What is the difference between expression and statement?

Expressions (eg ternary operator) -> This gets converted into a value after its execution and we can store it in a variable.

eg.

```
const ans = 5 > 4 ? `true` : `false`;
console.log(ans);
```

Statements (eg. if-else) -> This only runs line by line and it is not converted into a value.

```
if (5 > 4) {
  console.log(`true`);
} else {
  console.log(`false`);
}
```

Some more egs.

1. For loop (statement)
2. forEach loop (expression)
3. Switch case (statement)
4. Declaring variables (eg. let a = 0;) (statement)

i.e.

let b = 0; ☑
let a = let b = 0; ❌❌

5. Comparing cases

(eg. const ans = 5 > 4 => true)

(expression)

6. function call and function definition (expression)
7. Arithmetic expressions (eg. const sum = 5 + 4)
8. Logical Expressions

(eg. const ans = 5 && 6 => 6)

Some questions:

Q1 Will this work fine

```
const a = 6 > 7 ? "true" : "false";
console.log(a);
```

YES

```
if (6 > 7) {
  `true`;
} else {
  ("false");
}
```

Here we will see an answer but this is not the result of the expression, here the result `false` is due to the code line `false` rather than the result ans of (6 > 7).

Verify, store it in a variable and see for yourself.

```
const ans = if(6 > 7) {
    `true`
} else{
    'false'
}

console.log(ans)
// we get undefined as output
```

Hence due to this difference; in JSX React, we use expressions rather than statements, to get the output values in a variable.

If statements are absolutely needed, use functions (as they are expressions), write your statement code inside of it and then return and call the functions to make your statements work.

Or we can use IIFEs, arrow functions etc...

What is an IIFE in JavaScript? | JavaScript for ProCodrrs

IIFE -> Immediately Invoked Function Expression

---

eg.

```
(function () {
  console.log("Hi");
})();
```

Semicolons are a must in IIFE, use it before IIFEs or after the statement that is before an IIFE. Also add one after an IIFE ends.

Else the compiler is confused as to when an IIFE is starting or stopping.

```
(function () {
  console.log("Hi");
})();
```

or

```
const a = 5;

(function () {
  console.log("Hi");
})();
```

or

```
const a = 5;
(function () {
  console.log("Hi");
})();
```

We know that we need to pollute our global scope as less as possible. So we can write our code inside IIFEs to do this. If there are tasks or variables with names which have the same names as some packages or libraries then use IIFE to separate them and save the global environment.

We need not name IIFEs, as :

- IIFE comes
- IIFE executes once
- IIFE goes and never comes back.

So we do not use IIFEs again and again like normal functions.

eg.

```

```

```
(function () {
  const h1 = document.querySelector("hi");
  const p = document.querySelector("p");
  const num = 120;

  h1.style.backgroundColor = "pink";
  h1.style.color = "black";
  p.innerText = num;
})();
```

We can also use async in IIFEs

eg.

```
(async function () {
  const h1 = document.querySelector("hi");
  const p = document.querySelector("p");
  const num = 120;

  h1.style.backgroundColor = "pink";
  h1.style.color = "black";
  p.innerText = num;
})();
```

Different Ways to Create IIFEs:

1.

```
(function () {
  console.log("Hi");
})();
```

2.

```
(function () {
  console.log("Hi");
})();
```

3. Arrow Functions

```
(() => {
  console.log("Hi");
})();
```

4.

```
+(function () {
  console.log("Hi");
})();
```

5.

```
-(function () {
  console.log("Hi");
})();
```

`function` keyword means function declaration i.e. it is treated in as a statement. So when we use `+` or `-`, then that function is treated in as an expression.

6. Normal Expression

```
const a = (function () {
  console.log("Hi");
})();
```

7. Using `NOT` Operator

```
!(function () {
  console.log("Hi");
})();
```

8. Using Bitwise Operator

```
~(function () {
  console.log("Hi");
})();
```

9. Using void

```
void (function () {
  console.log("Hi");
})();
```

10. Using new

```
new (function () {
  console.log("Hi");
})();
```

11. using an extra set of () on functions

```
(function () {
  console.log("Hi");
})();
```

12. Making IIFE function an expression via `true &&`

```
true &&
  (function () {
    console.log("Hi");
  })();
```

13. `false ||` (same logic as `true &&`)

```
false ||
  (function () {
    console.log("Hi");
  })();
```

14. Using ternary operators to make expressions

```
true
  ? (function () {
      console.log("Hi");
    })()
  : "";
// a falsy value ''
```

15. Using `try and catch`

```
try {
  throw function () {
    console.log("Hi");
  };
} catch (e) {
  e();
}
```

Syntax is a little bit different here. `This is an IIF not an IIFE`

Using continue keyword with loops

`Continue` is used in loops to skip the current loop iteration and go on to the next one.

```
let text = "";

for (let i = 0; i < 10; i++) {
  if (i === 3) {
    continue;
  }
  text = text + i;
}

console.log(text);
// Expected output: "012456789"
```

In contrast to the break statement, continue does not terminate the execution of the loop entirely, but instead:

- In a while or do...while loop, it jumps back to the condition.
- In a for loop, it jumps to the update expression.
- In a for...in, for...of, or for await...of loop, it jumps to the next iteration.

Docs - VVIMP Click me

egs.

```
let i = 0;
let n = 0;

while (i < 5) {
  i++;

  if (i === 3) {
    continue;
  }

  n += i;
  console.log(`n: ${n}`);
}

console.log(`i after loop is over: ${i}`);
```

Q. Given is an array of numbers. Use `continue` here and `no else statement` to multiply the odd numbers by 2 and print them and print the even numbers as it is,

```
const numbers = [1, 2, 3, 4, 5];

for (let num of numbers) {
  if (num % 2 === 0) {
    console.log(num);
    continue;
  }

  console.log(num * 2);
}
```

---

for of vs for in Loop in JavaScript

`For of` loop

This loop is used to iterate over the iterable objects like arrays, strings, maps etc...

Using a for of loop to print the elements in the array

```
const fruits = ["apple", "banana", "mango", "peach", "guava"];

for (const fruit of fruits) {
  console.log(fruit);
}

/*
apple
banana
mango
peach
guava
*/
```

We can use let, const or var in this but it is preferred to use let or const.

We can do this also via a simple for loop:

```
const fruits = ["apple", "banana", "mango", "peach", "guava"];

for (let i = 0; i < fruits.length; i++) {
  console.log(fruits[i]);
```

```
}

/*
apple
banana
mango
peach
guava
*/
```

Using a for of loop to print the chars of a string

```
const letters = "Anurag";

for (const letter of letters) {
  console.log(letter);
}

/*
A
n
u
r
a
g
*/
```

Using for of in objects

We will get an error here as normal objects are not iterable objects.

So for this we will use a for in loop

For in loop

For in + normal objects

Gives us key in output

```
const persons = {
  firstName: "Anutar",
  lastName: "Sen",
  age: 30,
  male: true,
};

for (const key in persons) {
  console.log(`${key}`);
}
```

```
/*
firstName
lastName
age
male
*/
```

Gives us value in output

```
const persons = {
  firstName: "Anutar",
  lastName: "Sen",
  age: 30,
  male: true,
};

for (const key in persons) {
  console.log(persons[key]);
}

/*
Anutar
Sen
30
true
*/
```

Check this when we used `.` notation to print the values

```
const persons = {
  firstName: "Anutar",
  lastName: "Sen",
  age: 30,
  male: true,
};

for (const key in persons) {
  console.log(persons.key);
}

/*
undefined
undefined
undefined
undefined
*/
```

We get `undefined` here because we are trying to access a key named `key` in the persons object, since there is no such key named `key`, we will get undefined as output.

