DSA in JavaScript

Property Management Methods: -

```
const person = {
  firstName: "John",
  lastName : "Doe",
  language : "EN"
};
```

// Adding or changing an object property

Object.defineProperty(object, property, descriptor)

```
// Add a Property
Object.defineProperty(person, "year", {value:"2008"});

// Change a Property
Object.defineProperty(person, "language", {value : "NO"});
```

Property Attributes: -

All properties have a name. In addition they also have a value.

The value is one of the property's attributes.

Other attributes are: enumerable, configurable, and writable.

These attributes define how the property can be accessed (is it readable?, is it writable?)

In JavaScript, all attributes can be read, but only the value attribute can be changed (and only if the property is writable).

Changing Meta Data: -

writable : true // Property value can be changed

enumerable : true // Property can be enumerated

configurable : true // Property can be reconfigured

false means cannot be done as per the attribute.

This example makes language read-only:

Object.defineProperty(person, "language", {writable:false});

This example makes language not enumerable.

Object.defineProperty(person, "language", {enumerable:false});

// Adding or changing object properties

Object.defineProperties(object, descriptors)

// Accessing a Property

Object.getOwnPropertyDescriptor(object, property)

// Accessing Properties

Object.getOwnPropertyDescriptors(object)

// Returns all properties as an array

Object.getOwnPropertyNames(object) // it will also list not enumerable.

Object.getOwnPropertyNames(person);

// firstName,lastName,language

The Object.keys() method returns all enumerable properties.

// Accessing the prototype

Object.getPrototypeOf(object)

Adding Getters and Setters: -

The Object.defineProperty() method can also be used to add Getters and Setters.

```
// Define a getter
Object.defineProperty(person, "fullName", {
  get: function () {return this.firstName + " " + this.lastName;}
});
```

For setter, write set.

JavaScript Object Accessors: -

JavaScript can secure better data quality when using getters and setters.

```
const person = {
  firstName: "John",
  lastName: "Doe",
  language: "en",
  get lang() {  // access as a property
    return this.language;
```

```
  }
}; // person.lang

const person = {
  firstName: "John",
  lastName: "Doe",
  language: "",
  set lang(lang) {
    this.language = lang;
  }
};

// Set an object property using a setter:
person.lang = "en";
```

Object Protection Methods: -

```
// Prevents re-assignment

const car = {type:"Fiat", model:"500", color:"white"};


// Prevents adding object properties

Object.preventExtensions(object)

Object.preventExtensions(person);

// This will throw an error
person.nationality = "English";
```

Can also be done for an array since it is also an object.

```
// Returns true if properties can be added to an object

Object.isExtensible(object)

// This will return false
let answer = Object.isExtensible(person);


// Prevents adding and deleting object properties

Object.seal(object)

// Seal Object

Object.seal(person)
```

```
// This will throw an error
delete person.age;


// Returns true if object is sealed
Object.isSealed(object)
// This will return true
let answer = Object.isSealed(person);


// Prevents any changes to an object
Object.freeze(object)
```

Frozen objects are read-only.

```
// Freeze Object
Object.freeze(person)
// This will throw an error
person.age = 51;


// Returns true if object is frozen
Object.isFrozen(object)
// This will return true
let answer = Object.isFrozen(person);
```

Note: -

All JavaScript objects inherit properties and methods from a prototype:

Date objects inherit from Date.prototype

Array objects inherit from Array.prototype

Person objects inherit from Person.prototype

The Object.prototype is on the top of the prototype inheritance chain.

Date objects, Array objects, and Person objects inherit from Object.prototype

JavaScript Array Search: -

indexOf(): -

fruits.indexOf(“Apple”); // returns index if found. Not found means -1

another parameter can be added which is optional from where to start the search.

lastIndexOf(): -

returns the index of the last occurrence of the specified element.

Includes(): -

Fruits.includes(“Mango”) // returns true if exists

Find(): -

The find() method returns the value of the first array element that passes a test function.

Const numbers = [4, 9, 16, 25, 29];

Let first = numbers.find(myFunction);

Function myFunction(value, index, array) {

  Return value > 18;

}

findIndex(): -

It returns the index of the first array element that passes a test function.

findLast(): -

It will start from the end of an array and return the value of the first element that satisfies a condition.

Const temp = [27, 28, 30, 40, 42, 35, 30];

Let high = temp.findLast(x => x > 40); // 42

findLastIndex(): -

This method finds the index of the last element that satisfies a condition.

