    cur.className = "magic";
    cur.firstChild.textContent = "This is the magic paragraph.";
  }
  cur = node_after(cur);
}
```

This lesson was adapted from MDN Web Docs:
https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction
https://developer.mozilla.org/en-US/docs/Web/API/Document_object_model/Using_the_Document_Object_Model
https://developer.mozilla.org/en-US/docs/Web/API/Document_object_model/Locating_DOM_elements_using_selectors
https://developer.mozilla.org/en-US/docs/Web/API/Document_object_model/How_to_create_a_DOM_tree
https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Whitespace

Introduction to events

Events are **things that happen** in the system you are programming, which the system tells you about **so your code can react to them**.

For example, if the user clicks a button on a webpage, you might want to react to that action by displaying an information box. In this article, we discuss some important concepts surrounding events, and look at how they work in browsers. This won't be an exhaustive study; just what you need to know at this stage.

What is an event?

Events are things that happen in the system you are programming — the system produces (or "fires") a signal of some kind when an event occurs, and provides a mechanism by which an action can be automatically taken (that is, some code running) when the event occurs. **Events are fired inside the browser window** and tend to be attached to a specific item that resides in it. This might be a single element, a set of elements, the HTML document loaded in the current tab, or the entire browser window. There are many different types of events that can occur. There are more than 280 different events in different web technologies, libraries, or APIs.

Here are some commonly used events:

Mouse events	`click, mouseover, mouseout, mousedown, mouseup`
Keyboard events	`keydown, keyup, keypress`
Form events	`submit, input, change, focus, blur`
Window events	`load, resize, scroll, unload`
Touch events	`touchstart, touchmove, touchend`
Media events	`play ,pause, ended`
Drag and drop events	`dragstart, dragenter, dragover, drop, dragend`

To react to an event, you attach an **event handler** to it. This is a block of code (usually a JavaScript function that you as a programmer create) that runs when the event fires. When such a block of code is defined to run in response to an event, we say we are **registering an event handler**. Note: Event handlers are sometimes called event listeners — they are pretty much interchangeable for our purposes, although strictly speaking, they work together. The listener listens out for the event happening, and the handler is the code that is run in response to it happening.

An example: handling a click event

In the following example, we have a single `<button>` in the page:

```
<button>Change color</button>
```

Then we have some JavaScript. We'll look at this in more detail in the next section, but for now we can just say: it adds an event handler to the button's "`click`" event, and the handler reacts to the event by setting the page background to a random color:

```
const btn = document.querySelector("button");
function random(number) {
  return Math.floor(Math.random() * (number + 1));
}
btn.addEventListener("click", () => {
  const rndCol = `rgb(${random(255)}, ${random(255)}, ${random(255)})`;
  document.body.style.backgroundColor = rndCol;
});
```

Using addEventListener()

As we saw in the last example, objects that can fire events have an `addEventListener()` method, and this is the recommended mechanism for adding event handlers.
Let's take a closer look at the code from the last example:

```
const btn = document.querySelector("button");

function random(number) {
  return Math.floor(Math.random() * (number + 1));
}

btn.addEventListener("click", () => {
  const rndCol = `rgb(${random(255)}, ${random(255)}, ${random(255)})`;
  document.body.style.backgroundColor = rndCol;
});
```

The HTML `<button>` element will fire an event when the user clicks the button. So it defines an `addEventListener()` function, which we are calling here. We're passing in two parameters:

361

- the string "`click`", to indicate that we want to listen to the click event. Buttons can fire lots of other events, such as "`mouseover`" when the user moves their mouse over the button, or "`keydown`" when the user presses a key and the button is focused.
- a function to call when the event happens. In our case, the function generates a random RGB color and sets the `background-color` of the page `<body>` to that color.

It is fine to make the handler function a separate named function, like this:

```
const btn = document.querySelector("button");
function random(number) {
  return Math.floor(Math.random() * (number + 1));
}
function changeBackground() {
  const rndCol = `rgb(${random(255)}, ${random(255)}, ${random(255)})`;
  document.body.style.backgroundColor = rndCol;
}
btn.addEventListener("click", changeBackground);
```

Listening for other events

There are many different events that can be fired by a button element. Let's experiment. Let's take our previous example, and now try changing `click` to the following different values in turn, and observing the results in the example:

- `focus` and `blur` — The color changes when the button is focused and unfocused. These are often used to display information about filling in form fields when they are focused, or to display an error message if a form field is filled with an incorrect value.
- `dblclick` — The color changes only when the button is double-clicked.
- `mouseover` and `mouseout` — The color changes when the mouse pointer hovers over the button, or when the pointer moves off the button, respectively.

Some events, such as `click`, are available on nearly any element. Others are more specific: for example, the `play` event is available on some elements, such as `<video>`.

Removing listeners

If you've added an event handler using `addEventListener()`, you can remove it again using the `removeEventListener()` method. For example, this would remove the `changeBackground()` event handler:

