**JavaScript Basics**

- \*JavaScript Fundamentals\*: Strong understanding of JavaScript basics, including variables, data types, functions, and control flow.

- \*Spread Operator\*: Proficiency in using the spread operator for array and object manipulation.

- \*Rest Operator\*: Ability to use the rest operator to handle function arguments.

**Spread Operator (...)**

The spread operator allows an iterable (like an array, string, or object) to be expanded in places where zero or more elements are expected. It's commonly used for:

1. **Expanding Arrays**:
   * You can use the spread operator to expand an array into individual elements.

javascript

Copy code

const arr1 = [1, 2, 3];

const arr2 = [4, 5, 6];

const combined = [...arr1, ...arr2]; // [1, 2, 3, 4, 5, 6]

1. **Copying Arrays**:
   * It can be used to create a shallow copy of an array.

javascript

Copy code

const original = [1, 2, 3];

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

1. **Expanding Objects**:
   * You can use it to copy or merge objects.

javascript

Copy code

const obj1 = { a: 1, b: 2 };

const obj2 = { c: 3, d: 4 };

const merged = { ...obj1, ...obj2 }; // { a: 1, b: 2, c: 3, d: 4 }

1. **Function Arguments**:
   * The spread operator can also be used to pass an array of arguments to a function.

javascript

Copy code

function sum(x, y, z) {

return x + y + z;

}

const numbers = [1, 2, 3];

console.log(sum(...numbers)); // 6

**Rest Operator (...)**

The rest operator is used to collect multiple elements into an array. It is typically used in function parameters to handle an indefinite number of arguments or in destructuring assignments.

1. **Function Parameters**:
   * You can use the rest operator in a function parameter to gather all the remaining arguments into an array.

javascript

Copy code

function sum(...numbers) {

return numbers.reduce((acc, num) => acc + num, 0);

}

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

1. **Array Destructuring**:
   * When destructuring an array, the rest operator can be used to collect the remaining elements into a new array.

javascript

Copy code

const [first, ...rest] = [1, 2, 3, 4];

console.log(first); // 1

console.log(rest); // [2, 3, 4]

1. **Object Destructuring**:
   * Similarly, in object destructuring, the rest operator can collect the remaining properties into a new object.

javascript

Copy code

const { a, b, ...rest } = { a: 1, b: 2, c: 3, d: 4 };

console.log(a); // 1

console.log(b); // 2

console.log(rest); // { c: 3, d: 4 }

- \*Array/Object Destructuring\*: Experience with destructuring assignments to extract values from arrays and objects.

- \*Substring vs Slice\*: Knowledge of string manipulation methods and their differences

**substring() Method**

* **Syntax**: string.substring(startIndex, endIndex)
* **Behavior**:
  + The substring method returns a portion of the string between the startIndex and the endIndex (but not including the endIndex).
  + If endIndex is omitted, substring extracts the rest of the string starting from startIndex.
  + If startIndex is greater than endIndex, substring swaps them and returns the string between the indices.
  + If either startIndex or endIndex is negative or greater than the length of the string, they are treated as 0 or the string's length, respectively.
* **Example**:

javascript

Copy code

const str = "Hello, world!";

console.log(str.substring(0, 5)); // "Hello"

console.log(str.substring(7, 12)); // "world"

console.log(str.substring(7)); // "world!"

console.log(str.substring(7, 2)); // "ello," (swaps indices)

**slice() Method**

* **Syntax**: string.slice(startIndex, endIndex)
* **Behavior**:
  + The slice method also returns a portion of the string from startIndex to endIndex (not including the endIndex).
  + If endIndex is omitted, slice extracts the rest of the string starting from startIndex.
  + Unlike substring, slice does not swap the indices if startIndex is greater than endIndex. Instead, it returns an empty string.
  + Both startIndex and endIndex can be negative, in which case they are treated as offsets from the end of the string.
* **Example**:

javascript

Copy code

const str = "Hello, world!";

console.log(str.slice(0, 5)); // "Hello"

console.log(str.slice(7, 12)); // "world"

console.log(str.slice(7)); // "world!"

console.log(str.slice(7, 2)); // "" (no swapping, returns empty string)

console.log(str.slice(-6, -1)); // "world"

**Key Differences**

1. **Handling of Indices**:
   * substring swaps startIndex and endIndex if startIndex is greater than endIndex.
   * slice does not swap indices; if startIndex is greater than endIndex, it returns an empty string.
2. **Negative Indices**:
   * substring treats negative values as 0.
   * slice allows negative indices to count from the end of the string.
3. **Use Cases**:
   * Use substring if you want automatic index swapping or don't need to handle negative indices.
   * Use slice if you need to work with negative indices or prefer strict behavior regarding index order.

