JavaScript ES6 to 2023 versions

ES6 (ECMAScript 2015) is already a relatively old version of JavaScript, but it laid the foundation for many modern JavaScript features and syntax enhancements. Since then, several new versions of ECMAScript have been released, with ES2023 being a future version beyond ES2022.

overview of some key features introduced in ECMAScript versions after ES6 up to ES2022:

ES5 (ES2009):

- "use strict"
- String[*number*] access
- Multiline strings
- String.trim()
- Array.isArray()
- Array forEach()
- Array map()
- Array filter()
- Array reduce()
- Array reduceRight()
- Array every()
- Array some()
- Array indexOf()
- Array lastIndexOf()
- JSON.parse()
- JSON.stringify()
- Date.now()
- Date toISOString()
- Date toJSON()
- Property getters and setters
- Reserved words as property names
- Object.create()
- Object.keys()
- Object management
- Object protection

- Object defineProperty()
- Function bind()
- Trailing commas

ES6 (ES2015):

- The let keyword
- The const keyword
- Arrow Functions
- The {a,b} = Operator
- The [a,b] = Operator
- The ... Operator
- For/of
- Map Objects
- Set Objects
- Classes
- Promises
- Symbol
- Default Parameters
- Function Rest Parameter
- String.includes()
- String.startsWith()
- String.endsWith()
- Array entries()
- Array.from()
- Array keys()
- Array find()
- Array findIndex()
- Math.trunc
- Math.sign
- Math.cbrt
- Math.log2
- Math.log10
- Number.EPSILON
- Number.MIN_SAFE_INTEGER

- Number.MAX_SAFE_INTEGER
- Number.isInteger()
- Number.isSafeInteger()
- New Global Methods
- JavaScript Modules

ES7 (ES2016):

- Array.prototype.includes(): A method to determine whether an array includes a certain element.
- Exponentiation Operator (**): A shorthand for calculating exponentiation.

- JavaScript Exponentiation (**)
- JavaScript Exponentiation assignment (**=)
- JavaScript Array includes()

ES8 (ES2017):

- Async Functions: Provides a simpler syntax for writing asynchronous code using async/await keywords.
- Shared Memory and Atomics: Introduces shared memory and atomic operations for multi-threading capabilities (part of ECMAScript 2017 draft).
- JavaScript String padding
- JavaScript Object entries()
- JavaScript Object values()
- JavaScript async and await
- Trailing Commas in Functions
- JavaScript Object.getOwnPropertyDescriptors

ES9 (ES2018):

- Asynchronous Iteration: Adds support for asynchronous iteration using for-await-of loop.
- Rest/Spread Properties: Allows spreading properties of an object.

- Promise.prototype.finally(): Adds a finally method to Promise instances.
- Asynchronous Iteration
- Promise Finally
- Object Rest Properties
- New RegExp Features
- JavaScript Shared Memory

ES10 (ES2019):

- Array.prototype.flat() and Array.prototype.flatMap(): Provides methods to flatten and map arrays.
- Object.fromEntries(): Converts a list of key-value pairs into an object.
- Optional Catch Binding: Allows using a catch block without declaring an error parameter.
- String.trimStart()
- String.trimEnd()
- Object.fromEntries
- Optional catch binding
- Array.flat()
- Array.flatMap()
- Revised Array.Sort()
- Revised JSON.stringify()
- Separator symbols allowed in string litterals
- Revised Function.toString()

ES11 (ES2020):

- Optional Chaining (?.): Provides a way to access nested object properties without worrying about null or undefined references.
- Nullish Coalescing Operator (??): A logical operator that returns its right-hand operand when its left-hand operand is null or undefined.
- BigInt: A new primitive type for working with arbitrarily large integers.
- BigInt
- String matchAll()
- The Nullish Coalescing Operator (??)
- The Optional Chaining Operator (?.)