Writing key and value together using `for-in`

```
const persons = {
  firstName: "Anutar",
  lastName: "Sen",
  age: 30,
  male: true,
};

for (const key in persons) {
  console.log(key, ":", persons[key]);
}

/*
firstName : Anutar
lastName : Sen
age : 30
male : true
*/
```

For-in is a little bit heavy loop (less efficient) so use it as less as possible.

Alternative for objects printing

- Use Object.keys(objectName)
- We get an array of keys
- Use a normal for of loop now

```
const persons = {
  firstName: "Anutar",
  lastName: "Sen",
  age: 30,
  male: true,
};

const keys = Object.keys(persons);

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

/*
firstName
lastName
age
```

```
male
*/
```

To get values now

```
const persons = {
  firstName: "Anutar",
  lastName: "Sen",
  age: 30,
  male: true,
};

const values = Object.values(persons);

for (const value of values) {
  console.log(value);
}

/*
Anutar
Sen
30
true
*/
```

Alternative

```
const persons = {
  firstName: "Anutar",
  lastName: "Sen",
  age: 30,
  male: true,
};

const keys = Object.keys(persons);

for (const key of keys) {
  console.log(persons[key]);
}

/*
Anutar
Sen
30
true
*/
```

If we use `for in` with arrays, then we will get indices, as key of each array element is its index.

To get an array of arrays where each array has [key, value] do: `Object.entries(objectName)`

```
const persons = {
  firstName: "Anutar",
  lastName: "Sen",
  age: 30,
  male: true,
};

const entries = Object.entries(persons);

for (const entry of entries) {
  console.log(entry);
}

/*
[ 'firstName', 'Anutar' ]
[ 'lastName', 'Sen' ]
[ 'age', 30 ]
[ 'male', true ]
*/
```

---

forEach Array Method in JavaScript

This is a higher order function but it works in the same way as for of loop

```
const fruits = ["banana", "apple", "peach", "mango", "grapes"];

fruits.forEach((fruit) => console.log(fruit));

// or Simple function
console.log(`**************************`);

fruits.forEach(function (fruit) {
  console.log(fruit);
});
```

forEach will always return `undefined` at the end even if we put the `return` keyword. This is because we are writing return inside the callback function instead of the regular higher order function.

```
const fruits = ["banana", "apple", "peach", "mango", "grapes"];

fruits.forEach(function (fruit) {
```

```
  console.log(fruit.toUpperCase());


  return fruit.toUpperCase();
});

/*
VM959:4 BANANA
VM959:4 APPLE
VM959:4 PEACH
VM959:4 MANGO
VM959:4 GRAPES
undefined
/*
```

But if we use `map` then we will get an array of looped values as the return value.

```
const fruits = ["banana", "apple", "peach", "mango", "grapes"];

fruits.map(function (fruit) {
  console.log(fruit.toUpperCase());

  return fruit.toUpperCase();
});

/*
VM1432:4 BANANA
VM1432:4 APPLE
VM1432:4 PEACH
VM1432:4 MANGO
VM1432:4 GRAPES
(5) ['BANANA', 'APPLE', 'PEACH', 'MANGO', 'GRAPES']
*/
```

Using break keyword with loops

Used to exit the loop.

Q. Find first index of 9 (Donot use indexOf() array method)

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 9, 7];

const target = 9;

let indexValue = -1;

for (let i = 0; i < numbers.length; i++) {
```

```
    if (numbers[i] === target) {
      indexValue = i;
      break;
    }
  }

  console.log(indexValue);
```

Break enhances the loop performances.

break is used with

- for
- for of
- for in
- while
- do while

But `break` is `not used with forEach()` as it is an array method and a higher order function though it acts like a loop.

Also `break` is `not used with map, filter and reduced` as these are higher order functions.

Any code written below `break` statement will not be executed.

Map, Filter, Reduce in JavaScript

Map

This method also works like a loop but it will return an array of looped items as the return value.

```
const months = ["January", "February", "March", "April"];

months.map((month) => {
  console.log(month);
});
/*
VM68:4 January
VM68:4 February
VM68:4 March
VM68:4 April
(4) [undefined, undefined, undefined, undefined]
*/
```

Here we did not return any value

When we return the items via `return`

```
const months = ["January", "February", "March", "April"];

months.map((month) => {
  console.log(month);
  return month;
});

/*
VM305:4 January
VM305:4 February
VM305:4 March
VM305:4 April
(4) ['January', 'February', 'March', 'April']
*/
```

`Map` is a `non-destructive method`, i.e. the array it returns is not the modified original array but a copy of the original array, this copy is then modified.

eg.

```
const months = ["January", "February", "March", "April"];

const upperCaseMonths = months.map((month) => {
  console.log(month);
  return month.toUpperCase();
});

// VM1997:4 January
// VM1997:4 February
// VM1997:4 March
// VM1997:4 April
// undefined

upperCaseMonths;
// (4) ['JANUARY', 'FEBRUARY', 'MARCH', 'APRIL']

months;
// (4) ['January', 'February', 'March', 'April']
```

Here we see that though we made changes in months via map, the original months array is unchanged, hence proving the above statement right.

We can also write like this:

```
const months = ["January", "February", "March", "April", "May"];
```

```
console.log(
  months.map((month) => {
    console.log(month);
    return month.toUpperCase();
  })
);

/*
January
February
March
April
May
[ 'JANUARY', 'FEBRUARY', 'MARCH', 'APRIL', 'MAY' ]
*/
```

Indexing in JS map

Add another parameter in map() to get the index

```
const months = ["January", "February", "March", "April", "May"];

console.log(
  months.map((month, index) => {
    console.log(`${index} : ${month}`);
    return month.toUpperCase();
  })
);

/*
0 : January
1 : February
2 : March
3 : April
4 : May
[ 'JANUARY', 'FEBRUARY', 'MARCH', 'APRIL', 'MAY' ]
*/
```

Here numbering / indexing is starting from 0

Start numbering from 1

```
const months = ["January", "February", "March", "April", "May"];

console.log(
  months.map((month, index) => {
    console.log(`${index + 1} : ${month}`);
    return month.toUpperCase();
  })
```

```
);

/*
1 : January
2 : February
3 : March
4 : April
5 : May
[ 'JANUARY', 'FEBRUARY', 'MARCH', 'APRIL', 'MAY' ]
*/
```

Adding a third parameter in map - The whole array

```
const months = ["January", "February", "March", "April", "May"];

const capitalMonths = months.map((month, index, array) => {
  console.log(`${index + 1} : ${month}`);
  console.log(`Here is the array over which we map: ${array}`);
  return month.toUpperCase();
});

/*
1 : January
Here is the array over which we map: January,February,March,April,May

2 : February
Here is the array over which we map: January,February,March,April,May

3 : March
Here is the array over which we map: January,February,March,April,May

4 : April
Here is the array over which we map: January,February,March,April,May

5 : May
Here is the array over which we map: January,February,March,April,May
*/
```

Filter

Say we are mapping over an array using filter, then in this case:

Filter is used to take out the elements from the original array, copy those and put it in a new array. So the size of this new array may be equal to or less than the original array.

See this eg.

