ARRAY DESTRUCTURING

DESTRUCTURING SYNTAX

Destructuring assignment is a special syntax in JavaScript, introduced in ES6, that allows us to “**unpack**” arrays or objects into a bunch of variables.

```
const person = ["John", "Smith", 28];

//Using array index for accessing array elements & assigning it to a variable
const firstName = person[0];
const lastName = person[1];
const age= person[2];

const [firstName, lastName, age] = ["John", "Smith", 28]; //Array Destructure
```

DESTRUCTURING SYNTAX

Destructuring makes extracting data from an array or an object, very simple and readable.

OBJECT DESTRUCTURING

DESTRUCTURING SYNTAX

Destructuring assignment is a special syntax in JavaScript, introduced in ES6, that allows us to “**unpack**” arrays or objects into a bunch of variables.

```
const employee = { name: 'Steve', age: 28, gender: 'Male' }

//Assign value of employee object properties to a variable
const name = employee.name;
const age = employee.age;
const gender = employee.gender

const {name, age, gender} = employee; //Object Destructuring
```

DESTRUCTURING OBJECTS

When we have a variable name inside object destructing syntax, for which, a property does not exist in the object which we are destructuring, in that case, that variable will be assigned with the value **undefined**.

THE SPREAD OPERATOR

THE SPREAD OPERATOR

The **spread operator** in JavaScript is a powerful and versatile feature that allows you to expand iterable objects (like ***arrays***, ***strings***, and ***maps***) into individual elements. It's used in various contexts, making your code more concise and readable.

THE SPREAD OPERATOR

Copying Arrays: Creating a true copy of an array (instead of a reference) is a common use case. Spread operator can be used to create a copy of an array from original array.

```
const originalArray = [1, 2, 3];
const newArray = [...originalArray]; // Creates a new, independent copy

console.log(newArray); // Output: [1, 2, 3]
originalArray.push(4);

console.log(originalArray); // Output: [1, 2, 3, 4]
console.log(newArray); // Output: [1, 2, 3] (newArray is not affected)
```

THE SPREAD OPERATOR

Concatenating Arrays: Combining multiple arrays into a single array can be achieved by spread operator.

```
//Create a new array by combining two arrays
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];

const combinedArray = [...array1, ...array2];

console.log(combinedArray); // Output: [1, 2, 3, 4, 5, 6]
```

THE SPREAD OPERATOR

Adding Elements to an Array: Inserting elements at specific positions along with using the elements of a given array.

```
//Create a new array where at the first position add element 0, then the
elements of array and than 4 & 5

const array = [1, 2, 3];
const newArray = [0, ...array, 4, 5];

console.log(newArray); // Output: [0, 1, 2, 3, 4, 5]
```

THE SPREAD OPERATOR

Spreading String Characters: Strings are also iterable, so you can use the spread operator to break them down into individual characters.

```
//Create an array of charachters from a string value by spreading the string
using spread operator.

const str = "hello";
const chars = [...str];

console.log(chars); // Output: ['h', 'e', 'l', 'l', 'o']
```

THE SPREAD OPERATOR

Spreading Object Properties (in ES2018 and later): You can also use spread operator to spread the properties of an object to create a new object with all the existing properties. Then you can add new properties or override the value of an existing property.

```
const obj1 = { a: 1, b: 2 };
const obj2 = { ...obj1, c: 3 }; // Creates a new object with properties from
obj1 and adds c: 3

console.log(obj2); // Output: { a: 1, b: 2, c: 3 }

const obj3 = { ...obj1, a: 10, d:4}; // Overwrites property a with 10
console.log(obj3); // Output: { a: 10, b: 2, d: 4}

const obj4 = {x: 5, y:6};
const mergedObj = {...obj1, ...obj4};
console.log(mergedObj); // Output: {a: 1, b: 2, x: 5, y: 6}
```

THE SPREAD OPERATOR

Key Points:

👉 **Shallow Copy:** When used with objects, the spread operator creates a shallow copy. Nested objects are still referenced, not deeply cloned.

