

JavaScript - from BigBang

🕒 Created	@December 16, 2024 2:14 PM
🏷️ Tags	

`let`, `const`, and `var`

Keyword	Scope	Can Update	Can Re-declare	Hoisted	Must Assign Value
var	Function	Yes	Yes	Yes	No
let	Block	Yes	No	No	No
const	Block	No	No	No	Yes

```
function demo() {  
  if (true) {  
    var x = 1; // function-scoped  
    let y = 2; // block-scoped  
    const z = 3; // block-scoped and constant  
    y = 4; // allowed  
    // z = 5; // Error: can't reassign const  
  }  
  console.log(x); // 1 (accessible)  
  // console.log(y); // Error: y is not defined  
  // console.log(z); // Error: z is not defined  
}  
demo();
```

Values:

- **Primitive types** are basic, simple values (not objects) and are stored by value. They include:
 - **String**: Text data, like words or sentences.
 - **Number**: Any number, integer or decimal.
 - **Boolean**: Only true or false.

- **Null**: An intentional empty value.
- **Undefined**: A variable declared but not assigned any value.
- **Symbol**: A unique, unchangeable value, often used as object property keys¹²⁴⁸.
- **Non-Primitive types** are more complex, can hold multiple values, and are stored by reference. They include:
 - **Object**: A collection of key-value pairs.
 - **Array**: An ordered list of values (a type of object).
 - **Function**: A block of code that can be called and executed (also a type of object)
- **Arithmetic Operators**: `+`, `-`, `*`, `/`, `%`.
- **Assignment Operators**: `=`, `+=`, `=`, etc.
- **Comparison Operators**: `==`, `===`, `!=`, `!==`, `>`, `<`, `>=`, `<=`.
- **Logical Operators**: `&&`, `||`, `!`.

◆ Control Structures

- **Conditional Statements:**

- `if`, `else if`, `else`.
- `switch`

Comparison operators imp to know:

- `==`: Checks if values are equal, allowing type conversion (loose equality).
- `===`: Checks if values and types are both equal (strict equality).
- `!=`: Checks if values are not equal, allowing type conversion.
- `!==`: Checks if values or types are not equal (strict inequality).

```
5 == "5" // true (values are equal, type is converted)
5 === "5" // false (values are equal, but types are different)
```

```
5 !== "6" // true (values are not equal)
5 !== "5" // true (values are equal, but types are different)
```

loops:

```
// for loop
for (let i = 0; i < 3; i++) {
  console.log(i); // 0, 1, 2
}

// while loop
let j = 0;
while (j < 3) {
  console.log(j); // 0, 1, 2
  j++;
}

// do...while loop
let k = 0;
do {
  console.log(k); // 0, 1, 2
  k++;
} while (k < 3);

// for...of loop (values)
const arr = ['a', 'b', 'c'];
for (let value of arr) {
  console.log(value); // 'a', 'b', 'c'
}

// for...in loop (keys)
const obj = {x: 1, y: 2};
for (let key in obj) {
  console.log(key); // 'x', 'y'
}
```

functions:

- **Function Declaration:** Defines a named function. Can be called before it appears in the code (hoisted).
- **Function Expression:** Assigns a function to a variable. Not hoisted, so must be defined before use.
- **Arrow Function:** A shorter way to write functions. Great for simple, one-line functions. Does not have its own this context.

```
// Function Declaration
function greet(name) {
  return `Hello, ${name}!`;
}
greet("Alice"); // "Hello, Alice!"

// Function Expression
const greet2 = function(name) {
  return `Hello, ${name}!`;
};
greet2("Bob"); // "Hello, Bob!"

// Arrow Function
const greet3 = (name) => `Hello, ${name}!`;
greet3("Charlie"); // "Hello, Charlie!"
```

DOM - **Document Object Model:**

What is the DOM?

DOM stands for **Document Object Model**.

Think of a web page as a **tree** where every element (like a button, heading, image, etc.) is a **branch or leaf**. JavaScript can access and change these elements using the DOM.

Example:

If this is your HTML:

```
<h1 id="title">Hello, Adi!</h1>
<button id="btn">Click Me</button>
```

You can use JavaScript to **grab**, **change**, or **respond to** these elements.

