Spread & Rest Operator

1. ... (Three Dots)

Used as:

- **Spread Operator**
- **Rest Operator**

Depends on where it is used.

2. Spread Operator

Definition

Expands elements of an array or object.

With Objects

const emp = { ...person, job: "Trainer" };

- Creates a **shallow copy**
- Can add or override properties
- If duplicate keys → **last value wins**

With Arrays

const newArr = [...arr, 5, 6];

- Expands elements
- Creates a new array reference
- Useful for copying arrays

Important

- Spread does **not** create deep copy
- Nested objects remain referenced

3. Rest Operator

Definition

Collects multiple arguments into a single array.

function sum(...nums) {}

Rules

- Must be the last parameter
- Only one rest parameter allowed
- Always returns an array

4. Spread vs Rest

Spread	Rest
Expands values	Collects values
Used in arrays/objects	Used in function parameters
Creates shallow copy	Forms an array

5. Common Use Cases

- React state updates
- Merging objects
- Copying arrays
- Handling variable function arguments

Anonymous & Arrow Function

Anonymous Function

const fn = function () {};

- No function name
- Stored inside variable
- Uses function keyword
- Has its own this
- Can be used as constructor

Arrow Function

const fn = () => {};

- Short syntax
- No function keyword
- No own this (lexical this)
- Cannot be used as constructor
- No arguments object

Key Differences

Anonymous (function)	Arrow Function
Has its own this	Inherits this
Has arguments	No arguments
Can use new	Cannot use new
Traditional syntax	Short syntax

Implicit vs Explicit Return

Implicit Return

const square = (s) => s * s;

- Single expression
- No {}

Explicit Return

```
const fn = () => {
  return value;
};
```

- Uses {} block
- Must write return

Returning Object

Must wrap in parentheses:

const fn = () => ({ name: "DK" });

Without () → treated as block, not object.

Use Cases

map()

Definition

map() transforms each element of an array and returns a **new array** of same length.

Syntax

```
array.map((currentValue, index, array) => {
  return newValue;
});
```

- currentValue → current element
- index → position
- array → original array

Example

```
const doubled = numbers.map((value) => value * 2);
console.log(doubled);
```

Output

[2, 4, 6, 8, 10]

Internally What Happens

Iteration 1 → 1 × 2 → 2
Iteration 2 → 2 × 2 → 4
Iteration 3 → 3 × 2 → 6
Iteration 4 → 4 × 2 → 8
Iteration 5 → 5 × 2 → 10
Returned array length = original array length

Important Points

- Does NOT modify original array
- Always returns new array
- Output length = input length
- Used for transformation

filter()

Definition

filter() returns a new array containing only elements that satisfy a condition.

Syntax

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

Returns element only if condition is true.

Example

```
const evenNumbers = numbers.filter((value) => value % 2 === 0);
console.log(evenNumbers);
```

Output

[2, 4]

Internally What Happens

1 → false
2 → true (kept)
3 → false
4 → true (kept)
5 → false

Important Points

- Returns new array
- Output length ≤ original length
- Used for selection / condition filtering

reduce()

Definition

reduce() reduces an array to a **single value**.

Syntax

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

- accumulator → stores result
- currentValue → current element
- initialValue → starting value

Example – Product

const product = numbers.reduce((acc, value) => acc * value, 1);

console.log(product);

Step-by-step Execution

Initial acc = 1

1 × 1 = 1

1 × 2 = 2

2 × 3 = 6

6 × 4 = 24

24 × 5 = 120

Output

120

Example – Sum

const sum = numbers.reduce((acc, value) => acc + value, 0);

Output:

15

Important Points

- Returns single value
- Very powerful
- Can replace map + filter + loops
- Most difficult but most flexible

Comparison Table

Method	Returns	Length	Purpose
map()	New Array	Same	Transform
filter()	New Array	≤ Same	Select
reduce()	Single Value	1	Accumulate

Visual Understanding

numbers = [1,2,3,4,5]

map() → [2,4,6,8,10]

filter() → [2,4]

reduce() → 120

Callback & Asynchronous Nature

What is a Callback?

A callback is a function passed as an argument to another function.

- It is **not executed immediately**
- It is invoked by the parent function
- Usually executed after some operation (timer, API call, event)

setTimeout()

setTimeout(callbackFunction, delayInMilliseconds);

- Executes once
- Runs after minimum delay

setInterval()

setInterval(callbackFunction, delay);

- Executes repeatedly
- Continues until cleared using clearInterval()

How JavaScript Executes (Very Important)

JavaScript is:

- Single-threaded
- Non-blocking
- Asynchronous

Execution Flow

1. V8 Engine executes synchronous code
2. Timer functions go to Browser (Web APIs)
3. After delay → moved to Callback Queue
4. Event Loop checks:
    - If Call Stack empty → pushes callback
5. Callback executes

Why Output is “Start End SetTimeout”?

Because:

- console.log("Start") → runs immediately
- setTimeout() → goes to Web API
- console.log("End") → runs immediately
- After 3 sec → callback pushed to queue

- Event loop executes it

FIFO Nature

Callback Queue follows:

First In → First Out

But execution depends on:

- Delay completion time
- Not order written in code

Why 0ms Still Runs Later?

setTimeout(() => {}, 0);

Even 0ms:

- Goes to Web API
- Waits for call stack to clear
- Then executes

So it is still asynchronous.

Promise (Based on MongoDB Connection Example)

What is a Promise?

A Promise is an object that represents the eventual result of an asynchronous operation.

In our class example:

mongoose.connect("MONGO_URL")

mongoose.connect() returns a **Promise**.

