

# Template Strings

---

Template strings, also known as template literals, are a feature in JavaScript that allows you to create strings with embedded expressions. They are denoted by backticks `` instead of single or double quotes. Template strings provide a more flexible and concise way to construct strings, especially when they involve variables or expressions.

```
let firstName = "HuXn";
let lastName = "WebDev";

function intro() {
  return "Hello my name is HuXn WebDev & i'm 19 years old";
}

console.log(
  `Hello my name is ${firstName} ${lastName} & i'm ${19 + 1000} years old.`
);
```

# Arrow functions

---

Arrow functions, also known as fat arrow functions, are a concise and shorter way to define functions in JavaScript. They were introduced in ECMAScript 6 (ES6) and provide a more compact syntax compared to traditional function expressions

```
//Normal functions
function greet(username) {
  return `Hello ${username}`
}

console.log(greet("HuXn"));

//Arrow Functions
greet = (username) => {
  return `Hello ${username}`
}
console.log(greet('HuXn'));

greet = (username) => `Hello ${username}`;
console.log(greet("HuXn"));

const double = (number) => number * 2;
console.log(double(5));
```

# Enhanced Object Literals in JavaScript (ES6)

---

Enhanced object literals, introduced in **ECMAScript 6 (ES6)**, are a set of enhancements to the syntax for defining objects in JavaScript. These enhancements make it more convenient and concise to define object properties and methods

## ✅ What's Improved:

- Property shorthand: name instead of name: name
- Method shorthand: intro() instead of intro: function()

---

## ◆ Traditional Way (Pre-ES6)

```
function user(name, age, work) {  
  return {  
    name: name,  
    age: age,  
    work: work,  
    intro: function () {  
      console.log(`My name is ${name}, I'm ${age} years old, & I'm a  
${work}`);  
    },  
  };  
}  
  
const huxn = user("HuXn", 17, "Programmer");  
huxn.intro();
```

## ◆ Enhanced Object Literals (ES6+)

```
function user(name, age, work) {  
  return {  
    name,  
    age,  
    work,  
    intro() {  
      console.log(`My name is ${name}, I'm ${age} years old, & I'm a  
${work}`);  
    },  
  };  
}  
  
const huxn = user("HuXn", 17, "Programmer");  
huxn.intro();
```

## Default Function Parameters in JavaScript (ES6)

---

Default parameters allow you to assign default values to function arguments. If no value is passed, the default is used.

---

### ◆ Example 1: Counting to 5 (Conditional)

```
function countTo5(count = false) {  
  if (count === true) {  
    for (let i = 1; i <= 5; i++) {  
      console.log(`Count: ${i}`);  
    }  
  }  
}  
  
countTo5(true); // ✅ Prints 1 to 5  
countTo5();     // ❌ Skips loop (default is false)
```

### ◆ Example 2: Rating Check

```
function rating(rate = 0) {  
  if (rate === 5) {  
    console.log("High Rating :");  
  } else if (rate === 0) {  
    console.log("Low Rating :");  
  }  
}  
  
rating();      // Output: Low Rating :(  
rating(5);    // Output: High Rating :  
rating(3);    // Output: (no output, no matching condition)
```

## JavaScript Spread Operator (ES6)

---

The **spread operator (...)** allows you to *spread* elements of an iterable (**like arrays or strings**) or properties of objects into individual elements. It's widely used for copying, merging, and passing data concisely.

---

### ✅ Why use the spread operator?

- Clone arrays or objects
- Merge arrays/objects cleanly

- Expand arguments in function calls
- Increase readability and reduce boilerplate

## ◆ Spread in Function Calls

```
let mx = Math.max(2, 4, 6, 7, 8, 1, 5, 10);
let mn = Math.min(2, 4, 6, 7, 8, 1, 5, 10);
console.log(mx); // 10
console.log(mn); // 1

const nums = [56, 24, 12, 55, 11, 10];
console.log(Math.max(nums)); // ❌ NaN
console.log(Math.max(...nums)); // ✅ 56
```

## 📦 Spread with Function Arguments

```
function giveMe4(a, b, c, d) {
  console.log("a", a);
  console.log("b", b);
  console.log("c", c);
  console.log("d", d);
}

giveMe4(1, 2, 3, 4);

const colors = ["red", "orange", "yellow", "green"];
giveMe4(colors); // ❌ 'a' gets entire array
giveMe4(...colors); // ✅ Each item passed as a separate argument

let string = "HuXn WebDev";
giveMe4(...string); // ✅ H, u, X, n (first 4 letters)
```