```
const months = ["January", "February", "March", "April", "May"];
```

```
const filterMonths = months.filter((month) => {
  return month.length <= 5;
});

console.log(filterMonths);

/*
[ 'March', 'April', 'May' ]
*/
```

Here we put a condition in the return statement and only those values from the array will be returned which make this condition `true`.

Here we are returning only those month values whose length of chars is less than or equal to 5

eg2

```
const months = ["January", "February", "March", "April", "May"];

const filterMonths = months.filter((month) => {
  return month.length >= 5;
});

console.log(filterMonths);

/*
[ 'January', 'February', 'March', 'April' ]
*/
```

eg 3 Return those months only which have an 'M' in it

```
const months = ["January", "February", "March", "April", "May", "December"];

const filterMonths = months.filter((month) => {
  const lower = month.toLowerCase();
  return lower.includes("m");
});

console.log(filterMonths);

/*
[ 'March', 'May', 'December' ]
*/
```

Filtering using the index variable

```

```

```
const months = ["January", "February", "March", "April", "May", "December"];

const filterMonths = months.filter((month, index) => {
  return index >= 5;
});

console.log(filterMonths);

// Here we will get [ 'December' ] only
```

Now index of Jan = 0, Feb = 1, ..., May = 4, December = 5

so it will return December only as only its index is equal to or more than 5.

```
const students = [
  {
    name: "Ajay",
    age: 20,
  },
  {
    name: "Shankar",
    age: 10,
  },
  {
    name: "Bjio",
    age: 55,
  },
  {
    name: "Geet",
    age: 5,
  },
  {
    name: "gogo",
    age: 70,
  },
];

const filtering = students.filter((student) => {
  return student.age >= 18;
});

console.log(filtering);

/*
[
  { name: 'Ajay', age: 20 },
  { name: 'Bjio', age: 55 },
  { name: 'gogo', age: 70 }
]
*/
```

```
// Now we pick only names from these objects using a map

const namesArr = filtering.map((object) => {
  return object.name;
});

console.log(namesArr);

/*
[ 'Ajay', 'Bjio', 'gogo' ]
*/
```

Another way: (Chaining of Array Methods)

```
const students = [
  {
    name: "Ajay",
    age: 20,
  },
  {
    name: "Shankar",
    age: 10,
  },
  {
    name: "Bjio",
    age: 55,
  },
  {
    name: "Geet",
    age: 5,
  },
  {
    name: "gogo",
    age: 70,
  },
];

const filtering = students
  .filter((student) => {
    return student.age >= 18;
  })
  .map((student) => {
    return student.name;
  });

console.log(filtering);

/*
[ 'Ajay', 'Bjio', 'gogo' ]

*/
```

Another Chaining Example

```
const students = [
  {
    name: "Ajay",
    age: 20,
  },
  {
    name: "Shankar",
    age: 10,
  },
  {
    name: "Bjio",
    age: 55,
  },
  {
    name: "Geet",
    age: 5,
  },
  {
    name: "gogo",
    age: 70,
  },
];

const filtering = students
  .filter((student) => {
    return student.age >= 18;
  })
  .map((student) => {
    return student.name;
  })
  .filter((student) => {
    return student.toLowerCase().includes("o");
  });

console.log(filtering);

/*
[ 'Bjio', 'gogo' ]
*/
```

Reduce

Array has many values, so `reduce` does the job of turning those many values into one single value.

Syntax

```
reduce((accumulator, currentValue, index, array) => {
    return accumulator
}, initial Value);
```

Printing when no initial value is mentioned

- current value = 1st index
- accumulator = 0th index

```
const nums = [1,2,3,4,5,6];

const ans = nums.reduce((accumulator_or_previousValue, currentValue, index, array)
=> {
    return accumulator_or_previousValue + currentValue;
});

console.log(ans)

// 21 (sum of all elements of the array)

1 -> acc = 1 curr = 2, return 1 + 2
2 -> acc = 1 + 2 = 3, curr = 3, return 3 + 3
3 -> acc = 3 + 3 = 6, curr = 4, return 6 + 4
4 -> acc = 6 + 4 = 10, curr = 5, return 10 + 5
5 -> acc = 10 + 5 = 15, curr = 6, return 15 + 6
6 -> loop over (ans = acc = 15 + 6 = 21)
```

Printing when initial value is mentioned

- current value = 0th index
- accumulator = initial value

```
const nums = [1,2,3,4,5,6];


const ans = nums.reduce((accumulator_or_previousValue, currentValue, index, array)
=> {
  return accumulator_or_previousValue + currentValue;
}, 'x'); // we can give any value here as initial value, we can also give strings
here
// This 'x' is called INITIAL VALUE


console.log(ans);


0 -> acc = 'x', curr = 1, return 'x' + 1 (initial value or accumulator + current
```

```
value)
1 -> acc = 'x' + 1 = x1, curr = 2, return x1 + 2
2 -> acc = x1 + 2 = x12, curr = 3, return x12 + 3
3 -> acc = x12 + 3 = x123, curr = 4, return x123 + 4
4 -> acc = x123 + 4 = x12234, curr = 5, return x1234 + 5
5 -> acc = x1234 + 5 = x12345, curr = 6, return x12345 + 6
6 -> array over,  hence,  ans = accumulator = x12345 + 6 = x123456
```

As initial value is a string here, we see string concatenation when we tried to add acc and curr here

Return value is needed in reduce to get something, else we will get:

```
x
undefined
undefined
undefined
undefined
undefined
```

Return value will go to accumulator for next loop iteration

Result of reduce function => accumulated value

Some questions on reduce

Q1 Given an array of strings, count the number of times each string appears and return an object with the string as the key and the count as the value.

const fruits = ['apple', 'banana', 'apple', 'orange', 'banana', 'apple'];

Expected Output: { apple: 3, banana: 2, orange: 1 }

```
const fruits = ["apple", "banana", "apple", "orange", "banana", "apple"];

const ans = fruits.reduce((acc, curr) => {
  acc[curr] = (acc[curr] || 0) + 1;
  return acc;
}, {});

console.log(ans);
```

Here we are taking accumulator as an empty object and at each iteration, we are checking if the key exists in the object or not, so we use (acc[curr] || 0) to designate a key when there is no key and do + 1 to update the key's value from 0 to 1.

Problem Recap

We have an array of strings, and our goal is to count how many times each string appears in that array. The final result should be an object where:

- Keys are the unique strings from the array.
- Values are the number of times each string (key) appears in the array.

Code Walkthrough

1. The reduce Function

The reduce() method is a powerful array function in JavaScript. It allows us to iterate over an array and reduce it to a single value by applying a function on each element, one after the other.

The basic syntax for reduce looks like this:

```
array.reduce((accumulator, currentValue) => {
  /* logic */
}, initialValue);
```

- accumulator: This is the value that is accumulated during the iterations. It's where we will store the result—in this case, an object that keeps track of the counts of the strings.
- currentValue: This is the current element from the array that is being processed.
- initialValue: The initial value for the accumulator, which in our case is an empty object ({}), as we want to build an object that maps strings to counts.

2. Code Breakdown

```
const fruits = ["apple", "banana", "apple", "orange", "banana", "apple"];

const fruitCount = fruits.reduce((acc, fruit) => {
  acc[fruit] = (acc[fruit] || 0) + 1;
  return acc;
}, {});
```

Let’s go through this line-by-line:

1. Line 1: We have an array of strings called fruits which contains repeated values.
2. Line 2: The reduce function is called on the fruits array. The goal is to accumulate counts of each fruit.
3. Line 3: We use the acc[fruit] = (acc[fruit] || 0) + 1 expression:
    - acc[fruit] checks whether the fruit is already in the acc object (which is initially an empty object {}).
    - acc[fruit] || 0 checks if the current fruit has been encountered before. If it has, it returns the existing count. If it hasn't, it returns 0.
    - We then add 1 to this value, effectively counting the current instance of the fruit.

4. Line 4: We return the updated accumulator acc. The reduce function keeps passing this updated acc to the next iteration until all elements of the array have been processed.
5. Line 5: The initial value of acc is {}, an empty object, which is crucial because we want to build the object that contains the counts.

3. Example Walkthrough

Let's walk through the execution step-by-step with the input array ['apple', 'banana', 'apple', 'orange', 'banana', 'apple']:

```
First Iteration (fruit = 'apple'):

acc = {} (empty object).
acc['apple'] = (undefined || 0) + 1 = 1.
acc becomes {'apple': 1}.
```

```
Second Iteration (fruit = 'banana'):

acc = {'apple': 1}.
acc['banana'] = (undefined || 0) + 1 = 1.
acc becomes {'apple': 1, 'banana': 1}.
```

```
Third Iteration (fruit = 'apple'):

acc = {'apple': 1, 'banana': 1}.
acc['apple'] = (1 || 0) + 1 = 2.
acc becomes {'apple': 2, 'banana': 1}.
```

```
Fourth Iteration (fruit = 'orange'):

acc = {'apple': 2, 'banana': 1}.
acc['orange'] = (undefined || 0) + 1 = 1.
acc becomes {'apple': 2, 'banana': 1, 'orange': 1}.
```

```
Fifth Iteration (fruit = 'banana'):

acc = {'apple': 2, 'banana': 1, 'orange': 1}.
acc['banana'] = (1 || 0) + 1 = 2.
acc becomes {'apple': 2, 'banana': 2, 'orange': 1}.
```

```
Sixth Iteration (fruit = 'apple'):

acc = {'apple': 2, 'banana': 2, 'orange': 1}.
acc['apple'] = (2 || 0) + 1 = 3.
acc becomes {'apple': 3, 'banana': 2, 'orange': 1}.
```

4. Final Output

After the final iteration, reduce returns the accumulated object:

```
{ apple: 3, banana: 2, orange: 1 }
```

This is exactly what we expect—each fruit's count is stored as the value in the object.

Key Concepts Recap

- reduce Function: A flexible way to transform arrays into a single value by iterating through each element and accumulating a result.
- Object as Accumulator: We use an object (acc) to keep track of the counts. For each string in the array, we either initialize its count to 1 or increment it if it's already in the object.
- Short-circuiting with ||: The expression (acc[fruit] || 0) ensures that if a key does not exist in the object yet, it will default to 0 before adding 1 to it. This approach is efficient and clean for counting occurrences in an array!

Q2 Given an array of objects representing products with a price property, calculate the total cost of all products after applying a 10% discount.

```
const products = [
  { name: "Product 1", price: 100 },

  { name: "Product 2", price: 200 },

  { name: "Product 3", price: 300 },
];

const ans = products.reduce((acc, curr) => {
  curr.price = curr.price - 0.1 * curr.price;
  return curr.price + acc;
}, 0);

console.log(ans);

// Expected Output: 540
```

```
// ans = 540
```

Q3 Given three arrays, names, ages, and cities, write a program to combine them into one array of objects where each object contains a name, age, and city property.

const names = ['Alice', 'Bob', 'Charlie'];

const ages = [25, 30, 35];

const cities = ['New York', 'Los Angeles', 'Chicago'];

```
// Expected Output: [
// { name: 'Alice', age: 25, city: 'New York' },
// { name: 'Bob', age: 30, city: 'Los Angeles' },
// { name: 'Charlie', age: 35, city: 'Chicago' }
// ]
```

Approach 1 -> using map()

```
const names = ["Alice", "Bob", "Charlie"];

const ages = [25, 30, 35];

const cities = ["New York", "Los Angeles", "Chicago"];

const ans = names.map((name, index) => {
  return {
    name: name,
    age: ages[index],
    city: cities[index],
  };
});

console.log(ans);
```

Approach 2 (using reduce())

```
const names = ["Alice", "Bob", "Charlie"];

const ages = [25, 30, 35];

const cities = ["New York", "Los Angeles", "Chicago"];

const ans2 = names.reduce((acc, curr, index) => {
  acc.push({
    name: curr,
    age: ages[index],
    city: cities[index],
  });

  return acc;
}, []);

console.log(ans2);

/*
[
  { name: 'Alice', age: 25, city: 'New York' },
  { name: 'Bob', age: 30, city: 'Los Angeles' },
  { name: 'Charlie', age: 35, city: 'Chicago' }
]
*/
```

Some and Every Array Method in JavaScript

some() method in JavaScript

This helps us to loop over an array just like map, filter, forEach but with a twist.

For each iteration it checks if the iterative value is truthy or falsy.

If even one value is truthy, the final return value will be truthy, else it is falsy.

```
const evenNumbers = [0, 2, 4, 6, 8];

const result = evenNumbers.some((num) => {
  return num > 4;
});

console.log(result);

// true
```

Here for cases 0 and 2, the ans is false but for 4, 6, 8 values, the answer is true. So we get a true as return statement.

eg2

```
const evenNumbers = ["", 0, null];

const result = evenNumbers.some((num) => {
  return num > 4;
});

console.log(result);

// false
```

All are falsy values here, and all cases are false, hence output is a falsy value.

eg3

```
const evenNumbers = [0, 2, 4, 6, 8];

const result = evenNumbers.some((num) => {
  return num > 14;
});

console.log(result);

// false
```

No case is true here, hence ans is false

If first case is true, it will not check for the remaining cases after it (just like logical operators).

Application -> We have a large, very large array of even numbers, and it may/may not contain some odd numbers. So we need to check if the array has all even numbers or not. For this, we can use some().

If true, array is even, else array is not even

eg

```
const evenNumbers = [0, 2, 4, 6, 8];

const result = evenNumbers.some((num) => {
  return !(num % 2 == 0);
  // checking if there are odd numbers in the array
});

console.log(`${result} means the array is not odd, hence the array is even`);

// false means the array is not odd, hence the array is even
```

Returning the index of the odd number

```
const evenNumbers = [0, 2, 4, 6, 8, 5, 10];

const result = evenNumbers.some((num, index) => {
  if (num % 2 === 1) {
    console.log(`Number is odd at index ${index}`);
  }
  return !(num % 2 == 0);
  // checking if there are odd numbers in the array
});

console.log(result);

// Number is odd at index 5
// true means array is not even
```

`every()` method

This method will loop through an array and even if one value is false, it will return false, if all are true then only it will return true

```
const evenNumbers = [0, 2, 4, 6, 8, 5, 10];

const isEven = evenNumbers.every((num, index) => {
  if (num % 2 === 1) {
    console.log(`Number is odd at index ${index}`);
  }
  return num % 2 === 0;
  // checking if there are odd numbers in the array
});

console.log(isEven);

// Number is odd at index 5
// false means array is not even
```

Arguments Keyword in JavaScript

This keyword is found inside every function by default, except arrow functions.

```
function add(a, b) {
  console.log(arguments);
  return a + b;
```

```
}

add(1, 7);
// VM362:2 Arguments(2) [1, 7, callee: ƒ, Symbol(Symbol.iterator): ƒ]
// 8
```

`Arguments(2) [1, 7, callee: ƒ, Symbol(Symbol.iterator): ƒ]`

We get an Array like object when we console.log, arguments. This is not an array as we cannot use array methods like forEach, map etc. with this.

expanded view

```
Arguments(2) [1, 7, callee: ƒ, Symbol(Symbol.iterator): ƒ]
0
:
1

1
:
7

callee
:
ƒ add(a, b)

length
:
2

Symbol(Symbol.iterator)
:
ƒ values()

[[Prototype]]
:
Object
```

Now let us do this,

```
function add(a, b) {
  console.log(arguments);
  return a + b;
}

add(1, 7, 11, 8, 2, 5);

// VM1045:2 Arguments(6) [1, 7, 11, 8, 2, 5, callee: ƒ, Symbol(Symbol.iterator):
```

```
ƒ]0: 11: 72: 113: 84: 25: 5callee: ƒ add(a, b)length: 6Symbol(Symbol.iterator): ƒ
values()[[Prototype]]: Object
// 8 (sum)
```

So here we are using only the first 2 arguments from the call function, but we are giving it more than 2, so these extra arguments are stored in the Argument object as seen above.

Though we cannot use Array methods with the argument object but we can use normal `for loop` to access its inner argument elements.

eg.