How to Manipulate the DOM

There are **3 steps** you'll often follow:

✓ 1. Select an Element

Use JavaScript to "grab" an element so you can do something with it.

```
const title = document.getElementById('title'); // Grabs by ID
const button = document.querySelector('#btn'); // Grabs first match
(CSS style selector)
```

Other selectors:

- `document.getElementsByClassName('some-class')`
- `document.getElementsByTagName('p')`
- `document.querySelectorAll('.some-class')`

✓ 2. Change the Content, Style, or Attributes

Once you've selected an element, you can:

Change Text or HTML

```
title.textContent = 'Welcome, Adi!'; // Changes the visible text
title.innerHTML = '<i>Hi, Adi!</i>'; // Can insert HTML tags
```

Change Styles

```
title.style.color = 'blue';
title.style.fontSize = '30px';
```

Change Attributes

```
button.setAttribute('disabled', 'true'); // Disable the button
button.removeAttribute('disabled');      // Enable the button again
```

3. Add Interactivity with Event Listeners

You can make things happen when the user interacts with the page:

```
button.addEventListener('click', function() {
  title.textContent = 'You clicked the button!';
  title.style.color = 'green';
});
```

Here's what's happening:

- When the button is **clicked**,
- It **changes the text** of the title,
- And **changes the color**.

Example: Full HTML + JS

```
<!DOCTYPE html>
<html>
  <head>
    <title>DOM Demo</title>
  </head>
  <body>
    <h1 id="title">Hello!</h1>
    <button id="btn">Click Me</button>

    <script>
      const title = document.getElementById('title');
      const button = document.getElementById('btn');

      button.addEventListener('click', function() {
        title.textContent = 'Button was clicked!';
      });
    </script>
  </body>
</html>
```

```
title.style.color = 'purple';
});
</script>
</body>
</html>
```

💡 Common Things You Can Do with DOM:

Task	Example Code
Change text	<code>element.textContent = 'New text'</code>
Hide an element	<code>element.style.display = 'none'</code>
Show an element	<code>element.style.display = 'block'</code>
Change image source	<code>img.src = 'new-image.jpg'</code>
Add a class	<code>element.classList.add('myClass')</code>
Remove a class	<code>element.classList.remove('myClass')</code>
Toggle a class	<code>element.classList.toggle('active')</code>

Arrays and Objects:

Arrays and Objects

- **Arrays:** Ordered lists of items. Each item has a number (index).
- **Objects:** Collections of key-value pairs. Keys are strings, values can be anything.

```
javascriptconst fruits = ['apple', 'banana', 'cherry'];
```

```
javascriptconst person = { name: 'Adi', age: 25 };
```

Array Methods

- **push():** Adds an item to the end.
- **pop():** Removes the last item.
- **shift():** Removes the first item.
- **unshift():** Adds an item to the start.

- **map()**: Creates a new array by applying a function to every item.
- **filter()**: Creates a new array with only items that pass a test.
- **reduce()**: Reduces the array to a single value (like sum).
- **forEach()**: Runs a function for every item (does not return a new array).

Example:

```
const nums = [1, 2, 3];
nums.push(4);
// [1, 2, 3, 4]
nums.pop();
// [1, 2, 3]
nums.shift();
// [2, 3]
nums.unshift(0);
// [0, 2, 3]

const doubled = nums.map(n => n * 2);
// [0, 4, 6]
const even = nums.filter(n => n % 2 === 0);
// [0, 2]
const sum = nums.reduce((acc, n) => acc + n, 0);
// 5
nums.forEach(n => console.log(n));
// logs each number
```

Object Methods

- **Object.keys(obj)**: Returns an array of the object's keys.
- **Object.values(obj)**: Returns an array of the object's values.
- **Object.entries(obj)**: Returns an array of [key, value] pairs.

Example:

```
const person = { name: 'Adi', age: 25 };
Object.keys(person);
// ['name', 'age']
Object.values(person);
// ['Adi', 25]
Object.entries(person);
// [['name', 'Adi'], ['age', 25]]
```

Destructuring

- **Arrays**: Pulls out values by position.

```
const [a, b] = [1, 2]; // a = 1, b = 2
```


- **Objects:** Pulls out values by key.

```
const {name, age} = person; // name = 'Adi', age = 25
```

Is using `const` necessary?

- **No, not always.**
 - Use `const` if you don't plan to reassign the variable (best practice for arrays and objects you won't reassign).
 - Use `let` if you plan to change the variable's value.
 - Avoid `var` in modern code.

Example:

```
const arr = [1, 2, 3]; // Can't reassign arr, but can change contents: arr.push(4);  
let count = 0;  
// Can reassign: count = 1;
```

Tip: Prefer `const` for arrays/objects unless you need to reassign the whole variable.

Asynchronous JavaScript

First: What is "Asynchronous" in JavaScript?

Synchronous vs Asynchronous

- **Synchronous** = Code runs **line by line**, one thing at a time.
You have to **wait** for one line to finish before moving to the next.
 - **Asynchronous** = Code can **start**, then **pause**, and **continue later** when it's done.
JS won't **freeze** and wait — it moves on and comes back later.
-

How JS Does Asynchronous Tasks

JavaScript uses:

1. **Callbacks**
2. **Promises**

3. Async/Await

1. Callbacks (Old Way)

A **callback** is a function you pass into another function to be called **later**.