## ◆ Spread in Arrays

```
const strNums = ["one", "two", "three"];
const moreStrNums = ["four", "five", "six"];
const concat = [...strNums, ...moreStrNums];

console.log(concat);
// ["one", "two", "three", "four", "five", "six"]

let peoples = ["huxn", "john", "alex"];
console.log("kumar", ...peoples, "john doe");
// "kumar huxn john alex john doe"
```

```
let friends = ["jordan", "frad", "brad", ...peoples];
console.log(friends);
// ["jordan", "frad", "brad", "huxn", "john", "alex"]
```

## 📦 Spread in Objects

```
const obj1 = { x: 1, y: 2 };
const obj2 = { z: 3 };
const obj3 = { ...obj1, ...obj2 };

console.log(obj3); // {x: 1, y: 2, z: 3}

let person = {
  name: "HuXn",
  age: 17,
  gender: "Male",
};

const clone = { ...person, work: "Programming", location: "idk" };
console.log(clone);
// {
//   name: "HuXn",
//   age: 17,
//   gender: "Male",
//   work: "Programming",
//   location: "idk"
// }
```

## 🌟 JavaScript Rest Parameters (ES6)

The **rest parameter (...)** allows a function to accept an **indefinite number of arguments** as an array. It's useful for handling **variadic functions** — functions that take varying numbers of arguments.

### ✅ Why use Rest Parameters?

- To gather multiple arguments into one array
- Makes your functions more flexible and reusable
- Especially useful in utility/helper functions

### 💠 Example 1: Basic Rest Parameter Usage

```
function user(...userData) {
  console.log(userData);
}
```

```
user("HuXn", 17, "Male", "Programming");  
// Output: ["HuXn", 17, "Male", "Programming"]
```

### ◆ Example 2: Using Rest in Arrow Functions

```
const double = (...numbers) => numbers.map((num) => num * 2);  
  
console.log(double(1, 2, 3, 4, 5));  
// Output: [2, 4, 6, 8, 10]
```

### ▼ Example 3: Combining Named Params + Rest

```
function person(firstName, lastName, ...hobbies) {  
  console.log("First Name: ", firstName);  
  console.log("Last Name: ", lastName);  
  console.log("Hobbies: ", hobbies);  
}  
  
person("HuXn", "WebDev", "programming", "football");  
// Output:  
// First Name:  HuXn  
// Last Name:   WebDev  
// Hobbies:     ["programming", "football"]
```



## JavaScript Destructuring (ES6)

Destructuring allows you to "unpack" values from arrays or objects into distinct variables — making code cleaner and more readable.

### ✅ Why use Destructuring?

- Cleanly extract values
- Assign defaults
- Swap values without temp vars
- Handle returned arrays easily



### Array Destructuring

⚠ In array destructuring name doesn't matter but the order should match

#### ◆ Basic Assignment

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

console.log(red);    // "one"
console.log(yellow); // "two"
console.log(green);  // "three"
```

---

### ◆ Fewer Elements in Array

```
const foo = ["one", "two"];
const [red, yellow, green, blue] = foo;

console.log(red);    // "one"
console.log(yellow); // "two"
console.log(green);  // undefined
console.log(blue);   // undefined
```

---

### 📦 Default Values

```
let a, b;
[a = 5, b = 7] = [1];

console.log(a); // 1
console.log(b); // 7
```

---

### ↻ Swapping Variables

```
let a = 1;
let b = 3;

[a, b] = [b, a];
console.log(a); // 3
console.log(b); // 1

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

---

### ↻ From Function Returns

```
function f() {  
  return [1, 2];  
}  
  
let a, b;  
[a, b] = f();  
console.log(a); // 1  
console.log(b); // 2
```

## ✗ Ignoring Values

```
function f() {  
  return [1, 2, 3];  
}  
  
const [a, , b] = f();  
console.log(a); // 1  
console.log(b); // 3  
  
const [c] = f();  
console.log(c); // 1
```

## 📦 Rest with Destructuring

```
const [a, ...b] = [1, 2, 3];  
console.log(a); // 1  
console.log(b); // [2, 3]
```

## 📦 Object Destructuring (ES6)

Object destructuring allows you to extract properties from objects into variables.