Both methods are useful, but slice is generally more versatile due to its ability to handle negative indices and its consistency with the Array.prototype.slice() method

- \*Callback Functions\*: Understanding of callback functions and their use in asynchronous programming.

A **callback function** is a function that is passed as an argument to another function and is executed after some kind of event or action has occurred within that function. Callbacks are a key part of asynchronous programming in JavaScript, where they are often used to handle tasks like reading files, making network requests, or handling user inputs without blocking the main execution thread.

**Characteristics of Callback Functions**

1. **Passed as an Argument**: The callback function is passed to another function as an argument and is invoked inside that function to complete some action.
2. **Executed Later**: The callback is executed after the outer function has completed its operation or when a specific event occurs. This allows for asynchronous operations.
3. **Can Be Anonymous or Named**: Callback functions can be defined inline (anonymous) or passed as references to named functions.

**Example 1: Synchronous Callback**

In a synchronous context, the callback is executed immediately after the outer function is called.

javascript

Copy code

function greet(name, callback) {

console.log("Hello, " + name + "!");

callback();

}

function sayGoodbye() {

console.log("Goodbye!");

}

greet("Alice", sayGoodbye);

// Output:

// Hello, Alice!

// Goodbye!

**Example 2: Asynchronous Callback**

In an asynchronous context, the callback is executed after a time-consuming operation is completed, such as reading a file or making an HTTP request.

javascript

Copy code

// Simulating a network request with setTimeout

function fetchData(callback) {

console.log("Fetching data...");

setTimeout(() => {

console.log("Data fetched!");

callback("Sample Data");

}, 2000);

}

function processData(data) {

console.log("Processing: " + data);

}

fetchData(processData);

// Output:

// Fetching data...

// (waits 2 seconds)

// Data fetched!

// Processing: Sample Data

**Callback Hell**

When multiple asynchronous operations are chained together with nested callbacks, the code can become difficult to read and maintain. This phenomenon is known as "callback hell."

javascript

Copy code

doSomething(function(result1) {

doSomethingElse(result1, function(result2) {

doMore(result2, function(result3) {

doSomethingElseAgain(result3, function(result4) {

// And so on...

});

});

});

});

**Mitigating Callback Hell**

1. **Modularize Code**: Break down the code into smaller, reusable functions.
2. **Use Promises**: Promises are a cleaner way to handle asynchronous operations.
3. **Async/Await**: Introduced in ES2017, async/await syntax makes asynchronous code look more like synchronous code, improving readability.

**Example with Promises (Avoiding Callback Hell)**

javascript

Copy code

function doSomething() {

return new Promise((resolve) => {

setTimeout(() => resolve("Result 1"), 1000);

});

}

function doSomethingElse(result) {

return new Promise((resolve) => {

setTimeout(() => resolve(result + " -> Result 2"), 1000);

});

}

doSomething()

.then(result1 => doSomethingElse(result1))

.then(result2 => console.log(result2))

.catch(error => console.error(error));

**Summary**

* **Callback Function**: A function passed as an argument to another function and executed after some event or operation.
* **Synchronous Callbacks**: Executed immediately after the outer function.
* **Asynchronous Callbacks**: Executed after a time-consuming operation completes.
* **Callback Hell**: A situation where multiple nested callbacks lead to hard-to-read code.
* **Alternatives**: Promises and async/await are modern solutions to manage asynchronous operations more cleanly.

Callbacks are foundational to JavaScript, especially for handling asynchronous operations, but modern practices often favor Promises and async/await for better code readability and maintainability.

