Data Structures, Modern Operators, and Strings

1. Destructuring Arrays

Introduction

Destructuring arrays allows us to unpack values from an array into separate variables, making it easier to access and manipulate data.

Basic Syntax

```
const fruits = ["Apple", "Banana", "Cherry"];
const [first, second, third] = fruits;

console.log(first); // Output: Apple
console.log(second); // Output: Banana
console.log(third); // Output: Cherry
```

Skipping Elements

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

console.log(first); // Output: 1
console.log(third); // Output: 3
```

Using Default Values

If the array doesn't contain enough elements, default values can be used.

```
const colors = ["Red"];
const [primary, secondary = "Blue"] = colors;

console.log(primary); // Output: Red
console.log(secondary); // Output: Blue
```

---

2. Destructuring Objects:

Destructuring objects allows us to extract properties from an object and assign them to variables.

Basic Syntax for Destructuring Objects

```
const person = { name: "Alice", age: 25 };
const { name, age } = person;

console.log(name); // Output: Alice
console.log(age); // Output: 25
```

Using Different Variable Names

```
const product = { id: 1, title: "Laptop" };
const { id: productId, title: productTitle } = product;

console.log(productId); // Output: 1
console.log(productTitle); // Output: Laptop
```

Setting Default Values

```
const user = { username: "John" };
const { username, role = "Guest" } = user;

console.log(username); // Output: John
console.log(role); // Output: Guest
```

Nested Destructuring

```
const student = { name: "Sarah", marks: { math: 90, science: 85 } };
const {
  name,
  marks: { math, science },
} = student;

console.log(name); // Output: Sarah
console.log(math); // Output: 90
console.log(science); // Output: 85
```

---

3. The Spread Operator (...):

The spread operator expands an iterable (e.g., array, string) into individual elements.

For Arrays

- **Copying Arrays**

```
```

```
const original = [1, 2, 3];
const copy = [...original];

console.log(copy); // Output: [1, 2, 3]
```

- **Merging Arrays**

```
const arr1 = [1, 2];
const arr2 = [3, 4];
const combined = [...arr1, ...arr2];

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

For Objects

- **Copying Objects**

```
const original = { a: 1, b: 2 };
const copy = { ...original };

console.log(copy); // Output: { a: 1, b: 2 }
```

- **Merging Objects**

```
const obj1 = { a: 1 };
const obj2 = { b: 2 };
const merged = { ...obj1, ...obj2 };

console.log(merged); // Output: { a: 1, b: 2 }
```

---

4. Rest Pattern and Parameters:

The rest pattern collects multiple elements into a single array or object.

In Arrays

```
const [first, second, ...rest] = [10, 20, 30, 40, 50];
console.log(first); // Output: 10
console.log(second); // Output: 20
console.log(rest); // Output: [30, 40, 50]
```

In Functions

```
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}

console.log(sum(1, 2, 3, 4)); // Output: 10
```

In Objects

```
const { a, ...rest } = { a: 1, b: 2, c: 3 };
console.log(a); // Output: 1
console.log(rest); // Output: { b: 2, c: 3 }
```

---

5. Short-Circuiting (&& and ||):

Short-circuiting evaluates expressions from left to right and stops as soon as the result is known.

OR Operator (`||`)

Returns the first truthy value or the last falsy value.

```
const a = 0 || "Hello";
const b = "" || "Default";
console.log(a); // Output: Hello
console.log(b); // Output: Default
```

AND Operator (`&&`)

Returns the first falsy value or the last truthy value.

```
const a = true && "Hello";
const b = null && "World";
console.log(a); // Output: Hello
console.log(b); // Output: null
```

---

6. Nullish Coalescing Operator (`??`):

The nullish coalescing operator (`??`) returns the right-hand operand when the left-hand operand is `null` or `undefined`.

Example Usage

```
const value = null ?? "Default";
console.log(value); // Output: Default

const value2 = 0 ?? "Default";
console.log(value2); // Output: 0
```

Difference from OR (||)

- OR (||) checks for **falsy** values (e.g., 0, '', null, undefined).
- Nullish coalescing (??) only checks for **null or undefined**.

---

7. Logical Assignment Operators:

Logical assignment operators combine logical operators (&&, ||, ??) with assignment.

