Advanced JavaScript Concepts

1. Variable Scoping (`var`, `let`, `const`)

Definition: Variable scoping determines where variables are accessible in your code. JavaScript has **global scope**, **function scope**, and **block scope**.

- `var`: Function-scoped. Can be re-declared and re-assigned.
- `let`: Block-scoped. Cannot be re-declared in the same scope, but can be re-assigned.
- `const`: Block-scoped. Cannot be re-declared or re-assigned.

Example:

```
const appName = "MyWebApp"; // Global constant, value won't change

function processOrder(orderId, quantity) {
  var totalAmount = 0; // Function-scoped, might be updated
  let itemPrice = 100; // Block-scoped, might change within loops

  if (quantity > 10) {
    let discount = 0.1; // Block-scoped, only exists inside this if-block
    totalAmount = itemPrice * quantity * (1 - discount);
    console.log(`Applying ${discount*100}% discount.`);
  } else {
    totalAmount = itemPrice * quantity;
  }
  // console.log(discount); // Error: discount is not defined (outside its block)
  console.log(`Order ${orderId} total: ${totalAmount}`);
}

processOrder("A123", 15);
processOrder("B456", 8);
```

Use Cases:

- **`const`:**
  - Declaring **fixed values** like API keys (though sensitive keys shouldn't be hardcoded client-side), configuration settings, or mathematical constants (`PI`).
  - Referencing DOM elements that won't change (`const myButton = document.getElementById('btn');`).
  - Ensuring a variable's reference remains constant (e.g., an array or object that you will modify internally but won't reassign to a completely new array/object).
- **`let`:**
  - **Loop counters** (`for (let i = 0; i < 10; i++)`).

    - Variables that **change their value** within a specific block of code (e.g., a user's score, a counter, a temporary result in a calculation).
    - Variables used in `if/else` blocks where their scope should be limited to that block.
- **`var`:** (Generally **avoid in modern JS**)
    - Rarely used in new code. Its function-scoping behavior can lead to unexpected bugs (hoisting, variable leakage) which `let` and `const` solve. You might encounter it in legacy codebases.

---

2. Closure

Definition: A closure is when a function "remembers" and can access variables from its surrounding (lexical) scope, even after that outer scope has finished executing.

Example:

```
function createCounter() {
  let count = 0; // 'count' is in the lexical environment of increment/decrement

  return {
    increment: function() {
      count++;
      console.log(`Count: ${count}`);
    },
    decrement: function() {
      count--;
      console.log(`Count: ${count}`);
    },
    getCount: function() {
      return count;
    }
  };
}

const myCounter = createCounter();
myCounter.increment(); // Output: Count: 1
myCounter.increment(); // Output: Count: 2
myCounter.decrement(); // Output: Count: 1
console.log(myCounter.getCount()); // Output: 1

const anotherCounter = createCounter(); // Creates a completely separate counter
anotherCounter.increment(); // Output: Count: 1
```

Use Cases:

- **Data Encapsulation / Private Variables:** As shown in the `createCounter` example, closures allow you to create "private" variables (`count`) that are not directly accessible from outside the function, but can be manipulated by specific methods returned by the outer function. This is a common pattern for managing state.

3. Template Literals (Template Strings)

Definition: A way to create strings using backticks (`` ` ``) that allows for easy embedding of expressions (interpolation with `${}`) and multi-line strings.

Example:

```
const productName = "Laptop";
const price = 1200;
const taxRate = 0.08;
const finalPrice = price * (1 + taxRate);

// Creating a product summary string
const productSummary = `
  Product: ${productName}
  Price: $${price.toFixed(2)}
  Tax: ${taxRate * 100}%
  Total: $${finalPrice.toFixed(2)}
`;
console.log(productSummary);

const userStatus = "logged in";
console.log(`The user is currently ${userStatus}.`);
```

Use Cases:

- **Dynamic UI Content:** Generating HTML snippets or dynamic messages to display on a webpage.
- **Log Messages:** Creating detailed log messages with variable data.
- **SQL Queries / API Endpoints (with caution):** Dynamically building complex strings like SQL queries (though be very careful about SQL injection here, prefer parameterized queries) or API endpoint URLs.
- **Email Templates / Notifications:** Constructing personalized messages for users.