```
VM1045:2 Arguments(6) [1, 7, 11, 8, 2, 5, callee: ƒ, Symbol(Symbol.iterator): ƒ]0:
11: 72: 113: 84: 25: 5callee: ƒ add(a, b)length: 6Symbol(Symbol.iterator): ƒ
values()[[Prototype]]: Object

Arguments[0] // 1
Arguments[1] // 7
Arguments[2] // 11
```

Eg, return the sum of all argument object elements

```
function argumentsSum() {
  let sum = 0;
  for (let i = 0; i < arguments.length; i++) {
    sum += arguments[i];
  }

  return sum;
}

console.log(argumentsSum(1, 2, 3, 4, 5, 6, 7, 8, 9));

// 45
```

We can pass `n` number of arguments here.

Arguments work with function declaration (as seen above). It also works with function expressions (see below)

```
const add = function () {
  let sum = 0;
  for (let i = 0; i < arguments.length; i++) {
    sum += arguments[i];
  }

  return sum;
};
```

```
console.log(add(1, 2, 3, 4, 5, 6, 7, 8, 9, 10));

// 55
```

Used anonymous function here.

Now as mentioned before, we cannot use `arguments` with arrow functions as arrow functions came with ES6.

So for arrow functions, and the modern way, we use `rest` operator (...). We will see this later on.

Ways to convert argument object into arrays

```
const args = Array.prototype.slice.call(arguments);
// or
const args = Array.from(arguments);
// or
const args = [...arguments];
```

Now we solve the above question using an array method i.e. reduce()

```
const add = function () {
  const argArr = Array.from(arguments); // made into array

  const sum = argArr.reduce((acc, num) => {
    return acc + num;
  }, 0);

  return sum; // Stored the arguments sum result via reduce into a sum variable
};

console.log(add(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11));
```

[ES6 Features Default Parameters in JavaScript]

Default Parameters in JavaScript

`Default Parameter` in JS came with ES6, this allows us to enter a default value to a parameter whenever its argument is undefined or that parameter does not have an argument at all.

See these examples

```
function multiply(a, b) {
  return a * b;
}

console.log(multiply(2, 5));
// 10
```

Works fine

eg2

```
function multiply(a, b) {
  return a * b;
}

console.log(multiply(2));
// NaN
```

Now in eg2, a = 2, b = ? No argument value for b

Hence js considers b as `undefined`.

Hence `2 * undefined = NaN` and we get NaN as output

To prevent this, we can give `b a default value`

eg

```
function multiply(a, b = 1) {
  return a * b;
}

console.log(multiply(2));

// 2
```

Now a = 2, b = 1 (takes the default value in function as no value is given in argument), hence result is 2 * 1 = 2

If we gave b, some value then, it would override the default value and we will get normal results like eg1

Before ES6, i.e. before default parameter, we used to handle default cases as follows:

eg4

```
function multiply(a, b) {
  if (b === undefined) {
```

```
    b = 1;
  }
  return a * b;
}

console.log(multiply(2));
// 2
```

What if we gave falsy values as arguments and used default??

```
function multiply(a, b = 1) {
  return a * b;
}

console.log(multiply(2, ""));
// 0
```

Here a = 2, b = '' (overriding the default value)

Now '' is falsy so js converts '' to 0, hence 2 * 0 = 0

What if we gave default value a falsy argument??

```
function multiply(a, b = null) {
  return a * b;
}

console.log(multiply(2));
// 0
```

Here we gave default value a falsy argument, so a _ b = 2 _ null = 2 * 0 = 0

Application of Default

```
const rollNDie = function (dieSides = 6) {
  const roll = Math.floor(Math.random() * dieSides + 1);

  return roll;
};

console.log(rollNDie(9));

// console.log(rollNDie()); -> Default case
```

Here we made a function which gives us the output of a die roll, the die can have N sides, N = no. passed in as arguments. If no argument is passed, we gave a default value of 6.

- With arguments (no default) (We get a number from 1 to N)
- Without arguments (default = 6) (We get a number from 1 to 6)

Q. Write a JavaScript function called sendMessage that returns a message. Set the default value of name to "Myself" and the default value of msg to "Good Morning".

```
function sendMessage(name = "Myself", msg = "GoodMorning") {
  return `${name}, ${msg}`;
}

console.log(sendMessage(`Hritik`, `You are so tall!`));
console.log(sendMessage());

// Hritik, You are so tall! -> When we pass arguments
// Myself, GoodMorning -> When no arguments are passed and default values are
activated.
```

ES6 - Spread Operator

This is used to combine arrays and objects (...)

Combining Arrays

Earlier we used the concat() method

```
const a1 = [1, 2, 3, 4, 5];
const a2 = [6, 7, 8, 9, 0];

const finalArr = a1.concat(a2);

console.log(finalArr);

/* [
  1, 2, 3, 4, 5,
  6, 7, 8, 9, 0
]
*/
```

Now we can use the spread operator

```
const a1 = [1, 2, 3, 4, 5];
const a2 = [6, 7, 8, 9, 0];

const finalArr = [...a1, ...a2]; // an array here having all values of a1 and a2
together via ... (spread operator)

console.log(finalArr);

/*
[
  1, 2, 3, 4, 5,
  6, 7, 8, 9, 0
]
*/
```

Combining And Copying Objects

1. Combining Objects

Keep the keys of the object different while combining them, if they keys are same they will get copied and updated and not combined.

```
const u1 = {
  name: "KSD",
  age: 18,
};

const u2 = {
  name2: "Shukla",
  age2: 78,
};

const finalObj = { ...u1, ...u2 };

console.log(finalObj);

// { name: 'KSD', age: 18, name2: 'Shukla', age2: 78 }
```

2. Copying Objects

We use ... (spread operator) to copy objects.

The ... operator uses shallow copy so it will not copy the nested objects that effectively.

eg.

```
const u1 = {
  name: "KSD",
```

```
  age: 18,
};

const u2 = {
  name: "Shukla",
  age: 78,
};

const finalObj = { ...u2, number: 45679 };

console.log(finalObj);

// { name: 'Shukla', age: 78, number: 45679 }

console.log(u2);
// { name: 'Shukla', age: 78 }
```

Here final object has copied u2 perfectly and added an additional property of number.

We see u2 is still the same, i.e. unchanged

Spread Operators with Strings

```
const str = "Hello";

const charArr = [...str];

console.log(charArr);

// [ 'H', 'e', 'l', 'l', 'o' ]
```

Spread Operator with Functions

```
const arr1 = [1, 2, 3, 4, 5];
const arr2 = [6, 7, 8, 9, 10];

const ans = function () {
  let sum = 0;
  for (let i = 0; i < arguments.length; i++) {
    sum += arguments[i];
  }

  console.log(arguments);
  return sum;
};

console.log(ans(...arr1, 11, 12, ...arr2));

/*
```

```
[Arguments] {

// from arr1
  '0': 1,
  '1': 2,
  '2': 3,
  '3': 4,
  '4': 5,

  // manually given
  '5': 11,
  '6': 12,

  // from arr2
  '7': 6,
  '8': 7,
  '9': 8,
  '10': 9,
  '11': 10
}

// final result
78
*/
```

eg. 2

```
const arr1 = [1, 2, 3, 4, 5];
const arr2 = "Kaustubhya";

const ans = function () {
  let sum = 0;
  for (let i = 0; i < arguments.length; i++) {
    sum += arguments[i];
  }

  console.log(arguments);
  return sum;
};

console.log(ans(...arr1, 11, 12, ...arr2));