- Logical AND Assignment Operator (&&=)
- Logical OR Assignment (||=)
- Nullish Coalescing Assignment (??=)
- Promise.allSettled()
- Dynamic Import

ES12 (ES2021):

- String.prototype.replaceAll(): Replaces all occurrences of a substring within a string.
- Logical Assignment Operators (||=, &&=, ??=): Provide shorthand assignment syntax for logical operations.
- Numeric Separators: Allows inserting underscores (_) as separators in numeric literals for better readability.
- Promise.any()
- String replaceAll()
- Numeric Separators (_)

ES13 (ES2022):

- Class static initialization blocks: Allows static initialization in class definitions.
- Error Cause: Allows attaching a cause to an Error object.
- Array at()
- String at()
- RegExp /d
- Object.hasOwn()
- error.cause
- await import
- Class field declarations
- Private methods and fields

ES14 (ES2023):

- Array findLast()
- Array findLastIndex()
- Array toReversed()
- Array toSorted()

- Array toSpliced()
- Array with()
- #! (Shebang)

These features are relatively new.

Older browsers may need an alternative code (Polyfill)

https://www.w3schools.com/Js/js_2023.asp

ES15 (ES2024):

- Object.groupBy()
- Map.groupBy()
- Temporal.PlainDate()
- Temporal.PlainTime()
- Temporal.PlainMonthDay()
- Temporal.PlainYearMonth()

Warning

These features are relatively new.

Older browsers may need an alternative code (Polyfill)

Introduced let, const where var is global scope

modular code writing

classes, improved readability from prototype to classes

with nodeJS can write UI and server side code as well

What awaits JavaScript in future:

TC39(Technical committee 39) id responsible for analyzing proposals. Proposals are the contributions from the community to add new features to JavaScript. Some of the popular proposals are

Temporal API, import attributes, pipeline operator

Teporal API:

which is currently in stage 3 , is being developed to improve the current Date object, which is mostly known for its unexpected behaviour. Today there are lots of date-time libraries for JS, such as, data-fns, moments, js-joda and a huge number of others.They all try to help with unpredictable and unexpected behaviour of JS Date object by adding features such as timezones , date parsing and almost everything else.

with different object like Temporal.TimeZone, Temporal.PlainDate and others, Temporal API is trying to address all these issues and replace JS Date object. You can start testing the API using an npm package named @js-temporal/polyfill

import attributes:

aims to improve the import statement in JS by allowing developers to assert certain conditions about the import module. This can help catch errors at compile-time instead of runtime and make the code more robust. Currentl this proposal is in Stage 3.

With import attributes, you can specify the type of the imported module or ensure that a Specific version of the module used.

```
import json from "./foo.json" asset{ type : "json"};

import ("foo.json", {with: {type :"json"}});
```

pipeline operator:

proposal currently in stage 2, introduces a new operator |> that allows developers to chain multiple function calls together in a more readable and concise way. Together with the pipe

operator, the placeholder operator % is being introduces which will hold the previous function's value IT should enhance the readability and maintainability of the code, especially when performing a series of operations on a value.

Instead of nested function calls or long chains of dot notation the pipeline operator allows developer to write code in a more linear and intuitive manner. Here is a real-worlsd example from a React repository, which can be improved by using the pipeline operator:

```
console.log(
chalk.dim(
`$ ${Object.keys(envars)
.map((envar) => ` ${envar} = ${envars[envar]}`)
.join(' ')},
'node',
  args.join(' '),
  ),
)
);
```

As you can see, by adding nested function calls it becomes much harder to read and understand the code. By adding pipeline operator, this code can be updated and improved.

```
Object.keys(envars)
.map(envar => ` ${envar} = ${envars[envar]}`)
.join(' ')
|> `$ @{%}`
|> chal.dim(%, 'node, args.join(' '))
```

|> console.log(%)

Dacorators:

Although decorators are not yet available in JS, they are actively used with the help of such transpilers as TypeScript, Babel and webpack.

Currently, the decoraors proposal is in the Stage 3, which means it is getting closer to being a part of native JS. Decorators are the fucntions that are calles on other JS element, such as classes, class elements, methods or other fucntions by adding an additional fucntionality on those elements.

Custome decorators can be easily created and used. In the bottom, the decortors is just a fucntion, that accepts specific arguments:

Value - the element, on which the decorator is used

context - the context of the element, on which the decorator is used.

Here's the total type of decorator:

type Decorator = {

value : Input,

```
context : {
  kind : string,
  name: string | sysbol;
  access : {
    get?() : unknown;
set?(value : unknown) : void;
  };
  private?: boolean;
  static? : boolean;
  addInitializer(initializer: () => void): void;
  },
) => Output | void;
```

A useful example of using the dacorator would be for protecting the method with a validation rule. For that, we can create the following decorator.