**Example 2: Asynchronous Callback**

In an asynchronous context, the callback is executed after a time-consuming operation is completed, such as reading a file or making an HTTP request.

javascript

Copy code

// Simulating a network request with setTimeout

function fetchData(callback) {

console.log("Fetching data...");

setTimeout(() => {

console.log("Data fetched!");

callback("Sample Data");

}, 2000);

}

function processData(data) {

console.log("Processing: " + data);

}

fetchData(processData);

// Output:

// Fetching data...

// (waits 2 seconds)

// Data fetched!

// Processing: Sample Data

This example demonstrates how a callback function can be used in an asynchronous operation, where the callback is executed after a delay simulated by setTimeout.

- \*Closures\*: Ability to create and use closures effectively.

- \*String Reversal\*: Capability to reverse a string using various JavaScript techniques.

Reversing a string in JavaScript can be done in several ways. Below are a few common methods:

**1. Using split(), reverse(), and join()**

This is the most straightforward and commonly used method. It involves converting the string into an array, reversing the array, and then joining it back into a string.

javascript

Copy code

function reverseString(str) {

return str.split('').reverse().join('');

}

console.log(reverseString("Hello")); // Output: "olleH"

**2. Using a for Loop**

You can manually loop through the string from the end to the beginning, building the reversed string.

javascript

Copy code

function reverseString(str) {

let reversed = '';

for (let i = str.length - 1; i >= 0; i--) {

reversed += str[i];

}

return reversed;

}

console.log(reverseString("Hello")); // Output: "olleH"

**3. Using Recursion**

A more advanced approach is to use recursion to reverse the string.

javascript

Copy code

function reverseString(str) {

if (str === "") {

return "";

} else {

return reverseString(str.substr(1)) + str.charAt(0);

}

}

console.log(reverseString("Hello")); // Output: "olleH"

**4. Using reduce()**

This method leverages the reduce() function to accumulate the characters in reverse order.

javascript

Copy code

function reverseString(str) {

return str.split('').reduce((reversed, char) => char + reversed, '');

}

console.log(reverseString("Hello")); // Output: "olleH"

**5. Using Array.from() (ES6+)**

This method converts the string to an array using Array.from(), then applies the same reverse() and join() methods.

javascript

Copy code

function reverseString(str) {

return Array.from(str).reverse().join('');

}

console.log(reverseString("Hello")); // Output: "olleH"

- \*Minor Coding Consoles\*: Experience with coding challenges and problem-solving.

- \*map vs reduce\*: Proficiency in using array methods like map and reduce.

- \*Static vs Dynamic Typing\*: Understanding the differences between statically and dynamically typed languages.

**Dynamic Typing in JavaScript**

* **Definition**: JavaScript is a dynamically typed language, which means that variable types are determined at runtime. You do not need to declare a variable’s type when you write the code, and the type can change as the program executes.
* **Example**:

javascript

Copy code

let variable = 10; // variable is a number

console.log(typeof variable); // "number"

variable = "Hello"; // variable is now a string

console.log(typeof variable); // "string"

* **Characteristics**:
  + **Type Checking at Runtime**: Types are checked when the code is executed, not when it is compiled (because JavaScript is interpreted).
  + **Flexible Types**: Variables can hold any type of value, and the type can change dynamically.
* **Advantages**:
  + **Flexibility**: Allows for more flexible and concise code since you don’t need to specify types.
  + **Ease of Use**: Useful for quick scripting and prototyping where strict type checking is less critical.
* **Disadvantages**:
  + **Runtime Errors**: Type-related errors may only become apparent when the code is executed, which can lead to runtime errors that are harder to debug.
  + **Less Optimization**: Since types are resolved at runtime, there are fewer opportunities for compiler optimizations.

**Static Typing in JavaScript with TypeScript**

JavaScript can also use **static typing** through TypeScript, a statically typed superset of JavaScript. TypeScript introduces optional type annotations and checks types at compile time.

* **TypeScript Example**:

typescript

Copy code

let age: number = 30; // Static type declaration

let name: string = "Alice"; // Static type declaration

// TypeScript will catch type errors at compile time

age = "25"; // Error: Type 'string' is not assignable to type 'number'

* **Benefits of TypeScript**:
  + **Compile-Time Checking**: Provides early detection of type-related errors before running the code.
  + **Improved Tooling**: Better support for code editors and IDEs with features like autocompletion and refactoring.

