# Callback Function

A **callback function** is a function that is **passed as an argument** to another function **and gets executed after a task is completed**.

### Why do we use callbacks?

Callbacks are mainly used to handle:

- **Asynchronous operations**
- **API calls**
- **Time-based tasks** (`setTimeout`, `setInterval`)
- **Tasks that take unknown time**

### Example:

```
function greet(name, callback) {
    console.log("Hello " + name);
    callback();
}

greet("Jagath", function() {
    console.log("Welcome!");
});
```

---

# ✅Promises

A **Promise** is an object that represents the eventual completion of an asynchronous task.

- It helps convert **callback-based async code → cleaner async code**.
- Promises help avoid **callback hell**.
- Promises have 3 states:
  **Pending → Resolved → Rejected**

---

# JavaScript Synchronous or Asynchronous?

JavaScript is:

- **Synchronous by default**
- **Single-threaded** (executes one line at a time)
- But supports **Asynchronous behavior** using:
  - Callbacks
  - Promises
  - `async/await`
  - Event loop

So JS is synchronous but it **handles async tasks using browser APIs + event loop**

# Types of call back Functions

## ⭑ 1. Synchronous Callback (Important)

A **synchronous callback** runs **immediately** during the execution of the main function.

### Example:

```
function process(num, callback) {
    callback(num);
}

process(10, function(n) {
    console.log("Number:", n);
});
```

**Runs instantly** → No waiting.

---

## ⭑ 2. Asynchronous Callback (Important)

An **asynchronous callback** runs **later**, after some time or after an async task is completed.

Used in:

- API calls
- Timers (`setTimeout`, `setInterval`)
- Fetch
- Database calls

### Example:

```
setTimeout(function() {
    console.log("Async callback executed");
}, 2000);
```

Runs **after 2 seconds**.

---

## 3. Anonymous Callback

A callback function **without a name**.

### Example:

```
arr.forEach(function(item) {
    console.log(item);
});
```

Used most commonly in array methods and API calls.

---

# 4. Named Callback

A callback function **with a name**, passed as an argument.

### Example:

```
function done() {
    console.log("Task completed!");
}

function runTask(callback) {
    callback();
}

runTask(done); // passing a named callback
```

## Array Methods — Expanded Interview Answers

---

# 1. push()

`push()` adds one or more elements to the **end** of an array and returns the **new length**.
It **changes the original array**.

---

# 2. pop()

`pop()` removes the **last element** from the array and returns the **removed element**.
If the array is empty, it returns **undefined**.
It **modifies the original array**.

---

# 3. unshift()

`unshift()` adds one or more elements to the **beginning** of the array and returns the **new length**.
It **shifts existing elements to the right** and **modifies the original array**.

---

# 4. shift()

`shift()` removes the **first element** of the array and returns the **removed value**.
It shifts remaining elements to the left and **modifies the original array**.

---

# 5. slice(start, end)

`slice()` returns a **portion** of the array based on the start and end indexes.
It **does NOT modify** the original array.
The end index is **not included**.

Example: `slice(1, 4)` → takes index **1,2,3**.

---

# 6. splice(start, deleteCount, items...)

`splice()` **modifies** the original array.
It can:

- remove elements
- add elements
- replace elements

It returns the **removed elements** as a new array.

Example:
`splice(2, 1, "x")` → removes 1 item at index 2 and inserts `"x"`.

---

# 7. concat()

`concat()` combines two or more arrays and returns a **new array**.
It does **not** change the original arrays.
Used for merging arrays without mutation.

---

# 8. indexOf(value)

`indexOf()` returns the **first index** where a value appears.
If the value does not exist, it returns **-1**.
Search is **strict equality** (===).

---

# 9. includes(value)

`includes()` checks if a value exists in the array and returns **true or false**.
It is easier and more readable than checking `indexOf !== -1`.

---

# 10. reverse()

`reverse()` reverses the array **in place** — meaning it **modifies the original array**.
It returns the same reversed array reference.

---

# 11. join()

`join()` converts all elements of an array into a **single string**, separated by the given separator.
Default separator is a comma (`,`).

Example:
`[1,2,3].join("-")` → `"1-2-3"`

# Naming Conventions (Short Notes)

### camelCase

- First word small, next words start Capital.
- Used in JS variables/functions.
- Example: `firstName`

### PascalCase

- Every word starts Capital.
- Used for classes, React components.
- Example: `UserProfile`

### snake_case

- Words separated by `_`.
- Used in Python, SQL, file names.
- Example: `user_name`

### kebab-case

- Words separated by `-`.
- Used in CSS/Tailwind/Bootstrap.
- **Not allowed in JS variables**.
- Example: `text-center`
-

# Higher Order Function (HOF)

A **higher-order function** is a function that **takes another function as an argument**, **returns a function**, or **does both**.
Used in: `map`, `filter`, `reduce`, callbacks.

# Higher Order Component (HOC) – React

A **Higher Order Component** is a **function that takes a component and returns a new enhanced component**.
Used for: sharing logic, authentication wrappers, logging, conditional UI.

**Example:**

```
const withAuth = (Component) => (props) => {
  return <Component {...props} />;
};
```

## Higher-Order Array Methods (Require Callback & Work Only on Arrays)

Let:

```
let arr = [1, 2, 3];
```

---

# 1. forEach(callback)

**Interview Answer:**
`forEach()` executes a callback **for every element** in the array.
It is used only for **iteration / side effects** (logging, updating values, API calls).
It **does NOT return a new array**.