JavaScript Sorting Arrays: -

The sort() method sorts an array alphabetically.

const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort(); // Apple,Banana,Mango,Orange

The **reverse()** method reverses the elements in an array.

fruits.reverse(); // Mango,Apple,Orange,Banana

toSorted() Method: -

toSorted() method is a safe way to sort an array without altering the original array.

toReversed() Method: -

toReversed() method is a safe way to reverse an array without altering the original array.

Numeric Sort: -

the sort() function sorts values as strings.

Hence, sort() method will produce incorrect result when sorting numbers. Therefore, use compare function.

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return a - b});
```

The Compare Function: -

It defines an alternative sort order.

The compare function should return a negative, zero, or positive value, depending on the arguments.

When the sort() function compares two values, it sends the values to the compare function, and sorts the values according to the returned (negative, zero, positive) value.

If the result is negative, a is sorted before b.

If the result is positive, b is sorted before a.

If the result is 0, no changes are done with the sort order of the two values.

Sorting an Array in Random Order: -

```
points.sort(function(){return 0.5 - Math.random()});
```

Find the Lowest (or Highest) Array Value: -

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return a - b});
// now points[0] contains the lowest value
// and points[points.length-1] contains the highest value
```

Using Math.min() and Math.max() on an Array.

```
function myArrayMin(arr) {
  return Math.min.apply(null, arr);
}

function myArrayMax(arr) {
  return Math.max.apply(null, arr);
}
```

Sorting Object Arrays: -

```
const cars = [
  {type:"Volvo", year:2016},
  {type:"Saab", year:2001},
```

```
  {type:"BMW", year:2010}
];

cars.sort(function(a, b){return a.year - b.year});
```

Comparing string properties: -

```
cars.sort(function(a, b){
  let x = a.type.toLowerCase();
  let y = b.type.toLowerCase();
  if (x < y) {return -1;}
  if (x > y) {return 1;}
  return 0;
});
```

JavaScript Array map(): -

The map() method creates a new array by performing a function on each array element.

The map() method does not execute the function for array elements without values.

The map() method does not change the original array.

```
const numbers1 = [45, 4, 9, 16, 25];

const numbers2 = numbers1.map(myFunction);

function myFunction(value, index, array) {

  return value * 2;

}
```

Array flatMap(): -

The flatMap() method first maps all elements of an array and then creates a new array by flattening the array.

```
const myArr = [1, 2, 3, 4, 5, 6];
const newArr = myArr.flatMap((x) => x * 2); // 1,10,2,20,3,30,4,40,5,50,6,60
```

filter(): -

The filter() method creates a new array with array elements that pass a test.

```
const numbers = [45, 4, 9, 16, 25];

const over18 = numbers.filter(myFunction);

function myFunction(value, index, array) {

  return value > 18;

}
```

Array reduce(): -

The reduce() method runs a function on each array element to produce (reduce it to) a single value. The reduce() method does not reduce the original array.

```
const numbers = [45, 4, 9, 16, 25];
let sum = numbers.reduce(myFunction);
function myFunction(total, value, index, array) {
  return total + value;
}

// also accepts initial value as an another argument.
```

reduceRight(): -

The reduceRight() works from right-to-left in the array

every(): -

The every() method checks if all array values pass a test.

```
const numbers = [45, 4, 9, 16, 25];
let allOver18 = numbers.every(myFunction); // true or false
function myFunction(value, index, array) {
  return value > 18;
}
```

some(): -

The some() method checks if some array values pass a test.

```
const numbers = [45, 4, 9, 16, 25];
let someOver18 = numbers.some(myFunction); // true or false
function myFunction(value, index, array) {
  return value > 18;
}
```

Array.from(): -

The Array.from() method returns an Array object from any object with a length property or any iterable object.

```
Array.from("ABCDEFG");

// A,B,C,D,E,F,G
```

Array keys(): -

The Array.keys() method returns an Array Iterator object with the keys of an array.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
const keys = fruits.keys();
```

the keys contain indexes.

Array entries(): -

The entries() method returns an Array Iterator object with key/value pairs

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];

const f = fruits.entries();

for (let x of f) {

  document.getElementById("demo").innerHTML += x;

}
```

Array with(): -

It is a safe way to update elements in an array without altering the original array.

```
const months = ["Januar", "Februar", "Mar", "April"];
const myMonths = months.with(2, "March"); // index, new item
```

Array Spread (...): -

The ... operator expands an iterable (like an array) into more elements

```
onst q1 = ["Jan", "Feb", "Mar"];
const q2 = ["Apr", "May", "Jun"];
const q3 = ["Jul", "Aug", "Sep"];
const q4 = ["Oct", "Nov", "Dec"];
const year = [...q1, ...q2, ...q3, ...q4];

// Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec
```

Pattern problem: -

Same as discussed in doc3. different approaches to solve this problem.