4. Destructuring

Definition: A convenient way to extract values from arrays or properties from objects into distinct variables.

Example:

```
// User object received from an API
const apiResponse = {
  id: 101,
  username: "johndoe",
  email: "john.doe@example.com",
  profile: {
    firstName: "John",
    lastName: "Doe",
    age: 30,
    address: {
      street: "123 Dev St",
      city: "CoderVille"
    }
  },
  roles: ["admin", "editor"],
  isActive: true
};

// 1. Extracting specific properties from a user object
const { username, email, isActive } = apiResponse;
console.log(`User: ${username}, Email: ${email}, Active: ${isActive}`);

// 2. Renaming properties during extraction
const { username: userAlias, email: userEmail } = apiResponse;
console.log(`Alias: ${userAlias}, Email: ${userEmail}`);

// 3. Extracting nested properties
const { profile: { firstName, lastName, address: { city } } } = apiResponse;
console.log(`Name: ${firstName} ${lastName}, City: ${city}`);

// 4. Setting default values if a property doesn't exist
const { timezone = "UTC" } = apiResponse; // apiResponse does not have
'timezone'
console.log(`Timezone: ${timezone}`); // Output: Timezone: UTC

// 5. Destructuring arrays (e.g., from a function returning multiple values)
function getCoordinates() {
  return [10.25, 20.75];
}
const [latitude, longitude] = getCoordinates();
console.log(`Lat: ${latitude}, Lon: ${longitude}`);

// 6. Swapping variables (a classic trick)
let a = 1;
let b = 2;
[a, b] = [b, a];
console.log(`a: ${a}, b: ${b}`); // Output: a: 2, b: 1
```

Use Cases:

- **Processing API Responses:** Easily picking out the data you need from complex JSON objects.
- **Function Parameters:** Making functions more readable by directly extracting properties from an options object passed as an argument.
- **Working with Arrays:** Extracting specific elements from arrays, or easily swapping values.

5. Default Parameters

Definition: Allows you to assign default values to function parameters. If a parameter is not provided or is `undefined` when the function is called, its default value will be used.

Example:

```
function sendMessage(message, sender = "Anonymous", timestamp = new Date().toLocaleString()) {
  console.log(`[${timestamp}] ${sender}: ${message}`);
}

// User provides all arguments
sendMessage("Hello everyone!", "Alice", "2024-01-01 10:00:00");
// Output: [2024-01-01 10:00:00] Alice: Hello everyone!

// User omits sender, uses default
sendMessage("What's up?");
// Output: [current_time] Anonymous: What's up?

// User omits sender and timestamp, uses both defaults
sendMessage("I need help.");
// Output: [current_time] Anonymous: I need help.

// User explicitly passes undefined for a parameter
sendMessage("Checking in", undefined, "2024-06-21 15:30:00");
// Output: [2024-06-21 15:30:00] Anonymous: Checking in
```

Use Cases:

- **Providing sensible fallbacks:** For optional function arguments.

- **Configuration Objects:** When a function accepts a configuration object, default parameters can set up common defaults.
- **API Client Methods:** Setting default values for pagination, timeout, or other optional request parameters.

---

6. Rest Parameter (. . .)

Definition: Allows a function to accept an indefinite number of arguments as an array. It gathers all remaining arguments into a single array. Must be the last parameter.

Example:

```
function displayScores(studentName, ...scores) { // 'scores' will be an array of numbers
  console.log(`Scores for ${studentName}:`);
  if (scores.length === 0) {
    console.log("No scores available.");
    return;
  }
  let total = 0;
  for (const score of scores) {
    console.log(` - ${score}`);
    total += score;
  }
  console.log(`Average score: ${(total / scores.length).toFixed(2)}`);
}

displayScores("Bob", 85, 92, 78, 95);
// Output:
// Scores for Bob:
//  - 85
//  - 92
//  - 78
//  - 95
// Average score: 87.50

displayScores("Charlie");
// Output:
// Scores for Charlie:
// No scores available.

function createPlaylist(playlistName, ...songs) {
  console.log(`Playlist: ${playlistName}`);
  console.log(`Songs: ${songs.length}`);
  songs.forEach(song => console.log(` - ${song}`));
}
```

```

createPlaylist("Morning Vibes", "Song A", "Song B", "Song C");
```

Use Cases:

- **Functions with Variable Arguments:** When you don't know in advance how many arguments a function will receive (e.g., a logging function that can take multiple messages).