Example: Using `setTimeout` (wait 2 seconds)

```
console.log("Step 1: Start");

setTimeout(function() {
  console.log("Step 2: After 2 seconds");
}, 2000); // 2000ms = 2s

console.log("Step 3: Done");
```

Output:

```
Step 1: Start
Step 3: Done
Step 2: After 2 seconds
```

See how Step 2 *waited*? JS kept going — didn't pause.

2. Promises (Modern Way)

A **promise** is an object that **promises** to return a value in the future (either success  or failure .

Example: A Simple Promise

```
const promise = new Promise(function(resolve, reject) {
  let success = true;

  if (success) {
    resolve("🎉 Success!");
  } else {
    reject("❌ Failed!");
  }
});
```

```

    }
  });

  promise
    .then(function(result) {
      console.log(result); // If resolved
    })
    .catch(function(error) {
      console.log(error); // If rejected
    });

```

✓ Real Use Case: Fetch Data from API

```

fetch("https://jsonplaceholder.typicode.com/users")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.log(error));

```

- `fetch()` sends a request to the internet
- `.then()` waits for the result and parses it as JSON
- `.catch()` handles errors if something goes wrong

🏆 3. Async / Await (Most Modern and Readable Way)

This is just a **cleaner way** to write code that uses promises.

Same API Example using **async/await** :

```

async function getUsers() {
  try {
    const response = await fetch("https://jsonplaceholder.typicode.com/users");
    const users = await response.json();
    console.log(users);
  } catch (error) {

```

```

    console.log("Error:", error);
  }
}

getUsers();

```

What's happening?

- `async` means the function will use `await` inside it
- `await` pauses until the `fetch()` or `json()` is done
- It's like writing synchronous code, but it's actually async!

Summary Table

Feature	How it works	Example Use Case
Callback	Function passed to another function	<code>setTimeout()</code> , DOM events
Promise	<code>.then()</code> and <code>.catch()</code> methods	<code>fetch()</code> , file reading
Async/Await	Cleaner syntax using <code>async</code> and <code>await</code>	API calls, waiting for data

Practice Exercise (Try It!)

```

<!DOCTYPE html>
<html>
<head>
  <title>Async JS Demo</title>
</head>
<body>
  <button id="load">Load User Data</button>
  <div id="output"></div>

  <script>
    const btn = document.getElementById("load");
    const output = document.getElementById("output");
  </script>

```

```

btn.addEventListener("click", async () => {
  try {
    const res = await fetch("https://jsonplaceholder.typicode.com/users/1");
    const user = await res.json();
    output.textContent = `User: ${user.name}, Email: ${user.email}`;
  } catch (error) {
    output.textContent = "Failed to fetch user.";
  }
});
</script>
</body>
</html>

```

✅ When you click the button, it fetches user data and shows it on the page!

🏁 TL;DR

- **Async code** = code that runs in the background, doesn't block the page.
- **Callback** = function passed to run later.
- **Promise** = cleaner async control with `.then()` and `.catch()`.
- **Async/Await** = even cleaner syntax to write async code like it's synchronous.

Want me to give you mini challenges or quiz questions to test this? 😊

//my code

```

const heading = document.querySelector("h1");
const button = document.querySelector("button");

async function fetchData() {
  const response = await fetch("https://catfact.ninja/fact");
  const data = await response.json();
}

```

```
    heading.innerText = data.fact;
    button.innerText = "Get another fact";

}

button.addEventListener("click", fetchData);
```

Object-Oriented Programming (OOP) in JavaScript

◆ Classes and Objects

```
class Person {
  constructor(name) {
    this.name = name;
  }

  greet() {
    return `Hello, ${this.name}`;
  }
}


const adi = new Person('Adi');
console.log(adi.greet());
```

◆ Inheritance

```
class Student extends Person {
  constructor(name, course) {
    super(name);
    this.course = course;
  }

  study() {
    return `${this.name} is studying ${this.course}`;
  }
}
```

```
}  
}
```

 **Exercise:** Create a class `Car` with properties like `make`, `model`, and methods like `start()` and `stop()`.

ERRORSZZ:

Try...Catch

- **try...catch** lets you run code and handle errors if they happen, so your program doesn't crash.
- Code inside `try` runs first. If there's an error, it jumps to `catch`.

