**ES6 (ECMAScript 2015)**

ES6 (ECMAScript 2015) introduced several powerful features to JavaScript that made the language more expressive, concise, and easier to work with. Here are some of the most important ES6 features:

**1. Let and Const**

* let: Block-scoped variable declaration.
* const: Block-scoped constant declaration.

A math equation with numbers and symbols

AI-generated content may be incorrect.

**2. Arrow Functions**

Shorter syntax for writing functions.



**3.Template Literals**

Use backticks (`) for string interpolation and multi-line strings.

A close-up of a logo

AI-generated content may be incorrect.

**4.Destructuring Assignment**

Extract values from arrays or objects into variables. A close-up of a couple of words

AI-generated content may be incorrect.

**5.Default Parameters**

Set default values for function parameters.

A white background with black and white clouds

AI-generated content may be incorrect.

**6. Rest and Spread Operators**

* **Rest**: Collects arguments into an array.
* **Spread**: Expands elements of an array or object.

A white background with numbers and symbols

AI-generated content may be incorrect.

**7.Promises**

Handle asynchronous operations more cleanly.

A white background with colorful text

AI-generated content may be incorrect.

**8.Modules**

Use import and export to split code into reusable files.

A math equations on a white background

AI-generated content may be incorrect.

**forEach()**

* Purpose: Executes a function for each element in an array.
* Return Value: undefined — it does not return a new array.
* Use Case: When you want to perform side effects (e.g., logging, updating UI, etc.).

**map()**

* **Purpose**: Transforms each element in an array and returns a **new array**.
* **Return Value**: A **new array** with the transformed values.
* **Use Case**: When you want to create a new array based on the original one.

**for...in**

* **Used for**: Iterating over the **enumerable properties (keys)** of an object.
* **Returns**: The **keys** (property names) of the object.
* **Use Case**: Best for objects, not arrays.

**for...of**

* **Used for**: Iterating over **iterable objects** like arrays, strings, maps, sets, etc.
* **Returns**: The **values** of the iterable.
* **Use Case**: Best for arrays and other iterable collections.

A group of colorful text

AI-generated content may be incorrect.

**Difference between foreach and map?**

**forEach()**

* Purpose: Executes a function for each element in an array.
* Return Value: undefined — it does not return a new array.
* Use Case: When you want to perform side effects (e.g., logging, updating UI, etc.).

const numbers = [1, 2, 3];

numbers.forEach(num => {

  console.log(num \* 2); // Just logs, doesn't return anything

});

**map()**

* Purpose: Transforms each element in an array and returns a new array.
* Return Value: A new array with the transformed values.
* Use Case: When you want to create a new array based on the original one.

**for...in**

* Used for: Iterating over the enumerable properties (keys) of an object.
* Returns: The keys (property names) of the object.
* Use Case: Best for objects, not arrays.

const person = { name: "Alice", age: 25 };

for (let key in person) {

  console.log(key);        // name, age

  console.log(person[key]); // Alice, 25

}

**for...of**

* Used for: Iterating over iterable objects like arrays, strings, maps, sets, etc.
* Returns: The values of the iterable.
* Use Case: Best for arrays and other iterable collections.

const numbers = [10, 20, 30];

for (let num of numbers) {

  console.log(num); // 10, 20, 30

}

**Key Differences**

| **Feature** | **for...in** | **for...of** |
| --- | --- | --- |
| Iterates over | Keys (property names) | Values of iterable objects |
| Works on | Objects (and arrays, but not ideal) | Arrays, strings, maps, sets, etc. |
| Output | Property names (strings) | Values (elements) |
| Use with array | Not recommended (may include keys from prototype) | Recommended |

**Example with Array**

const arr = ['a', 'b', 'c'];

for (let i in arr) {

  console.log(i);      // 0, 1, 2 (indexes)

}

for (let val of arr) {

  console.log(val);    // a, b, c (values)

}

**Const, var and let (keywords)**

In JavaScript, var, let, and const are used to declare variables, but they differ in scope, reassignability, and re-declaration. var is function-scoped and allows re-declaration and reassignment, while let is block-scoped and allows reassignment but not re-declaration. const is also block-scoped and prevents both re-declaration and reassignment.

**Scope**:

* var: Function-scoped (globally if declared outside of any function).
* let: Block-scoped (within the code block where it's defined).
* const: Block-scoped (within the code block where it's defined).

**Reassignability:**

* var: Can be reassigned.
* let: Can be reassigned.
* const: Cannot be reassigned after initialization.

**Re-declaration:**

* var: Can be re-declared within the same scope.
* let: Cannot be re-declared within the same scope.
* const: Cannot be re-declared within the same scope.

**Hoisting:**

* var: Hoisted (can be used before declaration).
* let and const: Hoisted, but accessing them before declaration will result in an error ("Uncaught ReferenceError: Cannot access 'variable' before initialization").

**What is Hoisting?**

Hoisting in JavaScript is a behavior where variable and function declarations are conceptually moved to the top of their containing scope during the compilation phase, before the code is executed. This means that you can use variables and call functions before their actual declaration appears in the code

**Declarations are hoisted, not initializations:**

Only the declaration part of a variable or function is hoisted. If a variable is initialized at the same time it's declared, only the declaration moves to the top, while the assignment of the value remains in place.

**Variable Hoisting (var):**

Variables declared with var are hoisted and initialized with undefined. This allows you to access a var variable before its declaration, but its value will be undefined until the line of its initialization is reached.

**let and const Hoisting (Temporal Dead Zone):**

Variables declared with let and const are also hoisted, but they are not initialized. They enter a "Temporal Dead Zone" (TDZ) from the beginning of their scope until their declaration line is executed. Attempting to access them within the TDZ will result in a ReferenceError.

**Function Hoisting:**

Function declarations are fully hoisted, meaning both the function's name and its definition are moved to the top of their scope. This allows you to call a function before its declaration in the code.

**Function Expressions vs. Function Declarations:**

Hoisting applies to function declarations, not function expressions (where a function is assigned to a variable). Function expressions behave like variable hoisting, meaning the variable holding the function will be hoisted, but the function itself won't be accessible until the assignment line.

**What is currying**

Currying in JavaScript is a technique of transforming a function that takes multiple arguments into a sequence of functions, each taking a single argument. It essentially breaks down a function with multiple parameters into a series of functions that each accept one parameter and return another function until all parameters are received.

A screenshot of a computer program

AI-generated content may be incorrect.

**Benefits of Currying:**

* Code Reusability and Modularity:

Creates specialized functions from general ones, promoting reuse and making code more modular.

* Easier Function Composition:

Facilitates combining smaller, curried functions to build more complex operations.

* Improved Readability:

Can lead to more expressive and understandable code, especially when dealing with functions with many arguments.

* Delayed Execution:

Allows for the execution of a function to be delayed until all necessary arguments are available.

**What Is Closure in JavaScript?**

A closure in JavaScript is created when a function is defined within another function. It allows the inner function to access the variables and parameters of the outer function, even after the outer function has finished executing. This happens because the inner function maintains a reference to its lexical environment, capturing the state of the outer function at the time of its creation. In simpler terms, a closure allows a function to access variables from its outer scope, even after that scope has closed.

A screen shot of a computer code

AI-generated content may be incorrect.

**Key takeaways about closures:**

* **Lexical scoping:**

Closures rely on lexical scoping, where the scope of a variable is determined by its position in the source code.

* **Persistence:**

Closures preserve the values of variables from their surrounding scope even after the outer function has finished executing.

* **Use cases:**

Closures are fundamental in functional programming, enabling features like data encapsulation, state management, and creating reusable functions with persistent state.