7. Spread Operator (...)

Definition: Allows an iterable (like an array or string) to be expanded into individual arguments (for function calls) or elements (for array literals). It also allows an object to be expanded into key-value pairs (for object literals).

```
// 1. Combining Arrays
const vegetables = ["carrot", "spinach"];
const fruits = ["apple", "banana"];
const groceries = [...vegetables, ...fruits, "milk", "bread"];
console.log(groceries); // Output: ["carrot", "spinach", "apple", "banana", "milk", "bread"]

// 2. Copying Arrays (shallow copy)
const originalList = [1, 2, 3];
const copiedList = [...originalList];
console.log(copiedList); // Output: [1, 2, 3]
console.log(originalList === copiedList); // Output: false (they are different arrays in memory)

// 3. Merging Objects
const userDetails = { name: "David", age: 40 };
const userSettings = { theme: "dark", notifications: true };
const fullUserConfig = { ...userDetails, ...userSettings, id: "D40" };
console.log(fullUserConfig);
// Output: { name: "David", age: 40, theme: "dark", notifications: true, id: "D40" }

// 4. Passing Array Elements as Function Arguments
const temperatures = [22, 28, 19];
console.log(Math.max(...temperatures)); // Output: 28 (like Math.max(22, 28, 19))

// 5. Adding new properties or overriding existing ones in an object
const product = { name: "Old Laptop", price: 800 };
const updatedProduct = { ...product, name: "New Laptop", category: "Electronics" };
console.log(updatedProduct);
// Output: { name: "New Laptop", price: 800, category: "Electronics" }
```

Use Cases:

- **Immutability in State Management:** Creating new arrays or objects based on existing ones without modifying the originals, which is crucial in frameworks like React/Redux for predictable state updates.
- **Merging Data:** Combining multiple arrays or objects.
- **Function Arguments:** When a function expects separate arguments but you have them in an array.

8. Arrow Functions (=>)

Definition: A more concise way to write function expressions. They have a different `this` binding behavior (they don't have their own `this`; they inherit `this` from the surrounding lexical context) and cannot be used as constructors.

```
// Array method callback (common use)
const numbers = [1, 2, 3];
const doubled = numbers.map(num => num * 2); // Concise, implicit return
console.log(doubled); // Output: [2, 4, 6]

// Event handler example
const button = document.createElement('button');
button.textContent = 'Click Me';
document.body.appendChild(button);

// Traditional function: 'this' inside the handler refers to the button
button.addEventListener('click', function() {
  console.log(`Traditional: ${this.textContent} was clicked.`);
});

// Arrow function: 'this' inside the handler refers to the 'this' of its surrounding scope
// (e.g., the global window object if outside another function, or the object if used as a method)
// In a simple global script, 'this' would be window.
button.addEventListener('click', () => {
  // If this was inside a class method, 'this' would correctly refer to the class instance.
  // For event listeners, it generally means you don't need to bind 'this'.
  console.log(`Arrow: A button was clicked.`);
});

// Object method using arrow function (caution: 'this' will be global/undefined)
const game = {
    score: 0,
    increaseScore: () => {
        // 'this' here refers to the global object (window) or undefined in strict mode, NOT 'game'
    // console.log(this.score); // This will not work as expected
  },
  // Correct way for object methods: regular function expression or method shorthand
  increaseScoreCorrect: function() {
    this.score += 10;
    console.log(`Score: ${this.score}`);
  }
};
game.increaseScoreCorrect(); // Output: Score: 10
```

Use Cases:

- **Concise Callbacks:** Ideal for array methods (`map`, `filter`, `reduce`, `forEach`) and asynchronous operations (Promises, `setTimeout`).
- **Short, Single-Expression Functions:** When a function body is just a single `return` statement.

