

Explain reduceRight function



explain this reduce right
 const nested = arr.reduceRight((acc, curr) => ({ [curr]: acc }), {});
 console.log(nested);

Let's break this down **slowly and in beginner-friendly style**, Vipul.
 You will understand exactly what **reduceRight** is doing here.

✓ The Code

```
js

const nested = arr.reduceRight((acc, curr) => ({ [curr]: acc }), {});
console.log(nested);
```

🔥 First: What is reduceRight?

reduceRight works exactly like **reduce**, **but it starts from the LAST element and moves to the FIRST element**.

Example:

If **arr** = ["a", "b", "c"]

- **reduce** processes → a → b → c
- **reduceRight** processes → c → b → a

🎯 What does this logic do?

The function:

```
js

(acc, curr) => ({ [curr]: acc })
```

This means:

- For each element (**curr**)
- Create a **new object**
- Key = curr

- Value = acc

Your initial **acc** (accumulator) starts as {}
(because you passed {} as the second argument of reduceRight)

Example Walkthrough

Assume:

```
js
const arr = ["a", "b", "c"];
```

Start from the rightmost element ("c").

Step 1: curr = "c", acc = {}

```
json
{ "c": {} }
```

Step 2: curr = "b", acc = { c: {} }

```
css
{ "b": { "c": {} } }
```

Step 3: curr = "a", acc = { b: { c: {} } }

```
css
{ "a": { "b": { "c": {} } } }
```

Final Output:

```
js
{
  a: {
    b: {
      c: {}
    }
  }
}
```

★ Explanation in One Line

This code converts an array like:

```
css
```

```
[ "a", "b", "c" ]
```

into a **deeply nested object**:

```
css
```

```
{ a: { b: { c: {} } } }
```

Each element becomes a *key* inside the previous key.

👉 Why use `reduceRight`?

Because we want the **last element inside**, then outer elements wrap around it.

If you used normal `reduce`, you would get:

```
css
```

```
{ c: { b: { a: {} } } }
```

Which is reversed.

💡 Try it with your own array

Tell me your array, and I'll show you the exact step-by-step output.



what's the difference between doing this:

```
const nested = arr.reduceRight((acc, curr) => ({ [curr]: acc }), {});
```

and this

```
const nested = arr.reduceRight((acc, curr) => { [curr]: acc }, {});
```

the first one is giving this output:

```
{ a: { b: { c: {} } } }
```

and the later one giving this:

ERROR!

```
/tmp/5qiDHzX9bp/main.js:22
```

```
const nested = arr.reduceRight((acc, curr) => { [curr]: acc }, {});  
^
```

SyntaxError: Unexpected token ':'
 at wrapSafe (node:internal/modules/cjs/loader:1662:18)
 at Module._compile (node:internal/modules/cjs/loader:1704:20)
 at Object..js (node:internal/modules/cjs/loader:1895:10)
 at Module.load (node:internal/modules/cjs/loader:1465:32)
 at Function._load (node:internal/modules/cjs/loader:1282:12)
 at TracingChannel.traceSync (node:diagnostics_channel:322:14)
 at wrapModuleLoad (node:internal/modules/cjs/loader:235:24)
 at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:171:5)
 at node:internal/main/run_main_module:36:49

Node.js v22.16.0

Great question, Vipul!

This is a **very important JavaScript nuance** — the difference between:

✓ **Returning an object *implicitly***

vs.

✗ **Trying to write an object literal inside a function body without return.**

Let's break it down.

✓ 1. This works

js

```
const nested = arr.reduceRight((acc, curr) => ({ [curr]: acc }), {});
```

Why does this work?

Here you are using **implicit return with parentheses**:

css

```
({ [curr]: acc })
```

When you wrap an object literal inside ()

→ JavaScript understands that **this is a value** to return.

So it returns:

```
js
```

```
{ [curr]: acc }
```

✖ 2. This gives error

```
js
```

```
const nested = arr.reduceRight((acc, curr) => { [curr]: acc }, {});
```

Why this FAILS?

Because now your arrow function uses **curly braces {}**, which means:

👉 JavaScript treats it as a **function body**, NOT as an object.

Inside function body:

```
js
```

```
{ [curr]: acc }
```

Is treated like:

```
css
```

```
{
  [curr]: acc; // <- JS thinks this is a label, not an object
}
```

This is **invalid syntax**, causing:

```
javascript
```

```
SyntaxError: Unexpected token ':'
```

And also even if it didn't crash, it still would return **undefined** because there is **no return statement** inside the body.