👉 **Iterable Objects:** The spread operator works with iterable objects (arrays, strings, maps, sets). Regular objects are not directly iterable but work in the way shown above.

👉 **Readability:** The spread operator often makes code more concise and easier to understand compared to older methods (like concat for arrays or looping).

THE REST PATTERN & REST PARAMETER

THE REST PATTERN

The **rest operator** in JavaScript allows you to collect rest of the values into an array or rest of the properties into an object.

NOTE: The **rest operator** syntax is exactly like the **spread operator**. But rest operator does the opposite of spread operator.

👉 **Spread Operator:** Extract the elements of an array or properties of an Object into individual elements:

👉 **Rest Operator**: Create a new array with rest of the values in an expression.

THE REST PATTERN

The syntax of rest operator is same as the spread operator. To identify rest operator & spread operator, you can use following method:

👉 The rest operator always comes before the assignment operator.

👉 On the other hand, the spread operator always comes after the assignment operator.

```
//Below is an example of using spread operator:
const arr = [3, 4, 5];
const newArr = [1, 2, ...arr] //Spread operator comes on right side of
assignment operator

//Below is an example of rest pattern
const [a, b, ...rest] = [1, 2, 3, 4, 5];
```

THE REST PARAMETER

The **rest parameter** collects remaining arguments into an array within a function definition. This is used only in function parameter lists.

NOTE: Rest parameter should always be last in the parameter list.

```
function addNumbers(a, b, ...rest) { // a = 10, b = 20, rest = [30, 40, 50]
    let sum = a + b;

    for(let i = 0; i < rest.length; i++){
        sum += rest[i]
    }
    console.log(sum); // sum of all numbers
}

addNumbers(10, 20, 30, 40, 50)
```

UNDERSTANDING OPTIONAL CHAINING

OPTIONAL CHAINING

Optional chaining (`?.`) is a concise way to access potentially nested object properties without causing errors if an intermediate property doesn't exist. It short-circuits the property access if it encounters a **`null`** or **`undefined`** value, returning undefined instead of throwing an error.

NULLISH COALIESING OPERATOR

NULLISH COALESING OPERATOR

The **nullish coalescing** operator (`??`) is a logical operator in JavaScript that returns its right-hand side operand only when its left-hand side operand is **`null`** or **`undefined`**. It's a way to provide a default value when a variable is either explicitly null or undefined.

- 👉 It is a logical operator.
- 👉 Returns left hand side operand if it is not null or undefined.
- 👉 Else, return right hand side operand.

LOOPING OVER OBJECTS

UNDERSTANDING FOR-OF LOOP

THE FOR-OF LOOP

The `for...of` loop in JavaScript is a modern and elegant way to iterate over iterables. This includes **arrays**, **strings**, **maps**, **sets**, and other iterables. It simplifies iteration compared to the traditional `for` loop, especially when you just need the values of the iterable.

THE FOR-OF LOOP

The `for...of` loop in JavaScript is a modern and elegant way to iterate over iterables. This includes **arrays**, **strings**, **maps**, **sets**, and other iterables. It simplifies iteration compared to the traditional **for** loop, especially when you just need the values of the iterable.

```
//Loop over colors array and log each element
const colors = ["red", "green", "blue"];

for (const color of colors) {
  console.log(color);
}
```

THE FOR-OF LOOP

When to Use for...of vs. Traditional for:

👉 Use for...of when you need to iterate over the values of an iterable (arrays, strings, maps, sets, etc.) and you don't need the index.

👉 Use the traditional for loop when you need the index (e.g., to modify the original array or access elements based on their position) or when you need more fine-grained control over the iteration process.

👉 Also, if you're working with something that's not iterable, for...of won't work.

