📦 JavaScript Arrays

Arrays in JavaScript are powerful, flexible, and **mutable** data structures. Let's explore what they are and how we can work with them effectively.

🌟 What is an Array?

An **array** is a special variable that can hold more than one value at a time. It’s a type of **object** in JavaScript and is mutable — meaning its contents can be changed.

```
// Array literals
const fruits = ['apple', 'banana', 'orange', 7];
const numbers = new Array(1, 2, 3, 4, 5);
```

🎯 Accessing & Modifying Elements

```
console.log(fruits[0]); // 👉 'apple'
console.log(fruits[1]); // 👉 'banana'

// Modifying an element
fruits[1] = 'grape';
console.log(fruits); // 👉 ['apple', 'grape', 'orange', 7]
```

📏 Array Length & Type

```
console.log(fruits.length); // 👉 4
console.log(typeof fruits); // 👉 'object'
```

🧰 Commonly Used Array Methods

Method	Description
`push()` ➕	Add elements to the end
`pop()` ➖	Remove last element
`shift()` 🧹	Remove first element
`unshift()` ⬆️	Add elements to the beginning

Method	Description
`concat()` 🔗	Merge arrays
`slice()` ✂️	Extract section without modifying original
`splice()` 🧬	Add/remove elements
`includes()` 🔍	Check for existence
`indexOf()` 🔢	Find index
`reverse()` 🔄	Reverse array order
`sort()` 🔤	Sort elements
`map()` 🧠	Create new array by transforming values
`filter()` 🧼	Create new array based on condition
`reduce()` 🧮	Reduce to single value

And many more: `flat()`, `find()`, `every()`, `some()`, etc.

🧪 Array Practice Set

✅ Q1: How do you create an empty array?

```
const emptyArray = [];
```

✅ Q2: How to check if an array is empty?

```
const array = [];
if (array.length === 0) {
  console.log("✅ The array is empty");
} else {
  console.log("❌ The array is not empty");
}
// 👉 Output: ✅ The array is empty
```

✅ Q3: How to add elements to the end of an array?

```
const array = [1, 2, 3];
array.push(4, 5);
```

```
console.log(array);
// 👉 Output: [1, 2, 3, 4, 5]
```

✅ Q4: How to access an element at a specific index?

```
const array = [1, 2, 3];
console.log(array[1]);
// 👉 Output: 2
```

✅ Q5: How to remove the last element from an array?

```
const array = [1, 2, 3];
const removed = array.pop();
console.log(array);       // 👉 [1, 2]
console.log(removed);     // 👉 3
```

✅ Q6: How to find the index of a specific element?

```
const array = [1, 2, 3, 4, 5];
console.log(array.indexOf(3));
// 👉 Output: 2
```

✅ Q7: How to concatenate two arrays?

```
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];
const result = array1.concat(array2);
console.log(result);
// 👉 Output: [1, 2, 3, 4, 5, 6]
```

✅ Q8: How to check if an element exists in an array?

```
const array = [1, 2, 3, 4, 5];
console.log(array.includes(3));
// 👉 Output: true
```

☑ Q9: How to find the maximum value in an array?

```
const array = [4, 2, 7, 5, 1];
const max = Math.max(...array);
console.log(max);
// 👉 Output: 7
```

☑ Q10: How to reverse the order of elements in an array?

```
const array = [1, 2, 3, 4, 5];
array.reverse();
console.log(array);
// 👉 Output: [5, 4, 3, 2, 1]
```

📌 Summary

◇ Arrays are mutable objects that help store and manage lists of data. ◇ There are **dozens of built-in methods** to manipulate arrays easily. ◇ Always check length, use spread `...` for copying, and remember indexing starts from `0`.

◇ Creating Arrays

```
const myArr = [0, 1, 2, 3, 4, 5];
const myHeroes = ["Shaktiman", "Naagraj"];
const myArr2 = new Array(1, 2, 3, 4);
```

- `myArr` and `myHeroes` are arrays created using array literals.
- `myArr2` is created using the `Array` constructor.

◇ Accessing Elements

```
console.log(myArr[1]); // Output: 1
```

- Access elements using their index, starting from 0.

◇ Array Methods

◇ Adding and Removing Elements

```
myArr.push(6); // Adds 6 at the end
myArr.push(7); // Adds 7 at the end
myArr.pop();   // Removes the last element (7)

myArr.unshift(9); // Adds 9 at the beginning
myArr.shift();    // Removes the first element (9)
```

- `push()` adds elements to the end.
- `pop()` removes the last element.
- `unshift()` adds elements to the beginning.
- `shift()` removes the first element.