Examples

- **OR Assignment (||=)**

```
let x = null;
x ||= "Default";
console.log(x); // Output: Default
```

- **AND Assignment (&&=)**

```
let y = true;
y &&= "Updated";
console.log(y); // Output: Updated
```

- **Nullish Assignment (??=)**

```
let z = undefined;
z ??= "Assigned";
console.log(z); // Output: Assigned
```

---

8. Looping Arrays: The for...of Loop:

The for...of loop is a modern way to iterate over iterable objects like arrays, strings, or maps.

Basic Syntax for the `for...of` Loop

```
const fruits = ["Apple", "Banana", "Cherry"];

for (const fruit of fruits) {
  console.log(fruit);
}
// Output:
// Apple
// Banana
// Cherry
```

Getting the Index (Using `entries()`)

```
const fruits = ["Apple", "Banana", "Cherry"];

for (const [index, fruit] of fruits.entries()) {
  console.log(`Index ${index}: ${fruit}`);
}
// Output:
// Index 0: Apple
// Index 1: Banana
// Index 2: Cherry
```

---

9. Enhanced Object Literals:

Enhanced object literals make it easier to write and manage objects by introducing concise syntax for defining methods, setting dynamic keys, and using variables as properties.

Defining Methods

```
const person = {
  name: "Alice",
  greet() {
    console.log("Hello!");
  },
};

person.greet(); // Output: Hello!
```

Dynamic Property Names

```
const dynamicKey = "age";
const person = {
  name: "Alice",
  [dynamicKey]: 25,
};

console.log(person.age); // Output: 25
```

Using Variables as Properties

```
const name = "Bob";
const age = 30;

const person = { name, age };
console.log(person); // Output: { name: 'Bob', age: 30 }
```

---

10. Optional Chaining (`?.`):

Optional chaining allows you to safely access deeply nested properties without worrying about errors when a property is `null` or `undefined`.

Basic Usage

```
const user = { profile: { name: "Alice" } };

console.log(user.profile?.name); // Output: Alice
console.log(user.profile?.age); // Output: undefined
console.log(user.settings?.theme); // Output: undefined
```

With Method Calls

```
const user = {
  greet() {
    return "Hello!";
  },
};

console.log(user.greet?.()); // Output: Hello!
console.log(user.sayBye?.()); // Output: undefined
```

---

11. Looping Objects: Object Keys, Values, and Entries:

You can loop over the properties of an object using its **keys**, **values**, or **entries**.

Object.keys()

Extracts an array of property names.

```
const person = { name: "Alice", age: 25 };
for (const key of Object.keys(person)) {
  console.log(key);
}
// Output:
// name
// age
```

Object.values()

Extracts an array of property values.

```
for (const value of Object.values(person)) {
  console.log(value);
}
// Output:
// Alice
// 25
```

Object.entries()

Extracts an array of key-value pairs.

```
for (const [key, value] of Object.entries(person)) {
  console.log(`${key}: ${value}`);
}
// Output:
// name: Alice
// age: 25
```

Challenge #1

Using the concepts of destructuring, the spread operator, optional chaining, and looping arrays, solve the following:

Problem

A company tracks employee details and their skills. Your task is to:

1. Extract specific details using **destructuring**.
2. Combine multiple skill sets into one array using the **spread operator**.
3. Safely access nested properties using **optional chaining**.
4. Iterate through the combined skill set using a **for...of loop**.

Data

```
const employees = [
  { name: "Yinka", skills: ["JavaScript", "React"] },
  { name: "Moyo", skills: ["HTML", "CSS"] },
  { name: "Victor", skills: null },
];
```

Expected Output

- Extract and log Yinka's name and skills.
- Combine all skills into one array and log it.
- Handle null skills safely using optional chaining.
- Log all combined skills.