```
    btn.removeEventListener("click", changeBackground);
```

Event handlers can also be removed by passing an `AbortSignal` to `addEventListener()` and then later calling `abort()` on the controller owning the `AbortSignal`. For example, to add an event handler that we can remove with an `AbortSignal`:

```
const controller = new AbortController();

btn.addEventListener("click",
  () => {
    const rndCol = `rgb(${random(255)}, ${random(255)}, ${random(255)})`;
    document.body.style.backgroundColor = rndCol;
  },
  { signal: controller.signal } // pass an AbortSignal to this handler
);
```

Then the event handler created by the code above can be removed like this:

```
  controller.abort();  //  removes  any/all  event  handlers  associated  with  this
controller
```

For simple, small programs, cleaning up old, unused event handlers aren't necessary, but for larger, more complex programs, it can improve efficiency. Also, the ability to remove event handlers allows you to have the same button performing different actions in different circumstances: all you have to do is add or remove handlers.

Adding multiple listeners for a single event

By making more than one call to `addEventListener()`, and providing different handlers, you can have multiple handlers for a single event:

```
myElement.addEventListener("click", functionA);
myElement.addEventListener("click", functionB);
```

Both functions would now run when the element is clicked.

Other event listener mechanisms

We recommend that you use `addEventListener()` to register event handlers. It's the most powerful method and scales best with more complex programs. However, there are two other ways of registering event handlers that you might see: event handler properties and inline event handlers.

Event handler properties

Objects (such as buttons) that can fire events also usually have properties whose name is on followed by the name of the event. For example, elements have a property onclick. This is called an *event handler property*. To listen for the event, you can assign the handler function to the property.

For example, we could rewrite the random-color example like this:

```
const btn = document.querySelector("button");
function random(number) {
  return Math.floor(Math.random() * (number + 1));
}

btn.onclick = () => {
  const rndCol = `rgb(${random(255)}, ${random(255)}, ${random(255)})`;
  document.body.style.backgroundColor = rndCol;
};
```

You can also set the handler property to a named function:

```
const btn = document.querySelector("button");

function random(number) {
  return Math.floor(Math.random() * (number + 1));
}

function bgChange() {
  const rndCol = `rgb(${random(255)}, ${random(255)}, ${random(255)})`;
  document.body.style.backgroundColor = rndCol;
}
```

```
btn.onclick = bgChange;
```

With event handler properties, you can't add more than one handler for a single event. For example, you can call `addEventListener('click', handler)` on an element multiple times, with different functions specified in the second argument:

```
element.addEventListener("click", function1);
element.addEventListener("click", function2);
```

This is impossible with event handler properties because any subsequent attempts to set the property will overwrite earlier ones:

```
element.onclick = function1;
element.onclick = function2;
```

Inline event handlers — don't use these

You might also see a pattern like this in your code:

```
<button onclick="bgChange()">Press me</button>
```

```
function bgChange() {
  const rndCol = `rgb(${random(255)}, ${random(255)}, ${random(255)})`;
  document.body.style.backgroundColor = rndCol;
}
```

The earliest method of registering event handlers found on the Web involved *event handler HTML attributes* (or *inline event handlers*) like the one shown above — the attribute value is literally the JavaScript code you want to run when the event occurs. The above example invokes a function defined inside a `<script>` element on the same page, but you could also insert JavaScript directly inside the attribute, for example:

```
<button onclick="alert('Hello, this is my old-fashioned event handler!');">
  Press me
</button>
```

You can find HTML attribute equivalents for many of the event handler properties; however, you shouldn't use these — they are considered bad practice. It might seem easy to use an event handler attribute if you are doing something really quick, but they quickly become unmanageable and inefficient.

For a start, it is not a good idea to mix up your HTML and your JavaScript, as it becomes hard to read. Keeping your JavaScript separate is a good practice, and if it is in a separate file you can apply it to multiple HTML documents.

Even in a single file, inline event handlers are not a good idea. One button is OK, but what if you had 100 buttons? You'd have to add 100 attributes to the file; it would quickly turn into a maintenance nightmare. With JavaScript, you could easily add an event handler function to all the buttons on the page no matter how many there were, using something like this:

```
const buttons = document.querySelectorAll("button");

for (const button of buttons) {
  button.addEventListener("click", bgChange);
}
```

Finally, many common server configurations will disallow inline JavaScript, as a security measure. **You should never use the HTML event handler attributes** — those are outdated, and using them is bad practice.

Event objects

Sometimes, inside an event handler function, you'll see a parameter specified with a name such as event, evt, or e. This is called the **event object**, and it is automatically passed to event handlers to provide extra features and information. For example, let's rewrite our random color example again slightly:

```
const btn = document.querySelector("button");

function random(number) {
  return Math.floor(Math.random() * (number + 1));
}

function bgChange(e) {
  const rndCol = `rgb(${random(255)}, ${random(255)}, ${random(255)})`;
```

366

```
    e.target.style.backgroundColor = rndCol;
    console.log(e);
}

btn.addEventListener("click", bgChange);
```

Here you can see we are including an event object, **e**, in the function, and in the function setting a background color style on `e.target` — which is the button itself. The target property of the event object is always a reference to the element the event occurred upon. So, in this example, we are setting a random background color on the button, not the page.

Extra properties of event objects

Most event objects have a standard set of properties and methods available on the event object; see the `Event` object reference for a full list.

Some event objects add extra properties that are relevant to that particular type of event. For example, the `keydown` event fires when the user presses a key. Its event object is a `KeyboardEvent`, which is a specialized Event object with a key property that tells you which key was pressed:

```
<input id="textBox" type="text" />
<div id="output"></div>
```

```
const textBox = document.querySelector("#textBox");
const output = document.querySelector("#output");
textBox.addEventListener("keydown", (event) => {
  output.textContent = `You pressed "${event.key}".`;
});
```

In this example when you type in a box and in text below your last pressed key will be visible.

Preventing default behavior

Sometimes, you'll come across a situation where you want to prevent an event from doing what it does by default. The most common example is that of a web form. When you fill in the details and click the submit button, the natural behavior is for the data to be submitted to a specified

page on the server for processing, and the browser to be redirected to a "success message" page of some kind (or the same page, if another is not specified).

The trouble comes when the user has not submitted the data correctly — as a developer, you want to prevent the submission to the server and give an error message saying what's wrong and what needs to be done to put things right. Some browsers support automatic form data validation features, but many don't.

First, a simple HTML form that requires you to enter your first and last name:

```html
<form>
  <div>
    <label for="fname">First name: </label>
    <input id="fname" type="text" />
  </div>
  <div>
    <label for="lname">Last name: </label>
    <input id="lname" type="text" />
  </div>
  <div>
    <input id="submit" type="submit" />
  </div>
</form>
<p></p>
```

Here we implement simple check inside a handler for the `submit` event (the submit event is fired on a form when it is submitted) that tests whether the text fields are empty. If they are, we call the `preventDefault()` function on the event object — which stops the form submission — and then display an error message in the paragraph below our form to tell the user what's wrong:

```javascript
const form = document.querySelector("form");
const fname = document.getElementById("fname");
const lname = document.getElementById("lname");
const para = document.querySelector("p");

form.addEventListener("submit", (e) => {
  if (fname.value === "" || lname.value === "") {
    e.preventDefault();
    para.textContent = "You need to fill in both names!";
  }
});
```

Obviously, this is pretty weak form validation — it wouldn't stop the user from validating the form with spaces or numbers entered into the fields, for example — but it is OK for example purposes.

Event bubbling

Event bubbling is a mechanism in JavaScript where when an event is triggered on a nested element, the same event is also triggered on its parent elements all the way up to the root element. It's like a bubble that rises up from the innermost element to the outermost ancestor. This allows you to handle events at higher levels in the DOM hierarchy without attaching individual event listeners to each nested element.

Let's go through an example of event bubbling:

```
<div id="outer">
  <div id="middle">
    <div id="inner">Click me!</div>
  </div>
</div>
```

```
const innerElement = document.getElementById("inner");
const middleElement = document.getElementById("middle");
const outerElement = document.getElementById("outer");

innerElement.addEventListener("click", function (event) {
  console.log("Inner element clicked!");
});

middleElement.addEventListener("click", function (event) {
  console.log("Middle element clicked!");
});

outerElement.addEventListener("click", function (event) {
  console.log("Outer element clicked!");
});
```

In this example, clicking "Click me!" will show the following console output:

```
Inner element clicked!
Middle element clicked!
Outer element clicked!
```

Fixing the problem with stopPropagation()

Now, let's talk about `stopPropagation()`. It is a method available on the event object, and when called within an event handler, it **prevents the event from bubbling up to its parent elements**. This means that if `stopPropagation()` is used within the innermost element's event handler, the event won't reach the middle and outer elements.

Here's the modified code using `stopPropagation()`:

```javascript
const innerElement = document.getElementById("inner");
const middleElement = document.getElementById("middle");
const outerElement = document.getElementById("outer");

innerElement.addEventListener("click", function (event) {
  console.log("Inner element clicked!");
  event.stopPropagation(); // Stops the event from bubbling up further
});

middleElement.addEventListener("click", function (event) {
  console.log("Middle element clicked!");
});

outerElement.addEventListener("click", function (event) {
  console.log("Outer element clicked!");
});
```

Now, when you click on the "Click me!" text, you will see the following output in the console:

```
Inner element clicked!
```

The event stops at the innermost element and doesn't bubble up to the middle and outer elements.

Event capture

As for Event Capture, it is an alternative event handling phase to event bubbling. In Event Capture, the event is first captured at the root element and then propagated down to the target element. It is the **reverse of event bubbling**.

Here's an example using Event Capture:

```javascript
const innerElement = document.getElementById("inner");
const middleElement = document.getElementById("middle");
const outerElement = document.getElementById("outer");

innerElement.addEventListener(
  "click",
  function (event) {
    console.log("Inner element clicked!");
  },
  true // Adding 'true' as the third parameter enables Event Capture
);

middleElement.addEventListener(
  "click",
  function (event) {
    console.log("Middle element clicked!");
  },
  true
);

outerElement.addEventListener(
  "click",
  function (event) {
    console.log("Outer element clicked!");
  },
  true
);
```

Now, when you click on the "Click me!" text, you will see the following output in the console:

```
Outer element clicked!
Middle element clicked!
Inner element clicked!
```

The event starts capturing from the outermost element and propagates down to the innermost element, giving you the reverse order compared to event bubbling.

Event Capture is less commonly used than event bubbling, but it provides an alternative way to handle events in specific scenarios where you need to intercept events as they flow down the DOM hierarchy.

Event delegation

Event delegation is a design pattern in JavaScript where **instead of attaching event listeners to individual elements, you attach a single event listener to a parent element** that encompasses all the child elements you are interested in. Then, you use event bubbling to handle the events as they propagate up to the parent element. This approach is particularly useful when you have a large number of elements with similar behavior, as it helps improve performance and simplifies event handling code.

Here's an example to illustrate event delegation:

```
<ul id="myList">
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
  <li>Item 4</li>
  <li>Item 5</li>
</ul>
```

```
const list = document.getElementById("myList");

list.addEventListener("click", function (event) {
  if (event.target.tagName === "LI") {
    // Check if the clicked element is an <li> element
    console.log("Clicked on:", event.target.textContent);
  }
});
```

In this example, we have a list (`<ul>`) with multiple list items (`<li>`). Instead of attaching individual event listeners to each list item, we attach a single event listener to the parent `<ul>` element. When you click on any list item, the event bubbles up to the `<ul>` element, and the event listener checks if the clicked target (`event.target`) is an `<li>` element. If it is, it logs the text content of the clicked list item to the console.

The event delegation technique here allows us to handle clicks on any list item without adding separate event listeners to each of them. This is particularly beneficial when dealing with dynamic content where new list items can be added or removed, and you don't need to worry about attaching new event listeners to the new elements.

Event delegation is a powerful technique that promotes cleaner and more efficient code, especially when working with large sets of similar elements. However, it's essential to ensure that the parent element chosen for delegation is stable and present in the DOM at the time of attaching the event listener.

It's not just web pages

Events are not unique to JavaScript — most programming languages have some kind of event model, and the way the model works often differs from JavaScript's way. In fact, the event model in JavaScript for web pages differs from the event model for JavaScript as it is used in other environments.

For example, Node.js is a very popular JavaScript runtime that enables developers to use JavaScript to build network and server-side applications. The Node.js event model relies on listeners to listen for events and emitters to emit events periodically — it doesn't sound that different, but the code is quite different, making use of functions like `on()` to register an event listener, and `once()` to register an event listener that unregisters after it has run once.

You can also use JavaScript to build cross-browser add-ons — browser functionality enhancements — using a technology called WebExtensions. The event model is similar to the web events model, but a bit different — event listeners' properties are camel-cased (such as `onMessage` rather than `onmessage`), and need to be combined with the `addListener` function.