🧠 Simple Rule to Remember

✓ **If you want to return an object implicitly, use parentheses:**

```
js
```

```
(curr) => ({ x: 1 })
```

✓ **If you want to use curly braces {}, you MUST write return:**

```
js
(curr) => { return { x: 1 }; }
```

✓ Correct version using function body

```
js
const nested = arr.reduceRight((acc, curr) => {
  return { [curr]: acc };
}, {});
```

This will give the same output:

```
css
{ a: { b: { c: {} } } }
```

🔥 Why the error happens (in one line)

Because:

```
css
{ [curr]: acc }
```

inside an arrow function body
is treated as **a block**, not an object → JavaScript gets confused → syntax error.

If you want, I can explain how implicit vs explicit return works in arrow functions with diagrams.



```
const nested = arr.reduceRight((acc, curr) => {
  [curr]= acc
  return acc;
}
, {});
```

it is giving this issue :

ERROR!

/tmp/8kYx9qOZmn/main.js:23

```
[curr]= acc
^
```

TypeError: object is not iterable (cannot read property Symbol(Symbol.iterator))
at /tmp/8kYx9qOZmn/main.js:23:11

```

at Array.reduceRight (<anonymous>)
at Object.<anonymous> (/tmp/8kYx9qOZmn/main.js:22:20)
at Module._compile (node:internal/modules/cjs/loader:1730:14)
at Object..js (node:internal/modules/cjs/loader:1895:10)
at Module.load (node:internal/modules/cjs/loader:1465:32)
at Function._load (node:internal/modules/cjs/loader:1282:12)
at TracingChannel.traceSync (node:diagnostics_channel:322:14)
at wrapModuleLoad (node:internal/modules/cjs/loader:235:24)
at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:171:5)

```

Node.js v22.16.0

why?

Vipul, this is a **perfect question** — and the error makes total sense once you understand what JavaScript is trying to do.

Let's break it down cleanly.

✗ Your code

```

js

const nested = arr.reduceRight((acc, curr) => {
  [curr] = acc
  return acc;
}, {});

```

The error:

```

vbnet

TypeError: object is not iterable (cannot read property Symbol(Symbol.iterator))

```

✓ Why does this error happen?

Because this line:

```

js

[curr] = acc

```

Is interpreted by JavaScript as **array destructuring**, not as "setting a key".

JavaScript thinks you are trying to do:

cpp

```
const [someVariable] = acc; // destructure acc as an array
```

But **acc** is an **object**, not an array.

Objects are NOT iterable → BOOM  error.

So what did you actually want?

You wanted:

```
acc[curr] = accValue
```

But what you wrote:

csharp

```
[curr] = acc
```

Means:

- You are trying to assign **acc** into an array destructuring target
- With no variable name
- Which is invalid

Correct version (if you want to mutate the accumulator)

js

```
const nested = arr.reduceRight((acc, curr) => {
  acc = { [curr]: acc }; // create new object with curr as key
  return acc;
}, {});
```

Or explicitly:

js

```
const nested = arr.reduceRight((acc, curr) => {
  const obj = {};
  obj[curr] = acc;
  return obj;
}, {});
```

Why simple assignment doesn't work?

Because **setting an object key using a dynamic property** must be written like:

js

```
acc[curr] = something;
```

But you cannot assign to `[curr]` alone because it has no object on the left.

✓ Correct mutating version:

js

```
const nested = arr.reduceRight((acc, curr) => {
  const newObj = {};
  newObj[curr] = acc;
  return newObj;
}, {});
```

✨ Final Output (works):

js

```
{ a: { b: { c: {} } } }
```

Same as the very first version you wrote.

🔥 In one line: Why error?

Because `[curr] = acc` is treated as **array destructuring**, not as **setting an object's key**.

You must specify an object to assign the key into.

If you want, I can show all valid and invalid ways of writing dynamic keys.



```
const result = arr?.reduceRight((acc, crnt, idx) => {
  if(Object.keys(acc)?.length > 0) {
    acc[crnt] = {}
  } else {
    acc[crnt] = acc;
  }
});
```

```

return acc;

},{});

console.log("result:", result);
the above is giving this:

result: <ref *1> { c: [Circular *1], b: {}, a: {} }

why not the desired result

```

Vipul, **excellent question** — this is a classic mistake when building nested objects manually.

Your output:

```

css

result: <ref *1> { c: [Circular *1], b: {}, a: {} }

```

Means your code created a **circular reference**, not a nested object.

Let me explain why in the simplest way possible.