⚠ In object destructuring, order doesn't matter — but variable names must match the property names exactly.

### ✅ Why use object destructuring?

- Cleaner syntax
- Avoids repetitive `object.property` code
- Makes extracting data from objects quick and readable

## 💠 Example: Extracting Properties



```
const student = {
  name: "HuXn",
  position: "First",
  rollno: "27"
};

const { name, position, rollno } = student;

console.log(name);    // "HuXn"
console.log(position); // "First"
console.log(rollno);  // "27"
```

## ◆ Assigning New Variable Names

```
const num = { x: 100, y: 200 };
const { x: new1, y: new2 } = num;

console.log(new1); // 100
console.log(new2); // 200
```

## ▼ Assignment Without Declaration

```
let name, division;
({ name, division } = { name: "HuXn", division: "First" });

console.log(name);    // "HuXn"
console.log(division); // "First"
```

Note: The parentheses ( ) are required here because the {} on the left would be treated as a block otherwise.



## Using Rest with Object Destructuring

```
let { a, b, ...args } = { a: 100, b: 200, c: 300, d: 400, e: 500 };

console.log(a);    // 100
console.log(b);    // 200
console.log(args); // { c: 300, d: 400, e: 500 }
```



## JavaScript Object Destructuring in Function Parameters

---

You can directly destructure objects inside function parameters to access and use specific properties. This is super clean and avoids extra lines of code.

---

### ◆ Example 1: Basic Destructuring in Function Parameter

```
const person = {
  name: "John Doe",
  age: 30,
  country: "USA",
};

function printPersonInfo({ name, age, country }) {
  console.log(`Name: ${name}`);
  console.log(`Age: ${age}`);
  console.log(`Country: ${country}`);
}

printPersonInfo(person);
```

### ◆ Example 2: Renaming, Default Values, and Nested Destructuring

```
let options = {
  title: "My menu",
  items: ["Item1", "Item2"],
};

// width → w, height → h, items[0] → item1, items[1] → item2
function showMenu({
  title = "Untitled",
  width: w = 100,
  height: h = 200,
  items: [item1, item2],
}) {
  console.log(`${title} ${w} ${h}`); // My menu 100 200
  console.log(item1); // Item1
  console.log(item2); // Item2
}

showMenu(options);
```

### 📖 Why Rename Properties During Destructuring?

#### 1. Shorter & Cleaner Code

Instead of using `width` and `height` repeatedly, `w` and `h` are easier and cleaner.

## 2. Avoid Naming Conflicts

If `width` or `height` already exist in the scope, renaming avoids clashing.

## 3. Assign Default Values Easily

`width: w = 100` gives `w` a default value if `width` is missing from the object.

---

### ✅ Pro Tips:

- Works great with React props, config objects, and APIs.
- You can combine destructuring with default values, renaming, and even nested destructuring.

# 🎵 JavaScript Destructuring — Array of Objects

---

You can destructure directly from an array of objects to extract nested values.

In this case, we extract the **singer** of the **third song**.

---

## 💠 Example: Extracting Property from a Specific Object in an Array

```
const songs = [
  { name: "Lucky You", singer: "Joyner", duration: 4.34 },
  { name: "Just Like You", singer: "NF", duration: 3.23 },
  { name: "Humble", singer: "Kendrick Lamar", duration: 2.33 },
  { name: "Old Town Road", singer: "Lil Nas X", duration: 1.43 },
  { name: "Cold Shoulder", singer: "Central Cee", duration: 5.23 },
];

// Extracting the 'singer' property from the 3rd song (index 2)
const [, , { singer: s }] = songs;
console.log(s); // Kendrick Lamar
```

---

### ✅ What happened?

- `[, , { singer: s }]` skips the first two elements, accesses the third, and renames `singer` to `s`.

This is great for extracting specific values from structured arrays like API responses or datasets!

# ? JavaScript Ternary Operator

---

The **ternary operator** in JavaScript is a short and clean way to write conditional statements.

It is the **only operator that takes three operands**.