Example:

```
try {  
  let result = riskyFunction();  
  // might throw an error  
  console.log(result);  
} catch (error) {  
  console.error(error);  
  // handles the error  
}
```

Debugging Tools

- **Console Methods:**
 - `console.log()` : Prints info for debugging.
 - `console.error()` : Prints errors in red for better visibility.
 - `console.table()` : Displays data in a table format (great for arrays/objects).

Example:

```
console.log('Hello, world!');  
console.error('Something went wrong!');  
console.table([  
  {name: 'Adi', age: 25},  
  {name: 'Sam', age: 30}]);
```

- **Breakpoints:**
 - Use your browser's developer tools (usually F12 or right-click → Inspect).

- In the "Sources" tab, click the line number to set a breakpoint.
- When code runs, it will pause at the breakpoint so you can inspect variables and step through code line by line.

Working with Forms and Validation:

Forms are how users send data (like login info or contact forms). JS helps us control and validate what users type in before we send it to a server.

```
<form id="loginForm">
  <input type="text" id="username" placeholder="Username" />
  <input type="password" id="password" placeholder="Password" />
  <button type="submit">Login</button>
</form>
```

```
const form = document.getElementById("myForm");
const usernameInput = document.getElementById("username");

form.addEventListener("submit", function (event) {
  if (usernameInput.value.trim() === "") {
    alert("Username cannot be empty!");
    event.preventDefault(); // stop form from submitting
  } else {
    alert("Form submitted successfully!");
  }
});
```

Code Explanation:

- `getElementById("myForm")` : Gets the form element.
- `addEventListener("submit", function)` : Runs the function when the user clicks submit.
- `usernameInput.value.trim()` : Gets the input value, `.trim()` removes extra spaces.
- `event.preventDefault()` : Stops the default action (sending the form).

Fetch API and JSON:

The **Fetch API** is used to **get or send data** from/to a server or an API (like weather, GitHub, etc.)

- **JSON** stands for JavaScript Object Notation. It's how data is usually structured when sent or received from APIs. *// DAMN i didnt know this...*

GETTING DATA FROM API

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then(response => response.json()) // convert to JSON
  .then(data => {
    console.log(data); // see the data in the console
  })
  .catch(error => {
    console.error("Error fetching data:", error);
  });
```

Code Explanation:

- `fetch(...)` : Makes a network request to the URL.
- `.then(response => response.json())` : Converts the response into JSON.
- `.then(data => {...})` : Now we have the actual user data, and we can use it.
- `.catch(...)` : If something goes wrong (e.g., no internet), this runs.

SENDING

```
const userData = {
  name: "Adi",
  email: "adi@example.com",
};

fetch("https://jsonplaceholder.typicode.com/users", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
```

```

    },
    body: JSON.stringify(userData),
  })
  .then(response => response.json())
  .then(data => console.log("User created:", data))
  .catch(error => console.error("Error:", error));

```

Code Explanation:

- `method: "POST"` : We're sending (posting) data.
- `headers` : Tells the server the data is in JSON format.
- `body: JSON.stringify(userData)` : Convert JavaScript object into a JSON string.

Local Storage and Session Storage

Both are built-in browser features to **store small amounts of data**.

Type	Stored	Removed
<code>localStorage</code>	Until manually cleared	Persistent across sessions
<code>sessionStorage</code>	Only while browser tab is open	Cleared when tab is closed

LOCAL STORAGE

```

// Save to local storage
localStorage.setItem("username", "Adi");

// Get from local storage
const name = localStorage.getItem("username");
console.log(name); // Output: Adi

// Remove item
localStorage.removeItem("username");

```

SESSION STORAGE

```

// Save to session storage
sessionStorage.setItem("theme", "dark");

```

```
// Get from session storage
const theme = sessionStorage.getItem("theme");
console.log(theme); // Output: dark

// Remove item
sessionStorage.removeItem("theme");
```

JavaScript Modules (ES Modules)

Concept:

Modules let you **split your code into separate files**. This keeps your code organized and reusable.

You can `export` functions, variables, or classes from one file, and `import` them into another file.

This only works in modern browsers or tools like Webpack, Vite, etc.

✓ Example:

file: `math.js`

```
export function add(a, b) {
  return a + b;
}

export const name = "Adi";
```

file: `main.js`

```
import { add, name } from './math.js';
```

```
console.log(add(3, 4)); // 7
```

```
console.log(name); // Adi
```

In HTML:

```
<script type="module" src="main.js"></script>
```

Code Explanation:

- `export` : Makes code available for use in another file.
- `import` : Brings exported code into your file.
- `type="module"` : Tells the browser you're using modules (required in HTML).

This is all the basics that I thought would be useful. After completing all of this, you can refer to other JS frameworks like React, Vue, Angular, etc. Then, you can make some small projects to get started with JS and learn it properly.