UNDERSTANDING FOR-IN LOOP

THE FOR-IN LOOP

The `for...in` loop in JavaScript is used to iterate over the enumerable properties of an object. It's important to understand that **it's designed for objects, not arrays** (although it can technically be used with arrays, but it's generally not recommended).

THE FOR-IN LOOP

The `for...in` loop in JavaScript is used to iterate over the enumerable properties of an object. It's important to understand that **it's designed for objects, not arrays** (although it can technically be used with arrays, but it's generally not recommended).

```javascript
const person = {
  name: "Alice",
  age: 30,
  city: "New York",
};

for (const key in person) {
  console.log(`Key: ${key}, Value: ${person[key]}`);
}
```

ES6 ENHANCED OBJECT LITERAL SYNTAX

ES6 SET()
DATA STRUCTURE

SET DATA STRUCTURE

A **`Set`** in JavaScript is a collection of unique values. This means that a **`Set`** can only contain a value once. If you try to add the same value multiple times, it will only be stored once in the Set.

Sets can hold values of any data type, including primitive values (like **numbers**, **strings**, **Booleans**), **objects**, and even other **Sets**.

CREATING A SET

Using the `new Set()` constructor: This is the most common way. You can optionally pass an iterable (like an **array**) to the constructor to initialize the Set with values:

```
// Empty Set
const mySet = new Set();

// Set with initial values from an array
const anotherSet = new Set([1, 2, 3, 3, 4]); // Note: Duplicate 3 is ignored
console.log(anotherSet); // Output: Set { 1, 2, 3, 4 }

// Set with different data types
const mixedSet = new Set([1, "hello", true, {name: "John"}]);
console.log(mixedSet); // Output: Set { 1, 'hello', true, { name: 'John' } }
```

CREATING A SET

Using the add() method: You can add elements to a Set after it's created using the add() method. This is useful for building up a **Set** dynamically.

```
const mySet = new Set();

mySet.add(1);
mySet.add(2);
mySet.add(2); // Adding a duplicate - no effect
mySet.add("hello");

console.log(mySet); // Output: Set { 1, 2, 'hello' }
```

SET METHODS

Here are some of the important methods you can use with Sets:

👉 **`add(value)`:** Adds a value to the Set. Returns the Set itself, so you can chain add() calls.

👉 **`delete(value)`:** Removes a value from the Set. Returns true if the value was found and deleted, false otherwise.

👉 **`has(value)`:** Returns true if the Set contains the value, false otherwise.

👉 **`clear()`:** Removes all elements from the Set.

👉 **`size`:** Returns the number of elements in the Set (like length for arrays).

SET METHODS

Here are some of the important methods you can use with Sets:

👉 **`values():`** Returns an iterator of the values in the Set (in insertion order).

👉 **`entries():`** Returns an iterator of [value, value] pairs for each element in the Set (similar to Map entries).

👉 **`forEach(callback):`** Executes a provided function once per each value in the Set, in insertion order.

WHAT IS THE USE OF SET

Sets are about Uniqueness and Membership, not Retrieval by Index: The primary focus of a Set is to efficiently store unique values and to quickly check for the presence of a value. The core operations are **`add()`**, **`delete()`**, and **`has()`**. The concept of an index or key doesn't align with this core purpose. Sets are designed for fast membership checks, not for retrieving elements based on their position.

Order is Implicit, Not Explicit: While Sets do maintain insertion order (elements are iterated in the order they were added), this order is not considered a primary feature for element retrieval. It's more of a side effect that's useful for iteration but not the main point. Sets are not designed for random access like arrays.

WHAT IS THE USE OF SET

No Key-Value Association: Sets store values directly; they don't associate values with keys like objects or maps. The concept of a "key" for retrieval doesn't exist in a Set.

Efficiency: If Sets were to provide indexed access, it would likely impact the performance of other core operations, particularly add() and has(). Maintaining an index would add overhead, and the design of Sets is optimized for speed in membership checks, which is a common use case.