◇ **Checking for Elements**

```
console.log(myArr.includes(9)); // Output: false
console.log(myArr.indexOf(3));  // Output: 3
```

- `includes()` checks if an element exists in the array.
- `indexOf()` returns the index of the element or -1 if not found.

◇ **Joining Elements**

```
const newArr = myArr.join("-");
console.log(newArr); // Output: "0-1-2-3-4-5"
```

- `join()` combines array elements into a string, separated by the specified delimiter.

◇ Slice vs Splice

```
console.log("A ", myArr); // Output: A  [0, 1, 2, 3, 4, 5]

const myn1 = myArr.slice(1, 3);
console.log(myn1);        // Output: [1, 2]
console.log("B ", myArr); // Output: B  [0, 1, 2, 3, 4, 5]

const myn2 = myArr.splice(1, 3);
console.log("C ", myArr); // Output: C  [0, 4, 5]
console.log(myn2);        // Output: [1, 2, 3]
```

- `slice()` returns a shallow copy of a portion of an array without modifying the original array.
- `splice()` changes the contents of an array by removing or replacing existing elements.

◇ Combining Arrays

```javascript
const marvelHeroes = ["Thor", "Ironman", "Spiderman"];
const dcHeroes = ["Superman", "Flash", "Batman"];

const allHeroes = marvelHeroes.concat(dcHeroes);
console.log(allHeroes); // Output: ["Thor", "Ironman", "Spiderman", "Superman", "Flash", "Batman"]

const allNewHeroes = [...marvelHeroes, ...dcHeroes];
console.log(allNewHeroes); // Output: ["Thor", "Ironman", "Spiderman", "Superman", "Flash", "Batman"]
```

- `concat()` merges two or more arrays.
- Spread operator `...` also merges arrays.

◇ Flattening Arrays

```javascript
const anotherArray = [1, 2, 3, [4, 5, 6], 7, [6, 7, [4, 5]]];
const realAnotherArray = anotherArray.flat(Infinity);
console.log(realAnotherArray); // Output: [1, 2, 3, 4, 5, 6, 7, 6, 7, 4, 5]
```

- `flat()` creates a new array with all sub-array elements concatenated into it recursively up to the specified depth.

◇ Array Type Checks and Conversions

```javascript
console.log(Array.isArray("Hitesh"));      // Output: false
console.log(Array.from("Hitesh"));         // Output: ['H', 'i', 't', 'e', 's', 'h']
console.log(Array.from({ name: "Hitesh" })); // Output: []
```

- `Array.isArray()` checks if the value is an array.
- `Array.from()` creates a new array from an iterable or array-like object.

◇ Creating Arrays from Values

```javascript
let score1 = 100;
let score2 = 200;
let score3 = 300;

console.log(Array.of(score1, score2, score3)); // Output: [100, 200, 300]
```

- `Array.of()` creates a new array instance with the given arguments.

🧱 JavaScript Objects

◇ Creating Objects

```
const mySym = Symbol("key1");

const jsUser = {
    name: "Hitesh",
    "full name": "Hitesh Choudhary",
    [mySym]: "mykey1",
    age: 18,
    location: "Jaipur",
    email: "hitesh@google.com",
    isLoggedIn: false,
    lastLoginDays: ["Monday", "Saturday"]
};
```

- Objects can have properties with keys as strings, symbols, or numbers.

◇ Accessing and Modifying Properties

```
console.log(jsUser.email);           // Output: hitesh@google.com
console.log(jsUser["email"]);        // Output: hitesh@google.com
console.log(jsUser["full name"]);    // Output: Hitesh Choudhary
console.log(jsUser[mySym]);          // Output: mykey1

jsUser.email = "hitesh@chatgpt.com";
console.log(jsUser.email);           // Output: hitesh@chatgpt.com
```

- Access properties using dot notation or bracket notation.
- Modify properties by assigning new values.

◇ Freezing Objects

```
Object.freeze(jsUser);
jsUser.email = "hitesh@microsoft.com";
console.log(jsUser.email); // Output: hitesh@chatgpt.com
```

- `Object.freeze()` prevents modifications to the object.

◇ Adding Methods to Objects

```
jsUser.greeting = function() {
    console.log("Hello JS user");
};
```

```
jsUser.greetingTwo = function() {
    console.log(`Hello JS user, ${this.name}`);
};

jsUser.greeting();       // Output: Hello JS user
jsUser.greetingTwo();    // Output: Hello JS user, Hitesh
```

- Functions can be added as methods to objects.

◇ Creating Objects Using Constructors

```
const tinderUser = new Object();
tinderUser.id = "123abc";
tinderUser.name = "Sammy";
tinderUser.isLoggedIn = false;

console.log(tinderUser); // Output: { id: '123abc', name: 'Sammy', isLoggedIn: false }
```

- `new Object()` creates a new object.