9. Enhanced Object Literals

Definition: Provide shorthand syntax for defining object properties and methods, and for using computed property names.

```
const user = "Frank";
const isAuthenticated = true;
const getId = () => 123;

const appConfig = {
  // 1. Shorthand property names (when variable name is same as property name)
  user, // equivalent to user: user
  isAuthenticated, // equivalent to isAuthenticated: isAuthenticated

  // 2. Shorthand method definition
  displayStatus() { // equivalent to displayStatus: function() { ... }
    console.log(`${this.user} is ${this.isAuthenticated ? 'logged in' : 'logged out'}.`);
  },

  // 3. Computed property names (property key is an expression)
  [`user-${getId()}-status`]: "active", // key is 'user-123-status'

  // Can mix and match
  version: "1.0.0"
};

appConfig.displayStatus(); // Output: Frank is logged in.
console.log(appConfig["user-123-status"]); // Output: active
```

Use Cases:

- **Cleaner Object Definitions:** Reducing boilerplate when creating objects, especially configuration objects or data structures.
- **Dynamic Object Keys:** When object property names are not fixed but derived from variables or expressions (e.g., based on user input, loop variables).

10. Iterators & `for...of` Loop

Definition:

- **Iterable:** An object that can be iterated over (e.g., Arrays, Strings, Maps, Sets). It has a `Symbol.iterator` method that returns an iterator.
- **Iterator:** An object with a `next()` method that returns `{ value: ..., done: ... }`.
- **`for...of` loop:** A loop designed to iterate over iterable objects, directly accessing the values of the elements.

Example:

```
// Iterating over a Set (which is an iterable)
const uniqueTags = new Set(["coding", "javascript", "webdev", "coding"]);
console.log("Unique Tags:");
for (const tag of uniqueTags) {
  console.log(`- ${tag}`);
}
// Output:
// - coding
// - javascript
// - webdev

// Iterating over characters in a string
const myWord = "hello";
console.log("Characters:");
for (const char of myWord) {
  console.log(`- ${char}`);
}

// Iterating over Map entries (key-value pairs)
const userRoles = new Map([
    ["admin", "Administrator"],
    ["editor", "Content Editor"],
    ["viewer", "Guest Viewer"]
]);
console.log("User Roles:");
```

```
for (const [role, description] of userRoles) { // Destructuring in for...of
    console.log(`${role}: ${description}`);
}
```

Use Cases:

- **Simplifying Iteration:** The most common and direct way to loop through values of arrays, strings, Sets, Maps, and other custom iterable objects.
- **Working with New Data Structures:** Essential for cleanly accessing data within Maps and Sets.
- **Custom Iteration Logic:** If you create your own custom iterable objects, `for...of` will automatically work with them.

12. Modules (`import` / `export`)

Definition: A system that allows you to break your JavaScript code into separate files (modules), making it more organized, reusable, and maintainable.

- `export`: Makes variables, functions, classes, etc., available for other modules to use.
- `import`: Allows you to use exports from other modules in your current file.

```
// mathOperations.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

export default function multiply(a, b) {
  return a * b;
}
```

```
// app.js
import { add, subtract } from './mathOperations.js';
import multiplyValues from './mathOperations.js'; // Can rename default import

console.log(add(10, 5));        // Output: 15
console.log(subtract(10, 5));   // Output: 5
console.log(multiplyValues(10, 5)); // Output: 50
```

Use Cases:

- **Organizing Large Codebases:** Breaking down a complex application into smaller, manageable files (e.g., `user.js`, `product.js`, `auth.js`).

- **Code Reusability:** Exporting utility functions, constants, or components that can be used across multiple parts of your application or even in different projects.