### 📌 Syntax:

```
condition ? expressionIfTrue : expressionIfFalse;
```

## ◆ Example 1: Password Checker

```
let password = 2;

function passwordChecker(ps) {
  // Traditional if-else
  // if (ps === 8) {
  //   return "Strong Password";
  // } else {
  //   return "Password should be 8 characters";
  // }

  // Refactored using ternary
  return ps === 8 ? "Strong Password" : "Password should be 8 characters";
}

console.log(passwordChecker(password));
// Output: "Password should be 8 characters"
```

## ◆ Example 2: Age Check

```
const age = 25;

// Using ternary to check adulthood
const isAdult = age >= 18 ? "Adult" : "Not an Adult";

console.log(isAdult); // Output: "Adult"
```

### ✅ Why use the ternary operator?

- One-liner alternative to `if...else`
- Great for simple conditions
- Improves readability when used properly

🧠 **Tip:** Avoid using it for complex logic — it's best for short, clear conditions.

## 🔄 JavaScript `for...in` Loop

📘 Definition:

The **for...in** loop in JavaScript is used to **iterate over the enumerable properties of an object**. It loops through the keys (property names) of the object.

### Syntax:

```
for (let key in object) {  
  // code block to execute  
}
```

### Explanation:

- **key** holds each property name during the loop.
- **object** is the target being iterated over.
- Use it with **objects**, but be cautious when using it with arrays.

---

### Example 1: Iterating over an Object

```
let person = {  
  name: "HuXn",  
  age: 17,  
  gender: "Male",  
};  
  
for (let key in person) {  
  console.log(key, person[key]);  
}
```

### Output:

```
name HuXn  
age 17  
gender Male
```

---

### Example 2: Iterating over an Array using **for...in**

```
let list = ["one", "two", "three", "four"];  
  
for (let index in list) {  
  console.log(`${index}: ${list[index]}`);  
}
```

### Output:

```
0: one
1: two
2: three
3: four
```

⚠ **Note:** Although `for...in` can be used with arrays, it's better to use `for...of` or classic `for` loop for arrays to avoid unexpected behavior, especially if the array prototype is extended.

---

## JavaScript `for...of` Loop

### Definition:

The `for...of` loop in JavaScript is a **modern iteration statement** introduced in **ES6**. It provides a concise and readable way to **iterate over values** of iterable objects like:

- Arrays
- Strings
- Maps
- Sets
- NodeLists, etc.

Unlike `for...in`, which iterates over **keys/indexes**, `for...of` directly gives you the **element values**.

---

### Syntax:

```
for (variable of iterable) {
  // code to execute
}
```

- **variable**: A placeholder for the current value in each iteration.
  - **iterable**: Any iterable object (array, string, set, etc.)
- 

### Example 1: Iterating over an Array

```
let peoples = ["huxn", "alex", "john", "brad"];

for (let people of peoples) {
  console.log(people);
}
```

### Output:

```
huxn  
alex  
john  
brad
```

---

### Example 2: Iterating over a String





```
const text = "Hello";  
  
for (const char of text) {  
  console.log(char);  
}
```

#### Output:

```
H  
e  
l  
l  
o
```

---

### When to Use `for...of`:

Use Case	Recommended?
Arrays	 Yes
Strings	 Yes
Maps, Sets, NodeLists	 Yes
Plain Objects	 No (use <code>for...in</code> instead)

---

## JavaScript `forEach()` Method

### Definition:

`forEach()` is an **array method** in JavaScript that executes a **callback function once for each element** in an array, in order.

- It does **not return a new array**.
- It is mainly used for **side effects** (like printing, modifying).

---

### Syntax:

```
array.forEach((element, index, array) => {  
  // logic to apply to each element  
});
```

- **element**: Current element being processed
  - **index** (*optional*): Index of the current element
  - **array** (*optional*): The full array **forEach** is being applied to
- 

### Example 1: Printing Each Element

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

#### Output:

```
teal  
blue  
red  
green
```

This is equivalent to:

```
for (var i = 0; i < colors.length; i++) {  
  console.log(colors[i]);  
}
```

### Example 2: Capitalizing Words In-Place

```
const words = ["hello", "bird", "table", "football", "pipe", "code"];  
  
const capWords = words.forEach((word, index, arr) => {  
  arr[index] = word[0].toUpperCase() + word.substring(1);  
});  
  
console.log(words);
```

#### Output:



```
["Hello", "Bird", "Table", "Football", "Pipe", "Code"]
```

**Note:** Even though `forEach()` doesn't return anything, it can be used to **mutate the original array**.

### ⚠️ `forEach()` vs `map()`

Feature	<code>forEach()</code>	<code>map()</code>
Returns new array?	❌ No	✅ Yes
Purpose	Side effects	Data transformation
Mutates original?	Can (if you want)	Usually no

## 🔄 JavaScript Array Methods – `map()`, `filter()`, and `reduce()`

### 💠 `map()` Method

#### Definition:

The `map()` method creates a **new array** populated with the results of calling a provided function on **every element** in the original array.

