

# ES6 JavaScript

ECMAScript is known as “ES5, ES6, ES7, ES8” are the versions of the JavaScript. ECMAScript is the standard and JavaScript is the most popular *implementation* of that standard.

There are total 8 versions of the JavaScript

There are a total of 8 versions of JavaScript:

- **ES1:** June 1997
- **ES2:** June 1998
- **ES3:** December 1999
- **ES4:** Abandoned
- **ES5:** Released in 2009
- **ES6:** Released in 2015
- **ES7:** Released in 2016
- **ES8:** Released in 2017

Most of the major updates and new features were introduced in **ES6**, making it a significant release.

ES5 was released in 2009, and the ES6 was released almost six years later which was in 2015 and after that ES7 after the following year and ES8 next following year.


So, most of the updates and functions are introduced in the ES6.

1. “let” and “const” keywords.


Let and const are most useful keywords. How you might think?

=> Well the let helps to declare block-scoped variables, before this we had var which had no scope and some time using var we have to update the var in some place but var updates all the var in the file because this doesn't have scope.

To use the let just copy =>

```
let variable;  
let age = 25;  
age = 27; //  This is allowed
```

Along with let, const variable were added which is used in the place where we don't want to make any update for example API we mostly make API const.

```
const API = "https://api.example.com";  
API = "https://newapi.com"; //  This will cause an error
```

if we try to update or try to make changes in const variable that will give you an error.

## 2. Arrow Functions


Arrow function in ES6 is the shorter way to write functions in JavaScript. This helps in cleaner syntax.

```
const greet = (name) => `Hello, ${name}`;  
console.log(greet("Sukhdeep")); // Hello, Sukhdeep
```

Arrow functions do not have their own this. They use this from the parent scope. This is a good thing because in regular functions, this changes depending on *how* the function is called — which often confuses developers.

Regular functions created using function declaration or expression are “constructible” and “callable”  
How “this.” keyword works ?

```
function Car(name) {  
  this.brand = name;  
}
```

```
let carData = new Car("Maruti");  
console.log(carData); //  Works fine
```

This works but if we try to use this in an arrow function this will give us an error.

```
const Car = (name) => {  
  this.brand = name;  
};
```

let carData = new Car("Maruti"); // ❌ Error: Car is not a constructor

In **arrow functions**, this is **lexically inherited** — it comes from the **parent scope**.

```
const person = {  
  name: "Sukhdeep",  
  regularFunc: function () {  
    console.log("Regular Function:", this.name); // ✅ 'Sukhdeep'  
  },  
  arrowFunc: () => {  
    console.log("Arrow Function:", this.name); // ❌ 'undefined'  
  },  
};
```

```
person.regularFunc();  
person.arrowFunc();
```



Explanation:

- `regularFunc()` works because `this` refers to `person`.
- `arrowFunc()` does **not work** — arrow functions don't get their own `this`. Instead, `this` refers to the **outer/global object**, where `name` is undefined.

! When NOT to use arrow functions:

- As methods inside objects (if you need `this`).
- When using `this` in event handlers where `this` should refer to the DOM element.

```
const button = document.querySelector("button");
```

```
// ❌ Wrong: this won't refer to button  
button.addEventListener("click", () => {  
  console.log(this); // window or undefined  
});
```

```
// ✅ Correct  
button.addEventListener("click", function () {  
  console.log(this); // button element  
});
```

### 3. Template Literals

Use **backticks** ( ``` ) instead of quotes to insert variables directly in strings.

```
const name = "Sukhdeep";  
console.log(`Welcome, ${name}!`); // Welcome, Kamal!
```

### 4. Default Parameters

Set a **default value** for function arguments if they're not passed.

```
function sayHello(name = "Guest") {  
  console.log(`Hello, ${name}`);  
}
```

```
sayHello(); // Hello, Guest
```

### 5. Destructuring

Unpack values from arrays or objects easily.

```
// Array  
const colors = ["red", "green"];  
const [first, second] = colors;  
console.log(first); // red
```

```
// Object  
const user = { name: "Raj", age: 22 };  
const { name, age } = user;
```

## 6. Spread & Rest ( . . . )

### Spread

Expands arrays or objects.

```
const a = [1, 2];  
const b = [...a, 3]; // [1, 2, 3]  
  
const obj1 = { x: 1 };  
const obj2 = { ...obj1, y: 2 }; // { x: 1, y: 2 }
```

### Rest

Gathers the remaining items into an array.

```
function sum(...nums) {  
  return nums.reduce((total, n) => total + n, 0);  
}  
  
console.log(sum(1, 2, 3)); // 6
```

## Promises in JavaScript

# JavaScript Promises

Pending



# Promises in JavaScript

## What the hell is a Promise — and do we really need it?

*“She/he promised me! ‘I will never leave you.’”*

**STOP crying!** We’re not talking about that kind of promise, you love-blind human.

We’re talking about **JavaScript Promises** — and unlike your ex, they actually deliver what they say and don’t ghost you (unless rejected 😊).