13. Map

Definition: A built-in JavaScript object that holds key-value pairs, similar to objects, but with a key difference: `Map` allows keys of any data type (including objects, functions, or numbers), and it maintains the order of insertion.

```
const userSettingsMap = new Map();

const user1 = { id: 1, name: "Grace" };
const user2 = { id: 2, name: "Henry" };

// Storing complex objects as keys
userSettingsMap.set(user1, { theme: "dark", notifications: true });
userSettingsMap.set(user2, { theme: "light", notifications: false });
userSettingsMap.set("admin", { permissions: "all" }); // Can still use strings

console.log(userSettingsMap.get(user1)); // Output: { theme: "dark", notifications: true }
console.log(userSettingsMap.get("admin")); // Output: { permissions: "all" }

console.log(userSettingsMap.has(user1)); // Output: true
userSettingsMap.delete(user2);
console.log(userSettingsMap.size); // Output: 2

// Iterating over Map keys and values
for (const userObj of userSettingsMap.keys()) {
   console.log(`User key:`, userObj);
}
for (const settings of userSettingsMap.values()) {
   console.log(`Settings value:`, settings);
}
```

14. Array Methods

Definition: Built-in functions available on array objects that provide powerful ways to manipulate, iterate, and transform arrays without manually writing loops. These often return new arrays instead of modifying the original (pure functions), promoting immutability.

```
const products = [
  { id: 1, name: "Laptop", category: "Electronics", price: 1200 },
  { id: 2, name: "Keyboard", category: "Electronics", price: 75 },
  { id: 3, name: "Mouse", category: "Electronics", price: 25 },
  { id: 4, name: "Desk Chair", category: "Furniture", price: 300 },
  { id: 5, name: "Monitor", category: "Electronics", price: 250 }
];

// 1. map(): Transform each item in an array to a new item
// Use Case: Displaying a list of product names on a webpage
const productNames = products.map(product => product.name);
console.log("Product Names:", productNames); // Output: ["Laptop", "Keyboard", "Mouse", "Desk Chair", "Monitor"]

// 2. filter(): Select items that meet a certain condition
// Use Case: Finding all electronics products
const electronicsProducts = products.filter(product => product.category === "Electronics");
console.log("Electronics:", electronicsProducts);

// 3. reduce(): Aggregate all items into a single value
// Use Case: Calculating the total price of all products
const totalPrice = products.reduce((sum, product) => sum + product.price, 0);
console.log("Total Price:", totalPrice); // Output: 1850

// 4. find(): Get the first item that matches a condition
// Use Case: Finding a specific product by ID
const specificProduct = products.find(product => product.id === 3);
console.log("Found Product:", specificProduct); // Output: { id: 3, name: "Mouse", ... }

// 5. some() / every(): Check if *any* or *all* items meet a condition
// Use Case: Check if there's any product over $1000
const hasExpensiveProduct = products.some(product => product.price > 1000);
console.log("Has expensive product:", hasExpensiveProduct); // Output: true

// Use Case: Check if all products are under $2000
const allProductsAffordable = products.every(product => product.price < 2000);
console.log("All products affordable:", allProductsAffordable); // Output: true

// 6. forEach(): Iterate over items to perform an action (no new array created)
// Use Case: Logging each product to the console
console.log("--- All Products ---");
products.forEach(product => console.log(`${product.name} - $${product.price}`));
```

Use Cases:

- **Data Transformation:** Converting data from one format to another (e.g., list of objects to list of names).
- **Data Filtering:** Displaying only relevant data based on user input or specific criteria.

- **Aggregations:** Calculating sums, averages, counts from a list of numbers or objects.
- **Search and Validation:** Quickly finding specific elements or validating conditions across an array.
- **Building UI Components:** Dynamically rendering lists in frameworks like React.

15. Higher-Order Function

Definition: A function that either:

1. **Takes one or more functions as arguments (a callback).**
2. **Returns a function as its result.**

```
// Example 1: Takes a function as an argument (like Array.prototype.map)
function customMap(array, transformFunction) {
  const result = [];
  for (let i = 0; i < array.length; i++) {
    result.push(transformFunction(array[i], i, array)); // Pass value, index, array
  }
  return result;
}

const numbers = [1, 2, 3, 4];
const squared = customMap(numbers, num => num * num);
console.log("Custom Squared:", squared); // Output: [1, 4, 9, 16]

// Example 2: Returns a function
function createValidation(minLength) {
  return function(input) { // This inner function is returned (a closure)
    return input.length >= minLength;
  };
}

const isValidUsername = createValidation(5); // Returns a function that checks for >= 5 chars
const isValidPassword = createValidation(8); // Returns a function that checks for >= 8 chars

console.log("Is 'John' a valid username?", isValidUsername("John"));     // Output: false
console.log("Is 'Jonathan' a valid username?", isValidUsername("Jonathan")); // Output: true
console.log("Is 'abc' a valid password?", isValidPassword("abc"));       // Output: false
console.log("Is 'mypass123' a valid password?", isValidPassword("mypass123")); // Output: true
```

Use Cases:

- **Function Composition:** Building complex functions by combining simpler ones.