#### Syntax:

```
array.map((element, index, array) => {  
  // return something new  
});
```

#### 🔪 Example 1: Doubling Numbers

```
let numbers = [1, 2, 3, 4, 5];  
let double = numbers.map((num) => num * 2);  
console.log(double); // [2, 4, 6, 8, 10]
```

#### 🔪 Example 2: Extracting Names from Objects

```
let peoples = [  
  { firstName: "Macom", lastName: "Reynolds" },  
  { firstName: "Kaylee", lastName: "Frye" },  
  { firstName: "Jayne", lastName: "Cobb" },  
];  
  
const results = peoples.map((person) => {
```

```
    return [person.firstName, person.lastName];  
  });  
  console.log(results);
```

---

## ◆ `filter()` Method

### Definition:

The `filter()` method returns a new array **containing elements that pass a test condition** (i.e., for which the callback returns `true`).

### Syntax:

```
array.filter((element, index, array) => {  
  return condition;  
});
```

### ✍ Example 1: Filter Songs Based on Duration

```
const songs = [  
  { name: "Lucky You", duration: 4.34 },  
  { name: "Just Like You", duration: 3.23 },  
  { name: "The Search", duration: 2.33 },  
  { name: "Old Town Road", duration: 1.43 },  
  { name: "The Box", duration: 5.23 },  
];  
  
console.log(songs.filter((song) => song.duration > 3));
```

### ✍ Example 2: Filter High RAM Computers

```
const computers = [  
  { ram: 4, hdd: 100 },  
  { ram: 8, hdd: 200 },  
  { ram: 16, hdd: 300 },  
  { ram: 32, hdd: 400 },  
];  
  
console.log(computers.filter((com) => com.ram > 16));
```

---

## ◆ `reduce()` Method

### Definition:

The `reduce()` method reduces an array to a **single value** by applying a function to each element and

carrying forward the result.

### Syntax:

```
array.reduce((accumulator, currentValue, index, array) => {  
  return updatedAccumulator;  
}, initialValue);
```

#### Example 1: Summing an Array

```
const numbers = [1, 2, 3, 4, 5];  
const sum = numbers.reduce((prev, curr) => prev + curr, 0);  
console.log(sum); // 15
```

#### Example 2: Finding the Oldest Age

```
const people = [  
  { name: "HuXn WebDev", age: 18 },  
  { name: "Alex Mead", age: 29 },  
  { name: "Brain Griffin", age: 40 },  
];  
  
const oldestAge = people.reduce((prev, curr) => (curr.age > prev ?  
curr.age : prev), 0);  
console.log(oldestAge); // 40
```

#### Example 3: Word Frequency Counter

```
const words = ["apple", "banana", "orange", "banana", "apple", "orange",  
"apple", "grape"];  
  
const wordFrequency = words.reduce((map, word) => {  
  map[word] = (map[word] || 0) + 1;  
  return map;  
}, {});  
  
console.log(wordFrequency);  
// Output: { apple: 3, banana: 2, orange: 2, grape: 1 }
```

---

✅ These three methods make JavaScript super powerful for working with arrays — master them and you'll write cleaner, more efficient code.

---

## ✓ JavaScript Array Methods: `every()` and `some()`

### 📘 Definition:

- **`every()`**: Returns `true` if **all** elements in the array pass the test in the provided callback function. Returns `false` if **any one** element fails.
- **`some()`**: Returns `true` if **at least one** element passes the test. Returns `false` only if **none** of the elements pass.