USE OF SET()
DATA STRUCTURE

WHAT IS THE USE OF SET

In JavaScript, **Set** is a built-in object that stores unique values. Even though you cannot access elements by an index or a key like in arrays or objects, Set serves several useful purposes:

Ensuring Uniqueness: A Set automatically removes duplicate values, making it useful for filtering unique values in an array.

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

const uniqueNumbers = new Set(numbers);
console.log([...uniqueNumbers]); // [1, 2, 3, 4, 5]
```

WHAT IS THE USE OF SET

Fast Lookups: Checking if a value exists in a Set is faster than in an array because `Set.has(value)` operates in **O(1)** time complexity.

```
//Check if an item is available in Set
const mySet = new Set([1, 2, 3, 4]);

console.log(mySet.has(3)); // true
console.log(mySet.has(5)); // false
```

WHAT IS THE USE OF SET

Removing Duplicates from Arrays: A Set makes it easy to eliminate duplicate values in an array.

```
//Remove duplicate values from an array
const arr = [1, 2, 2, 3, 4, 4, 5];

const uniqueArr = [...new Set(arr)];
console.log(uniqueArr); // [1, 2, 3, 4, 5]
```

WHAT IS THE USE OF SET

Iteration Over Values: Even though you can't access values directly by index, you can iterate over a Set:

```
//Iterate over a set using for-of loop
const mySet = new Set(["apple", "banana", "cherry"]);

for (let item of mySet) {
    console.log(item);
}
// Output:
// apple
// banana
// cherry
```

WHAT IS THE USE OF SET

Useful in Set Operations (Union, Intersection, Difference): Sets are helpful for mathematical operations like union, intersection, and difference.

```
const setA = new Set([1, 2, 3]);
const setB = new Set([3, 4, 5]);

//Union (combine two sets)
const unionSet = new Set([...setA, ...setB]);
console.log([...unionSet]); // [1, 2, 3, 4, 5]

//Intersection (find common values)
const intersectionSet = new Set([...setA].filter(x => setB.has(x)));
console.log([...intersectionSet]); // [3]

//Difference (values in setA but not in setB)
const differenceSet = new Set([...setA].filter(x => !setB.has(x)));
console.log([...differenceSet]); // [1, 2]
```

WHAT IS THE USE OF SET

Removing Items Easily: Unlike arrays where you have to find the index to remove an element, `Set.delete(value)` makes it easy to remove an item.

```
//Remove values from a set
const mySet = new Set(["apple", "banana", "cherry"]);

mySet.delete("banana");
console.log(mySet); // Set { 'apple', 'cherry' }
```

LOOPING OVER SETS IN JAVASCRIPT

ES2025 NEW
SETS METHODS

ES6 MAP()
DATA STRUCTURE

MAP DATA STRUCTURE

In JavaScript, a `Map` is a data structure that holds key-value pairs, where both the keys and the values can be any data type (unlike plain `objects`, where keys are traditionally **strings** or **Symbols**).

`Maps` maintain the insertion order of elements, meaning the order in which you add key-value pairs is preserved. This is a key difference from regular objects.

CREATE MAP FROM ARRAY & OBJECT

LOOPING OVER MAPS IN JAVASCRIPT

PROJECT OVERVIEW
E-SHOPPING CART

SHOW UNIQUE CATEGORIES

RENDER ALL PRODUCTS IN UI

REMOVE MISSING PRODUCT DETAILS

SHOW SELECTED PRODUCT DETAILS

SHOW SELECTED PRODUCT FEATURES

ADD TO CART FUNCTIONALITY

REMOVE FROM CART FUNCTIONALITY

SHOW PRODUCTS ADDED TO CART

UPDATE CART PAGE ON PRODUCT REMOVE

CALCULATE & SHOW DISCOUNT PRICE

IMPLEMENT CHECKOUT PAGE

SHOW PRODUCTS IN CHECKOUT PAGE

SHOW TOTAL PRICE IN CHECKOUT PAGE

FILTER PRODUCTS BY CATEGORY