◇ Nested Objects

```
const regularUser = {
    email: "some@gmail.com",
    fullname: {
        userfullname: {
            firstname: "Hitesh",
            lastname: "Choudhary"
        }
    }
};

console.log(regularUser.fullname.userfullname.firstname); // Output: Hitesh
```

- Access nested properties using chained dot notation.

◇ Merging Objects

```
const obj1 = { 1: "a", 2: "b" };
const obj2 = { 3: "a", 4: "b" };
const obj3 = { ...obj1, ...obj2 };

console.log(obj3); // Output: { '1': 'a', '2': 'b', '3': 'a', '4': 'b' }
```

- Spread operator `...` merges objects.

◇ Object Methods

```
console.log(Object.keys(tinderUser));    // Output: [ 'id', 'name', 'isLoggedIn' ]
console.log(Object.values(tinderUser));  // Output: [ '123abc', 'Sammy', false ]
console.log(Object.entries(tinderUser)); // Output: [ [ 'id', '123abc' ], [ 'name', 'Sammy' ], [ 'isLoggedIn', false ] ]
console.log(tinderUser.hasOwnProperty('isLoggedIn')); // Output: true
```

- `Object.keys()` returns an array of keys.
- `Object.values()` returns an array of values.
- `Object.entries()` returns an array of key-value pairs.
- `hasOwnProperty()` checks if the property exists in the object.

◇ Destructuring Objects

```
const course = {
    coursename: "JS in Hindi",
    price: "999",
    courseInstructor: "Hitesh"
};

const { courseInstructor: instructor } = course;
console.log(instructor); // Output: Hitesh
```

- Destructuring extracts properties into variables.

📄 JSON Structure

```
{
    "name": "Hitesh",
    "coursename": "JS in Hindi",
    "price": "free"
}
```

- JSON is a lightweight data-interchange format.

📋 Array, String, and Object Methods Summary

◇ Array Methods

Method	Description

Method	Description
push()	Adds elements to the end
pop()	Removes the last element
unshift()	Adds elements to the beginning
shift()	Removes the first element
includes()	Checks if an element exists
indexOf()	Returns the index of an element
join()	Joins elements into a string
slice()	Extracts a section of the array
splice()	Adds/removes elements
concat()	Merges arrays
flat()	Flattens nested arrays
isArray()	Checks if a value is an array
from()	Creates an array from iterable objects
of()	Creates an array from arguments

◇ String Methods

Method	Description
charAt()	Returns character at specified index
indexOf()	Returns index of first occurrence

📦 You Don’t Know Arrays in JavaScript: PACKED vs HOLEY | 🔍 JSVU & V8 Debug

---

🧠 Quick Summary

JavaScript arrays are not as simple as they look! They are optimized in various **internal representations** based on **what type of elements they store**, **how they're initialized**, and whether **indices are continuous or have holes**.

Let’s explore:

- Packed vs Holey arrays
- SMI, Double, Elements kind
- How V8 optimizes arrays internally

- Why performance degrades
- How to debug array internals using V8
- Installing JSVU and V8 Debug on Windows

📦 Packed vs 🕳️ Holey Arrays — Explained with Analogy

🎒 Packed Arrays

Think of a **school bag** with **books kept back-to-back**. No empty slots.

```
const arr = [1, 2, 3, 4];
```

✅ Elements are **contiguous and without gaps** ✅ All values are of **same type (e.g., integers)** ✅ Fastest for the JS engine to process

➡️ Internally called:

- `PACKED_SMI_ELEMENTS` (for small integers)
- `PACKED_DOUBLE_ELEMENTS` (for floats)
- `PACKED_ELEMENTS` (for objects/strings)

🕳️ Holey Arrays

Now imagine a **bag with torn compartments** — books are missing in between.

```
const arr = [1, , 3]; // hole at index 1
```

❌ Contains **holes (missing elements)** ❌ More expensive to access ❌ Triggers fallback logic like `HasOwnProperty`, prototype checks

➡️ Internally called:

- `HOLEY_SMI_ELEMENTS`
- `HOLEY_DOUBLE_ELEMENTS`
- `HOLEY_ELEMENTS`

⚙️ How V8 Internally Optimizes Arrays

Depending on array usage, V8 optimizes arrays into **different element kinds**:

Type	Internal Kind	Example
Integers (no holes)	`PACKED_SMI_ELEMENTS`	`[1, 2, 3]`
Floats (no holes)	`PACKED_DOUBLE_ELEMENTS`	`[1.1, 2.2]`