### 🔧 Syntax:

```
array.every((element, index, array) => condition);  
array.some((element, index, array) => condition);
```

### ✍️ Example 1: Checking String Lengths

```
const peoples = ["huxn", "jordan", "alex"];  
  
const res = peoples.every((person) => person.length === 4); // false  
const res2 = peoples.some((person) => person.length < 3);    // false  
  
console.log(res); // false  
console.log(res2); // false
```

### ✍️ Example 2: Check Song Durations

```
const songs = [  
  { name: "Lucky You", duration: 4.34 },  
  { name: "Just Like You", duration: 3.23 },  
  { name: "The Search", duration: 2.33 },  
  { name: "Old Town Road", duration: 1.43 },  
  { name: "The Box", duration: 5.23 },  
];  
  
const everyRes = songs.every((song) => song.duration > 3); // false  
const someRes = songs.some((song) => song.duration > 3);    // true  
  
console.log(everyRes); // false  
console.log(someRes);  // true
```

## ✅ Summary Table:

Method	Returns <b>true</b> if...	Best Use Case
<code>every()</code>	<b>All</b> elements meet the condition	Validate that all items meet a rule
<code>some()</code>	<b>At least one</b> element meets the condition	Check if any item passes a test

These two methods are excellent for quick validations and conditions in arrays — simple yet powerful!

## 🔍 JavaScript `find()` Method

### 📘 Definition:

The `find()` method returns the **first element** in an array that **satisfies a given condition**. If no elements match, it returns `undefined`.

- ✅ Stops as soon as it finds the first match.
- ❌ Unlike `filter()`, it does **not** return all matches.

### 🔧 Syntax:

```
array.find((element, index, array) => {  
  // return true to find the match  
});
```

### 🔪 Example 1: Finding the First Match

```
const peoples = [  
  { name: "huxn", age: 17 },  
  { name: "john", age: 18 },  
  { name: "alex", age: 20 },  
  { name: "jimmy", age: 30 },  
  { name: "alex", age: 30 },  
];  
  
const res = peoples.find((person) => person.name === "alex");  
console.log(res);
```

### Output:

```
{ name: 'alex', age: 20 }
```

🧠 Only the **first** match is returned. Use `filter()` if you want **all** matching elements.

---

🔧 Example 2: Finding a Post by Content

```
const posts = [
  { id: 1, content: "Good Post" },
  { id: 1, content: "funny Post" },
  { id: 1, content: "sad Post" },
];

const postRes = posts.find((post) => post.content === "funny Post");
console.log(postRes);
```

Output:

```
{ id: 1, content: "funny Post" }
```

⚠ Be careful with casing: `"funny post" ≠ "funny Post"`

---

🔄 `find()` vs `filter()`

Feature	<code>find()</code>	<code>filter()</code>
Return Type	Single element or <code>undefined</code>	Array of matched elements
Stops Early?	✅ Yes	❌ No, goes through entire array
Performance	Fastest for first match	Slower if many matches exist

---

Use `find()` when you want **just one result** quickly from a large array.

---

🌐 JavaScript `Map` Object – ES6 Feature

📖 What is a `Map`?



`Map` is a built-in data structure introduced in **ES6** that allows you to store **key-value pairs**, where **keys can be of any data type** — unlike regular JS objects.

---

🔍 Key Differences from Objects:

Feature	Object	Map
🔑 Key Types	Strings, Symbols only	Any data type (obj, fn, etc.)
🔄 Iteration Order	Not guaranteed	Maintains insertion order

---

Feature	Object	Map
 Iteration Support	Needs manual handling	Has built-in iterators
 Size Tracking	No <code>.size</code> , use <code>Object.keys()</code>	Has <code>.size</code> property

### Creating a Map:

```
const map = new Map();

const keyOne = "string";
const keyTwo = {};
const keyThree = function () {};

map.set(keyOne, "Value of key one");
map.set(keyTwo, "Value of key two");
map.set(keyThree, "Value of key three");

console.log(map.get(keyOne));    // Value of key one
console.log(map.get(keyTwo));    // Value of key two
console.log(map.get(keyThree));  // Value of key three
```

### Useful Methods:

```
map.set(key, value);    // Add or update an entry
map.get(key);           // Retrieve value by key
map.has(key);           // Check if a key exists
map.delete(key);        // Remove entry by key
map.clear();            // Remove all entries
map.size;               // Number of entries
```

### Iterating Over a Map:

```
for (let [key, value] of map) {
  console.log(`${key} -- ${value}`);
}

for (let key of map.keys()) {
  console.log(key);
}

for (let value of map.values()) {
  console.log(value);
}
```

---

## Initializing a Map with Entries:


```
const prefilledMap = new Map([
  ["name", "Vinayak"],
  ["age", 19],
  ["role", "Web Dev"],
]);
```

---

## When to Use Map?

- When you need keys other than strings (like objects/functions)
- When insertion order matters
- When you're frequently adding/removing key-value pairs

---

 Map is flexible, performant, and perfect when object limitations become a bottleneck.




