Data Structures, Modern Operators & Strings

> 🎓 **CS Perspective:** This section covers JavaScript's built-in **data structures** (arrays, objects, Maps, Sets) and **pattern matching** through destructuring. Destructuring is a form of **structural pattern matching** common in functional languages like ML and Haskell. The spread/rest operators demonstrate **variadic functions** (functions accepting variable arguments). Maps provide O(1) average-case lookup (hash table implementation), while Sets enforce **uniqueness constraints**. Short-circuit evaluation (`&&`, `||`, `??`) relates to **lazy evaluation**—operands are only evaluated when necessary, which has implications for side effects and performance.

📌 Destructuring Arrays

Extract values from arrays into variables:

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

// Without destructuring
const a = arr[0];
const b = arr[1];

// With destructuring
const [x, y, z] = arr;
console.log(x, y, z); // 2, 3, 4
```

Skip Elements

```
const [first, , third] = [1, 2, 3];
console.log(first, third); // 1, 3
```

Swap Variables

```
let [main, secondary] = ['Pizza', 'Pasta'];
[main, secondary] = [secondary, main];
// main = 'Pasta', secondary = 'Pizza'
```

Nested Destructuring

```
const nested = [2, 4, [5, 6]];
const [i, , [j, k]] = nested;
console.log(i, j, k); // 2, 5, 6
```

Default Values

```
const [p = 1, q = 1, r = 1] = [8, 9];
console.log(p, q, r); // 8, 9, 1
```

📌 Destructuring Objects

```
const restaurant = {
  name: 'Italiano',
  categories: ['Italian', 'Pizzeria'],
  openingHours: { thu: { open: 12, close: 22 } }
};

// Basic
const { name, categories } = restaurant;

// Rename variables
const { name: restaurantName, categories: tags } = restaurant;

// Default values
const { menu = [], starterMenu: starters = [] } = restaurant;

// Nested objects
const { openingHours: { thu: { open, close } } } = restaurant;
```

Mutating Variables

```
let a = 111;
let b = 999;
const obj = { a: 23, b: 7 };

// Must wrap in parentheses
({ a, b } = obj);
console.log(a, b); // 23, 7
```

📌 Spread Operator (...)

Expands an iterable into individual elements.

Arrays

```
const arr = [7, 8, 9];
const newArr = [1, 2, ...arr]; // [1, 2, 7, 8, 9]
```

```
// Copy array
const arrCopy = [...arr];

// Merge arrays
const merged = [...arr1, ...arr2];
```

Strings

```
const str = 'Jonas';
const letters = [...str]; // ['J', 'o', 'n', 'a', 's']
```

Objects (ES2018)

```
const newObj = { ...oldObj, newProp: 'value' };
```

Function Arguments

```
const add = (a, b, c) => a + b + c;
const nums = [1, 2, 3];
add(...nums); // 6
```

📌 Rest Pattern (...)

Collects multiple elements into an array. **Opposite of spread!**

```
// SPREAD: Right side of =
const arr = [1, 2, ...[3, 4]];

// REST: Left side of =
const [a, b, ...others] = [1, 2, 3, 4, 5];
console.log(others); // [3, 4, 5]
```

In Objects

```
const { sat, ...weekdays } = openingHours;
```

Function Parameters (Rest Parameters)

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

add(2, 3);           // 5
add(5, 3, 7, 2);     // 17
```

📌 Short Circuiting (&& and ||)

OR (||) - Returns first truthy value

```
console.log(3 || 'Jonas');     // 3
console.log('' || 'Jonas');    // 'Jonas'
console.log(undefined || null); // null
console.log(undefined || 0 || '' || 'Hello'); // 'Hello'

// Use for default values
const guests = numGuests || 10;
```

AND (&&) - Returns first falsy value

```
console.log(0 && 'Jonas');     // 0
console.log(7 && 'Jonas');     // 'Jonas'

// Use to execute code conditionally
restaurant.orderPizza && restaurant.orderPizza('mushrooms');
```

📌 Nullish Coalescing Operator (??)