**Node.js**

- \*Event Loop\*: In-depth knowledge of the Node.js event loop and asynchronous programming.

- \*PUT vs PATCH\*: Understanding the differences between HTTP PUT and PATCH methods.

**Key Differences**

1. **Scope of Update**:
   * **PUT**: Typically requires the full resource representation. It replaces the entire resource with the provided data.
   * **PATCH**: Allows partial updates. Only the specified fields are updated.
2. **Idempotency**:
   * **PUT**: Idempotent; repeated identical requests yield the same result.
   * **PATCH**: May or may not be idempotent; the effect of multiple identical requests may vary depending on the implementation.
3. **Use Cases**:
   * **PUT**: Use when you need to replace or fully update a resource.
   * **PATCH**: Use when you need to make partial updates or modifications to a resource.

**Summary**

* **PUT**: Replaces or updates a resource with a new representation, typically requiring the full data.
* **PATCH**: Applies partial updates to a resource, modifying only the specified fields.

- \*CORS\*: Knowledge of Cross-Origin Resource Sharing (CORS) and how to configure it.

- \*Cron Jobs\*: Experience with scheduling tasks using Cron jobs and understanding cron patterns.

- \*Error-First Callback\*: Proficiency in using error-first callback patterns in Node.js.

- \*Middlewares\*: Experience with writing and using middleware functions in Express.js.

- \*package-lock.json\*: Understanding the role and structure of package-lock.json in npm.

* **Purpose**: Ensures consistent dependency versions and faster installations.
* **Structure**: Contains a detailed and exact snapshot of all dependencies and their versions.
* **Management**: Typically included in version control and updated automatically by npm.

By keeping package-lock.json in your project, you can ensure that your project dependencies are predictable and consistent across all environments.

- \*Status Codes\*: Familiarity with common HTTP status codes (e.g., 401, 404, 502, 201) and their meanings.

- \*File Uploading & Multer\*: Experience with handling file uploads using Multer.

- \*Promise.all vs Promise.allSettled\*: Knowledge of handling multiple promises using Promise.all and Promise.allSettled.

**MongoDB**

- \*Update Query\*: Ability to write and optimize update queries in MongoDB.

- \*Projection\*: Proficiency in using projection to limit fields returned by a query.

- \*Populate\*: Experience with Mongoose's populate method for referencing documents in other collections.

- \*Joins in MongoDB\*: Understanding of performing joins in MongoDB using aggregation framework.

- \*Third Highest Salary\*: Ability to write queries to find the third highest salary record.

**Aggregation Pipeline to Find the Third Highest Salary**

1. **Sort**: First, sort the salaries in descending order.
2. **Group**: Group by a placeholder (to make sure the result is single document).
3. **Skip and Limit**: Skip the first two documents and limit the result to one.

db.employees.aggregate([ // Step 1: Sort salaries in descending order { $sort: { salary: -1 } }, // Step 2: Skip the first two highest salaries { $skip: 2 }, // Step 3: Limit to one document (the third highest salary) { $limit: 1 } ])

- \*Distinct by Two Keys\*: Capability to query distinct documents by multiple keys.

db.orders.aggregate([

// Step 1: Group by the combination of `customer` and `product`

{

$group: {

\_id: {

customer: "$customer",

product: "$product"

}

}

},

// Step 2: Project the fields to return the distinct combinations

{

$project: {

\_id: 0,

customer: "$\_id.customer",

product: "$\_id.product"

}

}

])

* **$group**: This stage groups the documents by a composite key (customer and product), creating a unique combination for each group. The \_id field of the output documents will contain these unique combinations.
* **$project**: This stage reshapes the output to include the customer and product fields, removing the default \_id field that MongoDB creates when using $group.

**Result**

The output will include distinct combinations of customer and product:

**Preferred Skills**

- Experience with front-end frameworks such as React.js or Angular.

- Familiarity with TypeScript.

- Knowledge of DevOps practices and tools.

- Experience with Docker and containerization.