---

## JavaScript Set – ES6 Feature

---

### What is a Set?

A **Set** is a built-in JavaScript object that stores a **collection of unique values** (no duplicates allowed). It can hold **any data type**: primitives, objects, or functions.

-  Automatically removes duplicates
-  Maintains insertion order
-  Cannot access elements by index

---

### Creating a Set:

```
const mySet = new Set(); // Empty set
const initialValues = [1, 2, 3];
const mySet2 = new Set(initialValues); // Set with values from array
```

---

### Example: Basic Operations

```
const mySet = new Set();

mySet.add("apple");
mySet.add("banana");
mySet.add("orange");
```



```
mySet.add("apple"); // Duplicate - ignored

console.log(mySet); // Set(3) { 'apple', 'banana', 'orange' }

console.log(mySet.has("banana")); // true
console.log(mySet.has("grape")); // false

mySet.delete("orange");
console.log(mySet); // Set(2) { 'apple', 'banana' }

mySet.clear(); // Removes all items
console.log(mySet); // Set(0) {}
```

### Example: Mixed Data Types

```
const set = new Set();

set.add(); // undefined
set.add("string");
set.add({ name: "huxn" });
set.add(10);

console.log(set.size); // 4
console.log(set.has(10)); // true
console.log(set.has("string")); // true
console.log(set.has({ name: "huxn" })); // false (different reference)

set.delete(10);
console.log(set); // Set now excludes 10
```

### Iterating Over a Set:

```
for (let item of set) {
  console.log(item);
}
```

### Useful Set Methods:

Method	Description
<code>add(value)</code>	Adds a new value to the set
<code>has(value)</code>	Returns <code>true</code> if value exists
<code>delete(value)</code>	Deletes a value from the set

Method	Description
<code>clear()</code>	Removes all values
<code>size</code>	Returns number of elements
<code>forEach(cb)</code>	Iterates like an array

### 🧠 Use Cases for Set:

- Removing duplicate values from an array
- Tracking unique items (e.g. visited URLs)
- Checking existence of a value in constant time
- Representing a mathematical set

### 💡 Tip:

```
const nums = [1, 2, 2, 3, 4, 4];
const unique = [...new Set(nums)]; // [1, 2, 3, 4]
```

✅ `Set` is powerful when uniqueness matters and indexing isn't required.

## 🔗 JavaScript `Symbol` – ES6 Feature

### 📖 What is a `Symbol`?

A `Symbol` is a **primitive data type** introduced in **ES6**.

It is **unique** and **immutable**, and is often used as a **key** for object properties to avoid naming conflicts.

✅ Even if two symbols have the same description, they are **always unique**.

### ✍️ Creating Symbols:

```
const sym1 = Symbol(); // Anonymous symbol
const sym2 = Symbol("My custom symbol"); // With description

console.log(sym1); // Symbol()
console.log(sym2); // Symbol(My custom symbol)
console.log(typeof sym1); // "symbol"
```

### ✍️ Symbols Are Always Unique:

```
const symbol1 = Symbol("name");
const symbol2 = Symbol("name");

console.log(symbol1 === symbol2); // false
```

Even though both have the same description, they are **not equal**.

### Using Symbols as Object Keys:

```
const symbol1 = Symbol("name");
const symbol2 = Symbol("name");


const huxn = {};
huxn.age = 17;
huxn["gender"] = "male";
huxn["female"] = "female";

huxn[symbol1] = "Alex";
huxn[symbol2] = "John";

console.log(huxn);
```

### Output:

```
{
  age: 17,
  gender: "male",
  female: "female",
  [Symbol(name)]: "Alex",
  [Symbol(name)]: "John"
}
```

 Even though both symbols have the same description ("name"), they are unique and do not overwrite each other.

### Why Use Symbols?

Use Case	Why It Matters
Unique object keys	Avoid key collisions
Hiding internal object details	Symbols don't show up in <code>for...in</code> loop
Implementing custom behavior (e.g., <code>Symbol.iterator</code> )	Useful in advanced scenarios

### ⚠ Note:

- Symbols are not enumerable in `for...in` loops.
  - They are ignored by `JSON.stringify()`.
- 

✅ Use `Symbol` when you want **non-conflicting, hidden keys** in objects, or need to create **constants** that are guaranteed to be **unique**.

---