Type	Internal Kind	Example
Mixed types	`PACKED_ELEMENTS`	`[1, 'a', {}, () => {}]`
Integers + holes	`HOLEY_SMI_ELEMENTS`	`[1, , 3]`
Floats + holes	`HOLEY_DOUBLE_ELEMENTS`	`[1.1, , 3.3]`
Mixed + holes	`HOLEY_ELEMENTS`	`[1, , 'a']`

📊 Performance Ranking (Fast ➡️ Slow)

1. 🥇 `PACKED_SMI_ELEMENTS` (pure integers)
2. 🥈 `PACKED_DOUBLE_ELEMENTS` (floats)
3. 🥉 `PACKED_ELEMENTS` (objects, strings)
4. ⚠️ `HOLEY_SMI_ELEMENTS`
5. ⚠️ `HOLEY_DOUBLE_ELEMENTS`
6. ⚠️ `HOLEY_ELEMENTS`

⛔ Once downgraded (e.g., from SMI to Double or Holey), **it cannot be upgraded back**.

🔬 Examples of Array Downgrades

```
const arr = [1, 2, 3];  // PACKED_SMI_ELEMENTS

arr.push(7.0);          // ➡️ still okay (int + float → PACKED_DOUBLE_ELEMENTS)

arr.push("8");          // ➡️ mixed types → PACKED_ELEMENTS

arr[10] = 11;           // ➡️ holes → HOLEY_ELEMENTS

console.log(arr[19]);   // undefined, adds more holes
```

🧪 Debugging Arrays with `%DebugPrint()`

🚨 Only works inside V8 or with debug flag

```
%DebugPrint(arr);  // Prints internal array details
```

Shows:

- Kind (PACKED/Holey)
- Elements type
- Length and memory layout

🛠️ How to Install JSVU & V8 Debug on Windows

📌 Step 1: Install JSVU

JSVU = JavaScript Virtual Machines Updater

```
npm install -g jsvu
```

Add to PATH:

```
set PATH=%USERPROFILE%\.jsvu;%PATH%
```

Install V8:

```
jsvu --os=windows --engines=v8
```

Now run V8:

```
v8
```

🔍 Using V8 with Debug Flags

```
v8 --allow-natives-syntax
```

Then:

```
const arr = [1, 2, 3];
%DebugPrint(arr);
```

🧠 Technical Concepts Explained

🔢 SMI (Small Integers)

- Efficient memory representation of integers in JS engine (usually 31-bit)
- Fastest element type

🔣 Double

- For floats, NaN, Infinity
- Uses more memory than SMI

🧩 Object Elements

- Strings, functions, objects
- Slowest to process

🧠 Why `arr[19]` is expensive?

```
console.log(arr[19]);
```

- Requires **bounds check**
- Then `HasOwnProperty(arr, 19)`
- Then `HasOwnProperty(arr.prototype, 19)`
- Then `Object.prototype` check

⚠️ Holes make this **super expensive**!

🚫 Once Downgraded, Can't Go Back!

```
const arr = [1, 2, 3];  // Packed SMI

arr.push('4');          // → Packed Elements
arr.pop();              // still Packed Elements

// ❌ Does NOT revert to Packed SMI
```

🕳️ Creating Holey Arrays

```
const arr = new Array(3);  // [ <3 empty items> ]
// HOLEY_SMI_ELEMENTS

arr[0] = 1;
arr[1] = 2;
arr[2] = 3;
// Still holey because initialized with holes
```

❓ What is NaN in JS?

- `NaN` = Not a Number
- Type: `number`

- Example of **Double**

```
typeof NaN; // 'number'
```

⚡ Tip: Use Internal Methods (Like `.forEach`) Whenever Possible

```
const arr = [1, 2, 3];
arr.forEach(el => console.log(el)); // ✅ Fastest
```

Instead of:

```
for (let i = 0; i < arr.length; i++) {
  if (arr.hasOwnProperty(i)) {
    console.log(arr[i]);
  }
}
```

🛑 User-defined logic = slower, less optimized

🧠 Recap Cheatsheet

Concept	Description
Packed Array	No holes, uniform types
Holey Array	Gaps in index, degraded performance
SMI	Small Integers (fastest)
Double	Floats, NaN, Infinity
Packed → Holey	Downgrade happens automatically
Downgrade irreversible	Can’t go from HOLEY → PACKED
V8 Optimization	Based on content & structure of array
%DebugPrint	Shows V8 internals (only in debug environment)
Use built-ins	`.forEach`, `.map` are better than custom loops

🙌 Author: Darshan Vasani

🔗 dpvasani56.vercel.app 🐙 GitHub | 💼 LinkedIn | 📬 Topmate