Only considers `null` and `undefined` as falsy (NOT `0` or `''`).

```
restaurant.numGuests = 0;

const guests1 = restaurant.numGuests || 10; // 10 ❌
const guests2 = restaurant.numGuests ?? 10; // 0 ✅
```

📌 Logical Assignment Operators

```
const rest1 = { numGuests: 0 };
const rest2 = { owner: 'Giovanni' };

// OR assignment (||=)
rest1.numGuests ||= 10; // 10 (0 is falsy)
rest2.numGuests ||= 10; // 10

// Nullish assignment (??=)
rest1.numGuests ??= 10; // 0 (0 is not null/undefined)

// AND assignment (&&=)
rest2.owner &&= '<ANONYMOUS>'; // '<ANONYMOUS>'
rest1.owner &&= '<ANONYMOUS>'; // undefined (no change)
```

📌 Optional Chaining (?.)

Returns `undefined` if property doesn't exist instead of error.

```
// Without optional chaining
if (restaurant.openingHours && restaurant.openingHours.mon) {
  console.log(restaurant.openingHours.mon.open);
}

// With optional chaining
console.log(restaurant.openingHours?.mon?.open); // undefined

// With methods
console.log(restaurant.order?.(0, 1) ?? 'Method not found');

// With arrays
const users = [{ name: 'Jonas' }];
console.log(users[0]?.name ?? 'User not found');
```

📌 Looping Objects

```
const openingHours = {
  thu: { open: 12, close: 22 },
  fri: { open: 11, close: 23 }
};

// Keys
const days = Object.keys(openingHours); // ['thu', 'fri']

// Values
const hours = Object.values(openingHours);
```

```
// Entries (key-value pairs)
const entries = Object.entries(openingHours);

for (const [day, { open, close }] of entries) {
  console.log(`On ${day} we open at ${open}`);
}
```

📌 The for-of Loop

```
const menu = ['Pizza', 'Pasta', 'Risotto'];

for (const item of menu) {
  console.log(item);
}

// With index
for (const [index, item] of menu.entries()) {
  console.log(`${index + 1}: ${item}`);
}
```

📌 Sets

Collection of **unique** values.

```
const ordersSet = new Set(['Pasta', 'Pizza', 'Pizza', 'Risotto']);
console.log(ordersSet); // Set(3) {'Pasta', 'Pizza', 'Risotto'}
```

Set Methods

```
ordersSet.size;           // 3
ordersSet.has('Pizza');   // true
ordersSet.add('Bread');
ordersSet.delete('Risotto');
ordersSet.clear();
```

Convert Array to Unique Values

```
const staff = ['Waiter', 'Chef', 'Waiter', 'Manager'];
const uniqueStaff = [...new Set(staff)];
// ['Waiter', 'Chef', 'Manager']
```

Set Operations (ES2024)

```
const setA = new Set([1, 2, 3]);
const setB = new Set([2, 3, 4]);

setA.intersection(setB);        // Set {2, 3}
setA.union(setB);               // Set {1, 2, 3, 4}
setA.difference(setB);          // Set {1}
setA.symmetricDifference(setB); // Set {1, 4}
setA.isDisjointFrom(setB);      // false
```

📌 Maps

Key-value pairs where keys can be **any type**.

```
const rest = new Map();

// Set values (chainable)
rest.set('name', 'Italiano')
    .set(1, 'Firenze')
    .set(true, 'We are open');

// Get values
rest.get('name'); // 'Italiano'
rest.get(true);   // 'We are open'

// Other methods
rest.has('name'); // true
rest.delete(1);
rest.size;        // 2
rest.clear();
```

Create Map from Array

```
const question = new Map([
  ['question', 'Best language?'],
  [1, 'C'],
  [2, 'Java'],
  [3, 'JavaScript'],
  ['correct', 3],
  [true, 'Correct! 🎉 '],
  [false, 'Try again!']
]);
```

Convert Object to Map

```
const hoursMap = new Map(Object.entries(openingHours));
```

Iterate Map

```
for (const [key, value] of question) {
  if (typeof key === 'number') {
    console.log(`Answer ${key}: ${value}`);
  }
}
```