- So, JavaScript (js) is Single-Threaded by nature often poses challenges, especially when your dealing with **asynchronous operations** (Remember that this is going to be so important like your ex). Promises, a solution to these challenges.
- 🌐 JavaScript is Single-Threaded

JavaScript runs on a **single thread**, which means it can do only **one task at a time**. This becomes a challenge when you're dealing with tasks that take time — like fetching data from a server.

That's where **Promises** come in — to handle **asynchronous operations** without freezing the rest of your code.

## What is **asynchronous** and **synchronous** ?

- **Asynchronous**

In synchronous programming, operations are performed one after the other, in sequence. So, basically each line of code waits for the previous one to finish before proceeding to the next. This means that the program executes in a predictable, linear order, with each task being completed before the next one starts.

- **Synchronous**

Asynchronous programming, on the other hand, allows multiple tasks to run independently of each other. In asynchronous code, a task can be initiated, and while waiting for it to complete, other tasks can proceed. This non-blocking nature helps improve performance and responsiveness, especially in web applications.

## How to use Promise in JS?

A **Promise** in JavaScript is an object that represents the eventual **completion (or failure)** of an **asynchronous operation** and its resulting value.

## Syntax

```
let promise = new Promise(function(resolve, reject) {  
  // async operation  
});
```

- `resolve(value)` — if the operation is successful.
  - `reject(error)` — if the operation fails.
- 

## States of a Promise

A Promise has **three states**:

1. `pending` – initial state.
  2. `fulfilled` – operation completed successfully.
  3. `rejected` – operation failed.
- 

## Example

```
let promise = new Promise(function(resolve, reject) {  
  let success = true;  
  if (success) {  
    resolve("Operation successful!");  
  } else {  
    reject("Operation failed!");  
  }  
});  
  
promise.then(function(result) {  
  console.log(result); // if resolved  
}).catch(function(error) {  
  console.error(error); // if rejected  
});
```

---

## Chaining Promises

```
fetch('https://api.example.com/data')  
  .then(response => response.json())  
  .then(data => {  
    console.log(data); // Handle data  
  })  
  .catch(error => {  
    console.error('Error:', error);  
  });
```

## Why Use Promises?

## ✓ 1. To Avoid Callback Hell

**Callback Hell** happens when you nest many asynchronous functions, leading to messy and hard-to-maintain code:

```
// Without Promises
doTask1(function(result1) {
  doTask2(result1, function(result2) {
    doTask3(result2, function(result3) {
      // and so on...
    });
  });
});
```

**With Promises:**

```
doTask1()
  .then(result1 => doTask2(result1))
  .then(result2 => doTask3(result2))
  .then(result3 => console.log(result3))
  .catch(error => console.error(error));
```

---

## ✓ 2. Cleaner & More Readable Asynchronous Code

Promises allow you to write **linear-looking** code even for asynchronous operations, making it **easier to follow and debug**.

```
fetchData()
  .then(processData)
  .then(displayResult)
  .catch(handleError);
```

---

## ✓ 3. Used Heavily in Modern APIs

Modern JavaScript APIs and libraries **return Promises** instead of using callbacks:

- `fetch()` – to get data from APIs
- `navigator.geolocation.getCurrentPosition` (wrapped in Promise)
- `async/await` – built on top of Promises

Example with `fetch`:

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(json => console.log(json))
  .catch(err => console.error(err));
```



# How you can use promise function to call API?

```
function fetchUserDataWithDelay() {
  return new Promise((resolve, reject) => {
    console.log('Fetching user data...');

    setTimeout(() => {
      fetch('https://jsonplaceholder.typicode.com/users')
        .then(response => {
          if (!response.ok) {
            throw new Error('Network response was not ok');
          }
          return response.json();
        })
        .then(data => {
          resolve(data); // Pass data to the next then()
        })
        .catch(error => {
          reject('Fetch failed: ' + error.message);
        });
    }, 2000);
  });
}

// Call the function
fetchUserDataWithDelay()
  .then(users => {
    console.log('User data received after 2 seconds:');
    console.log(users);
  })
  .catch(error => {
    console.error('Error:', error);
  });
```