That means:

- Database connection will take time.
- JavaScript will not wait.
- Instead, it gives a Promise object.

Promise States

A Promise has three states:

1. Pending → Initial state (connection in progress)
2. Fulfilled (Resolved) → Connection successful
3. Rejected → Connection failed

Using .then() and .catch()

Class Example

```
mongoose
 .connect("MONGO_URL")
 .then(() => {
  console.log("Connected to MongoDB");

  app.listen(5050, () => {
   console.log("Server is running on port 5050");
  });
 })
 .catch((err) => {
  console.log(err);
 });
```

How This Works

Step 1

mongoose.connect() starts DB connection

Promise state → Pending

Step 2

If success → Promise becomes Fulfilled

.then() block executes

Step 3

If error → Promise becomes Rejected

.catch() block executes

Why Server Starts Inside .then()?

```
.then(() => {
  app.listen(5050)
})
```

We start the server only after DB connection succeeds.

If DB is not connected:

- Server should not start
- Otherwise API may fail

This is real production practice.

Async / Await Version

Class Example

```
const connectDB = async () => {
 try {
  await mongoose.connect("MONGO_URL");

  console.log("Connected to MongoDB");

  app.listen(5050, () => {
   console.log("Server is running on port 5050");
  });

 } catch (err) {
  console.log(err);
 }
};

connectDB();
```

How Async/Await Works

- async makes function return a Promise
- await pauses execution until Promise resolves
- try block → handles resolved state
- catch block → handles rejected state

.then() vs async/await

.then() / .catch()	async / await
Chain-based	Looks synchronous
Harder if nested	Cleaner structure
Uses promise methods	Uses try/catch
Good for chaining	Better readability

Both do the same thing internally.

Behind the Scenes (Important Concept)

When mongoose.connect() runs:

1. Connection goes to Node environment (handled asynchronously)
2. Promise is returned immediately
3. When completed:
    - If success → .then() goes to Microtask Queue
    - If failure → .catch() goes to Microtask Queue
4. Event Loop pushes it to Call Stack
5. Code executes

Important:

Promise callbacks use **Microtask Queue**, which has higher priority than setTimeout.

API Calls in Express (Backend)

What is an API Call?

An API call is:

Sending an HTTP request to a server and receiving a response.

In backend (Node.js):

- We use **Axios** or **fetch**
- Axios returns a **Promise**
- We handle it using:
  - .then() / .catch()
  - async / await

Express Request Object (req)

The req object contains data sent by the client.

req.params

Used for route parameters.

```
app.get("/user/:id", (req, res) => {
  console.log(req.params.id);
});
```

Request:

GET /user/101

Output:

101

req.query

Used for query parameters.

```
app.get("/search", (req, res) => {
  console.log(req.query.name);
});
```

Request:

GET /search?name=DK

Output:

DK

req.body

Used for POST/PUT requests.

Requires middleware:

app.use(express.json());

Example:

```
app.post("/user", (req, res) => {
  console.log(req.body);
});
```

Client sends:

```
{
  "name": "DK",
  "age": 35
}
```

Express Response Object (res)

Used to send data back to client.

res.json()

res.json({ message: "Success" });

Sends JSON response.

res.status()

res.status(200).json({ success: true });

Common status codes:

- 200 → OK
- 201 → Created
- 400 → Bad Request
- 401 → Unauthorized
- 403 → Forbidden
- 404 → Not Found
- 500 → Server Error

Making API Calls Using Axios

Install:

npm install axios

Basic GET Request

```
const response = await axios.get("https://api.example.com/data");
console.log(response.data);
```

Axios returns a **Promise**.

Using Params (Cleaner Way)

```
await axios.get(BASE_URL, {
  params: { year: 2020 }
});
```

Axios automatically builds:

?year=2020

Sending Headers

```
await axios.get(BASE_URL, {
  headers: {
    "User-Agent": "Mozilla/5.0"
  }
});
```

Handling Promise

Using .then() / .catch()

```
axios.get(URL)
  .then(res => {
    console.log(res.data);
  })
  .catch(err => {
    console.log(err.message);
  });
```

Using async / await (Recommended)

```
try {
  const response = await axios.get(URL);
  console.log(response.data);
} catch (error) {
  console.log(error.message);
}
```

Cleaner and more readable.

Multiple Promises

Promise.all()

Runs all promises in parallel.

```
const results = await Promise.all([
  axios.get(url1),
  axios.get(url2),
]);
```

✔ Fast

❌ Fails if one fails

Promise.allSettled()

Waits for all promises, even if some fail.

```
const results = await Promise.allSettled([
  axios.get(url1),
  axios.get(url2),
]);
```

Returns status for each:

- fulfilled
- rejected

Promise.race()

Returns the first completed promise.

```
const result = await Promise.race([
  axios.get(url1),
  axios.get(url2),
]);
```

Whichever finishes first wins.

Promise.any()

Returns first successful promise.

```
const result = await Promise.any([
  axios.get(url1),
  axios.get(url2),
]);
```

✔ Ignores failures

❌ Fails only if all fail

Sequential vs Parallel Calls

Sequential:

```
await axios.get(url1);
await axios.get(url2);
```

Runs one after another (slow).

Parallel:

```
await Promise.all([
  axios.get(url1),
  axios.get(url2),
]);
```

Runs together (fast).

Enterprise Best Practices

✔ Use async/await
✔ Use try/catch
✔ Centralize axios config
✔ Use environment variables
✔ Validate request inputs
✔ Handle errors properly
✔ Never expose API keys