Challenge #1: Solution

```
const employees = [
  { name: "Yinka", skills: ["JavaScript", "React"] },
  { name: "Moyo", skills: ["HTML", "CSS"] },
  { name: "Victor", skills: null },
];

// 1. Extract Yinka's details
const [yinka] = employees;
console.log(`${yinka.name}'s skills: ${yinka.skills}`);

// 2. Combine all skills using the spread operator
const allSkills = [
  ...employees[0].skills,
  ...employees[1].skills,
  ...(employees[2].skills ?? []), // Handle null skills safely
];
console.log("All Skills:", allSkills);

// 3. Iterate through all skills using for...of
for (const skill of allSkills) {
  console.log(skill);
}
```

Challenge #2

Using the concepts of enhanced object literals, optional chaining, and looping objects, solve the following:

Problem for Challenge #2

A school keeps student information in objects. Your task is to:

1. Create a student object with properties: `name`, `age`, and a method `introduce()` using **enhanced object literals**.
2. Use **optional chaining** to safely access nested data.
3. Loop through the student's keys, values, and entries.

Data for Challenge #2

```
const studentInfo = {
  name: "Sarah",
  age: 20,
  scores: {
    math: 95,
    science: 88,
  },
};
```

Expected Output for Challenge #2

- Print the student's introduction using the `introduce()` method.
- Safely access and print the math score using optional chaining.
- Loop through keys, values, and entries, and log them.

Challenge #2: Solution

```
// 1. Create student object using enhanced object literals
const student = {
  name: "Sarah",
  age: 20,
  scores: { math: 95, science: 88 },
  introduce() {
    return `Hi, I'm ${this.name} and I'm ${this.age} years old.`;
  },
};

// 2. Print the introduction
console.log(student.introduce());

// 3. Safely access math score
console.log(`Math Score: ${student.scores?.math}`);
```

```
// 4. Loop through keys, values, and entries
console.log("Keys:");
for (const key of Object.keys(student)) console.log(key);

console.log("Values:");
for (const value of Object.values(student)) console.log(value);

console.log("Entries:");
for (const [key, value] of Object.entries(student)) {
  console.log(`${key}: ${value}`);
}
```

12. Sets:

A `Set` is a collection of unique values. It can store any type of value, whether primitive or object.

Key Characteristics of Sets

1. Unique values only (no duplicates).
2. Iteration is possible with loops.
3. Sets are unordered.
4. They are not indexed (no access by position).

Basic Syntax for `Set`

```
const fruits = new Set(["Apple", "Banana", "Apple", "Orange"]);
console.log(fruits); // Output: Set(3) { 'Apple', 'Banana', 'Orange' }
```

Key Methods in Sets

- `add(value)`: Adds a value.
- `delete(value)`: Deletes a value.
- `has(value)`: Checks if a value exists.
- `size`: Gets the number of items.
- `clear()`: Removes all values.

Example Usage for `Set`

```
const mySet = new Set();

// Adding values
mySet.add("JavaScript");
mySet.add("Python");
mySet.add("JavaScript"); // Duplicate is ignored

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

```
console.log(mySet.has("Python")); // Output: true

// Deleting a value
mySet.delete("JavaScript");
console.log(mySet); // Output: Set(1) { 'Python' }

// Iterating a set
for (const language of mySet) {
  console.log(language);
}
// Output: Python
```

---

13. Maps: Fundamentals:

A `Map` is a collection of key-value pairs where keys can be of any type.

Key Characteristics of Maps

1. Keys can be objects, arrays, or primitives.
2. Maps are ordered (iteration happens in insertion order).
3. Size is determined with `size`.

Basic Syntax for `Map`

```
const map = new Map([
  ["name", "Alice"],
  ["age", 25],
]);

console.log(map); // Output: Map(2) { 'name' => 'Alice', 'age' => 25 }
```

Key Methods in Maps

- `set(key, value)`: Adds or updates a key-value pair.
- `get(key)`: Retrieves the value of a key.
- `has(key)`: Checks if a key exists.
- `delete(key)`: Removes a key-value pair.
- `clear()`: Removes all pairs.
- `size`: Gets the number of pairs.

Example Usage for `Map`

```
const user = new Map();
user.set("name", "Bob");
user.set("role", "Admin");
```

```
console.log(user.get("name")); // Output: Bob
console.log(user.has("role")); // Output: true
console.log(user.size); // Output: 2
```

---

14. Maps: Iteration:

You can iterate over Maps to extract keys, values, or entries.

Using Loops with Maps

Iterate Over Keys