16. Callback

Definition: A function passed as an argument to another function, which is then invoked inside the outer function to complete some kind of routine or action. Callbacks are fundamental for asynchronous operations but can also be used in synchronous contexts.

```
function loadUserData(userId, callback) {
  console.log(`Attempting to load data for user ID: ${userId}`);
  setTimeout(() => {
    // Simulate network delay and data fetching
    const userData = { id: userId, name: `User_${userId}`, email:
`user${userId}@example.com` };
    console.log(`Data for user ${userId} loaded.`);
    callback(userData); // Call the provided function with the data
  }, 2000); // Simulate 2-second delay
}

function displayUserData(data) {
  console.log(`Displaying User: ${data.name} (${data.email})`);
}

function sendWelcomeEmail(data) {
  console.log(`Sending welcome email to ${data.email}...`);
  // Simulate sending email
}

loadUserData(101, displayUserData); // displayUserData is the callback
loadUserData(102, sendWelcomeEmail); // sendWelcomeEmail is another callback

loadUserData(103, (user) => { // Inline anonymous callback
  console.log(`Received user ${user.id} and processing it directly.`);
});
```

Use Cases:

- **Asynchronous Operations:** The primary use for `setTimeout`, `setInterval`, network requests (`XMLHttpRequest` in older JS, or Promise-based `fetch` under the hood), and reading files.
- **Event Handling:** Responding to user interactions (clicks, key presses) or system events.

- **Customizable Behavior:** Allowing users of your function to define what happens after a certain operation completes (e.g., `array.forEach(item => doSomething(item))`).

17. Promises

Definition: An object representing the eventual completion or failure of an asynchronous operation. They provide a more structured and readable way to handle asynchronous code compared to nested callbacks (Callback Hell).

```
function fetchProductDetails(productId) {
  console.log(`Fetching details for product ${productId}...`);
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (productId === 123) {
        resolve({ id: 123, name: "Smartwatch", price: 299.99 });
      } else if (productId === 404) {
        reject(`Product with ID ${productId} not found.`);
      } else {
        reject("Unknown product ID.");
      }
    }, 1500);
  });
}

// Successful scenario
fetchProductDetails(123)
  .then(product => {
    console.log("Product found:", product);
    // You can chain more .then() for sequential async operations
    return fetch(`https://api.example.com/reviews/${product.id}`); // Imagine another async call
  })
  .then(response => response.json())
  .then(reviews => console.log("Reviews:", reviews))
  .catch(error => { // Catch any error in the chain
    console.error("Error during product fetch or review fetch:", error);
  })
  .finally(() => {
    console.log("Product details fetch process concluded.");
  });

// Error scenario
fetchProductDetails(404)
  .then(product => {
    console.log("Product found (unexpected):", product);
  })
  .catch(error => {
    console.error("Expected Error:", error); // Output: Expected Error: Product with ID 404 not found.
  });

// Handling multiple promises concurrently
const promise1 = fetchProductDetails(123);
const promise2 = new Promise(res => setTimeout(() => res("Another task done!"), 1000));

Promise.all([promise1, promise2]) // Waits for ALL promises to resolve
   .then(results => {
      console.log("All tasks completed:", results);
   })
   .catch(error => {
      console.error("One of the tasks failed:", error);
   });
```

Use Cases:

- **HTTP Requests:** The most common use case (e.g., `fetch` API returns Promises).
- **Reading Files:** Asynchronous file I/O operations.
- **Database Operations:** Interacting with databases.