/*
[Arguments] {
  '0': 1,
  '1': 2,
  '2': 3,
  '3': 4,
  '4': 5,
  '5': 11,
  '6': 12,
  '7': 'K',
```

```
  '8': 'a',
  '9': 'u',
  '10': 's',
  '11': 't',
  '12': 'u',
  '13': 'b',
  '14': 'h',
  '15': 'y',
  '16': 'a'
}
38Kaustubhya

*/
```

eg

```
const arr1 = [1, 2, 3, 4, 5];
const arr2 = "Kaustubhya";

const ans = function () {
  let sum = 0;
  for (let i = 0; i < arguments.length; i++) {
    sum += arguments[i];
  }

  console.log(arguments);
  return sum;
};

console.log(ans(...arr1, ...arr2, 11, 12));

/*
[Arguments] {
  '0': 1,
  '1': 2,
  '2': 3,
  '3': 4,
  '4': 5,
  '5': 'K',
  '6': 'a',
  '7': 'u',
  '8': 's',
  '9': 't',
  '10': 'u',
  '11': 'b',
  '12': 'h',
  '13': 'y',
  '14': 'a',
  '15': 11,
  '16': 12
}
```

```
15Kaustubhya1112
*/
```

Notice where string concatenation and addition are happening in eg2 and eg3

Rest Parameters in JavaScript

Rest Parameters AKA Rest Params are used when we are defining the function.

`Do not call it Rest Operator`

In earlier examples we used the argument keyword and passed some arguments and got its sum.

Here in `Rest Parameters`, we can pass this in function definition and get the answers

see this eg

```
function add(...nums) {
  console.log(nums);
  let sum = 0;
  for (let i = 0; i < nums.length; i++) {
    sum += nums[i];
  }
  return sum;
}

console.log(add(1, 2, 3, 4, 5, 6));

/*
[ 1, 2, 3, 4, 5, 6 ]
21 -> sum
*/
```

Here see this

In the function after using ...nums, this happens:

arguments => 1,2,3,4,5,6

Rest parameter makes all these arguments fit into an array

1,2,3,4,5,6 => [1,2,3,4,5,6]

and then from this array, each number is passed one by one, which is then added.

What if we do nums instead of ...nums

```
function add(nums) {
  console.log(nums);
  let sum = 0;
  for (let i = 0; i < nums.length; i++) {
    sum += nums[i];
  }
  return sum;
}

console.log(add(1, 2, 3, 4, 5, 6));

// 1 (it takes only 1 from [1,2,3,4,5,6])
// 0 -> as nums = 1 and nums is not an array, hence sum += nums[i] => sum = 0 + 0
(nums[i] = 0) = 0
```

Rest vs Spread

Rest	Spread
1. Rest parameter collects all argument values and fits them into an array	1. Spread operator spreads all the values from an array
2. Rest is used in function definition	2. Spread is used in function call

When using rest parameters + other manual parameters, make sure to put the rest parameter at the end as it considers all the remaining parameters. If we do not follow this, we will get an error.

some egs...

```
const arr = [1, 2, 3, 4, 5, 6];

function add(a, b, ...nums) {
  console.log(...nums);
  let sum = 0;
  for (let i = 0; i < nums.length; i++) {
    sum += nums[i];
  }
  return sum;
}

console.log(add(...arr));
// Here we get add(1,2,3,4,5,6)

/*
a = 1
b = 2
...nums = [3 4 5 6]
```

```
18 -> sum (...nums sum i.e. 3 + 4 + 5 + 6 = 18)
*/
```

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

function add(a, b, c, ...nums) {
  console.log("nums:", ...nums);
  let sum = 0;
  for (let i = 0; i < nums.length; i++) {
    sum += nums[i];
  }
  return sum;
}

console.log(add(...arr));
// Here we get add(1,2,3)

// a = 1
// b = 2
// c = 3

// ...nums = []

// sum = 0 (as we are doing ...nums ka sum)
```

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

function add(a, b, c, ...nums) {
  console.log("nums:", ...nums);
  let sum = 0;
  for (let i = 0; i < nums.length; i++) {
    sum += nums[i];
  }
  return sum;
}

console.log(add(...arr));
// Here we get add(1,2)

// a = 1
// b = 2
// c = undefined

// ...nums = []

// sum = 0 (as we are doing ...nums ka sum)
```

This practice gives us error as we did not write rest parameter at the end

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

function add(a, b, ...nums, c) {
  console.log('nums:', ...nums);
  let sum = 0;
  for (let i = 0; i < nums.length; i++) {
    sum += nums[i];
  }
  return sum;
}

console.log(add(...arr));
// Here we get add(1,2)
```

Let us see one more eg.

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

function add(a, b, c, ...nums) {
  console.log("nums:", ...nums);
  let sum = 0;
  for (let i = 0; i < nums.length; i++) {
    sum += nums[i];
  }
  return sum;
}

console.log(add(...arr, 10, 12));
// Here we get add(1,2, 10, 12)

// a = 1
//  b = 2
// c = 10
// nums = [12]

// sum = 12 (sum of ...nums)
```

Also rest is automatically converting the arguments object into an array, we can use array methods like reduce also to find the sum of arguments object

Let us look at some ways to do so

1.

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

function add() {
```

```
  return [...arguments].reduce((acc, curr) => acc + curr);
}

const result = add(...nums);
console.log(result);
// 10
```

2.

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

function add() {
  return Array.from(arguments).reduce((acc, curr) => acc + curr);
}

const result = add(...nums);
console.log(result);

// 10
```

3. IMP

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

function add(...nums) {
  return nums.reduce((acc, curr) => acc + curr);
}

const result = add(...nums);
console.log(result);

// 10
```

What will be the output of the following code using rest parameters?

```
function concatenate(separator, ...strings) {
  return strings.join(separator);
}

console.log(concatenate(", ", "apple", "banana", "cherry"));
```

Ans => apple, banana, cherry

Destructuring in JavaScript

Now this is basically used with objects and arrays mainly and used to extract their values. It came with ES6.

Let us see how to do it.

Destructuring with Arrays

Earlier we used to extract values from arrays like this:

```
const colors = ["red", "green", "yellow", "pink", "black"];

const color1 = colors[0];
const color2 = colors[1];
const color3 = colors[2];

console.log(color1);
console.log(color2);
console.log(color3);

/*
red
green
yellow
*/
```

With Destructuring, we can shorten it:

```
const colors = ["red", "green", "yellow", "pink", "black"];

const [color1, color2, color3, four, paanch, chhakka] = colors;

console.log(color1);
console.log(color2);
console.log(color3);
console.log(four);
console.log(paanch);
console.log(chhakka); // no value exists in the array at sixth place, so we get
undefined

/*
red
green
yellow
pink
black
undefined
*/
```

On LHS, we can give any value as seen with `four and paanch`.

Now what if we just need pink from the array??

```
const colors = ["red", "green", "yellow", "pink", "black"];

const [, , , pinkDedo] = colors;

console.log(pinkDedo);

// pink
```

Destructuring with Objects

Basic Way

```
const user = {
  name: "Anurag",
  age: 25,
  address: {
    city: "Bangalore",
    state: "Karnataka",
  },
};

const name = user.name;
const state = user.address.state;

console.log(name);
console.log(state);
```

Destructuring way

```
const user = {
  name: "Anurag",
  age: 25,
  address: {
    city: "Bangalore",
    state: "Karnataka",
  },
};

const { name, age, address, city, state, keyX } = user;
```

```
console.log(name);
console.log(age);
console.log(address);
console.log(city);
console.log(state);
console.log(keyX);

/*
Anurag
25
{ city: 'Bangalore', state: 'Karnataka' }
undefined
undefined
undefined
*/
```

Now let us see what happened:

In Object destructuring, it is important to give the variable name same as the key of the object. If we do not do so then it becomes difficult for the object to locate the key and its value, hence we get undefined. (see output of keyX).