```
for (const key of user.keys()) {
  console.log(key);
}
// Output:
// name
// role
```

Iterate Over Values

```
for (const value of user.values()) {
  console.log(value);
}
// Output:
// Bob
// Admin
```

Iterate Over Entries

```
for (const [key, value] of user.entries()) {
  console.log(`${key}: ${value}`);
}
// Output:
// name: Bob
// role: Admin
```

---

15. Summary: Which Data Structure to Use?

Objects vs. Maps

Feature	Objects	Maps
Key Types	Strings, Symbols	Any (Strings, Objects, Arrays)
Order	Not guaranteed	Maintains insertion order
Iteration	Requires `Object.keys()`	Directly iterable
Performance	Better for small key-value pairs	Efficient for frequent operations

Sets vs. Arrays

Feature	Sets	Arrays
Duplicates	Not allowed	Allowed
Performance	Faster for unique checks	Slower for unique checks
Use Case	Ensuring uniqueness	Maintaining order and duplicates

---

Challenge #3

Using the concepts of `Sets` and `Maps`, solve the following:

Problem for Challenge #3

You are building a music library system that stores song details. Your task is to:

1. Use a `Set` to store unique genres.
2. Use a `Map` to store song details, with the song title as the key and an object containing the artist and duration as the value.
3. Safely retrieve the details of a song using its title.
4. Iterate over the `Map` to log all songs and their details.

Data for Challenge #3

```
const genres = ["Pop", "Rock", "Jazz", "Pop", "Classical", "Jazz"];
const songs = [
  { title: "Imagine", artist: "John Lennon", duration: 183 },
  { title: "Bohemian Rhapsody", artist: "Queen", duration: 354 },
  { title: "Hotel California", artist: "Eagles", duration: 391 },
];
```

Expected Output for Challenge #3

- Log all unique genres.
- Add all songs to the `Map` and log it.
- Retrieve the details of "Imagine".
- Log each song and its details.

Challenge #3: Solution

```
// 1. Create a Set for genres
const uniqueGenres = new Set(genres);
console.log("Unique Genres:", uniqueGenres); // Output: Set(4) { 'Pop', 'Rock',
'Jazz', 'Classical' }

// 2. Create a Map for songs
const songMap = new Map();
songs.forEach((song) =>
  songMap.set(song.title, { artist: song.artist, duration: song.duration })
);

console.log("Song Map:", songMap);
// Output:
// Map(3) {
//   'Imagine' => { artist: 'John Lennon', duration: 183 },
//   'Bohemian Rhapsody' => { artist: 'Queen', duration: 354 },
//   'Hotel California' => { artist: 'Eagles', duration: 391 }
// }

// 3. Retrieve details of a specific song
console.log("Details of Imagine:", songMap.get("Imagine"));
// Output: { artist: 'John Lennon', duration: 183 }

// 4. Iterate over the Map
for (const [title, details] of songMap.entries()) {
  console.log(`${title} by ${details.artist}, Duration: ${details.duration}s`);
}
// Output:
// Imagine by John Lennon, Duration: 183s
// Bohemian Rhapsody by Queen, Duration: 354s
// Hotel California by Eagles, Duration: 391s
```

16. Working With Strings - Part 1

Strings are one of the most commonly used data types. JavaScript provides numerous methods to manipulate and analyze strings efficiently.

Basic String Syntax

```
const singleQuote = "Hello, World!";
const doubleQuote = "Hello, World!";
const templateLiteral = `Hello, ${"World"}!`;
```

Common String Methods

1. `length`: Returns the length of the string.
2. `toLowerCase()` / `toUpperCase()`: Changes the case of the string.
3. `trim()`: Removes whitespace from both ends.
4. `indexOf()` / `lastIndexOf()`: Finds the index of a substring.

Examples for Common String Methods

```
const str = "  Hello, JavaScript!  ";
console.log(str.length); // Output: 21
console.log(str.trim()); // Output: "Hello, JavaScript!"
console.log(str.indexOf("JavaScript")); // Output: 8
console.log(str.toUpperCase()); // Output: "  HELLO, JAVASCRIPT!  "
```

---

17. Working With Strings - Part 2

Continuing with string manipulation, we explore slicing and replacing substrings.

Advanced String Methods

1. `slice(start, end)`: Extracts a part of a string.
2. `substring(start, end)`: Similar to `slice`, but cannot accept negative indices.
3. `replace()` / `replaceAll()`: Replaces part(s) of a string.

Examples for Advanced String Methods