18. Exponentiation Operator (`**`)

Definition: A mathematical operator that performs exponentiation (raises the first operand to the power of the second operand). It's a cleaner alternative to `Math.pow()`

```
// Calculating powers
const base = 2;
const power = 4;
console.log(`${base} to the power of ${power} is: ${base ** power}`); // Output: 16 (2 * 2 * 2 * 2)

// Calculating square roots (power of 0.5)
const num = 25;
console.log(`Square root of ${num} is: ${num ** 0.5}`); // Output: 5

// Compound assignment
let value = 3;
value **= 2; // Equivalent to value = value * value; or value = value ** 2;
console.log("Value after squaring:", value); // Output: 9
```

Use Cases:

- Mathematical Calculations: Any scenario involving powers (e.g., financial calculations, physics simulations, geometric calculations).
- Cleaner Code: Replacing `Math.pow()` for more readable expressions.

20. Ternary Operator

Definition: A shorthand way to write a simple `if-else` statement. It's the only JavaScript operator that takes three operands: a condition, a value if the condition is true, and a value if the condition is false.

Syntax: **`condition ? value_if_true : value_if_false;`**

```
const userRole = "admin";
const canEdit = (userRole === "admin" || userRole === "editor") ? true : false;
console.log(`User can edit: ${canEdit}`); // Output: User can edit: true

const isLoggedIn = false;
const welcomeMessage = isLoggedIn ? "Welcome back!" : "Please log in to continue.";
console.log(welcomeMessage); // Output: Please log in to continue.

const score = 75;
const grade = score >= 90 ? "A" :
              score >= 80 ? "B" :
              score >= 70 ? "C" :
              score >= 60 ? "D" : "F";
console.log(`Grade: ${grade}`); // Output: Grade: C (can be chained for multiple conditions)
```

Use Cases:

- **Conditional Assignments:** Assigning a value to a variable based on a condition.
- **Inline Conditional Rendering (e.g., React JSX):** Displaying different UI elements based on a condition directly within template code.
- **Simple Logic:** When you need a quick, single-line conditional expression for a value.
- **Function Returns:** Returning different values from a function based on a condition.

21. Optional Chaining (`?.`)

Definition: A safe way to access properties or call methods of an object that might be `null` or `undefined` without causing a `TypeError`. If a property in the chain is `null` or `undefined`, the expression short-circuits and returns `undefined`.

```
const user1 = {
  name: "Ivy",
  preferences: {
    theme: "dark",
    notifications: {
      email: true,
      sms: false
    }
  },
  isAdmin: () => true
};

const user2 = {
  name: "Jack",
  preferences: null // preferences is null
};

const user3 = {
  name: "Kelly" // no preferences property
};

// 1. Accessing nested properties safely
console.log(user1.preferences?.notifications?.email); // Output: true
console.log(user2.preferences?.notifications?.email); // Output: undefined (no error)
console.log(user3.preferences?.notifications?.email); // Output: undefined (no error)

// 2. Accessing array elements safely
const users = [{ id: 1 }, null, { id: 3 }];
console.log(users?.[0]?.id); // Output: 1
console.log(users?.[1]?.id); // Output: undefined
console.log(users?.[5]?.id); // Output: undefined (index 5 doesn't exist)

// 3. Calling methods safely
console.log(user1.isAdmin?.());    // Output: true (method exists and is called)
console.log(user2.isAdmin?.());    // Output: undefined (method doesn't exist, no error)

// 4. Using with nullish coalescing for fallback values
const userConfig = { settings: null };
const defaultTheme = userConfig.settings?.theme ?? 'light'; // If settings?.theme is undefined/null, use 'light'
console.log(`Default Theme: ${defaultTheme}`); // Output: Default Theme: light

const userConfig2 = { settings: { theme: "blue" } };
const userTheme = userConfig2.settings?.theme ?? 'light';
console.log(`User Theme: ${userTheme}`); // Output: User Theme: blue
```

Use Cases:

- **Handling API Responses:** When JSON data might have optional or missing nested fields.

- **Accessing Configuration Objects:** When parts of a configuration might be missing.
- **Improved Readability:** Making code cleaner by avoiding multiple nested `&&` checks (`user && user.address && user.address.city`).