Also we get `undefined` for nested keys as they are inside a different object (see city, state output).

Now let us see, how can we destructure city and state

1. Basic Way

```
const user = {
  name: "Anurag",
  age: 25,
  address: {
    city: "Bangalore",
    state: "Karnataka",
  },
};

const { address } = user;
// makes an address variable and puts an object i.e. value of address key

const { city } = address;
const { state } = address;

console.log(city);
console.log(state);

Bangalore;
Karnataka;
```

2. Multi Level Destructuring

```
const user = {
  name: "Anurag",
  age: 25,
  address: {
    city: "Bangalore",
    state: "Karnataka",
  },
};

const {
  address: { city },
} = user;
console.log(city);
// console.log(address);
```

In this method we shortened the code in 1 line, but here we are not creating the address variable as separate. So it makes one less global variable which is better.

eg2

```
const user = {
  name: "Anurag",
  age: 25,
  address: {
    city: "Bangalore",
    state: "Karnataka",
  },
};

const {
  address: { city, state },
} = user;
console.log(city);
console.log(state);

// Bangalore
// Karnataka
```

`console.log(address);` will give us an error as it has not been created.

Now what if we want to customize our variable name, apart from just just object keys as variable names

```
const user = {
  name: "Anurag",
  age: 25,
```

```
  address: {
    city: "Bangalore",
    state: "Karnataka",
  },
};

const {
  name: userName,
  age: userAge,
  address: { city: myCity, state: myState },
} = user;
console.log(userName);
console.log(userAge);
console.log(myCity);
console.log(myState);

/*
Anurag
25
Bangalore
Karnataka
*/
```

Here we created 4 variables named userName, userAge, myCity and myState and we took the values from name, age, address: {city} and address: {state} and put them in these variables.

Now earlier we used to do with arrays:

```
const colors = ["red", "green", "yellow", "pink", "black"];

const [, , , pinkDedo] = colors;

console.log(pinkDedo);

// pink
```

But we know arrays are objects too!! So we can also use the array index as object keys `BUT, we have to mandatorily rename the keys`

eg

```
const colors = ["red", "green", "yellow", "pink", "black"];

const { 3: pinkDedo, 0: redDedo } = colors;

console.log(pinkDedo);
console.log(redDedo);
```

```
// pink
//  red
```

Destructuring Arrays and Objects in Function Parameters

Functions with Arrays 1

```
const colors = ["red", "green", "yellow", "pink", "black"];

function myArr({ 2: two, 0: zero }) {
  console.log(two, zero);
}

myArr(colors);

// yellow red
```

Another Way

```
const colors = ["red", "green", "yellow", "pink", "black"];

function myArr([zero, , , , four]) {
  console.log(four, zero);
}

myArr(colors);

// black red
```

Functions with Objects

```
const user = {
  name: "Shukla",
  age: 20,
  address: {
    city: "varanasi",
    state: "UP",
  },
};

const myObj = ({ name, address: { city: userCity } }) => {
  console.log(name, userCity);
};
myObj(user);

// Shukla varanasi
```

Array.sort() and Array.toSorted() methods

Array.sort()

The `Array.sort()` method takes in all the input elements that it is sorting, considers all of it as strings and then sorts it. String Sorting means lexicographical sorting.

Also, it does not create a new array while sorting, it does `in-place sorting`.

```
const numbers = [4, 78, 6, 23, 5, 8, 997, 12, 1];

const fruits = ["banana", "apple", "mango", "pineapple", "cherry", "fig"];

console.log(numbers.sort());

console.log(fruits.sort());

/*
[1, 12, 23, 4, 5, 6, 78, 8, 997]

[ 'apple', 'banana', 'cherry', 'fig', 'mango', 'pineapple' ]
*/
```

Now let us properly sort numbers:

```
const numbers = [4, 78, 6, 23, 5, 8, 997, 12, 1];

const ans = numbers.sort((a, b) => {
  // return "hi";  // no change in array
  // return 0;  // no change in array
  // return false; // no change in array
  // return 1; // no change in array
  // return true;  // no change in array
  /* [
  4,  78,  6, 23, 5,
  8, 997, 12,  1
] */

  return -1; // array is reversed

  /*
    [
   1, 12, 997, 8, 5,
  23,  6,  78, 4
]
    */
```

```
});

console.log(ans);
```

Now, a and b are 2 numbers in the array

a -> the later number b -> first number

eg. [4, 78, 6, 23, 5, 8, 997, 12, 1]

here take initially. 1, 12

then a = 12 b = 1

This is when we are returning the array as it is i.e. `a will come after b`

But when we are returning the array in `reversed order`: `a will come before b`

[1, 12, 997, 8, 5, 23, 6, 78, 4]

eg. 4, 78

a = 78 b = 4

Normal Ascending Order Sorting of numbers

```
const numbers = [4,78,6,23,5,8,997,12,1];

const ans = numbers.sort((a, b) => {
   return (a - b);
})

console.log(ans);

/*
[
   1,  4,  5,   6, 8,
  12, 23, 78, 997
]
*/
```

We are returning a +ve value from the function and doing `(a - b)` means always b is greater than a

i.e. if `a` is a smaller no. i.e. difference of (a-b) is negative, then that -ve value is stored and the array is fully traversed. Then that smaller number will be pushed forward and the difference is tried to be made +ve, this happens till all difference is +ve and in result we get a sorted array in ascending order.

Normal Descending order sorting of array

```
const numbers = [4,78,6,23,5,8,997,12,1];

const ans = numbers.sort((a, b) => {
   return (b - a);
})

console.log(ans);

/*
[
  997, 78, 23, 12, 8,
    6,  5,  4,  1
]

*/
```

Here logic is same but we are trying to return a negative value that is `(b - a)` should be negative.

The number of times this sort function runs depends on the JS Engine.

Another way of writing sort functions

1. Ascending

```
const numbers = [4,78,6,23,5,8,997,12,1];

const ans = numbers.sort((a, b) => {
    if(a > b) {
        return 1;
        // it means the later number is bigger than previous one
    }
    else if (b > a) {
        return -1;
        // it means the later number is smaller than previous one

    }
    else return 0;
    // It means both nos are same
})

console.log(ans);

/*
[
   1,  4,  5,   6, 8,
  12, 23, 78, 997
]

*/
```

Descending

```
const numbers = [4,78,6,23,5,8,997,12,1];

const ans = numbers.sort((a, b) => {
    if(a > b) {
        return -1;
        // it means the later number is samller than previous one
    }
    else if (b > a) {
        return 1;
        // it means the later number is bigger than previous one

    }
    else return 0;
    // It means both nos are same
})

console.log(ans);

/*
[
  997, 78, 23, 12, 8,
    6,  5,  4,  1
]


*/
```

Adding undefined and an empty value in the array and then sorting them

1. Ascending

```
const numbers = [4,78,6,undefined,, 147, undefined, 475, 4, ,23,5,8,997,12,1];

const ans = numbers.sort((a, b) => {
   return (a - b);
})

console.log(ans);

/*
[
  1,
  4,
  4,
  5,
  6,
  8,
  12,
```

```
  23,
  78,
  147,
  475,
  997,
  undefined,
  undefined,
  <2 empty items>
]
*/
```

2. Descending

```
const numbers = [4,78,6,undefined,, 147, undefined, 475, 4, ,23,5,8,997,12,1];

const ans = numbers.sort((a, b) => {
   return (b - a);
})

console.log(ans);

/*
[
  997,
  475,
  147,
  78,
  23,
  12,
  8,
  6,
  5,
  4,
  4,
  1,
  undefined,
  undefined,
  <2 empty items>
]
*/
```

We notice that undefined is always at the last and empty values are after undefined values at the last too, be it ascending or descending order.