📌 When to Use What

Data Structure	Use When
Arrays	Ordered list, may have duplicates
Sets	Unique values, high-performance lookup
Objects	"Traditional" key-value, methods, JSON
Maps	Keys can be any type, better performance

📌 Strings - Part 1

```
const airline = 'TAP Air Portugal';
const plane = 'A320';

// Access characters
plane[0];           // 'A'
'B737'[0];          // 'B'

// Length
airline.length;     // 16

// Find position
airline.indexOf('r');      // 6 (first)
airline.lastIndexOf('r');  // 10 (last)
airline.indexOf('Portugal'); // 8

// Extract parts
airline.slice(4);        // 'Air Portugal'
airline.slice(4, 7);     // 'Air'
airline.slice(-2);       // 'al'
airline.slice(1, -1);    // 'AP Air Portuga'

// First/last word
```

```
airline.slice(0, airline.indexOf(' '));        // 'TAP'
airline.slice(airline.lastIndexOf(' ') + 1);   // 'Portugal'
```

📌 Strings - Part 2

```
const airline = 'TAP Air Portugal';

// Case
airline.toLowerCase();  // 'tap air portugal'
airline.toUpperCase();  // 'TAP AIR PORTUGAL'

// Trim whitespace
'  Hello  '.trim();      // 'Hello'
'  Hello  '.trimStart(); // 'Hello  '
'  Hello  '.trimEnd();   // '  Hello'

// Replace
'288,97£'.replace('£', '$').replace(',', '.'); // '288.97$'
'door door'.replace('door', 'gate');    // 'gate door'
'door door'.replaceAll('door', 'gate'); // 'gate gate'

// Booleans
airline.includes('Air');      // true
airline.startsWith('TAP');    // true
airline.endsWith('Portugal'); // true
```

📌 Strings - Part 3

```
// Split
'a+very+nice+string'.split('+'); // ['a', 'very', 'nice', 'string']
'Jonas Schmedtmann'.split(' ');  // ['Jonas', 'Schmedtmann']

// Join
['Mr.', 'Jonas', 'SCHMEDTMANN'].join(' '); // 'Mr. Jonas SCHMEDTMANN'

// Padding
'Jonas'.padStart(10, '+'); // '+++++Jonas'
'Jonas'.padEnd(10, '+');   // 'Jonas+++++'

// Repeat
'Bad weather! '.repeat(3); // 'Bad weather! Bad weather! Bad weather! '
```

Mask Credit Card

```
const maskCard = function(number) {
  const str = number + '';
  const last = str.slice(-4);
  return last.padStart(str.length, '*');
};

maskCard(43378463864647384); // '************7384'
```

🧪 Quick Reference Cheatsheet

```
// Destructuring
const [a, b] = [1, 2];
const { name, age } = person;

// Spread (expand)
const newArr = [...arr1, ...arr2];
const newObj = { ...obj, newProp: 'value' };

// Rest (collect)
const [first, ...rest] = arr;
function fn(...args) {}

// Short-circuit
value || defaultValue;  // OR
condition && action();  // AND
value ?? defaultValue;  // Nullish

// Optional chaining
obj?.prop?.nested;

// Sets
new Set([1, 2, 2, 3]); // {1, 2, 3}

// Maps
const map = new Map([['key', 'value']]);
map.set('key', 'value');
map.get('key');

// Strings
str.slice(start, end);
str.split(separator);
arr.join(separator);
str.padStart(length, char);
str.includes(substring);
str.replace(old, new);
```

✅ Exam Tips

1. **Spread** = expand (right side), **Rest** = collect (left side)
2. `||` returns first truthy, `&&` returns first falsy
3. `??` only treats `null`/`undefined` as falsy (not `0` or `''`)
4. **Optional chaining** `?.` prevents errors on undefined properties
5. **Sets** are for unique values, **Maps** for any-type keys
6. `Object.entries()` converts object to array for iteration
7. **Strings are immutable** - methods return new strings
8. `split()` → array, `join()` → string
9. Use `for...of` for iterating arrays/strings
10. **Template literals** use backticks and `${}` for interpolation