```
const greeting = "Hello, JavaScript!";

console.log(greeting.slice(7, 17)); // Output: "JavaScript"
console.log(greeting.substring(7, 17)); // Output: "JavaScript"
console.log(greeting.replace("JavaScript", "World")); // Output: "Hello,
World!"
console.log(greeting.replaceAll("o", "O")); // Output: "HellO, JavaScript!"
```

---

18. Working With Strings - Part 3

We conclude string manipulation with splitting, joining, and checking for substrings.

String Methods for Splitting and Joining

1. `split(delimiter)`: Splits the string into an array.
2. `join(delimiter)`: Joins elements of an array into a string.

Checking for Substrings

1. `includes(substring)`: Checks if a string contains a substring.
2. `startsWith()` / `endsWith()`: Checks the start or end of a string.

Examples for String Methods for Splitting, Joining and Substrings

```
const sentence = "JavaScript is awesome!";

console.log(sentence.split(" ")); // Output: [ 'JavaScript', 'is', 'awesome!' ]
console.log(["JavaScript", "is", "awesome!"].join(" ")); // Output: "JavaScript
is awesome!"

console.log(sentence.includes("awesome")); // Output: true
console.log(sentence.startsWith("Java")); // Output: true
console.log(sentence.endsWith("!")); // Output: true
```

---

19. String Methods Practice

Scenario-Based Exercises

1. Extract the domain name from an email:

```
const email = "user@example.com";
console.log(email.slice(email.indexOf("@") + 1)); // Output:
"example.com"
```

2. Capitalize the first letter of each word:

```
const title = "javascript is fun";
console.log(
  title
    .split(" ")
    .map((word) => word[0].toUpperCase() + word.slice(1))
    .join(" ")
); // Output: "JavaScript Is Fun"
```

3. Mask credit card numbers:

```
const cardNumber = "1234567812345678";
console.log(cardNumber.slice(-4).padStart(cardNumber.length, "*"));
// Output: "************5678"
```

Challenge #4

Problem for Challenge #4

You are developing a text analytics tool. Use the concepts of string methods to solve the following:

1. Given a string, count the number of words.
2. Extract and log all email addresses from a paragraph.
3. Standardize and format names in "LAST, First" format.
4. Mask sensitive data (e.g., credit card numbers).

Data for Challenge #4

```
const paragraph = `
  Welcome to JavaScript learning. Contact us at user1@example.com or
user2@test.org.
  Also, don't forget your card number: 1234567812345678.
`;

const names = ["john doe", "JANE DOE", "aDam smITh"];
```

Expected Output for Challenge #4

1. Word count: 16
2. Emails: `['user1@example.com', 'user2@test.org']`
3. Names formatted: `['DOE, John', 'DOE, Jane', 'SMITH, Adam']`
4. Masked card number: `************5678`

Challenge #4: Solution

```
// 1. Word count
const wordCount = paragraph.split(/\s+/).filter((word) => word.trim()).length;
console.log(`Word count: ${wordCount}`); // Output: Word count: 16

// 2. Extract emails
const emailRegex = /[\w.-]+@[\w.-]+\.\w+/g;
const emails = paragraph.match(emailRegex);
console.log("Emails:", emails); // Output: Emails: [ 'user1@example.com',
'user2@test.org' ]

// 3. Format names
const formattedNames = names.map((name) => {
  const [first, last] = name.toLowerCase().split(" ");
  return `${last.toUpperCase()}, ${first[0].toUpperCase()}${first.slice(1)}`;
});
console.log("Formatted Names:", formattedNames);
// Output: [ 'DOE, John', 'DOE, Jane', 'SMITH, Adam' ]
```

```
// 4. Mask sensitive data
const cardRegex = /\d{16}/;
const maskedCard = paragraph.replace(cardRegex, (match) =>
  match.slice(-4).padStart(match.length, "*")
);
console.log("Masked Paragraph:", maskedCard);
// Output: Paragraph with masked card number
```