Putting in null and NaN too

Array is not sorted that properly but we see in ascending, null is always at first, in descending, null is always at last and NaN is somewhere in the series.

```
const numbers = [4,78,null,6,undefined,, 147, undefined,, null, 77, NaN, 475, 4,
,23,5,8,997,12,1];

const ans = numbers.sort((a, b) => {
   return (a - b);
})

console.log(ans);

/*
[
  null,            null,
  1,               4,
  4,               5,
  6,               8,
  12,              23,
  77,              78,
  147,             NaN,
  475,             997,
  undefined,       undefined,
  <3 empty items>
]
*/
```

2. descending

```
const numbers = [4,78,null,6,undefined,, 147, undefined,, null, 77, NaN, 475, 4,
,23,5,8,997,12,1];

const ans = numbers.sort((a, b) => {
   return (b - a);
})

console.log(ans);

/*
[
  997,             475,
  147,             78,
  77,              23,
  12,              8,
  6,               5,
  4,               4,
  1,               null,
  null,            NaN,
  undefined,       undefined,
  <3 empty items>
]

*/
```

So it is not preferred to use null and NaN while sorting array of numbers

Let us sort strings (Dictionary Sorting)

Normally, all uppercase strings come, first then lower case due to their ASCII value, but here, we are sorting them in the order they come in the dictionary.

1. Ascending order

```
// const numbers = [4,78,6,23,5,8,997,12,1];

const fruits = ['banana', 'Banana', 'apple', 'Apricot', 'mango', 'pineapple',
'cherry', 'fig', 'FIG'];

console.log(fruits.sort((a, b) => {
    const x = a.toLowerCase();
    const y = b.toLowerCase();

    if(x > y) {
        return 1;
    }
    else if(y > x) {
        return -1;
    }
    else {
        return 0;
    }
}));

/*
[
  'apple',     'Apricot',
  'banana',    'Banana',
  'cherry',    'fig',
  'FIG',       'mango',
  'pineapple'
]

*/
```

2. Descending

```
// const numbers = [4,78,6,23,5,8,997,12,1];

const fruits = ['banana', 'Banana', 'apple', 'Apricot', 'mango', 'pineapple',
'cherry', 'fig', 'FIG'];

console.log(fruits.sort((a, b) => {
```

```
    const x = a.toLowerCase();
    const y = b.toLowerCase();

    if(x > y) {
        return -1;
    }
    else if(y > x) {
        return 1;
    }
    else {
        return 0;
    }
}));


/*
[
  'pineapple', 'mango',
  'fig',       'FIG',
  'cherry',    'banana',
  'Banana',    'Apricot',
  'apple'
]
*/
```

Shortcut

```
// const numbers = [4,78,6,23,5,8,997,12,1];

const fruits = ['banana', 'Banana', 'apple', 'Apricot', 'mango', 'pineapple',
'cherry', 'fig', 'FIG'];


// Ascending
console.log(fruits.sort((a, b) => {
    return a.toLowerCase().localeCompare(b.toLowerCase());
}));


// Descending
console.log(fruits.sort((a, b) => {
    return b.toLowerCase().localeCompare(a.toLowerCase());
}));

/*

Ascending
[
  'apple',     'Apricot',
  'banana',    'Banana',
  'cherry',    'fig',
```

```
  'FIG',       'mango',
  'pineapple'
]


Descending
[
  'pineapple', 'mango',
  'fig',       'FIG',
  'cherry',    'banana',
  'Banana',    'Apricot',
  'apple'
]

*/
```

`a.localeCompare(b)` means

- if a is larger than b, return 1 (ascending order sorting)
- if a is smaller than b, return -1 (descending order sorting)
- If a and b are same, return 0 (no change)

Let us now focus on Array of Objects

Sorting on the basis of marks

```
const user = [
  {
    name: 'Adil',
    marks: 25,
  },
  {
    name: 'Kushal',
    marks: 58,
  },
  {
    name: 'Jatin',
    marks: 99,
  },
  {
    name: 'Vikas',
    marks: 5,
  },
];

const ans = user.sort((a, b) => (
    a.marks - b.marks
));

console.log(ans);
```

```
/*
[
  { name: 'Vikas', marks: 5 },
  { name: 'Adil', marks: 25 },
  { name: 'Kushal', marks: 58 },
  { name: 'Jatin', marks: 99 }
]
*/
```

Array.toSorted()

Now Array.sort() modifies the existing array but Array.toSorted() creates a new array copy of the original array and then sorts it.

Array.sort() eg

```
const user = [
  {
    name: 'Adil',
    marks: 25,
  },
  {
    name: 'Kushal',
    marks: 58,
  },
  {
    name: 'Jatin',
    marks: 99,
  },
  {
    name: 'Vikas',
    marks: 5,
  },
];

const ans = user.sort((a, b) => (
    a.marks - b.marks
));

console.log(ans);
console.log(user);

/*

new array ans
[
  { name: 'Vikas', marks: 5 },
  { name: 'Adil', marks: 25 },
  { name: 'Kushal', marks: 58 },
  { name: 'Jatin', marks: 99 }
```

```
]

old array user
[
  { name: 'Vikas', marks: 5 },
  { name: 'Adil', marks: 25 },
  { name: 'Kushal', marks: 58 },
  { name: 'Jatin', marks: 99 }
]

*/
```

Using Array.toSorted()

```
const user = [
  {
    name: 'Adil',
    marks: 25,
  },
  {
    name: 'Kushal',
    marks: 58,
  },
  {
    name: 'Jatin',
    marks: 99,
  },
  {
    name: 'Vikas',
    marks: 5,
  },
];

const ans = user.toSorted((a, b) => (
    a.marks - b.marks
));

console.log(ans);
console.log(user);

/*

new array ans
[
  { name: 'Vikas', marks: 5 },
  { name: 'Adil', marks: 25 },
  { name: 'Kushal', marks: 58 },
  { name: 'Jatin', marks: 99 }
]

old array user
[
```

```
  { name: 'Adil', marks: 25 },
  { name: 'Kushal', marks: 58 },
  { name: 'Jatin', marks: 99 },
  { name: 'Vikas', marks: 5 }
]

*/
```

This is a new method so if it is not supported, use sort() and do like this: (use ...)

```
const user = [
  {
    name: 'Adil',
    marks: 25,
  },
  {
    name: 'Kushal',
    marks: 58,
  },
  {
    name: 'Jatin',
    marks: 99,
  },
  {
    name: 'Vikas',
    marks: 5,
  },
];

const ans = [...user].sort((a, b) => (
    a.marks - b.marks
));

console.log(ans);
console.log(user);

/*

new
[
  { name: 'Vikas', marks: 5 },
  { name: 'Adil', marks: 25 },
  { name: 'Kushal', marks: 58 },
  { name: 'Jatin', marks: 99 }
]

old
[
  { name: 'Adil', marks: 25 },
  { name: 'Kushal', marks: 58 },
  { name: 'Jatin', marks: 99 },
```

```
    { name: 'Vikas', marks: 5 }
  ]

  */
```

So in React, use Array.toSorted() or [...].sort()