### Example:

```
arr.forEach((element, index) => {
  console.log(element, index);
});
```

---

## ❓Why should we NOT use forEach in React? (Interview Answer)

**Interview Answer:**
`forEach()` should not be used in React rendering because:

1. **It does not return anything**, so you **cannot return JSX** from it.
2. React rendering requires a **returned array of elements**.
3. `map()` returns a **new array**, making it perfect for creating UI lists.

4. `forEach()` is only for side-effects, not for building components.

### Example (Correct):

```
{arr.map(item => <p>{item}</p>)}
```

### Example (Wrong):

```
arr.forEach(item => <p>{item}</p>) // returns undefined
```

## 2. map(callback)

**Interview Answer:**
`map()` loops through each element and **returns a new array** with the transformed values.
It does **not modify** the original array.

### Example:

```
const nums = [1, 2, 3];
const doubled = nums.map(n => n * 2);
console.log(doubled); // [2, 4, 6]
```

---

## ☑3. filter(callback)

**Interview Answer:**
`filter()` returns a **new array** containing only the elements that **pass the condition** (true).

### Example:

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

---

## ☑4. find(callback)

**Interview Answer:**
`find()` returns the **first matching element** that satisfies the condition.
If no match → returns **undefined**.

### Example:

```
const nums = [10, 20, 30];
const result = nums.find(n => n > 15);
console.log(result); // 20
```

---

## ✅5. findIndex(callback)

**Interview Answer:**
`findIndex()` returns the **index** of the first element that matches the condition.
If no match → returns **-1**.

### Example:

```
const nums = [10, 20, 30];
const index = nums.findIndex(n => n === 20);
console.log(index); // 1
```

---

## ✅6. reduce(callback, initialValue)

**Interview Answer:**
`reduce()` reduces the array to a **single value** (sum, product, total, object, etc.).
Uses an **accumulator** that stores the running result.

### Example (sum):

```
const nums = [1, 2, 3];
const total = nums.reduce((acc, num) => acc + num, 0);
console.log(total); // 6
```

---

## ✅7. some(callback)

(*You wrote sum(), correct word is* **some()**)

**Interview Answer:**
`some()` returns **true** if **any one** element matches the condition.
If all are false → returns **false**.

### Example:

```
const nums = [1, 2, 3];
const result = nums.some(n => n > 2);
console.log(result); // true
```

---

## ✅8. every(callback)

**Interview Answer:**
`every()` returns **true** only if **all** elements pass the condition.
If even one false → returns **false**.

### Example:

```
const nums = [2, 4, 6];
const result = nums.every(n => n % 2 === 0);
console.log(result); // true
```

---

## ✅9. sort()

**Interview Answer:**
`sort()` sorts the array **in place** (modifies original).
For numbers, we use a **compare function**:
`(a, b) => a - b` sorts in **ascending** order.

### Explanation:

```
a - b
```

- negative → a comes first
- positive → b comes first

### Example:

```
const nums = [30, 10, 20];
const sorted = nums.sort((a, b) => a - b);
console.log(sorted); // [10, 20, 30]
```

## Shallow Copy vs Deep Copy (Very Important for Interviews)

# Shallow Copy

☞ Creates a **new array**, but **inner objects/arrays still refer to the original memory**.
☞ Changing nested values **affects the original**.

**Interview Definition:**
*A shallow copy copies only the outer structure, but nested objects are still shared.*

**Example:**

```
const a = [1, {x:10}];
const b = [...a];  // shallow copy
b[1].x = 20;
console.log(a[1].x); // 20 (changed)
```

---

# Deep Copy

☞ Creates a **completely independent copy**, including all nested objects.
☞ Changing deep values **does NOT affect** the original.

**Interview Definition:**
*A deep copy copies the entire structure, including nested objects, creating a fully separate object.*

**Example:**

```
const a = [1, {x:10}];
const b = JSON.parse(JSON.stringify(a));  // deep copy
b[1].x = 20;
console.log(a[1].x); // 10 (not changed)
```

---

## ☑Array Methods That Make Shallow Copy vs Deep Copy

# ✔ Shallow Copy Methods (Important)

These **return a new array** but still **share reference** for nested objects:

- **slice()**
- **concat()**
- **map()** (if returning same object)
- **filter()**
- **spread operator** `[...]`
- **Array.from()**

✔ They copy the outer array only.

---

# ✕Deep Copy Methods

JavaScript has **no built-in deep copy method** for arrays.

But commonly used techniques:

- `JSON.parse(JSON.stringify(array))`
- `structuredClone(array)` ✔ (Modern JS)
- Manual deep copy using recursion

---

## ❗ Important: Mutating Methods (Not Copying)

These do NOT create copies — they **modify the original array** directly:

- **push()**
- **pop()**
- **shift()**
- **unshift()**

- **splice()**
- **sort()**
- **reverse()**

These are **deeply mutating**, not copying.

### Expression vs Statement (Very Important for Interviews)

# What is a Statement?

A **statement** performs an action.
It **does not return a value**.

**Examples:**

```
let a;        // declaration statement
if (x > 5) {}  // condition statement
for (...) {}   // loop statement
```

**Interview Definition:**
*A statement tells JavaScript to do something.*

---

# What is an Expression?

An **expression** produces a **value**, and it **can be used inside statements**.

**Examples:**

```
10 + 20          // expression → returns 30
x > 5            // expression → true/false
let a = 10;      // 10 is expression
```

**Interview Definition:**
*An expression evaluates to a value and can be used wherever a value is expected.*