✖ Your code

```

js

if(Object.keys(acc)?.length > 0) {
    acc[crnt] = {}
} else {
    acc[crnt] = acc;
}

```

⚠ The real problem

The first iteration (when acc is {})

You do:

```

js

acc[crnt] = acc;

```

Which means:

You are assigning the object to itself.

Example:

If `acc = {}`
and `curr = "c"`

You do:

```
arduino
acc["c"] = acc;
```

This becomes:

```
makefile
acc = {
    c: acc (itself)
}
```

So it literally becomes:

```
vbnet
c: <ref to the whole acc object>
```

That is what “circular reference” means.

💥 Why the code NEVER produces nested structure

Because once you set:

```
arduino
acc["c"] = acc
```

Your accumulator becomes:

```
makefile
acc = {
    c: acc
}
```

And it stays WRONG forever.

🎯 Correct Approach

To build:

```
css
{ a: { b: { c: {} } } }
```

You must create a new object each time and set the previous accumulator inside the new one:

✓ Working version

```
js

const result = arr.reduceRight((acc, curr) => {
  return { [curr]: acc };
}, {});

console.log(result);
```

🎯 If you want to do it manually (step-by-step imperative)

```
js

const result = arr.reduceRight((acc, curr) => {
  const newObj = {};
  newObj[curr] = acc;
  return newObj;
}, {});
```

✓ FINAL OUTPUT

```
css

{
  a: {
    b: {
      c: {}
    }
  }
}
```

🔥 Why your logic fails (summary)

- `acc[curr] = acc` assigns the whole object **inside itself**
- This creates an infinite loop (circular reference)
- The object can never form nested layers
- That's why you got:

```
css

{ c: <Circular *1>, b: {}, a: {} }
```

Because the very first assignment polluted `acc`.

If you want, I can rewrite your original code to fix it without simplify, keeping your logic structure intact.



```
newObj[curr] = acc;
```

like this newObj holds the reference of acc obj or it is entirely copied to newObj means deep copied means if we changes something in acc than will it effect newObj data or not and at what level copy happens

Great question, Vipul — **this is the core of JavaScript behavior:**

👉 *Objects are assigned by reference, not by value.*

Let me explain clearly and practically.

🔥 Your code:

js

```
newObj[curr] = acc;
```

What does this do?

- ➡ It does **NOT deep copy** the accumulator.
- ➡ It does **NOT shallow copy** either.

It simply stores the **reference** of acc inside **newObj[curr]**.

Meaning:

csharp

```
newObj[curr] ----> same memory object as acc
```

🧪 Example to prove it

js

```
let acc = { x: 1 };
let newObj = [];

newObj["a"] = acc;
acc.x = 999;

console.log(newObj);
```

Output:

```
css
{ a: { x: 999 } }
```

Why?

Because both point to the **same object in memory**.

⚡ But in your nested reducer code, it still works

When building the nested structure:

```
js
return { [curr]: acc };
```

It works **because you never mutate acc after attaching it**.

You always do:

```
pgsql
acc = new outer object
```

You never modify the inner object directly.

So the references remain safe.

📌 Very Important: "At what level is the copy happening?"

✓ The outer object (newObj) is new each iteration

This part is a fresh object:

```
js
const newObj = {};
```

✗ The inner value (acc) is a reference, not a copy

This part is just a pointer:

```
js
newObj[curr] = acc;
```

So:

- The **wrapper** object is new each time

- The **nested inside (acc)** is a reference to the previous accumulator

Thus you get:

```
css
```

```
{ a: (ref) }
```

Where (ref) points to:

```
css
```

```
{ b: (ref) }
```

And so on.

😎 Visualization (Very Clear)

Say arr = ["a", "b", "c"]

Iteration 1:

```
kotlin
```

```
acc = {}
return { c: acc } // newObj1
```

Iteration 2:

```
kotlin
```

```
acc = { c: {} } // acc now points to newObj1
return { b: acc } // newObj2
```

Iteration 3:

```
kotlin
```

```
acc = { b: { c: {} } }
return { a: acc } // newObj3
```

You get nested structure because each level wraps the previous by reference.

🧠 Final Answer (short version)

✓ Does `newObj[curr] = acc` create a deep copy?

No. It stores a reference.

✓ If you change acc, does it also change inside `newObj[curr]`?

Yes, it will — because both point to the same object.

✓ At what level is copying done?

Only **one level**:

The outer `{ curr: acc }` is a **new object**, but `acc` inside it is **referenced**, not copied.

If you want, I can show you:

- deep copy
- shallow copy
- reference copy
with diagrams and examples.