1st one: - Using a 2D Array and Alternating Rows

```
function generatePattern(n) {

   let matrix = [];

   let num = 1;

   // Initialize the matrix with zeros

   for (let i = 0; i < n; i++) {

      matrix.push(new Array(n).fill(0));

   }
```

```
    let top = 0, bottom = n - 1;
    while (top <= bottom) {
        // Fill the top row
        for (let i = 0; i < n; i++) {
            matrix[top][i] = num++;
        }
        top++;
        if (top > bottom) break;
        // Fill the bottom row
        for (let i = 0; i < n; i++) {
            matrix[bottom][i] = num++;
        }
        bottom--;
    }
    // Print the matrix
    for (let i = 0; i < n; i++) {
        for(let j=0;j<n;j++){
            process.stdout.write(matrix[i][j]+ " ");
        }
        console.log();
    }
}
let n = parseInt(prompt("Enter the length (n): "));
generatePattern(n);
```

Time Complexity: -

Filling the matrix takes $O(n^2)$ because we iterate over all n×n elements.

Printing the matrix also takes $O(n^2)$

Total Time Complexity: $O(n^2)$

Space Complexity: -

The matrix itself takes $O(n^2)$ space.

Total Space Complexity: $O(n^2)$

2nd one: - Mathematical Index Calculation

```
function generatePattern(n) {
    for (let i = 0; i < n; i++) {
        let row = [];
        for (let j = 0; j < n; j++) {
            let pos;
            if (i < Math.ceil(n / 2)) {
                pos = i * 2 * n + j + 1;
            } else {
                pos = (n - 1 - i) * 2 * n + n + j + 1;
            }
            row.push(pos);
        }
        console.log(row.join(" "));
    }
}
let n= parseInt(prompt("Enter a number:"));
generatePattern(n);
```

Time Complexity: -

Calculating and printing each cell takes O(1) and there are n x n cells.

Total Time Complexity: $O(n^2)$

Space Complexity: -

No additional space is used apart from the row array for printing.

Total Space Complexity: O(n) (for the row array).

Explanation: -

1. Top Half Formula

For the top half, the rows are filled in the order $0,1,2,\ldots,[n/2]-1$

The first row (i=0) starts at 1 and ends at n.

The second row (i=1) starts at 2n+1 and ends at 3n.

The third row (i=2) starts at 4n+1 and ends at 5n.

And so on...

From this pattern, we can see that:

The starting number for row i in the top half is

$start=i\times 2n+1$

The value at position (i,j) is:

$pos=start+j=i\times 2n+j+1$

2. Bottom Half Formula

For the bottom half, the rows are filled in the order $n-1,n-2,n-3,\ldots,[n/2]$.

The last row (i=n−1) starts at n+1 and ends at 2n.

The second-last row (i=n−2) starts at 3n+1 and ends at 4n.

The third-last row (i=n−3i=n−3) starts at 5n+15n+1 and ends at 6n6n.

And so on...

From this pattern, we can see that:

The starting number for row ii in the bottom half is:

$start=(n-1-i)\times 2n+n+1$

The value at position (i,j) is:

$pos=start+j=(n-1-i)\times 2n+n+j+1$

Console app for train ticket: -

Start: The app begins by displaying the main menu.

Display Main Menu: shows options

1.Display available trains.

2.Book a ticket.

3.Cancel a booking.

4.Exit the app.

User Input: The user selects an option by entering a number (1, 2, 3, or 4).

Option 1: Display Trains:

The app displays a list of available trains with details (train number, name, source, destination, departure time, arrival time, distance, and seats available).

Option 2: Book Ticket:

The app asks the user to select a train by entering its ID.

The app then asks for passenger details (name and age).

It checks if seats are available for the selected train.

If seats are available, it generates a unique booking ID.

If no seats are available, it informs the user and returns to the main menu.

Option 3: Cancel Booking:

The app asks the user to enter the booking ID.

It checks if the booking exists.

If the booking exists, it releases the seat, removes the booking, and displays a cancellation confirmation.

If the booking does not exist, it informs the user and returns to the main menu.

Option 4: Exit:

The app closes.